TU WIEN Informatics

Fehlertolerante GALS-Architektur auf Basis von Pausable Clocking

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Wolfgang Dür
Matrikelnummer 1526990

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 26. Jänner 2022

Wolfgang Dür	Andreas Steininger

TU WIEN Informatics

Fault-Tolerant GALS Architecture based on Pausable Clocking

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Wolfgang Dür
Registration Number 1526990

to the Faculty of Informatics

at the TU Wien

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

Vienna, 26th January, 2022

Wolfgang Dür	Andreas Steininger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Wolfgang Dür

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.

Wien, 26. Jänner 2022

Wolfgang Dür

Danksagung

Ich möchte meinem Betreuer Professor Andreas Steininger für die tolle Betreuung und Unterstützung bei sämtlichen Fragen bedanken. Weiters haben auch Florian Huemer, Jürgen Maier sowie Robert Najvirt mich tatkräftig unterstützt. Sei es durch bereitstellen von Libraries oder konstruktiven Diskussionen. Zu guter Letzt möchte ich mich bei meiner Familie und all jenen, welche mich nach ihren Möglichkeiten unterstützt haben und hier nicht namentlich genannt wurden und dazu beigetragen haben, diese Arbeit möglich zu machen, bedanken.

Acknowledgements

I want to thank my advisor Professor Andreas Steininger for his great advisory and always having an open ear when I had questions or problems. Furthermore Florian Huemer, Jürgen Maier and Robert Najvirt also helped a lot through providing libraries but also through inspiring discussions. Last but not least I want to thank my family and all the people not mentioned specifically that supported me in their best possible way to make this possible.

Kurzfassung

Zwei wesentliche Herausforderungen bei der Entwicklung moderner Systems-on-Chips(SoCs) sind die Taktverteilung und die Zuverlässigkeit eines Systems.
Bei der Taktverteilung geht es nicht nur darum, alle Module mit einem Takt zu versorgen, sondern auch mit dem für ihre Funktionalität notwendigen. Dies führt zu sogenannten GALS Systemen, welche auf lokaler Ebene synchron arbeiten, wegen den unterschiedlichen Taktdomänen aber asynchron kommunizieren. Für die asynchrone Kommunikation stehen wertsichere Methoden zur Verfügung, bei denen der Takt vorübergehend angehalten wird. Dies hat den Vorteil, dass die Latenz im Vergleich zu zeitsicheren Methoden, welche mit Synchronizern arbeiten, wesentlich verkürzt wird.
Das zweite Problem ist die Zuverlässigkeit eines Systems, denn diese garantiert die Kontinuität des Dienstes auch im Fehlerfalle einer Komponente und ist für viele sicherheitskritischen Anwendungen wie Flugzeugen, Kernkraftwerken oder selbstfahrenden Autos erforderlich. In solchen Anwendungen darf es keinen einzelnen Ausfallspunkt geben der dazu führt, dass das gesamte System ausfällt. Eine Klasse von Systemen, die einen einzigen Fehler tolerieren können, sind die dreifachen modularen Redundanzsysteme (TMR). Hier führen drei Replikate die gleichen Aufgaben aus und nach erfolgreichem Abschluss wird über die Ergebnisse abgestimmt. Damit ein Fehler andere Replikas nicht beeinträchtigt läuft jegliche Kommunikation, welche über eine Replika hinaus geht, über gewisse Abstimmungsmechanismen. Eine fehlerhafte Replika kann so überstimmt werden.

Das Ziel dieser Arbeit ist es nun einen Bus auf Basis pausierbarer Takte zu entwerfen. Bei dieser neuen Architektur handelt es sich um einen sogenannten Mehrkanalbus, der asynchronen Kommunikationsmustern folgt und auch selbst vollständig asynchron ist. Der Ansatz mehrere Kanäle anstatt nur einen einfachen Bus zu haben, ermöglicht mehrere gleichzeitige Transaktionen anstelle von nur einer, ohne dass dabei derselbe Flächenaufwand wie für Punkt-zu-Punkt-Verbindungen erforderlich ist.
In einem zweiten Schritt wird dieser neue Mehrkanalbus so erweitert, dass er keinen einzelnen Ausfallspunkt hat und Teil eines Systems mit gemischter Kritikalität sein kann, wo sowohl TMR-Nachrichten, als auch gewöhnliche Nachrichten, denselben Bus zur Kommunikation verwenden.
Es konnte gezeigt werden, dass der neu entwickelte Mehrkanalbus hinsichtlich der Latenz zwar schlechter abschneidet als bekannte und einfachere Konzepte, die Stärke des neu konzipierten Mehrkanalbusses jedoch neben des möglichen Durchsatz im Verhältnis zur benötigten Fläche auch in der Fehlertoleranz liegt.

Abstract

Two major challenges in the development of modern Systems-on-Chips(SoCs) are the global clock distribution and the reliability of the system.
Clock distribution is not only about providing all modules with a clock, but also with the one necessary for their functionality. This leads to so called GALS systems, which operate synchronously at the local level but due to the different clock domains they use asynchronous methods to communicate with each other. For asynchronous communication, value safe methods are available in which the clock is temporarily stopped. This has the advantage that the latency is significantly reduced compared to time safe methods, which work with synchronizers.
The second problem is the reliability of a system, because this guarantees the continuity of service even in the event of a component failure and is required for many safety-critical applications such as aircraft, nuclear power plants, or self-driving cars. In such applications, there must not be a single point of failure that results in the entire system failing. One class of systems that can tolerate a single component failure are the triple modular redundant (TMR) systems. Here, three replicas perform the same tasks and, upon successful completion, the results are voted on. To ensure that an error does not affect other replicas, all communication that goes beyond one replica runs through certain voting mechanisms. A faulty replica can thus be outvoted.

The goal of this work is now to design a bus based on pausable clocks. This new architecture is a so called multi channel bus, which follows asynchronous communication patterns and is also fully asynchronous itself. The approach of having multiple channels instead of just a single bus allows for multiple simultaneous transactions instead of just one, without requiring the same area overhead as point-to-point connections.
In a second step this new multi channel bus is extended in a way such that it does not have a single point of failure and can be part of a mixed criticality system where both, TMR messages and best effort messages use the same bus for communication.

It was shown that while the newly designed multi channel bus performs worse than known, simpler concepts in terms of latency, the strength of the newly designed multichannel bus is the possible throughput relative to the required area, as well as its fault tolerance.

Contents

CHAPTER 1

Introduction

On modern systems-on-chip (SoCs) global clock distribution is challenging. The chips have become so complex that it is extremely difficult to distribute a single global clock all over the architecture with an acceptable skew for the system to run with a clock rate that is in the GHz area. Furthermore these systems often require more than one clock domain because some of the modules are fulfilling specific tasks where special clock rates are necessary. A consequence of these requirements is that there are a lot of modules each running synchronously with its frequency but the communication between the modules has to follow asynchronous principles. Those systems are called globally asynchronous locally synchronous (GALS) systems. A first systematic approach to the GALS system design was described by Chaprio [Cha84] in 1984. Although in nowadays designs his assumptions are almost infeasible still all current work is founded on his work.
The challenge here is to find suitable communication ways between the modules to avoid metastability issues when sampling data from the sender's clock domain with the receiver clock. There one of the most straight forward methods is to insert synchronizers between the different clock domains [Gin03]. Since the synchronizer consists of two or more cascaded flip flops, latency is introduced and nevertheless there is still a non-zero risk for metastability. Therefore other approaches are interesting. An interesting but power and area consuming approach is to use asynchronous FIFOs [Gre95, SF01].
A quite different approach for secure communication is to use so called pausible clocks [Cha84, YD96, BC97, SM00, MVF00, MTMR02, KPWK03, MCS04, DGS04]. These clocks can be stopped at the receiver side during communication such that all data is transferred correctly.

Besides the need for different clock rates, many applications like airplanes, spacecrafts, nuclear power plants or self driving cars require the integrated circuits to be dependable. One attribute of a dependable system is reliability. Reliability guarantees the continuity of service even in case a fault occurred. One simple technique, for allowing and masking away one fault at a time, is to use a triple modular redundant (TMR) system [LV62]. In

this approach three replicas of the same system fulfill the same task and then when all are finished, the result is voted on. When now a fault occurs in one of the replicas, the two other ones are still working properly and they outvote the faulty replica.

1.1 Aims and Contribution

The aim of this thesis is to present a novel bus architecture, that addresses both problems stated above. The new architecture shall not only be able to handle simple data transactions between clock domains but also voted messages when being used in a TMR system. To achieve this in a first step Villiger's MOGLI bus [Vil05], that is based on pausable clocking, is extended by introducing channels. Over each channel an output port can send data to those input ports it selected by address. With introducing channels the latency can be reduced as multiple transactions can happen simultaneously but without introducing too much area overhead as it might be the case with point-to-point connections. There is a control unit that regulates the channel access. When one output port wants to send something, it sends a request and when there is a free channel the requesting port is assigned to the free channel and can send its data. The number of channels can be adapted depending on the usage scenario in order to find an optimal trade-off between efficient channel usage and minimal blocking through waiting times in relation to the average latency of a data transmission.
The following problems need to be solved for the multi channel bus:

- **Reliable channel arbitration**
  Here the challenge is to find a method ensuring that every port that requests the bus finally gets a channel assigned. Furthermore there should be no idling channel when there is an open bus request in order to avoid increasing the latency unnecessarily long.

- **Minimum channel blocking time**
  Villiger split up the acknowledge signal into two different acknowledge signals to distinguish whether the data was taken or not. Therefore the bus is occupied unnecessarily when the receiving port is not ready yet. It has to be analyzed how time consuming this is and if there are methods that avoid unnecessary bus occupation and do not consume too much area.

- **Possibility of multicast**
  The big issue that needs to be handled correctly when allowing multicasts is what is done when not all ports were able to receive the data yet. Here some mechanisms need to be implemented. Furthermore it might happen that multiple ports want to send data to the same input port concurrently. This should not be a problem when the module has own input ports for each channel. However it must be possible that multiple ports of a module can receive data concurrently.

In a second step the above multi channel bus is used to interconnect a TMR GALS system. Each locally synchronous module is triplicated and therefore also the communication channels including the ports must be tripled. To accept a received message it needs to be received from at least two different sender replicas. Therefore the multicast only needs to wait for two of three receivers to accept the data.

For the TMR capable multi channel bus the following problems need to be solved:

- **Synchronization**
  Unlike in a TMR system, where all replicas are using the same clock source, here each has its own local clock (This is actually a key feature of the proposed architecture, as it removes the single point of failure usually formed by the clock in a globally synchronous TMR architecture). Thus the phase shift between the replicas is not constant and predefined. As a consequence when receiving two correct messages, the replicas still have to wait for an adequate time as the last replica may just be late.

- **Mixed criticality**
  The bus system should be able to cope with best effort messages as well as with TMR messages. However those two types need to be distinguishable and it is important, that TMR messages are prioritized as they are more relevant to the overall system.

- **No single point of failure**
  Last but not least it is crucial for TMR messages that there is no point in the path, where a fault can lead to a failure of the overall system. Each fault must be confined to the replica within which it first occurred. Thus for communication between replicas, always voters are necessary and thus only TMR messages.

The scientific contribution of this thesis lies in an extension of the existing concepts and frameworks that enables attaining the envisioned goal and, to that end, overcomes the problems listed above. To the best of the author's knowledge, no respective solution has been published so far.

1.2 Methodological Approach

The expected results were achieved in the following steps. These occurred in iterations, as after a design is simulated, different optimization possibilities might be considered and compared against each other in order to analyze their advantages and disadvantages.

Literature study: In a first step the existing approaches were studied. This helps to understand the problem better and also points out the benefits and drawbacks of the different approaches that currently exist.

Implementation of selected approach: In a second step the most promising approach was then implemented in hardware for two reasons. Reason one was to have a starting point for the new design and reason two was to have a reference for comparison at a later point.

Design new approach: Here first the channels were introduced and then in a second step the goal was to adapt the multi channel solution for TMR use.

Simulation, Analysis and Comparison: The new approach was simulated exhaustively after being designed. As a baseline Villigers MOGLI [Vil05] was used. In addition to fault-resilience, throughput and latency, an interesting analysis is the behavior of the area in the different approaches.

Optimization: A final step then was to optimize the new approach after analyzing the simulation and understanding its behavior as well as advantages and disadvantages.

1.3 Tools

The chosen methodology requires to use multiple tools that are described in more detail in the following.

1.3.1 Workcraft

As the whole multi channel bus ought to be asynchronous, special design techniques have to be used. A great tool for designing asynchronous finite state machines is Workcraft [PSM07]. Workcraft is designed to develop interpreted graph models. For this work state transition graphs were designed, that then were verified and synthesized. For this thesis the Complex gate (Petrify) synthesis has been chosen among the various methods that Workcraft supports.

1.3.2 VHDL

Very High Speed Integrated Circuit Hardware Description Language (VHDL) [VHD19] is a hardware description language which allows to describe digital system in text based manner. VHDL is a description language and not a programming language. Thus not everything that can be described is synthesizable.

1.3.3 ModelSim

For simulating the system, that was programmed with VHDL, ModelSim [Cora] was used. ModelSim allows the user to simulate various hardware description languages such as VHDL, Verilog and system C. It supports not only prelayout but also postlayout simulations.

1.4 Outline

This thesis is structured as follows:

Chapter 2 gives a short overview of the related work. It includes various different GALS communication concepts as well as simple clock domain crossings.

In chapter 3 the already mentioned new bus architecture and its different components are presented in detail. It is then analyzed how a transaction looks like. This includes not only the different phases of a transaction but also the whole communication mechanism. The latter is shown in a flow chart for better illustration.

With this new bus then a mixed criticality system is presented in chapter 4. The focus of this chapter is mainly the TMR communication, as the best effort transactions were already discussed in the previous chapter. As the different ports had to be adapted to be able to handle TMR messages, the modified ports are also discussed in detail.

An analysis of all possible faults that have an influence on the bus or the TMR communication is then presented in chapter 5. The analysis distinguishes between transient bit flips and permanent faults. Beside the signals that cross the replica borders, also other interesting signals have been analyzed.

Different properties like the throughput, latency and area of the multi channel bus are then analyzed in chapter 6 and also compared to the MOGLI bus by Villiger [Vil05] that is used as a baseline. It is furthermore analyzed how much a faulty replica can influence the duration of a TMR transaction. Furthermore lower bounds for the timeout of the voters are presented, to keep the additional transaction duration as minimal as possible.

Chapter 7, which concludes this work, contains a conclusion of the work and a short overview on future work.

CHAPTER 2

Related Work

In the literature two different types of GALS systems are distinguished. On so called networks on chip (NoCs) [JT+03, BDM02, BDM01, DT01, KJS+02], a network with routers connects the locally synchronous islands with each other, whereas on a conventional GALS system the islands that communicate are connected directly which each other. In both cases the most challenging part in GALS systems is the handling of the clock domain crossing. Therefore asynchronous concepts have to be considered for communication, which always includes some kind of handshaking mechanism.

2.1 Analysis of different GALS concepts

There are mainly two big different concepts currently used for data transfers between locally synchronous islands. On the one side these are approaches with synchronizers like bundled data and asynchronous FIFOs. These have a non zero probability for metastability and are therefore not error free. On the other side there is an alternative approach which allows error free operation by halting the clock if necessary for an error free transmission.

2.1.1 Synchronizers

As already mentioned above with synchronizers there always exists a non zero probability for metastability. This probability can be reduced through adding more stages to the synchronizer. However, each synchronization stage increases the latency. In some settings a single stage synchronizer is still sufficient and is therefore preferred due to its low latency. Therefore the following two cases have been analyzed with a one stage synchronizer.

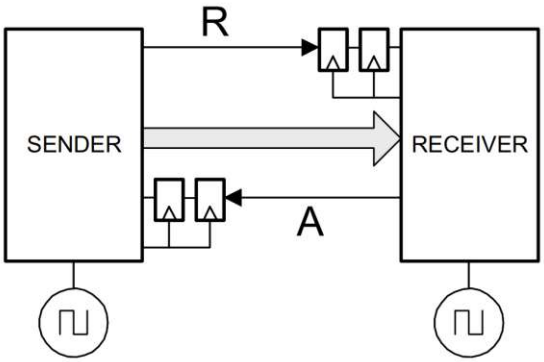


Figure 2.1: Bundled data based communication with a one stage synchronizer (from [Gin03])

Bundled data

For this concept (shown in Figure 2.1), the bundled data communication protocol is used between the two locally synchronous modules. In order to guarantee that the data is valid when it is read at the receiver, the acknowledge and the request signals have to be synchronized. For synchronization flip flop cascades are used. The more flip flops are cascaded the higher is the mean time between failure, that is defined in Equation 2.1. The mean time between failure depends on how often a metastable state occurs (fraction) and how long it takes to resolve(exponential part). For the number of metastability occurrences the data sending rate λ_{dat}, the sampling frequency f_{clk} and the technology depending parameter T_0 are responsible. The resolution on the other side depends on the available resolution time t_r and the technology depending τ_c.

$$MTBF = \frac{1}{\lambda_{dat} f_{clk} T_0} \exp^{\frac{t_r}{\tau_c}} \tag{2.1}$$

Since a one stage synchronizer consists of two cascaded flip flops the average data throughput is $\frac{1}{1.5(T_A+T_B)}$ where T_A and T_B are the clock periods of the two communicating systems.

Asynchronous FIFO (General Approach)

An asynchronous FIFO is a buffer that is accessed by two different clock domains. The schematics of such a FIFO is shown in Figure 2.2. The heart of the asynchronous FIFO is a dual port buffer. This allows reading and writing operations simultaneously. In order to only read valid data and not overwrite data which has not been read yet, the current write respective read address have to be transmitted to the other clock domain, where

an *empty*, respectively *full*-flag is generated. This can be done via Gray encoding of the address. Since in Gray code only one bit changes when a binary number is incremented by one, no acknowledge or request signal is needed. Each bit can be synchronized separately without generating intermediate values. For calculating the empty respective full flag, the remote and local address are compared with each other. Although the remote address may already be outdated when calculating the flags, the system is still safe. For the empty flag this holds because it may state empty although the sender might have written something new to the FIFO. The same argumentation can be used for the full. There again it might state full although the reader has already read an element from the FIFO. The advantage of this concept is, that a higher throughput compared to the bundled data synchronizer approach can be reached. Let's assume that the writer is faster than the reader. Then in the beginning the reader waits an average of $1.5T_R$ where T_R is the clock period of the reader's clock until it starts reading. If now a lot of data is transmitted, the FIFO will be full at some point and the reader can read one entry each cycle. If the reader is faster than the writer the throughput depends on the writer , since the writer can without delay write one entry every cycle.

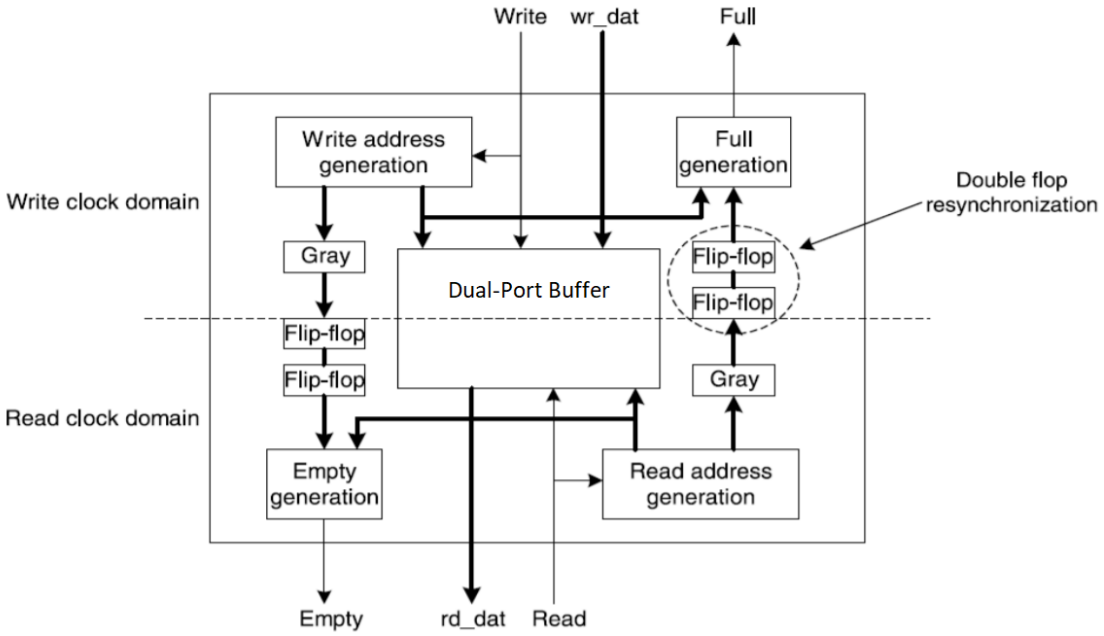


Figure 2.2: Structure of an asynchronous FIFO (from [Kil07])

A recently made internal analysis of Najvirt and Steininger [NS20] has shown that in general synchronization FIFOs are overdesigned most of the time. They were able to show that for a clock ratio larger than two a FIFO depth of two is sufficient whereas relatively equal clocks also do not require a greater depth than four if a synchronizer delay of one cycle is assumed.

Asynchronous FIFO (Pipeline approach by Huemer et al.)

Instead of exchanging the Gray encoded read and write pointers in this approach the idea is to just transmit pointer increment information across the timing domain boundaries in form of read tokens. The transmitter exchanges the increment information over a Muller pipeline [Sut89] with the same depth as the FIFO. For calculating the full and empty signal Huemer et al. introduced in [HS20] a so called desynchronizer and resynchronizer, also shown in Figure 2.3. They both sample the current state of the Muller pipeline. As there might be active transitions during sampling, metastable states might be sampled. Therefore synchronizers are introduced and depending on the flip-flop stages the number of states that are sampled needs to be adapted. This is necessary because when there are n flip-flop stages it takes up to n cycles until the pattern is analyzed. The Desynchronizer checks if the pattern is constant. As long as it is constant meaning that there is no alternating state pattern read, there is space left in the FIFO as with every write a token is generated through a toggle flip-flop. The Resynchronizer on the other side checks for an alternating pattern and also if there are still single tokens in the pipeline that are not detected by the alternating pattern detector.

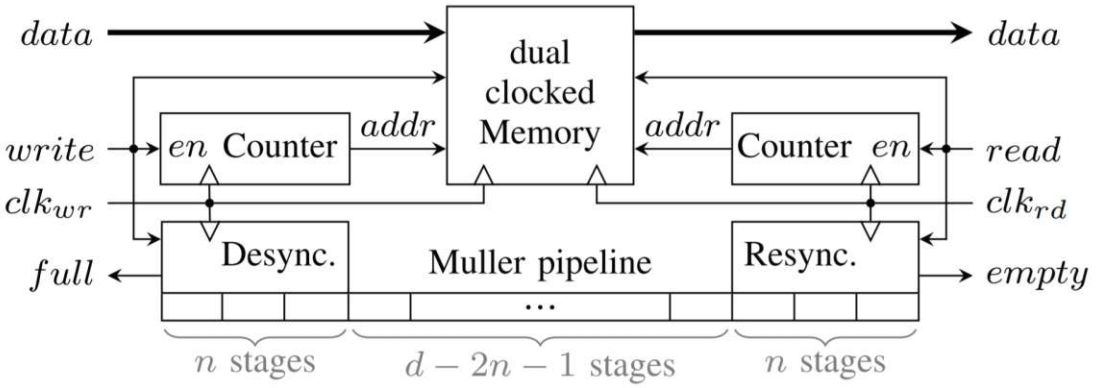


Figure 2.3: Structure of an bisynchronous FIFO with with Muller pipeline (from [HS20])

2.1.2 Pausable Clocking

The idea here is to pause or stretch the local module's clock to guarantee that the communication signals never violate the setup and hold time constraints. A pausable clock like the one shown in Figure 2.4 is generated via a timing loop (which is a ring oscillator) and a switching loop. The timing loop is responsible for the pulse width. The pulse width is generated through delaying the clock signal, which is done in D. This delay is mostly achieved by sending the clock signal through an inverter chain. Clock signals generated this way are not as precise as the ones generated by a crystal quartz. The switching loop is to stop the clock at a pause request. Therefore every high pulse and every pause request has to request for a grant signal at the mutual exclusive element M. Only one of them gets it and therefore the clock is paused when the pause requests

gets its grant. Despite M can become metastable, when both requests arrive at the same time. However when a low-threshold metastabilitiy filter [SMC80] is added at its output, the operation is value safe. When now the switching and the timing loop are ready for a transition, the Muller gate $C1$ forwards this transition to its output and a new clock edge is generated.

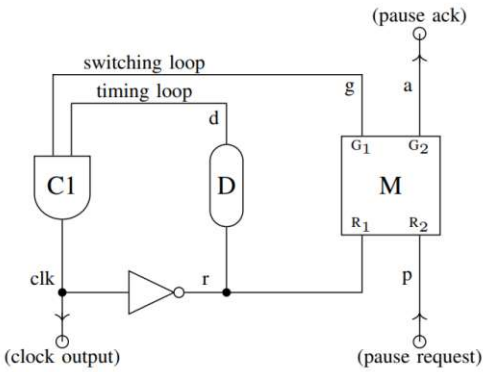


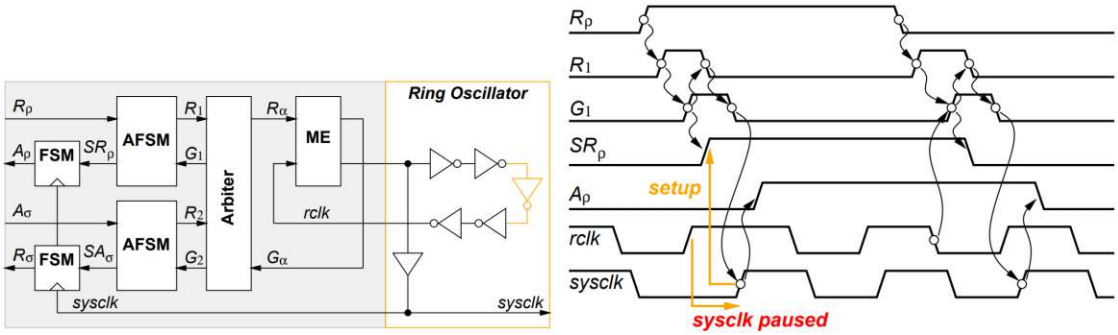
Figure 2.4: pausable clock (from [NS15])

For asynchronous communication different methods exists which are described in the following:

Variations

Pausable clocking control PCC The pausable clocking control shown in Figure 2.5a was developed in 1996 by Yun et al. [YD96] to transfer data between locally synchronous modules. It comprises the generation of the stretchable clock as well as the processing of the handshake. Two local clock cycles are at least required to transfer data and at most one port per module can be active at a time. Since the high fan-ins and fan-outs make the arbiter very large and impractical. its use is only limited to systems with circular data flow. The data is always written to and read from a FIFO which is located between two synchronous modules. When a request occurs the asynchronous finite state machine (AFSM) activates R_1 and when it gets its grant G_1, SR_ρ it is activated in a way such that no setup and hold time violations occur when the finite state machine (FSM) synchronously samples it. The AFSM performs effectively a two-phase to four-phase transformation. An illustration of a a read can also be seen in Figure 2.5b.

Asynchronous wrapper Bormann et al. In 1997 Bormann et al. [BC97] came up with asynchronous wrappers. In contrast to the previously described PCC concept the asynchronous wrapper is more flexible regarding data flow organization but on the other hand it lacks proper arbitration between concurrent requests to the clock generator. A data transfer is only possible every second clock cycle. This is because of the port enabling logic. For a new data transfer to initiate the input/output signal has to transmit



(a) PCC for bidirectional communication (b) PCC timing for one way communication

Figure 2.5: PCC (both pictures from [YD96])

a rising edge. Since the input/output signal is generated synchronously by the locally synchronous module an extra cycle is needed to switch the input/output signal to low.

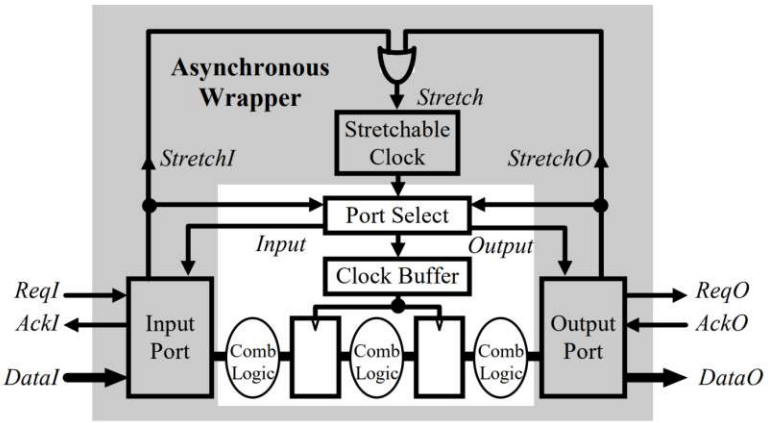
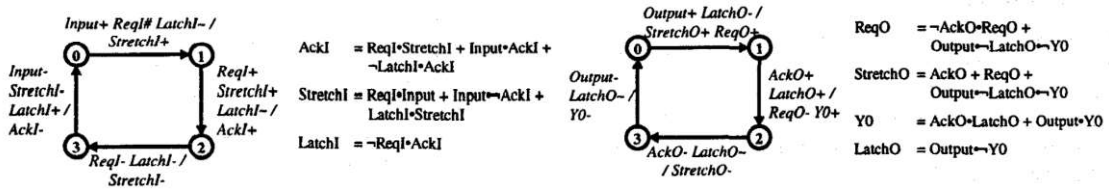


Figure 2.6: Asynchronous wrapper by Bormann (from [BC97])

Depending on the application ports may be active or passive. This means that they initiate a request or just wait for a request to occur and then answer. Therefore for both input and output port an active and a passive port specification was presented. In Figure 2.7 the asynchronous state machine for a passive input and an active output are shown. There a # indicates a directed don't care meaning that the signal may either remain at 0 or monotonically change from 0 to 1 or remain at 1. On the other hand $\sim$ means that the signal remains at 1 or monotonically changes from 1 to 0 or remains at 0. In this specification furthermore a late data-valid bundling convention is used which means that the data is not guaranteed to be valid until after $ReqI-$. The data then is latched and after $AckI-$ occurs the data at the input may change at any time.

One big problem with the design of the ports is that no matter if they're active or passive, they immediately stretch the clock when being enabled. So there was also a so called



(both pictures from [BC97])

(a) Passive input port (b) Active output port

Figure 2.7: Extended-burst-mode circuit

Q-Port presented that allows active polling for requests. Furthermore this system is prone to deadlocks. When for example two modules are communicating in both directions and they both enable the output enable signal but not the input enable signal, then they are both waiting for the input to become ready but since the clock is already paused on both modules a deadlock occurs.

Asynchronous wrapper Muttersbach et al. Muttersbach et al. [MVF00] analyzed the problems of the previously presented methods for communicating with pausable clocks. They then came up with a new asynchronous wrapper model shown in Figure 2.8 and also with a new set of ports.

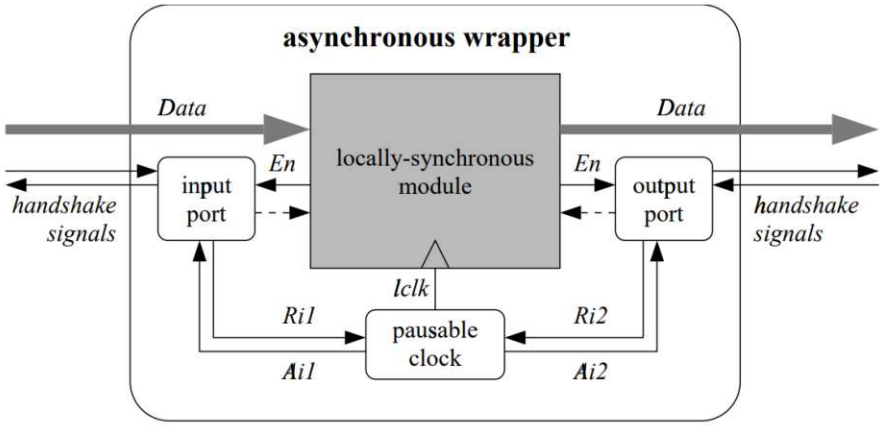
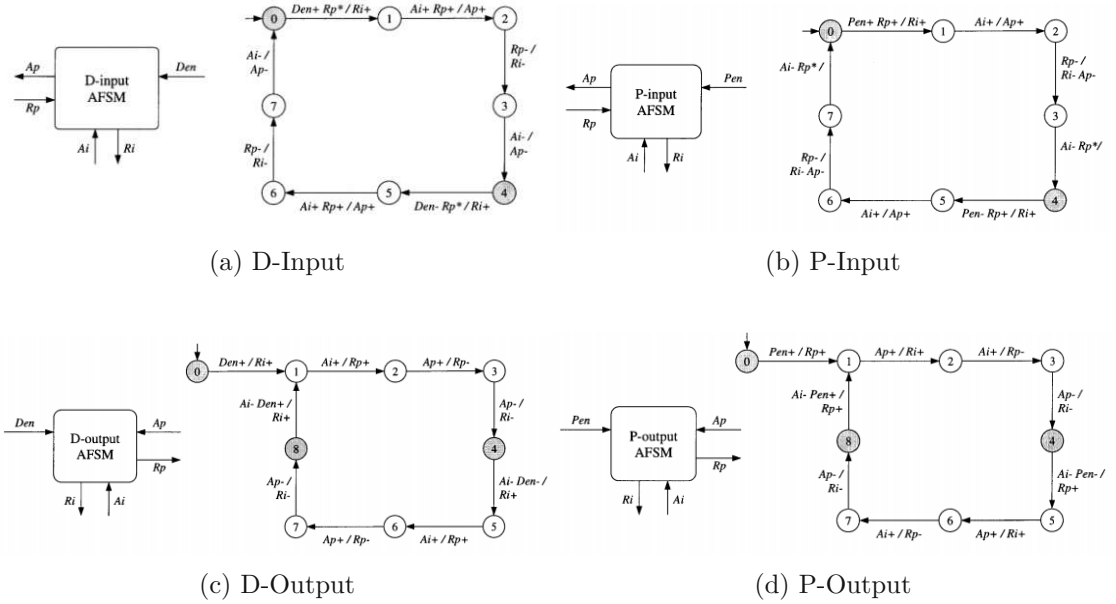


Figure 2.8: Asynchronous wrapper around a locally synchronous module (from [MVF00])

There are two types of port controllers used in Muttersbach's asynchronous wrapper. Those are the demand-ports (D-ports) and poll-ports (P-ports). They all get activated by the locally-synchronous module and enter an asynchronous finite state machine. As can be seen in Figure 2.9 the D-ports immediately send a request to the clock to be stopped when being activated whereas the P-ports wait for the request to stop the clock until the handshaking process is in a stage where stopping the clock is necessary in order to prevent metastability. Furthermore the problem with at least one necessary recovery

cycle to disable the enable signal and enable it again, like Bormann had, was solved by enabling a new transition through transition signaling on the enable line.



(a) D-Input (b) P-Input

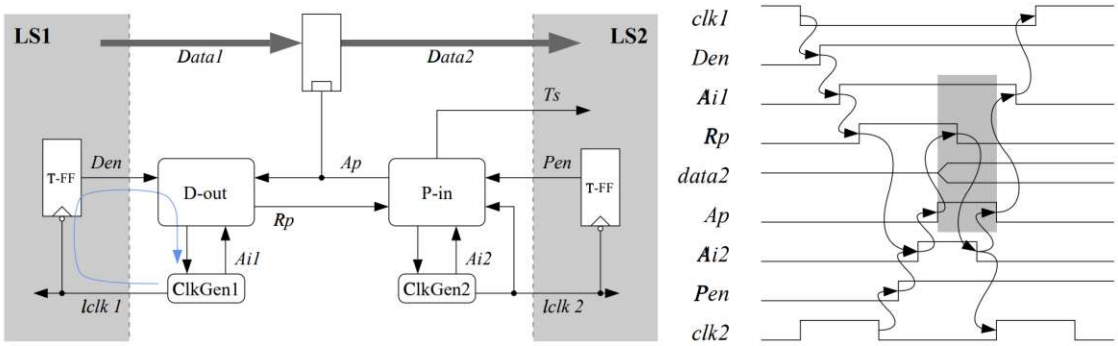
(c) D-Output (d) P-Output

Figure 2.9: All Port types presented by Muttersbach (pictures taken from [Mut01])

Figure 2.10 shows the data exchange circuit between two locally synchronous modules and also the corresponding waveforms. There the output port is a demand port. So $clk1$ is immediately halted after *Den* is set. The input port on the other side is a poll port. It is only halted shortly for sampling. Although the clock is released by the P-port towards the end of the transfer cycle, it is not known exactly when the next clock cycle might appear when the module has multiple ports. The other ports might keep the clock stretched for an unknown amount of time as they also might be in an active transmission. Therefore it is necessary to latch the transmitted data vector between both modules. The latching guarantees data correctness and decouples both modules from each other. The *Ap* signal is used for latching because then it is sure that the input port already got enabled and therefore is ready and that $clk1$ is paused during the transparency phase.

There still may occur deadlocks in this system when using demand ports due to the same reason as in the Bormann approach.

Pausable Bisynchronous FIFO In 2.1.1 already an asynchronous FIFO has been presented. There the read and write pointers are exchanged via synchronizers to reduce the probability for metastability. Keller et al. presented in [KFK15] an asynchronous FIFO where the read and write pointers are updated by a two phase increment protocol with pausable clocks instead of synchronizers. They were able to keep the average latency



(a) Data exchange channel between two locally synchronous modules

(b) Corresponding waveform (Latch is transparent in shaded area)

Figure 2.10: Data transfer mechanism (both pictures taken from [MVF00])

at a very low level of of around 1.34 clock cycles and additionally have a zero probability for metastable upsets.

In Figure 2.11 the sequence necessary for synchronization of data through the FIFO is indicated by letters. It furthermore is assumed that in the beginning the FIFO is empty and all two-phase increment and acknowledge lines are free to use. Data is then written on the rising edge of TX Clock to the FIFO address pointed to by the writer logic and the valid signal is asserted (A). The write pointer logic now increments the write pointer and toggles one of the pointer increment lines (B). On the receiver side the XOR gets active as the latched value is different to the asserted value and requests a grant to safely read the value from the TX Clock domain. For each increment line a separate mutex is needed. The RX Clock only runs if all mutex give their grant to the clock. When the increment line finally gets its grant the latch gets transparent. After the data is read by the latch, the xor is deasserted by the data from the feedback loop and so is the request (C). The new value is also transmitted to the read pointer logic in order to keep track of the write pointer and set the valid signal at the output system (D).

So far only the TX domain does not know that the RX domain has already processed the increment message. Therefore the RX domain also toggles the corresponding acknowledge line (E) and this information then has to be synchronized on the TX side in the same manner as on the RX side (F). Finally the write pointer logic receives the acknowledge signal (G) and the increment line is free for future use.

The more increment and acknowledge lines are used the more pointer increments can be transmitted within a single clock period. Experiments, though, have shown that three increment acknowledge pairs in either direction already guarantee full throughput for a sender receiver clock ratio between 1/2 and 2.

The FIFO is also adaptable to just have one module with a fixed reference such as a

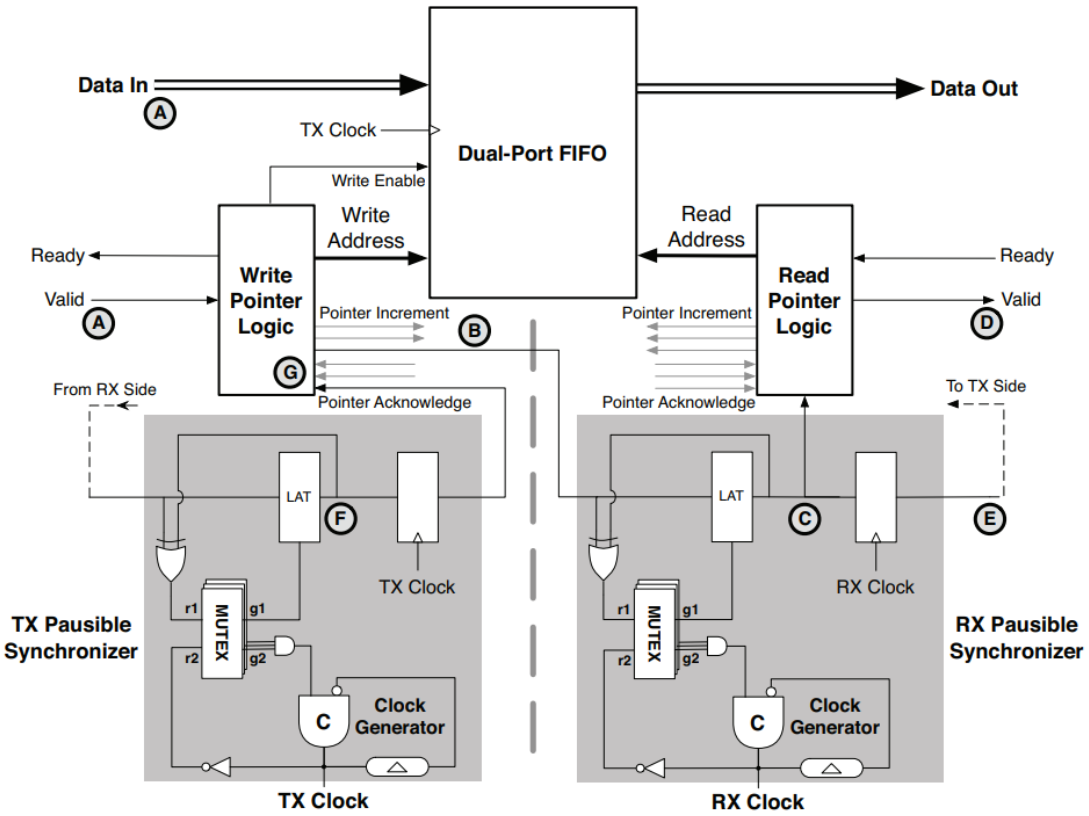


Figure 2.11: Pausable asynchronous FIFO with signal path for pointer increment highlighted (from [KFK15])

PLL and some synchronizers and the other module works with a pausable clock and the shown mechanism. This would still allow low latency communication in one direction.

Compared to the previously shown pausable clock approaches which were prone to deadlocks in multiport applications here no such deadlocks can occur as long as all modules work properly.

Optimizations

Here some minor optimizations regarding the asynchronous wrapper of Muttersbach will be presented. Muttersbach's wrapper is so far the best model presented as it allows multiple ports to communicate at the same time and also with minimal time effort.

Optimized pausable clock generator In [FKG09] Fan et al. presented a way to widen the request acknowledge window (RAW). The RAW is the duration in each clock cycle where port requests can be acknowledged immediately. For a clock generator like the one shown in Figure 2.4 it holds that $t_{RAW} \approx T_{CLK}/2$. This is because the clock

signal requests for the grant directly after the inverter and if it gets the grant immediately an eventually arriving pause request has to wait until the inverted clock gets low again, which is in the worst case after $T_{CLK}/2$ and since the low phase of the inverted clock is $T_{CLK}/2$ the RAW is also $T_{CLK}/2$.

A way to widen the RAW is to let the clock not instantly request the grant. Instead the delay line is split up into two parts as shown in Figure 2.13, which shows a simplified version, that is comparable with Figure 2.4. The two parts are namely the programmable delay line $D0$ followed by the fixed delay line $D1$ which are defined as listed below:

$$d_{D0} = T_{LClk}/2 - (d_{D1} + d_{C-ELE} + d_{NOT}) \tag{2.2}$$

$$d_{D1} = d_{AND} + (d^0_{MUTEX} + \Delta d_{MUTEX}) \tag{2.3}$$

As Equation 2.2 and 2.3 show, the request of the clock is asserted at the last possible time, such that the pulse width is not influenced. A possible additional delay due to the time it takes the MUTEX element to resolve metastability is considered in Δd_{MUTEX}. As a result the time window where a pause request gets an immediate grant is widened to $t_{RAW} = T_{LCkl} - d_{D1}$.

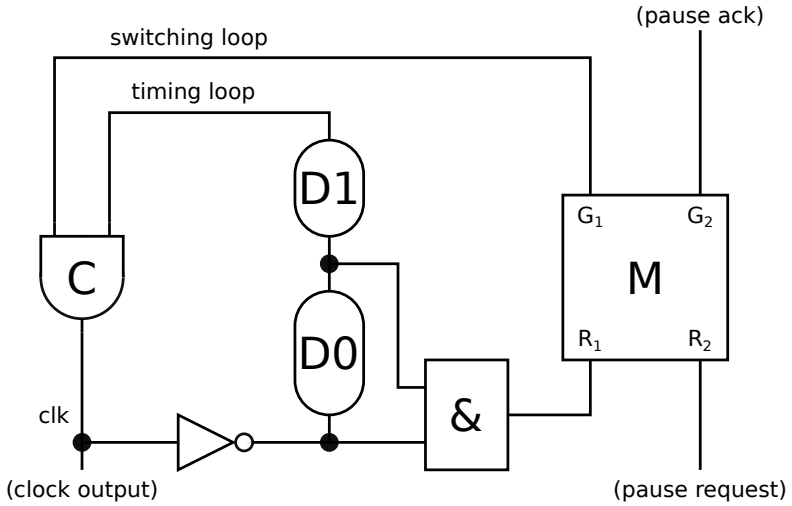


Figure 2.12: Simple optimized clock generator

The originally presented solution of Fan et al. [FKG09] is shown in Figure 2.13 and outlines a solution for multiple ports and also a possibility to reset the clock. The introduced delay of the programmable delay line $D0$ and the fixed delay line $D1$ are defined as listed below:

$$d_{D0} = T_{LClk}/2 - (d_{D1} + d_{C-ELE} + d_{NOR}) \tag{2.4}$$
$$d_{D1} = d_{AND0} + d_{AND1} + (d^{0}_{MUTEX} + \Delta d_{MUTEX}) \tag{2.5}$$

The programmable delay line and the $LClkB$ are merged together with an AND gate. Since in the $MUTEX$ element metastability is resolved there needs to be added some additional delay Δd_{MUTEX} to the delay line. The on-phase period of the resulting $RClk$ that requests the grant is then only the gate delays of the path from $MUTEX$, $AND0$, $C-ELE$, NOR and $AND1$. Thus the probability of introducing a one cycle latency in the receiver can be reduced dramatically since the request of the clock at the mutex arrives as late as possible. The resulting t_{RAW} of the optimized version is therefore $t_{RAW} = T_{LCkl} - d_{D1}$.

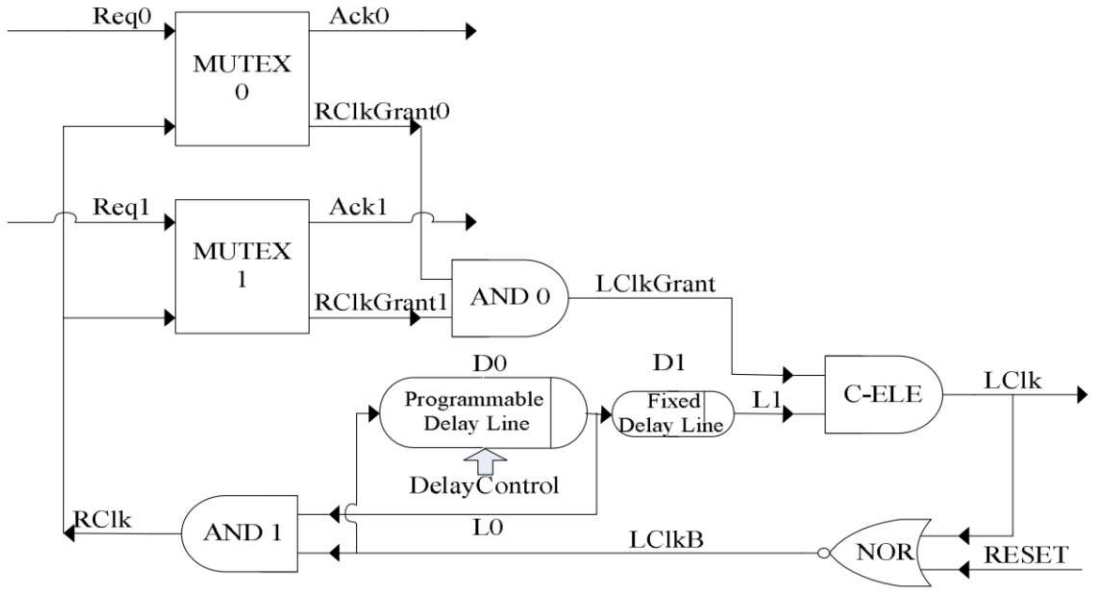


Figure 2.13: Optimized clock generator with two ports (from [FKG09])

Optimized Input Port Double latching The data transfer mechanism in 2.10 is prone to synchronization failures caused by the local clock tree insertion delay on the receiver side. As the clock tree insertion delay and the handshaking signals propagation delay are independent from each other, a rising edge of the local clock can arrive at the sampling FF at any time during the time where the latch of the input port is transparent and thus might lead to metastability in the FF. In [FKG09] the double latch mechanism shown in Figure 2.14 was presented. The idea here is to reduce the width of the critical

window to a minimum. Since the grant signals of the $MUTEX$ enable the latches $L1$ and $L2$, only one latch is transparent at a time. The simultaneous occurrence of Req_{Rx} and $RCLK$ may result in a random resolution time and therefore any rising edge of $LClk_{RxDly}$ may fall in the on-phase of $RClk$ and thus could lead to a conflict, as $L2$ is transparent at that time. If the already optimized clock generator, as shown above, is used, the duration of the safe window W_S is $T_{LCLK} - d_{D1}$ when two latches are used. In contrast, no matter how large the clock period is, it never exceeds $T_{LCLKRx}/2$ when only one latch is used. For clock periods way larger than d_{D1} the double latching method makes nearly the entire clock period safe. It can be furthermore shown that for any clock period, the width of the safe timing region is always approximately doubled with the double latching mechanism compared to the single latch mechanism.

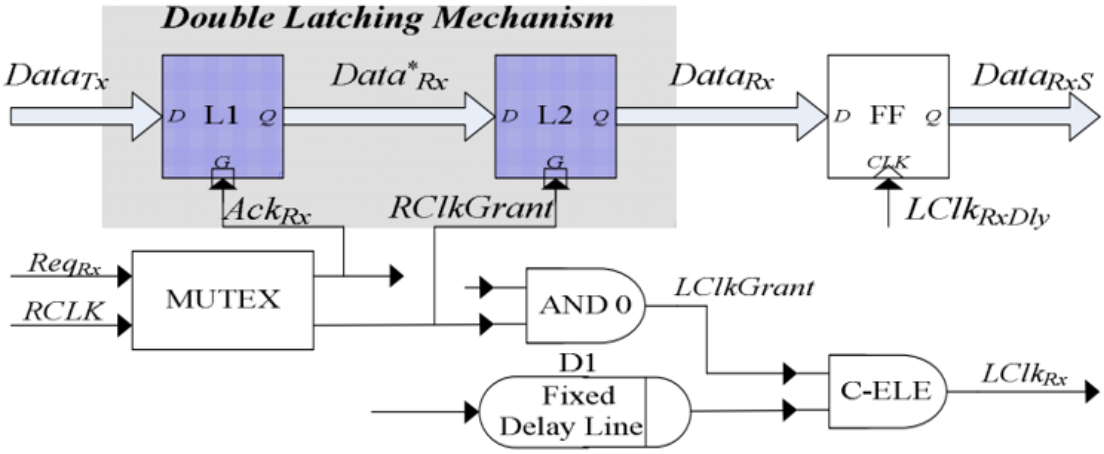


Figure 2.14: Double latching mechanism (from [FKG09])

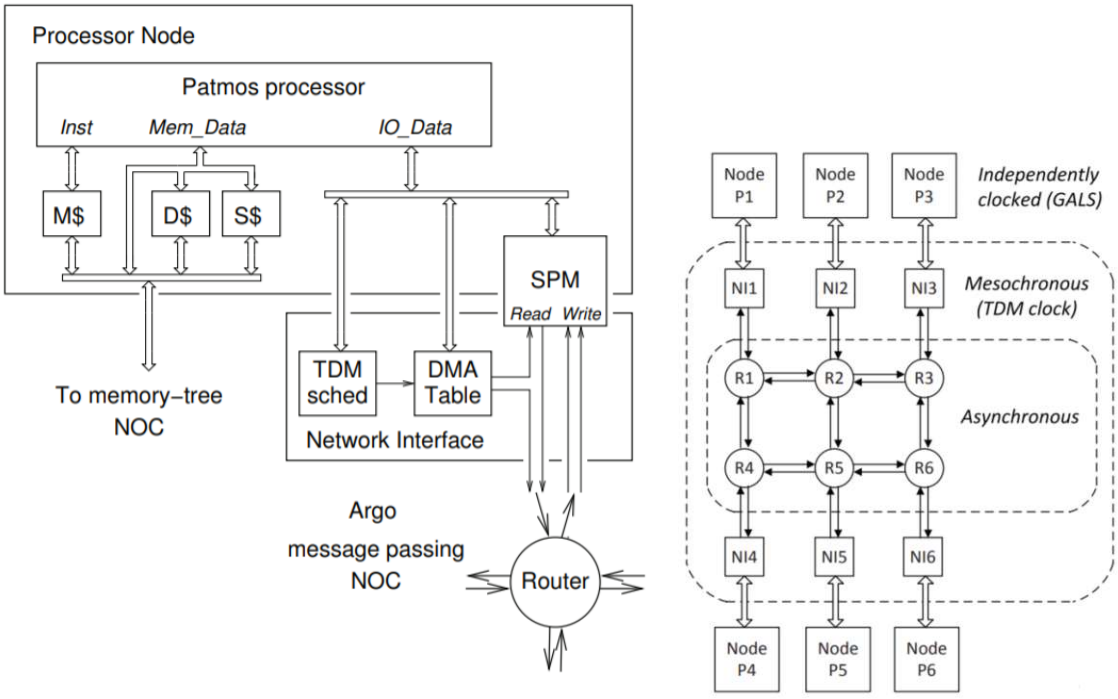
Optimized Poll Input Port Controller Poll input ports only halt the local clock during the handshaking process. Therefore the local clock is again released when the request signal from the remote module is deasserted. This leads to the large and unpredictable stretching on the input's local clock. In [FKG09] Fan et al. presented an optimized poll input port, where an additional transfer acknowledge signal is introduced. After acknowledging the clock pause request, the request acknowledge signal is sent to the transmitter and in parallel the transfer acknowledge signal is asserted. Subsequently the local clock can resume immediately after these two events. For the transfer acknowledge signal an additional latch is required that is enabled by the $RClkGrant$ signal, in order to synchronize the transfer acknowledge signal to the received data.

2.2 Concept of NoCs described on Argo

In [KSS+15] Kasapaki et al. presented a NoC approach called Argo that supports message passing across virtual channels and uses statically scheduled time division multiplexing

of the resources to ensure hard real-time properties. As illustrated in Figure 2.15b the routers work totally asynchronous whereas the network interfaces are mesochronous to each other.

Routers consist of three pipeline stages. One stage is for link traversal, the second one for header processing and the third one is for crossbar traversal. This allows routers without any circuitry for arbitration or buffering. The data is sent in in so called phits and every handshaking cycle one phit is consumed on all inputs by the router. One packet consists of three phits, each of them is a 32 bit data word along with three control bits. One phit is the header phit containing the destination write address and the static route.



(a) Architecture of a argo processor node (both pictures from [KSS+15])

(b) Timing organization of the Argo NoC

Figure 2.15: Argo NoC

The network interface of each processor node is a clocked synchronous circuit that handles the time division multiplexing. As the network interfaces are mesochronous to each other a FIFO is used as interface between the router and the network interface. For the data clock domain crossing between the processor node and its network interface the dual ported scratchpad memory (SPM) is used. In this way, explicit clock domain crossing is only needed for programming the DMA controllers.

2.3 Multi-point interconnects for GALS architectures

In his PhD thesis [Vil05] Thomas Villiger presented three different multi-point interconnect methods for GALS systems.

2.3.1 Modular GALS Interconnect (MOGLI)

MOGLI is a shared bus architecture, where the sender ports request the bus and after getting the grant to access the bus, data is transmitted to the receiver port. In Figure 2.16 a block level diagram is shown. A central tree arbiter decides which port gets the bus next and a central address decoder then decodes the address that is currently assigned to the bus and generates a request signal for the corresponding receiver. In order to not block the bus and produce deadlocks, a NAp signal is introduced. So no matter a port is ready or not, the sender immediately gets a response from the receiver and therefore frees the bus again for other transactions. If a NAp is received, the sender again requests the bus in its next clock cycle. This is then done until an Ap signal is received.

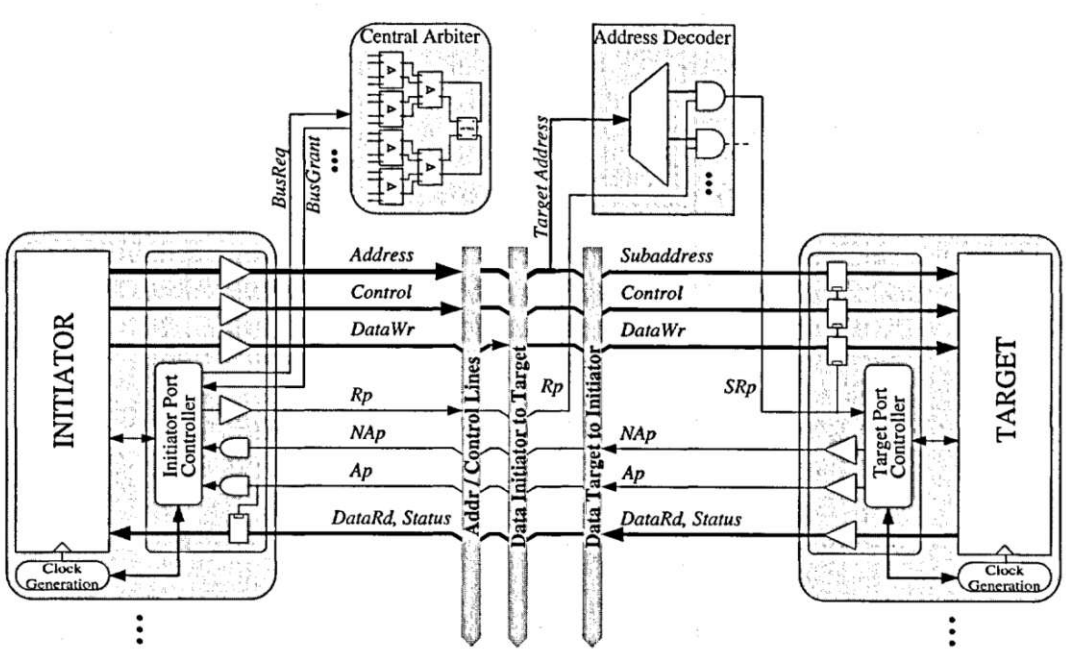


Figure 2.16: Single channel MOGLI (from [Vil05])

Villiger also presented a second version, where he introduced a second channel for responses. This is to reduce the time the bus is occupied by a transaction to a minimum, as the bus is not blocked for the time a memory is read out or a simple status request is taking place. However a second channel introduces latency, as now the response also must request the bus over an arbiter and the clocks need to be paused again.

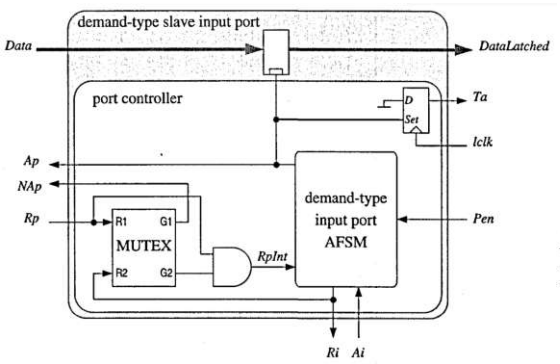
Adapted ports for bus access

The ports from Muttersbach that are shown in Figure 2.9 do not support bus access whereas the ports shown in Figure 2.17 that are presented by Villiger in [Vil05] do.

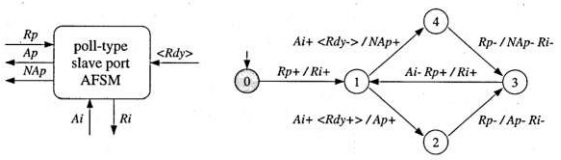
He extended the output ports from Muttersbach by a bus request and acknowledge signal. The request signal goes to an arbiter and when the arbiter grants bus access the acknowledge signal becomes active.

In order to avoid blocking the bus when the receiver is not yet ready to consume the data, not only an Ap signal is there for the handshake but also a NAp signal. So for example for the demand type input port a NAp is sent, when the clock has not been halted yet when the request arrives.

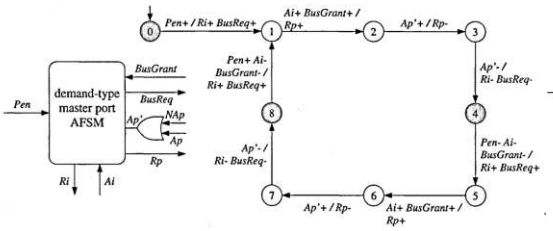
The poll input port from Villiger differs significantly from the one from Muttersbach. Instead of being only able to receive one data word and then being deactivated, it is activated as long as the $< Rdy >$ signal is active. This allows the port to receive multiple data words, one after another, while being active.



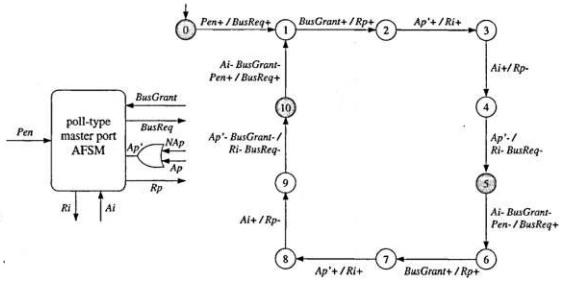
(a) D-Input



(b) P-Input



(c) D-Output



(d) P-Output

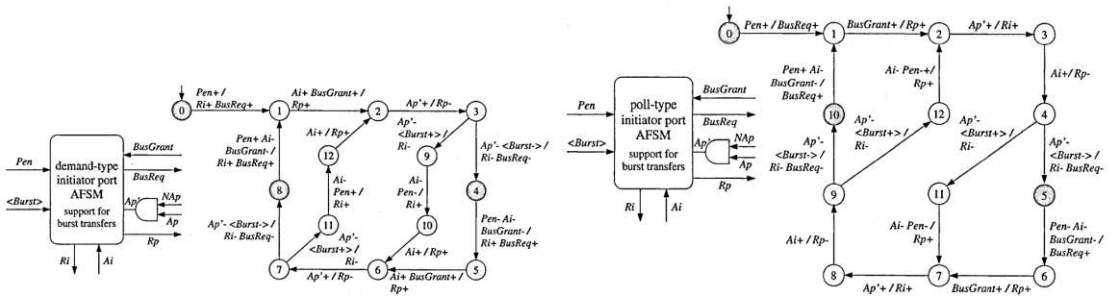
Figure 2.17: Default ports for bus communication presented by Villiger (from [Vil05])

Burst mode transactions With the adapted output ports from above it is not possible to send multiple data packages when the bus is granted. Instead the bus is released by

the output port after each single data transaction. This introduces quite some overhead, when a module wants to send multiple data packages over the bus with the same output port. Therefore Villiger also introduced the modified output ports given in Figure 2.18, that allow burst data transfers. The burst mode output ports do not release the bus request as long as not all messages are sent. However between every message that is sent a clock cycle is required to assign new data to the sender port. Thus every $\left\lceil \frac{T_r}{T_s} \right\rceil$ clock cycle, where T_r is the clock period of the receiver module and T_s is the clock period of the sender port, the sender can successfully transmit a data word. In the best case when data is sent to a module with a faster clock frequency, every cycle a data word can be sent.

According to Villiger the ports that support burst data transfers require a multiple of the area of the ordinary output ports. The burst transfer capable demand port requires even more than four times the area, whereas the corresponding poll port requires more than twice the area. When it comes to the additional duration for a data transaction, they both need roughly twice the time. However despite the higher transaction time an overall speedup due to the arbitration that is only done once in a burst transaction. The number m of burst transfers necessary to achieve a speed up compared to ordinary transaction method is calculated in Equation 2.6 where t_A is the delay introduced by the arbitration and t_B respectively t_N are the transaction times for burst transfers and normal data transfers.

$$m \geq \frac{t_A}{t_A - (t_B - t_N)} \tag{2.6}$$



(a) Burst D-Output Pipeline (b) Burst P-Output

Figure 2.18: Output ports for burst transactions (from [Vil05])

2.3.2 Self-timed Ring for GALS (STRING)

For ring based communication, all nodes are connected through a circular path where every node decides on whether to consume the message or bypass it to the next node.

Figure 2.19 shows such a structure. Villiger developed special ring transceiver modules that work fully asynchronously and consume the data depending on the address or

forward it. When forwarding it needs to be taken care of on whether to send the own output data, or the one currently stored in the transceiver.

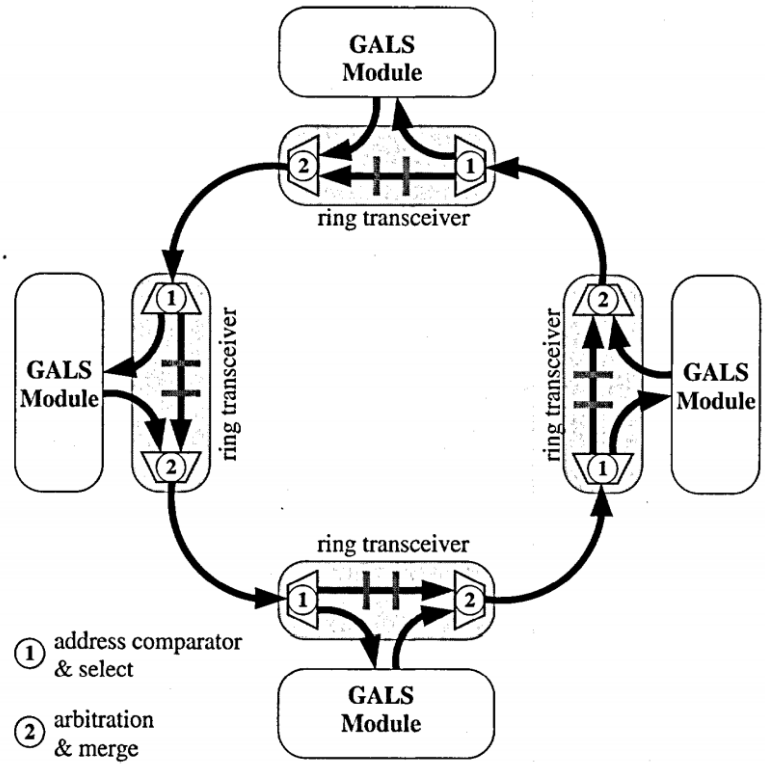


Figure 2.19: Ring architecture Villiger (from [Vil05])

2.3.3 Switching Network for GALS (SWING)

The third approach that Villiger presented is a switching network that implements a NoC architecture, where switches are responsible for setting up a direct connection between sender and receiver. In an enhanced approach shown in Figure 2.21 he also presents a switch element, that allows to not only set up one connection but even two, in case this does not lead to a collision. Collisions occur when due to the static routing a route is already blocked by another message.

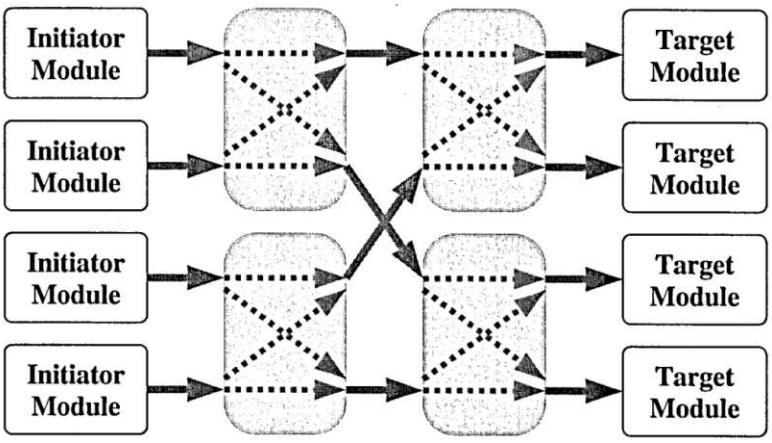


Figure 2.20: Switching Network Villiger (from [Vil05])

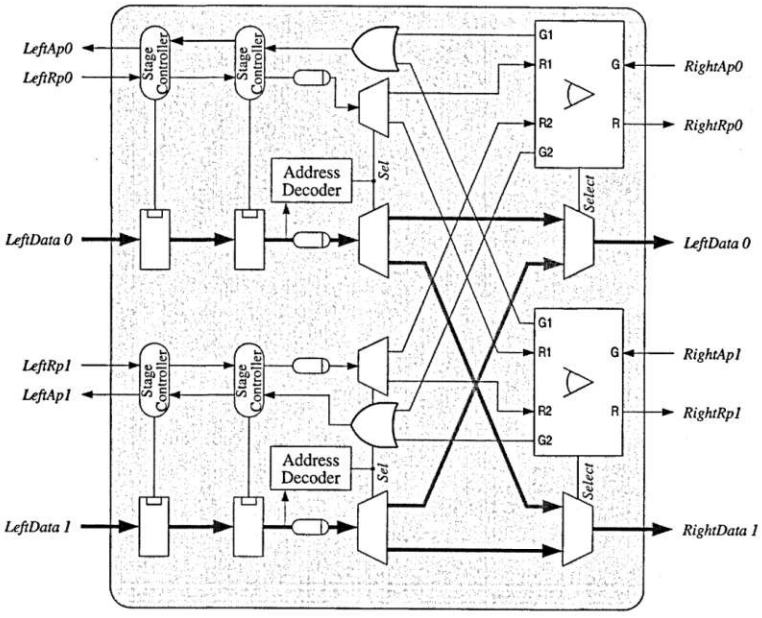


Figure 2.21: Switch crossbar element Villiger (from [Vil05])

CHAPTER 3

Multi Channel Bus

As outlined in the related work chapter 2 there are already approaches for point to point connected systems like the one from Muttersbach [Mut01] and bus designs like the ones from Villiger [Vil05]. In this chapter a novel architecture, shown in Figure 3.1, is presented. The idea is to get the best out of both designs. This is done by improving the bus architecture designed by Villiger by introducing so called channels.

Like in Villiger's approach, when a module wants to initiate a transaction, one of its output ports is enabled. After being enabled an ordering request is sent. This is because multiple ports can be activated simultaneously and therefore some ordering process is necessary. Since now more than one bus is available, this ordering mechanism is more complex than just one tree arbiter like in Villiger's MOGLI. The address of the output port that is selected by the ordering process is stored in a pipeline from where a channel that is currently idling, tries to get the address through a channel selection module. The requests from the idling channels again have to be arbitrated, as multiple channels may try to request an address simultaneously.

Through introducing multiple channels the throughput can be increased, while the additional area and interconnect that is needed can be kept low. Each channel can establish a point to point connection between an input port and an output port. Whenever a module wants to send something, it first requests a channel. After a free channel is assigned and a point to point connection is established correctly, data is transmitted. Also burst mode transactions are possible, where more than one data word is sent per bus access. Therefore simply burst mode supporting output ports like the one from Villiger [Vil05] can be used instead of Villiger's adapted output ports from Muttersbach. Everything between the sender and receiver port works asynchronous. This also includes the channel selection process which is fully asynchronous.

The difference to the two channel MOGLI [Vil05] presented by Villiger, is that there is not one dedicated channel for request and one for response. Instead all channels can

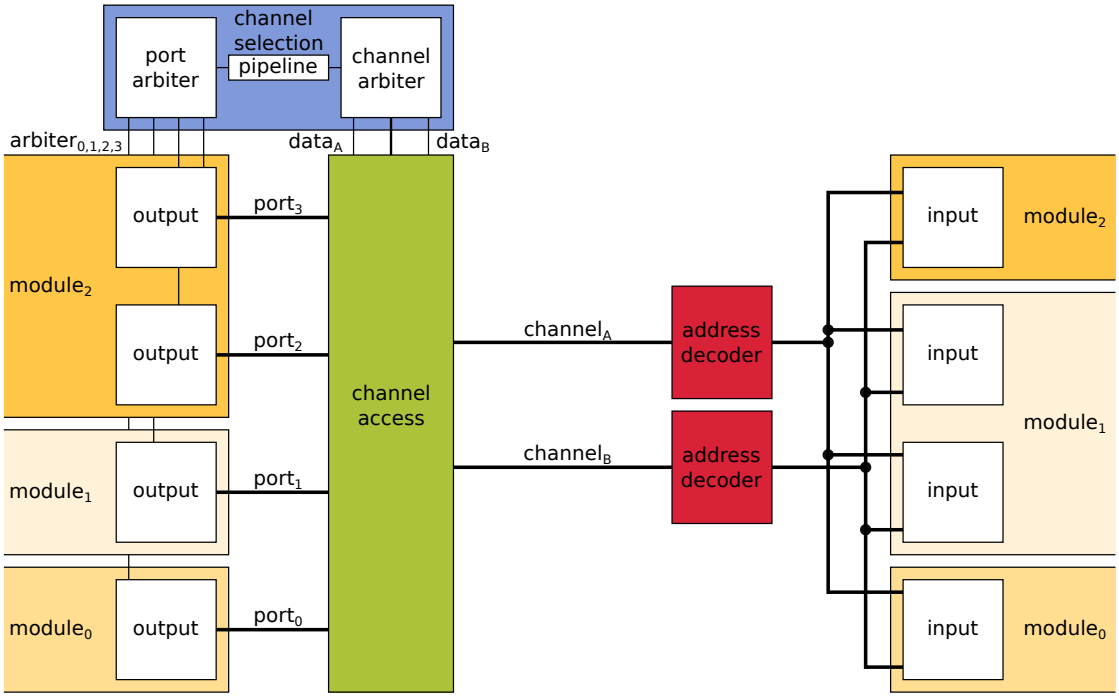


Figure 3.1: Block diagram of a multi channel bus with two channels and 3 modules and one single pipeline

support both types of messages and therefore more than two channels are possible.

3.1 General problem description

The idea behind the above described approach is to provide a communication network for a distributed system with n nodes and s services. Each node $i \in [0, n]$ of the system may provide or use data of some of those services. n_{i_s} is the total number of services that are provided at node i and n_{i_r} is the amount of services that send something to node i. In a first step for every service that may send something from node i a sender port $k_{i,j}$ ($j \in [0, n_{i_s}]$) is introduced and for every service where data can be received, a receiver port $p_{i,j}$ ($j \in [0, n_{i_r}]$) is introduced. So in total there are $k = \sum_{i=0}^{n} \sum_{j=0}^{n_{i_s}} k_{i,j}$ sender ports and $p = \sum_{i=0}^{n} \sum_{j=0}^{n_{i_r}} p_{i,j}$ receiver ports. Albeit k and p are fixed by the underlying algorithm, it might be that one port is able to handle multiple services, when they are in a way interlocked that they can never occur concurrently. Thus the number of actually implemented ports might be even less than calculated above.

The communication between the k sender ports and p receiver ports takes place over m channels. Those m channels receive the information of which output port they are wiring through next from one of the q pipelines, where $0 < q \leq m$ holds. Those pipelines allow hidden arbitration, which means that while a transaction is still in process, it is already

evaluated which output port is allowed next to send data.

A very challenging problem is to determine m and q properly as the best trade off between used area and latency may highly depend on the algorithm. Since one big advantage of pausable GALS application is that the latency is kept as low as possible, this should also be the case for the multi channel bus. However to achieve this, it is necessary that there is one idling channel whenever a request occurs. Thus to keep the latency at its minimum, m needs to be equal to the maximal number of service requests that might be active simultaneously, which requires an in depth analysis of the algorithm. As this might be difficult to analyze and furthermore the maximal number of active services at a time might differ greatly from the average amount of active services it might be more reasonable to choose m in a way such that it is somewhere between the average and the maximal number of simultaneously active requests. Consequently all channels are well utilized but the requesting sender ports do not need to wait disproportionately long until they get assigned to a channel.

Another important decision is how many pipelines q are necessary and furthermore how many channels should be assigned to one single pipeline. So in general here there are three different methods:

- One single pipeline ($q = 1$)
  In this case all channels get their data from one single pipeline and thus the output ports may be assigned to one of the m channels. Using only one pipeline is clearly saving area but might also introduce quite some additional latency especially when the transaction durations are way less than the time it takes to acquire a channel.

- multiple pipelines ($1 < q < m$)
  To reduce latency more pipelines are introduced. As now not all ports have access to all channels anymore but only to a disjoint subset, the workload needs to be balanced accordingly. This requires some analysis, as it needs to be known, which ports are sending data more frequently than others or do burst transactions which occupy the bus for quite some time. Pipelines that store requests from such ports may feed data to multiple channels, as otherwise other requests need to wait quite some time until they get assigned to a channel. In case one output port is very active and needs to send time critical messages, it might be even possible to assign one pipeline that feeds one channel with this port.

- One pipeline for each channel ($q = m$)
  One pipeline for each channel is the maximal number of pipelines that is possible. Although being the most area intense method it might not be the one where the latency is at its minimum. This is because in case the workload is not balanced well or can not be analyzed statically there might be one pipeline that is full and its channel is also busy while all other pipelines are empty and their channels are idling.

The scheme of a multi bus channel presented in Figure 3.2 is an example for the multiple pipelines method where $n < m$. The analysis of the underlying algorithm might have shown that $sender_0$-$sender_4$ rarely want to send something and thus they all share one pipeline and as it might be the case that when one of the ports is active it occupies the channel for quite some time, two channels are there for those five ports. $sender_5$ and $sender_6$ share one pipeline and one channel. Here it might be the case that both channels do not occupy the channel very long when they request it or if they do, it is not critical for the overall system, when one of the senders has to wait longer until it can send its message. For $sender_7$-$sender_9$ holds the same as for $sender_0$-$sender4$ although those three senders might be more active and thus they got assigned two channels to send their data over.

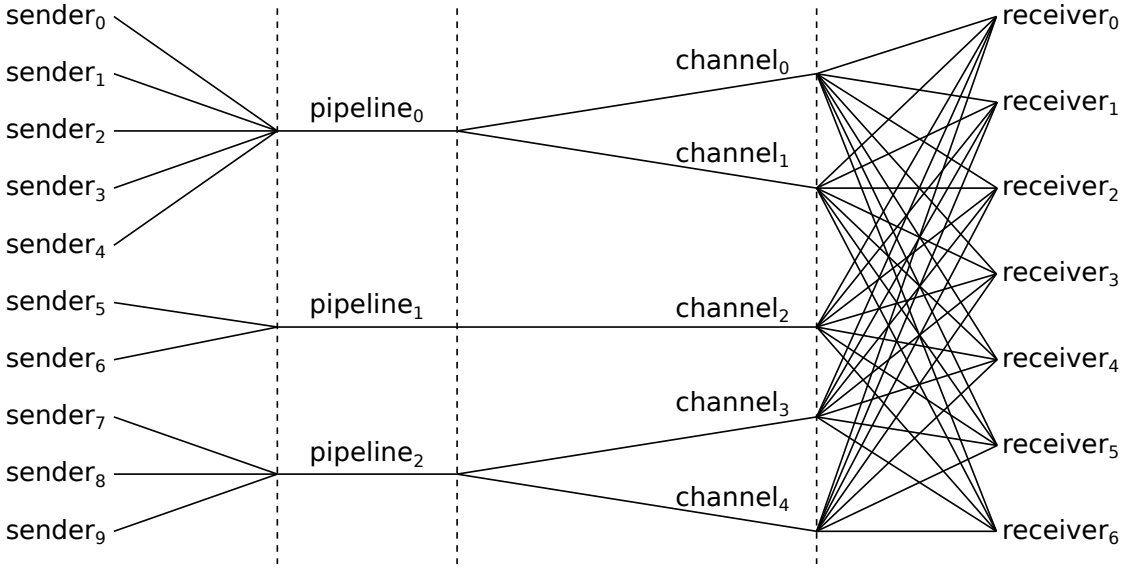


Figure 3.2: Scheme of a multi bus channel where $k = 10, p = 7, m = 5$ and $q = 3$

3.2 Channel selection

The most challenging part, when extending the MOGLI approach to support multiple channels, is the question how a free channel is selected and assigned to an output port that is requesting a channel. Therefore the requests need to be sorted somehow and then matched with one of the channels.

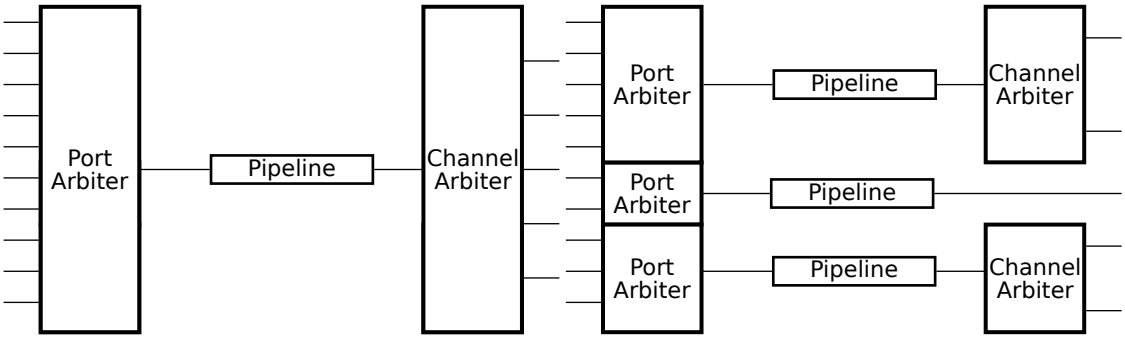
3.2.1 Single Pipeline

One way to achieve this is by queuing all requests in an asynchronous Muller pipeline like the one presented in [Sut89]. Which sender address is stored next is determined by an arbiter. On the output of the queue there is another arbiter that decides to which channel the sender gets assigned. Figure 3.3a shows a schematic of this approach. Each

sender can be assigned to any of the available channels. However the single pipeline is a bottleneck as outlined above because it requires all requests to be serialized before they are assigned to parallel channels. So it is only applicable when the time spent for the actual transaction is longer than the time it needs to traverse the queue.

3.2.2 Multiple Pipelines

To overcome this bottleneck, more pipelines can be introduced, going up to one pipeline per channel. However now each sender port has only dedicated channels over which it can communicate. In case there are as many pipelines as channels, there is even only a unique channel over which it can communicate, thus the arbiter at the output of the pipeline becomes obsolete. When using the multi pipeline approach it is important to first analyze which senders send most frequently, in order to distribute them well over all channels. A schematic of how this looks like is shown in Figure 3.3b.



(a) Single Pipeline (b) Multiple Pipelines

Figure 3.3: Channel selection mechanism

3.2.3 Channel selection block implementation

A complete implementation of a channel selection block with one pipeline is shown in Figure 3.4. When a port wants to send data over a channel it first sends an ordering request to the channel selection block. The requests are ordered with an arbiter, which then handles the pipeline access. This is done by using the grant signals of the arbiter to feed a 1-of-n asynchronous QDI pipeline [ZSG+13]. When there is space left, the grants are forwarded into the pipeline and after the first stage received the data, the output data of the first stage is used to indicate the output port adapter that the request was successfully written into the pipeline. The acknowledge signal that is generated in the last pipeline stage is also used as request signal for the output of the pipeline, as the pipeline itself only needs an acknowledge signal for working properly.

Reading an entry from the pipeline is a bit more challenging than writing an entry to the pipeline. The reason for this is that the channel arbiter only requests the port address

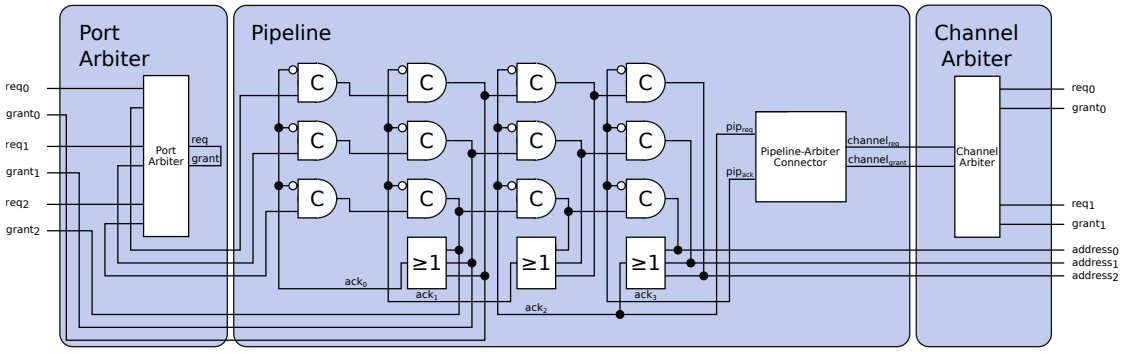


Figure 3.4: Implementation of single pipeline with 3 stages and two entries

from the pipeline until a grant is received and does not take care of removing the entry from the pipeline properly.

To overcome this issue a special pipeline-arbiter connector is introduced. An STG of the connector is presented in Figure 3.5. An arbiter request is only granted, when the pipeline indicates that it is not empty, by activating pip_{req}. After the channel helper consumed the port address and thus removed the channel arbiter request signal, the pipeline entry is consumed and removed by setting pip_{ack} and deasserted again after pip_{req} becomes inactive.

The synthesis of the STG by Workcraft [PSM07] results in the Equations 3.1 and 3.2.

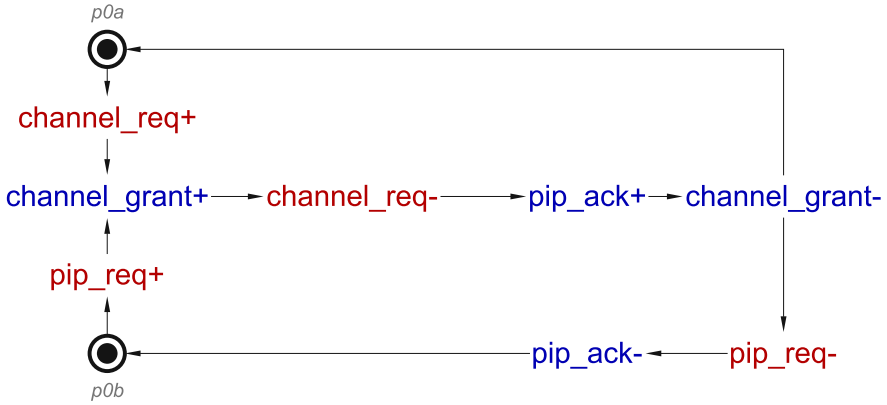


Figure 3.5: Pipeline-Arbiter connector STG

$$channel_{grant} = pip_{req} \cdot \overline{pip_{ack}} \cdot channel_{req} + channel_{grant} \cdot \overline{pip_{ack}} \tag{3.1}$$

$$pip_{ack} = channel_{grant} \cdot \overline{channel_{req}} + pip_{ack} \cdot pip_{req} \tag{3.2}$$

The timing diagram of Figure 3.9 shows how two addresses are read from the pipeline successively. Although the channel arbiter requests the next port address while the

request from the pipeline is still active, it does not get the grant until the next port address is ready in the pipeline. The $pip_{address}$ is the address that is currently waiting in the pipeline until it is read by a channel. When it is consumed by a channel it is stored as $channel_{address}$ in the channel helper that got assigned to the address and the address in the pipeline can be overwritten by the next request.

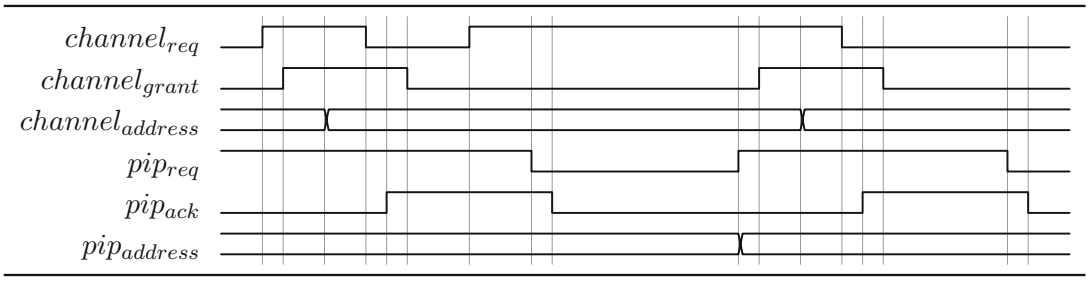


Figure 3.6: Timing diagram of the pipeline-arbiter connector

3.3 Channel access

To connect the output port, the channel selection module and the channel itself with each other, two additional asynchronous finite state machines (AFSM) are needed. One is needed as an adapter for the output port and the other one is needed for the channel. Those AFSMs are then connected with each other like in Figure 3.7.

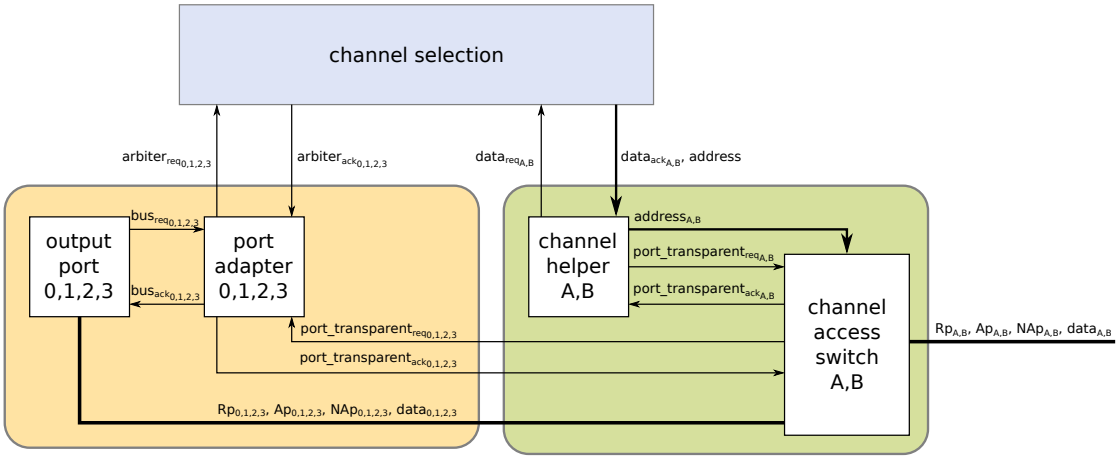


Figure 3.7: Block diagram of channel access with four output ports, two channels and one pipeline

When an output port requests the bus, the channel helper sends an arbiter request and after a channel finally gets assigned the *port_transparent_req* signal becomes active and the output port is informed that it now can send data over a channel. The output port does not need to know which channel it got assigned to.

3.3.1 Output port adapter

The output ports of Villiger are not designed for the way of channel selection described above. Therefore the AFSM in Figure 3.8 is introduced. It consists mainly of three different phases. The first phase is initiated after a bus request. The adapter then sends a request to the arbiter until this one responds with an acknowledge signal. Now the sender address is stored in the pipeline. In the second phase the adapter waits until a channel acknowledges that data can be sent over it safely. After that acknowledge the third phase is entered, where data is then really sent over the bus.

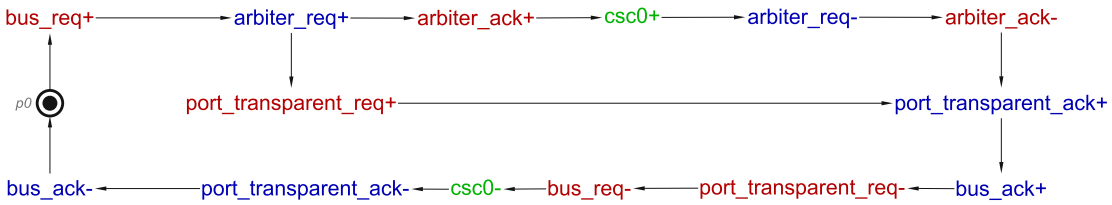


Figure 3.8: Output port adapter STG

The synthesis of the STG results in the Equations 3.3-3.6. The green label $csc0$ results from a complete state coding (csc) confilct. A csc conflict arises when two semantically different reachable states have the same encoding. The encoding is done with all signals of the STG and thus in case two states have an identical encoding a new variable must be introduced. In this case this variable is $csc0$ and it is representing a state indicating whether currently the port can request a channel ($csc0 = LOW$) or the channel already has been requested successfully but the port has not completed the transaction ($csc0 = HIGH$).

$$csc0 = bus_{req} \cdot csc0 + arbiter_{ack} \tag{3.3}$$

$$port\_transparent_{ack} = csc0 \cdot port\_transparent_{ack} + port\_transparent_{req} \cdot csc0 \cdot \overline{arbiter_{ack}} \tag{3.4}$$

$$arbiter_{req} = bus_{req} \cdot \overline{csc0} \tag{3.5}$$

$$bus_{ack} = port\_transparent_{ack} \tag{3.6}$$

A possible timing diagram where the three phases are marked can be seen in Figure 3.9. Note that here no exact timing is provided.

3.3.2 Channel helper

The channel helper is responsible for providing the channel with new transactions. Therefore whenever the channel is not occupied by some ongoing transaction, it requests a new output port address from the channel selection module. After being assigned a new address, the second phase is initiated. During this phase *valid* is active and

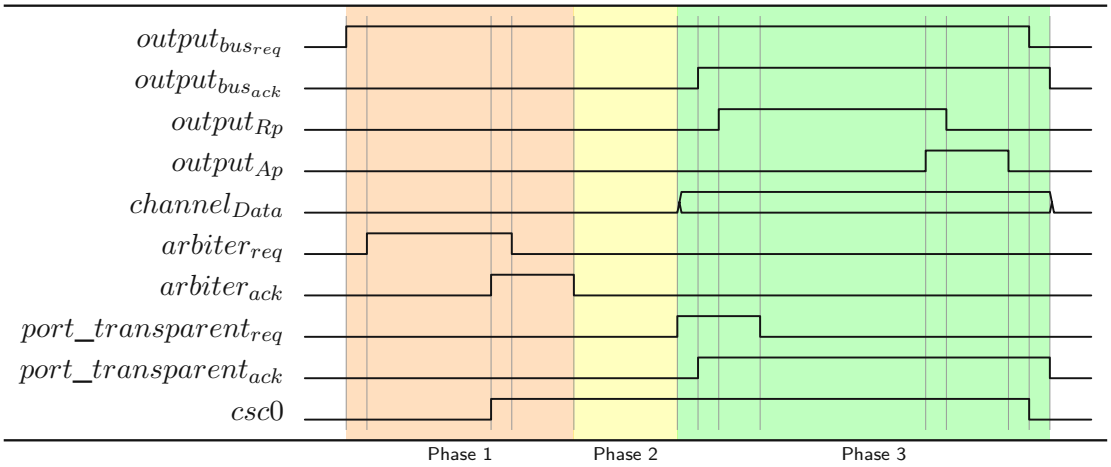
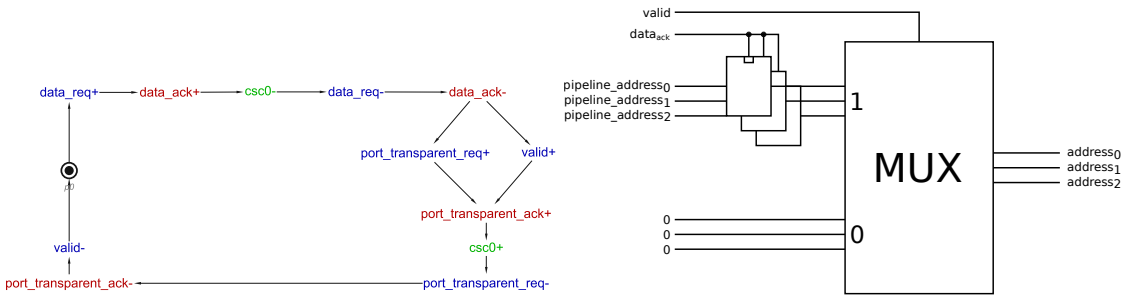


Figure 3.9: Timing diagram of output adapter with control signals from the output port

therefore the address is propagated to the switching module as shown in Figure 3.10b. Over the switching module the sender port is also informed, that it can start its communication with the receiver. This communication channel is then kept intact until the *port_transparent_ack* signal is deactivated. Then *valid* is inactive and no port is selected by the address anymore.



(a) Channel helper STG

(b) Channel address activation

Figure 3.10: Channel helper

The synthesis of the STG shown in Figure 3.10a results in the Equations 3.7-3.10. Again a csc conflict is resolved by an additional state variable *csc*0 that is initiated with zero and is responsible for separating the phases. When *csc*0 is zero a new address has been received but the corresponding port has not responded yet.

In the timing diagram of Figure 3.11 the address signals are also added to show how the internal valid signal sets the *address* which then is propagated to the switching module.

$$csc0 = port\_transparent_{ack} + csc0 \cdot \overline{data_{ack}} \tag{3.7}$$

$$data_{req} = \overline{valid} \cdot csc0 \tag{3.8}$$

$$port\_transparent_{req} = \overline{data_{ack}} \cdot \overline{csc0} \tag{3.9}$$

$$valid = \overline{data_{ack}} \cdot \overline{csc0} + port\_transparent_{ack} \tag{3.10}$$

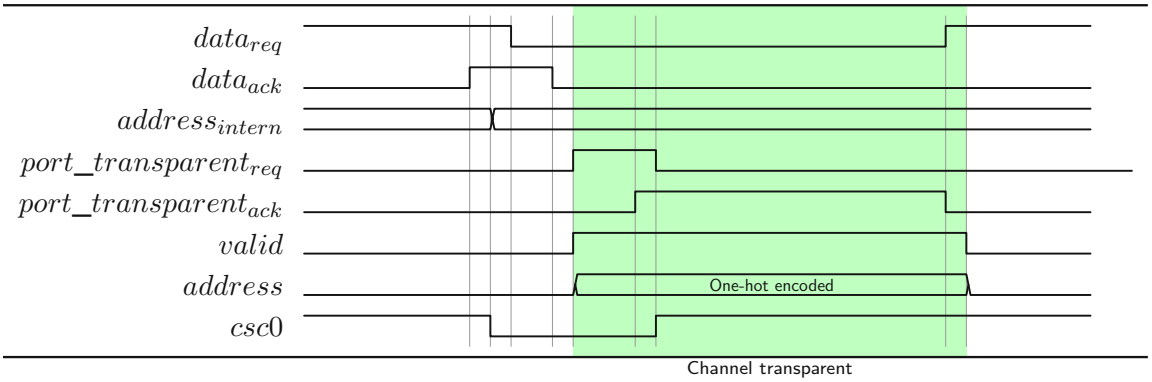


Figure 3.11: Timing diagram of channel helper

3.3.3 Channel access switch

The channel access switch is responsible for mapping the output port selected by the address to the channel. The address here works as a multiplexer select signal. Important is that the channel switch not only maps the signals that are required for the asynchronous bus communication but also the *port_transparent* signals that are used as communication between the channel helpers and the port adapters. When no port is active, the channel's request and data lines are kept zero. The same holds for all *Ap* and *NAp* signals of the ports that are currently not selected by the channel. Since there the other channels also need to be able to write to, this is done via pull downs.

3.4 Data reception and addressing

Depending on the communication assumptions the receiver side needs to be adapted accordingly compared to the MOGLI approach, with a central address decoder, presented by Villiger.

3.4.1 Poll input port clock pausing problem

Xin Fan optimized the pausable clock generator in [FKG09] and presented an example circuit (Figure 2.13) with two pause requests. Let's assume now that *Req*0 comes from a demand input port and that *Req*1 from a poll input port. As the demand port

immediately halts the clock, $Ack0$ gets the grant. After delay $D0$, $RClkGrant1$ gets the grant from the clock. When the demand port has not received anything so far, the clock is still halted but because $MUTEX1$ already gave the grant to the clock, nothing can be received at the poll port.

Since for the ports it only matters if the clock is halted or not and not whether they halted the clock or someone else, one MUTEX element is enough. All pause requests are ored to one single pause request req_{pause}. The ports then only check if the clock is paused. Important here is that the request signal to the AND gate is delayed and that it holds that $\Delta > OR + MUTEX$. This is because it might be the case that the port that paused the clock so far just deasserted the request signal but the ack_{pause} is still active when the new request req_i arrives. A glitch for ack_i might be the consequence, without the delay elements.

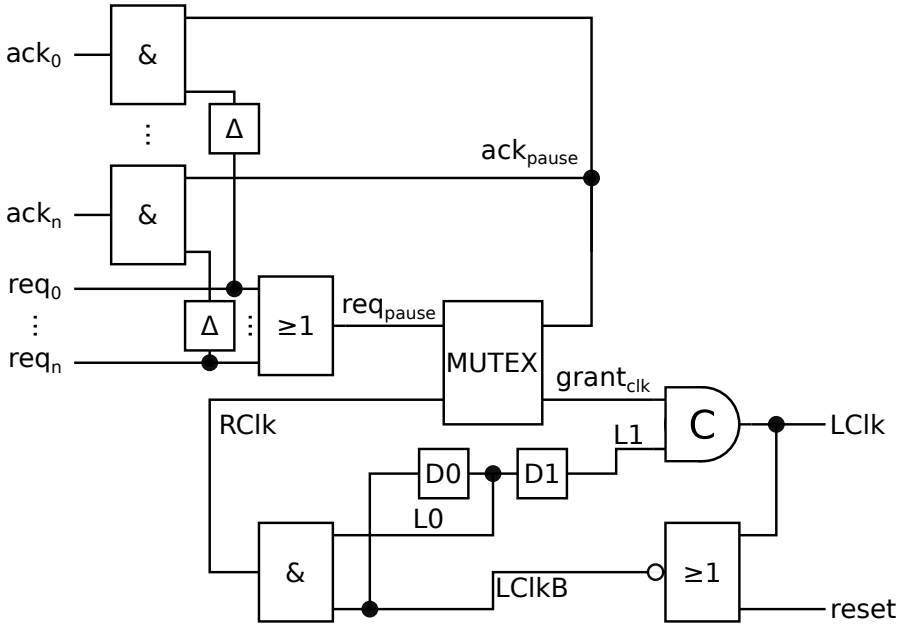


Figure 3.12: Adapted clock pausing mechanism

Due to the adaptation of the pausing request also the calculation for delay d_{D1} changes as outlined in equation 3.12. The change compared to equation 2.5 is that the $AND0$ gate is not necessary in the new pausing mechanism and thus d_{D1} can be further reduced.

$$d_{D0} = TLClk/2 - (d_{D1} + d_{C-ELE} + d_{NOR}) \tag{3.11}$$

$$d_{D1} = d_{AND} + (d^0_{MUTEX} + \Delta_{MUTEX}) \tag{3.12}$$

3.4.2 Valid data identification

Muttersbach designed all his ports such that they can send respectively receive one data word per port enable. The enabling of the ports works synchronously with the clock. To avoid the extra cycle for the return to zero, which is necessary when the enable signal follows a four phase protocol, the enable signal follows a two phase protocol instead. Whenever it toggles its logical value it enables the port for another data word.

Since Villiger's ports also work with this principle, when not considering the poll port input that works slightly different, a circuit, that identifies whether data was already received successfully or not, is presented in Figure 3.13. The idea is that while the data is latched, an additional latch, functioning as edge detector, latches the *Pen* signal, which is responsible for enabling the port. An XNOR gate then decides whether the current data in the latches is valid or not. When the current *Pen* signal and the latched *Pen* signal are identical, the data that is stored is correct, otherwise not. Since the clock is paused during the time data is stored in the latch, no metastable valid is read. Furthermore immediately after the port is enabled, the data is marked as not valid and therefore not even an explicit reset for the valid signal is necessary.

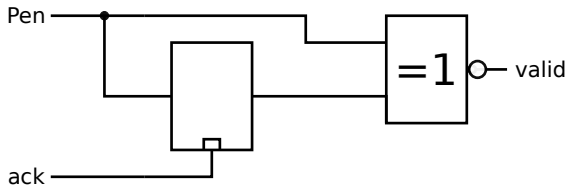


Figure 3.13: Data valid identification

3.4.3 Point to Point

For this approach the assumption is that, like in the design presented by Muttersbach, there are always pairs of input and output ports that communicate exclusively with each other over the bus. They communicate with no other ports. So even when there are multiple channels and therefore multiple data transmissions going on simultaneously, there occurs no conflict because never two ports want to write to the same port. This allows the use of only one input for all channels and like in Figure 3.14, the request signals of the channels are used as select signal for the multiplexer. The input port then writes the data into a latch and also sets a valid flag in the way described above.

Per channel one global address decoder like in Villiger's MOGLI approach can be used. The downside however is, that even though the same service is accessed by multiple modules, for every distinct module a unique input port is needed.

3.4.4 Concurrent receiving

To overcome this issue, there are basically two methods. The first one is, that the requests of all channels are arbitrated whereas the second method is that every channel has its

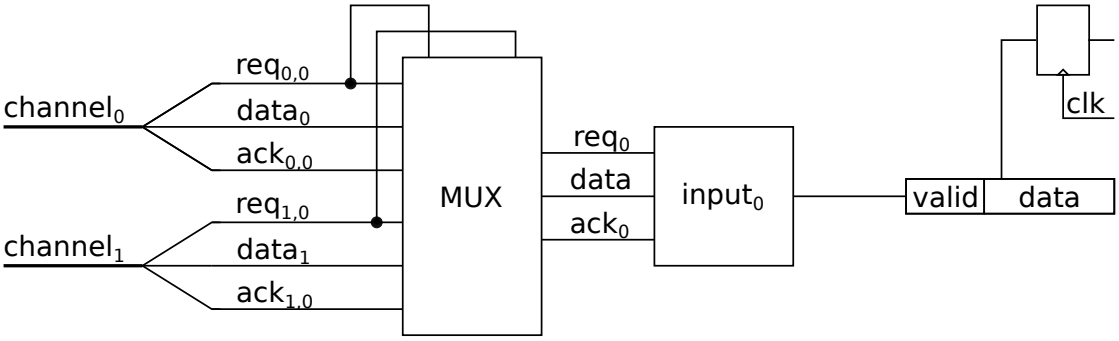


Figure 3.14: Channel multiplexing for point to point communication

own set of ports. The first method increases the latency, as all requests now have to be arbitrated and the second method introduces quite some overhead but allows receiving data really concurrently. To keep the latency respectively area overhead minimal, this can also be only an option for input ports that may receive data from multiple ports at the same time.

To process the data correctly the sender now also has to send its address or some other information with the data package.The additional information is used, to store the data package in the correct data entry. Depending on how the input works it might be ideal to even send this address in a delay insensitive encoding like the one-hot encoding, because then no additional delay has to be introduced, in order to make sure that no metastable address is read.

Concurrent Demand type input port

As demand type input ports are only able to receive one data word each time they are enabled and to not halt the module forever, it makes sense here to serialize the concurrent requests with a MUTEX.

Figure 3.15 shows this approach. Depending on which channel gets the grant, a multiplexer forwards the data of the corresponding channel and when no channel is selected, a $zero_{vector}$ is assigned. It is important here, that no channel is selected as default, as then it might happen, that the selected channel is switched in the middle of an ongoing transaction. This happens when the arbitration takes so long, that the input port already started with processing the request and the arbiter gave the grant to another channel but not the one that is currently active. Important to note here is that the concurrent demand input port needs no *valid* signal as the module anyway only resumes, after some data has been received.

Concurrent Poll type Port Unlike the demand type input port from Villiger, the poll type input port is not immediately deactivated after receiving a data word. This allows the port to receive more than one data word, even within one clock cycle and

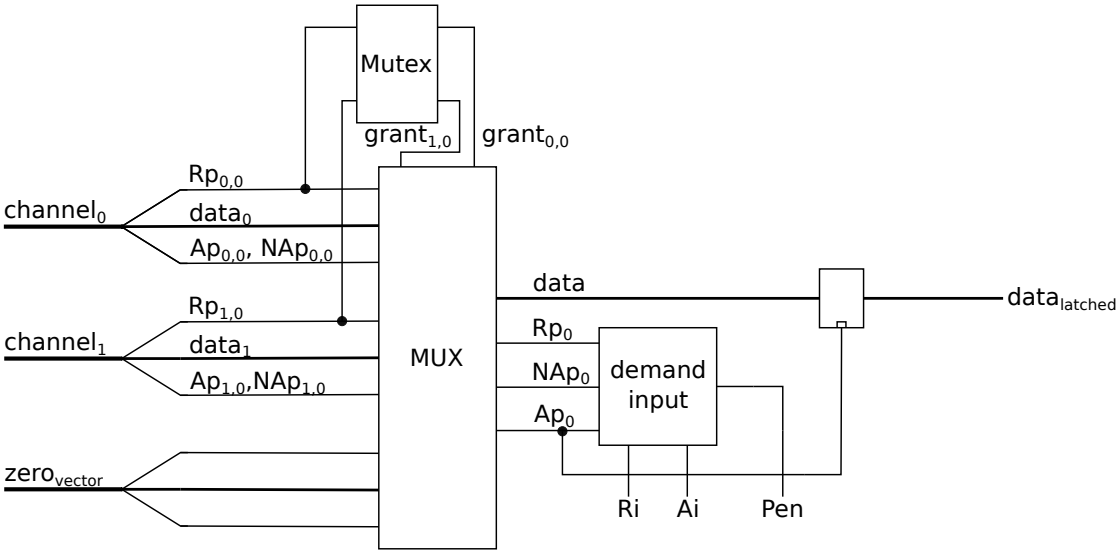


Figure 3.15: Concurrent receiving with Mutex

store the data in a table entry. The table is built with latches and for every possible sender/service one unique entry exists. This is important in order to not get problems with concurrent writes to the same table address. Depending on the kind of data that is transmitted it might be necessary for the system to not overwrite it until the module has consumed it. On the other hand there might be data like for example a value from a sensor that is read and sent repeatedly and thus no special handling is necessary in that case.

Consumption check In case it is important that the receiving module really processes the message that is sent, it needs to be taken care of during the receiving process, that the table entry is only updated, when its valid flag is indicating that there is currently no valid data stored and thus data can be written to the entry as previous received data is already processed.

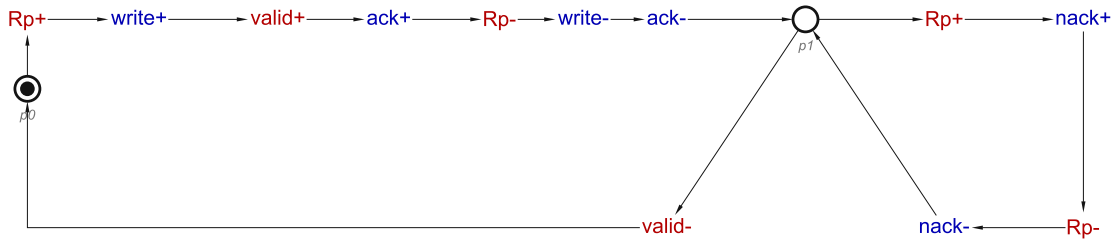


Figure 3.16: Write helper STG

To check this a special asynchronous write helper module is designed. Its STG is shown in Figure 3.16 and results in the Equations 3.13-3.15. As demonstrated in the timing

diagram 3.17, it only allows a write, when valid is not set at the beginning of a transaction. When a second request Rp arrives while $valid$ is active, a $nack$ is sent immediately.

$$ack = valid \cdot write \tag{3.13}$$
$$nack = valid \cdot \overline{write} \cdot Rp \tag{3.14}$$
$$write = \overline{valid} \cdot Rp + write \cdot Rp \tag{3.15}$$

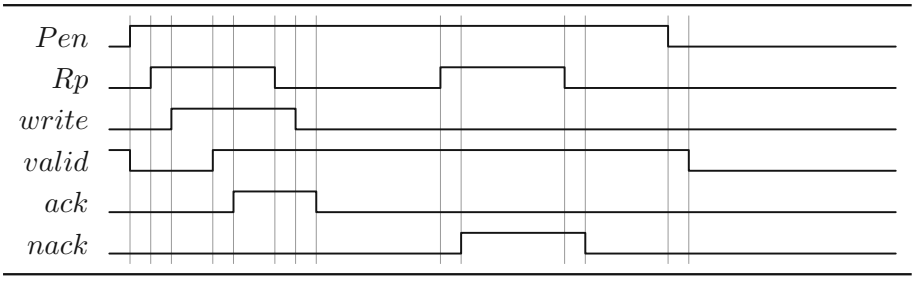


Figure 3.17: Timing diagram of write process

The address sent with the data must be DI encoded when the concurrent receiving with the latch table is implemented like in Figure 3.18. Otherwise it might happen, that in the beginning of the receiving process another address is selected as in the end, because not all address signals arrive simultaneously. Another work around is to delay the channels Rp signal long enough so that the address signals are certainly correct.

When now a request Rp is detected by the poll port, the clock is halted and after the clock is halted successfully the request is acknowledged with Ap_p. In case the poll port is not even activated, the port responds with a NAp_p. Instead of directly sending the Ap_p signal back to the receiver, it is now sent to $write\ helper_i$ and there works as Rp_i. The $write\ helper_i$ then sends depending on whether it was able to write the data to the latch or not an ack_i or a $nack_i$. Those signals are then routed back to the according channel via the switch module. The ack_i is there directly wired to the corresponding Ap signal of the channel whereas the $nack_i$ is ored with the NAp_p signal to form the NAp signal of the channel.

Update value only When the table entry only contains data where no explicit consumption is necessary, the whole write helper module can be omitted. Even the valid signal is of no interest in that case. However important here might be that the initial values of the latches lead to no problem within the algorithm. Figure 3.19 shows a poll input port where all entries are just receiving uncritical update values. Again Ap from the poll port is used to for storing the data. Since now no valid can be taken to make sure that the data is already stored in the latch it might be necessary to delay the acknowledge signal, that is sent to the sender port sufficiently enough, such that there was enough time for the data to be stored correctly.

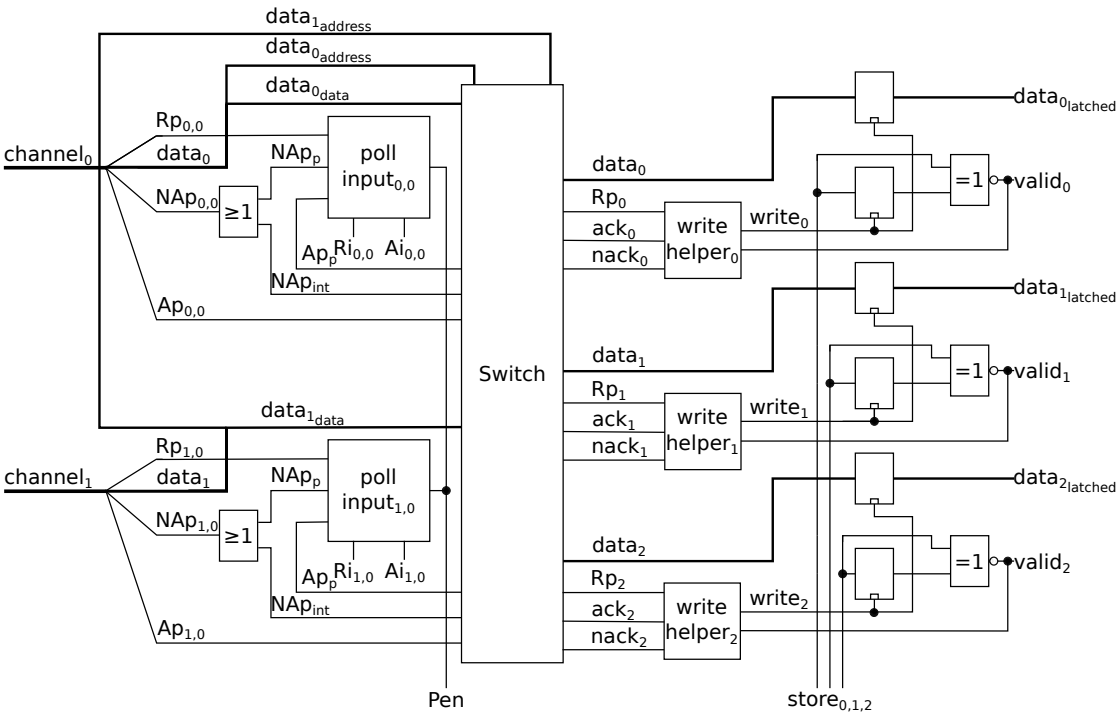


Figure 3.18: Concurrent receiving Villiger's poll input port and write helper

However there might be also input ports with mixed table entries. Some where explicit consumption is required and some where this is not necessary. In that case the Ap from the input poll port is forwarded over the switch to then be immediately returned ($Rp_i = ack_i$) and the corresponding $nack_i$ is set to zero permanently.

3.4.5 Multicast

Another property that is interesting for some applications is the possibility of multicasts. Therefore the central address decoder has to be removed and instead for each input port the request and acknowledge signals can be accessed directly by the output port. A transmission is completed on the sender side, when from all ports that should receive the message either an acknowledge or a not acknowledge is received. At the sender's next clock cycle a new transaction is initiated and a channel is requested, to resend the message to all ports that answered with a not acknowledge previously. This is then repeated until no port is left that has not answered with an acknowledge.

A possible implementation is presented in Figure 3.20. First it is checked whether a Ap_i or a NAp_i is received and then it is checked if this matches with the request Rp_i. In case all ored acknowledge signals match with their corresponding request signal, Ap' is active. Ap is the acknowledge signal that is forwarded to the sender port to terminate the transaction and is only active as long as there is at least one Ap_i or NAp_i signal

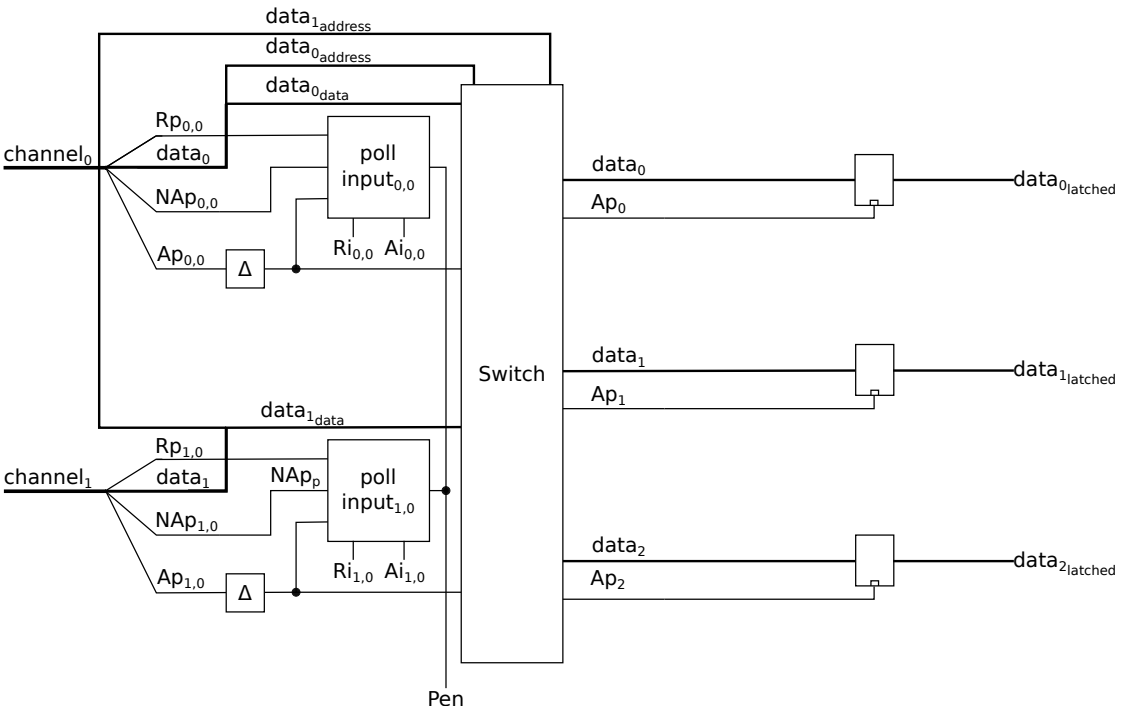


Figure 3.19: Concurrent receiving Villiger's poll input port without valid signal

active. All Ap and NAp signals are latched when Rp of the sender port is active. This is necessary because the module needs to assert the Rp_i signals of those input ports, that answered with a NAp signal again. The locally synchronous module checks the Ap' signal. When it is active the module knows that the transaction is over.

3.5 Data transaction

Figure 3.21 shows a flow chart of a full data transaction. The fat black route marks a possible path for the data path. However on the sender side multiple different paths are available, depending on whether the input port is already ready to take data or not. For simplicity the clock pause requests are left out in the flowchart as depending on the port type, this request is at different positions. The green marked circuit is handling the bus access and makes sure that only one port at a time can access a channel. The orange marked connections are helpers for completing a transaction. Another simplification in the chart is that all handshakes that are not marked with $+-$ signals are working such that when an acknowledge is received, the request signal returns to zero followed by the acknowledge signal.

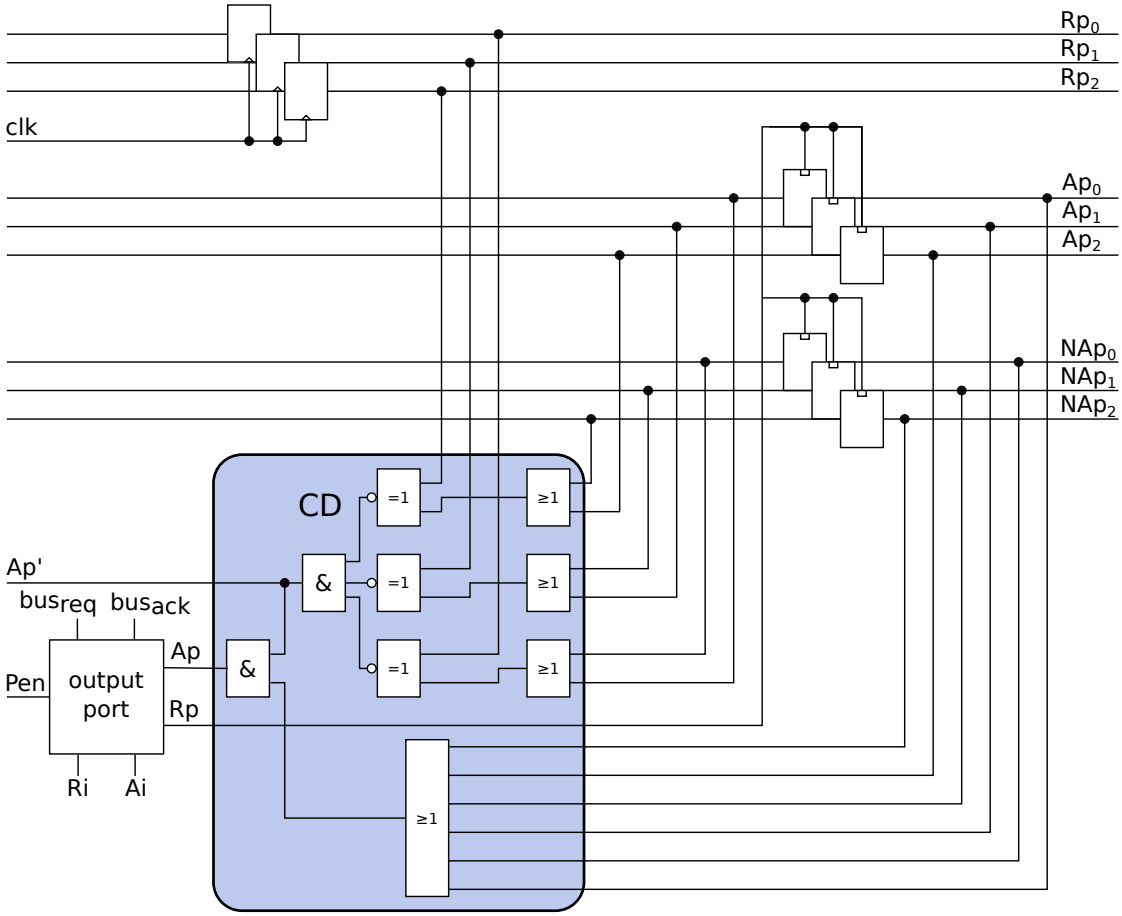


Figure 3.20: Multicast Ap' and Ap calculation for sender port with three possible receiver ports

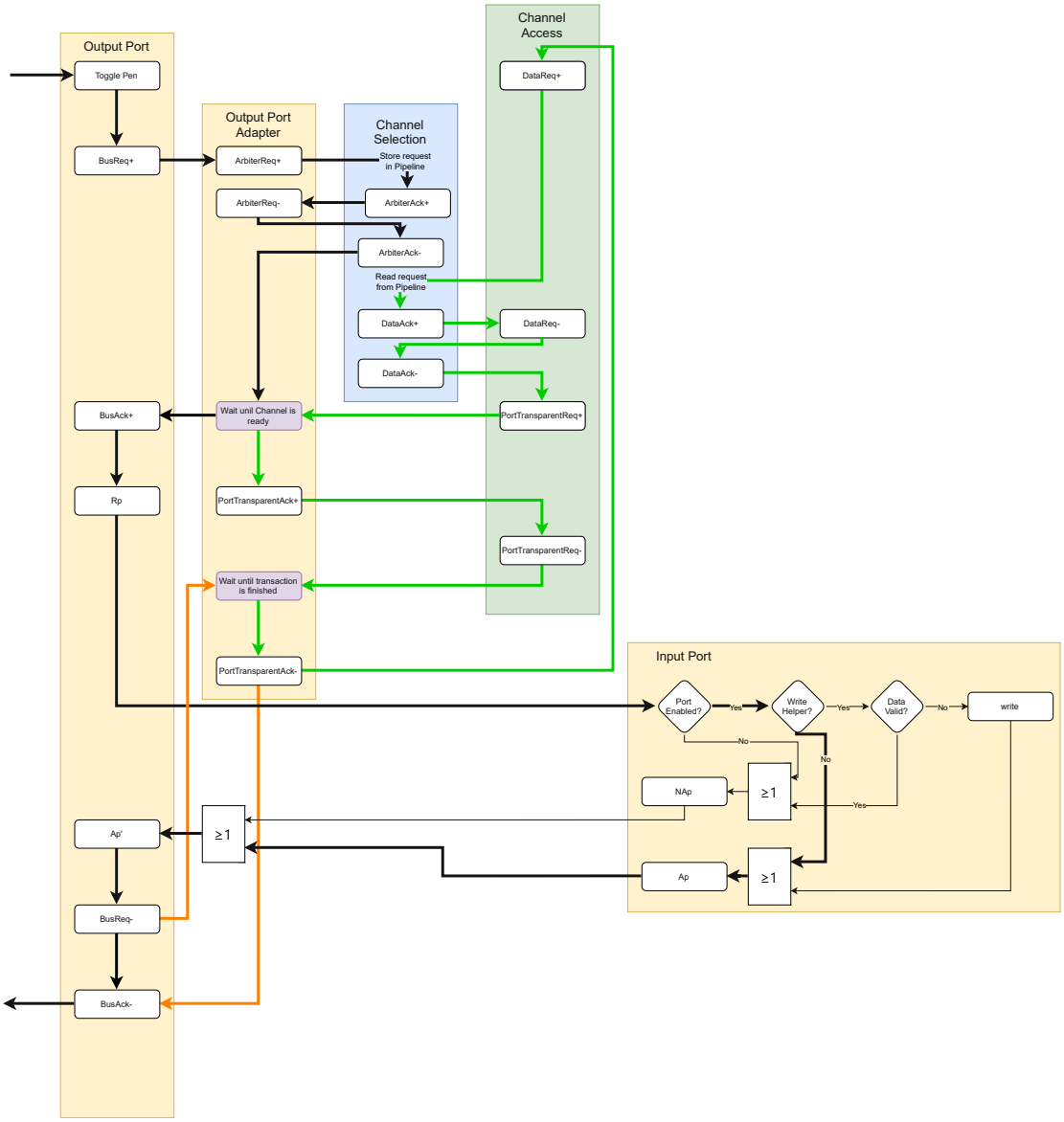


Figure 3.21: Flow chart of data transaction

CHAPTER 4

Fault tolerant bus architecture for TMR usage

Reliability is an important aspect in many modern applications, as a lot of systems nowadays need to be able to continue working properly after an occurrence of a fault as otherwise in the worst case human lifes are in danger. A way to make a system more reliable and tolerant against faults is by replicating it multiple times and then vote in order to get a correct result. In case two replicas are used, faults can only be detected, but not corrected. To correct one fault, three replicas are necessary. Such systems are called triple modular redundant (TMR) systems [LV62].

In this chapter a novel TMR capable bus architecture is presented that allows every node's replica to run with its own independent clock, as long as the phase shift between the different replicas is within a predefined range. Thus it might be necessary to synchronize the different replicas from time to time, which can be done without extra effort as it is automatically done during a TMR transaction. The fault tolerant bus uses the multi channel bus presented in chapter 3 as basis and allows to send TMR messages as well as best effort messages.

4.1 Mixed criticality system

As already outlined the bus system shall be able to transmit TMR messages as well as uncritical messages. However the critical messages should be prioritized. To achieve this, a flag is introduced that indicates whether a message is a TMR message or not. In case it is a TMR message it is arbitrated differently and prioritized higher than best effort messages. To avoid failure propagation over different replicas, data transactions between replicas always take place over voters. Consequently best effort messages can only be sent within the same replica. Multicasts are not only possible for best effort messages

but also TMR messages, as it is assumed that only one fault can occur. This means that when one full replica is not working properly, this is only counted as one fault as one failure might be propagated through the whole replica.

4.1.1 Data package

Due to the given requirements it is necessary to send additional information with the data to the receiver. Due to the requirement that the overall system allows both best effort messages within the replica and TMR messages that are propagated to all replicas, the message must be marked with a flag that indicates the message type. This allows the receiver to handle the message accordingly.

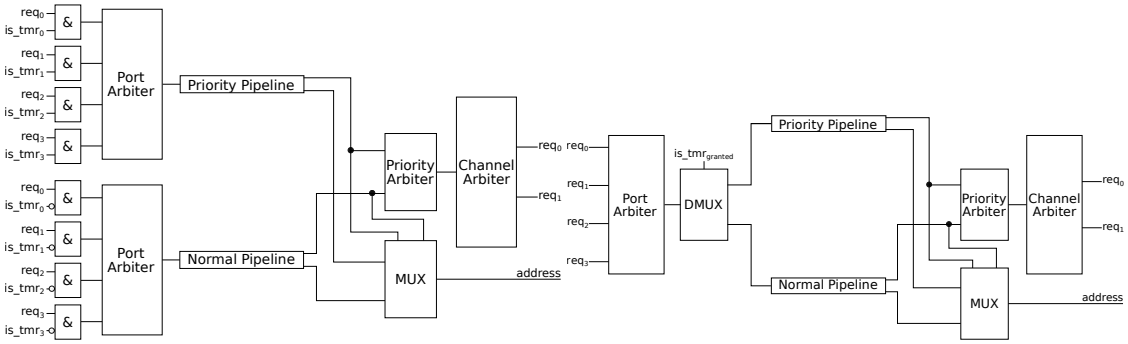
The second additional information that is added, is the sender's address, that is also very important for the receiver in order to know how to process the data, but also to store it in the correct table entry. Depending on the implementation, this also means, that the sender's address is one-hot encoded in the data package.

Figure 4.1: TMR data package

4.1.2 Message prioritization

As TMR messages tend to be more critical than other messages and to not block the bus unnecessarily long, because the requests might be arbitrated differently in each replica, two pipelines are introduced. One for TMR messages and one for normal messages. Two different approaches to prioritize TMR messages are shown in Figure 4.2. The problem with approach 4.2a, where only one arbiter is used, is that it might happen that a normal request is blocking the arbiter, because the normal pipeline is already full. So additional normal messages with no prioritization need to be sent before the TMR message can be forwarded to the TMR pipeline. To avoid this, a more area consuming circuit is presented in the second approach presented in Figure 4.2b, where two different arbiters are used, and depending on whether it is a TMR message or not the request signal is forwarded to the corresponding arbiter. When now a channel is idling and requests an address, a priority arbiter decides from which pipeline the address is taken. The grant signals of the priority arbiter are then used as select signals for the multiplexer, that then multiplexes the address that is forwarded to the channel helper module. The communication of the pipelines with the priority arbiter works like in the single pipeline implementation (Figure 3.4).

A possible priority arbiter for this application is presented in Figure 4.3. The prioritization in this approach is achieved through masking the request from the pipelines with the request from the channel. For the request with normal priority, the channel request req_c is delayed long enough such that req'_p gets the grant, when both pipelines have stored



(a) separate arbiters (b) shared arbiter

Figure 4.2: Priority pipeline

requests. Once req'_p or req'_n is active, it holds the state until the request of the associated pipeline is deasserted. Since the internal mutex element of the arbiter is in charge of deciding which of the pipelines will get the grant to request the source, these mutex grants m_grant_p and m_grant_n can be used as select signals for the mux that forwards the address from the selected pipeline. The pipeline-arbiter connector module in this approach is located right after the arbiter for the pipeline.

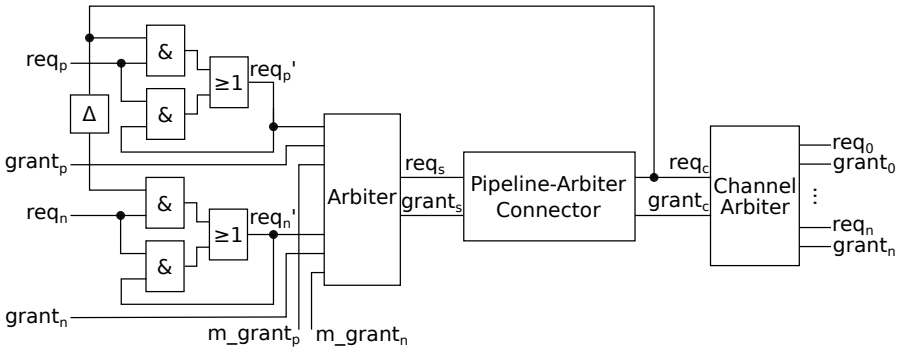


Figure 4.3: Priority arbitration

4.2 TMR system

In a triple modular redundant (TMR) system there are three replicas of a module that calculate the same and the result is then voted in order to eliminate one single failure. For a GALS system this means that there are three replicas of the whole system. A TMR variant of the system in Figure 3.1 is presented in Figure 4.4. All replicas have their own multi channel bus and important on the input side is now, that each input port must be capable of receiving data from all possible channels from all replicas. So in this case where the bus consists of two channels, each input port must be able to receive data from 6 unique channels.

The system is able to cope with one fault. It might be the case that a module of one replica is faulty. When being faulty, a module can request channels arbitrarily or pause the clock infinitely long, which then can lead to other modules becoming faulty too. Therefore it is important, that communication among different replicas only follows voting mechanisms.

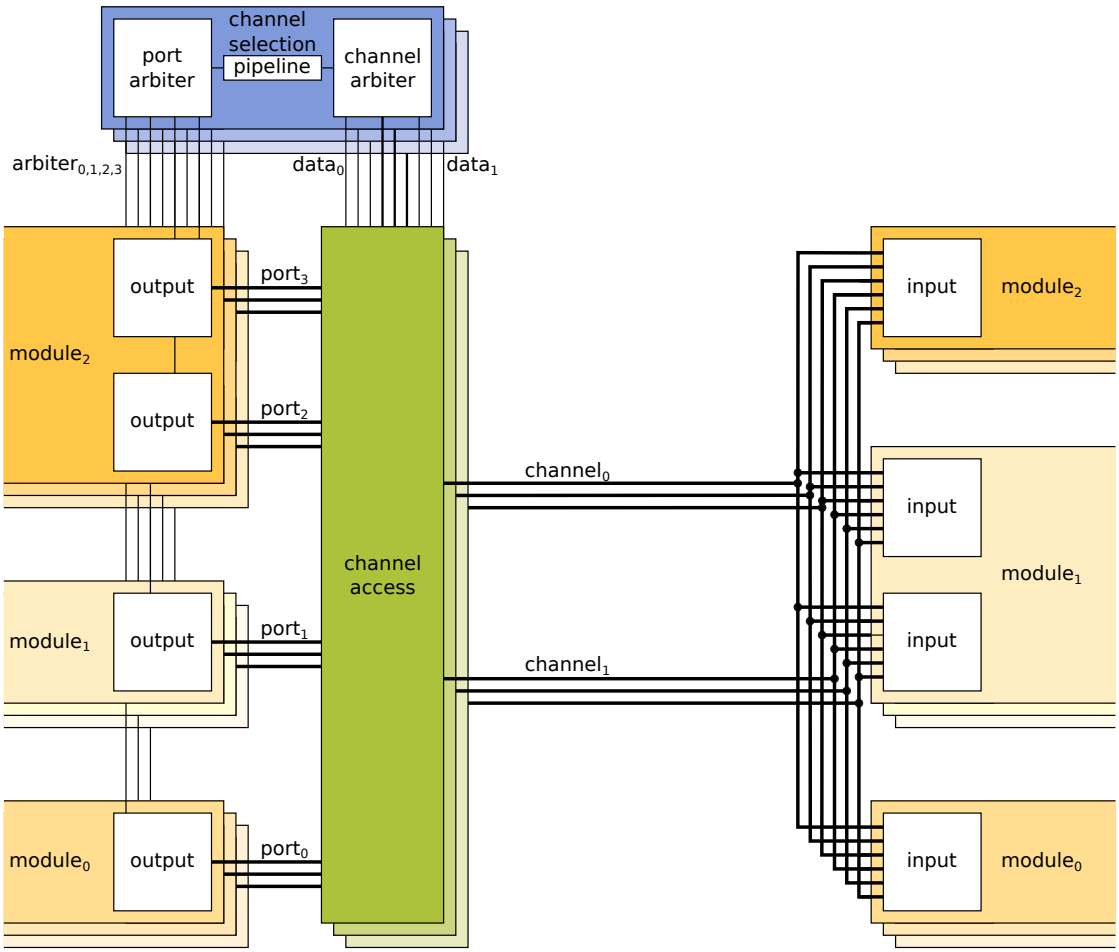


Figure 4.4: Schematic of GALS TMR

4.2.1 Clock

A very specific property of a GALS system is, that each node runs with its own self generated clock. This has to be taken care of, when receiving data and voting it. Furthermore also synchronization methods have to be considered, as it is important, that the slowest replica does not fall too much behind and therefore becomes faulty and introduces errors into the system. To overcome this, TMR messages are also used for synchronization purposes.This is done by halting the clocks of the involved modules until a majority of the replicas agree on a successful transaction.

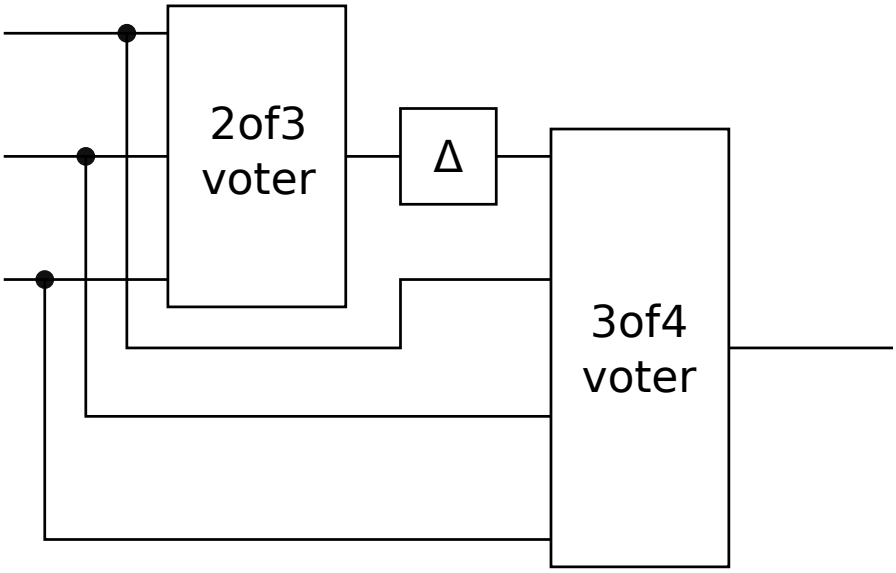


Figure 4.5: 3of4 voter for synchronization

4.2.2 3of4 voter

Since all nodes run with their own clock, it is challenging to tell whether a message is just late or will never arrive. Thus a 3of4 voting mechanism like the one presented in Figure 4.5 is necessary. After two replicas agree on a value it is waited for time Δ which is the maximal phase shift that the slowest replica is allowed to be behind the other ones in a failure free environment. If no value is received, it is assumed that the slowest one is dead and the 3of4 voter indicates that the voting process is complete.

However the problem of this approach is that the time between single messages needs to be at least Δ as otherwise glitches in the voted signal might occur. Therefore Lechner presented a special circuit in [Lec14], that replaces the delay line with a watchdog timer. When this timer runs in a timeout, the Muller gate is fed with the the signals from the level based majority voter (LB voter). Important to note here is that this special voting mechanism is only used for the request signals. The data is still voted with a level based majority voter as in conventional TMR approaches.

The f_{en} block is in charge of enabling the watchdog. This happens according to the truth table presented in Table 4.1. Important is here also to enable the watchdog timer when one request signal is high while all others are low and the output of the Muller element is also high. This might be an early transition and it might be necessary to clear the Muller element.

Lechner also presented a possible solution for the watchdog module in his thesis [Lec14]. A schematic of it is shown in Figure 4.7. After being enabled a ring oscillator is used as clock for a linear feedback shift register (LFSR). after reaching a predefined value the timeout is triggered. This then deactivates the ring oscillator and the LB voter value is

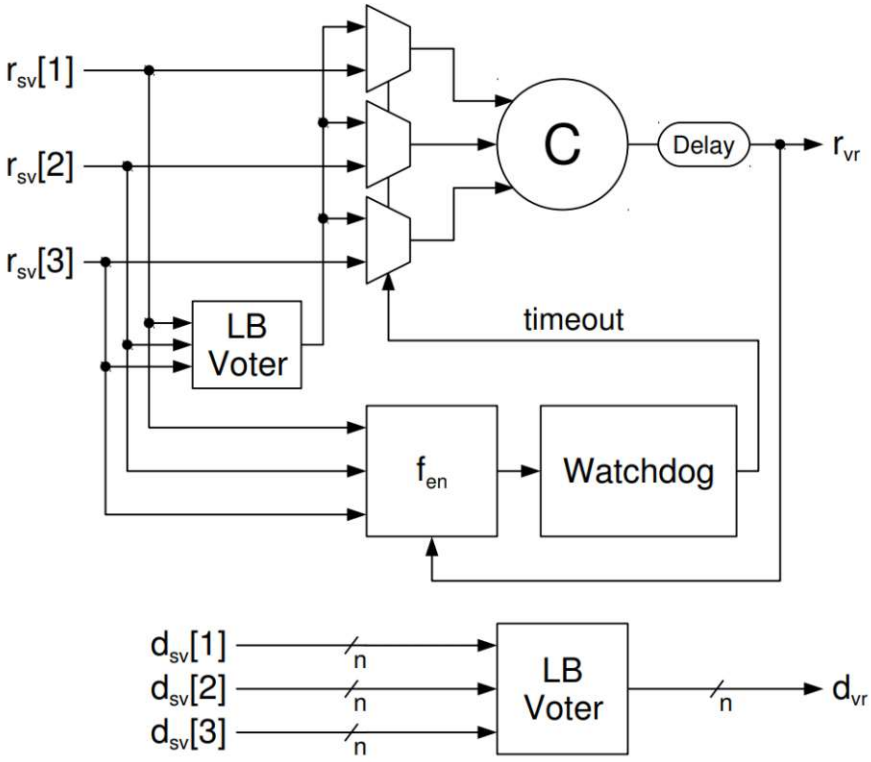


Figure 4.6: GALS TMR voter designed by Lechner (from [Lec14])

$r_{sv}[1]$	$r_{sv}[2]$	$r_{sv}[3]$	r_{vr}	f_{en}
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Table 4.1: Watchdog enable

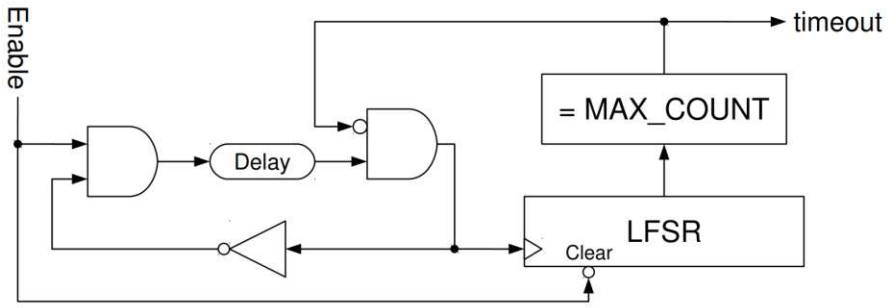


Figure 4.7: Watchdog module designed by Lechner (from [Lec14])

forwarded to the Muller gate inputs. In case the watchdog is disabled before the timeout, the LFSR module is cleared and the ring oscillator is halted.

4.2.3 TMR transaction

The time a port occupies a channel should be kept to a minimum and furthermore it is important to free the channel when a receiver is not yet ready for the data because otherwise the system might deadlock due to data dependencies on the algorithm abstraction level.

A very important task that TMR messages fulfill, is keeping the replicas synchronous to each other. This is quite challenging when the different replicas involved in a transaction, have to release the bus immediately in case the receiver is not ready and then request the bus again. Therefore an approach is presented, where TMR messages block the bus until the voter sends an acknowledge, indicating that the data was taken and the replica has agreed on a value. This approach requires that there are at least as many channels as concurrent TMR messages.

Figure 4.8 and 4.9 show the idea behind the synchronization. In the blue area each replica works on its own. When a replica now reaches a point where a TMR message is sent, it waits until a 3of4 voter (brown voter on the sender side) agrees on sending the TMR message (orange area). This is to reduce the amount of time spent on the bus and in case a node is faulty the bus is not taken unnecessarily.
When the voter allows sending the TMR message the bus is requested. This might take different amounts of time, depending on how the serialization method looks like and what other sender ports are doing and if one of them is requesting the channel selection simultaneously.
After a channel is assigned to the sender port the actual transaction takes place. On the receiver side a 3of4 voter (green voter) waits for the requests of the different replicas. After voting successfully an acknowledge is forwarded to all senders. Those senders also wait for those acknowledge signals with a 3of4 voter (green voter on the sender side). This voter also is in charge of the second synchronization phase, as it waits until it gets

a response from all three senders or runs in a timeout. After the voter approves that the message was sent successfully the channel is finally freed.

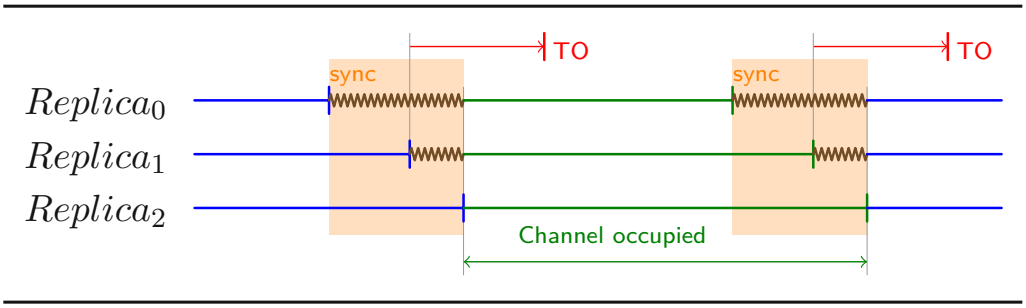


Figure 4.8: Timing diagram of TMR transaction

4.2.4 Sender side

When synchronization is important, only demand output ports make sense for the output side. For the sake of completeness a poll output port is also presented in the following section.

TMR supporting demand-type output port implementation

The heart of the TMR supporting demand-type output port is the demand-type output port for the MOGLI bus architecture presented by Villiger.

In order to not unnecessarily block a channel although no other replica wants to send the same TMR message, it is first waited for all replicas to be ready to send the message by a 3of4 voter. This is done by checking if the data that is sent is a TMR message or not. After the voter indicates that the others are now ready to transmit data too, the bus_{req} is sent to the output port adapter and after getting the bus_{ack}, the data is asserted to the bus until an acknowledge Ap is received.

After two replicas send an acknowledge signal for all requests, it is waited again via a 3of4 voter for the third one. When nothing is received it is assumed that the missing replica is dead.

In order to keep the phase shift between the replicas at a minimum, it is possible to introduce another synchronization stage where the Ap signals from the 3of4 voter of all replicas are again fed into a 3of4 voter and the result is then used to inform the output port that the transaction can be completed. Since in this approach the bus is kept until the data is really received, it is not really necessary to sync again as the phase shift is already below one clock period after the TMR transaction.

The red marked parts in 4.10 are optional and ensure that temporarily late modules do not become faulty immediately but instead get a chance to make up time by not even requesting the bus and sending the data. The diff module that is shown in detail in

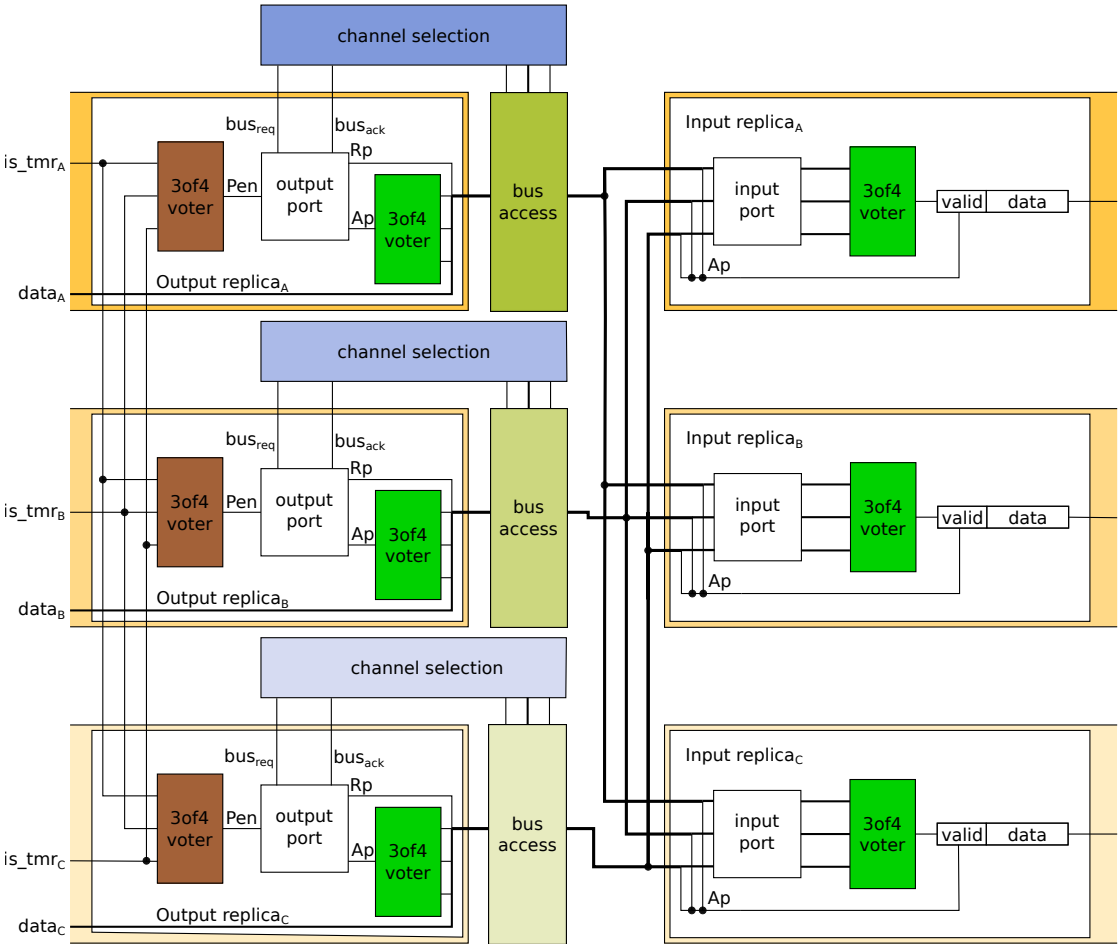


Figure 4.9: Voting mechanisms during a TMR transaction

Figure 4.10 is in charge of enabling the switches that then redirect the bus_{req} and Rp signal directly back to the bus_{ack} and Ap signal. For the diff module two counter modules from the DARTS implementation [FS11] are used. Only the GR^o is needed to determine whether the other replicas already finished sending or not. The depth of the remote pipelines defines how many TMR transactions the slowest module is allowed to be behind. For one message two Muller gates are needed. In order to be able to separate two TMR messages internally a *is_tmr_active* flag is used that is enabled when the output port is activated and the *is_tmr* flag from the message is active. It is then deactivated after the port is inactive again. This is the case after Ap and Ri both are again zero. When both the local and remote pipeline have a matching edge it is removed from both pipelines. As the local replica always has to wait until at least a second one is ready to send, the local pipeline depth can be kept very minimal. One Muller Gate is here enough. The problem with this approach however is, that the slowest port might be activated while the other two ports already received their acknowledge signals and are already releasing

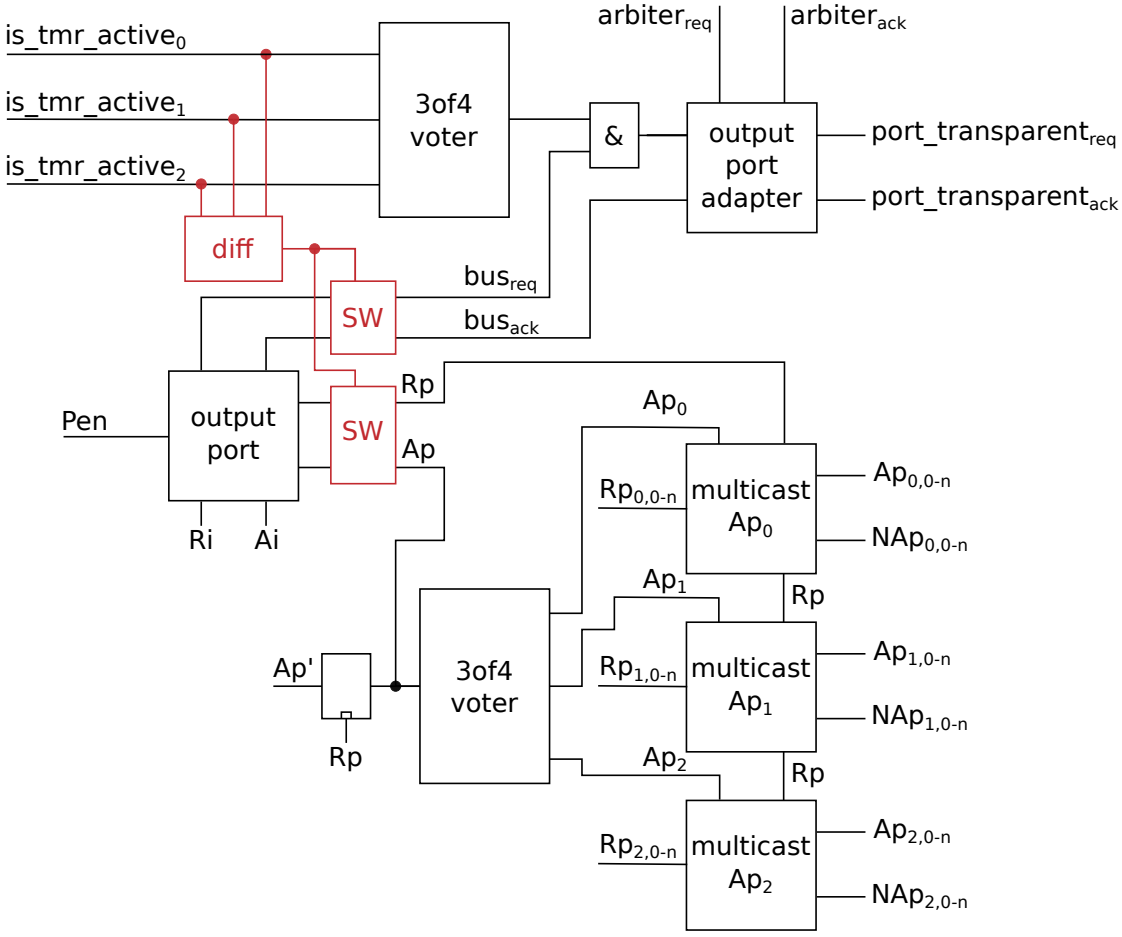


Figure 4.10: Block diagram of a tmr sender port with optional transient failure suppression

the bus. Also the module that just received the data resumes and might already prepare itself for the next data sent from the same port. In that case it might happen that one table entry is filled with wrong data. However the transaction is aborted immediately after the *is_tmr_active* is deactivated by the other two replicas.

TMR supporting poll-type output port implementation

In an application where synchronization is important, a poll port as sender is not really practicable. For sake of completeness and because the only difference is the acknowledge handling, it is presented here.

Unlike in the demand port the acknowledge signal is forwarded to the poll port when the first replica acknowledge all requests. The idea behind this is, to eliminate the phase shift that exists on the receiver side also on the sender side. After the 3of4 voter that votes over the acknowledge signals confirms that the transaction was successful the bus

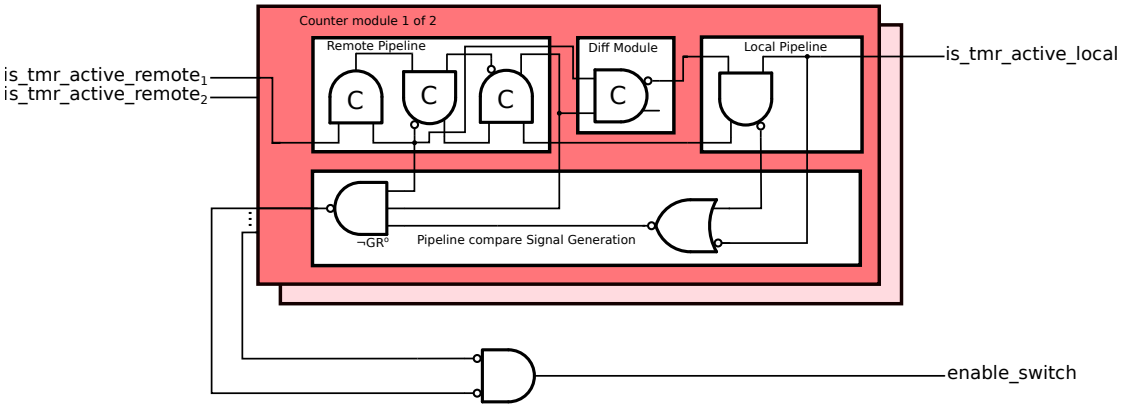


Figure 4.11: Diff module for enabling switches

is freed and the clock is enabled.

4.2.5 Receiver side

On the receiver side again two different scenarios are considered. The one is when only one sender can send data to a receiver port and the other one is that multiple ports can send data to one single receiver port.

One possible sender

For synchronization purposes it again makes sense, to use demand type input ports. Two possible input ports are presented in the following section. One input port uses one demand type input port per replica, whereas the second one only uses a single demand type input port.

Seperate demand ports Figure 4.12 shows a block diagram of a demand type input port for TMR messages, where data from only one sender can be received. Like in the multi bus channel, the channel that is active, is wired to the demand input port. Unlike in the multi channel bus approach here the Rp signal can not be directly forwarded to the port. The reason for this is that adapted demand ports are used, which can only be deactivated when no request has arrived yet. This functionality is necessary because one module might be faulty and requesting the port all the time leading it to block the module forever. Requests are only forwarded to the port, when at least two replicas are requesting the port and the pause request of the demand port already has been granted.

The data is latched when the currently stored data is not valid while Ap_i is active, indicating that the demand port accepted the request. After the 3of4 voter successfully voted and *valid* is active, the sender is informed, that the data has been consumed and the receiver port has valid data to process. At this point it is important, that the demand port is halted until *Rp* becomes inactive. However it might be the case that one *Rp* never

becomes inactive as the sender replica is faulty and thus the inverted request signals from the different sender replicas are fed into a 3of4 voter and the voted result is then used to halt the abortable demand input port as long as under normal circumstances necessary. Alternatives without this voter lead to possible states, where the receiver port already resumed its clock while still more than one *Rp* signal is active. This happens for example, when instead of the voted *Rp* signal the *valid* signal of the 3of4 voter is used that votes the latched data and valid signals. In case the simple 3of4 voter with just one delay line is used, the inputs and the output of the voter are inverted. This is necessary to keep the request still active when all ports except of one already deasserted the request.

The red buffer pipelines in Figure 4.12 are in order to keep a replica that was too slow once running and thus not let a transient failure become a permanent one.

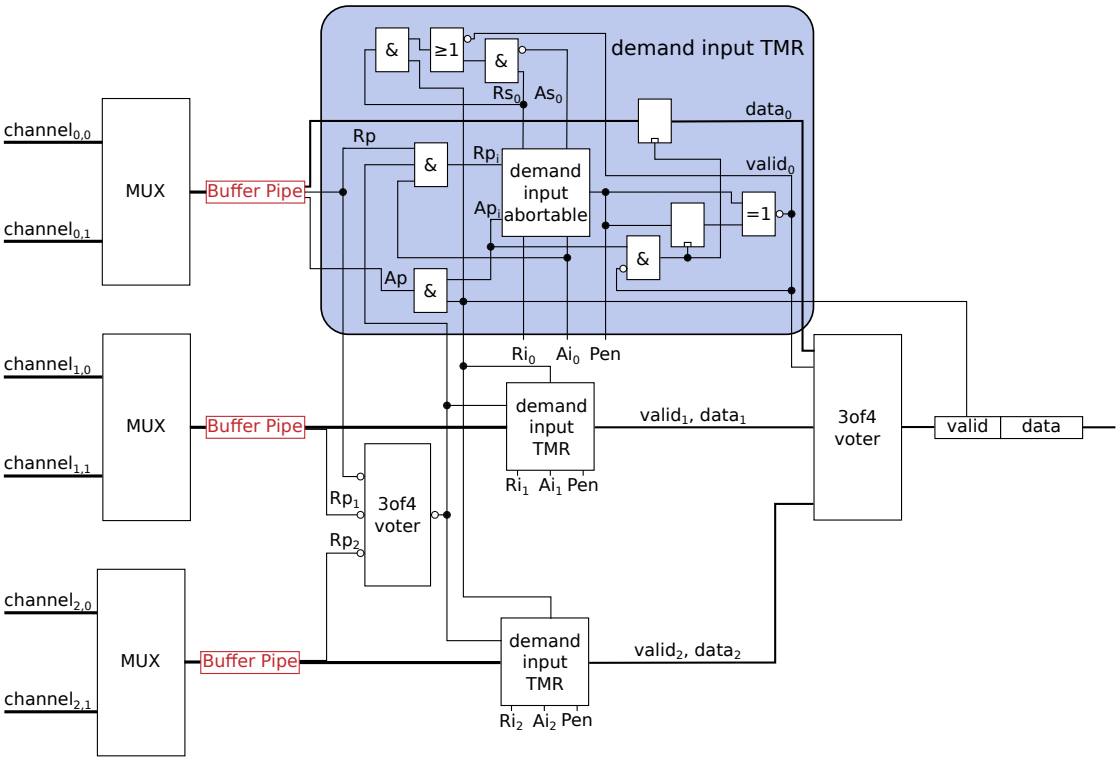


Figure 4.12: Separate demand type input ports for TMR messages with optional buffers for late transitions

Abortable demand type input port In case an input port uses one demand port per channel to receive data, the demand ports that received nothing need to be deactivated in order to resume the clock of the module.

An STG of a demand type input port that allows to disable the port although nothing has been received yet, is shown in Figure 4.13. The port can only be disabled by setting

Rs, when the clock is halted but no request has been received yet. After getting the grant signal As, the disable request can be deactivated again.

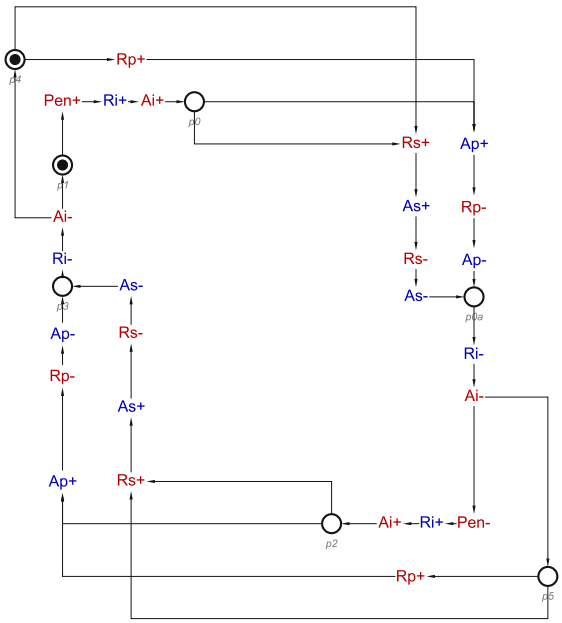


Figure 4.13: Demand type input port with abort functionality

The synthesis of the given STG leads to the Equations 4.1-4.4. Again an additional state variable $Z0$ is necessary to generate a proper circuit. After a successful transaction the port is disabled automatically until the enable signal Pen is toggled. Then the clock is immediately halted by the Ri request and only released, after a message has been received or the transaction has been aborted by the Rs request. An example of obtaining a message successfully followed by aborting the receive process during the second enable phase, is shown in Figure 4.14.

$$Ap = Rp \cdot Ai + Ap \cdot (Pen \cdot \overline{Z0} + \overline{Pen} \cdot Z0) \tag{4.1}$$
$$As = Rs \cdot Z0 + As \cdot \overline{Pen} \cdot Z0 \tag{4.2}$$
$$Ri = Pen \cdot \overline{Z0} + \overline{Pen} \cdot Z0 + As + Rs + Ap \tag{4.3}$$
$$Z0 = Pen \cdot Ap + Rs + Z0 \cdot (\overline{Ap} \cdot \overline{As} + Pen) \tag{4.4}$$

Buffer Pipeline As outlined above buffer pipelines can be added to keep a replica that missed one deadline running correctly. However the normal Muller pipeline has to be adapted slightly like shown in Figure 4.15. While Rp and Ap can be directly connected at the end of the pipeline, a *grant* that is going back to the sender module is only given, when the sender module is still active (indicated by a high *req* signal) and

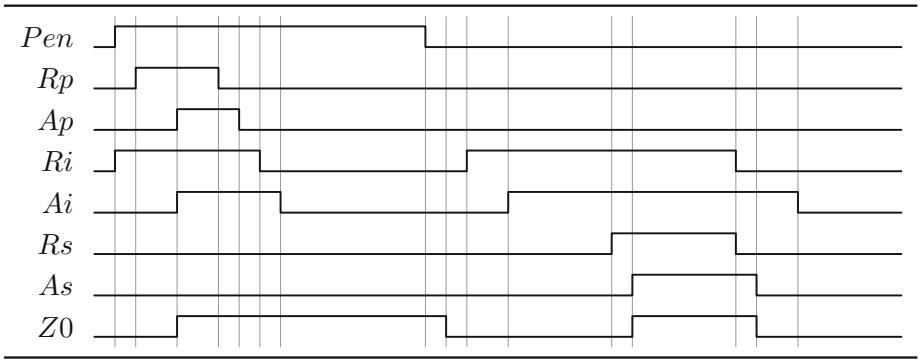


Figure 4.14: Timing diagram of the abortable demand type input port

that the current active request is the only one in the pipeline (guaranteed by checking if all internal request signals of the pipeline are also high). Otherwise the acknowledge *Ap* is referring to a previous transition. The writing into the pipeline can be done without acknowledge signal because due to the 3of4 voter there are no setup and hold violations, as the overall system waits until a timeout is met and the slowest replica is considered as dead.

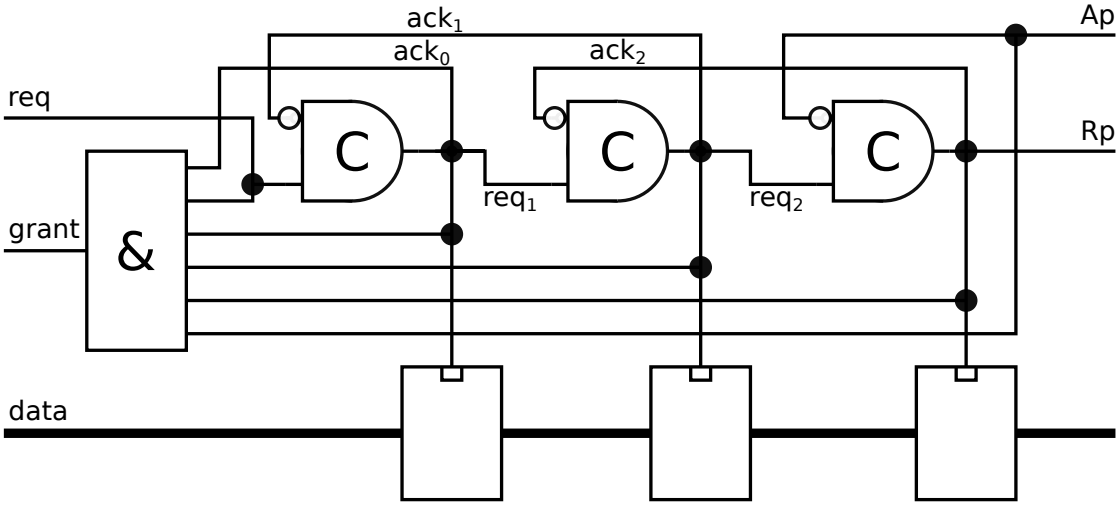


Figure 4.15: Buffer pipeline for inputs of late replicas

Shared demand port As the demand port immediately halts the whole module and only one single sender can send data to this port, it is possible to just use one single demand type input port. This not only makes it obsolete to take care of aborting ports that did not receive anything but also it is less area intense.

The request signals of the different replicas are fed into a 3of4 voter and this result is then forwarded to the *Rp* input of the demand port. In case the simple 3of4 voter with

just one delay line is used, the inputs and outputs of the voter are inverted. This is necessary to keep the request still active when all ports except of one already deasserted the request. The *Ap* signal of the port is then used as a condition, together with the *valid* signal, to make the latches transparent and thus store the data in the latches. Eventually the 3of4 voter agrees on a correct data word and the resulting valid signal is then used to acknowledge the successful receiving of the data to the sender.

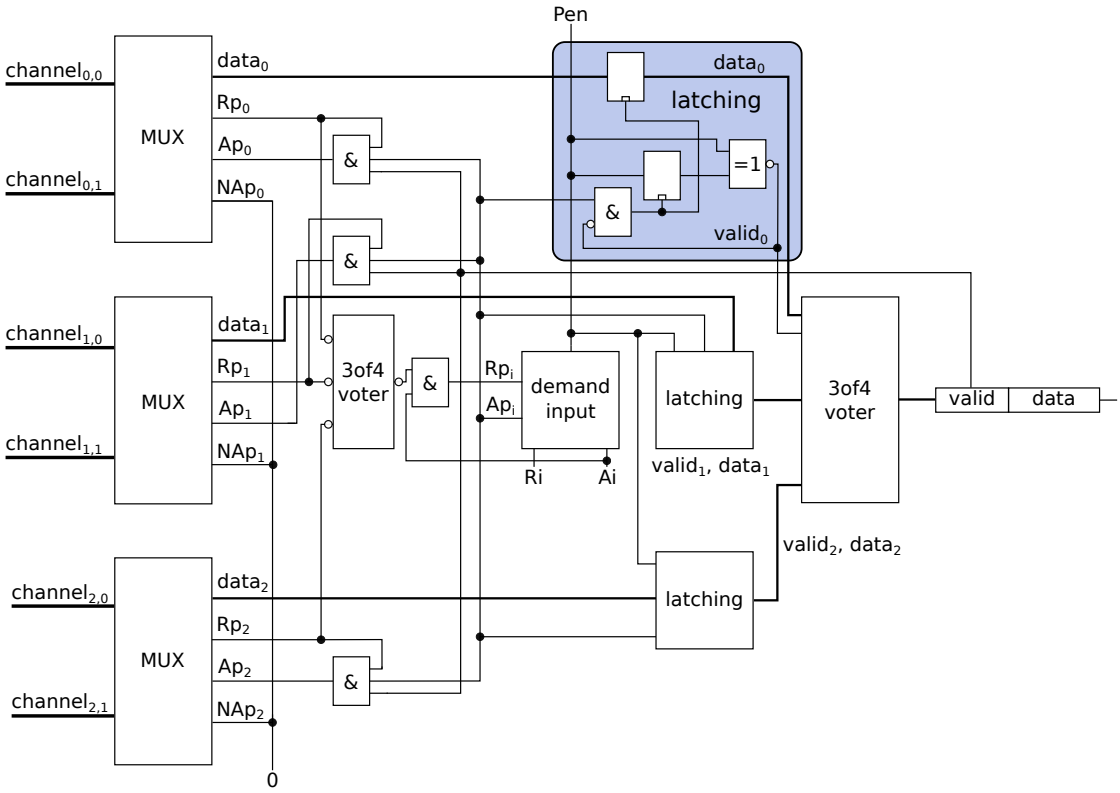


Figure 4.16: Single demand type input port for TMR messages without buffer pipelines

Multiple possible senders

In case multiple senders might address the same input port, an input port is not only needed for each replica but also for each possible channel. Since now multiple transactions per replica can occur concurrently, the single input ports can not be merged to one shared port per replica like in the single sender case above. Furthermore it needs to be known in advance which ports might send something, as for everyone an unique entry exists in a table like in Figure 4.17 where an input port is presented, that can obtain messages from three unique senders. For each possible sender one voter is introduced that votes on the three entries in the different tables. In order to not introduce additional delay elements to make sure that the address is for sure valid, it is one-hot encoded, like in the multi channel bus.

Poll input TMR For the poll input TMR port the poll type port from Villiger is used. This means that unlike the other ports, this one can receive data, as long as Pen is enabled. To not provoke a NAp_p from the poll port, the channels Rp is only forwarded, when Pen is enabled and to not receive a *nack* from the write helper, it is also checked if the currently stored data is not valid at the beginning of a transaction. If one of those conditions is not satisfied, the sender waits until they come true and blocks the channel for the whole time. To guarantee proper functioning, the valid signal is active coming out of the switch is active, when there is no correct address selected. Otherwise a invalid TMR message might block all replicas.

Write helper TMR The write helper from the multi channel bus approach onle needs to be slightly adapted. While the *nack* signal can be ignored, as it is avoided from the beginning, the *ack* signal is only set, when the sync module gives the grant for sending it. Important here is that here not the Pen signal can be used for identifying whether there is a valid data word stored or not. Thus an additional *store* signal must be introduced for each TMR table entry.

Buffer pipeline Depending on whether the poll input port is disabled at some points or not, the buffer needs to be before the poll input port, directly at the end of the channel, or included in the write helper TMR module. When the port is disabled, it needs to be before the input port but this is problematic because of the ordering, as when disabled too long, it might be the case that two messages to the same address were received at different channels and thus the ordering can not be guaranteed. However when the port is always enabled this can be easily solved by integrating the buffer pipeline into the write helper TMR module as shown in Figure 4.17.

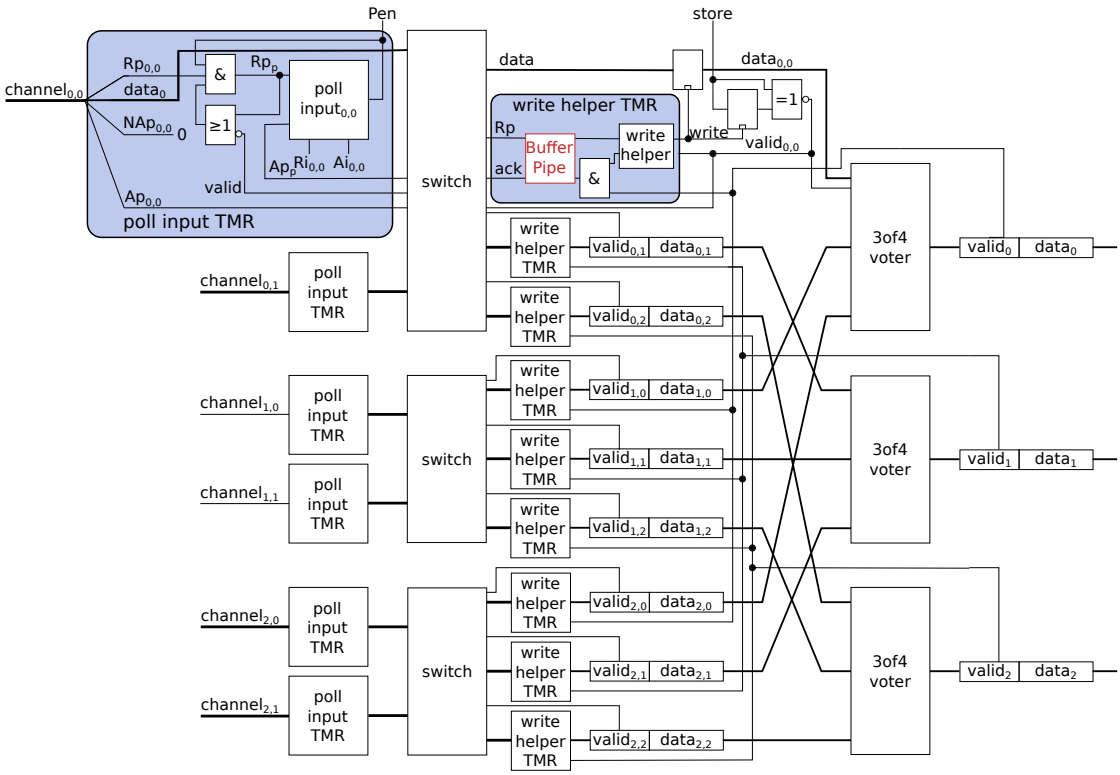


Figure 4.17: TMR receiver port with table and buffer pipeline within write helper TMR

CHAPTER 5

TMR fault handling

In the following the impacts of single faults in the newly designed TMR system presented in chapter 4 are analyzed. As already redundancy is provided, the focus in this analysis is the TMR bus communication, as faults within single replica modules can be masked away by voters. Thus a special focus is laid on signals that connect the different replicas with each other and the voting mechanisms. As visible in Figure 5.1, this holds for all channel signals, as each sender has request and acknowledge lines for all replicas, and also for the *is_tmr* flag, as those are also fed into each single replica. The affected signals are listed by component in Table 5.1 and for all other signals, it holds that they are just influencing the replica that they are located in. However in this analysis we will also have a closer look at some of these, as some faults there might have a huge impact on the replica and hence unnecessarily degrade MTBF through spare exhaustion.

An analysis like this could be done nicely through formal verification approaches like Model checking. However, modeling the proposed architecture on the relatively low abstraction level that is required for this type of analysis is a substantial effort that could not be accommodated within the scope of this thesis. It is therefore left for future work.

Component	**Signal**
Bus	Data Package (TMR, flag, Address, Data)
	Rp
	Ap (Nap is not involved in TMR messages)
Sender	*is_tmr* flag
	is_tmr voter
	Transaction complete voter
Receiver	Data voter
	Request voter (in case of demand type input)

Table 5.1: Components and affected signals

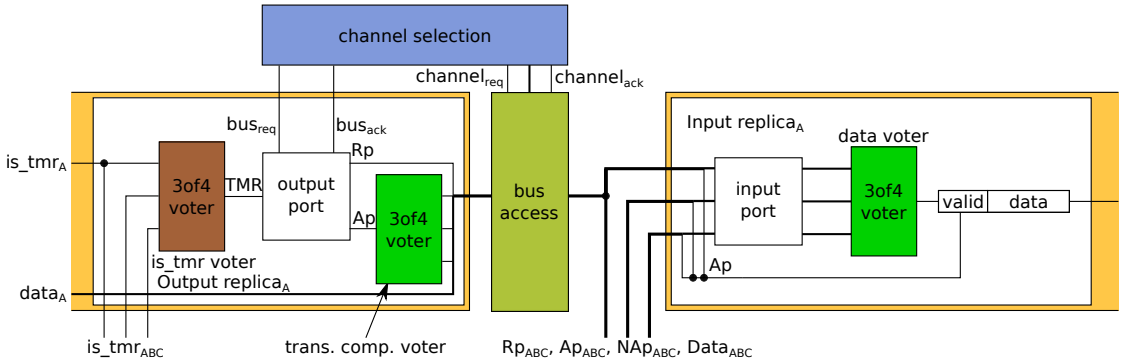


Figure 5.1: TMR replica with signals crossing replica border

5.1 Transient bit flips

The reason for transient bit flips are high energy particles like alpha particles, protons or neutrons that come from cosmic rays or packaging materials and strike sensitive p-n junctions of a transistor [ZCM+96, BSH75]. Such events that do not cause permanent damage to the device, but change memory or register contents, are called single event upsets or soft errors. Depending on where such a bit flip occurs it has different outcomes for the overall system. We assume in the following that only a single data item is corrupted per particle hit.

5.1.1 Faults on signals crossing the replica borders

In a first step those bit flip faults are analyzed in detail that can occur on signals involved in the communication with other replicas. The different faults and their effects are also listed in Table 5.2.

Data Although this might lead to a permanent failure of one replica when it happens in a best effort message, it does not harm a TMR message as there are two other replicas that provide the correct data in case of just one error.

TMR flag A bit flip in the TMR flag is already more critical as the following analysis shows: First two cases have to be distinguished here. One is where the flag flips from one to zero and the other one is where a best effort message suddenly becomes a TMR message.

TMR to best effort When a TMR message becomes a best effort one, at most one replica is fed with wrong data in case the best effort input port with that address exists and is furthermore ready to receive data. In case no port with such an address exists, the transaction is never completed and thus the channel is blocked forever if the bit flip does not resolve. The TMR input ports of the other two replicas wait until the

data voter timeout is triggered and then accept the data. In case the data is sent to a demand input port the malicious replica releases the request only after the request voter on the receiver input also runs into a timeout. Afterwards the system continues operating without errors. However when the bit flip resolves fast enough, no problem at all occurs and no voter timeout is hit.
When there is a corresponding best effort input port with the same address, it might be fed with wrong data. But this only happens, when the port is ready to receive data and is part of the replica where the fault occurred. This is because input ports only accept best effort messages from the replica they are operating in.

Best effort to TMR When a best effort message becomes a TMR message again only one replica, namely the one where the bit flip was introduced, is affected and two cases have to be distinguished. The first case is when no corresponding input port exists. In that case neither an Ap nor a NAp is received to acknowledge the transaction. Thus the sender module of the affected replica is halted forever and the same holds for the channel that is occupied by that port. Furthermore the receiver module never receives data and thus is also not working properly. The second case is when there exists a best effort input port with the same address. Then the message can be delivered at some point. However this leads to a lasting error when pipelines are used to keep a too slow replica running, as the correct data is then always too late because the input was already fed with the data from the previous transaction. In case no such pipeline is used, there is only one transaction affected and thus the transient failure does not become lasting.

Address Here the effect of a bit flip highly depends on how the addressing is implemented. Furthermore only messages to poll type input ports are affected, as there is no address decoding for demand type input ports, as only one sender is allowed there.

One-hot encoded In case of a bit flip at the one hot encoding, the affected sender never gets an acknowledge, as either two bits are active and thus no table entry is selected or no bit at all is active which leads to the same result.

Enumeration In case all ports are just enumerated a single bit flip means that the data may be sent to a wrong table entry. In case this entry does not exist the sender module is again stuck forever in the worst case.

Hamming code Using encodings that allow to detect errors helps to get over the problem with single bit flips. However the question arises whether this helps to reduce the probability of failure or through the increased complexity it is even more likely that a failure is introduced into the system.

Request signals A bit flip here might lead to a request to the wrong input port. But since this request is only affecting one replica, it has no influence on the overall system. The same holds in case that one active bit is flipped to zero.

However it might be the case that all three request signals of one replica are wrong. This happens when there is some problem during the address conversion within the replica itself. But even in this case, this has no influence, as still each receiver replica has two correct request signals. Nevertheless this might lead to a replica that is stuck forever, when there is no corresponding input receiver or all request lines are zero.

Again here in case of optional buffer pipelines the wrong request is stored into the receiver pipeline and thus affecting the wrong input port even though it is not active at that time.

Acknowledge signals For the acknowledge signals especially bit flips to zero are problematic, as a bit flip has no direct influence as all replicas wait at least for a second acknowledge to continue.

However when an acknowledge signal flips to zero, the affected sender replica is holding the request until either the transaction complete voter runs into a timeout or, in case a demand type port is used, the request voter runs into a timeout. When all acknowledge signals of one receiver replica are bit flipped the result is quite similar. The only difference there is that all transaction complete voters run into a timeout.

is_tmr **flag** One single bit flip for the *is_tmr* flag has no effect on the overall system and not even on the replica itself as this fault is outvoted by the other two replicas. At most the *is_tmr* voter timeout is triggered and added to the transaction duration. However because time is needed to resolve the flag voting, it might happen that the voters on the receiver side need to wait a bit for the affected replica to deliver the data.

is_tmr **voter** A bit flip of the *is_tmr* voter only has an influence, if the output port is enabled. In case the voter is active a transaction might be initiated although no other replica is ready to do so either and thus the output port is holding the channel until a second replica is ready and all voter timeouts are reached. In case the voter should be indicating a TMR message due to its inputs and it is not active, two possibilities exist. When the flip is short enough the transaction happens as usual while for the other case, where the flip only resolves after the two other replicas already completed their transaction, the affected replica is from then on one transaction behind if unless a mechanism for too slow messages is built in. The overall transaction duration is then increased, depending on the *is_tmr* voter and the data voter timeout.

Transaction complete voter The worst case that can happen here is shown in Figure 5.2: The output port completes the transaction too early, before any replica was able to consume it. However this is no problem for the overall system, as there are two other working replicas. The transaction duration is in this case increased by the data voter timeout, as the faulty sender already assumed that the transaction is complete.

When a flip from one to zero occurs the affected sender port can never complete its transaction and is thus blocked and blocking the channel it got assigned to. However,

the transaction duration for the other replicas is only increased in case the receiver port was a demand type input port, as then the request voter runs into a timeout.

Data voter Bit flips here only have influence on one replica and even in case the valid signal is affected by the bit flip, only one false acknowledge signal is sent to the replicas and each replica has two correct ones to vote on. So this is no problem and may only lead to wrong data on the replica where the glitch occurred, as it might not have saved the data correctly yet, while the other two replicas might already have stored the data.

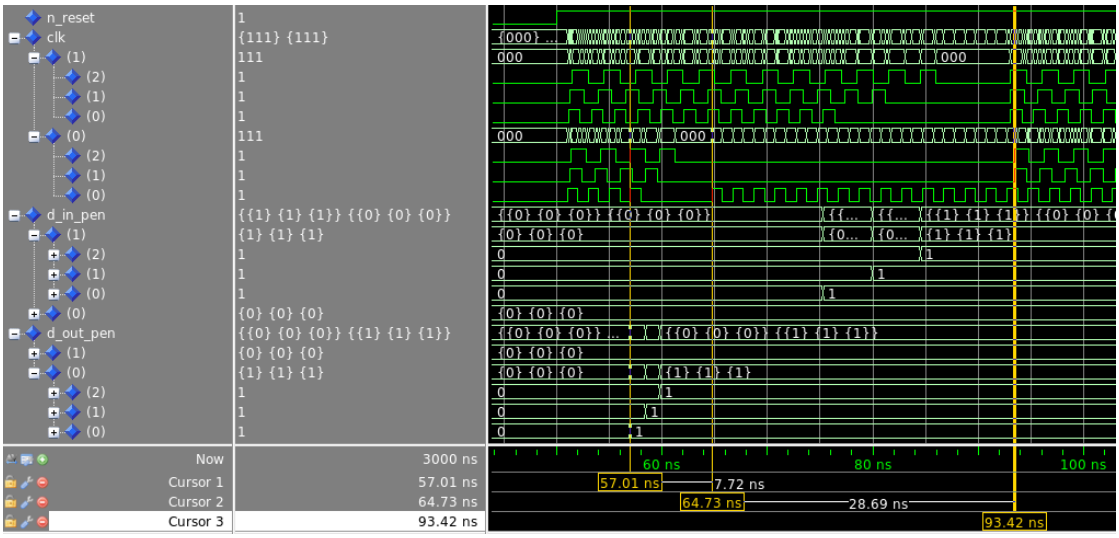


Figure 5.2: Bit flip in transaction complete voter

Request voter A bit flip for the request voter only has consequences, when the demand type input port is ready to process a request signal, which is the case when the port already halted the clock. When now a bit flip on the request voter occurs in that situation it can lead to the problem, that the demand type port may already finish the transaction process although no data was yet stored. Thus the affected (single) replica may then continue working with invalid data and the transaction duration is increased by the data and transaction complete timeouts.

5.1.2 Faults on signals that stay within one replica

In the following a few interesting bit flips and glitches of signals are analyzed that stay within one replica but where a fault might have a huge impact of the overall system and may lead to a erroneous replica and thus to a degraded mode of the overall TMR system. Table 5.3 gives a short overview of the faults and their effects.

Pipeline A transient bit flip glitch in the grant signals of the port arbiter that feeds the pipeline might introduce a wrong request address that is then propagated through

Fault	Blocked port	Blocked channel	Blocked bus	Local *is_tmr* voter timeout	Local request voter timeout	Local data voter timeout	Local trans. comp. voter timeout	Remote *is_tmr* voter timeout	Remote request voter timeout	Remote data voter timeout	Remote trans. comp. voter timeout	Wrong port	Pipelining problematic	Local data wrong
Data														
TMR flag zero, no corresp. input	✗	✗			✗	✗			✗	✗				
TMR flag zero, corresp. input					✗	✗			✗	✗		✗		✗
TMR flag one, no corresp. input	✗	✗												
TMR flag one, corresp. input												✗	✗	✗
Address (one-hot encoding)		✗				✗				✗				
Rp, zero, one line of one replica					✗		✗		✗					
Rp, zero, all lines of one replica	✗	✗				✗				✗				
Rp, one, one line of one replica					✗				✗			✗	✗	
Ap, zero, one line of one replica					✗		✗		✗					
Ap, zero, all lines of one replica					✗		✗		✗		✗			
Ap, one, one line of one replica														
is_tmr flag, zero				✗		✗		✗		✗				
is_tmr flag, one														
is_tmr voter, zero	✗					✗				✗				
is_tmr voter, one														
Trans. comp. voter, zero	✗	✗			✗				✗					
Trans. comp. voter, one						✗				✗				
Data voter, zero						✗				✗				✗
Data voter, one														✗
Request voter, zero							✗			✗	✗			✗
Request voter, one	✗													

Table 5.2: Bit flip effects

the whole pipeline and finally an output port is granted access to the bus that might not even has requested bus access. In that case the channel is kept until this port really wants to send something. However in that case already the new data word is stored in the pipeline and after the successful transaction a channel is again blocked by this port.

It might also happen that within the pipeline bit flips occur. When the one bit that is active flips, the pipeline continues functioning but the request is lost and the affected port will never get a grant.
In case a second bit becomes active, there might be two active bits in the worst case that then allows two output port to access the same channel. However it also might be the case that the wrong active bit is way faster in the pipeline and thus the only one that is propagated from some point on, as the acknowledge is already triggered. This then leads to a port, that never gets a grant to access a channel. While another one blocks one channel like before.

When a bit flip of the request signal on the output side of the pipeline is introduced, an output address might be read, without being propagated to a channel. In that case, the affected output port never gets assigned to a channel and waits forever to get the grant to access the bus.

Channel grant In case a bit flip glitch from zero to one happens here to a grant line where the request line has just become active but not given the grant to request the source yet, one output port request that is the next to be served, is propagated to two channels. The one that gets it through the regular way and the other one that got it because of the bit flip. Given that the channel that gets assigned the output port correctly receives the grant simultaneously, two channels try to send the same data to the same receiver. As both channels are then freed again at the same time, this fault stays transient. However when the correct grant is not given promptly, the output port has already completed its transaction and thus the channel that was actually assigned is blocked.

Port transparent request Since the *port_transparent_ack* signal stays active until the transaction is completed and because the output is not forwarded onto the bus when the address that is responsible for the switching is selecting another output port, no acknowledge is received and thus the transaction is only completed, when a channel really selects the port. Therefore this bit flip has no influence to the overall system.

Output port adapter arbiter request grant Here the problem occurs when one output port adapter that is already requesting the arbiter gets a grant, that the data has already been stored in the pipeline. As this does actually not hold, the data in this case is never sent, as the port never gets access to a bus channel.

Fault	Blocked port	Blocked channel	Blocked bus	Additional wrong message	Message request lost	Two senders on one channel
Port arbiter grant, glitch		✗				
Pipeline address signal, zero	✗				✗	
Pipeline address signal, one		✗		✗	✗	✗
Pipeline out req signal, glitch	✗			✗		
Channel grant, glitch		✗		✗		
Port transparent, glitch						
Output port adapter arbiter request grant, glitch	✗					

Table 5.3: Interesting bit flip and glitch faults, beside the replica border crossing signals

5.2 Replica too slow

A solution for this problem was already presented in the previous chapter and as seen above although the pipelining solution helps to prevent a replica that is temporarily too slow to become considered faulty, it may be the cause for other transient failures to become permanent until the next reset. Thus the trade off has to be analyzed for the specific application, to decide which property is more important.

5.3 Simultaneous write

Although the system has to be designed in a way such that there are no simultaneous writes possible to the same address, these might nevertheless happen in a failure case. When two output ports want to write to the same address of a demand input port however, the multiplexer can only let one through so here despite two simultaneous requests, only one is granted to write. For poll type input ports the situation is a bit different, as there it depends on how the switches are implemented. In case that they are implemented as the one shown in Figure 2.21 which was designed by Villiger also no problem occurs, as there the MUTEX elements also take care that only one element at a time can be written.

5.4 Permanent faults

The errors analyzed before were all soft errors. However errors can also be permanent. Those also need to be tolerated by the system. As the main interest is the new designed mixed criticality bus, only faults are analyzed that have an influence on the communication. When one module has internal errors and does not request the bus anymore there are two other replicas of this module that still work properly.

5.4.1 Faults on signals crossing the replica borders

In a first step again the stuck at faults that can occur on signals that are used for the communication with other replicas are analyzed in detail. All the affected signals are listed in Table 5.1 and the possible faults and their behavior is listed in Table 5.4. As the analysis shows, most stuck at faults lead to the same effects as the bit flip faults in their worst case. The only difference then is, that through a reset, a bit flip fault can be resolved while this does not hold for a stuck at fault.

Data package single bit stuck at faults Stuck at faults in the data package have mainly the same influence as single bit flips. Thus the analysis from above also holds for permanent faults and only affects those input ports that are selected by a request signal of the channel that has the faulty wire. The only difference is that the fault may not resolve during the ongoing transaction, as it might be the case for the bit flip faults.

Output port request signal stuck at zero When one of the request signals is stuck at zero, it has no influence at all until one sender is trying to activate it. In that case one input port receives only data from two replicas and thus the data voter has to run in a timeout to complete the transaction on the receiver side. As Figure 5.3shows, also the transaction complete voter runs in a timeout on the replica that introduced the fault. As the receiver port is a demand type input port, the request voters on the replicas that received a request from the malicious replica, also runs into a timeout.

Output port request signal stuck at one As already outlined in the previous chapter, it is important that a receiver can continue working also in case that one request signal flips to one. This also holds for signals that are stuck at one. Thus a request is only forwarded to the receiver port when this port is ready to receive something, as otherwise the input port might be blocked by a faulty request signal. However here it might happen that a receiver that is ready, takes the data although it is not intended for it, but has a corresponding table entry address and thus is able to store the data there. Although the request is stuck at one, data is stored whenever the requested input is ready to receive data and this data is only correct, when the sender is indeed requesting the receiver.

Input acknowledge stuck at zero A receiver that is not responding is a huge problem for a best effort message as this means that the replica might not work correctly and

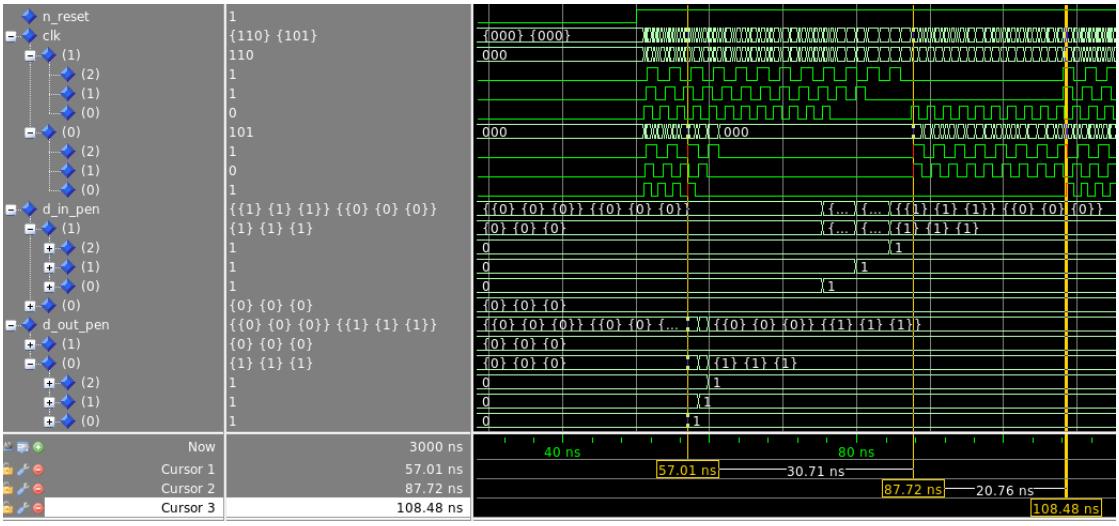


Figure 5.3: Stuck at zero of a request signal

that this error will slowly propagate over that replica. However it has no influence on TMR messages, as here a voter checks all acknowledgment signals and after receiving acknowledgments from two replicas it is waited until the voter reaches its timeout and the transaction is completed and thus only the receiver module of one replica is erroneous. However, all transactions will run into the timeout, hence performance will be worst case.

In case only one acknowledge signal is stuck at zero, only one transaction complete voter runs into a timeout and is then behind the other two replicas.

Input acknowledge signal stuck at one The acknowledge can not only be stuck at zero but also stuck at one. However this has no big impact either as the sender does release everything after the transaction complete voter acknowledges that the input ports have received the message correctly. So in case one acknowledge signal is stuck at one, this has no influence and all channels are free for the next transaction. Here it does not matter if only one acknowledge signal is stuck or all acknowledge signals of one replica.

is_tmr **flag stuck at fault** A stuck at fault of one of the *is_tmr* flags has no effect on the overall system and not even on the replica itself as the 3of4 voter can mask this fault away.In case of a stuck at one fault, the transaction duration is not even influenced whereas in case of a stuck at zero fault it is increased at least by the timeout voter.

is_tmr **voter stuck at zero** When the voter is stuck at zero, no TMR message can be sent. In this case the output port is blocked and this might influence the complete replica but should be no problem for the overall system.

is_tmr **voter stuck at one** In case one *is_tmr* voter is stuck at one the output port immediately requests the bus and thus the synchronization purpose of this voter is not given. As the voters are replicated through providing each node replica an own voter, only the node replica with the erroneous voter requests the bus earlier than the others and is then blocking one channel longer than the other two ports.

Transaction complete voter stuck at zero A stuck at zero fault of the transaction complete voter leads to the problem, that the output port never stops trying to send the data, as it is waiting for a response. Thus this will block one channel of a replica and may also lead to a broken replica, as the affected module can not continue working properly. In case the receiver is a demand type input port, the request voter timeouts on all replicas are reached as the broken replica continues requesting the port.

Transaction complete voter stuck at one On the other side a stuck at one fault of the transaction complete voter will lead the output port to stop the transaction immediately after it started, as it assumes that the receiver already got the data. Thus the receivers have to work with the data from the other two replicas. Although the data voter timeouts are reached, all replicas can continue and work properly.

Data voter stuck at fault This only has an influence on the replica itself, as this is then erroneous. No additional delay is added, when the voter is stuck at one. However when it is stuck at zero, the transaction complete voters run into a timeout as they only receive an acknowledge from two replicas.

Request voter stuck at fault Either stuck at fault is very problematic for the replica, as in both cases the pause request for the clock is never released and thus the module is blocked forever. For a stuck at zero fault this furthermore means that nothing can be received and thus the data voter never acknowledges correct data. This then leads to transaction complete timeouts on all replicas.

5.4.2 Faults on signals that stay within one replica

In the following a few interesting stuck at faults of signals that stay within one replica and their effects on the replica and overall system are analyzed. A overview of the faults and their effects is given in Table 5.5. It can be seen that most of these faults lead at least to a blocked channel and some even to a blocked bus.

Channel/Output port arbiter request stuck at one Failures that directly happen in the bus logic, have the biggest impact as they not only might lead to one blocked channel but to the whole bus being blocked. One case where this happens is when a request that is already granted by the arbiter is not released. This can happen on the output port arbiter as well as on the channel arbiter. In both cases this leads to the bus being blocked. Not even the current transaction can be finished, as shown in Figure

Fault	Blocked port	Blocked channel	Blocked bus	Local *is_tmr* voter timeout	Local request voter timeout	Local data voter timeout	Local trans. comp. voter timeout	Remote *is_tmr* voter timeout	Remote request voter timeout	Remote data voter timeout	Remote trans. comp. voter timeout	Wrong port	Pipelining problematic	Local data wrong
Data														
TMR flag zero, no corresp. input	✗	✗			✗	✗			✗	✗				
TMR flag zero, corresp. input					✗	✗			✗	✗		✗		✗
TMR flag one, no corresp. input	✗	✗												
TMR flag one, corresp. input												✗	✗	✗
Address (one-hot encoding)		✗				✗				✗				
Rp, zero, one line of one replica					✗		✗		✗					
Rp, zero, all lines of one replica	✗	✗				✗				✗				
Rp, one, one line of one replica					✗				✗			✗	✗	
Ap, zero, one line of one replica					✗		✗		✗					
Ap, zero, all lines of one replica					✗		✗		✗		✗			
Ap, one, one line of one replica														
is_tmr flag, zero				✗		✗		✗		✗				
is_tmr flag, one														
is_tmr voter, zero	✗					✗				✗				
is_tmr voter, one														
Trans. comp. voter, zero	✗	✗			✗				✗					
Trans. comp. voter, one						✗				✗				
Data voter, zero						✗				✗				✗
Data voter, one														✗
Request voter, zero	✗						✗				✗			
Request voter, one	✗													

Table 5.4: Stuck at fault effects

5.4. This is because the channel helper AFSM is stuck in this case and thus also the *port_transparent_req* is never triggered. The impact of this failure on the overall TMR system is however minimal, as this only blocks one replica and the other two work as before and only the TMR messages may take longer as now always the timeouts are reached in the *is_tmr*, data and transaction complete voter.

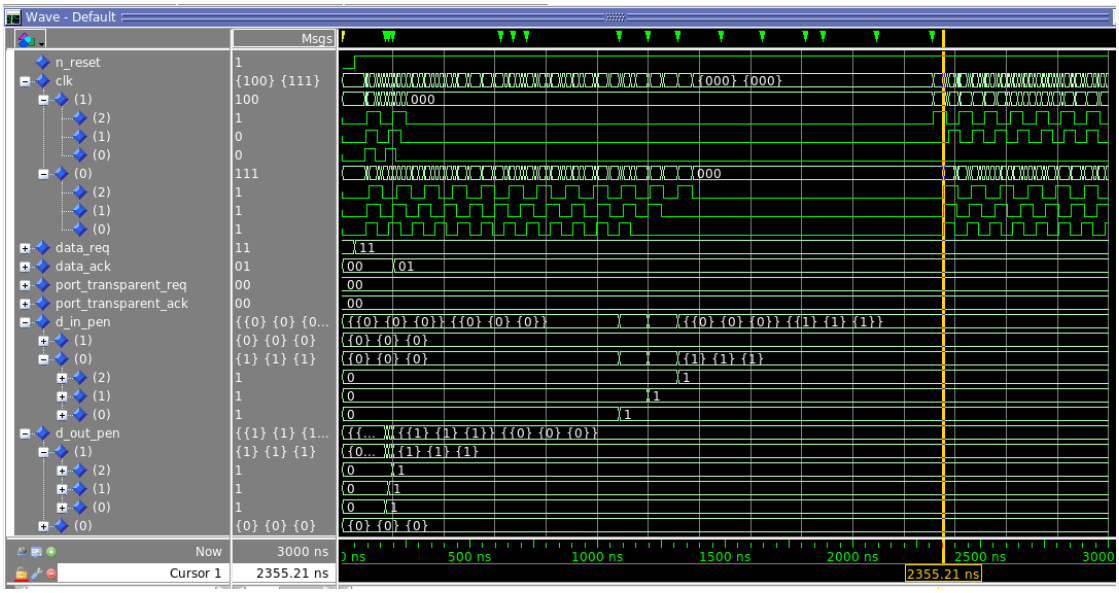


Figure 5.4: Channel request, with *data_req* signal stuck at one after being enabled once

Output port request Rp or acknowledge Ap stuck at one Unlike above where all channels were blocked, when the *Rp* signal or the *Ap* of the output designed by Villiger, that is within the TMR output port, is stuck at one, only one channel is blocked, while the other ones can continue to work properly. To avoid propagating this failure over the complete TMR system, the inputs must be able to handle such behavior. They do it by not directly forwarding the request from the bus to the input. Instead it is only forwarded, when the input port is ready to receive data and there is no valid data yet stored at the address defined in the message. The latter requirement is responsible for releasing the receiver port after a successful transaction. As Figure 5.5 shows, one channel is indeed blocked forever, as the *port_transparent_ack* signal is never released because the AFSM is stuck.

Sender port babbling idiot Here it highly depends on how the babbling idiot behaves. Depending on whether the sender waits until the transaction is completed or not, the storing on the receiver side is affected. In case it comes from a loose contact it might toggle too fast for the storing mechanism to be completed until the request is removed. While in a TMR poll type input port on the next request the data is tried to be written again, in a TMR demand type input port, data is only written, when at least another

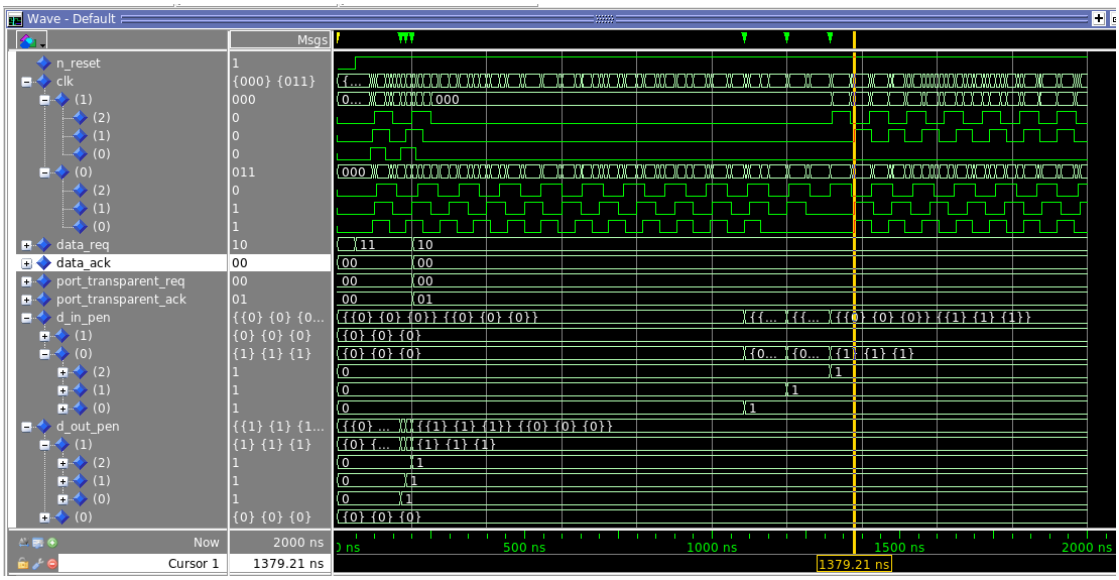


Figure 5.5: Data transmission, with request *Rp* stuck at one after being enabled once

port also wants to write. However when buffer pipelines are used, a babbling idiot might fill them. Thus this is an error case where buffer pipelines are counterproductive.

Pipeline broken An acknowledge signal within the pipeline that is either stuck at zero or at one, stops the propagation of output addresses throughout the pipeline and has the same effect as when the channel request is stuck at one. As there is no output address read from it, the channels are idling whereas the output ports do not get a grant to their bus request. On the other hand, when there are bit flips within the pipeline, there are different cases to distinguish. When one address bit is stuck at one, the same as above holds, as no null phase can be introduced. However, when one address bit in a stage is stuck at zero, this has no influence until the corresponding output is activated and requests the bus. In that case the pipeline also stops working, because the broken stage is stuck within the null phase.

Channel helper stuck at faults beside channel request stuck at one The biggest influence when it comes to stuck at faults in the channel helper module, is a stuck at one fault of the channel request as then the whole bus is blocked. However when one of the other modules is stuck at one only one channel is blocked and depending on where the stuck at fault is, also one sender port might not be released.

Output port/ Output port adapter stuck at faults beside output port arbiter request stuck at one In case one of the two AFSMs is stuck at and this stuck at fault is not the one requesting the arbiter, only one channel is blocked forever in the worst case. In case the stuck at fault is before the output port arbiter request happens, not

even a channel is requested and thus only the output port is blocked. As shown above the TMR input ports can handle one replica that is permanently requesting and sending data and thus these faults only lead to longer transaction times, as 3of4 voter timeouts are reached.

Fault	Blocked port	Blocked channel	Blocked bus	Wrong port	Pipelining problematic
Channel arbiter stuck at one	✗	✗	✗		
Port arbiter stuck at one	✗	✗	✗		
Output port Rp stuck at one	✗	✗			
Output port Ap stuck at one	✗	✗			
Babbling idiot				✗	✗
Pipeline Ack signal	✗	✗	✗		
Pipeline address signal	✗	✗	✗		
Channel helper AFSM	✗	✗	✗		
Output port/port adapter AFSM	✗	✗			

Table 5.5: Interesting stuck at fault, beside the replica border crossing signals

5.4.3 Conclusion

The in-depth fault analysis has shown that all possible faults that may propagate over the bus to other modules, stay within the replica where they occurred. This has been achieved by analyzing possible bit flip and stuck at faults of all signals that are used for communication between the replicas. This also includes the parts where these signals are merged together but also where the signals are split apart and sent to different modules. Furthermore merging signals is always done via voters and thus also faults for the voters were analyzed. As there is no other way a fault can propagate to another replica, this analysis can be considered complete.
However the replica where the fault occurred, might be working improperly until the next system reset. This also holds for the transient faults, as many of them introduce lasting errors to the system. Especially the best effort ones, while they have less influence on TMR messages
Another outcome that the analysis has shown is, that the optional buffer pipelines, that help a way too slow module to keep functioning, are worsening the things for many other faults and tend to cause a lasting failure of the affected resource. While this can be tolerated by the TMR architecture, it unnecessarily exhausts the available redundancy,

and thus degrades tolerance against further faults. So here it needs to be checked how probable the different failure cases are and with that information it can be then decided whether the buffer pipelines are a help or not for the specific use case.

CHAPTER 6

Performance analysis

The aim of this chapter is to compare the newly designed multi channel bus to the baseline MOGLI as well as discussing different parameters and how to choose them depending on the overall system. The results of this analysis are then verified via simulations.

6.1 Setup and Definitions

In the following the simulation setup and the definitions used for the performance analysis are described in more detail.

6.1.1 Simulation setup

For verifying the different attributes of the newly designed bus and also for better understanding and comparison, various simulations with ModelSim [Cora] have been made. As the synthesis of a VHDL project with Quartus [Corb] is non deterministic, which means that every synthesis comes up with a slightly different physical layout, it is difficult to deeply analyze a system with a postlayout simulation. Thus only prelayout simulations are presented in the following. For the gate delays the demo values shown in Table 6.1 were used. Again here it is difficult to get correct delays, as multiple parameters are responsible for the delay times, like number of transistors necessary, fan out etc..

6.1.2 Transaction duration definitions

The transaction duration is defined as the time it takes from the moment an output port is enabled by *Pen* until the transaction is completed and the local clock resumes.

In general the transaction can be split into three phases:

Gate	Delay [ps]
NOT	10
AND	40
OR	30
XNOR	50
C	60
LATCH	60
MUTEX	100
MUX	400

Table 6.1: Gate delays for the prelayout simulation

Bus access In this first phase the time from the output port enable signal *Pen* onward is measured until the output receives the grant signal to access the bus.

Data transfer During the second phase the actual communication between sender and receiver takes place. The data transfer phase starts with the bus grant signal and ends with the Ap' signal going to zero.

Bus release After the acknowledge signal Ap' becomes inactive, the output port releases the bus and resumes the local clock. The third phase is completed when both the bus grant signal *BusGrant* and the *Ai* signal are zero again.

Marking of Transitions

Durations that measure the time between two transitions have the structure $\delta_{A\circ,B\circ}$, where $\circ$ is either $+$(rising edge) or $-$(falling edge). However in case the *Pen* signal is involved, no transition mark is there, as the direction of the transition does not matter in this case.

6.2 MOGLI

In a first step Villiger's MOGLI approach is analyzed as it serves as baseline for the multi channel bus. For better comparison the dual channel MOGLI approach is used and only the transaction duration from the initiator to the target is considered but not the time it takes for calculating the response and then send it back over the response channel to the initiator. Furthermore since the influence of additional channels is the main focus of this analysis, it is considered that the MOGLI also works with the same ports as the multi channel bus, which allows multi casting and thus no central address decoder is necessary. The composition of the single phases for the MOGLI approach are outlined in equation 6.1-6.4.

$$\delta_{BusAccess_M} = \delta_{Pen,BusReq+} + \delta_{BusReq+,BusGrant+} \tag{6.1}$$

$$\delta_{DataTrans_{M_{p_{out}}}} = \delta_{BusGrant+,Rp+} + \delta_{Rp+,Ap'+} + \delta_{Ap'+,Ri+} + \delta_{Ri+,Ai+} + \delta_{Ai+,Rp-} + \delta_{Rp-,Ap'-} \tag{6.2}$$

$$\delta_{DataTrans_{M_{d_{out}}}} = \max(0, \delta_{Pen,Ri+} + \delta_{Ri+,Ai+} - \delta_{BusAccess_M}) + \delta_{Ai+BusGrant+,Rp+} + \delta_{Rp+,Ap'+} + \delta_{Ap'+,Rp-} + \delta_{Rp-,Ap'-} \tag{6.3}$$

$$\delta_{BusRelease_M} = \max(\delta_{Ap'-,Ri-} + \delta_{Ri-,Ai-}, \delta_{Ap'-,BusReq-} + \delta_{BusReq-,BusGrant-}) \tag{6.4}$$

The duration of $\delta_{BusAccess_M}$ highly depends on $\delta_{BusReq+,BusGrant+}$ as this is the delay arising from the arbiter and thus this can be arbitrarily long. However, as outlined in [Vil05] it is at least $\delta_{MUTEX} + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1)$, where k is the number of output ports that can request the bus.

Depending on the output port type, the local clock is halted immediately after the port is enabled or only after Ap' becomes active. In case of a demand output port $BusReq+$ and $Ri+$ happen directly after the port is enabled. Thus only the slower path needs to be considered. The actual clock pausing happens during the $\delta_{Ri+,Ai+}$ time frame. While in the original MOGLI this is in the best case just δ_{MUTEX}, it is $\delta_\Delta + \delta_{AND}$ where $\delta_\Delta > \delta_{OR} + \delta_{MUTEX}$ holds, in the adapted pausable clock mechanism, that allow poll ports to work even when the clock is already halted for a longer time by some other port. In the worst case the clock got its grant when $Ri+$ arrives at the MUTEX. With the clock generator presented by Fan [FKG09] the worst caste time until the clock is halted is $d_{D1} = d_{AND0} + d_{AND1} + (d^0_{MUTEX} + \Delta d_{MUTEX})$ whereas for the optimized clock generator from chapter 3, where only one MUTEX element is used, it is $\delta_{D1} = \delta_{AND} + (\delta^0_{MUTEX} + \delta_{\Delta MUTEX})$.
In the following when comparing the multi channel approach to the MOGLI approach it is always assumed, that the optimized clock generator as presented in Figure 3.12 is used, although the transaction duration might increase by $2 \cdot (\delta_\Delta - \delta_{MUTEX} + \delta_{AND}) + (\delta_{OR} + \delta_{AND})$ in the worst case when two poll ports communicate with each other. This is because the special focus of this analyze is the effect of having more than one channel. Furthermore the the multi channel bus also works with the clock generated of Fan [FKG09], although the same issues with poll ports occur as in Villiger's MOGLI [Vil05] approach.

Two other durations that are variable are $\delta_{Rp+,Ap'+}$ and $\delta_{Rp-,Ap'-}$. They highly depend on what kind of input port is used and whether this one is ready to take data or not. These durations are as defined in Equation 6.5-6.10. For Villiger's poll type input port it makes no difference for the response time whether the input was ready or not. Both sequences look the same. For the demand type input port the situation looks quite different as when the port is not yet ready only a MUTEX is traversed, whereas when it is ready quite some combinational logic gets involved. Especially for $\delta_{Rp-,Ap-_{d_{in}}}$ not only a MUTEX is traversed while releasing the clock but also combinational logic.

$$\delta_{Rp+,Ap'+_{p_{in}}} = \delta_{Rp+,Ri+} + \delta_{Ri+,Ai+} + \delta_{Ai+,Ap'+} \tag{6.5}$$

$$\delta_{Rp-,Ap'-_{p_{in}}} = \delta_{Rp-,Ap'-} \tag{6.6}$$

$$\delta_{Rp+,Ap+_{d_{in}}} = \delta_{Rp+,RpInt+} + \delta_{RpInt+,Ap+} \tag{6.7}$$

$$\delta_{Rp-,Ap-_{d_{in}}} = \delta_{Rp-,RpInt-} + \delta_{RpInt-,Ri-} + \delta_{Ri-,Ai-} + \delta_{Ai-,Ap-} \tag{6.8}$$

$$\delta_{Rp+,NAp+_{d_{in}}} = \delta_{MUTEX} \tag{6.9}$$

$$\delta_{Rp-,NAp-_{d_{in}}} = \delta_{MUTEX} \tag{6.10}$$

After a successful handshake the sender port releases both the bus and clock. For releasing the bus it holds that $\delta_{BusReq-,BusGrant-} = \delta_{MUTEX} + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1)$ and for the clock it holds that $\delta_{Ri-,Ai-} = \delta_{OR} + \delta_{MUTEX} + \delta_{AND}$.

6.3 Multi channel bus

When it comes to a single transaction the main difference between the multi channel bus with no multicast and Villiger's MOGLI is the bus access mechanism. Since the multi channel bus has more than one channel the resulting logic to access one of the channels is more complex and therefore introduces more delay as outlined below in Equation 6.11-6.14. In case multicast is possible additional combinational logic is required to merge all *Ap* and *Nap* signals to one single acknowledge signal but therefore no address decoding is needed as this is already done by the synchronous module.

$$\begin{aligned}\delta_{BusAccess_{MC}} =& \delta_{Pen,BusReq+} + \delta_{BusReq+,ArbiterReq+} \\ &+ \delta_{ArbiterReq+,ArbiterAck+} + \delta_{ArbiterAck+,ArbiterReq-} \\ &+ \delta_{ArbiterReq-,ArbiterAck-} + \delta_{ArbiterAck-,BusGrant+}\end{aligned} \tag{6.11}$$

$$\delta_{DataTrans_{MC_{p_{out}}}} = \delta_{DataTrans_{M_{p_{out}}}} + \delta_{MUX_{Rec}} \tag{6.12}$$

$$\delta_{DataTrans_{MC_{d_{out}}}} = \delta_{DataTrans_{M_{d_{out}}}} + \delta_{MUX_{Rec}} \tag{6.13}$$

$$\delta_{BusRelease_{MC}} = \max(\delta_{Ap'-,Ri-} + \delta_{Ri-,Ai-}, \delta_{Ap'-,BusReq-} + \delta_{BusReq-,BusGrant-}) \tag{6.14}$$

As Equation 6.13 and 6.12 show the a single data transaction itself takes the same amount of time as in Villiger's MOGLI approach. However when more than one channel is used and data is sent to a demand type input port an additional delay $\delta_{MUX_{Rec}}$ is added. This delay is the time it takes to arbitrate which channel can access the demand type port, followed by the traversing time of the multiplexer. When now the request is removed, it again has to traverse the arbiter first and then only one multiplexer delay is considered as the multiplexer is deactivated and thus setting all output signals to zero.

$$\delta_{MUX_{Rec}} = \begin{cases} 2 \cdot (\delta_{MUTEX} + \delta_{ARBITER}) \cdot (\lceil \log_2 m \rceil - 1) + 3 \cdot \delta_{MUX}, & \text{if receiver} = d_{in} \\ & \text{and } m > 1 \\ 0, & \text{otherwise} \end{cases} \tag{6.15}$$

Like in the MOGLI approach $\delta_{ArbiterReq+,ArbiterAck+}$ results from the output arbiter, that arbitrates all bus requests from the output channels and the time it takes to store the arbiter grant signals into the first pipeline stage. The overall introduced delay for k output ports is at least like in Equation 6.16, when there is a free space in the pipeline.

$$\begin{aligned} \delta_{ArbiterReq+,ArbiterAck+} \geq\ & \delta_{MUTEX} + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1) \\ & + 2 \cdot \delta_{C-ELEM} \end{aligned} \tag{6.16}$$

Depending on the number of pipeline stages l the delay resulting from the pipeline itself is $l \cdot \delta_{C-ELEM}$. As the request is directly stored in the pipeline when there is a free space the bus request can be removed from the arbiter and the output port starts waiting until it can access the bus. Due to this decoupling the duration for $\delta_{ArbiterAck+,BusGrant+}$ (Equation 6.17) is the longer path of the two existing ones. The first one is the time it takes the combinational logic to deassert the port arbiter request and the time it takes for the deasserted request to propagate through the arbiter and the second path is the time it takes the stored request in the pipeline to traverse the $(l-1)$ remaining stages, then be requested by one of the channels and finally given the grant to access the bus. Here the best case is assumed for the latter path when the channel helper is already requesting an output address from the pipeline. Thus it is assumed that the channel arbiter is already requesting the source and only the time it takes from granting the source and traversing the arbiter is considered in $\delta_{ARBITER_{sourceAck}}$. Since the request is not just fed back at the channel arbiter, there is no MUTEX in the last arbitration stage but instead also an arbiter. Thus the delay inserted by a tree arbiter for m channels is $\delta_{ARBITER} \cdot (\lceil \log_2 m \rceil))$. The delay that results from the pipeline-arbiter connector is considered with $\delta_{pip,arb}$. The output port adapter has to wait for both paths to complete until a grant for the bus is given to the output port. To indicate this the variable $adapterForkComp$ is introduced which is activated after both $ArbiterAck-$ and $portTransparentReq+$.

$$\begin{aligned} \delta_{ArbiterAck+,BusGrant+} \geq\ & \max(\delta_{ArbiterAck+,ArbiterReq-} + \delta_{MUTEX} \\ & + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1) + 2 \cdot \delta_{C-ELEM}, \\ & (l-1) \cdot \delta_{C-ELEM} + \delta_{OR} + \delta_{pip,arb} \\ & + \delta_{ARBITER_{sourceAck}} \cdot \lceil \log_2 m \rceil + \delta_{dataAck+,dataReq-} \\ & + \delta_{ARBITER} \cdot \lceil \log_2 m \rceil + \delta_{dataAck-,portTransparentReq+}) \\ & + \delta_{adapterForkComp+,BusGrant+} \end{aligned} \tag{6.17}$$

$$\begin{aligned} \delta_{pip,arb} =\ & \delta_{pipReq+,channelGrant+} + \delta_{channelReq-,pipAck+} \\ & + \delta_{pipAck+,channelGrant-} \end{aligned} \tag{6.18}$$

Another difference to the MOGLI approach is, that the $\delta_{BusReq-,BusGrant-}$ of the multi channel approach is not traversing an arbiter as the *BusGrant* signal is generated by combinational logic that does not include an arbiter.

6.3.1 MOGLI vs. Multi channel bus

In the following the MOGLI approach from Villiger and the new multi channel bus are compared with each other. This is done by analyzing relevant metrics. For better comparison, it is assumed, that the the multi channel bus only uses one pipeline, where the requests from the k output ports are stored, and only one channel. In this case the effect of hidden arbitration can be analyzed best.

Latency

One of the most important metrics is the latency. Thus the questions is, how much the latency is increased due to the higher complexity of the multi channel approach.

The difference of the latency between the MOGLI and the multi channel bus is calculated in Equation 6.26, based on partial results shown in Equation 6.19- 6.25.

$$\delta_{Transaction_M} = \delta_{BusAccess_M} + \delta_{DataTrans} + \delta_{BusRelease_M} \tag{6.19}$$

$$\delta_{Transaction_{MC}} = \delta_{BusAccess_{MC}} + \delta_{DataTrans} + \delta_{BusRelease_{MC}} \tag{6.20}$$

$$\begin{aligned}\delta_{Transaction_{MC}} - \delta_{Transaction_M} =& \delta_{BusAccess_{MC}} + \delta_{BusRelease_{MC}} \\ & - \delta_{BusAccess_M} - \delta_{BusRelease_M}\end{aligned} \tag{6.21}$$

$$\begin{aligned}\delta_{BusAccess_M} =& \delta_{Pen,BusReq+} \\ & + \delta_{MUTEX} + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1)\end{aligned} \tag{6.22}$$

$$\begin{aligned}\delta_{BusAccess_{MC}} =& \delta_{Pen,BusReq+} + \delta_{BusReq+,ArbiterReq+} \\ & + \delta_{MUTEX} + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1) \\ & + 2 \cdot \delta_{C-ELEM} + \delta_{ArbiterAck+,ArbiterReq-} \\ & + \delta_{MUTEX} + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1) + 2 \cdot \delta_{C-ELEM} \\ & + \delta_{ArbiterAck-,BusGrant+}\end{aligned} \tag{6.23}$$

$$\begin{aligned}\delta_{BusAccess_{MC}} - \delta_{BusAccess_M} =& \delta_{BusReq+,ArbiterReq+} + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1) \\ & + \delta_{ArbiterAck+,ArbiterReq-} + \delta_{ArbiterAck-,BusGrant+} \\ & + 4 \cdot \delta_{C-ELEM} + \delta_{MUTEX}\end{aligned} \tag{6.24}$$

$$\begin{aligned}\delta_{BusRelease_{MC}} - \delta_{BusRelease_M} =& \max(\delta_{Ap'-,Ri-} + \delta_{Ri-,Ai-}, \\ & \delta_{Ap'-,BusReq-} + \delta_{BusReq-,BusGrant-}) \\ & - \max(\delta_{Ap'-,Ri-} + \delta_{Ri-,Ai-}, \\ & \delta_{Ap'-,BusReq-} + \delta_{MUTEX} + \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1))\end{aligned} \tag{6.25}$$

$$\begin{aligned}\delta_{Transaction_{MC}} - \delta_{Transaction_{M}} \geq & \delta_{BusReq+,ArbiterReq+} \\ & + \delta_{ArbiterAck+,ArbiterReq-} + 4 \cdot \delta_{C-ELEM} \\ & + \delta_{adapterForkComp+,BusGrant+} \\ & + \delta_{BusReq-,BusGrant-} \end{aligned} \tag{6.26}$$

In the best case the output arbiter path is the slower one in Equation 6.17, as despite the additional write to the first pipeline stage, this path has to be traversed in the MOGLI approach too. With large enough n this path also is the dominant one, when it is assumed that only one channel is used and thus on the channel side no arbiter is needed. Furthermore one pipeline stage needs less time than an arbiter stage, as it only consists of one C element and an OR gate, while in the arbiter also a C and a MUTEX element have to be traversed.

When analyzing $\delta_{BusRelease_{MC}} - \delta_{BusRelease_{M}}$ the minimum is reached, when the first term is minimal while the second is maximal. For large k the arbiter delay becomes dominant in the $\delta_{BusRelease_{M}}$ but for the $\delta_{BusRelease_{MC}}$ delay it is not that easy to tell which delay is dominant. However $\delta_{Ri-,Ai-}$ always involves traversing a MUTEX element while in the other path there are simpler gates to traverse. Thus it is assumed that the latter one is the faster one.

As Equation 6.26 shows, the additional latency of the multi channel bus is in the best case only the write to the first pipeline stage and the combinational delays that result from the output port adapter. This timing analysis also shows that the output port request arbiter is the limiting source when there are a lot of output ports. Thus splitting the output ports into different groups and using one tree arbiter for each of this groups helps increasing not only the throughput but also decreasing the latency.

Figures 6.1 and 6.2 show a data transaction between a demand type output port and a poll type port input port that responds with a NAp. It also shows that the simulation corresponds with Equation 6.26. While all delays from the output port adapter can be directly taken from Figure 6.2, the delay for the C gate was assumed with 60 ps as outlined in Table 6.1. Thus we have $40 + 120 + 160 + 190 + 4 \cdot 60 = 750$ which is exactly the difference measured in Figure 6.1.

Equation 6.26 only gives a lower bound for the latency difference between the MOGLI and the multi channel approach. However, when having a closer look it turns out, that the latency difference increases with the number of channels (due to the channel arbitration mechanism) and also when hardly any sender ports are used. When hardly any sender ports are used, the advantage of hidden arbitration has not such a high impact. Figure 6.3 shows the latency difference between the multi channel bus and the MOGLI approach for two different scenarios. The first one is where only 4 sender ports were used. This one is assumed to be the worst case scenario regarding the latency difference, as a multi channel port for less sender ports hardly make any sense regarding the area overhead needed for the port arbitration and channel selection. Whereas the second one is considered the best case scenario, where the latency difference between the MOGLI and multi channel bus is

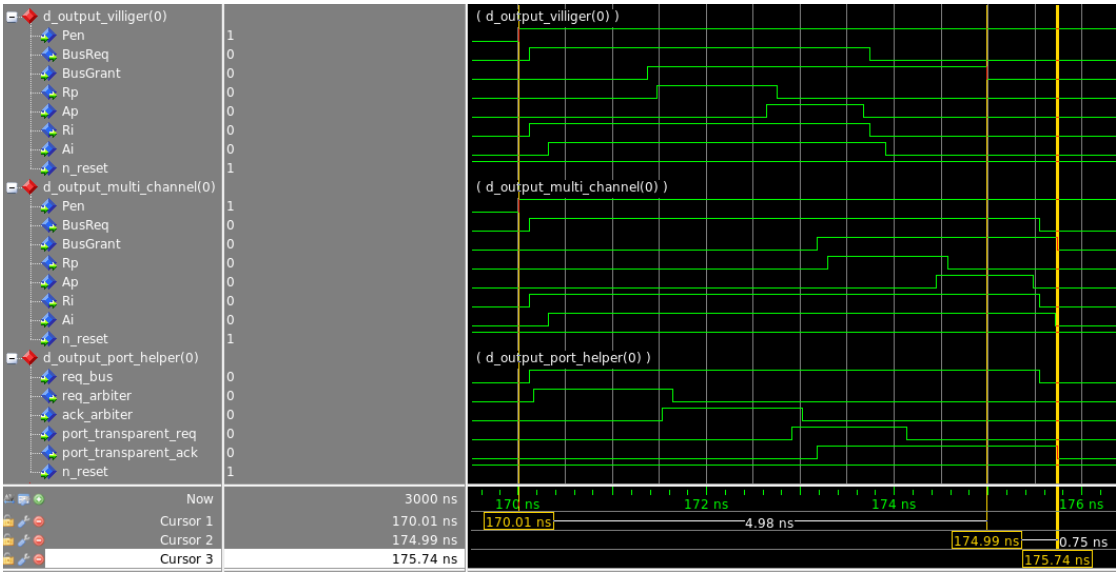


Figure 6.1: Latency difference between MOGLI and multi channel approach

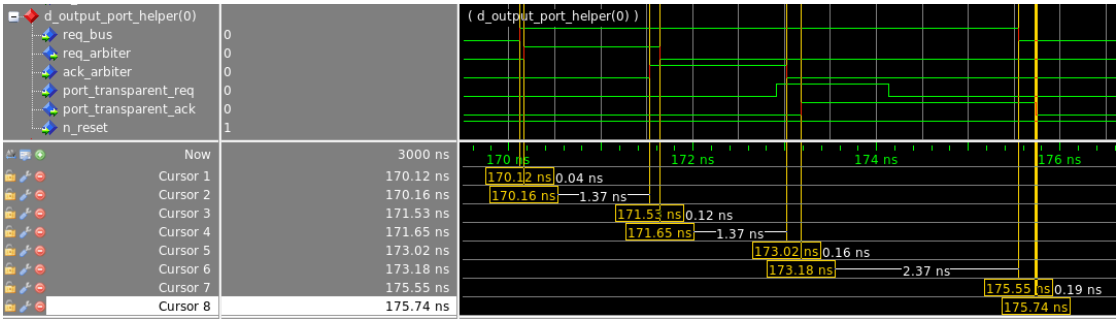
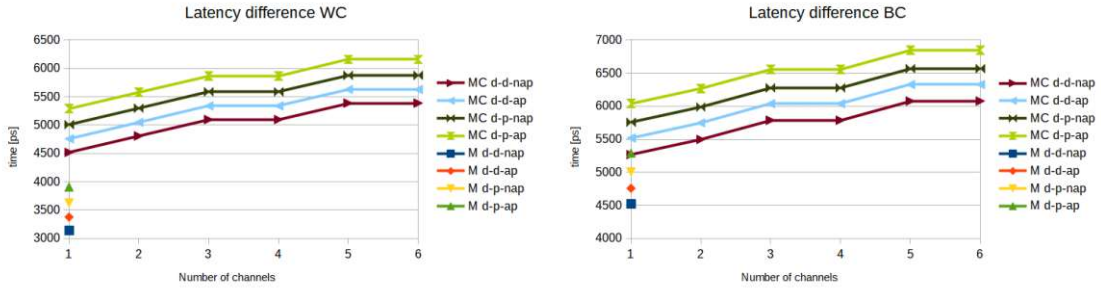
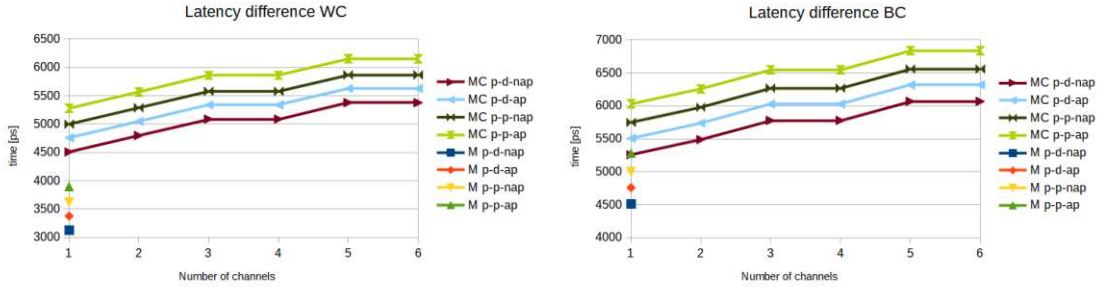


Figure 6.2: Latency difference between MOGLI and multi channel approach

minimal. Therefore 5 arbiter stages where needed, which allows for a total of 32 sender ports. The four diagrams also show all possible combinations of sender and receiver ports and it can be seen that the latency is larger when the receiver is a poll port. This is because there way more checks where necessary, as there exist multiple table entries and thus an address check needs to be done. Furthermore the clock of the receiver is halted during the transaction and not before, like it is the case for demand type input ports. As the storing of the data also consumes some time the transactions where data is stored (marked with *ap*) are longer. Another thing that can be seen is that there is hardly any difference between a poll type sender and a demand type sender, when it comes to timings. This is because the only difference between these two port types is the time when the clock is stopped. While for demand type ports the clock is stopped immediately after the port is enabled, for poll type ports it is only stopped after an acknowledge is detected. Thus the poll type ports may need longer to get a grant, as the handshaking



(a) Worst case demand output (4 possible senders) (b) Best case demand output (32 possible senders)



(c) Worst case poll output (4 possible senders) (d) Best case poll output (32 possible senders)

Figure 6.3: Latency difference between MOGLI and multi channel approach

signal might arrive while the MUTEX element is blocked by the clock generator.

Throughput

When analyzing the throughput the time that can not be beaten in the multi channel approach is $\delta_{ChannelTime_{MC}}$ defined in Equation 6.27, as this is the part of the transaction where the channel is involved. Here also the time it takes to request an output port that is ready for a transaction, is considered. To achieve this time, it needs to hold that $\delta_{ChannelTime_{MC}} \geq \delta_{PrepareTrans_{MC}}$. Note that as preparation time only the arbitration process is considered as this is the point where the serialization takes place and it is assumed that there is a port already waiting and thus already requesting the arbiter. With higher k the arbitration time takes over and from some point on it will be the dominant delay source and thus the main limitation for the throughput.

$$\begin{aligned}\delta_{ChannelTime_{MC}} =& \delta_{dataReq+,dataAck+} + \delta_{dataAck+,dataReq-} + \delta_{dataReq-,dataAck-} \\ &+ \delta_{dataAck-,portTransparentReq+} + \delta_{portTransparentReq+,BusGrant+} \\ &+ \delta_{DataTrans_{MC}} + \delta_{Ap'-,BusReq-} + \delta_{BusReq-,BusGrant-} \\ &+ \delta_{BusGrant-,valid-} + \delta_{valid-,DataReq+} \end{aligned} \tag{6.27}$$

$$\begin{aligned}\delta_{PrepareTrans_{MC}} =& \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1) + \delta_{MUTEX} + 2 \cdot \delta_{C-ELEM} \\ &+ \delta_{ArbiterAck+,ArbiterReq-} + \delta_{MUTEX} \\ &+ \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1) \end{aligned} \tag{6.28}$$

While in Villiger's approach the whole arbitration process is affecting the throughput, in the multi channel approach, the next output port is already arbitrated while there is an ongoing transaction. Thus as long as the overhead necessary for the hidden arbitration is not larger than the time gained through arbitrating the next port immediately after the previous one, the new approach is better regarding the throughput i.e. as long as Equation 6.29 holds, the throughput of the multi channel bus is better than the one of MOGLI.

$$\delta_{Transaction_M} - \delta_{Pen,BusReq+} > \delta_{ChannelTime_{MC}} + \max(\delta_{PrepareTrans_{MC}} - \delta_{ChannelTime_{MC}}, 0) \tag{6.29}$$

From Equation 6.29 directly follows Equation 6.30 and therefore it only needs to be shown, that both $\delta_{ChannelTime_{MC}}$ and $\delta_{PrepareTrans_{MC}}$ are not larger than $\delta_{Transaction_M} - \delta_{Pen,BusReq+}$. $\delta_{Pen,BusReq+}$ is subtracted because it is assumed that there is always a output port waiting to be arbitrated. This is also the case where the throughput is maximal.

$$\delta_{Transaction_M} - \delta_{Pen,BusReq+} > \max(\delta_{PrepareTrans_{MC}}, \delta_{ChannelTime_{MC}}) \tag{6.30}$$

Thus two cases have to be analyzed. One where the $\delta_{ChannelTime_{MC}}$ is the dominant duration and one where the $\delta_{PrepareTrans_{MC}}$ is responsible for the throughput:

- Case 1: $\delta_{ChannelTime_{MC}} > \delta_{PrepareTrans_{MC}}$
  Here the channel time is the longer duration. As there are at most as many channels as output ports, it is clear that the channel arbiter is not larger than the output port arbiter. However the last stage of the channel arbiter is just a MUTEX element and not an arbiter like it is the case for the output port arbiter. Furthermore the delays $\delta_{dataReq+,dataAck+}$ and $\delta_{dataReq-,dataAck-}$ not only include the delays coming from the tree arbiter. There also the delays for the pipeline-arbiter connector are

included. Thus the channel arbiter even needs to be smaller, in order for those two delays to be not larger than the $\delta_{BusReq+,BusGrant+}$ and $\delta_{BusReq-,BusGrant-}$ delay.

$$\begin{aligned}
\delta_{Transaction_M} - \delta_{Pen,BusReq+} &> \delta_{ChannelTime_{MC}} \\
\delta_{BusReq+,BusGrant+} + \delta_{BusRelease_M} &> \delta_{dataReq+,dataAck+} + \delta_{dataAck+,dataReq-} \\
&+ \delta_{dataReq-,dataAck-} + \delta_{dataAck-,portTransparentReq+} \\
&+ \delta_{adapterForkComp+,BusGrant+} \\
&+ \delta_{Ap'-,BusReq-} + \delta_{BusReq-,BusGrant-} \\
&+ \delta_{BusGrant-,valid-} + \delta_{valid-,DataReq+} \qquad (6.31)
\end{aligned}$$

Equation 6.31 only holds, when the output port arbiter is large enough to cover all the delays that are inserted not only by the channel arbiter but also the channel helper AFSM. However, when the output port arbiter becomes too dominant, which is the case when there are a lot of output ports, the preparation time is larger than the channel time.

- Case 2: $\delta_{ChannelTime_{MC}} \leq \delta_{PrepareTrans_{MC}}$
  In case the preparation time exceeds the channel time, the output port arbiter must be already relatively big compared to the channel arbiter. Thus also it is assumed that the output port arbiter path is longer than the pipeline path but also longer than the clock release path.

$$\begin{aligned}
\delta_{Transaction_M} - \delta_{Pen,BusReq+} &> \delta_{PrepareTrans_{MC}} \\
\delta_{DataTrans_M} + \delta_{BusRelease_M} &> 2 \cdot \delta_{C-ELEM} \\
&+ \max(\delta_{ArbiterAck+,ArbiterReq-} + \delta_{MUTEX} \\
&+ \delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1), \\
&(l-1) \cdot \delta_{C-ELEM} + \delta_{OR}) + \delta_{ArbiterAck-,BusGrant+} \\
\delta_{DataTrans_M} + \delta_{Ap'-,BusReq-} &> 2 \cdot \delta_{C-ELEM} + \delta_{ArbiterAck+,ArbiterReq-} \\
&+ \delta_{ArbiterAck-,BusGrant+} \qquad (6.32)
\end{aligned}$$

With these preconditions Equation 6.32 has to hold in order for the multi channel bus approach to have a higher throughput than the MOGLI bus. Since a transaction also includes various combinational circuits and thus delays, this should be no problem even in case a NAp is received, as the output ports designed by Villiger are already far more complex than the newly introduced output port adapter.

Number of channels

The above analysis also helps in finding a reasonable number of channels for the bus. However it has to be noted that there for the arbitration always the worst case was assumed, where the next request is only arbitrated when the previous one finished and the

grant is deactivated again. So in general an even higher throughput might be achieved. So the following numbers are just a lower bound for the number of channels to achieve the best possible throughput.

- $\delta_{ChannelTime_{MC}} \leq \delta_{PrepareTrans_{MC}}$:
  In this case the preparation time exceeds the channel time and thus it makes no sense to have more than one channel, as only one channel is busy at a time.

- $\delta_{ChannelTime_{MC}} > \delta_{PrepareTrans_{MC}}$:
  When the channel time is larger than the preparation time however more channels can be added such that the maximal throughput can be reached. The maximal throughput is when the port arbiter is always busy. Thus there has to be a free channel every $\delta_{PrepareTrans_{MC}}$. This leads to Equation 6.33 for the lower bound of channels for the multi channel bus where the prepare time is smaller than the channel time. However it has to be noted here that the channel time also might increase when more channels are used.

$$N_{channels} \geq \left\lceil \frac{\delta_{ChannelTime_{MC}}}{\delta_{PrepareTrans_{MC}}} \right\rceil \tag{6.33}$$

Comparison with Simulation

For validation purposes above formulas for the throughput were checked with a simulation. Two modules, where the demand type ports were communicating with each other, were simulated. In order to avoid any influence on the clock of the other module, the input ports were not ready and thus sending a NAp after δ_{MUTEX}. The results for different port arbiter sizes are listed in Table 6.2 and presented graphically in Figure 6.4. While the preparation time is identical for both approaches, the actual transaction time highly differs between the two approaches. However the multi channel approach is able to beat the MOGLI approach, even when it also is built with just one channel, when the number of port arbiter stages is high and thus the preparation time becomes dominant and exceeds the channel time. However when the number of channels is as large as defined in Equation 6.33, the throughput can be maximized. The maximal possible throughput is then only limited by the preparation time. To decrease the preparation time, an approach with multiple pipelines is then necessary.
As the only parameter that is changed in this analysis is the output port adapter, the difference between the preparation time and the MOGLI throughput stays constant.

Figure 6.5 shows that the possible throughput for a multi channel bus with one channel and a port arbiter with 7 arbiter stages is indeed one transaction per 4430 ps. On the other hand, the throughput can be increased per instruction to 3660 ps when a second channel like in Figure 6.6 is used. However it is not possible to increase the throughput with an additional channel any further as this is already the limiting preparation time.

Arbiter stages	1	2	3	7	8	9	10
$\delta_{Throughput_M}$	3030	3490	3950	5790	6250	6710	7170
$\delta_{ChannelTime_{MC}}$	4430	4430	4430	4430	4430	4430	4430
$\delta_{PrepareTrans_{MC}}$	900	1360	1820	3660	4120	4580	5040
single channel $\delta_{Throughput_{MC}}$	4430	4430	4430	4430	4430	4580	5040
max possible $\delta_{Throughput_{MC}}$	900	1360	1820	3660	4120	4580	5040
$\delta_{ChannelTime_{MC}}/\delta_{PrepareTrans_{MC}}$	4.922	3.257	2.434	1.210	1.075	0.967	0.879

Table 6.2: Demand type output ports communicating with demand type input ports and receiving NAp

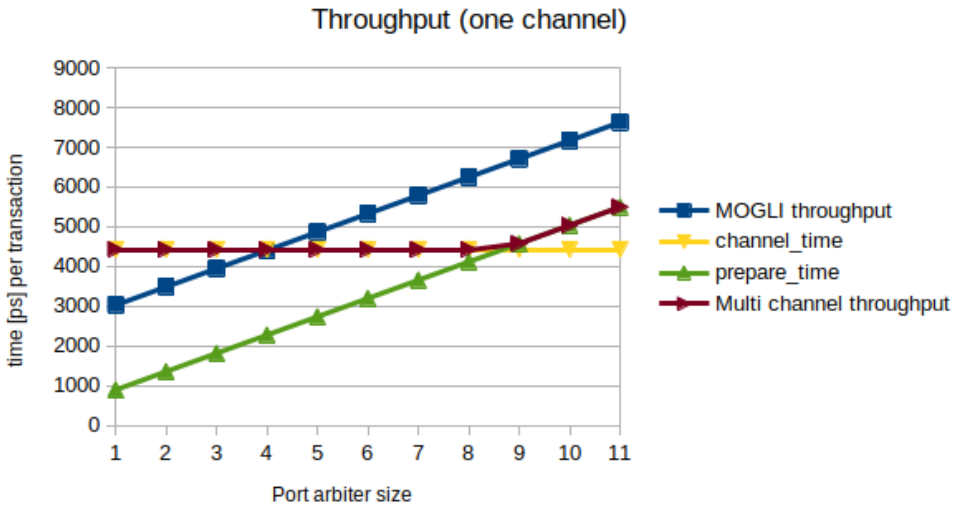


Figure 6.4: Throughput comparison with different port arbiter depths

6.3.2 Pipeline depth

An important property that has not been discussed yet, is how to choose the depth l of the pipelines properly. The pipelines were introduced to allow hidden arbitration. This means that during an ongoing transaction it is already evaluated which output port is allowed to send next. However the deeper a pipeline is, the more latency is added to the overall latency. Thus in the following different depths are discussed. As the pipeline that is presented in 3.2.3 is based on a four phase handshake, only in every second stage a new data word is stored.

- No pipeline ($l = 0$)
  In case the pipeline is totally omitted, no hidden arbitration is possible, as the port arbiter and channel arbiter have to communicate directly with each other in this case.

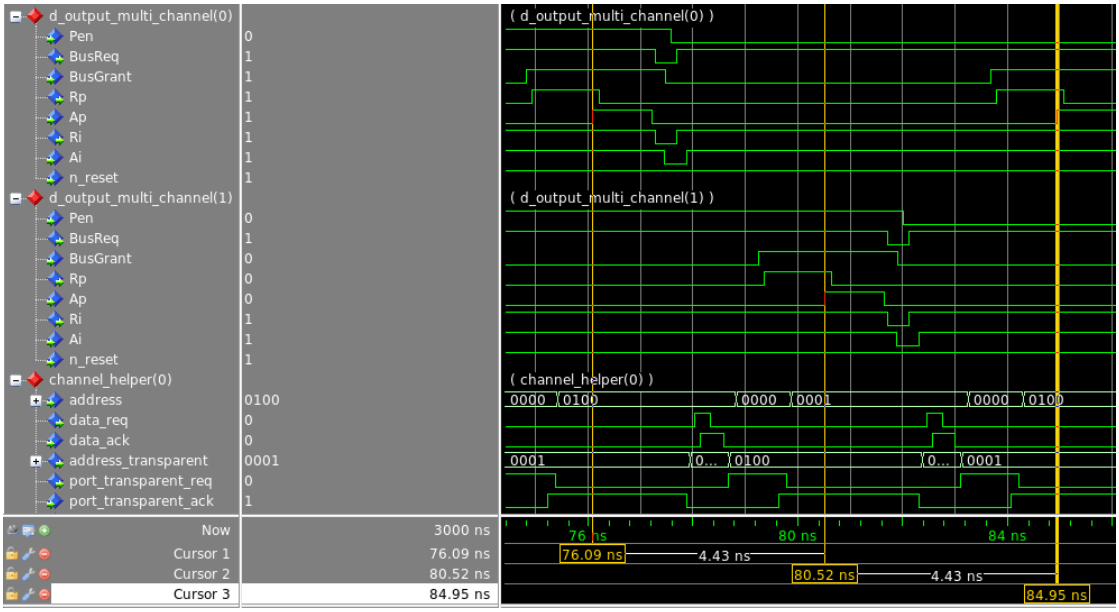


Figure 6.5: Throughput with 7 arbiter stages and one channel

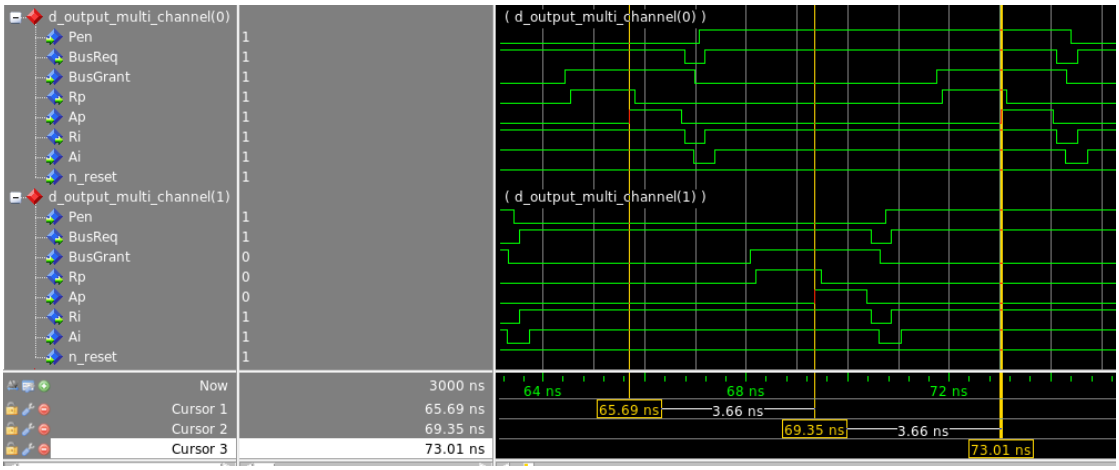


Figure 6.6: Throughput with 7 arbiter stages and two channels

- One entry ($l = 1$)
  When the pipeline consists of only one entry, hidden arbitration is already possible, as the next output port can already be evaluated while the one that is stored in the pipeline still waits for being assigned to a channel. However the one that is arbitrated can not be stored in the pipeline until a channel has requested the output address stored.

- More than one entry ($l \geq 3$)

With a pipeline depth of $l = 3$ already two elements can be stored, while a third one is arbitrated. However a deeper pipeline only makes sense, when this has a positive effect on the latency or throughput. One upper bound that can be defined for the number of elements is the number of channels. In case there are as many entries as channels, for every channel the next output port can be arbitrated while the transaction is still going on. However this is also only applicable, as long as the pipeline delay does not exceed the output port arbiter delay.

6.3.3 Multicast

Multicasts are an efficient way to send the same massage to various receivers and thus lead to fewer bus requests, which leads to more time for other services to access the bus. However the best case where a multi cast transaction is completed with acquiring the bus only once is only possible, when all ports that are addressed in the multicast are indeed ready to receive the data. In the worst case as many bus requests as with unicast are necessary. However here the hardware has no influence, as this highly depends on the algorithm. Thus the best way is to wait with initiating the multicast until it is ensured by the algorithm that the receiver ports are really ready to receive the data.

6.4 Multi channel TMR bus

For the multi channel TMR bus the same timing constraints hold as for the multi channel bus, as long as non critical messages are sent. For TMR messages the constraints defined in Equation 6.34-6.37 hold. The big difference is the voting mechanisms that need to be considered for TMR messages.

$$\begin{aligned}
\delta_{BusAccess_{TMR}} =& \delta_{Pen,VoteTMR} + \delta_{VoteTMR,BusReq+} + \delta_{BusReq+,ArbiterReq+} \\
&+ \delta_{ArbiterReq+,ArbiterAck+} + \delta_{ArbiterAck+,ArbiterReq-} \\
&+ \delta_{ArbiterReq-,ArbiterAck-} + \delta_{ArbiterAck-,BusGrant+} && (6.34) \\
\delta_{DataTrans_{TMR_{p_{out}}}} =& \delta_{BusGrant+,Rp+} + \delta_{Rp+,Ap+} + \delta_{Ap+,Ri+} + \delta_{Ri+,Ai+} + \delta_{Ai+,Rp-} \\
&+ \delta_{Rp-,Ap-} && (6.35) \\
\delta_{DataTrans_{TMR_{d_{out}}}} =& \max(0, \delta_{Pen,Ri+} + \delta_{Ri+,Ai+} - \delta_{BusAccess_M}) + \delta_{Ai+BusGrant+,Rp+} \\
&+ \delta_{Rp+,Ap+} + \delta_{Ap+,Rp-} + \delta_{Rp-,Ap-} && (6.36) \\
\delta_{BusRelease_{TMR}} =& \delta_{BusRelease_{MC}} && (6.37)
\end{aligned}$$

Equation 6.38-6.41 show in detail the different timings for $\delta_{Rp+,Ap+}$ and $\delta_{Rp-,Ap-}$ depending on the input port type. The additional time needed due to the voting process is hidden in the $\delta_{valid+,Ap+}$ signal.

$$\delta_{Rp+,Ap+_{p_{in}}} = \delta_{Rp+,Ri+} + \delta_{Ri+,Ai+} + \delta_{Ai+,Ap_p+} + \delta_{Ap_p+,write+} + \delta_{write+,valid+} + \delta_{valid+,Ap+} \quad (6.38)$$

$$\delta_{Rp-,Ap-_{p_{in}}} = \delta_{Rp-,Ap_p-} + \delta_{Ap_p-,write-} + \delta_{write-,Ap-} \quad (6.39)$$

$$\delta_{Rp+,Ap+_{d_{in}}} = \delta_{Rp+,Rp_i+} + \delta_{Rp_i+,Ap_i+} + \delta_{Ap_i+,valid+} \quad (6.40)$$

$$\delta_{Rp-,Ap-_{d_{in}}} = \delta_{Rp-,Ap-} \quad (6.41)$$

6.4.1 3of4 voter delays

Unlike synchronous TMR systems where all replicas are fed with the same clock or with synchronized clocks that only have a fixed phase shift between each other, in an asynchronous TMR system the phase shifts between the clock can increase over time. Thus TMR messages are also used for synchronization and therefore a TMR transaction might need more time than in a synchronous TMR system. However this difference becomes very evident, when one replica is dead. In that case the asynchronous TMR system may reduce its speed dramatically depending on the number of TMR transactions that take place. This is because the voter does not know whether the third replica is just late or dead and has to wait until it is clear that the third replica is slower than the maximal allowed phase shift. Thus when choosing proper delays this must also be considered as the system must be still able to meet deadlines in this degraded mode. Lechner already defined a lower bound presented in Equation 6.42 for the timeout value in his thesis [Lec14]. The variable p_i denotes the time it takes the sender i until the next transaction request to this port takes place. The delay Δ then has to be at least the difference between the slowest and the fastest possible period that is possible between two transactions.

$$\Delta > \max(p_i^{wc}) - \min(p_j^{bc}),\ i \neq j \quad (6.42)$$

In case that this resulting Δ is quite large due to the algorithm, it might be necessary to introduce additional TMR messages to the system to reduce the necessary delay Δ and thus keep the performance degradation in an acceptable area, when one replica fails, which also improves synchrony among the replicas.

The minimal delay for the transaction complete voter is calculated quite similar. In case the output port only sends data to one input port it is the same delay as for the data voter on the input side. However in case multiple modules might be addressed by one single output port, the worst case has to be considered. This results in Equation 6.43 where R_j is the set of all receiver ports addressed by output port s_j and $\Delta_{data_{r_i}}$ is the time until the timeout is reached on the data voter of receiver r_i.

$$\Delta_{trans\_comp_j} = \max(\Delta_{data_{r_i}} :\ r_i \in R_j) \quad (6.43)$$

For the special 3of4 voter that is used to release the demand type input port after a transaction, the release timeout is important. This has to be chosen long enough such that also the slowest replica can finish its transaction. The receiver only completes a transaction, when either all receivers have acknowledged the transaction or when at least two have and the 3of4 voter runs in a timeout. As all the senders were synchronized before accessing the bus, the difference of their timings only results from the time it takes to request the bus and do the arbitration. However since the multi channel bus is designed such that there have to be as many channels as possible simultaneous TMR messages, this time will be rather small and highly depends on the arbitration time. In Equation 6.44 the minimal delay for the request 3of4 voter for the demand input port with k output ports and u simultaneous TMR messages is presented. Important here is, that only the output port arbitration is considered, as the channel arbitration is the same for all replicas. The only difference is the order in which the output port addresses are written into the pipeline.

$$\Delta_{d_{in_{req}}} > (u-1) \cdot (\delta_{ARBITER} \cdot (\lceil \log_2 k \rceil - 1) + \delta_{MUTEX}) \tag{6.44}$$

However for this voter especially the deassertion is from interest. As here the question is, how long should the voter wait until it is assumed that the requesting replica is not working properly. This depends on the acknowledge voter, as it might be the case, that one request signal is stuck at zero. In that case two replicas can finish their transaction immediately. The replica with the fault however is still waiting for the acknowledge voter to run in a timeout. However since the other two requests are deasserted, also the internal request signal of the input port is deasserted and thus the faulty output ports get stuck as never an acknowledge is generated by the voter. However when the deassertion of the internal request signal of the receiver is delayed sufficiently this can be avoided. This then leads to the release timeout Equation 6.45 where S_j is the set of all sender ports addressing input port r_j and $\Delta_{trans\_comp_{s_i}}$ is the time until the timeout is reached on the transaction complete voter of sender s_i.

$$\Delta_{d_{in_{rel_j}}} \geq \max(\Delta_{trans\_comp_{s_i}} : \ s_i \in S_j) \tag{6.45}$$

6.4.2 Synchronization

As already outlined before, the TMR messages are also responsible for synchronizing the different replicas with each other and the TMR voters are responsible for the actual synchronization. Figure 6.7 show such a synchronization. The second fastest replica starts the timer for the timeout and in this case the slowest replica of the sender and the receiver are both fast enough ($timeout_{is\_tmr} = 10000$ and $timeout_{Data} = 5000$). However due to the fact that the transaction is finished faster than it takes the slowest receiver clock to finish the current clock period, it is still behind after the synchronization. But

it is behind less than one clock cycle. So as long as the phase shift is in the allowed range, after a TMR transaction the phase shift between the replica clocks is less than the slowest replica's clock period.

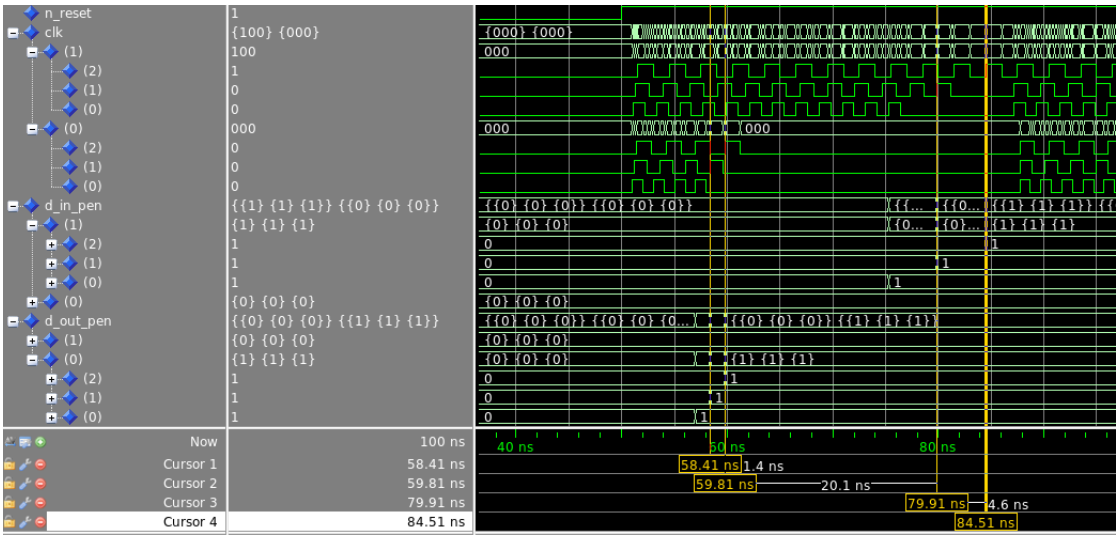


Figure 6.7: Replica synchronization with the help of a TMR message

6.4.3 Erroneous Replica

In case one replica fails, the best case transaction time of a TMR message is highly affected and can lead to tremendous performance losses as there are 3 3of4 voters in the overall system when data is sent to a poll type TMR port and even 4 when data is sent to a demand type TMR port. However as outlined above, not all of these voters need the same Δ to work properly but when one replica is dead and not responding, then all these delays sum up and the additional transaction time needed is stated in Equation 6.46 and depending on the average phase shift, this might highly increase the transaction duration.

$$\Delta_{total} = \Delta_{is\_tmr} + \Delta_{data} + \Delta_{trans\_comp} + \delta_{Rp} \tag{6.46}$$

$$\delta_{Rp} = \begin{cases} \Delta_{d_{in_{Rp}}}, & \text{when receiver} = \delta_{d_{in}} \\ 0, & \text{otherwise} \end{cases} \tag{6.47}$$

The results from Table 6.3 show the voter delays for a TMR transaction from a demand type output port to a demand type input port, where the overall TMR system had an

output port arbiter depth of 7 and 3 channels. The transaction time for the normal operation mode was 7970 ps while it was 45590 ps for the case where one replica was dead and all the signals of that replica were stuck at zero. This results in an additional duration of 37620 ps, which also matches up with the total difference from table 6.3, where only the voters were considered. The results from that table also show, that the voter in the best case needed at least 200 ps whereas due to the timeout mechanism not only the timeout delay was added in the worst case, but also additional routing delays. Note that it was assumed that the replicas are synchronous to each other. Thus it has to be considered, that those are the minimal transaction times for both cases and they might even increase in case the fastest replica is tremendously faster than the second fastest replica. Furthermore only the small request timeout was considered for the request voter. This is because since one whole replica is dead, the transaction complete voters of both replicas run into a timeout.

Operation mode	Normal operation			One dead replica			
	1. req.	voted	duration	1. req.	voted	duration	diff.
is_tmr voter	0	200	200	0	10840	10840	10640
Rp voter	4910	5110	200	15550	16450	900	700
Data voter	5450	5650	200	16790	22630	5840	5640
Ap Voter active	6280	6480	200	23260	44100	20840	20640
Ap Voter deactive	7510	7710	200	45130	45330	200	0
			1000			38620	37620

Table 6.3: Delays in ps added from voters when one Replica is dead ($timeout_{is\_tmr} = 10000, timeout_{Rp} = 1, timeout_{Data} = 5000, timeout_{Ap} = 20000$)

6.5 Area

Another important property to look at, is the area consumption. The focus here will be the area consumption of the multi channel bus compared to the area consumed by the MOGLI approach.

Due to the multiple channels a channel selection mechanism was necessary. This additional complexity also needs to more area consumtption. The receiver ports were also adapted to handle data from different channels and are more complex now. While this is adaption is a MUTEX element for demand type inputs, it is one poll type port for every single channel, when the input port is a poll type input. However all channels share one single data table, where the received data is latched and stored. This is also the reason, why not two senders are allowed to write data to the same table entry simultaneously.

While this means more area consumption for the multi channel bus compared with the MOGLI approach, it is still possible to save area when comparing it to the point-to-point approach. However this is quite difficult and only possible, when one port can handle

multiple services, as it also has to be considered that Villiger's bus capable ports [Vil05] are already more complex than the ones from Muttersbach [Mut01].

CHAPTER 7

Conclusion & Future Work

7.1 Conclusion

In this thesis a new bus architecture for pausable GALS applications was proposed. The main goal was to build a fault tolerant bus architecture with no single point of failure, that is further able to handle phase shifts between the clocks of the different replicas up to a certain degree. This is necessary as every module's replica runs with its own frequency as each has its own pausable clock.

In a first step Villiger's MOGLI [Vil05] was used as a basis for the multi channel bus that consists of multiple channels where each can handle one transaction. The multiple channels allow it to work on multiple transactions simultaneously. However the arbitration is kind of a bottleneck. With splitting up the arbitration and having multiple pipelines that feed the channels, the pressure on this bottleneck can be reduced dramatically.

In a second step this design was then triplicated for being part of a mixed criticality system, where TMR messages as well as best effort messages can be sent. While the best effort messages can be only sent within the same replica, the TMR messages are used not only for communication between the replicas but also for synchronization purposes. The input and the output ports of Villiger [Vil05] had to be adapted for TMR usage. This not only includes the voters needed to vote on the data but also releasing the ports after a timeout is reached, in order to not block the whole system, when there is no response from one replica.

In the last step the system then was analyzed regarding the fault tolerance and the impact of different faults on the system. The analysis showed that in the worst case only the replica where the fault was introduced, stopped working properly. There even the complete bus might be blocked and thus requests might also be propagated to other replicas. However their inputs are designed such that a single request can not block the input forever. Furthermore a comparison between the MOGLI and the multi channel

approach regarding the latency and throughput has been done. This has shown that the overhead of the multi channel bus lead to a higher latency. However with the hidden arbitration the throughput can be increased. Especially for systems with many output ports the throughput is increased quite noticeably, as when introducing multiple pipelines, ports can be arbitrated simultaneously and multiple transactions can go on side by side on the different channels.

7.2 Future Work

While many problems could be solved in the current work, there is still room for improvements. Especially when it comes to performance.

Channel selection In the channel selection module the most critical and time consuming part is the arbitration. In this thesis all the analyses have been done with tree arbiters. This means that conventional arbiters with two inputs are structured in a tree formation such that in the end the overall arbiter can handle more than two input ports. The question here is, if the arbitration can be done in a more efficient way. For these arbiters also area is less critical than latency they introduce.

Output port At the moment for the output ports the ones from Villiger are used and adapter is added. In the future the adapter and the output from Villiger can be merged together to one new output port. This might help to decrease the latency as well as space used.

Formal verification As outlined in chapter 5 the formal verification of the TMR system was not within the scope of this thesis but is left for further work.

Test on hardware It has not been a goal for this thesis, but one of the next steps is to actually build the architecture and test the performance on hardware to compare the results with the analysis and simulations.

List of Figures

List of Tables

Bibliography

[BC97] David S Bormann and Peter YK Cheung. Asynchronous wrapper for heterogeneous systems. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 307–314. IEEE, 1997.

[BDM01] Luca Benini and Giovanni De Micheli. Powering networks on chips: energy-efficient and reliable interconnect design for socs. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 33–38, 2001.

[BDM02] Luca Benini and Giovanni De Micheli. Networks on chip: A new paradigm for systems on chip design. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 418–419. IEEE, 2002.

[BSH75] Daniel Binder, Edward C Smith, and AB Holman. Satellite anomalies from galactic cosmic rays. *IEEE Transactions on Nuclear Science*, 22(6):2675–2680, 1975.

[Cha84] Daniel M Chapiro. Globally-asynchronous locally-synchronous systems. Technical report, Stanford Univ CA Dept of Computer Science, 1984.

[Cora] Intel Corporation. Modelsim. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html`. Accessed: 2021-11-10.

[Corb] Intel Corporation. Quartus prime. `https://www.intel.de/content/www/de/de/software/programmable/quartus-prime/overview.html`. Accessed: 2021-11-10.

[DGS04] Rostislav Dobkin, Ran Ginosar, and Christos P Sotiriou. Data synchronization issues in gals socs. In *10th International Symposium on Asynchronous Circuits and Systems, 2004. Proceedings.*, pages 170–179. IEEE, 2004.

[DT01] William J Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, pages 684–689, 2001.

[FKG09] Xin Fan, Miloš Krstić, and Eckhard Grass. Analysis and optimization of pausible clocking based gals design. In *2009 IEEE International Conference on Computer Design*, pages 358–365. IEEE, 2009.

[FS11] Gottfried Fuchs and Andreas Steininger. Vlsi implementation of a distributed algorithm for fault-tolerant clock generation. *Journal of Electrical and Computer Engineering*, 2011, 2011.

[Gin03] Ran Ginosar. Fourteen ways to fool your synchronizer. In *Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings.*, pages 89–96. IEEE, 2003.

[Gre95] Mark R Greenstreet. Implementing a stari chip. In *Proceedings of ICCD'95 International Conference on Computer Design. VLSI in Computers and Processors*, pages 38–43. IEEE, 1995.

[HS20] Florian Huemer and Andreas Steininger. Timing domain crossing using muller pipelines. In *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 44–53. IEEE, 2020.

[JT+03] Axel Jantsch, Hannu Tenhunen, et al. *Networks on chip*, volume 396. Springer, 2003.

[KFK15] Ben Keller, Matthew Fojtik, and Brucek Khailany. A pausible bisynchronous fifo for gals systems. In *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 1–8. IEEE, 2015.

[Kil07] Steve Kilts. *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons, 2007.

[KJS+02] Shashi Kumar, Axel Jantsch, J-P Soininen, Martti Forsell, Mikael Millberg, Johny Oberg, Kari Tiensyrja, and Ahmed Hemani. A network on chip architecture and design methodology. In *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, pages 117–124. IEEE, 2002.

[KPWK03] Joep Kessels, Ad Peeters, Paul Wielage, and Suk-Jin Kim. Clock synchronization through handshake signalling. *Microprocessors and Microsystems*, 27(9):447–460, 2003.

[KSS+15] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christoph Müller, Kees Goossens, and Jens Sparsø. Argo: A real-time network-on-chip architecture with an efficient gals implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, 2015.

[Lec14] Jakob Lechner. *Building robust GALS circuits; fault-tolerant and variation-aware design. Techniques for reliable circuit operation*. PhD thesis, TU Wien, 2014.

[LV62] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2):200–209, 1962.

[MCS04] Joycee Mekie, Supratik Chakraborty, and Dinesh K Sharma. Evaluation of pausible clocking for interfacing high speed ip cores in gals framework. In *17th International Conference on VLSI Design. Proceedings.*, pages 559–564. IEEE, 2004.

[MTMR02] Simon Moore, George Taylor, Robert Mullins, and Peter Robinson. Point to point gals interconnect. In *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems*, pages 69–75. IEEE, 2002.

[Mut01] Jens Muttersbach. Globally-Asynchronous Architectures for VLSI Systems. *University of Cologne*, 2001.

[MVF00] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000)(Cat. No. PR00586)*, pages 52–59. IEEE, 2000.

[NS15] Robert Najvirt and Andreas Steininger. How to synchronize a pausible clock to a reference. In *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 9–16. IEEE, 2015.

[NS20] Robert Najvirt and Andreas Steininger. Performance limits in clock domain crossing: Choosing synchronization fifo parameters, Internal work, TU Wien, 2020.

[PSM07] Ivan Poliakov, Danil Sokolov, and Andrey Mokhov. Workcraft: a static data flow structure editing, visualisation and analysis tool. In *International Conference on Application and Theory of Petri Nets*, pages 505–514. Springer, 2007.

[SF01] Ivan Sutherland and Scott Fairbanks. Gasp: A minimal fifo control. In *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, pages 46–53. IEEE, 2001.

[SM00] Allen E Sjogren and Chris J Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):573–583, 2000.

[SMC80] Charles L Seitz, C Mead, and L Conway. System timing. *Introduction to VLSI systems*, pages 218–262, 1980.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[VHD19] Ieee standard for vhdl language reference manual. *IEEE Std 1076-2019*, pages 1–673, 2019.

[Vil05] Thomas Villiger. *Multi-point interconnects for globally-asynchronous locally-synchronous systems*, volume 157. ETH Zurich, 2005.

[YD96] Kenneth Y Yun and Ryan P Donohue. Pausible clocking: A first step toward heterogeneous systems. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 118–123. IEEE, 1996.

[ZCM+96] James F Ziegler, Huntington W Curtis, Hans P Muhlfeld, Charles J Montrose, B Chin, Michael Nicewicz, CA Russell, Wen Y Wang, Leo B Freeman, P Hosier, et al. Ibm experiments in soft fails in computer electronics (1978–1994). *IBM journal of research and development*, 40(1):3–18, 1996.

[ZSG+13] Guangda Zhang, Wei Song, Jim D Garside, Javier Navaridas, and Zhiying Wang. Transient fault tolerant qdi interconnects using redundant check code. In *2013 Euromicro Conference on Digital System Design*, pages 3–10. IEEE, 2013.

Appendix

Simulation results

In the following, the simulation results of the most important failure cases are presented. Despite the address fault all transactions were done from a demand output port to a demand input port. Where the demand input port initiates the transaction after 4 clock cycles and the receiver is ready to receive the data after 12 clock cycles. The sender clocks have a period of 2, 2.4 and 2.8 ns whereas the receiver clocks have a clock period of 2.2, 2.6 and 3 ns.
The voter timeouts for the simulation were:
$timeout_{is\_tmr} = 10000ps$
$timeout_{Rp} = 20000ps$
$timeout_{Data_d} = 5000ps$
$timeout_{Data_p} = 15000ps$
$timeout_{Ap} = 20000ps$
Furthermore all faults were introduced in replica zero and no optional buffer pipelines where used.

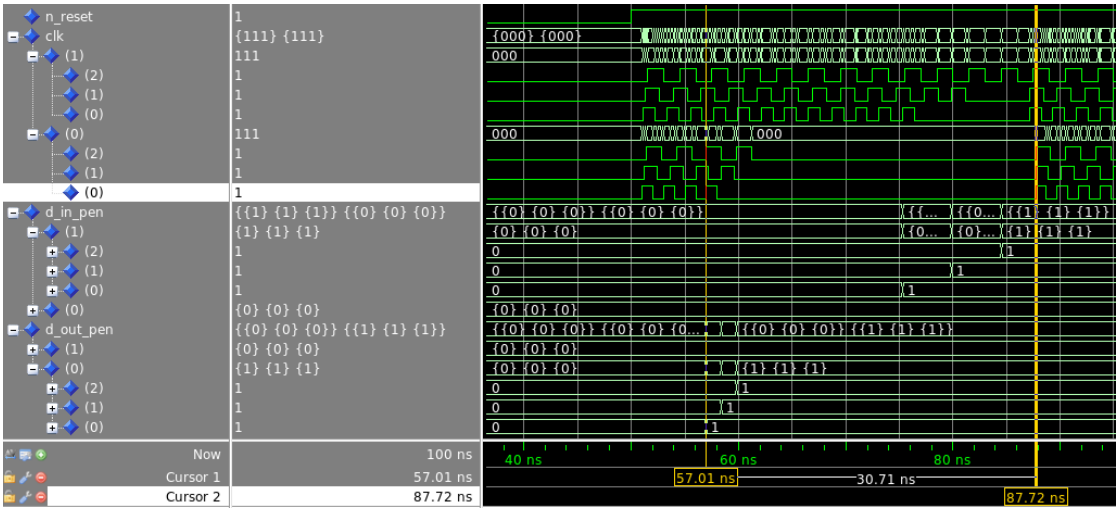


Figure 1: Normal data transmission

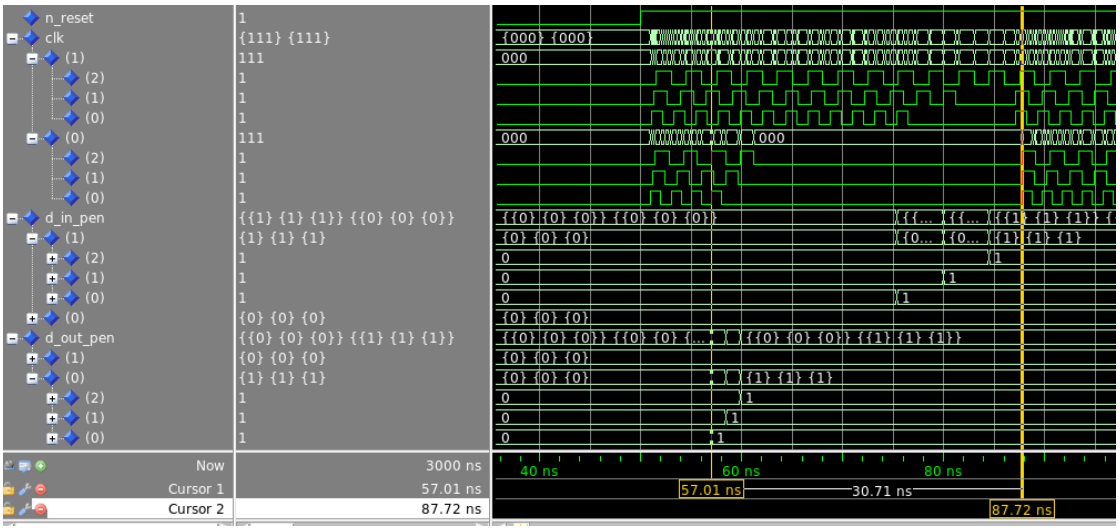


Figure 2: Bit flip in data word

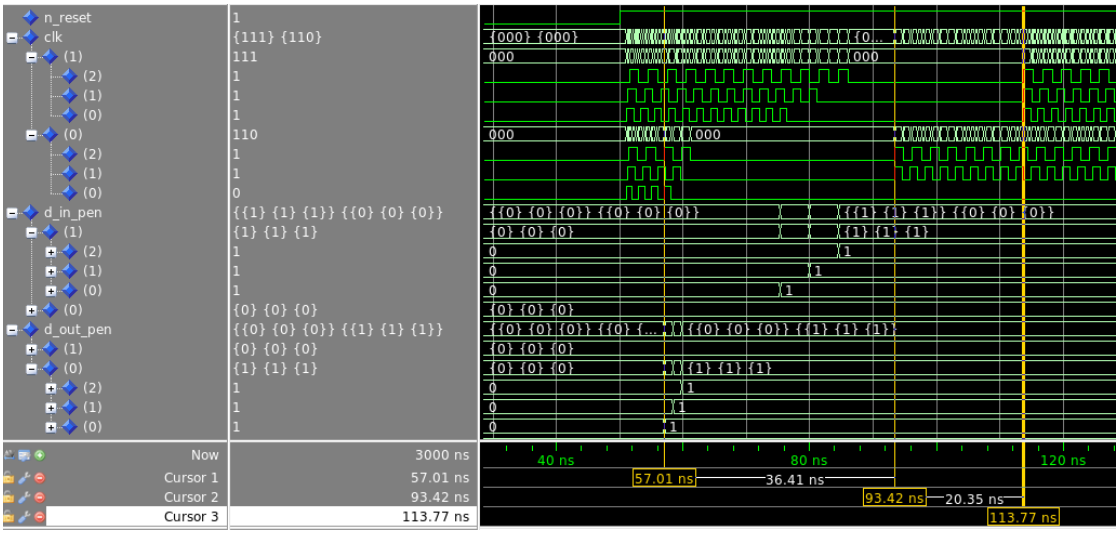


Figure 3: Bit flip from one to zero in TMR flag and no corresponding input port of non TMR message

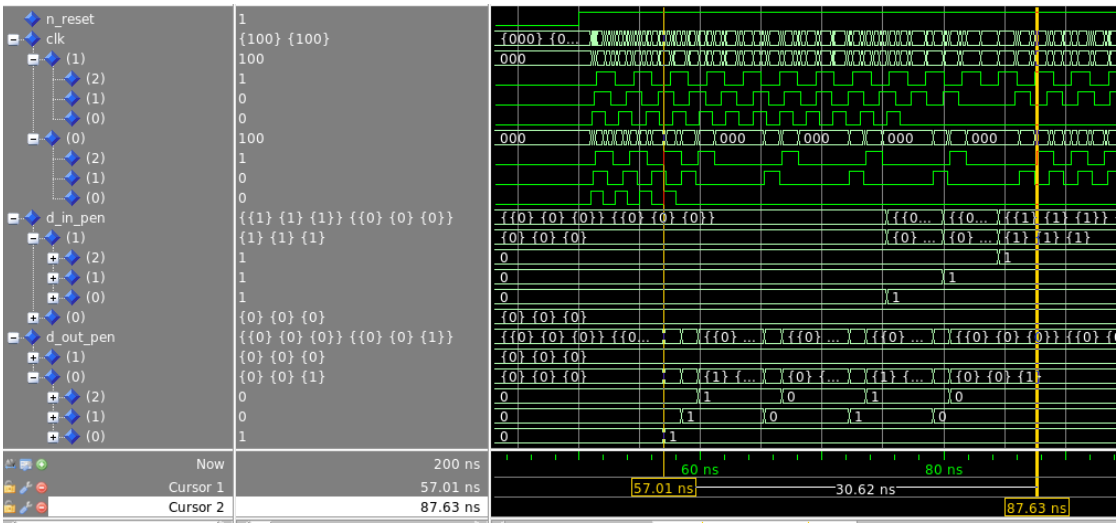


Figure 4: Bit flip of TMR flag from zero to one, whit no receiver addressable

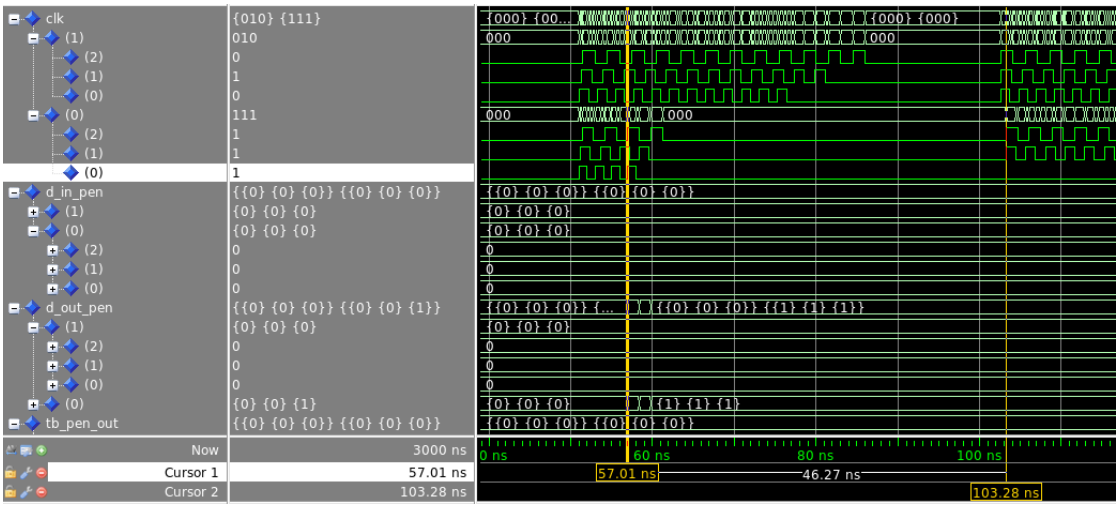


Figure 5: Bit flip in address of data word sent to a poll input port

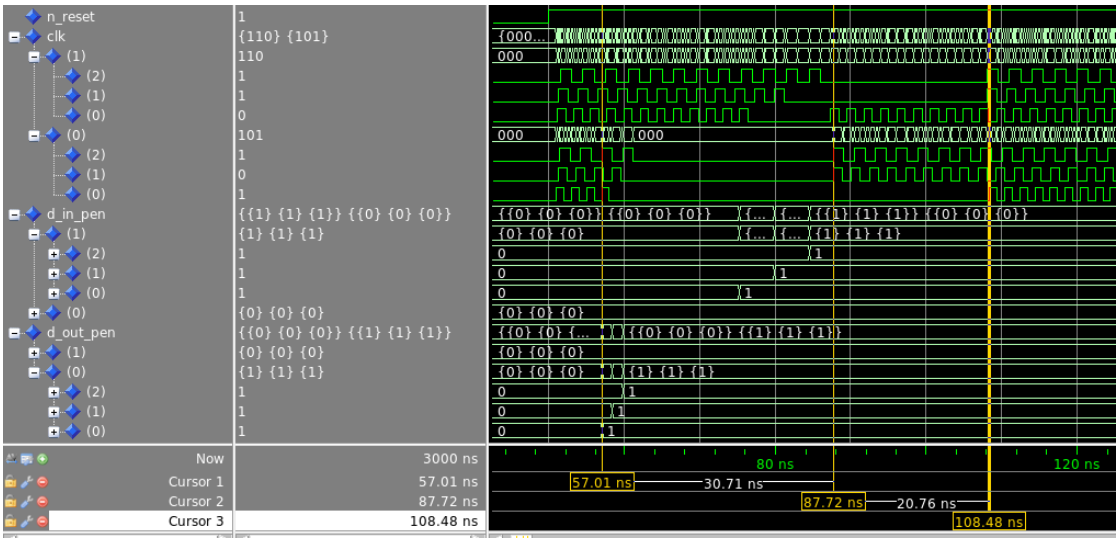


Figure 6: Bit flip of one request signal to zero

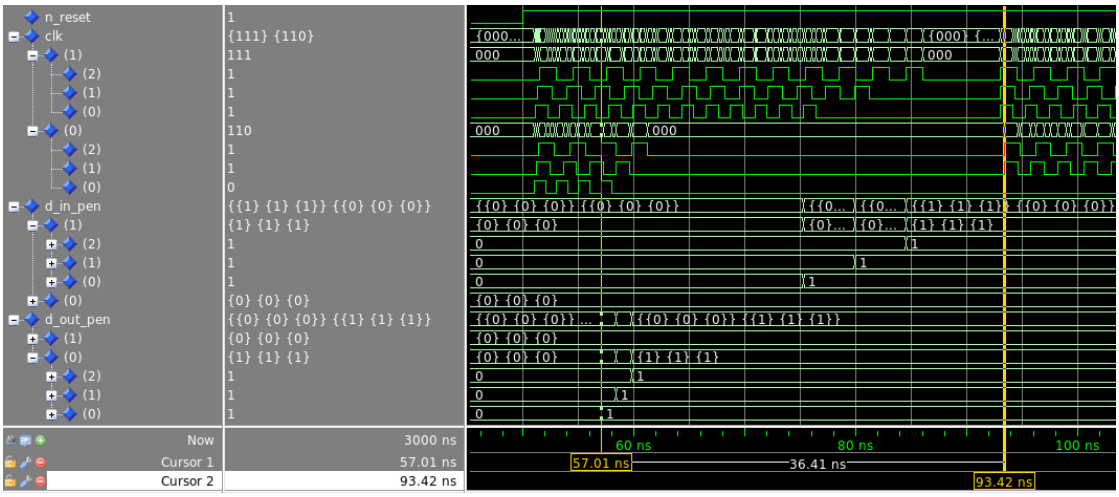


Figure 7: Bit flip of all request signals of one replica

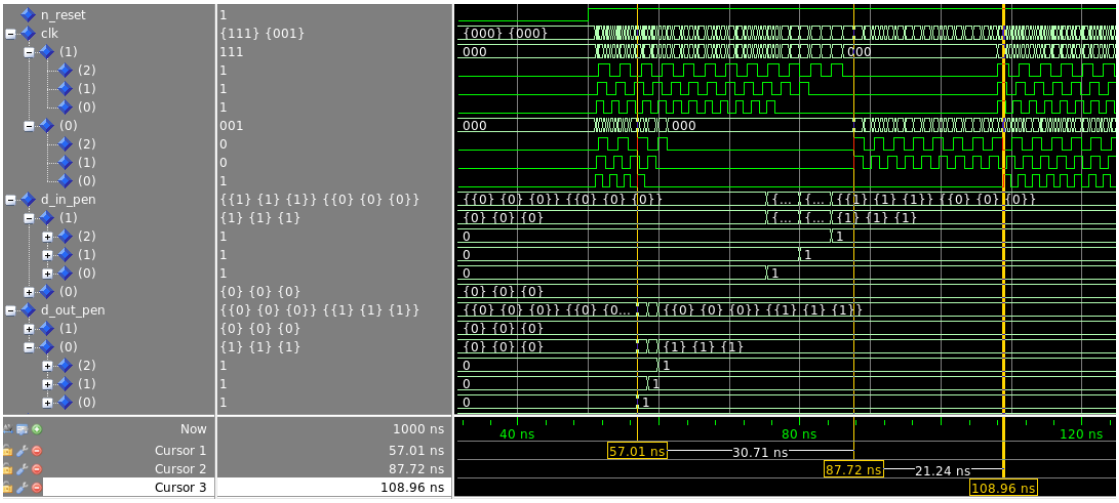


Figure 8: Bit flip of one request signal to one

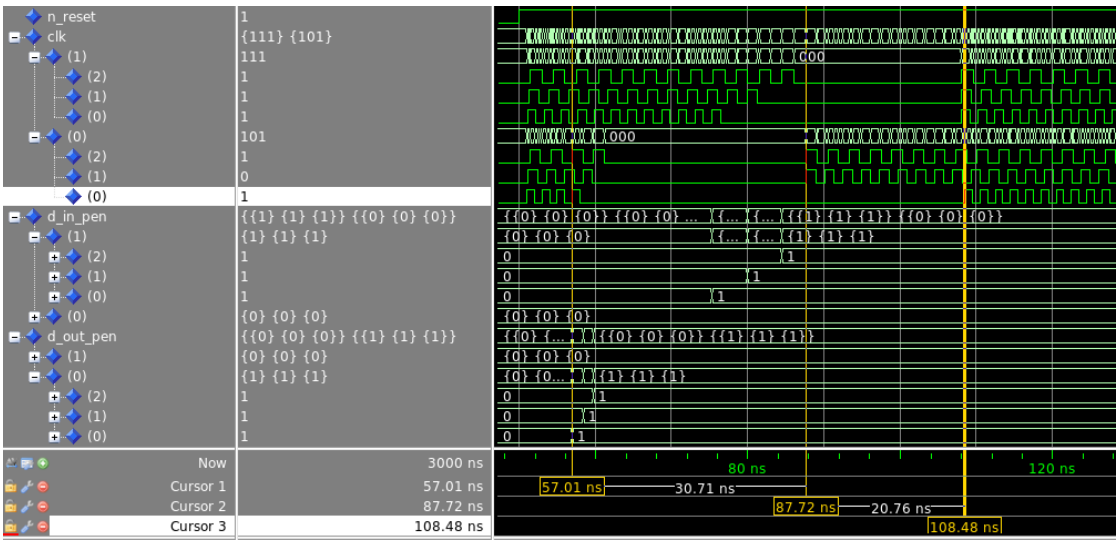


Figure 9: Bit flip of one acknowledge signal to zero

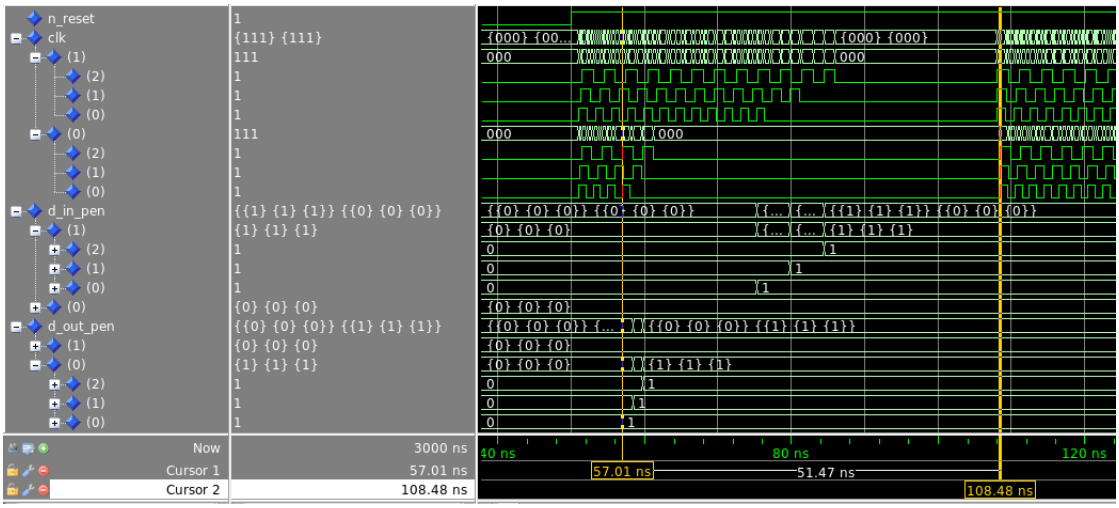


Figure 10: Bit flip of all acknowledge signals of one replica to zero

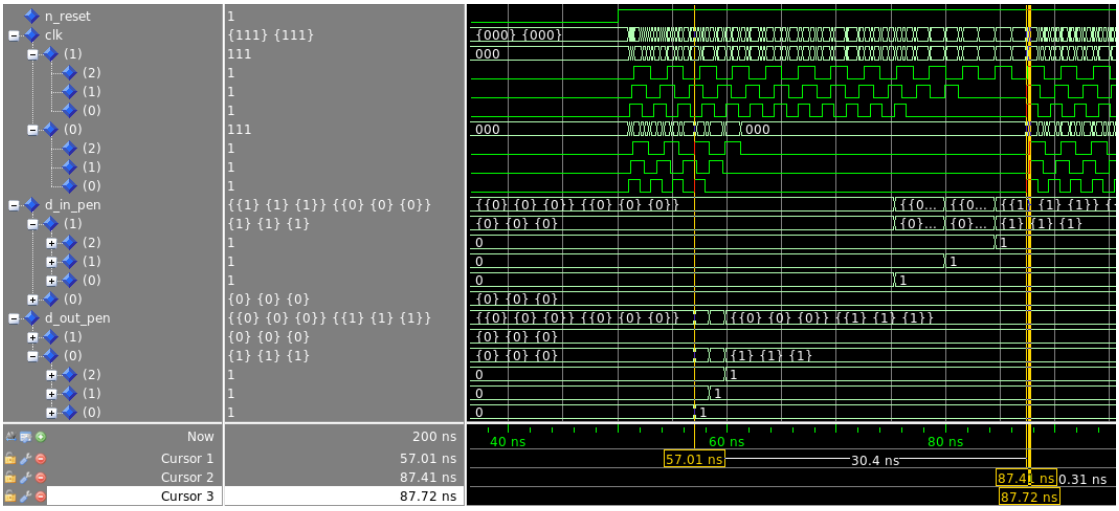


Figure 11: Bit flip of TMR flag to zero

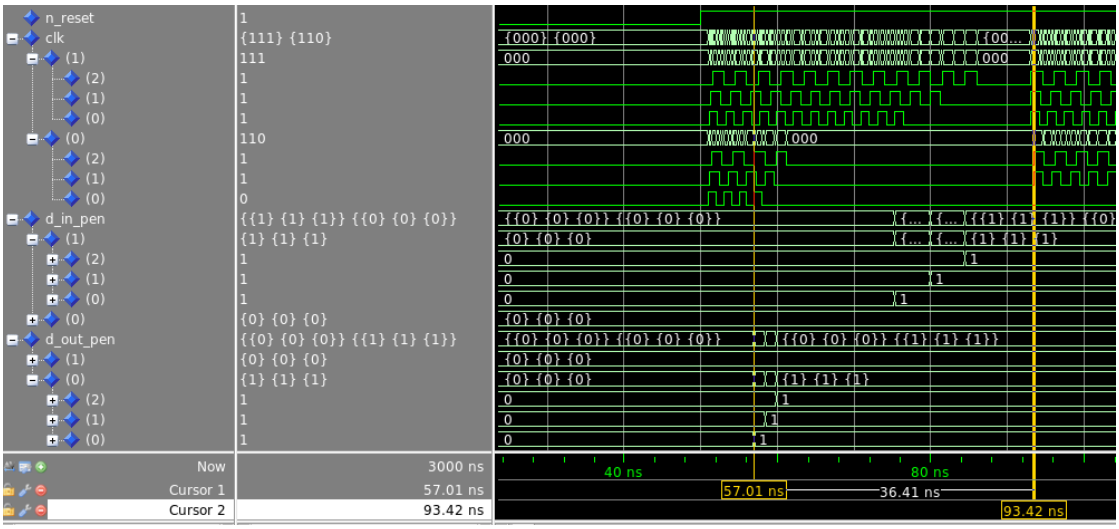


Figure 12: Bit flip in TMR voter

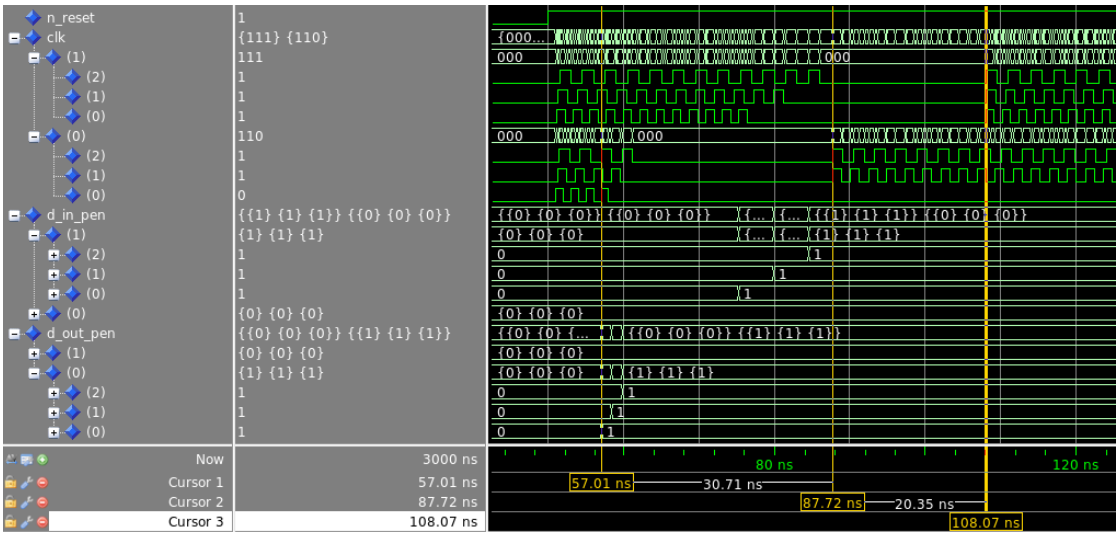


Figure 13: Bit flip to zero in transaction complete voter that is not resolved

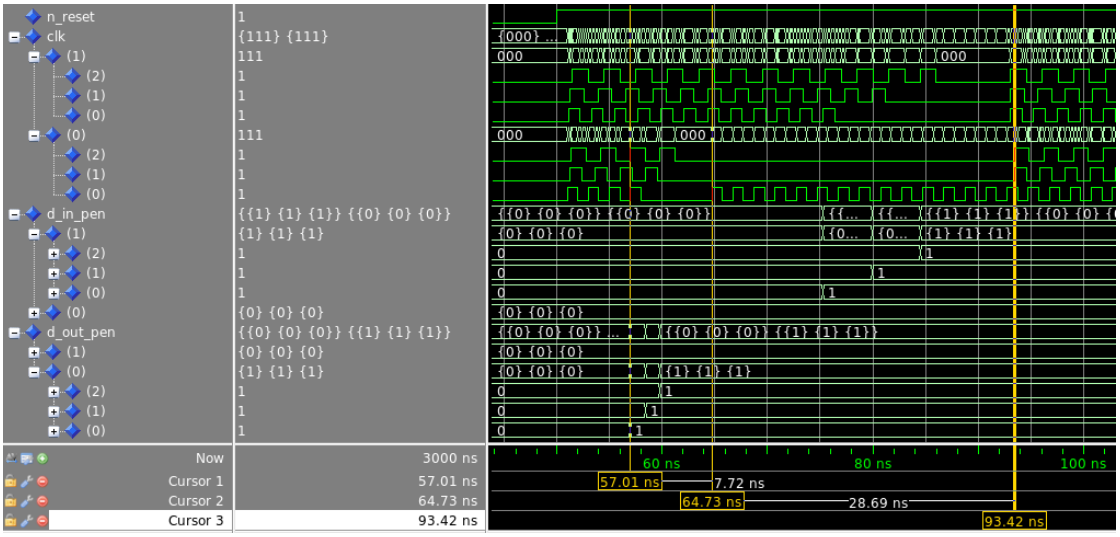


Figure 14: Bit flip glitch with in transaction complete voter before data is acknowledged

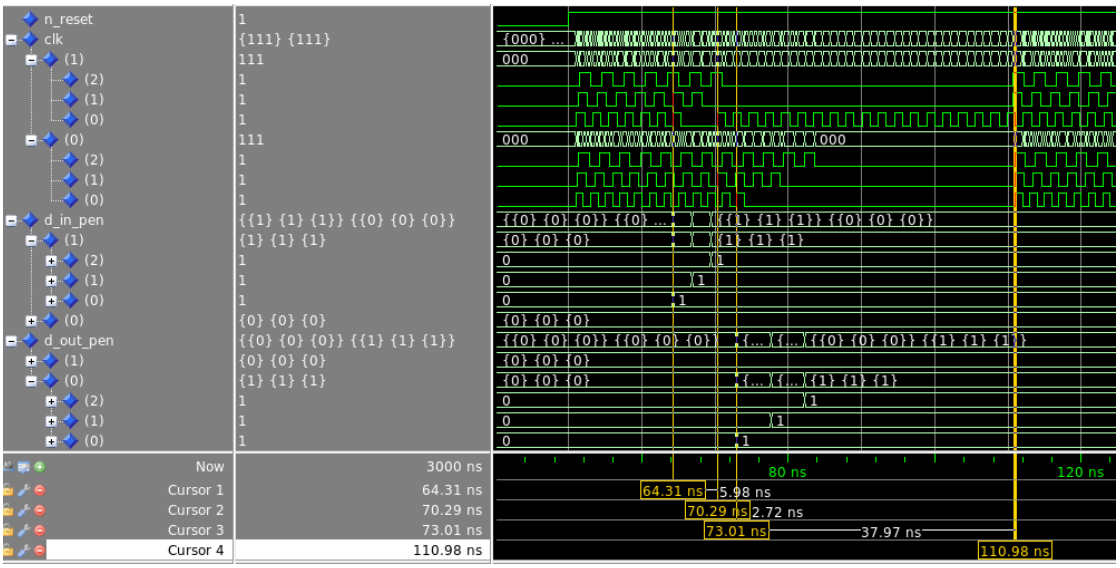


Figure 15: Bit flip glitch of 5 ns at the rp voter

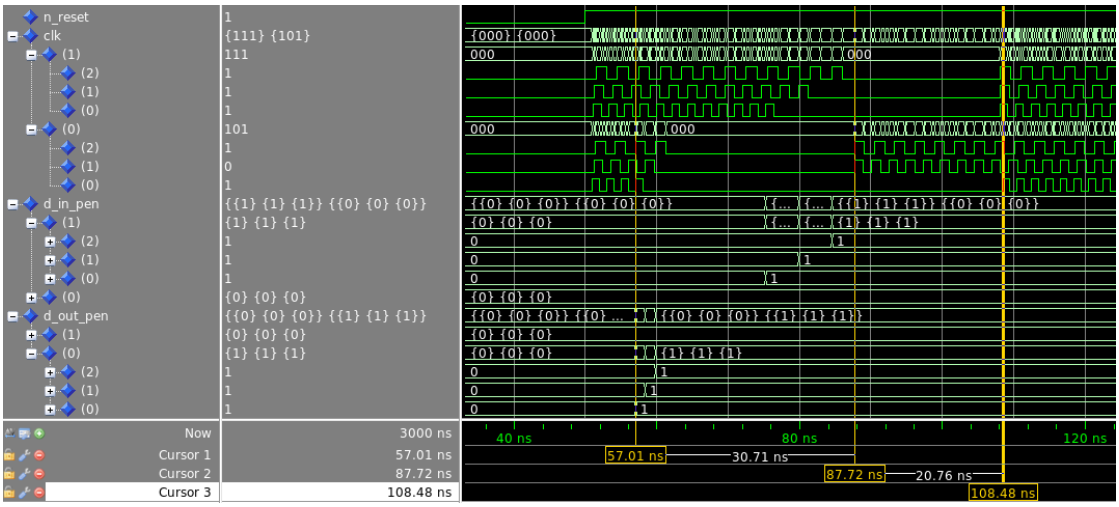


Figure 16: Stuck at zero of an acknowledge signal

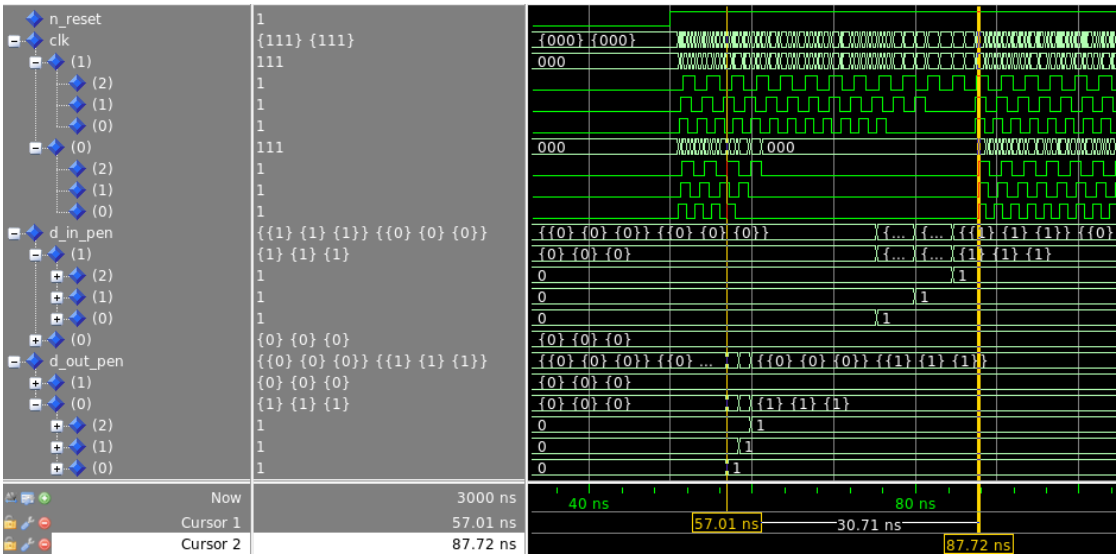


Figure 17: Stuck at one of an acknowledge signal

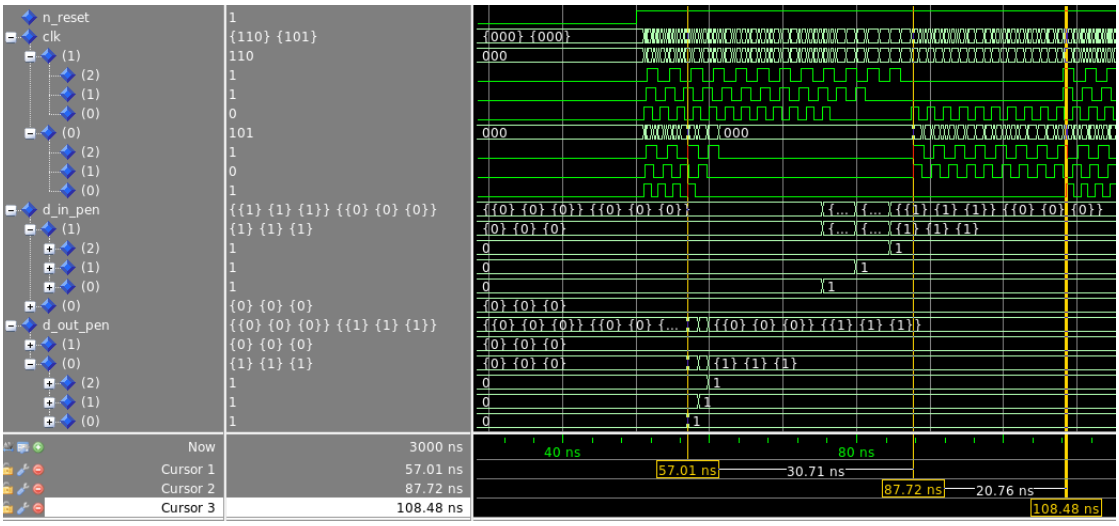


Figure 18: Stuck at zero of a request signal

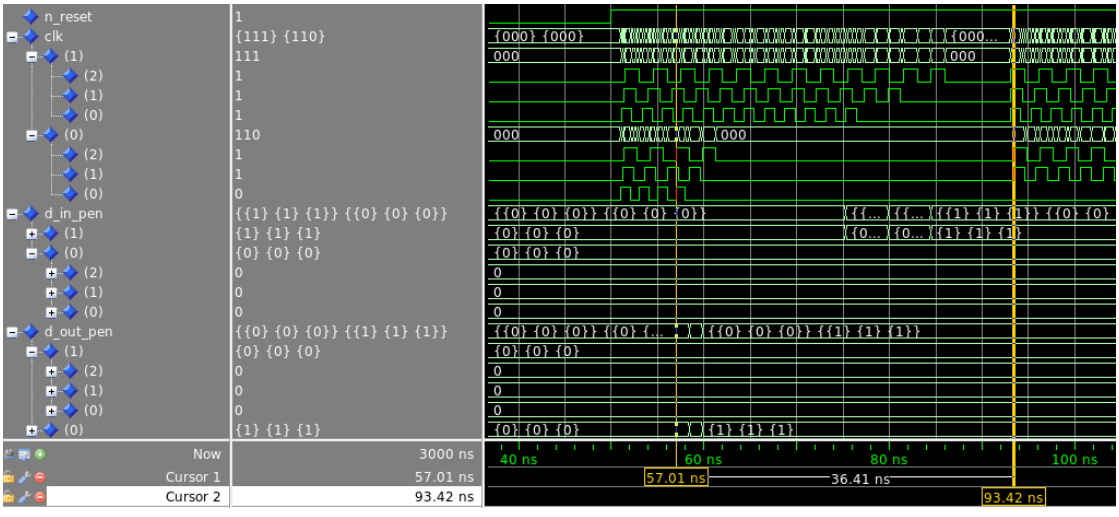


Figure 19: Stuck at zero of all request signals of one replica

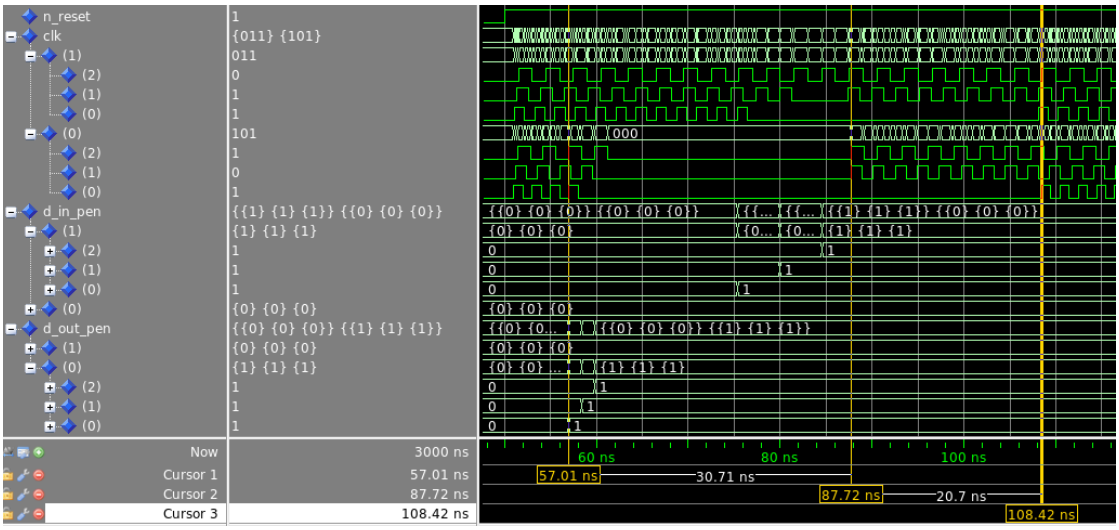


Figure 20: Stuck at one of a request signal

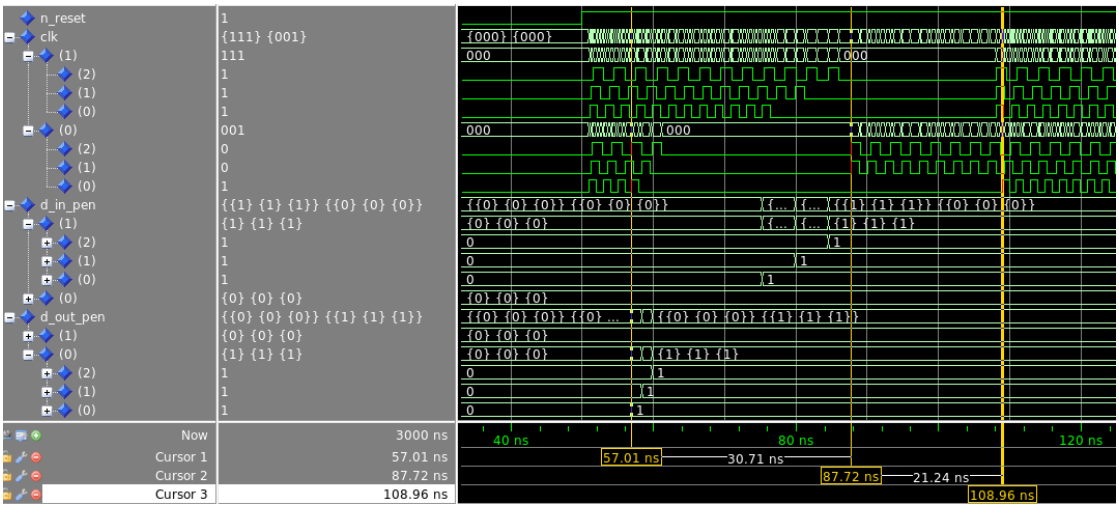


Figure 21: Stuck at one of all request signals of one replica

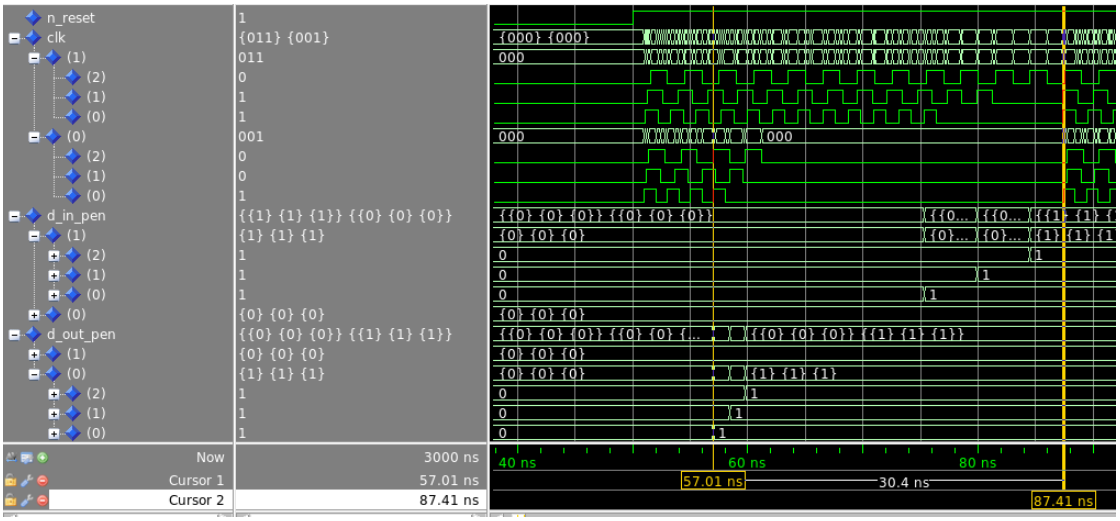


Figure 22: Stuck at zero fault of one *is_tmr* signal

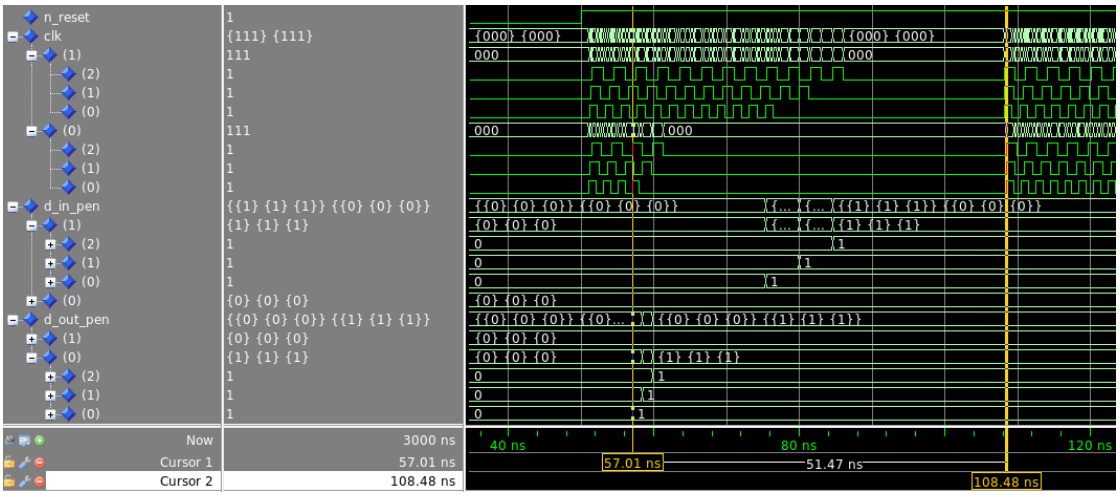


Figure 23: Stuck at zero fault of data voter

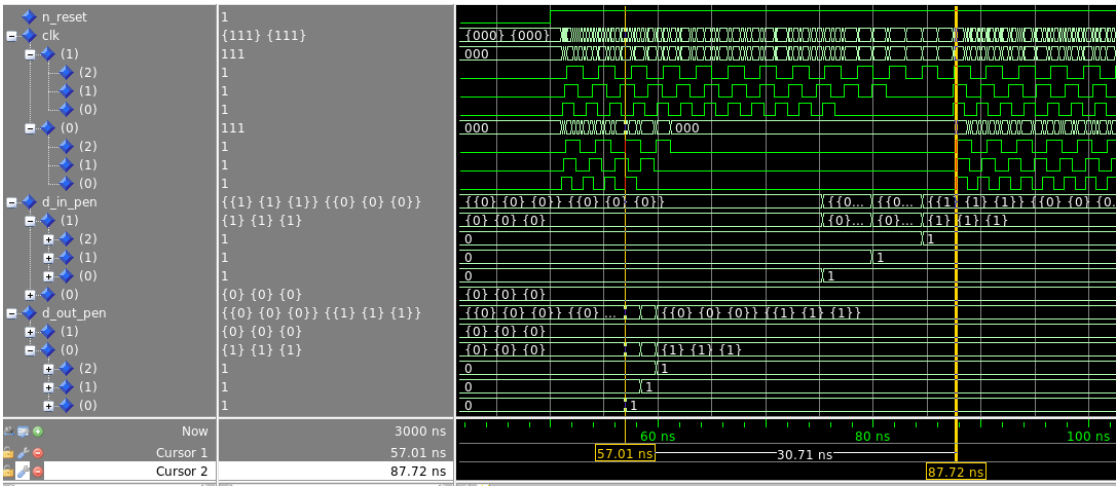


Figure 24: Stuck at one fault of data voter

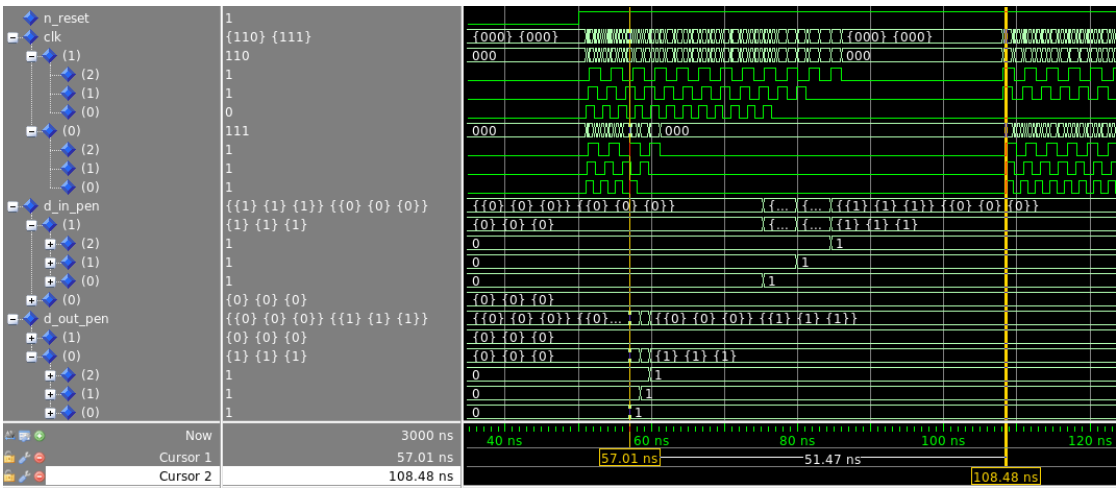


Figure 25: Stuck at zero fault of request voter at demand input port

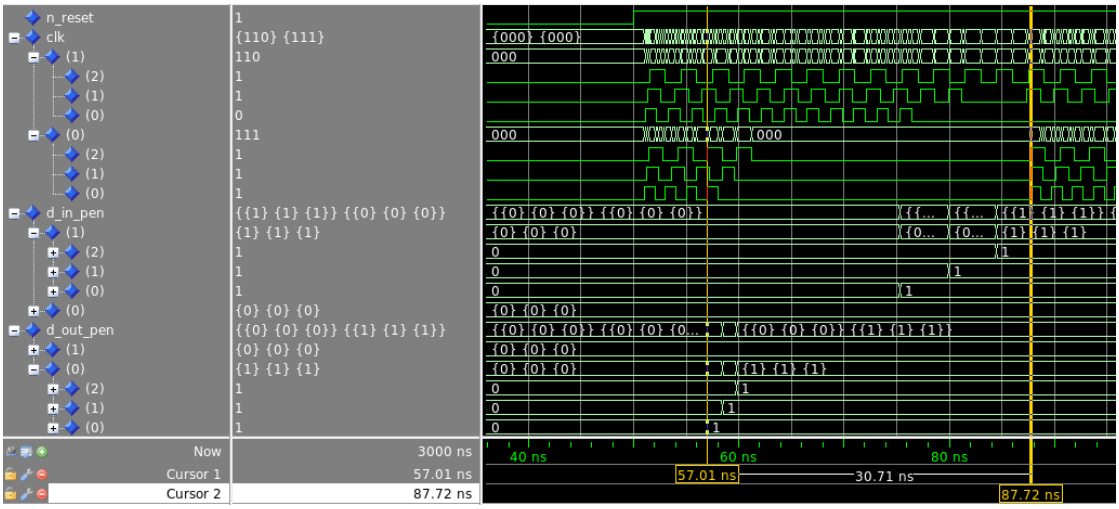


Figure 26: Stuck at one fault of request voter at demand input port