FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Non-Blocking Fault-Tolerant Asynchronous Networks-on-Chip Router

PhD THESIS

submitted in partial fulfillment of the requirements of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Syed Rameez Naqvi
Registration Number 0928544

to the Faculty of Informatics
at the Vienna University of Technology

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

External reviewers:
Prof. Dr.-Ing. Eckhard Grass. Institute for Informatics, Humboldt-University of Berlin, Germany.
Ao. Prof. Martin Schöberl. Dept. of Applied Math. and Comp. Science, DTU, Denmark.

Wien, 15.10.2013

(Signature of Author) (Signature of Advisor)

Declaration of Authorship

Syed Rameez Naqvi
Längenfeldgasse 22, Top 16, 1120 Vienna, Austria

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

_____ _____

(Place, Date) (Signature of Author)

Acknowledgments

First and foremost I am sincerely grateful to Prof. Andreas Steininger who has not just been a supervisor but also a mentor to me. His guidance, encouragement and patience kept me going throughout my studies here at the Vienna University of Technology. He has always given me immense courage on a personal as well as a professional level and I owe him for inspiring me on various levels of study.

I am also obliged to my other colleagues, especially Robert Najvirt, Jakob Lechner, and Varadan Savulimedu Veeravalli, who coauthored most of my research papers and always helped to add that extra tinge of flavor to the spirit of my work. Our long discussions and hard talks have been very insightful and opened many new horizons for me regarding my research. I will miss all our fun times and lunches together.

I cannot thank Prof. Hannes Werthner, enough for having faith in me and considering me a rightful candidate for this prestigious scholarship at the PhD School of Informatics. My deepest regards to Mrs. Clarissa Schmid, Mrs. Edeltraud Sommer and Ms. Maria del Carmen Calatrava Moreno, whose cooperation from day one has added tremendously to ease my concerns regarding any and all administrative matters and helped me concentrate better on my studies. I would also like to pay my gratitude to Heinz Deinhart and Karl Malle, the system and network administrators, for providing the required tools and software, by and large, on a very short notice. I truly appreciate the diligence with which all of you work day in and day out as well as your resourcefulness in ensuring that all researchers stay up-to-date in terms of the network status.

Last but not the least I have no words to thank my family, especially my mother, and my wife for all their support and encouragement. You are the reason I kept going through the lows and stayed grounded through the highs.

This is merely a humble effort to express my gratitude towards everything you all have contributed in helping me realize my dream. I will always cherish the memories and try my best not to let you down to the best of my potential.

Kurzfassung

Die Gesamtleistung der heutigen komplexen Systems-on-Chip (SoCs) wird entscheidend durch die Leistungsfähigkeit der Networks-on-Chip (NoCs) bestimmt. Die zunehmende Verkleinerung der Halbleiter-Strukturgrößen ermöglicht hier zwar immer höhere Taktraten, stellt jedoch gleichzeitig das übliche global synchrone Paradigma vor wachsende Herausforderungen. Hierzu zählt die Verteilung des Taktsignals über den gesamten Chip ebenso wie die Beherrschung der signifikanten Variationen der Signallaufzeiten. Im Gegensatz dazu basieren asynchrone Schaltungen auf Handshakes, womit sie Probleme mit dem Takt elegant vermeiden. Gleichzeitig erlauben sie einen effizienten Umgang mit Prozessvariationen und sogar dynamisch auftretenden Veränderungen von Laufzeitparametern. Das Interesse an asynchronen Schaltungen nimmt daher stark zu, und auch wir konzentrieren uns in dieser Arbeit auf asynchrone NoCs.

Ein entscheidender Vorteil von NoCs gegenüber traditionellen Busstrukturen ist ihre Fähigkeit, über verschiedene Pfade parallel Nachrichten zu übertragen. Das Blockieren eines Pfades ist in diesem Kontext unerwünscht, da es mit gravierenden Leistungseinbußen und dem Verlust der Echtzeitfähigkeit verbunden ist. Demgemäß beschäftigt sich die Literatur auch ausführlich mit nicht-blockierendem Verhalten, allerdings nicht immer in korrekter Weise. So konnten wir zeigen, dass einige existierende Lösungen einen sicheren Datenaustausch nicht immer garantieren können. Ein Ergebnis unserer Arbeit ist daher ein Framework, welches notwendige Bedingungen zur Konstruktion eines nicht-blockierenden NoCs definiert. Wir zeigen die Korrektheit unseres Frameworks anhand des Entwurfes eines neuen NoC Protokolls, welches nicht nur alle funktionalen Anforderungen erfüllt, sondern auch den Leistungsverbrauch auf langen Verbindungen reduziert und die Anforderungen an die Bandbreite lockert.

Eine weitere Folge der Miniaturisierung ist eine erhöhte Anfälligkeit von Chips gegenüber transienten Fehlern. Ursache dafür sind kleinere Transistor-Geometrien sowie niedrigere Spannungspegel, was zu geringeren kritischen Ladungen führt. Dem entsprechend wächst auch die Bedeutung von Fehlertoleranz. Asynchrone NoCs enthalten allerdings einige Komponenten, für welche die konventionelle Fehlertoleranzmethode der Replikation nicht bzw. nicht effizient eingesetzt werden kann. Ein Beispiel hierfür sind Arbiter, die auf nicht deterministische Weise den Zugriff auf geteilte Ressourcen regeln. Das zweite wesentliche Ergebnis dieser Arbeit ist daher der systematische Schutz einiger dieser Komponenten, insbesondere eines Arbiters, vor transienten Fehlern. Außerdem präsentieren wir das Design eines vollständigen fehlertoleranten Routers, welcher zudem einen hochperformantenArbiter enthält. Zur Evaluierung unseres Designs verwenden wir funktionale Verifikation anhand von teilautomatisierten post-layout Simulationen, gepaart mit formaler Verifikation mittels Model Checking.

Abstract

It is well understood that the throughput of Networks-on-Chip (NoCs) is decisive for the performance of today's complex Systems-on-Chip (SoCs). On the one hand, proceeding miniaturization has allowed these systems to operate at ever increasing clock rates, on the other hand, however, the globally synchronous designs face certain challenges that are difficult to overcome for recent technology nodes. This includes the distribution of clocks across a complete chip, and robustness against delay variations. The handshake based style of control flow in asynchronous communication style naturally eliminates the need for a global clock, and at the same time provides an inherent ability to adapt to uncertainties and even dynamic changes of timing parameters. Because of these reasons they are receiving increasing attention, and for the same reason we will also concentrate on asynchronous NoCs in this work.

A crucial advantage of NoCs over traditional bus structures is the ability to perform several independent message transfers in parallel, namely along different routes. In this situation blocking of a link is highly undesired, as it severely degrades performance and real time capabilities of a NoC. A very important property of a NoC is therefore non-blocking behavior. The latter is widely addressed in literature, but unfortunately not always correctly understood and presented. We will show that a few existing solutions do not guarantee a safe data exchange between two communicating entities. One contribution of our work, therefore, is to present a framework that precisely elaborates the minimum requirements of building a nonblocking NoC. We also demonstrate the correctness of our framework by proposing a novel solution that not just satisfies all the functional requirements, but reduces the power consumption on long interconnects, and relaxes the bandwidth requirements.

Another undesired consequence of the miniaturization process is the higher susceptibility of VLSI circuits to transient faults, which is due to the smaller geometries and lower supply voltages which in turn reduce the critical charge. Fault tolerance is therefore predicted to become a crucial property in context with future technologies. As far as asynchronous NoCs are concerned, they comprise several such circuits that cannot be made fault-tolerant by using conventional replication techniques. One such circuit is an arbiter that allows resource sharing in a nondeterministic manner. Therefore the second major contribution of this work is to systematically harden a few of those circuits, including an arbiter cell. Other than these, we also present the design of a complete fault-tolerant asynchronous router, which in addition promises high speed resource sharing capability. Our evaluation is supported with formal verification using model checking, and post layout functional verification based on Modelsim scripts.

Contents

List of Figures

List of Tables

Acronyms

ANoC	Asynchronous Networks-on-Chip
BER	Bit Error Rate
CMP	Chip Multiprocessor
CRC	Cyclic Redundancy Check
DI	Delay Insensitive
ECC	Error Control Code
E2E	End-to-End
FEC	Forward Error Correction
Flits	Flow-control Units
FSM	Finite State Machine
FT	Fault-Tolerance/Tolerant
GALS	Globally Asynchronous Locally Synchronous
HBH	Hop-by-Hop
IO	Input/Output
IP	Intellectual Property
LEDR	Level Encoded Dual Rail
MC	Muller C-element
MUTEX	Mutual Exclusion
MUX	Multiplexer
NCL	Null Convention Logic

NoC	Networks on Chip
NRZ	Non Return to Zero
PE	Processing Element
PVT	Process, Voltage, and Temperature
QDI	Quasi Delay Insensitive
RTZ	Return to Zero
SET	Single Event Transient
SER	Soft Error Rate
SEU	Single Event Upsets
SG	State Graph
SI	Speed Independent
SoC	Systems on Chip
SPL	Shared Physical Link
STG	Signal Transition Graph
S2S	Switch-to-Switch
TAC	Tree Arbiter Cell
TFF	Toggle Flip-Flop
VC	Virtual Channel
VLSI	Very Large Scale Integration

CHAPTER 1

Introduction

While we have witnessed a tremendous reduction in feature sizes during the last decade, the basic techniques behind building high speed processors, such as pipelining and superscalar approaches, did not prove proportionally beneficial [44]. In the quest for high performance, the designers were willing to integrate many computing resources on a single die, rather than increasing the complexity of just one. These computing resources included intellectual property (IP) cores from various manufacturers, or identical processing elements (PE), thus leading to *systems-on-chip* (SoC) [74] and *chip multiprocessors* (CMP) [68] platforms.

However, to make such systems a reality and allow leveraging their full potential, the designers had, and still have to overcome certain bottlenecks, perhaps the most important of which was the slow interconnection of the PEs. For the global interconnect it soon turned out that the conventional shared bus topology was inefficient, since it did not scale suitably. Consequently, it had to be replaced with something more robust and flexible, which would allow several independent message transfers in parallel, and might be programmed according to the application requirements, the so called Networks-on-Chip (NoC) [12, 31, 96]. An NoC provides a generic regular structure, comprising a network of routers, which facilitates more flexibility and high throughput, and is also amenable to 3D VLSI technology [60, 112, 145].

While the advent of this programmable interconnect solution has somewhat addressed the bandwidth issues, the constantly growing number of function units on a chip has given rise to some other critical challenges, such as *clock distribution* in the GHz range. In order to make a clock tree with sufficiently low skew possible at all, a significant share of the power budget must be spent for strong clock drivers [116], thereby giving rise to another grave concern, *energy efficiency*. Secondly, a fixed clock rate determined by a corner case analysis of the maximum propagation delays is considered overly pessimistic in face of the significant variances of chip parameters, specifically timing parameters. An automated balanced multiple clock domain H-tree [123] was recently proposed to reduce clock skew and PVT variations impact, but unfortunately no work guarantees their complete elimination.

Issues such as energy efficiency and clock distribution proved to be cumbersome to achieve in a globally synchronous design style. Instead, the globally asynchronous locally synchronous

design style [22] has become popular, allowing fully synchronous design within locally constrained islands that are, in turn, connected in an asynchronous fashion. Often an NoC connects function units of the SoC that run at their individual clock rates optimized for their respective purpose. In order to circumvent performance penalties incurred by the synchronizers required at the timing domain boundaries between the function units and the routers within the NoC [80, 97, 151], an asynchronous implementation of those routers suggests itself. As a further benefit, with their demand driven activity, asynchronous designs facilitate a better energy efficiency than their synchronous counterparts [48, 139]. And finally, the delay-insensitive asynchronous design styles exhibit great flexibility with respect to PVT variations [21, 32], thus solving a problem that is currently considered most severe in synchronous nanoscale implementations. In view of these advantages we put our focus on asynchronous NoCs (ANoCs) in the rest of this work.

1.1 Motivation

During our literature survey for a systematic design of fault-tolerant (FT) ANoCs, we came across a few concepts related to ANoCs that we felt were either not sufficiently addressed, or somehow misunderstood despite being used formerly on a number of occasions. We believe they deserved a thorough investigation, and providing (one of the possible) solutions to those concerns makes the core message of this work. In the following we briefly overview each of them in turn.

1.1.1 Non-blocking Behavior

A crucial advantage of NoCs over traditional bus structures is the ability to perform several independent message transfers in parallel, namely along different routes. This requires routers and the interconnect links between them to be flexibly available for different transfers. In this situation blocking of a router or link is highly undesired as it severely degrades performance and real time capabilities of the NoC. Blocking may occur due to the unavailability or slow speed of the destination, due to overload of a router (too many incoming messages from many directions), in which case back-pressure may keep further routers or links along the path to the origin blocked and thus unavailable for other transfers, or due to the failure of a path.

As shall be discussed in detail in chapters 2 and 3, several measures can be taken to mitigate this risk, one of them being the concept of virtual channels, where the physical link is shared among several transfers in a time division manner. The virtual channels are always augmented with another mechanism called flow-control, responsible for the backpressure management. Although known to the community for a long time, there is no formally defined set of requirements for building a safe flow control scheme. As a result of this lack of knowledge, their design can get deceptive, not fulfilling the nonblocking criterion. In the coming chapters we shall highlight a few examples in support of our argument. Therefore, we thought a framework that could (somehow formally) elaborate the minimum requirements of building a safe flow control mechanism was much needed by the NoC community.

1.1.2 Reliability

An undesired consequence of the miniaturization process is the higher susceptibility of VLSI circuits to two types of faults: 1) those that occur due to variations in manufacturing process – they normally lead to alteration in the circuit's delays, and 2) those that occur due to variations in the environment, such as temperature, electrical noise, and most importantly radiation particle hits – they typically cause undesired voltage pulses of very limited length, called transient faults or *single event transients* (SET).

At deep submicron technology, due to the smaller geometries and lower supply voltages which in turn reduce the critical charge, SETs are becoming more frequent. When a high energy particle (alpha particles or neutrons from atmosphere) strikes a slicon substrate, it can generate electron-hole pairs inducing current, which may easily upset the small critical charge, thereby changing the output of the gate [62]. In case an SET somehow gets stored in one or more storage elements, it is said to be a *single event upset* (SEU) [108] or a *soft fault* [54]. An SEU has a potential to propagate to other parts of the circuit as well, giving rise to persistent errors [87] that may remain in the logic circuit indefinitely. According to a few research works [98, 136], in today's deep submicron technology, around 80% of the total errors in digital circuits are due to SETs. The *soft error rate* (SER) in digital circuits in general is thus rising, and this concerns the NoC as well.

Since the operation of the asynchronous circuits does not rely on stringent timing assumptions, the latter by default (usually) proves more robust against the first type of faults than the synchronous circuits [8]. However, the other type of faults is equally critical for the operation of both asynchronous, and synchronous circuits.

Because of the absence of the clock signal, a fault in an ANoC may exhibit a behavior that would not be observed in the synchronous equivalent, thereby making the current fault tolerant schemes ineffective [104]. This especially applies to those circuits that are simply not seen in the synchronous setup, mostly controllers. Two such circuits are *Muller Pipeline* and *Arbiter*, which play a vital role in the ANoC architectures, and their protection against SETs has become prudent. Unfortunately, neither their protection is sufficiently addressed, nor is their behavior in presence of the faults (to the best of our knowledge) properly understood. This is what convinced us to explore these circuits, and their response to SETs in more detail, in order to formulate the possible fault cases that a designer must consider while designing a fault-tolerant version of the same circuits.

1.2 Contribution and Significance of the Work

In this study we not just propose a working solution, but also emphasize on defining a certain set of protocols for building efficient and reliable ANoC architectures. We specifically focus on two critical concepts: flow control, and resource sharing, the correct operation of which is extremely critical for the performance of the ANoCs.

As far as the first concept, i.e., the flow control is concerned, our proposed framework elaborates the minimum requirements for building a safe, nonblocking, communication infrastructure between switches, which is supposedly the most fundamental property of the NoCs. We used

our framework to identify a weakness in one of the existing works in literature. Although the weakness could have been solved by other means as well, nevertheless, our systematic approach did point out its implication if not tackled properly. It also allowed us to come up with a novel mechanism, that not just follows all the protocols, but also saves dynamic power on the communication channel.

Considering the importance of resource sharing in an ANoC, beside building a high speed solution, we propose to systematically harden it against SETs and upsets. More importantly, we present an indepth analysis of the sensitivity of a *tree arbiter cell* (TAC) against such faults; our study shall benefit the designers who might overlook a few of those faulty cases. Other than this, we also propose to harden a few other critical components forming an ANoC router, such as input buffer controllers, and the crossbar.

Pointing out deficiencies in existing works, supporting our claims with simulations, and (partially) formal verification, describing a certain set of requirements that must be fulfilled by the components bulding the ANoC, and finally proposing a working solution that meets all the criterion defined in the framework, are a few credits to our study that make it useful to the community, and a reasonable contribution. In short, the objective of this work is to elaborate a systematic design of FT ANoCs.

1.3 Organization

Chapter 2 is divided into two parts. The first one presents the background on Async Circuits. We briefly overview the elementary primitives that form the foundation of asynchronous design methodology, the communication protocols between asynchronous components, the hardware supporting the protocols, and finally the tools used to model, simulate, and synthesize the asynchronous systems. In the second half, the emphasis is on discussing the fundamentals of the NoC architectures. Starting from the minimum requirements to build an NoC, we go to discuss the widely adopted network topologies, switching techniques, routing algorithms, virtual channels and flow control. The chapter is concluded by highlighting the main topics of the thesis, i.e., flow control and reliability concerns with the ANoCs.

Chapter 3 presents the design of a simple ANoC prototype with virtual channel support. We also present our framework that eases the design of an ideal (nonblocking) flow control mechanism for the ANoC. The chapter is concluded with a comparison of our proposed flow control scheme with the state-of-the-art mechanisms in terms of bandwidth requirement and dynamic power consumption on the communication channels.

Chapter 4 presents the design of a high speed arbiter, which is the basic building block of a router used in the NoCs designs. We propose to internally pipeline the existing TAC, and compare the throughput of the resulting design with a few others available in literature.

Chapter 5 discusses the fault sensitivity of the controller for the input buffers used in routers, called dataless *Muller Pipeline*. We propose a systematic hardening of the controller for a single stage, and subsequently address the possible interconnection of multiple stages. Our verification methodology is based on model checking, for which we define a set of properties that the proposed fault-tolerant solution must satisfy. The chapter is concluded with a comparison with two other possible hardening solutions in terms of area and performance.

Chapter 6 follows the same structure as chapter 5, except for that here we analyze the fault sensitivity of a 4-way TAC. We once again define a set of properties that an arbiter is supposed to guarantee, and verify the operation of the arbiter by injecting transient faults of limited length, at arbitrary points in time, using model checking. Then we propose a systematic hardening of the arbiter, verify its operation, and compare the solution with other possible solutions.
Chapter 7 proposes a mechanism for the protection of the data path (payload). In contrast to conventional error detection with retransmission schemes, we propose a method to resample data from the communication channel, rather than requesting for a retransmission. The latter is only needed in case of a soft fault, that does not fadeoff with time. A comparison is made with conventionally used mechanisms in terms of latency and area utlization.
Chapter 8 puts together the pieces to form a complete fault-tolerant switch by means of a few other components; this mainly concerns the simple logic circuits, such as multiplexers, demultiplexers, and latches. The primary method used here to harden those components is the hardware replication. We ensure that all the previously hardened circuits could easily be integrated with the rest.
Chapter 9 sumarrizes the findings of our work, and discusses a few prospective areas of further research.

CHAPTER 2

Background

This chapter is divided into two parts: In the first half, the principles of asynchronous designs and their communication styles are discussed, followed by an overview of fundamentals of the NoC architectures in the second half. At the end of the chapter we highlight some fault-tolerance aspects of each of them. Note that there is much more to this class of digital circuits than what has been described here; we only emphasize on their basics, and on what is sufficient to build the rest of the thesis on. To this end, we describe an asynchronous circuit as a class of digital design in which sequential components are driven by handshakes rather than a global clock signal.

2.1 Principles of Asynchronous Design

2.1.1 Data-path and Control

The part of the design that is responsible to perform operations on data, such as addition, multiplication, encoding, decoding, and storing is termed as the data-path. On the other hand, the part of the circuit that controls the timing and sequence of operations that the data-path performs is called the control-path. The control-path instructs the data-path when is the right time to perform an operation, and the latter simply follows that instruction. Two asynchronous circuits are connected in such a way that their data-paths are connected to each other, and the control-paths are directly connected to each other over a pair of signals, *request* and *acknowledge*. The latter are responsible to communicate to one another the completion of operation by their corresponding data-paths. In simple cases, a control circuit assumes the completion of operation after a certain delay (requires making some timing assumptions) and signals this to the other control circuit, in others, however, a number of computation steps must be taken to guarantee the correctness before communicating completion to the other circuit. This choice classifies asynchronous circuits to be using either *bounded* or *unbounded* delay models. Usually these two paths are synthesized to gate level netlists separately since they require a different methodology and set of tools.

2.1.2 The Concept of Handshaking

Two asynchronous circuits are connected to each other on a *channel*. The channel comprises of at least three signals, one of which is the acknowledgement, as mentioned above, while the rest forms the data bus with the request signal. In some cases, request is added as an explicit signal, in others it is encoded into the data bus. Hence a channel forms a unidirectional, point-to-point, communication link between the two circuits, where the *sender* submits its data on the data bus, and the *receiver* accepts them and performs the desired operation (arithmetic, logical, storing, forwarding etc). Once the sender has delivered its data on to the bus, it is its responsibility to indicate the validity of the data to the receiver as well. This exchange of request and acknowledgment between the sender and receiver pair is termed as *handshake*. Usually there are two naming conventions used for a channel that determine *who initiates the communication*: 1) If the sender initiates the communication by applying its data on the bus and asserting the request signal, then the channel is named as a *push* channel where the receiver accepts the data and asserts the acknowledgment in response. 2) In case the receiver wishes to initiate, it asserts the request signal first (normally indicating to the receiver about its availability, or enough storage space to hold the next data item), following which the sender applies the data on the bus, and asserts the acknowledgment signal in response. This type of a channel is called a *pull* channel. The block diagram and waveform depicting the operation of each type of channel are presented in fig.2.1. Historically, *producer* and *initiator* terminologies have also been used alternatively for the sender, and similarly *consumer* and *target* have been used to refer to the receiver. However, in case of a pull channel, the receiver is always the initiator. In the rest of the work, the term channel will always refer to a push channel, unless mentioned otherwise, and *req* and *ack* will be used as short for request and acknowledgment signals respectively.

2.1.3 Classification of Asynchronous Circuits/ Delay Models

Asynchronous circuits are said to have three classes according to their robustness against delay variations.

Delay Insensitive Circuits

Delay Insensitive (DI) circuits are the most robust form of asynchronous circuits. They do not make any assumption about wire or gate delays [25, 132]. Every transition from the sender must be properly acknowledged by the receiver. This indirectly means that every transition at the input of a gate/circuit must be correctly seen at the output before the next transition could be allowed to happen. However, the number of useful circuits that may be built DI is very limited [78], and have certain restrictions on their states for the correct operation: e.g., an OR gate must never enter the state where both inputs are high since the entry into this state will not have any impact on the output, and similarly the departure from this state will not be seen on the output as well [18]. In their implementation, *req* is embedded within the data, so the transitions on the data bus themselves indicate their own validity.

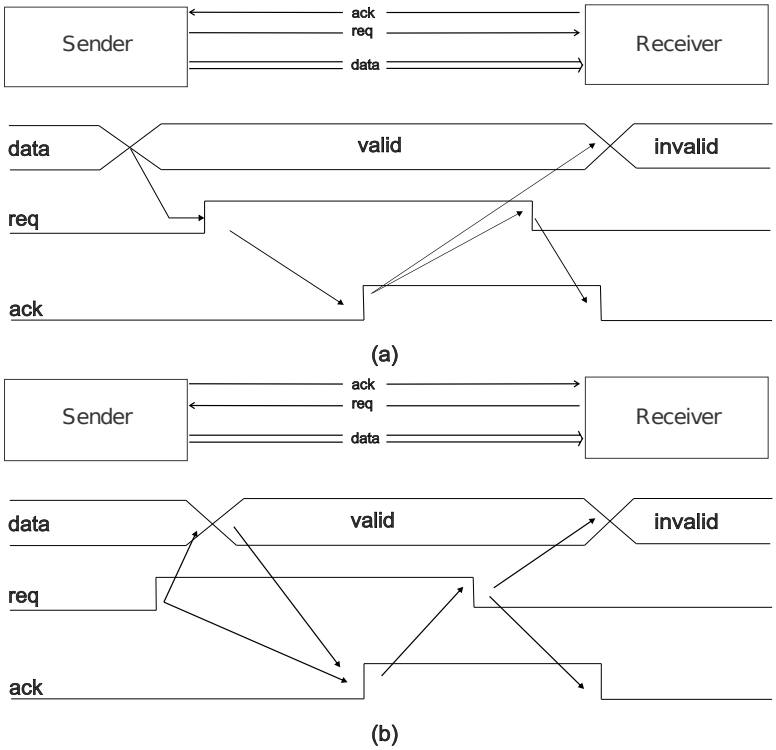


Figure 2.1: (a) push channel, (b) pull channel

Quasi Delay Insensitive Circuits

The Quasi Delay Insensitive (QDI) circuits assume all the forks to be *isochronic*, thereby compromising the delay insensitivity of the circuit. In isochronic forks, it is assumed that the receivers connected to the two prongs of the fork will, more or less, receive the data at the same time. In other words, the two prongs add identical delays. So, only one of the receivers needs to acknowledge the data, and it is assumed that the other has also received them correctly. Although very widely used, isochronic forks, if not implemented carefully, can lead to hazardous behavior in the circuits as demonstrated by Berkel [140].

A block diagram highlighting the difference between QDI and DI methods is presented in fig.2.2. The *black box* in (b) is a component that ensures the acknowledgment to the sender is forwarded only once both the receivers have correctly acknowledged the data. The detail on this black box shall be discussed later in the chapter.

Speed Independent Circuits

This class of asynchronous circuits most closely resembles the synchronous style: They assume that the data are stable at the receiver before the validity signal *req* is asserted, just as the clock edge must occur only once the data are valid and stable in the synchronous approach. In order to achieve that, appropriate delay elements must be added in the *req* path. In doing so, however,

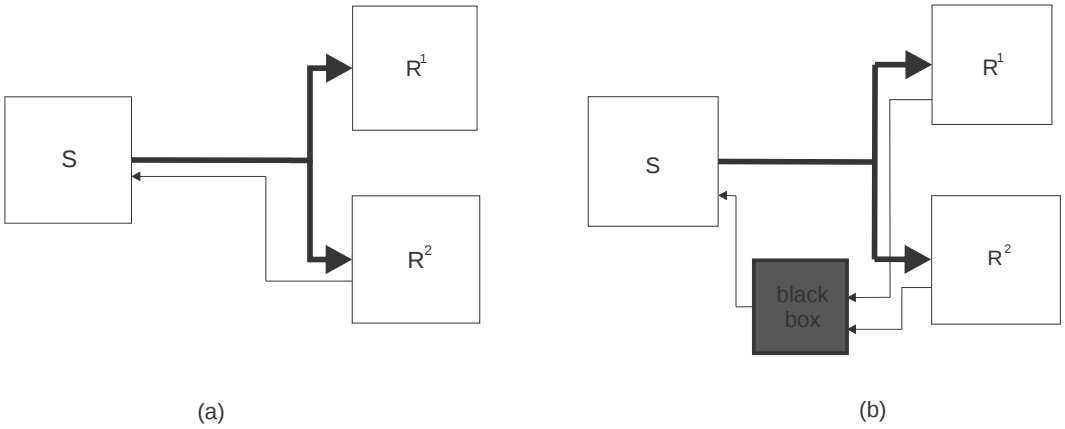


Figure 2.2: (a) QDI, (b) DI

these circuits lose their robustness against PVT variations. The only difference between these and synchronous designs, thus, is that the former are not worst case designs since they do not require global synchronization; as soon as one computation has finished, the next may start. Thus they exhibit average-case performance [117]. The push and pull channels shown in fig.2.1 are made as matched delay circuits without delay elements, which must be added on *req* and *ack* in (a) and (b) respectively.

2.1.4 Signaling Conventions and Data Representation

Since there is no global time involved in asynchronous designs, local controllers govern all transactions that happen on a channel [137]. No matter it is a push or a pull channel, the control signals, i.e., *req* and *ack* must follow an order of transitions. Normally there can be two types of signaling that would guarantee the validity of data on both ends, as discussed next.

4-phase Signaling

The 4-phase signaling protocol makes use of four transitions in total on the control path to complete one handshake cycle, or exchange one message in other words. Two transitions take place on *req* and *ack* signals each. An example of these protocols is already shown in fig.2.1. It may be noted in any part of the figure that validity and acknowledgment of data are indicated by level 1, whereas the down transitions on both signals simply make the phase, which is important because it resets the communicating entities to their default states, and thus the successive handshake cycle may happen. That is why this protocol is also known as return-to-zero (RTZ) protocol. It may also be possible to use the inverted logic levels, i.e., level high for *req* and *ack* may indicate the reset state. In that case, however, the name RTZ will be no more applicable.

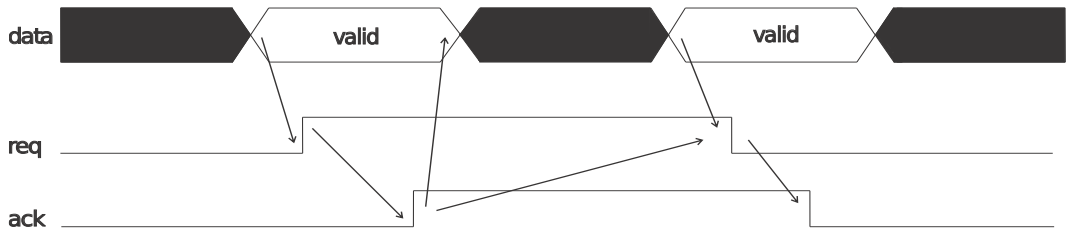


Figure 2.3: 2-phase Signaling

2-phase Signaling

In contrast to the 4-phase signaling protocol in which only one level indicates the validity, the 2-phase protocol makes use of both levels; hence each transition, i.e., *up-down* and *down-up*, becomes a signaling event. This protocol is also known as *Non-return-to-zero* (NRZ) protocol, fig.2.3.

Comparison between 4-phase and 2-phase signaling

Since the 2-phase protocol does not have to go through the RTZ phase, clearly it can attain twice as much data rate as that of the 4-phase protocol [133]. Furthermore, the less the number of transitions to forward a given number of messages (from the sender to the receiver), the more energy efficient the protocol will be.

On the other hand, the 4-phase protocol is somewhat more robust against PVT variations as compared to its counter part [106, 109, 111]. Another advantage that the 4-phase protocol provides is the several design possibilities according to the required functionality. For example, *late data validity* protocol may be employed when several senders like to communicate with a common receiver, and the latter is in charge of making a choice between the contenders. Hence, the first active (high e.g.) transition from the sender simply informs the receiver about its desire to use the channel, and only once acknowledged by the receiver, can the sender place the data on the bus, and their validity may be indicated by the opposite (falling e.g.) transition. In other situations where the sender is in control of the channel, the more traditional *early data validity* protocol may be employed in which the first transition is the active signal indicating the validity of data, while the following transitions simply form the RTZ phase. The remaining design possibility is called the *broad data validity* protocol, which requires data to be valid during the entire handshake protocol. Hence it may be used in place of either of the preceding options. In our work, Ch. 5 and Ch. 6, we will show that such protocols come in very handy in tackling SETs. The operation of these schemes is summarized in fig.2.4, and their detailed description may be found in [103].

Finally, the most important difference between the two protocols becomes visible when it comes to implementing the storage elements, latches. The 4-phase protocol can simply use the control signals to drive a level controlled latch (one level makes it transparent, and the other makes it opaque). In 2-phase protocol, however, a latch needs some additional logic, which must make its transitions twice as fast as the inputs, since every alternate input transition repeats

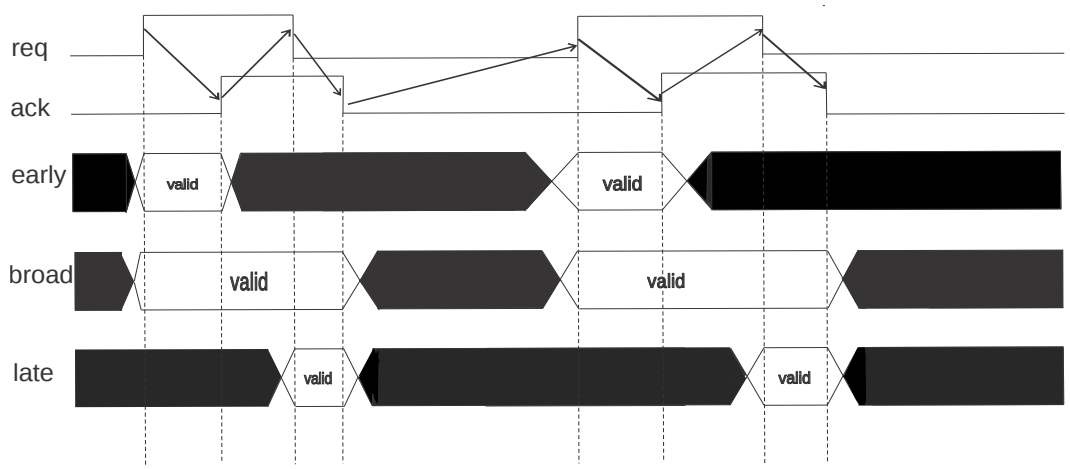


Figure 2.4: Various 4-phase Signaling Conventions

the same operation. Although complicated, the 4-phase protocol allows rather simple implementation of the control circuits as well as the latches [6]. Throughout in the remainder of this work, we make use of the 4-phase protocol, unless mentioned otherwise. Especially because our major concern is reliability of asynchronous circuits, we have experienced that the redundancy that these protocols have (RTZ phase), may play a vital role in building fault-tolerant circuits. In chapters to come, we will further elaborate on our findings in the same context.

So far we have discussed about the handshakes that happen on the control path. The data, which form the actual information to be communicated must find a suitable representation as well, *encoding*. Usually there are two ways of representing these messages: 1) each wire carries one bit of information, *single rail encoding*, 2) each bit of the message may be transported on multiple lines, *M-of-N encoding*. These encoding schemes are discussed next.

Single Rail Encoding

As mentioned above, in single rail encoding, each wire carries one bit of information [103]. There are explicit rails to carry the control signals. Since the control signals are somehow bundled with the data signals, this encoding scheme is also called as *bundled data encoding*. If the corresponding control signals follow 4-phase handshaking, then the protocol is termed as 4-phase bundled data protocol, otherwise 2-phase bundled data protocol. These schemes are most widely adopted due to their simplicity, and area cost which is roughly the same as that of the synchronous designs [6]. All the simulations shown so far used this type of data representation.

M-of-N Encoding

This type of encoding is used within the DI class of asynchronous circuits, where *N* wires carry $log_2(N)$ bits of information, and there is an explicit wire to carry the acknowledgment [137]. The *dual rail* encoding [144] is a special case of *1-of-N* encoding, with $N = 2$. Each bit is encoded using two rails, *true*, and *false*. Level 0 is represented by logic '1' on the *false* rail,

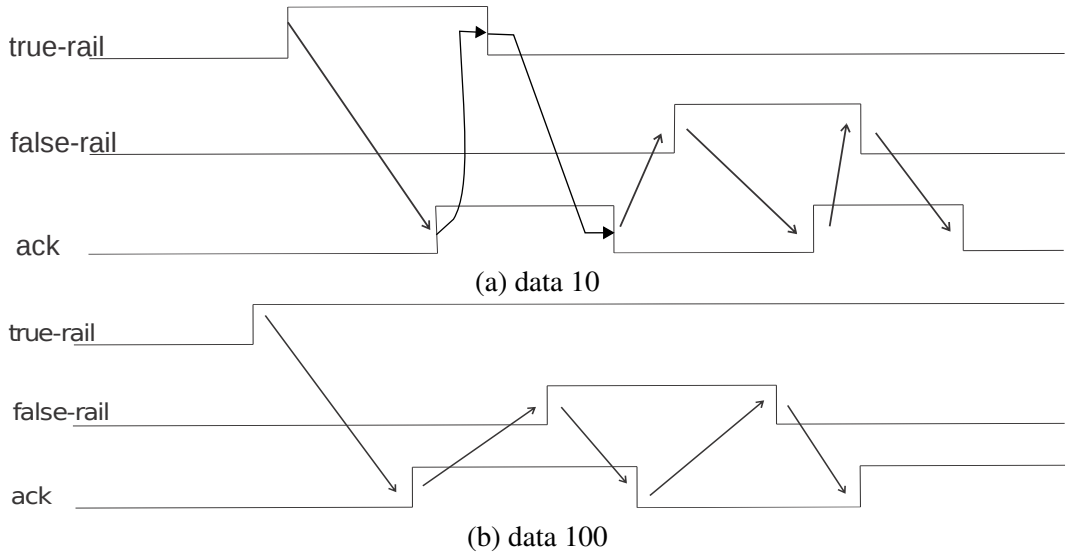


Figure 2.5: Dual rail signaling: (a) 4-phase, (b) 2-phase

while logic '1' on the *true* rail represents logic level 1. Both rails when at logic level 0 means no valid data are available. It might also be possible to reverse this representation, i.e., level 1 may indicate that no valid data are available. Recently in their work, Moreira et al. [86] showed that this representation, *Return-to-One* protocol, reduces static power dissipation in Muller C-elements (MC) of QDI circuits.

As may be understood conveniently, the two rails are mutually exclusive, i.e., only one of them is allowed to make a transition at a time. Since there is no explicit *req* available, the receiver is supposed to detect that transition to be sure about the validity of new data. The circuit responsible to perform this task is called *completion detection mechanism*. An example of completion detection and the corresponding dual rail encoding for each type of signaling is presented in fig. 2.5 and fig. 2.6.

Note that it is equally important to detect the RTZ phase on all lines within the completion detection mechanism. Since just one input being low on an AND-gate may drive it to logic level low, it is essential to replace it with something that must wait for both data lines to return to zero. An MC [88] replaces the AND-gates in the completion detection circuit. The MC is discussed in the next section. The 4-phase dual rail protocol is specifically used in QDI designs, however, its cost due to the duplicated wires, and the completion detection circuit is a major drawback.

One-hot encoding also belongs to the same class of 1-of-N encoding. They use 2^n lines to represent *n* bit data. For $n = 2$, the primary difference between a dual rail implementaion and that of one-hot encoding is that in the former each bit is encoded separately using two bits, whereas, in the latter the entire two bit message is encoded using a unique 4-bit code. Table 2.1 [121] demonstrates this difference for $n = 2$. While the overhead incurred by the two schemes is the same (for this specific case), the smaller number of transitions in the one-hot encoding method forms a more energy efficient mechanism. The 4-phase protocol in the 1-of-2 encoding is also called *Null Convention Logic* (NCL) [41], in which each pair of code words

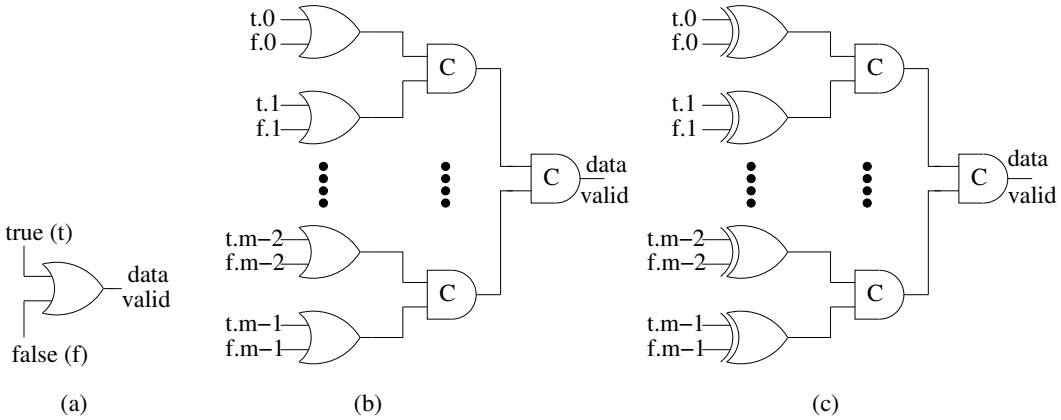


Figure 2.6: Completion Detection Mechanism: (a) 1-bit message and 4-phase signaling, (b) m-bit message and 4-phase signaling, (c) m-bit message and 2-phase signaling

Table 2.1: Representation of 2-bit message using dual rail, and 1-of-4 codes

Message	Dual Rail Code				1-of-4 Code
	true.1	false.1	true.0	false.0	
00	0	1	0	1	0001
01	0	1	1	0	0010
10	1	0	0	1	0100
11	1	0	1	0	1000

Table 2.2: LEDR Encoding

Parity/Data	0	1
Even	00	11
Odd	01	10

is separated by an *empty word* or a *spacer*. Instead of the MCs, as used in dual-rail codes in completion detection circuit, *majority* or *threshold* gates [125] are used in NCL. However, for $n = 2$, the completion detection circuits are identical to those of dual rail codes.

Another important dual rail encoding scheme is the *level encoded dual rail* (LEDR), proposed in [33]. In this scheme the first bit of the code represents the original data bit, and the second bit forms a phase, which keeps alternating between *odd* and *even* representation. Even if two consecutive words are identical, their phase difference will be sufficient to trigger the completion detection circuit at the receiver. This way, the requirement to having a spacer or

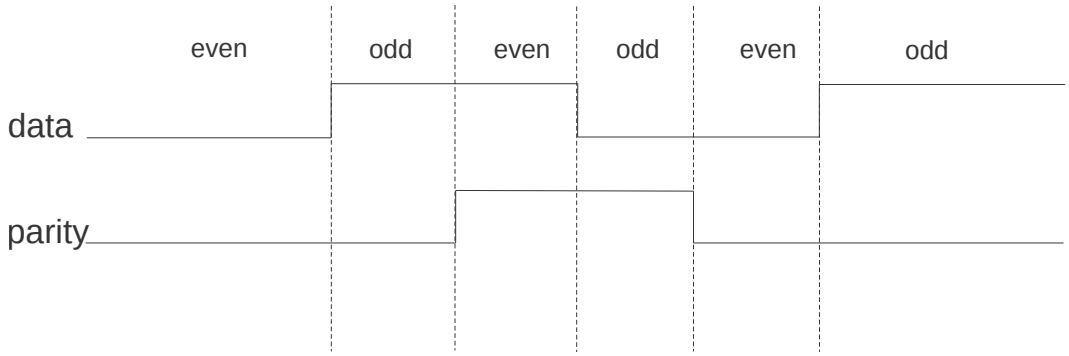


Figure 2.7: An example of LEDR encoding

the empty word separating the codes (as in case of NCL) is no more needed, resulting in more power and performance efficient codes. Table 2.2 and fig. 2.7 summarize the concept of LEDR encoding. Communication dominant systems [81] have been designed using LEDR encoding.

In our work we have emphasized solely on the single rail, 4-phase handshake protocol. Therefore, we have deliberately ommitted details on the rest of the schemes that we have mentioned here. Further literature may be found in [127].

2.1.5 Asynchronous Circuits and Pipeline Implementations

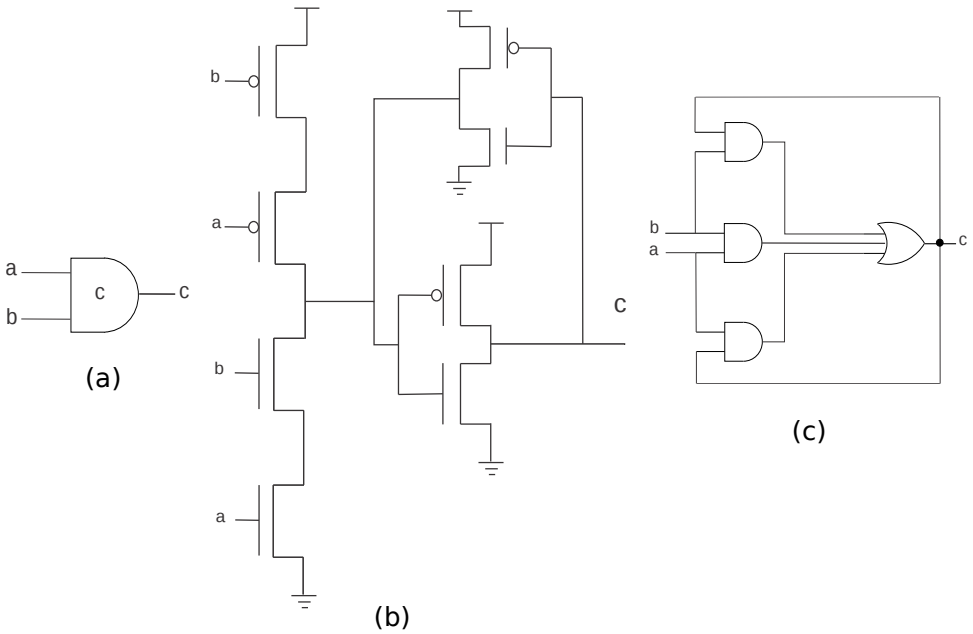
There are some fundamental components that lay the foundation of most of the asynchronous circuits and systems. Since in our chapters to come all those components will be repeatedly used, it is, therefore, essential to briefly review them in turn. Based on these components, and the protocols that we have revisited in the previous sections, we will discuss the design and implementation of various asynchronous pipeline styles.

The Concept of Valid Tokens, Empty Tokens, and Bubbles

Data transfer between communicating circuits (including data- and control- paths) may be thought of as *tokens* which travel from the sender to the receiver [127]. The *req* becoming high is referred to as a valid token, and the RTZ phase (in 4-phase protocol) is termed as an empty token. Hence every pair of valid tokens is separated by an empty token. Once the receiver has asserted *ack*, a *bubble* is said to have been sent to the sender, only after which the sender can place the empty token. So, every pair of valid and empty token is separated by a bubble. The tokens are capable of diverging-to, and converging-from, thus allowing one-to-many and many-to-one connectivity. There are dedicated circuits that perform these mappings, also discussed in the following elementary primitives.

Elementary Primitives

The MC is the basic building block of asynchronous circuits. Fig. 2.8 presents symbol, transistor- and gate- level schematics of a 2-input MC. For identical inputs, an MC behaves like an AND-



a	b	c_next
0	0	0
0	1	c
1	0	c
1	1	1

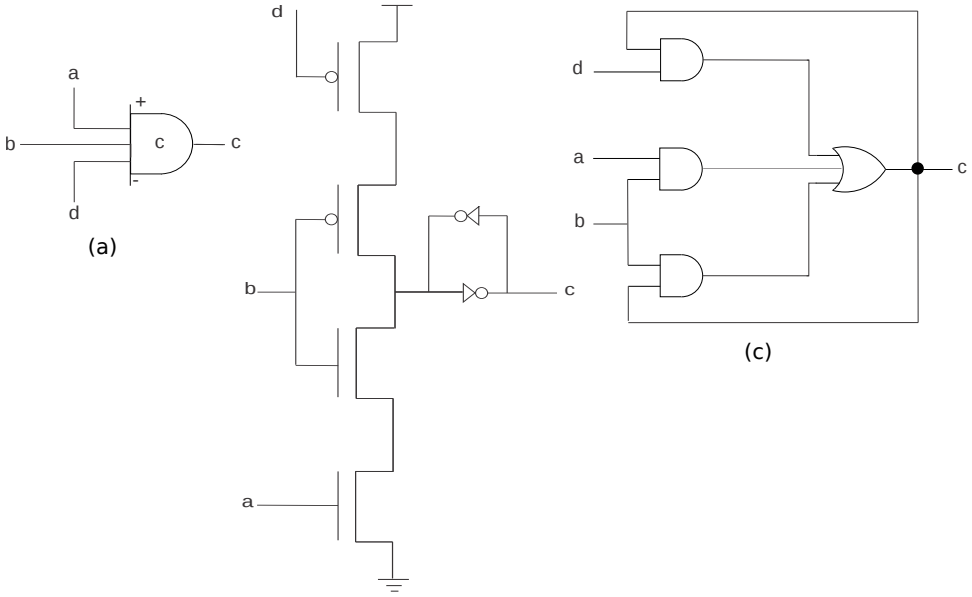
(d)

Figure 2.8: MC: (a) symbol, (b) transistor level, (c) gate level, (d) truth table

gate. On the other hand, it applies a hysteresis when its inputs are at different logic levels, i.e., it behaves like a conventional latch.

There is a variant of an MC called an *Asymmetric MC*, in which only some of the inputs affect the operation of the gate either in rising or in falling transitions. In other words, some inputs become irrelevant for the operation. Naturally, in such MCs *n*- and *p*- stacks have different structures, unlike the symmetric ones in which n- and p- stacks are images of each other [6]. Fig. 2.9 presents the corresponding symbols and truth tables of asymmetric MCs. Note that in the example shown, input *b* is common, and hence required in both rising and falling transitions. On the other hand, *a* is just needed in rising, and *d* is needed in falling transitions.

On various occassions it is required that all the clients be active before they can access the common resource (receiver). In other words, it must never happen that only some inputs may access the resource while others are still inactive. What is needed here is called *synchronization* of all inputs. A component that synchronizes (waits for) all the inputs, and then initiates the handshake is called *Join*. Subsequently, *ack* from the receiver must be forwarded to all the clients. Fig. 2.10 presents the operation of this module: in (a) one input is still missing while the other already has valid data available, token in other words. In (b) the missing input has also



a	b	d	c_next
-	0	0	0
0	1	-	c
-	0	1	c
1	1	-	1

(d)

Figure 2.9: Asymmetric MC: (a) symbol, (b) transistor level, (c) gate level, (d) truth table

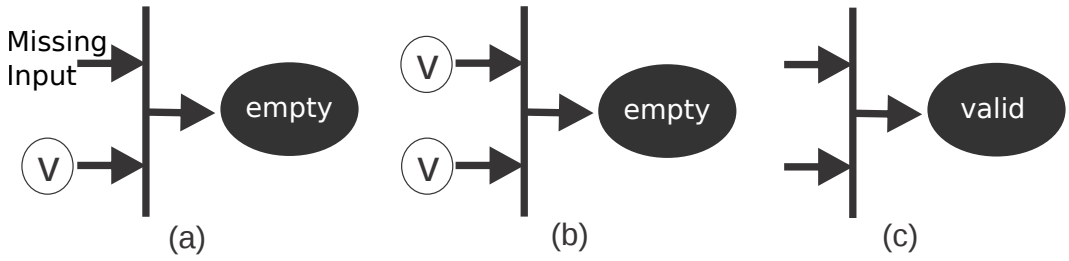


Figure 2.10: Operation of a Join module

arrived, followed by the transfer of token to the receiver, as depicted in (c).

In contrast to join, there is a component called *fork*. It simply forwards the data available at the input to two (or multiple) outputs, called prongs of the fork. This process is also called *multicasting*. Interestingly, the join and fork modules always work together in a DI environment. For example, *ack* in case of join must be forked to the clients. Similarly, *ack* from both the receivers in fork, must be joined using a symmetric MC. This organization is depicted in fig.

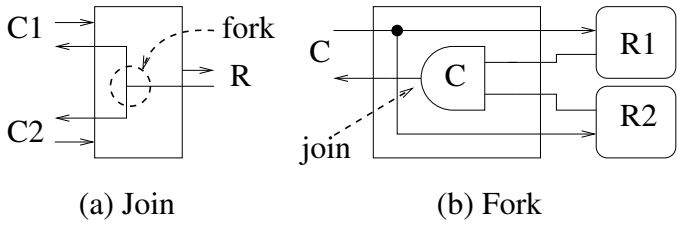


Figure 2.11: (a) Fork in a Join, (b) Join in a Fork

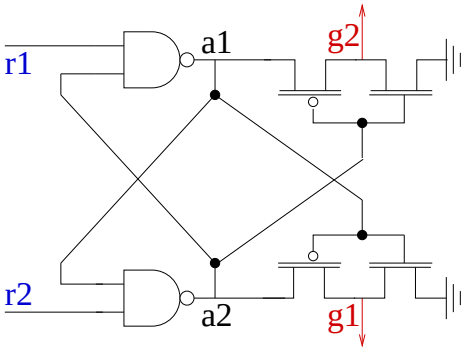


Figure 2.12: Gate level schematic of MUTEX

2.11.

Closely related to the concepts of join and fork are *merge* and *split* respectively. However, the merge module simply forwards the token to the receiver as soon as one of the inputs is available, forcing the other input to wait for the current handshake to complete. Similarly, the split module only forwards the token to the active client: In case *ack* from the receiver is being split, then the circuit must remember the client that recently accessed the receiver. In case *req* from the sender needs to be split, then there must be some additional information to choose the winner, just as *multiplexers* (mux) have explicit selection lines.

The merge module, operating in an environment where the inputs are mutually independent, and may occur simultaneously, can lead to hazardous results: the acknowledgment may be forwarded to all the inputs for instance. In such environments the merge module is normally replaced with a circuit called *MUTEX* [79] short for mutual exclusion element, which guarantees *just one winner*. It is constructed using two cross-coupled NAND-gates, which tend to block one another from making a transition. Simultaneous, or very close input transitions may result in metastable (lingering between levels high and low for some time) outputs of these gates, which are then filtered using a pair of inverters, as shown in fig. 2.12. The arrangement of the inverters, i.e., Vdd of one is the input to the other, ensures that an inverter will only switch to high if the input to the other one is sufficiently high as well, thereby waiting for the metastability to completely vanish. Usually instead of *ack*, the term that is used to denote the output of the mutex is *grant* (*g* in the figure). Details on mutex will follow in the chapters to come.

Another interesting component is a variant of a merge module called *toggle*, which possess

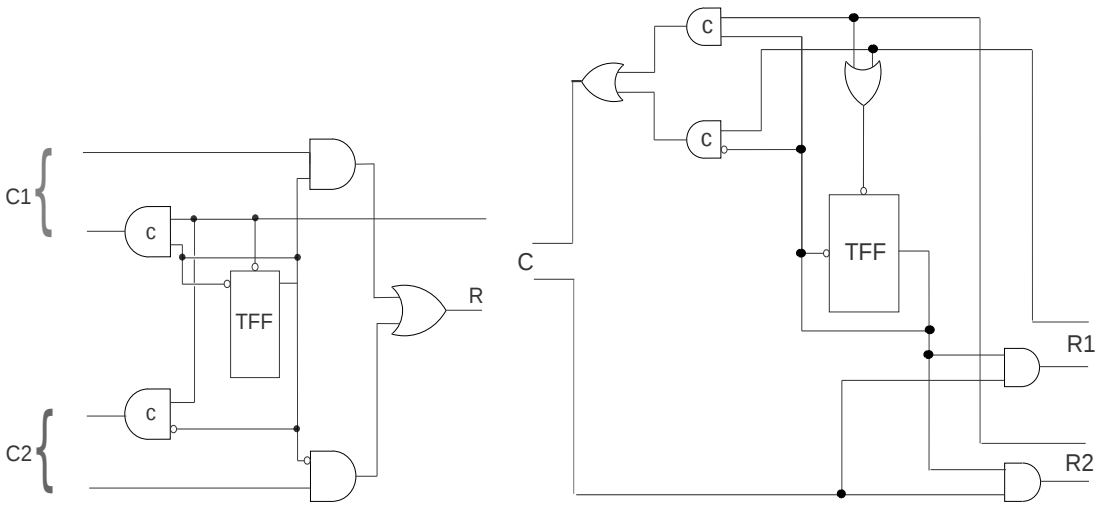


Figure 2.13: Toggle Circuits: (a) Merge, (b) Split

a token that freely circulates between the clients. The client that currently holds the token is only allowed to access the resource, and the other one is simply put to wait until the bearer of the token has finished its handshake. Once the current handshake is complete, the token flies to the other client, which can then proceed with its transfer. This module is made up of a *toggle flip-flop* (TFF), which after completion of every handshake flips its state, thereby allowing the other client access to the output port. On reset, the first winner depends upon the initial value of TFF. Likewise, toggling may be done in a split module, where the input is alternately switched to the two output ports. Toggle merge and split modules are presented in fig. 2.13. Note that the TFF is activated with the falling transition of output *ack*, i.e., with the completion of the handshake. The MC at each of the input *ack* makes sure that the TFF has already flipped and settled to a valid state before the handshake with the left environment may be completed. This guarantees correct behavior of the circuit.

Efficient asynchronous circuits are usually built as pipelines, which increase the overall throughput by distributing the task among several function units operating in parallel on different data values. There are several types of asynchronous pipelines, micropipelines [133], mousetrap [124], GasP [134], QDI [76, 99], asP* [83], wave [19, 53], and surfing [146]; all of them have a common Muller pipeline as their backbone though. The Muller pipeline is a simple arrangement of MCs, such that each of them forms a single stage. The output of each stage (MC) serves two purposes: 1) it becomes the input *req* to the successor stage, 2) it is sent as the inverted *ack* to the predecessor stage. The first stage receives its input *req* from the sender, and generates *ack* in return. Similarly the last stage generates the output *req* to the receiver, and receives the *ack* in return. The Muller pipeline, shown in fig. 2.14 is a mechanism that relays handshakes [127]. The pipeline is said to be empty when all the MCs are initialized to zero. At this point in time, the left-environment (also called sender or producer) can initiate the handshake by asserting *req*. While this transition ripples through the pipeline to the right-

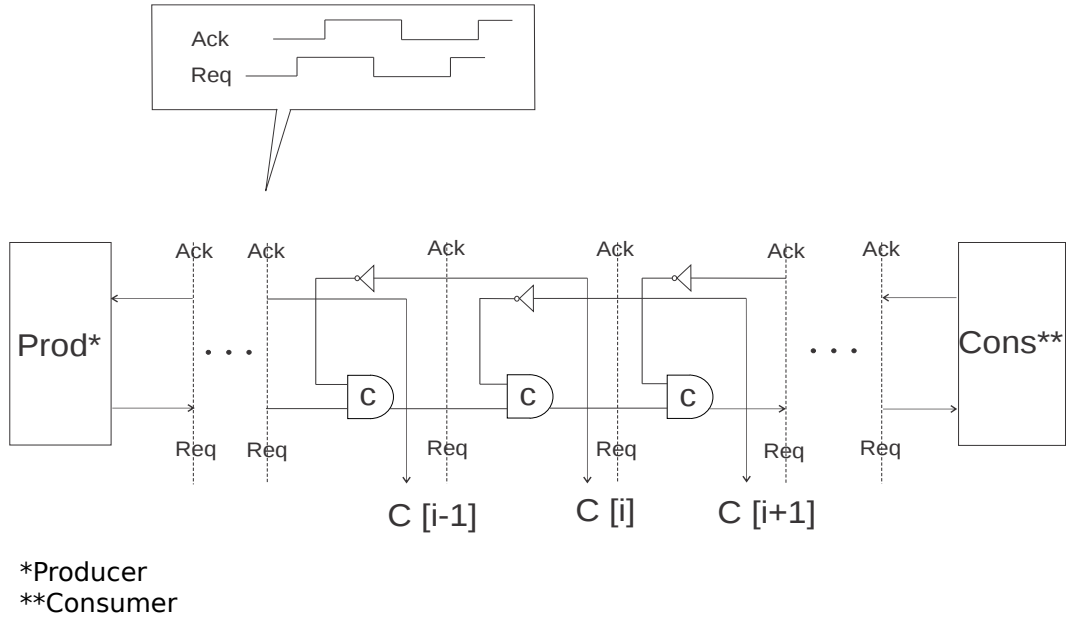


Figure 2.14: Muller Pipeline [127]

environment (also called receiver or consumer), due to the symmetry, each stage sends the *ack* to the previous stage. Now in case the producer is faster than the consumer, it may deassert its *req* which should traverse the entire pipeline up to the last stage and get blocked, waiting for the receiver to consume the token by asserting the *ack*. Sooner or later, a time may come when all the stages get blocked because of the slow nature of the receiver. A fully filled pipeline has an interesting characteristic, i.e., alternating stages will always store opposite values. Singh et al. [124] made use of this feature to build the mousetrap pipeline, and the same feature will be repeatedtly discussed in the coming chapters. Here we discuss only basic pipeline styles that are relevant to our work.

4-phase Bundled Data Pipeline

The Muller pipeline once equipped with the datapath becomes a 4-phase bundled pipeline, fig. 2.15. Each box with an MC and a storage element forms a pipeline stage, where each output of an MC makes the associated latch transparent or opaque. The remaining boxes comprise the actual processing units with delay elements on top to ensure that the control signals reach the successor stage only once the function unit has completed its processing. An interesting observation is that in a completely filled pipeline (i.e., successive MCs hold alternating logic levels), only half of the stages store some data, every other stage is transparent. This is just like a Master-Slave setup in synchronous environment, and requires $2n$ storage elements to store n values. This is one drawback of such pipelines. The other drawback is the slow processing speed, since each stage needs to perform handshake with both neighbors, only after which a data item may proceed to the next stage [127].

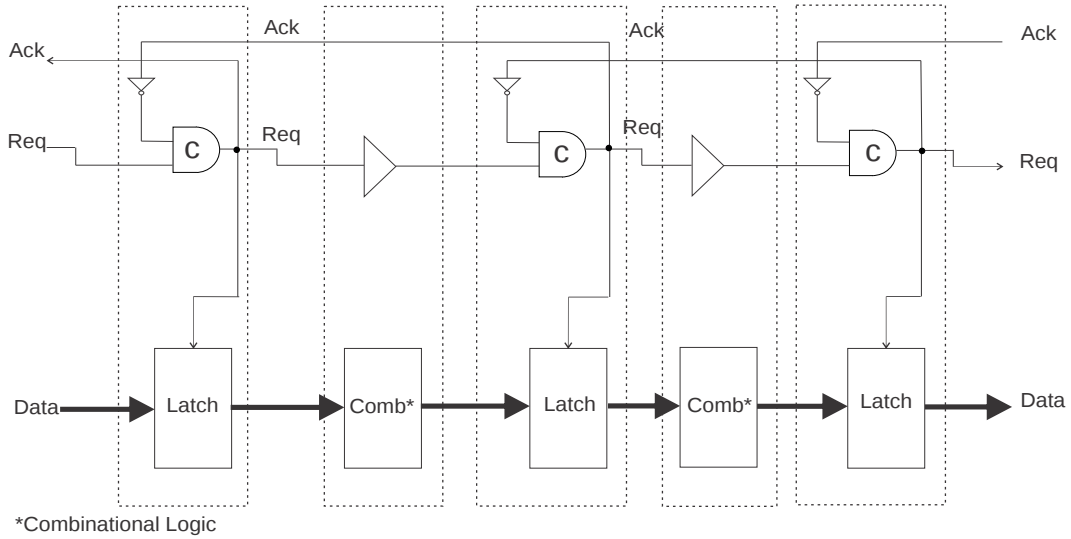


Figure 2.15: 4-phase bundled data pipeline [127]

Micropipelines

Micropipelines or 2-phase bundled data pipelines were proposed by Sutherland [133]. The control path is identical to the Muller pipeline once again, fig. 2.16, only the interpretation of the signals is different, which makes it follow the 2-phase handshaking. Sutherland proposed a set of dedicated components, one of which was the design of an event controlled, the so called *Capture-Pass* latch. Each event controlled latch comprises two side by side latches that are activated alternatively to generate similar response on rising and falling events [137]. This makes the design of the latch more complex and the circuit slow. Each MC is responsible to determine if the current and the successor stage are at the same or different levels. In the latter case, this indicates that one of the latches is full, and the other empty, which allows the current stage to forward new data to the successor.

2.1.6 Modeling and Synthesis of Asynchronous Circuits

The designers of asynchronous circuits and systems do not have a large number of modeling and synthesis tools that match their specifications. Usually for smaller designs event- or state machine- based specification is employed. The former requires the designer to describe the behavior of the entity (that he wishes to model) in a *Petrinet* [105] or a *state transition graph* (STG)[1], modeled using a tool, such as *Workcraft* [107], and sythesized using *Petrify* [28]. The state machine based specification, on the other hand, is based on Huffman's asynchronous *finite state machine* (FSM) [58], which was later modified and formalized to operate in burst mode on several occassions [95, 129], and supported by the synthesis tools [47, 128] respectively. Unfor-

[1] An STG is an interpreted petrinet in which each transition is either marked with $+/-$ indicating a rising/falling transition, or $\sim$ representing a level change.

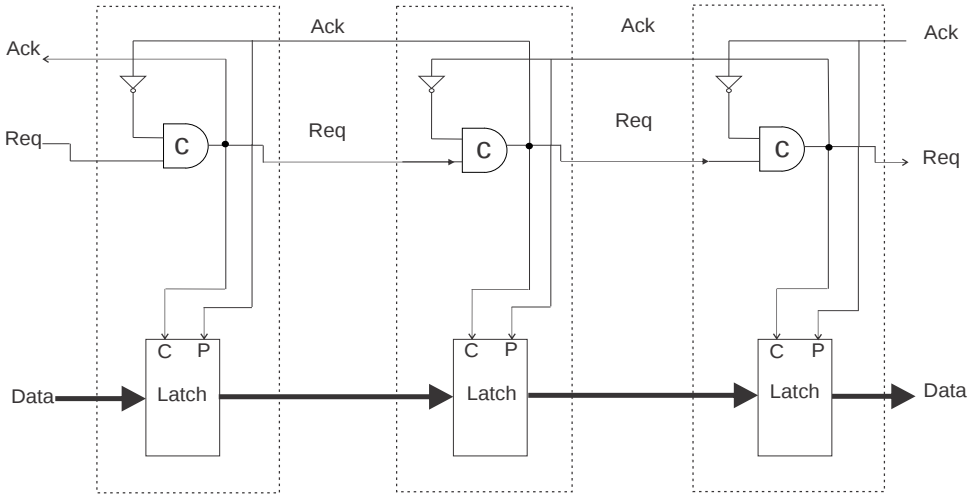


Figure 2.16: 2-phase bundled data pipeline [127]

tunately this methodology does not prove too feasible when it comes to describing large systems. In such situations either small circuits synthesized individually are integrated using some *hardware description language* (HDL), or specialized high level description and synthesis platforms for asynchronous circuits, such as Balsa [7], are used. Although Balsa makes the description of some complicated components extremely easy (adds the control signals automatically, the designer simply needs to describe the functionality), and the synthesis process entirely invisible to the designer, it has some disadvantages: The gate-level netlist that it generates is fairly complicated to understand and modify as may be needed occassionally. It is already equipped with certain predefined (frequently used) components and does not provide flexibility to the designer to mold the description according to his needs, e.g., $select()$, and $arbitrate()$ are two predefined functions implementing merge and MUTEX operations. In case the resulting netlists do not meet the designer's requirements, he will eventually have to design his circuit using the former approach based on STGs. Subsequently, he needs to spend a lot of effort and time to locate the piece of code in Balsa's netlist that described those components, and replace them with Petrify's netlist, which becomes a tedious task to accomplish. Based on these observations, and the nature of our objectives, in our work we always used the former approach; and in the rest of this work whenever we mention of synthesizing an asynchronous circuit, it would always refer to the same, i.e., describing the logic as STG, and verifying using *Workcraft*, synthesizing using *Petrify*, integrating small circuits using *Verilog* HDL.

An STG comprises of transitions, places and arcs. The arcs join a transition with another transition, a transition with a place, and a place with a transition, and desribe the sequence of operations. Transitions may be rising or falling, represented with $+$ or $-$ respectively. Arcs coming in, and going out of a place can deposit and withdraw a *token* into/from the place respectively, upon transitions connected on the other sides of the arcs. A transition can only happen if each arc coming into it either possesses a token, or the place before the arc possesses one. A *safe* STG guarantees that each place and arc may only have one token at a time. In some cases, an STG

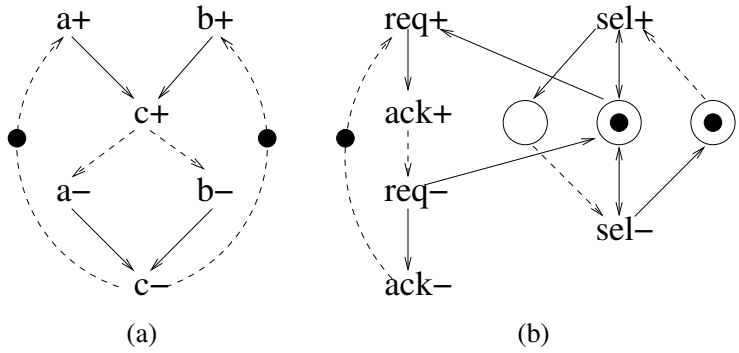


Figure 2.17: Simple STGs, (a) MC, (b) Data and Control Mixed

may not have a place; one such example is an MC shown in fig. 2.17(a). The black marks on the arcs between $c-$ and $a+$, and $c-$ and $b+$ indicate the availability of tokens. It means upon reset, the first transition that may happen is either $a+$ or $b+$. Recall the operation of an MC: Both rising and falling transitions of the output *require* all the inputs to make their corresponding transitions, i.e., in this case, $c+$ will wait for both $a+$ and $b+$; likewise for $c-$. This behavior is nicely modeled in the STG since c always has two arcs coming into it, and hence requires each arc to possess a token before it can make a transition.

In some cases, control signals must be used with the data signals, although in most cases this is undesirable. Mux and demultiplexer (demux) are two such circuits in which the selection line must be taken into account other than the validity of data. A very simple, dummy, STG is shown in fig. 2.17(b) where sel is an example of such selection line, and *req* and *ack* are the associated control signals. Note the two bi-directional arcs, called *read arcs* between a place and the two transitions of sel. They do not steal the token from the place, but the place must have a token inside for sel to make its transitions. As may be conveniently understood, sel can keep making its alternating transitions until stopped by $req+$, which steals the token from the middle place. The token is returned with $req-$ allowing the sel to change as per its wish. The $ack+$ in between guarantees that the receiver has already stored or used the previous data.

A very usual problem with a carelessly described STG is a *deadlock*. Deadlock refers to situations where none of the transitions in the STG can happen due to poor initial placement of the tokens. For example, if in fig. 2.17(a), the arc between $c-$ and $b+$ was not initialized with a token, then only $a+$ could happen, after which the circuit would be in the deadlock state, not allowing $c+$ to happen. *Workcraft* provides this feature to check the STG if it contains any traces that may lead to deadlocks. However, it is completely a manual effort to check and subsequently correct the STG.

2.2 Fundamentals of Networks-on-Chip

An NoC is a programmable grid-like arrangement (network) of routers, allowing interconnection of several subsystems, called *processing elements* (PE) or *computing resources*. Unlike the shared bus architecture, an NoC allows several PEs (not all) to communicate with each other

simultaneously, therefore scales well with the increasing number of PEs. An NoC is termed as *homogenous* if all the PEs in the network are identical processors leading to a symmetric *chip multiprocessor* (CMP) architecture [141], and *hetrogenous* if the CRs are various *intellectual property* (IP) modules with largely varying parameters and characteristics, leading to an asymmetric *multiprocessor systems-on-chip* (MPSoc) design [26]. Because of varying characteristics of the PEs (such as clock speed, width of the data bus, etc), a component, called *network interface* (NI), is needed between each PE and the network of routers, which would allow each of them to access one another by encoding their messages into a suitable (compatible) format. This leaves the router, and the entire network for that matter to be made generic. Since messages from one PE may be directed to any other PE within the network, it is another responsibility of the NI to append the destination information (address of the target node) with each message, and to request the routers to route those messages towards their appropriate destinations. While an NI is specifically designed to operate between the network and the subsystem it is connected to, in this work we limit ourselves to more generic issues related to the network of routers, which makes our study applicable to both types of the multiprocessor platforms. In what follows, we briefly describe the components, protocols, and the requirements of building such a network of routers.

2.2.1 The Basic Architecture

A *switch*[2] allows several devices to be interconnected efficiently. Since the shared bus architecture does not fulfil the bandwidth requirements, the switch has become the core component of the NoC designs. The primary objective of a switch is to guide the incoming traffic from a computing resource to its desired output port of the router connecting to another computing resource. This job becomes complicated when there are multiple resources requesting access to the same output port. As a result, the switch must choose the winner amongst them, and store the other traffic (that has been blocked) to avoid congestion on the input port. This description suggests the minimum requirements to designing a router: 1) Input buffers at each input port to temporarily store the incoming traffic, 2) An internal switch, and 3) An *arbiter*[3], which is a circuit responsible to make a fair choice among the contending data messages. Since a switch must know which inputs are to be connected to which outputs, a router is thus equipped with an additional controller that must provide this information to the switch. Sometimes this controller is referred to as *routing and arbitration unit*, and sometimes as *input controllers*. Fig. 2.18 presents a simple design of a router.

The data packages, from the source PE while traversing their route (towards their target node), are divided into packets, which are further divided into fixed sized *flow control units* (flits). It is one of the responsibilities of the NI to perform this division. Normally, each package has two or more flits, where the first one, called the *header*, carries the routing information, i.e., the information about the desired computing resource. This information is supplied to the routing logic, which then programs the switch to form the required connection between

[2]Sometimes the terms switch and router are used alternatively. In our work we assume a switch to be a component inside a router.

[3]Details on arbiters and arbitration are given in the coming chapters.

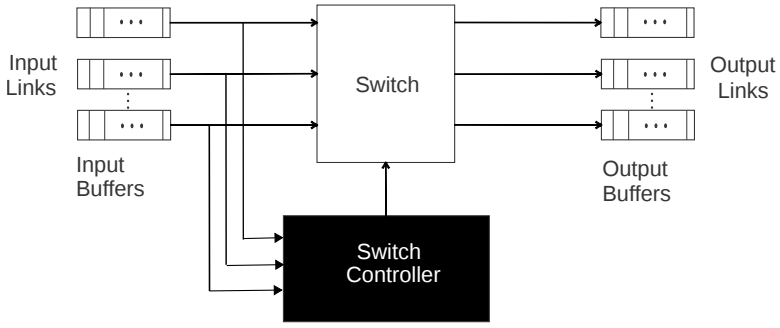


Figure 2.18: A simple router

the requesting computing resource and the one it needs to access. Since size of each packet may be different from the others, dedicating a large storage space on each router may cause an unnecessary overhead. Dividing those variable sized packets into fixed sized flits, thus, proves to be a more elegant solution.

2.2.2 Networks with Multiple Routers

With the increasing number of computing resources, the complexity of the switch grows badly, thereby making it infeasible to use a single router. It is, therefore, required to build networks comprising multiple routers, and connect them with each other using some pattern, and specify how a path may be established between the requesting computing resource and its desired destination. This pattern is generally termed as the *network topology*. Fig. 2.19 presents a simple schematic (showing the minimum requirements) of an NoC. Each router in a network may connect to none, one, or multiple PEs depending upon the topology. For example, in a *mesh* of routers, each router is normally connected to one PE (such type of a network is termed as a *direct* network [126]), whereas, in case of a *butterfly* network, the corner routers are connected to two, and the intermediate ones are not connected to any PE. The type of a network in which there are some routers not connected to any processing nodes is called an *indirect* network [126]. Some most widely adopted topologies are depicted in fig. 2.20. The 2-D mesh of routers, shown in fig. 2.19 is the most commonly used topology, the reason being its easy scalability.

2.2.3 Function Layers

There is no concensus about the function layers of an NoC [126]; while a few authors have argued that it may be divided into application, transport, network, data-link, and physical layers [138], others have described physical, switching, and routing layers as the only function layers of NoCs [39]. The description of the layers provided in the latter is simply a subset of that described in the former; here we would summarize what was discussed in detail in [138], emphasizing on those aspects that later will constitute the major portion of this thesis.

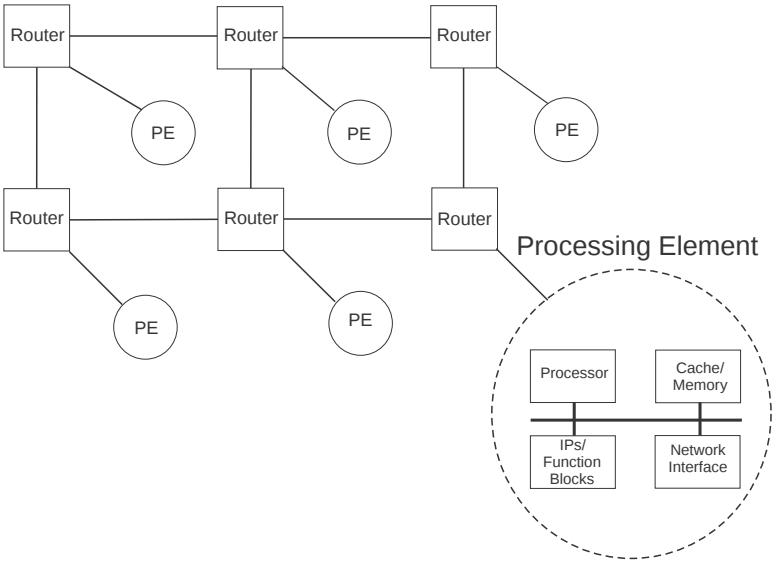


Figure 2.19: A typical NoC with minimum requirements [126]

Application Layer

At the *application layer* the target application is broken into smaller computation tasks to achieve optimum performance and energy parameters.

Transport Layer

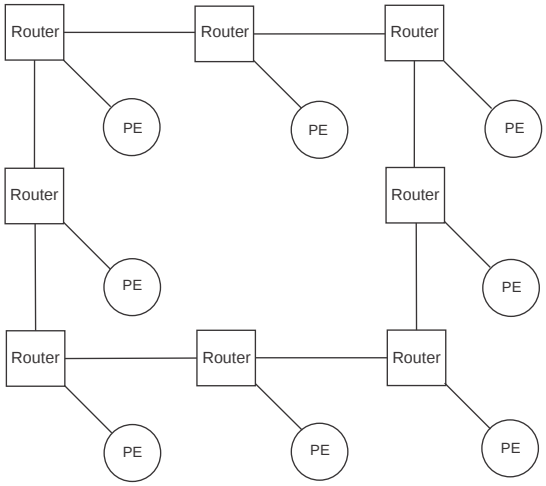
The primary task of the *transport layer* is to make sure that overflow in the input buffers at any node never occurs, otherwise, the channel between the communicating routers (also called a global link, or a shared physical link-SPL) may be indefinitely blocked, thereby, preventing other trafic to use the SPL as well. This functionality is termed as *flow control*.

Network Layer

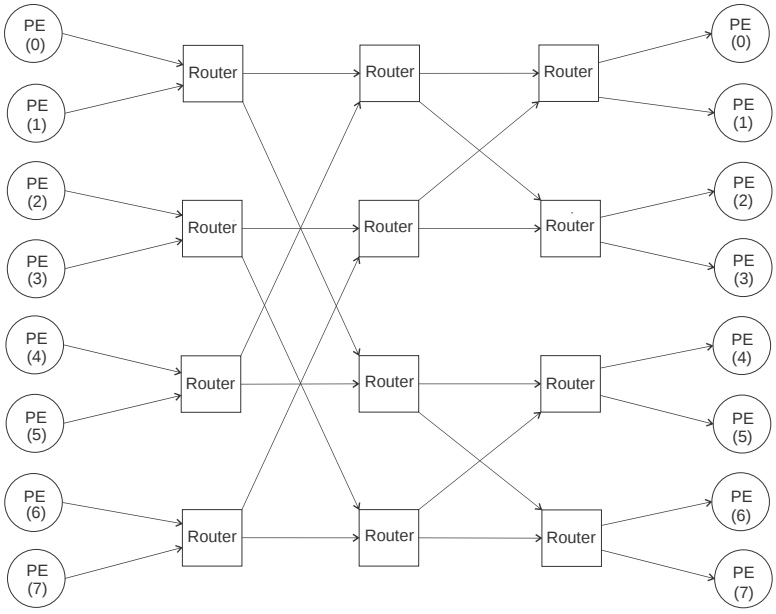
Issues related to the network topology, and the interconnection architectures, (such as how the resources are connected etc.) are handled at the *network layer*. Transfer of packets between nodes, i.e., routing, is another task performed at this layer.

Physical Layer

The focus of this layer is on signal drivers and receivers, design of the input and output buffers, and pipelining on long global interconnects. Because of the decrease in the noise margin due to low voltage swings in submicron technology, reliability of the interconnects has become a grave concern for the NoC architects. For the same reason, the *data-link layer* has been introduced to the NoC's function layers between the network and the physical layers. It addresses all the reliability concerns of the interconnection networks.



(a)



(b)

Figure 2.20: Most widely adopted NoC topologies: (a) Ring, (b) Butterfly [126]

Data-link Layer

At this layer, the data packages traveling on the long interconnects are protected against radiation induced transient faults, and electrical noise due to cross-talk. A wide range of error-detection with retransmission, forward error correction (FEC), and temporal redundancy approaches have been presented. The objective of all these approaches has been to come up with a fault-tolerance mechanism that does not incur an enormous performance and energy overhead. While some authors have advocated the use of switch-to-switch (S2S), also known as hop-by-hop (HBH), data protection, the others have argued for end-to-end (E2E) approach. The latter is based on the law *make the common case fast*, since an error free communication is expected to happen more often than the erroneous ones, the E2E approach proves very efficient avoiding several error detection checks. However, the S2S approach is preferable because of its simplicity: 1) a retransmission request only needs to make one hop to its immediate neighbor rather than going all the way to the source, 2) the errors will normally be contained within a packet; its recovery does not affect other packets in transit. This type of protection already suits the packet-by-packet communication style always adopted in the NoC architectures.

In the coming chapters our emphasis shall be on the last four layers: We present fault-tolerant designs for a few critical components within the last three layers, and describe a formal methodology to classify flow control mechanisms for ANoCs. Subsequently, we propose our own flow control mechanism that minimizes the number of transitions on the global interconnects, thereby resulting in smaller dynamic power consumption. Meanwhile in the sections to follow within this chapter, we provide a background on different flow control mechanisms, routing algorithms, reliability issues within the ANoCs, and conclude the chapter with an overview of the state-of-the-art ANoCs.

2.2.4 Flow Control

Flow control determines how efficiently the network resources, such as channel bandwidth and buffer capacity, may be allocated to the arriving packets. In case of competing requests, the flow control resolves the contention by allocating the resources to one candidate, while tackling the blocked packet by other means: This choice is something that categorizes the flow control mechanisms into two. The first category is the *bufferless flow control*, in which the blocked packet may either be dropped, and thus requires the source to resend that packet later, or simply rerouting the blocked packet to any other available channel. Clearly, the source of the blocked packet must ensure that it has a valid copy of the packet still available. In addition, flow control requires an acknowledgment or negative acknowledgment [30] signal between every pair of communicating nodes, so that whether the packet was discarded, may be communicated back to the predecessor node. *Circuit switching* is another form of bufferless flow control in which the header flit reserves the entire path, and then the body flits traverse the reserved path. In case of blocking, instead of being dropped, the header flit is forced to wait, and contend for the channel every cycle repeatedly, unless it finally gets the resources. On one hand, the circuit switching approach does not waste any resources (on a packet that is later dropped), on the other hand, it leads to high average packet latency, low throughput [30], and may indefinitely block the channel (it traversed to reach the current node) thereby prohibiting all other packets from

progressing that might wish to access one of the other output ports having free resources. Note that this type of blocking is equally applicable in other flow control mechanisms as well, and this is addressed in Sec. 2.2.5.

An alternative to the bufferless flow control is the *buffered flow control*. In this approach, the blocked packet need not be dropped or rerouted; while the winner is traversing the allocated channel with all the other resources in possession, the blocked packet can simply wait in the available buffer slots until the resources are once again available[4]. This scheme does add some overhead, but results in more efficient flow control in that the channel bandwidth is not wasted in rerouting or discarding a valid packet [30]. The manner in which the buffers and the channel bandwidth are allocated to packets, determines the type of buffered flow control mechanism. These types are overviewed next.

Packet-Buffer Flow Control

In this type of flow control, the resources (buffers and channel bandwidth) are allocated to entire packets. The *packet-buffer flow control* is further divided into two classes: *Store-and-Forward* and *Virtual Cut-Through* flow controls. In the former, the resources are allocated to an incoming packet only once it has been completely received. Thus, its transfer to the subsequent node may only commence after it has been completely buffered and resources allocated, leading to high average packet latency. In contrast to this, in the virtual cut-through flow control, a packet may be transferred as soon as the header flit has arrived. While the header flit progresses through, the body flits keep following up. In this manner, this class of the buffered flow control achieves better average packet latency as compared to the other alternative. However, once again the channel bandwidth and the buffers are allocated to the entire packet, thus requiring the intended receiver to have sufficient space to hold the entire packet – inefficient use of buffer space. Secondly, whenever the channel bandwidth is allocated to an entire packet, high priority[5] traffic may be forced to wait for a long time before acquiring the channel [30].

Wormhole Flow Control

The *wormhole flow control* operates exactly like the virtual cut-through does. The only difference is that, in this type of flow control, the resources are allocated to flits rather than the entire packets. The header flit reserves a one-flit buffer, and just one flit of the channel bandwidth is needed. This means, in case of competing requests, all the contenders can access the physical channel on a flit-by-flit basis, (ideally) without having to wait for a long packet latency. The body flits simply follow the path reserved by the header, where the tail flit is responsible to release all of the reserved resources. With just one input buffer as a requirement, this flow control provides efficient buffer space utilization. Having more than one input buffer can significantly increase the throughput though, provided there is a sufficient number of pipeline stages on the physical channel; this shall be discussed in one of the coming chapters.

[4]Normally the number of buffers is sufficiently large to store an entire packet, thus avoiding a possible buffer overflow.

[5]Priorities are considered in the context of Quality of Service (QoS). Throughout our work, we do not consider priorities for any type of traffic.

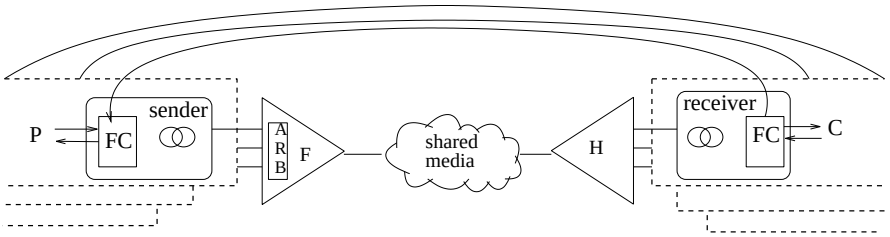


Figure 2.21: Overview of Flow Control: Producer (P), Consumer (C)

Consider a case where a flit on the physical channel fails to find a buffer space in the receiver (due to congestion at the physical channels on intended subsequent hops, say towards East). It will keep the physical channel blocked until a space has been finally created, no matter there are other packets that may use the available bandwidth to reach their respective destinations (say, North, South, etc). This indicates that the wormhole flow control has a potential to block the channel indefinitely, leading to extended average packet latencies. This type of blocking is called *Head-of-line* blocking. This drawback is circumvented by employing multiple virtual channels (VC) per physical channel, introducing a scheme called *virtual channel flow control*.

Virtual Channel Flow Control

The virtual channel flow control resolves the problem of channel blocking by associating the physical channel to multiple VCs. While flits belonging to one packet fail to progress due to traffic congestion, this flow control allows other packets to share the channel bandwidth that would have remained idle in the other flow control mechanisms. However, this requires the channel bandwidth to be properly decoupled from the channel state, otherwise, flit(s) of a blocked packet (in possession of the physical channel) would simply not let other packets use the idle bandwidth. Decoupling the channel means that as soon as a flit is stored in an available buffer slot, the channel must be free for other packets to use, no matter it gets the resources it requested for or not. This can be ensured only by providing the sender knowledge about the buffer state on the correspoding VC in the receiver, so that if there is not sufficient space to hold further, the sender must not apply any more flits. This phenomenon, of avoiding buffer overflow on each VC, is termed as *backpressure management*.

2.2.5 Backpressure Management

As stated above, the backpressure management schemes are essential to avoid buffer overflow in each VC. There are several algorithms that have been proposed to perform this task. One thing that all those algorithms have in common is their structure, shown in fig. 2.21. To this end, the common interconnect, called *shared physical link* (SPL) requires circuitry to merge data streams from different channels including arbitration of access requests, the *funnel* (F), and at the receiving end a demux distributing the data, the *horn* (H). The circuits required for flow control are called the *sender*, connecting the producer with the funnel, and the *receiver*, connecting the consumer to the horn.

To facilitate flow control, information about the receiver's buffer fill level must be conveyed to the sender requiring an additional link in the reverse direction (*upstream*) of the main data link (*downstream*). This information has to be available separately for each VC, requiring either one link per VC or again merging those links into an (upstream) SPL.

There are several VC flow control schemes for links in NoCs employing different ways of upstream communication. On some occasions, the senders are made aware of the total amount of space available with the receiver, so that they do not block the SPL by sending more flits than the receiver can take. Generally this scheme is termed *credit based communication*, the implementation details may slightly vary though. In [43] the authors proposed to make use of the same strategy, but dedicated counters were employed for each VC at the sender, that would always keep a count of transmitted flits. These counters were decremented once a flit was submitted to the scheduler, and incremented once a credit was received back from the receiver, indicating an available slot. Alternatively, a credit-uncredit approach was presented in [15], in which each sender comprised a credit FIFO containing as many credits as the number of buffers available in the receiver's data FIFO. These *credits* are tokens, each of which (virtually) flows to the receiver with a transmitted flit, and (physically) returns to the receiver over the credit link once a flit has been consumed. Another interesting access control scheme, called share-unshare, was presented in [15]. Here the authors got rid of the complex credit and data FIFOs, and circulated a single token between a pair of communicating nodes. The sender could not send the next flit until the credit had been received. Although the methodology had a poor bandwidth utilization, the design was much simpler, and energy efficient.

On other occasions [10, 11, 36, 38, 115], the senders do not have knowledge of the storage capacity of the intended receivers, they simply keep transmitting until they are explicitly stopped. While these so-called *ON/OFF* protocols [30] reduce the traffic on the upstream channel, their implementation in NoCs must satisfy relatively critical timing assumptions.

One disadvantage of flow control schemes for VCs is that for each transmitted flit a token denoting its removal from the receiver's buffer needs to be sent back. This inevitably means that the upstream link has to provide the same total bandwidth (for credits, obviously data are not supposed to be sent back) as the downstream link. In the following chapter, we will introduce a credit scheme that achieves lower bandwidth requirements for the upstream link.

2.2.6 Routing Algorithms

In the networks of multiple routers, data must reach their destinations via several hops (intermediate routers). The set of routers that a specific packet goes through to reach its destination, determines the *path* or *route* of that packet. For each packet, the path is selected by some protocol, called *routing algorithm*. Various routing algortihms have been proposed, each having its advantages and shortcomings, however, the principal distinction between these lies in their deterministic or nondeterministic nature. In the former, there is always a fixed route between a specific pair of nodes that the packets follow, thereby enabling a rather simple design of the router. Usually, the source node (the point of origin of the packet) incorporates the entire routing information within the header flit of the packet (called source encoding), i.e., each switch on the way knows to which output port it must guide the incoming (header) flit. While the header flit progresses towards its destination, each node keeps storing the relevant routing information

for this packet, so that the successive flits (that arrive without any routing information) may be swiftly guided to the correct output port. Because of the predetermined path, the average flit latency per switch is generally very small when using these algorithms, however, the unbalanced nature of traffic on channels may lead to contention, and hence, poor throughput. The unbalanced nature of traffic means that there may be several channels that are idle, while a few have full utilization, and hence becoming a bottleneck.

On the other hand, in nondeterministic, sometimes also refered to as adaptive, routing, the subsequent hop (jump to one of the neighboring nodes) is determined on the fly, depending upon a certain criterion, one of which is traffic congestion on links. The routers supporting such adaptive routing mechanisms are said to have *load balancing* strategies employed. Although this feature allows efficient and dependable routing strategies, this leads to complex router designs, and some major problems associated with this approach are *out-of-order arrival* of flits at the destination, *livelocks*, and *deadlocks*. Addressing the first is not so complicated: An additional flit identifier field needs to be incorporated within each flit, that would help the destination node rearrange (sort) the arrived flits. As understandable, this requires the width of each flit to be increased, consequently, increasing the complexity of the overall network architecture.

A livelock refers to an undesirable scenario, where a flit might get stuck (everytime rerouted) in a loop formed by a certain set of routers without progressing towards its intended destination. Resolving this once again requires addition of a state to each flit, which keeps a count of rerouting on each router, and beyond a certain threshold, a flit is not allowed to be rerouted. *Age-based priority* is another method to address this issue, in which the oldest flits are always given the priority in case of conflicts (competing requests).

A deadlock in the network occurs when a set of flits wait on one another to release a resource, forming a cyclic dependency. Adaptive routing, which is based on the concept of rerouting, can very often form such cycles, and end up in a deadlocked network. Various deadlock free routing mechanisms are available in literature, however, the most widely adopted approach has been to restrict the turns for a given flit. For example, the XY-routing algorithm does not allow a flit/packet arriving on north/south input ports to go on east/west output ports, i.e., a given flit/packet must first move towards the destination in the hortizontal (X-axis) direction (until it is in the same column of routers as the destination), followed by moving in the vertical (Y-axis) direction. This way, a dependency cycle can never be formed. The permissible turns in XY-routing are depicted in fig. 2.22. Similarly, ODD-EVEN turn models [24], and West first routing mechanisms guarantee deadlock free routing as well [148].

Since, in this work, building a high performance NoC is not the ultimate objective, we have decided to stick to the simple deterministic source-encoded, XY-routing algorithm, that guarantees deadlock/livelock free routing. Details on our baseline NoC architecture shall be provided in the following chapter.

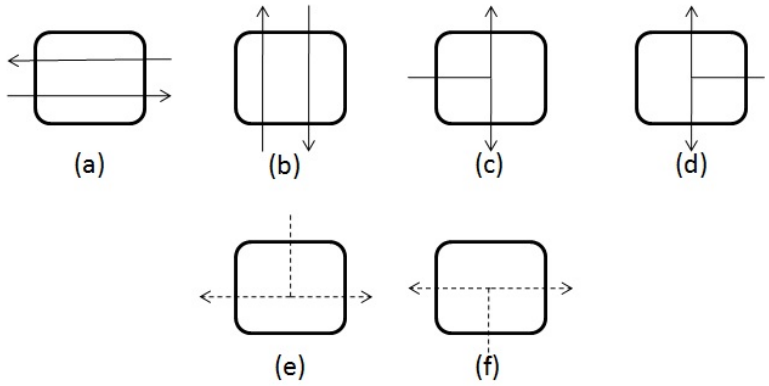


Figure 2.22: Permissible (a, ... ,d) and forbidden (e, f) turns in XY-routing algorithm

2.3 Communication Infrastructure for MP Platforms

2.3.1 Globally Asynchronous Locally Synchronous Systems

The global asynchronous communication paradigm has attracted the designers for its features detailed above. However, the fact remains that plentiful CAD support is available for synchronous circuits and systems, which take much shorter to design as compared to their asynchronous counterparts. In view of these advantages, use of partial asynchrony was proposed [22]: The router-router (global) communication was done asynchronously, and the routers were made as conventional synchronous circuits. While the asynchronous communication got rid of the global clock and its distribution problem, the synchronous routers could easily operate with a local clock that was anyway needed for the computing resource (forcing each router to run at a different speed, thereby leading to multiple clock domains across the chip). This arrangement of synchronous islands (routers) communicating using asynchronous means, is given the name *globally asynchronous, locally synchronous* (GALS) design paradigm. While the global interconnects reduce the design difficulties [121], the synchronous routers may be easily implemented using the design tools available. The elastic nature of asynchronous global interconnects allows easy integration of multiple clock domains on a single chip.

2.3.2 Asynchronous NoC

The ANoCs are a special case of GALS, in which even the routers do not share the local clock with the computing resource. Instead, a synchronous-asynchronous interface circuit is installed within the NI on each node [119], which would allow the synchronous computing resource to communicate with the asynchronous environment, the router. This approach seems more promising as compared to the conventional GALS in that the data while traversing the intermediate switches along their path do not need to cross several clock domain boundaries [120], thereby eliminating the need for synchronizers that would have been there on every IO port in the switch while using GALS approach. This significantly reduces the area overhead, and dynamic power dissipation across the entire network. Several ANoCs have been developed,

mostly in the academics SpiNNaker [100,122], MANGO [14–16], QoS [42,43], FAUST [10,11], QNoC [36, 38, 115], [56], a few times commercially [4]. Considering the potential of asynchrnous design style discussed above, we have also decided to build our NoC using the same, fully asynchronous approach. However, as mentioned earlier, asynchronous circuits prove to be more vulnerable against soft errors, therefore, considering and adressing their reliability issues has become a critical challenge for the designers.

2.4 Reliability Concerns in ANoCs

Proceeding miniaturization has allowed digital systems to operate at ever increasing clock rates, thus boosting their performance. At the same time the miniaturization process has made these circuits more susceptible to faults, in particular transient faults like radiation particle hits, crosstalk or electromagnetic interference. These faults typically cause undesired voltage pulses of very limited length. Since the rate of transient faults is thus in general rising, fault tolerance techniques, as known from highly critical and aerospace systems, receive attention in more general applications as well. This concerns the NoCs as well. Fault tolerance is therefore predicted to become a crucial property in context with future technologies.

Protecting an ANoC requires fault tolerance on two axes: asynchronous communication on global interconnects, and, the asynchronous components that constitute a router in general. Dally and Towles [30] listed some failure modes and fault models for a typical interconnection network. The list included transient bit errors on channels, soft errors on memory modules and logic circuits, stuck-at faults, fail-stop faults and/or Byzantine faults. These faults, typically, would lead to data corruption, but in the worst case, a (finite) number of flits (or packets) may go missing, sometimes resulting in orphan flits within the network, which require a mechanism to drop those flits, and retransmit them from the source.

In case of hard faults within the router components, and/or link break down, it is essential to have a fault-tolerant routing algorithm that would bypass the faulty node or link, and establish an alternate (deadlock free) path to reach the destination. Several such algorithms exist in literature [34, 55, 59, 64–66, 101], however, in this study we do not focus on hard faults, and restrict ourselves on soft errors on the channels, and logic circuits only. Therefore, we do not provide any robust routing strategies in this work. Generally for a given system having a total of 6,144 channels, and a channel rate of 10^{10} bps, it is observed that the *soft error rate* (SER) on off-chip links may be between an error every 16 seconds and two errors per month, depending upon the reliability of channels, the *bit error rate*(BER) of which, may vary between 10^{-15} and 10^{-20} [30]. These figures surely indicate how essential it is to have some protection against transient faults in the NoCs as well.

The number of ANoCs that somewhat addresses the reliabilty concerns mentioned above is unfortunately scarce. There have been some contributions towards fault-tolerant asynchronous communication on global interconnects [73, 98, 122], however, the only significant work that employs fault-tolerance features both on global interconnects, as well as the fully asynchronous router, was by Imai and Yoneda [59]. Pontes [108] has also presented schemes for hardening different components in a GALS NoC against soft errors. We shall review a few ANoCs in

context of flow control in the following chapter, here we will briefly discuss some of the works towards fault-tolerance in fully asynchronous routers and/or on global interconnects.

2.4.1 Error Control on SPL

The global interconnects are usually protected by the logic circuits, augmented on the two ends of the communication link, working together, the so called encoder and decoder. The protection is a three fold procedure: detection, recovery, and link shut down. The detection may easily be done using a parity bit, but normally, a *cyclic redundancy check* (CRC) of sufficient length is employed, capable of detecting multple bit errors. The recovery is normally done via retransmission of the faulty flit (packet) – requesting the sender for a retransmission using a separate *retransmission request* signal. Furthermore, the channels may have explicit counters that count the number of faulty receptions on each channel. Once a certain threshold has expired, the channel must be considered as permanently faulty and shut down. In this work, we do not consider permanent faults. The details on the error- detection and correction codes follow in Sec. 7.1.

2.4.2 Fault-Tolerance in Routing Components

Since we have decided to build the entire router using asynchronous components, it is not possible to make it fault-tolerant without considering the fault sensitivity of the elementary components used at the lowest level in its design. Considering the input buffers for example, they not just require their datapath (latches) to be protected (which may be done at E2E level using some FEC schemes), the protection of their control path (formed as a Muller pipeline) is even more essential, going to the level of an MC. An MC, having an internal storage loop, has this tendency to make a transient fault an upset: when the inputs to the MC are at different logic levels, any transient fault may manifest itself in the storage loop, inverting the state of the MC. Since it is a component that is always seen in the pipelines and other critical designs, an upset in one MC can thereby cause the entire design to malfunction, or worse, lead it to a deadlocked state.

Shi [121] presented a fault sensitivity analysis of the MC (originally done in [84]) and various asynchronous pipelines. It was shown that the sensitivity of a 2-input MC to a transient pulse on one of its input ports is 1/2, since out of the total of four possible states, two can respond to a transient fault by flipping to the incorrect state. For example, if an input port of an MC receives a short *up-down* pulse ($\uparrow\downarrow$) while the second input is already high, it can conveniently flip the output state to high, however, the second input if faulty would not make a difference. Using the same explanation, it may be concluded that the sensitivity of the Muller pipeline is the same as that of a 2-input MC, since each stage consists of a single MC. Since the focus of our work is just the 4-phase bundled data pipeline, we simply do not consider the sensitivities of other pipelines. Details on the fault sensitivity of the 4-phase bundled data pipeline will follow shortly, however it is worth mentioning that such faults can create superfluous tokens in the pipelines, or may lead to the loss of correct tokens. In any case, it is essential to protect asynchronous pipelines.

Usually the methods used to make circuits tolerant against transient faults include various types of redundancy. *Triple Modular Redundancy* (TMR) is a form of *hardware redundancy*, in which the desired operation is performed by three units, and the decision is based on majority voting [49]. Sometimes the hardware redundany causes enormous overhead, replicating mem-

ory units for instance. On other occasions, replication based approaches simply do not work correctly, in nondeterministic circuits for example. We will later show such a scenario for the TAC. *Information redundancy* requires that more information must be provided to the computation unit than what is adequate, i.e., it must not rely on one source of information. *Temporal redundancy* requires the information to be computed multiple times. A fault in one computation is likely to not show up in the subsequent computations. Sometimes it is essential to use a hybrid methodology that adopts all or some of these mechanisms. In our work, we also make use of all of these redundancies to protect the data- and control- paths, and argue which of them is the best under given circumstances.

Asynchronous Transient Resilient Links

Ogg et al. have proposed delay insensitive communication links for ANoCs resilient against single event transients [98]. The number of wires used to encode *n* data bits is $2n + 2$, where two additional bits represent a reference symbol beside dual-rail codes. For each data symbol, a reference symbol is also generated and transmitted to the receiver, which upon reception compares the two symbols. If they are found to be in-phase, the data bit is decoded as 0, and if they are 180 degree out of phase, the bit is considered to be 1. A phase difference of 90 degree indicates an invalid data, corresponding to an error. It is mentioned that a fault that occurs during the setup time of a flip-flop on the receiver may go undetected, and operating at a frequency beyond 1GHz, the probability of undetected faults approaches 1. The links are synthesized for $120nm$ technology, resulting in an area cost of $409\mu m^2$ per bit, and energy per bit of $356fJ$. The latency through the link is found to be $800ps$, and a maximum operational frequency of 1.056GHz.

Fault-Tolerant DI Codes for GALS Setup

A single/double error correction with DI global communication for GALS setup has been recently proposed by Lechner and Najvirt [73]. Just as was done for SpiNNaker, they have combined delay insensitivity with error detection codes; what is different here is that once a fault is detected, the receiver samples the data from the global interconnect once again rather than requesting a retransmission. At the output port of the node, the data to be transmitted are encoded for error detection (parity bit or Double Error Detection Hamming codes), followed by DI encoding, and then submitted onto the global interconnect. Once new data are available at the input port of a node, they are first checked for completeness, and DI decoded simultaneously. The output of the completion detection circuit is responsible to generate the local clock, which stores the decoded data in a register. After this, the error detection decoder checks for any possible faults, generating an error signal, which, when 1, causes resampling of the input data. If the decoder does not indicate an error, the control unit generates the acknowledgment, completing the handshake protocol. It is assumed that a transient fault on the links will disappear in a finite number of decoding cycles. The implemented circuits are mapped to UMC $90nm$ standard cell library. The experiments are performed using various data widths, and a comparison, in terms of various metrics, between 2-phase and 4-phase dual rail protocols is given. It is shown that the latencies (encoder and decoder pair with zero wiring delays, single parity bit, and 16bit wide data) for

2-phase and 4-phase protocols are $1.27ns$ and $1.37ns$ for the fault free case respectively, which increase to $1.78ns$ and $1.92ns$ for the faulty case with one time resampling respectively. The area utilization for the two protocols in the same order is $1767\mu m^2$ and $1642\mu m^2$ respectively.

Dependable Fully Asynchronous On-Chip Networks

Imai and Yoneda have proposed a complete design of a dependable ANoC architecture [59]. Their contributions include a novel dependable routing algorithm, which is capable of detouring a faulty link or a router by using the local fault information communicated to all the neighbors of the faulty router. Furthermore, the algorithm analyzes the traffic on all the links of immediate neighbor nodes to adaptively choose the next hop. The routers are based on transition signaling internally, and are protected against SETs mostly through duplicated logic. The links are LEDR encoded, and a novel time-out mechanism is proposed to detect permanent faults on them. The elegant design is accompanied with an in depth evaluation: Four designs are implemented for $130nm$ technology, including a non fault-tolerant asynchronous design and its synchronous equivalent, and two fault-tolerant versions, first with duplicated control-path, and the other with the data-path duplicated as well. SETs of length up to $5ns$ are randomly injected with a constant rate of $1\text{x}10^{-10}$ per $\mu m^2.ns$ with flits injected at $0.067flits/ns/node$. It is shown that the synchronous design costs the smallest area and latency, with a failure rate of 2.86, better than the two asynchronous designs in almost every aspect (except for the failure rate which is almost the same as the asynchronous version with only duplicated control). The fully duplicated asynchronous version reduces the failure rate to 0.897 with a tremendous area and latency penalties (area reaching almost three times than the unprotected designs, and two times in latency) making it an infeasible option.

A few loop holes that we could identify in their work includes: 1) the global interconnects are not protected against transient faults, hence there is no retransmission mechanism provided, 2) the authors have correctly pointed out that the arbiters are such circuits that cannot be protected by hardware duplication, but the solution they have provided is also not bullet proof. They have simply duplicated the input requests that are joined using a single MC, making it a single point of failure. In our work, Ch. 6 we will demonstrate a step-by-step design of an arbiter that guarantees elimination of all single point of failures, the performance, however, is slightly compromised.

2.5 Major Contributions of this Work

We have identified some outstanding issues in ANoCs especially within the control path that either have not been sufficiently addressed in the available literature, or have been historically misunderstood. In the next chapter after presenting the design of our own asynchronous router with VC support, we formally describe the requirements of building a functionally correct flow control mechanism, and point out a deficiency in a widely adopted scheme. We conclude the chapter by proposing and evaluating our own flow control mechanism that fulfills all the requirements that we had pointed out in our framework earlier in the same chapter.

Other than the flow control mechanism, we also propose a novel high speed switch controller (arbitration circuit) to support our router. The proposed arbiter makes use of internal pipelining, which allows other clients to resolve the next winner (the one who accesses the output port) while the current is still accessing the shared resource. This significantly improves the throughput of the router, as shall be presented in the fourth chapter.

The remainder of the thesis is strictly dedicated to addressing some of the fault tolerance issues within the ANoCs. Broadly, fig. 2.18 depicts all the major areas that we want to emphasize on in this work. We begin with the protection of the control path for the input buffers in the fifth chapter, where we present a systematic approach to harden the 4-phase handshake controller. Unlike the conventional approaches of duplication, we are specifically concerned about resolving all single points of failure, and make use of model checking to verify the correctness of our approach.

While a switch comprises of several relatively simpler asynchronous components, such as mux and demux that can be protected using replication methods, we rather concentrate on hardening the switch controller, which is a nondeterministic circuit, and cannot be protected using replication. Hardening an arbiter is another topic widely misunderstood; in our work we not just point out problems with a few existing solutions in this respect, but also present a step-by-step hardening methodology for its protection against soft errors that once again guarantees elimination of all single points of failure. This is covered in chapter 6.

Although the payload (data traversing the SPL) may easily be protected using any E2E FEC scheme, in chapter 7 however, we propose a simple error detection with retransmission method that suits our ANoC. Our approach is based on the double error detection (DED) algorithm, but we aim to improve its performance by avoiding unnecessary retransmissions, leading to a simple, yet an effective methodology.

In chapter 8, we build the complete FT switch, compatible with all the individual FT components discussed above, before we conclude the thesis in chapter 9.

CHAPTER 3

The Baseline NoC Design

The design of a novel ANoC is presented in this chapter. We specifically emphasize on three important aspects: 1) Designing a fully asynchronous router with VC support, 2) elaborating a generic framework to formulate requirements for building a correct VC flow control mechanism for ANoCs, and 3) proposing our own flow control scheme. Before we proceed with those however, in the next section we overview some of the distinguished relevant works available in literature.

3.1 Related Work

One of the earliest contributions to ANoCs was the QoS router proposed in [42]. It employed the 4-phase handshake protocol and 1-of-4 data encoding. The data bus was 32 bits wide, and the packet format, having an *end of packet* symbol, allowed variable sized packets to be transmitted. The NoC adopted the deterministic, source encoded routing algorithm, with the credit-based flow control mechanism [30] to avoid blocking the SPL. The sender maintained a count of the flits transmitted on each VC in a dedicated counter that was decremented upon submission of a flit to the scheduler, and incremented once a credit was received from the receiver. If the counter reached zero, the specific channel remained blocked. Dedicated wires were used to transport credits for each channel. The latter were arbitrated by means of a *static priority arbiter* (SPA), which would allow one channel (having the highest priority) to block the rest indefinitely. Thus each output port of the router provided QoS to one channel, and considered the rest to carry only the *best-effort* (BE) traffic (for which there were no hard guarantees).

FAUST [10], [11] also employed the 4-phase, 1-of-4 encoding, with source routing, and wormhole packet switching. It supported two service levels (SLs): *real-time* (RT) and BE, and the backpressure was controlled with the acknowledgment signal of the handshake protocol. Although the RT traffic could preempt the BE one, but this was one example where the concept of flow control had been misused: The SPL could be blocked indefinitely [37] until the receiver had consumed the token.

The authors of [115] proposed the QNoC asynchronous router that also made use of 4-phase handshaking with 1-of-4 encoding, wormhole packet switching, and provided four SLs. A specific SL could acquire the SPL only if there was a free space available with the receiver. Hence, a separate *free space indication* (FSI) signal was required for each SL. The receiver employed a Muller pipeline, and the control signals associated with the first two stages were used to compute the FSI signal. The same authors in [38], [36] modified the QNoC to support multiple VCs per SL. However, none of their designs provided SI flow control; they had to make some stringent timing assumptions on the FSI signal [89].

Bjerregaard [15] proposed the MANGO architecture. The router was built having two sub-routers: one dedicated to guaranteed services (GS) traffic, and the other for the BE. The design adopted the wormhole packet switching, where each header flit traversed the BE router, and programmed the GS router (using the routing information) on every hop. This way the following body flits would always find a path already established, just as in the case of circuit switching. The handshaking was done using the 4-phase bundled protocol. In one of the other works [16], the same authors proposed a supporting scheduling algorithm to guarantee QoS in the MANGO architecture. Furthermore, the same authors proposed two VC access control schemes [14]; a brief overview of each of those is as follows.

In the first method, called *share-unshare*, a single token was circulated between sender and receiver per VC by means of an unlock wire: the latter toggled once every flit reception, i.e., only when the output handshake cycle completed. This generated a small pulse at the sender side, which allowed the share-box to complete the handshake cycle at the input, followed by placing another token onto the shared media. The authors acknowledged that in case of a *single eager VC* the bandwidth utilization was poor due to the long handshake cycle spanning the share and unshare boxes, shared media, and the unlock wire. Furthermore, the scheme scaled badly (same as for [42], [115]) with the number of VCs, since each of them needed an independent unlock signal [15]. However, one transition per flit reception on a single *unlock* wire made the scheme extremely energy efficient.

The other scheme that the authors proposed in [15] was an efficient credit-based flow control mechanism, in which the sender maintained a credit FIFO in the *credit box*, which lost a credit with every submission of a flit to the scheduler. The credits returned to the sender as soon as the flits left the *uncredit box* at the receiver side. This flow of credits still maintained the handshake protocol, requiring two wires per VC, which turned out to be even worse in scaling as compared to the previous approaches. The authors proposed to merge the credit links for all the VCs on to a single shared credit channel, which reduced the number of wires spanning between the two communicating nodes.

3.2 Baseline Router Design

Note that most of the ANoCs presented above have a certain features in common, such as, wormhole packet switching in which the header flit containing the routing information encoded by the source node reserves a path towards the target node, and the body flits simply progress without having to wait on every hop. Because of its simplicity, efficiency, and smaller input buffer requirements, the wormhole packet switching technique has become the standard in ANoC ar-

chitectures. The best suited routing algorithm for this packet switching approach is the deterministic XY-routing, since it tends to keep the packet latency minimal (obviously having a few drawbacks as well). Moreover, all the NoCs made use of VCs with a flow control mechanism to provide GS to a certain class of traffic. Finally the topology that each ANoC adopted was the 2D mesh of routers due its symmetric and easily scalable structure.

To be able to compare our architecture with the existing solutions, we have also decided to stick to the same features with 4x4 routers arranged in a 2D mesh. The only difference between our work and the existing solutions is that we argue about the number of VCs to employ considering their overhead on area, and complexity of the switch and the VC allocation controllers. A router may have a miximum of five IO ports: east (towards right), west (towards left), north (upwards), south (downwards), and local (connected to the PE). We assume that the NI divides a packet into two or more flits, the first of which is always a header flit containing the routing information. There can be three types of a flit, header, body and tail. Table 3.1 presents the chosen flit- size and format for all the flit types. The two little-endian, most significant bits (MSBs) indicate the type of flit. '00' represents a header flit, '11' is reserved for a tail flit, and '01' and '10' for the body flits. Since we keep an explicit specifier for the tail flit, we do not need an extra adder to count the number of flits that have arrived. In addition, this gives a flexibility to transmit and receive packets of variable sizes, just like the QoS router [42]. The addressing scheme that we have adopted is influenced by MANGO [14], in which each pair of bits, starting from 31 down to 0, in the header flit, indicates the next hop; and thus each pair needs to be removed/rotated on every hop so that the next pair can indicate the next subsequent hop. For instance in Table 1(A), '00' at positions 31:30 tells the switch that the incoming data has to be directed to east. Therefore, '00' after being written into the destination latch, must be rotated, thus bringing '10' at its positions. Subsequently, at the next hop, the switch will direct the incoming data to north corresponding to the pair '10'. In the same manner, '01' corresponds to the west output port, and '11' to the south. If the next hop is identical to the input port, then the packet is assumed to be directed to the PE; therefore, backtracking [66] is not supported.

Fig. 3.1 presents the block diagram of our router. Primarily, the overall design is divided into two parts: An Input Handler (IH) and an Output Generator (OG) per input port and output port respectively. The former is responsible for capturing input data, and switching them to the appropriate OG. On the other hand, the latter comprises an arbiter module for each output port, which is reserved with an incoming request from the corresponding IH. Once the requested arbiter is reserved, the OG guides the incoming data to the appropriate output port. Note that this setup only corresponds to one input-output (I/O) port. The same needs to be replicated for each IO port. In the following we briefly describe the submodules building IH and OH.

3.2.1 IH – Flit Categorization Logic (FCL)

As explained above, the two MSBs indicate the type of the incoming flit. Depending upon the type, the flit has to be directed to the appropriate unit. The Flit Categorization Logic (FCL) is responsible to: i) identify the flit, ii) report its type to the Input Controller (ICON), and iii) guide it to the output handler through DeBS, which only operates on header flits.

Table 3.1: Packet Format and Size: (a) Header Flit, (b) Tail Flit, (c) Body Flit

1	1	1	0	...	0	1	0	0	0	0
Bit-0	1	2	3	...	28	29	30	31	32	33
Dest-16		Dest-15		...	Dest-2		Dest-1		Flit-Type	
(a)										

1	0	0	0	...	0	0	1	0	1	1
Bit-0	1	2	3	...	28	29	30	31	32	33
Payload									Flit-Type	
(b)										

0	0	0	0	...	0	0	1	0	1	0
Bit-0	1	2	3	...	28	29	30	31	32	33
Payload									Flit-Type	
(c)										

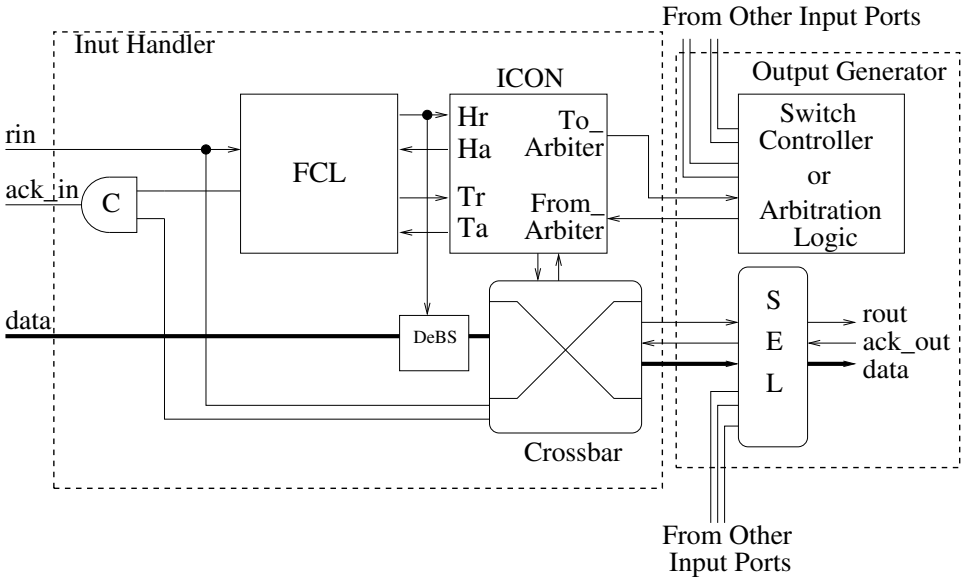


Figure 3.1: Block Diagram of the Async Router

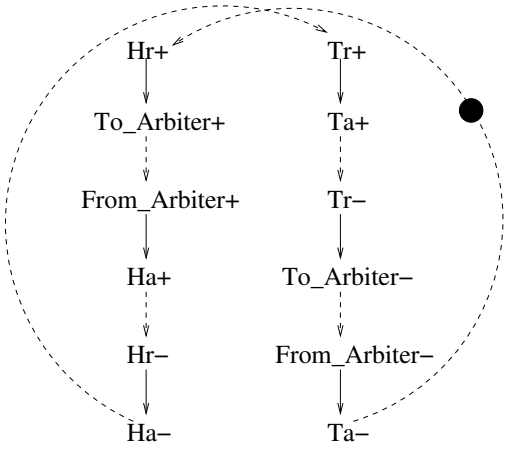


Figure 3.2: STG of ICON

3.2.2 IH – Destination Bits Shifter (DeBS)

The DeBS module is responsible to rotate the bits 31:30 of the header flit to the least significant bit (LSB) places, so that the new pair at places 31:30 indicates the output port of the succeeding node.

3.2.3 IH – Destination Bits Latch

Since the header flit is the only flit to have the destination bits, these bits need to be stored in a latch until the tail flit has been switched, and the other resources released. This latch is made a part of the crossbar, and is enabled with the grant coming from the associated arbiter to the ICON, i.e., once an arbiter is acquired, the destination bits may be safely latched, and reset as soon as the arbiter has been released.

3.2.4 IH – Input CONtroller (ICON)

Two important functionalities that ICON is made to perform are: 1) on-demand reservation of the arbiter associated with each output port, 2) generating the latch-enable signal for the destination latch. The ICON has been modeled as an STG in Workcraft [107] and synthesized using Petrify [28]. In fig. 3.2 we have presented its STG along with explanation of the variables used in Table 3.2. Its gate level schematic is presented in fig. 3.3. Following is the brief description of its operation.

At the arrival of the header flit, a request is raised and sent to the ICON, so as indicated by $Hr+$ in the STG. The arrival of the header flit must be followed by the reservation of the arbiter. As a result, a request is sent to the arbiter associated with the target output port. This is indicated by $To\_arbiter+$. Once the grant from the arbiter is received, $From\_arbiter+$, the destination bits must be latched. The body and tail flits proceed similarly except for the release

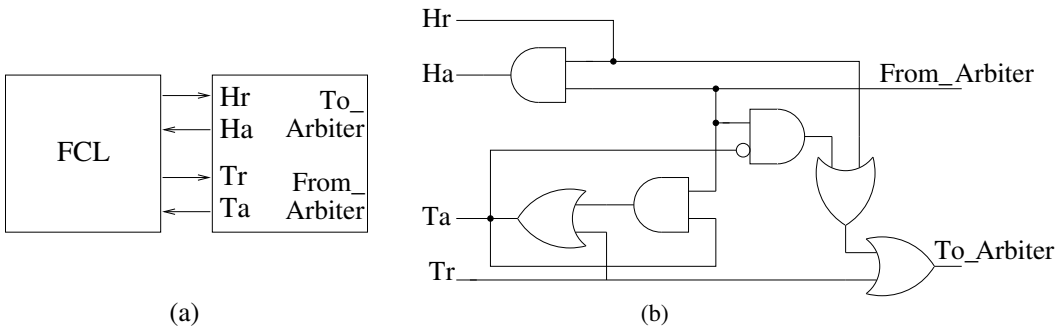


Figure 3.3: Schematic of ICON

Table 3.2: Variables used in fig. 3.1

Signal	Input/Output	Explanation
Hr	Input	request signal from the header flit
Tr	Input	request signal from the tail flit
From_Arbiter	Input	grant/ack signal from the arbiters
To_Arbiter	Output	req signal to the arbiters
Ha	Output	ack signal to the header flit
Ta	Output	ack signal to the last/tail flit

of the arbiter, which happens with the tail flit. This is done by lowering the request to the arbiter, $To\_arbiter-$, on receiving $Tr-$ from the FCL.

3.2.5 IH – Crossbar

The crossbar is implemented as a set of demuxs. Each input port is connected to one of the demuxs, which is controlled by a destination latch, write-enabled by its corresponding ICON. Each output signal of the crossbar submits the bundled data to exactly one of the OGs in such a way that the data (without control) reach the select module directly, while the output request signal goes through an MC where it is joined with the $To\_Arbiter$ signal from the ICON. The MC ensures that only one OG must be active for a given incoming packet.

3.2.6 OG – Output Port Arbiter

Each output port is equipped with an arbiter, which receives requests from all the input ports that may access that specific output port. For example, considering XY-routing, the East output port may only be accessed either by the West input port or the local PE. Therefore, a 2-way arbiter shall suffice. In contrast to this, the North, South, and local output ports require 4-way arbiters since all the input ports are allowed to access these ports. There are various designs of arbitration mechanisms available in literature; throughout this work, we have made use of

TACs, the design and description of which shall follow in the chapters to come. As far as its operation is concerned, all that an arbiter does is that it ensures exactly one winner (giving grant to just one IH out of all those requesting to the common output port), even if it receives several simultaneous requests. It may take quite long in reaching its decision though.

3.2.7 OG – Select/Merge

Once an IH has received the grant from the arbiter, it asserts a request signal with bundled payload going to the select module. The select module is the merge module presented in the background chapter. Recall that it requires only one active request at a time, which is already guaranteed by the arbiter discussed above. Therefore, the data with the active request make their way through to the output port.

3.2.8 Summary of Operation

Fig. 3.4 depicts the flow of a flit arriving at the Local input port through the router. The input request signal of an incoming packet (initiated with a header flit, on any input port) goes to the ICON through FCL, as well as to the switch. The latter guides the data towards its intended output port, and the ICON simultaneously sets the request to reserve the arbiter for that port. Upon acquisition of the arbiter, data propagate to the desired output port through a select module. Note that the allocation of a switch is done on per packet basis, i.e., until and unless a packet has completely traversed through the router, it keeps the switch locked. Had this not been the case, the flits belonging to different packets (from different input ports, willing to follow different paths on subsequent nodes, etc) could easily interleave, thereby requiring even more complex strategies to distinguish and reorder them on the receiving routers. Our scheme does bring forth a drawback in that if the packet currently in possession of the switch (output port in other words) fails to make progress, all others waiting for their turn, would in a way starve, causing the congestion to go upstream (path that led the packets to this node). As a remedy, we incorporate a few VCs and input/output (IO) buffers in our router.

3.3 Virtual Channel Design

Historically VCs have been used to serve multiple purposes: On one occasion, designers have used them to reduce the latency of the network by hopping over a few intermediate nodes [67, 69–71], on other occasions employed them to achieve fault tolerance [17, 20, 75, 147]. Mostly however, they are used to by-pass the slowly progressing packets on the SPL that would otherwise lead to extended delays, causing traffic congestion on the upstream as mentioned above, and even deadlocks on some occasions [1, 9, 30, 153]. Just as their utility, the number of VCs per node, and their allocation are the designer's choice. In what follows, we describe and justify our methodology.

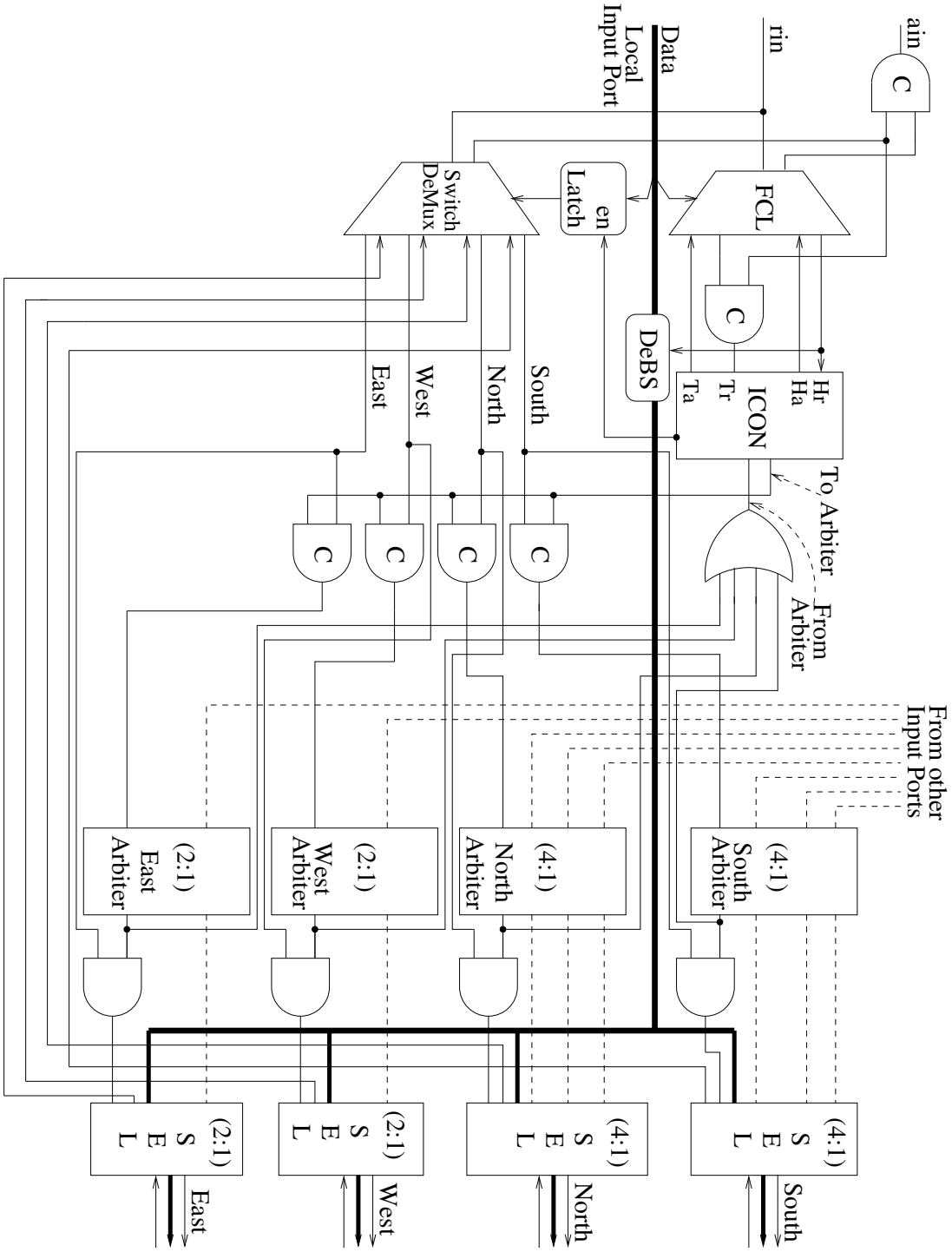


Figure 3.4: Traversal of a flit on Local input port

3.3.1 Number of VCs per IO Port

As mentioned previously, it is up to the designer to choose an appropriate number of VCs per IO port. However it must be considered that the more VCs, the higher shall be the area cost, the more complex the arbiters[1] shall be, which allow each VC to access the SPL, and the higher shall be the average packet latency since flits from different packets are to be time multiplexed on the SPL. In short, the choice of the number of VCs per IO port must be kept reasonable, and justified.

The routing algorithm may suggest the number of VCs to be employed, e.g., in the XY-routing algorithm, since turns from y-axis to x-axis are forbidden, the number of packets contesting for the y-direction will always be greater than those contesting for the x-direction. This suggests that the number of VCs in the two directions needs not be the same. However, which of the two directions must be given more number of VCs, once again depends upon the requirements of the design, and somewhat on the topology as well. We will keep our discussion restricted to 2D mesh of 4x4 nodes. One thing that is certain for now is that the network must be built generic: Assuming that the computing resource is neither capable of processing multiple incoming packets, nor can it generate multiple packets at a time, the local IO port will always have a single VC support.

Now consider the network presented in fig. 3.5. The routers at positions (1,1), (1,2), (2,1), and (2,2) have the maximum connectivity, i.e., five IO ports. On the other hand, the routers in the corners, i.e., (0,0), (1,3), (3,0), and (3,3) have the minimum connectivity of three, and the others have the medium connectivity of four. Beginning with the simplest, say (0,0), the north input port can only access the local output port capable of handling only one packet at a time. This means that the north input port may not need more than a single VC. By the same definition, the east input port must have at least two VCs since it has access to both local and north output ports. Applying the same method on node (1,0) would suggest that there must be two VCs on the south input port, and so on. Addressing the same for the west input port of (0,1) raises an important issue though: it requires three VCs to provide access to east, north, and local output ports, implying three VCs on the east output port of (0,0). However, note that the only source driving these VCs is the local input port of (0,0). Since it can only submit one packet on the network at a time, only one VC on the east output port will be used, hence this leads to wastage of resources. In order to keep the resources minimal, each node in the network must be analyzed and built separately with different VC requirements considering the number of sources feeding them, as well as the number of output ports they are going to feed on the neighbor tile. Fig. 3.6 depicts our methodology for the 4x4 network.

It is obvious that there may be several other solutions, and arguments in their support. All we have done is that we have employed minimal number of VCs that satisfy our criterion, i.e., every input port must always find an available virtual channel on its desired output port. Consider the node (2,2) for illustration: It is equipped with just one east input VC, since the local input of (2,3) is the only one to access it. It has two VCs on north input port since there are only two output ports that it can drive, local and south. The west input port has two VCs, since there are

[1] These arbiters are not the same as were used for switch allocation. Although the two functionalities may be performed together, but only at the cost of complexity of each arbiter, and the switch allocation would become even more complicated, as shall be highlighted in the next subsection.

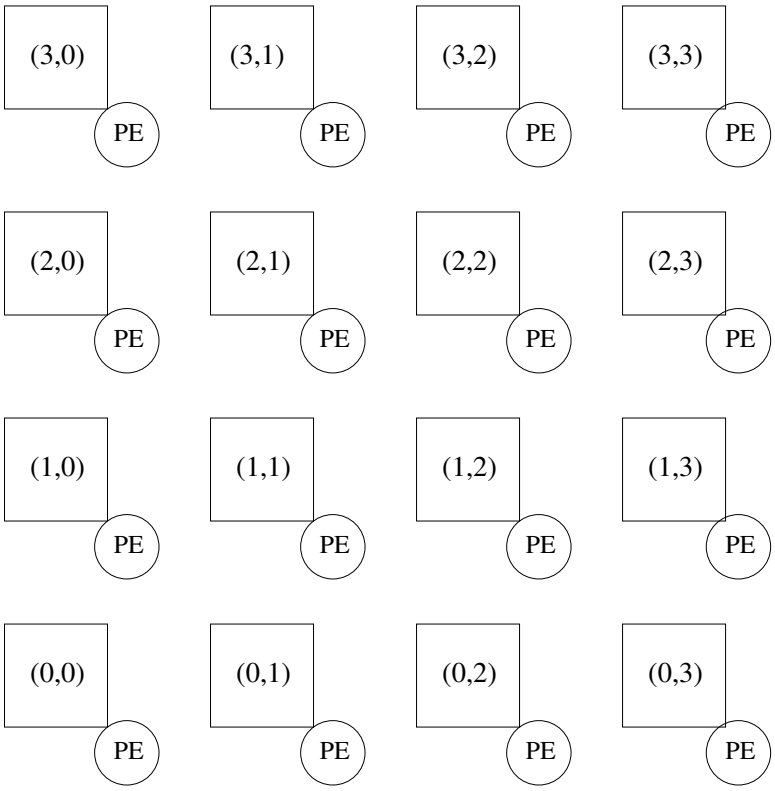


Figure 3.5: 4x4 2D mesh of routers

only two sources that can feed them, local port of (2,0) and local port of (2,1). Similarly, two VCs on south input port suggest that there are only two possible routes, local and north output ports. Similarly, all other nodes are built using the same aproach. In the next section we describe how the VCs are allocated to incoming packets, and the modifications necessary in each node to support their allocation.

3.3.2 Allocation of VCs

The foremost reason to employ VCs in our router was to ensure that no slowly progressing packet blocked the SPL indefinitely. This means each VC may only access the SPL for a limited time, and then wait for its next turn while the other VCs access it. Since there is no notion of time available in asynchronous design, the allocation of the SPL to each VC may rather be done on per flit basis. Therefore, each flit must acquire an arbiter, traverse the SPL, and release the arbiter subsequently. In such situations round robin arbitration is usually employed [14, 15], which guarantees fair allocation of channel bandwidth among the contenders.

From the previous discussion, it builds that the number of VCs between the output and input ports of a communicating pair of routers is the same, i.e., if the node (0,0) has two output

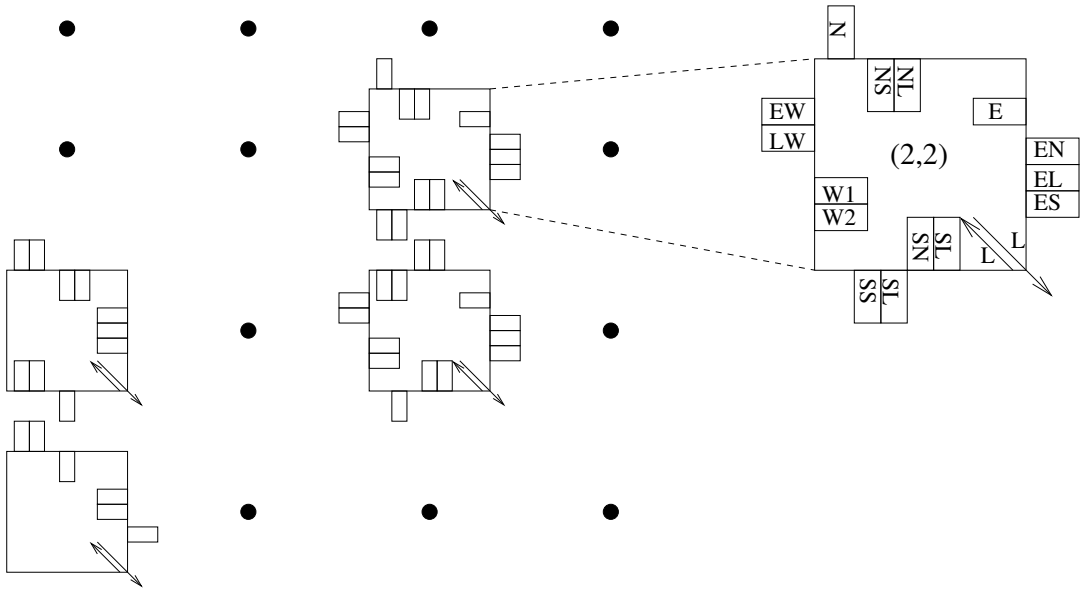


Figure 3.6: Minimal requirements in terms of VCs per routing node

VCs on its north port, then the node (1,0) has the same number of input VCs on its south port, for example. This makes the input VC allocation quite simple to implement: whenever a flit traverses the SPL, a bit called VC-identifier is appended on it, which informs the receiver about the VC it belongs to – a demux does this job conveniently. The allocation of the output VC on the other hand, is somewhat more complicated, and requires considering three scenarios:

1. If the number of VCs on an output port is the same as the sources that can feed them, then it does not matter how slow each packet progresses. For example, the two output VCs on north port on tile (0,0) are driven exactly by two sources, even if one of them is too slow to progress, the other will still get access to the free VC. Therefore, one-to-one mapping in those cases works perfectly.

2. If the number of VCs on an output port is smaller than the input ports willing to access them, then slow and/or blocked packets must not be allowed to retain all the VCs indefinitely. To resolve this matter to some extent, we define a rule that *the packets that are directed towards the same output port on the next hop, must be transferred one after the other on the same VC*. For example, if packets from local and east input ports on (3,2) wish to access the south port of (2,2), then both of them will be allocated the same output VC on (3,2), leaving the second VC available for the west input port in case it wanted to access the local port on (2,2). Therefore it is required to read the four bits (31:28) of each packet to be able to direct it to an appropriate output VC: the bits (31:30), as before, are needed for switching, the next two bits (they must not be rotated through DeBS) are used for VC allocation.

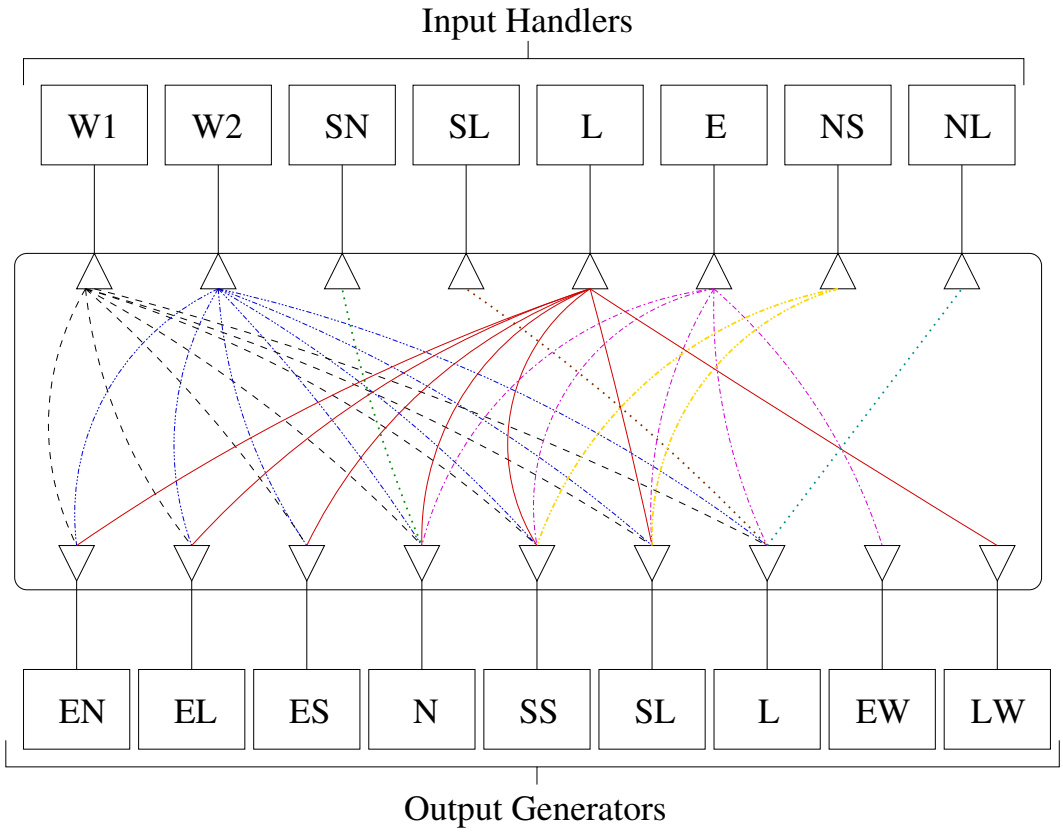


Figure 3.7: Connections needed on node (2,2) shown in fig. 3.6

3. If 1. holds on any node (n,m), and the VCs associated with one output port drive a larger number of output VCs on a neighbor tile (n,m+1) or (n,m-1), then (n,m) does not use the rule defined in 2. to perform VC allocation. Instead, any output VC can carry packets for any direction on the neighbor tile. For example, the east output port of node (2,1) has two VCs, and it may access the output port in any direction on (2,2). By following this rule (each output VC can carry packets for any direction), we increase the complexity of the output VC arbiters on (2,2), but we ensure that no output VC on (2,1) remains idle, which could have been the case with rule 2.

Summing everything up, our scheme has some pros and cons: Sticking to the same analogy, if the west input port on (3,2) is simply not active, we still allocate the packets on east and local ports on the same VC, which leads to under utilization of the available resources. However, this ensures that a slow south output port on (2,2) (that may be connected to a slow computing resource, such as a memory controller) does not cause contention upstream, since data from the third input port on (3,2) will always find a VC available that it may use to reach the local port of (2,2), and so on. Another benefit of our scheme is described next. On tile (2,2), now we

know that data arriving on one VC on north input port is always directed towards south, and that arriving on the other VC tends to go to the local output port, the arbiter on the local output port, for example, needs only be made 5-way (supporting two VCs for the west, and one VC for each of the other input ports) instead of supporting eight VCs in total. The type of allocation in which an input VC is always assigned the same output VC is termed as *static*, whereas, the output VC allocation, which requires information embedded within the header flits, is called *dynamic* VC allocation. Fig. 3.7 presents the possible mapping of input VCs on to the output VCs on node (2,2). It is clear that our approach reduces the number of VC connections (31 connections instead of 58 that would be needed if each input VC could access all output VCs) significantly because of static allocation between several IO ports. All the nodes having smaller connectivity than (2,2), will have even smaller number of VC mappings, thereby further reducing the switch complexity.

Note that evaluating our VC allocation scheme, in terms of efficiency against overhead, is not a part of this work. In what follows, we describe a framework that prescribes the minimum requirements of buiding a safe VC access control mechanism to avoid blocking the SPL.

3.4 Classification of the Access Control Schemes

In the virtual channel flow control paradigm, several data transfers share one common SPL, dividing the available bandwidth by multiplexed access through an arbitration scheme. As shown in fig. 2.21, handshaking is performed between any two neighboring blocks along the message path: data producer / sender / funnel / horn / receiver / data consumer. The proper planning of these handshakes is one problem we want to address. From the view of the SPL the procedure works as follows: A rising edge of the request wire at the funnel's interface denotes a request of a VC for exclusive use of the SPL to transmit a flit. This starts arbitration between competing requests from other VCs. Once the SPL has been allocated and the flit was transmitted, the funnel raises the acknowledgement wire. With the falling transition of the request wire, the VC frees the SPL. The interface to the SPL at the receiving end also expects a full handshake cycle for each transmitted flit. Therefore, a flit is only considered delivered when both the request and acknowledgement wires return to low.

3.4.1 VC Controllers

Recall that implementing a correct VC flow control mechanism requires the producer and consumer to be properly decoupled from the SPL (Sec. 2.2.4). We will derive constraints on the controllers that must be fulfilled in order to truly decouple producers and consumers from the SPL while still generating an appropriate sequence of events for guiding the data flow through the latches without loss or corruption of data. For doing this we will use state graphs (SGs) and STGs to describe handshake controllers to be used in the sender and receiver circuits.

SGs with all but the request and acknowledgement signals hidden were shown to elegantly describe handshake controllers and allow their systematic classification [13, 142]. In the following, we will use the same structure for SGs as the authors of [142], depicted in fig. 3.8. State transitions on the horizontal axis correspond to events on the left (L) hand interface (of any

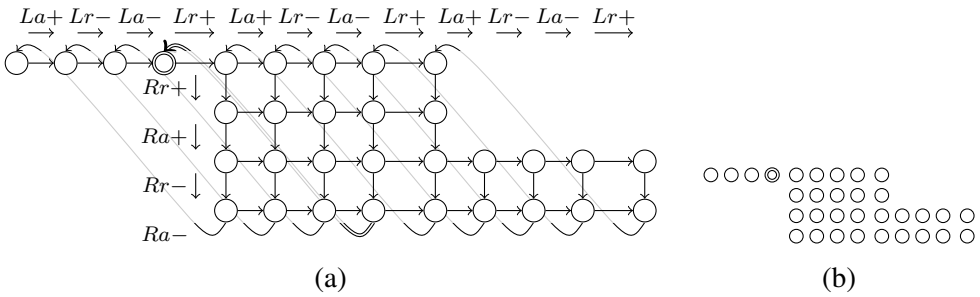


Figure 3.8: The maximally concurrent controller: (a) SG, (b) Simplified SG

block along the message path) while transitions on the vertical axis correspond to events on its right (R) hand interface. Starting from the initial state, at both interfaces the transitions are in the (repeating) sequence $r+$, $a+$, $r-$ and $a-$ corresponding to the 4-phase protocol. Note that the transition $Ra-$ wraps around the SG – the resulting state upon the firing of this transition is in the top row, shifted four columns to the left.

STGs, in comparison to SGs, clearly depict the causal dependencies of signal transitions and are well established for description of asynchronous control circuits. However, it is not so simple to abstract away internal signals (such as the signal controlling the latch in a handshake controller). As a consequence, STGs usually imply implementation details. In the figures, edges in STGs that restrict input signals and thus represent an assumption about the environment, will be drawn with dashed lines.

As a starting point for our discussions, fig. 3.8(a) shows the state graph of the maximally concurrent handshake controller with storage for one data word (one latch), referred to as *max* in [142] and in the rest of this chapter. Fig. 3.8(b) shows the simplified representation of the same with all edges removed, and fig. 3.9 shows the corresponding STG. By *maximally concurrent* we mean that as soon as the left hand interface receives an input request and data are latched, the two interfaces may handshake with their respective environments independent of each other. While adding concurrency to this controller would result in violating the protocol and therefore loss of data/synchronization, there are several ways to reduce concurrency – either by removing reachable states from the SG ("cutting" states) or adding edges in the STG – deriving optimized controllers respecting the 4-phase handshake protocol. To keep the shape of the simplified SG representation constant, states of *max* that are not reachable in a less concurrent handshake controller are replaced by dots (see e.g. fig. 3.10).

The green (shaded) bars in fig. 3.10(a) denote when data are stable at the input (vertical bar) and when data are expected to be stable at the output (horizontal bar). We can now make the following important observation: If a handshake controller only allows crossing the horizontal bar within the vertical bar, this implies that data will not be removed from the input before they have been safely consumed on the output side. Thus there is no need for storing them. The handshake controller depicted in fig. 3.10(a), is thus the most concurrent one which allows a direct connection of the data wires from the left interface to the right without the need for a latch. It corresponds to the STG in fig. 3.10(b), where an edge was added from $L-$ to $La+$ in *max* allowing the latch (and its control signal L) to be optimized away completely. We will call

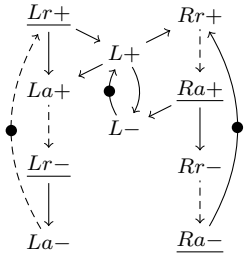


Figure 3.9: STG of the maximally concurrent controller

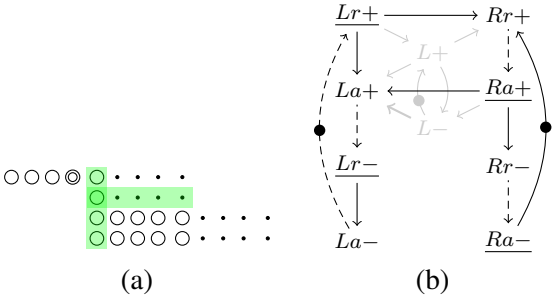


Figure 3.10: Maximally concurrent bufferless controller: (a) SG, (b) STG

subsets of these protocols *bufferless*.

As a further observation, the handshake controller that has all concurrency removed can be implemented with a wired connection of the request and acknowledgement wires respectively.

3.4.2 Decoupled producer

To prevent a producer from stalling the SPL by not lowering its request, we need a handshake controller in the sender, that allows arbitrarily slow producers, yet completes the handshake with the funnel immediately after the latter raises the acknowledgement. In the STG of a sender fulfilling this requirement, the handshake on the right interface must be allowed to finish without any dependencies on transitions on the left interface after it has started with $Rr+$; $Rr-$ shall not have an incoming edge other than the one from $Ra+$. We can see, that *max* has this behavior. In the SG, the right interface (vertical) must be allowed to complete once the request is generated, without any transitions on the left interface (horizontal). Note that this also allows bufferless controllers to be used. They may, however, only remove additional states from the right of the maximally concurrent bufferless controller – a wired connection of all signals is not sufficient.

3.4.3 Decoupled consumer

To protect the SPL from slow consumers, we require that once the horn issues a request with a new flit, the handshake must complete without any actions from the consumer. The receiver will thus have a similar restriction on its STG as the sender – transitions on the left interface

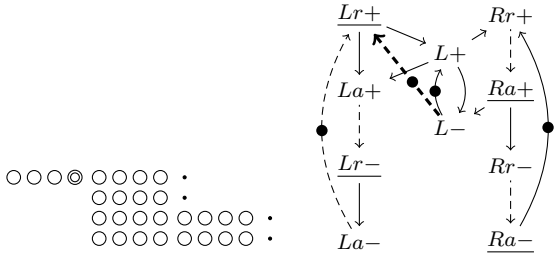


Figure 3.11: Decoupling of the consumer: (a) SG, (b) STG

must not depend on any transitions on the right interface. This is, however, somewhat more intricate to fulfill, since removing dependencies of $La+$ on the state of the latch or consumer would inevitably lead to loss of data, since the acknowledgement of flits would no longer depend on them being properly stored or consumed.

An examination of the STG of *max* reveals that $La+$ is dependent on $L+$, which in turn is dependent on the consumer through $L-$; unless a stored flit is consumed, the latch cannot store a new one. Since the most possible amount of concurrency for a handshake controller does not provide decoupling of the consumer, it is clear that concurrency must be restricted by adding a new edge to the STG. The only possibility to restrict the left interface without losing the behavior required for decoupling is to restrict when the $Lr+$ transition may occur – only after the flit in the latch has been consumed. Fig. 3.11(b) depicts the resulting STG.

In the SG of *max*, a similar observation can be made. Once a request on the left interface is received, the handshake must be allowed to complete without any action on the right interface. This means that for each reachable state, all states to the right until the boundary to a new handshake cycle (just before the $Lr+$) must be reachable. Fig. 3.11(a) depicts the SG of the most concurrent controller for the decoupling of the consumer. It is also corresponding to the STG in fig. 3.11(b).

In both the STG and SG it can be observed that, since $Lr+$ is an input, an assumption about the environment was necessary to achieve decoupling of the consumer. In [142], such controllers are called *timed*. However, this assumption cannot be implemented by modifying system timing. If it were a timing assumption, it would not allow backpressure, i.e. a slower consumer than the producer. This would however somehow defeat the purpose of VC schemes. Since we do allow the consumer to stall for arbitrarily long times and we do not want to restrict the transmission speed of the producer other than by back-pressure, the assumption when a new request (flit) may arrive at the receiver must be enforced by a circuit. This is the task of what is generally known as *flow control*.

Furthermore, it can be observed that bufferless handshake controllers cannot be used for decoupling the consumer. It is thus apparent, that the main purpose of latches in VC receivers is not to serve as a FIFO storage for incoming flits, even though they obviously do. Their primary purpose, however, is to allow decoupling. When there is a requirement for more storage in the receiver, it should be placed between the receiver and the consumer and not become part of the receiver, where it increases the overhead of flow control schemes.

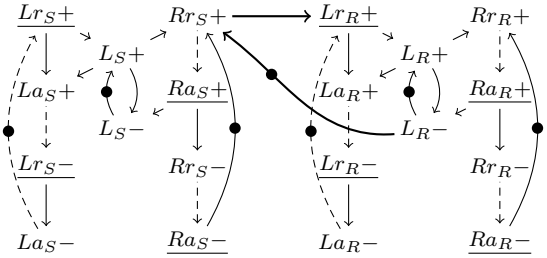


Figure 3.12: STG showing dependencies in a VC

3.5 Flow control schemes

In this section we will analyze two most widely adopted SI flow control schemes with respect to the requirements prescribed above. In order to change the assumption in the receiver to a real dependency enforced by the implementation, first we must consider both the sender and the receiver connected over the SPL. The sender's right interface performs its handshake with the funnel and the receiver's left interface with the horn. Depending on the SPL implementation, the handshakes can be independent of each other. However, there is one real causal dependency between the two – the Lr input at the receiver only goes high when Rr at the sender went high. It is even more obvious when stated more informally – a flit can only be received once it has been sent. Considering the STGs of the sender and receiver, we can thus add an edge from Rr_S+ to Lr_R+, the indexes S and R used to distinguish between signals at the sender and receiver, respectively. To remove the assumption made about the Lr_R+ input, we can now move the required edge $L_R- \rightarrow Lr_R+$ to the causally preceding Rr_S+. Fig. 3.12 depicts the resulting STG with both the sender and receiver being maximally concurrent as discussed in the previous section.

It should be noted that flow control establishes a communication path between the sender and the receiver of each VC in a link. If a shared media is also used for these backwards paths like in [15], the same requirements hold as for the data path in the forward direction. In particular, proper decoupling is required.

Out of the two flow control mechanisms, conceptually the *share-unshare* scheme introduced in [15] is much simpler. It directly corresponds to the STG in fig. 3.12. The idea is to let one *flow control token* be exchanged between the sender and the receiver (on the $Rr_S+ \rightarrow Lr_R+$ and $L_R- \rightarrow Rr_S+$ edges) corresponding to a flit being sent and an event on a flow control wire. Since both edges are safe – the reception of the event is "acknowledged" by the next request, if the event is implemented with a single transition (2-phase handshake) – this scheme really requires only a single additional wire for flow control, even for delay insensitive links.

Having formalized the requirements on circuits for true decoupling in the previous section, we are now able to verify the chosen implementation for the share-unshare scheme in [15]. Both STG and SG of the handshake controller implemented in the *sharebox* (sender) are depicted in fig. 3.13. Flow control is implemented by delaying $Lr+$ of the handshake controller until the flow control token from the unsharebox is available. This is depicted with the unconnected green edge in the STG and the green arrow in the SG. As can be seen, the requirement for proper

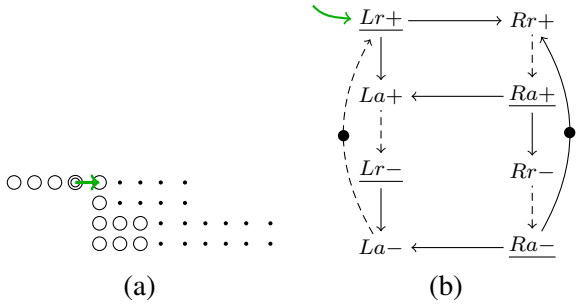


Figure 3.13: The sharebox from [15]: (a) SG, (b) STG

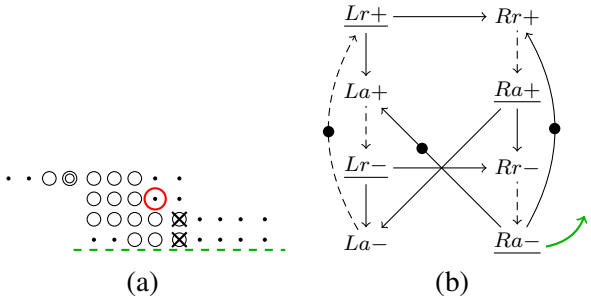


Figure 3.14: The unsharebox from [15]: (a) SG, (b) STG

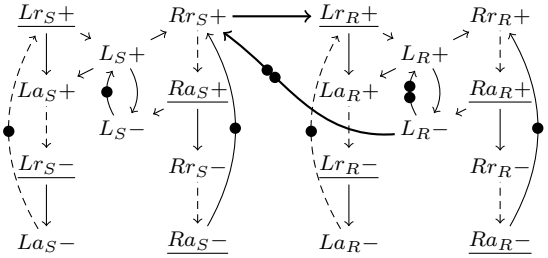


Figure 3.15: STG of a credit-uncredit scheme with two credits

decoupling of the producer from the funnel is met.

The descriptions of the *unsharebox* (receiver) are depicted in fig. 3.14. Here, flow control is implemented by issuing a transition on the unlock wire with the falling edge of Ra, again depicted with the unconnected edge in the STG and the green dashed line in the SG marking the transition which causes the flow control token to be returned. The crossed states represent states allowed by the handshake controller implementation, however not reachable due to flow control. It can now be seen that in both representations the requirement is not met – in the STG $La-$ is dependent on $Ra+$, in the SG, a state is missing in the second row on the right (circled). This means that this implementation, connected to an SPL without decoupling on its own, would allow a single consumer to block the whole SPL.

The next scheme to be discussed is the *credit-uncredit* scheme, also proposed in [15]. Here

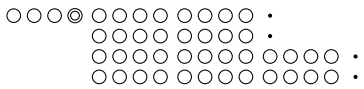


Figure 3.16: SG of the receiver for a credit-uncredit scheme with two credits

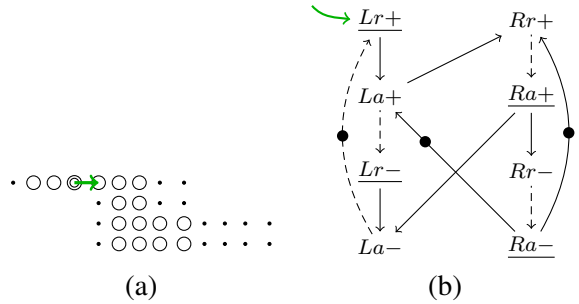


Figure 3.17: The creditbox from [15]: (a) SG, (b) STG

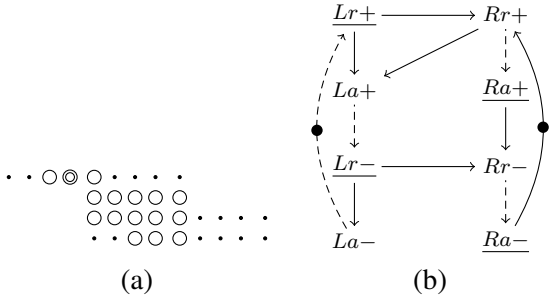


Figure 3.18: The uncreditbox from [15]: (a) SG, (b) STG

the idea is to allow more tokens in the sender-receiver cycle. Considering the STG in fig. 3.12, this would mean initializing the edge $L_R- \rightarrow Rr_S+$ with more than one token. (As an important side note, to keep the discussion general, with as little implementation details as possible, we sacrifice the safeness of the STG allowing places with higher capacity.) This is used to allow the sender to place multiple flits on the link without any flow control event from the receiver, thereby preventing underutilization of a pipelined SPL when only one VC is transmitting. Since the tokens control how much a sender can send, they are known as *credits*. When adding tokens to the flow control loop, the same number of tokens must also be added to the storage loop in the receiver (the $L_R- \rightarrow L_R+$ edge) for decoupling to work. This effectively means that there must be as many latches provided for incoming flits in the receiver, as there are credits in the VC. Fig. 3.15 depicts an STG for a VC with two credits, fig. 3.16 depicts the SG of a receiver for a two credit VC scheme. Also, since the flow control edges in the STG are no longer safe, either timing assumptions are required or a bidirectional handshake must be implemented for delay insensitivity.

We now proceed with the analysis of the chosen implementation for the credit-uncredit

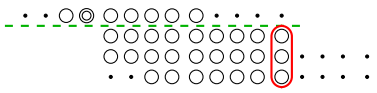


Figure 3.19: SG of the uncreditbox from [15]

scheme in [15]. Again, describing only the handshake controller and sketching the effect of the flow control circuit, fig. 3.17 depicts the STG and the SG for the creditbox (sender). Again, it is properly implemented, satisfying the requirements. Since the presented implementation has two credits in the VC, the uncreditbox (receiver) also requires two latches. For each, the handshake controller described in fig. 3.18 was chosen. Fig. 3.19 thus depicts the SG of the uncreditbox. Here, the problem is that credits are returned to the sender with the rising transition of Rr, making the three states not allowed by the requirement for non-blocking not only allowed by the handshake controller, but also reachable. As a result, this scheme also allows blocking of the SPL by a single consumer if the horn is not decoupled itself[2].

To summarize, our framework has allowed us the following key conclusions:

1. For a *bufferless implementation*, in the STG of the handshake controller, the rising edge of the left acknowledgement must depend on that of the right acknowledgement. See fig. 3.10.

2. For *decoupling of the producer*, the handshake on the right interface must be allowed to finish without any dependencies on the left interface. In the STG $Rr-$ must not have an incoming edge other than the one from $Ra+$. See Sec. 3.4.2.

3. *Decoupling of the consumer* can only be achieved with a matched choice of handshake controllers and flow control circuits. In the STG, transitions of the left interface may only depend on themselves and on $L+$. Flow control must ensure $Lr+$ does not fire before $L-$. See Sec. 3.4.3.

4. Requirements for decoupling of the consumer and for bufferless handshake controllers are contradictory. There is thus a direct requirement for *storage in the receiver for decoupling*, more precisely for as many buffers as there are credits. Eventual additional buffers are not required and should be placed outside of the receiver. See Sec. 3.4.3.

3.6 Proposed Flow Control Scheme

In this section we build upon the *credit-uncredit* scheme from [15] which we refer to as "baseline" in the following, and while keeping its main concept unchanged, we modify the way credits are returned to the sender – instead of transmitting each credit separately, we propose to send them in *bundles*, i.e. M credits at a time. This is possible, since credits are only synchronization

[2]It should be mentioned that the authors of [15] are aware of this deficiency but did not further address it in that publication

events (no data is transferred). The obvious choice to transmit them is a synchronization handshake on a channel for each credit, however, the semantics of this dataless handshake can also be chosen to mean M credits. For example, with $M = 2$ each transmission on the upstream link corresponds to a pair of credits being returned.

This concept will be referred to as *Multi-Credit Flow Control* (MCFC) in the rest of the thesis. Note that a choice of $M = 1$ forms the baseline scheme, therefore we can consider the latter a special case of MCFC. Moreover, we will use N to denote the total number of credits in a virtual channel. Here, a choice of $N = 1$ leads to the *share-unshare* scheme from [15] where a new flit can only be sent if the consumer has read the previous one. Since MCFC is a modification of the baseline credit scheme, we will first show how they compare.

Fig. 3.20 shows a typical (high level view of) timing of a virtual channel with baseline flow control. In this figure, we assume communication on only one VC of the SPL (no arbitration overhead), between an eager producer-consumer pair. We abstract the communication cycle into the following four phases and respective time intervals:

- *Crossing the downstream SPL* (δ_{DL}). This resembles the propagation delay of the link and overheads of the funnel, horn and pipelining.
- *Consumption of the flit* (δ_R). This subsumes crossing the FIFO pipeline in the receiver, consumption of the data by the consumer and the generation of the credit.
- *Crossing the upstream link* (δ_{UL}). This includes the propagation delay of the upstream link and overheads of pipelining and of the funnel and horn if the link is shared.
- *Credit handling in the sender* (δ_S). This represents the time from receiving the credit from the upstream link until a new pending flit can be sent, including the overheads of the credit FIFO.

Additionally, T_{DL} denotes the minimal transmission cycle time for the downstream link, thus the reciprocal value of the maximal bandwidth. The circled values denote which credit (if they were numbered) has been consumed for a transmission or has become available, respectively. As can be seen in the figure, we assume both pipelined downstream and upstream links (a new flit/credit is sent before the previous one reaches its destination) and the chosen system timing is such that four credits are required to sustain the maximum downstream transfer rate (the first credit has returned and is available just before the fifth transmission). Note that there is a slack time t_{SL} between the reception of a credit and its use. Its value is a subject of optimization when designing links employing a credit based flow control. When it is negative, the downstream link operates below its performance limits. A too high t_{SL} indicates that the upstream link can be further optimized for low power, or a credit be removed from the loop.

Fig. 3.21 shows how the system's behavior changes when the MCFC scheme with $M = 2$ is used, while the timing parameters remain unchanged. The most prominent observation is the lower bandwidth on the upstream link (fewer credit arrivals depicted with vertical lines) which reduces to $1/M$ of the baseline scheme. This allows the removal of pipeline stages from the upstream link; in this example the link would no longer require pipelining since, as can be seen in the figure, a new credit (pair) is only sent when the previous has already arrived at the sender.

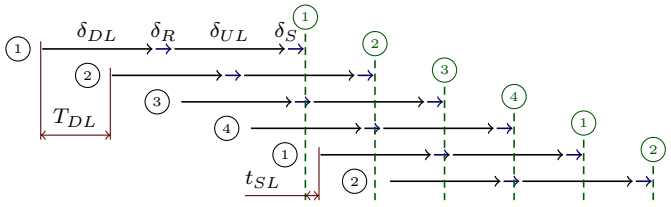


Figure 3.20: VC timing using the baseline credit scheme

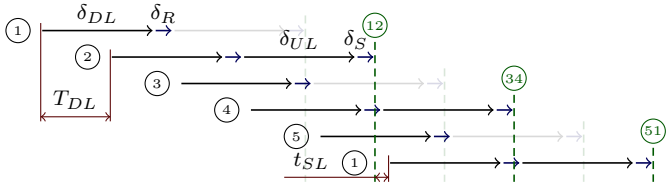


Figure 3.21: VC timing using the MCFC scheme

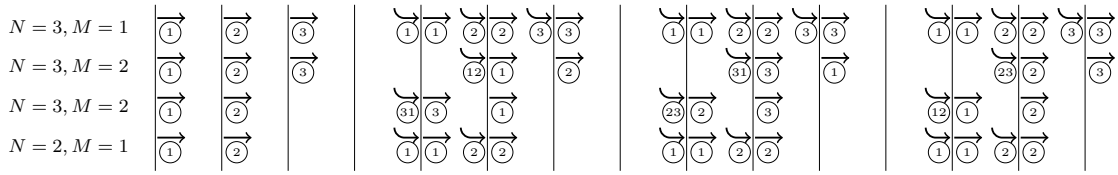


Figure 3.22: Comparison of flow control mechanisms

Another observation is the unchanged slack: While the upstream bandwidth is reduced with our scheme, the latency requirement (δ_{UL}) is not relaxed. In other words, the upstream link will transfer fewer tokens but it must do so with the same speed as compared to the baseline scheme. A final observation, that unveils the overhead of the MCFC scheme, is that the VC now requires five credits to reach maximum performance. This directly follows from the fact, that every other credit is returned one flit (T_{DL}) later compared to the baseline credit scheme, thus an extra credit must be added to the loop to allow transmission during the time where this credit is missing. Inevitably, each credit added to the loop also requires one more buffer in the receiver. This does not necessarily result in an overhead – often buffers are already there to even out speed variations in producer and consumer, and we can take advantage of them for MCFC.

In general, the MCFC scheme is equivalent to a baseline credit scheme where the link's timing is the same and $M - 1$ credits are added to each VC. To illustrate this, fig. 3.22 shows the flow of tokens in a link with four configurations of MCFC. For simplification, the slack is assumed to be exactly zero (allowing us to use 'cycles' with period T_{DL}), and the four times listed above are assumed to sum up to $4T_{DL}$ and are reduced to two events – flits being sent and credits becoming available. The values again denote the credit numbers.

The first row shows the baseline credit scheme ($M = 1$) with three credits and the last row shows the same with two credits. The two rows in the middle show the MCFC scheme with three credits in total and transmission in pairs ($N = 3$, $M = 2$) in the two possible stable states, which can also be chosen for the initial state. In one case all credits are in the credit FIFO of the sender (second row), in the other case one credit has been consumed already but not returned

since the receiver is waiting for the pair to complete (third row). As can be seen, after the initial burst, a scheme with $M = 2$ really does perform equivalently to the baseline credit scheme with two credits validating the claim in the above paragraph.

Now that the N and M constants have been introduced, an interesting property can be observed: When $M > N/2$ the receiver will never produce two successive events on the upstream link without having received at least one flit in between. The correctness of this claim can be seen by considering the worst case – while a consumer was halted, the receiver's FIFO has been filled with flits (the sender has consumed all tokens). Now the producer is halted and the consumer starts a burst read emptying the FIFO in minimal time. The generation of two successive events requires the consumption of $2M$ flits, so because the FIFO can only store N flits, it will not happen.

There are also other stable states to be considered. If the receiver was initialized holding an incomplete bundle of $k < M$ credits, the first event will be produced after reading $M - k$ flits, the second after $2M - k$. While this number can be smaller than N, it should be noted that the k credits that are being held by the receiver initially are also missing in the sender's credit FIFO. Therefore, it cannot have sent more than $N - k$ flits, which again is smaller than $2M - k$. As a conclusion, there can never be enough flits on the fly (in the SPL) and in the receiver's FIFO such that their consumption alone would generate two credit bundles.

This property can be used to design systems with a unidirectional upstream link (no acknowledgment wire) while preserving delay insensitivity of the system. When M and N are chosen according to the requirement above, it is certain that each event on the upstream link will be followed by at least one event on the downstream link (flit) which causally depends on the former being correctly processed. Therefore the reception of a flit is semantically equivalent to an acknowledgment signal rendering the explicit acknowledgment wire in the upstream link unnecessary. Of course, this only holds if the link is not shared. As an example, consider the two flow control schemes presented in [15] with respect to this property. The *credit-uncredit* scheme ($N > 1$ and $M = 1$) requires a full delay insensitive upstream link while the *share-unshare* scheme ($N = 1$, $M = 1$) fulfills the requirement above and therefore can be – and is – implemented without an acknowledgment in the upstream link.

3.7 Proposed Implementation

From the design space sketched in Sec. 3.6 we will now pick one specific instance, namely $N = 3$ and $M = 2$ to give an implementation example. This choice allows us to use a unidirectional upstream link and represents a good tradeoff between link performance and area overhead. In Fig. 2.21, we have already seen the fundamental elements making up a flow control mechanism. While in this work the funnel and horn modules are made identically to those proposed in [15, 42], here we will only present the design and implementation of the sender and the receiver modules.

3.7.1 Sender

Fig. 3.23 presents the proposed sender for our MCFC. Its operation may be divided into four steps: (1) generating a short pulse for every transition on the credit link (2-phase/4-phase con-

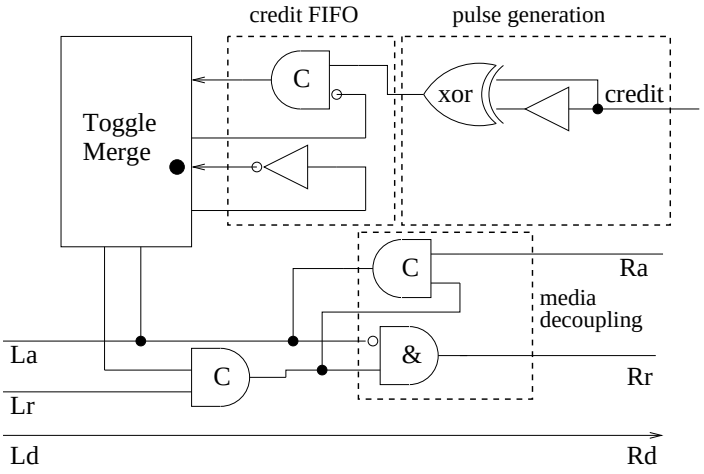


Figure 3.23: Proposed sender

version), (2) generating multiple credits from one received token on the upstream channel, (3) joining a credit with the producer's request to transmit a new flit on to the SPL, and (4) decoupling of the SPL from a (possibly) slow producer. While the steps 1, 3, and 4 are identical to those described for the share-unshare scheme proposed in [15], (2) is something that distinguishes the proposed sender from the existing baseline methodology.

The *toggle merge* module (already introduced in the background chapter) merges the *M* incoming requests on to its output in an alternating manner ($M = 2$ in our case study), i.e. no particular input channel may transmit two credits in succession, no matter how many more it possesses. Fig. 3.23 depicts the initial state of the module, where the black dot points to the bearer of the token: an inverter generating requests from acknowledgments always possesses a credit for the producer to transmit a new flit on the SPL. The other input receives a credit via an MC, which is also initialized to 1 (i.e., holding an initial credit), thereby giving the producer a permission to submit *N* (3 in our case) flits in quick succession, without waiting for a credit to arrive. This also indicates that the receiver has to provide *N* available places to store those new flits.

Obviously after three transmissions, the input channel with the MC possesses the token, and is forced to wait until a new credit has been received. From this point onwards, with reception of every credit, the toggle merge module allows the producer to submit *M* (2 in this case) new flits on to the SPL.

3.7.2 Receiver

The receiver comprises two major units: data FIFO, and credit generation unit. The FIFO is identical to the one used in the uncredit box from [15] except for that it contains one more storage element (now *N*). Besides storing, the FIFO naturally provides decoupling of the SPL from a possibly slow consumer. The design of the FIFO is presented in fig. 3.24.

The design of the credit generation unit is illustrated in fig. 3.25. It performs two major tasks: (1) to provide decoupling from a potentially slow consumer, a credit must only be generated once the handshake is completed between the FIFO and the consumer, i.e., with $Ra\_receiver-$, and

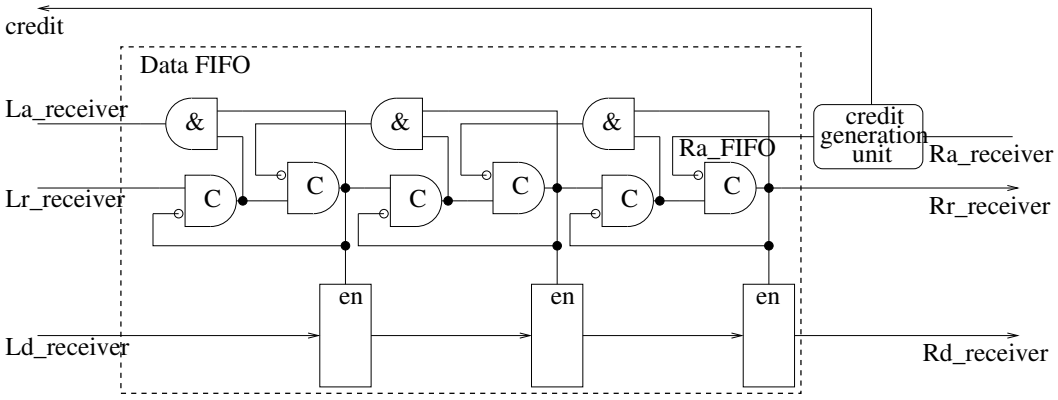


Figure 3.24: Proposed receiver

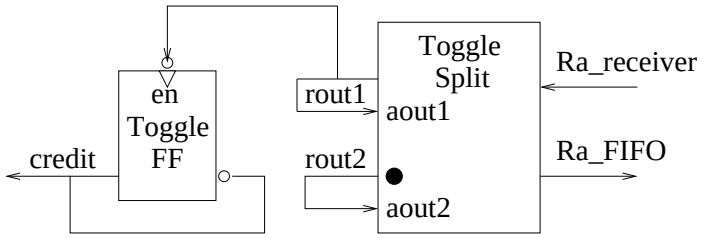


Figure 3.25: Proposed Credit Generation Unit

(2) it must send a credit on the upstream channel only after consumption of two data flits by the consumer. The latter is achieved with a *Toggle Split* module (also introduced in the background chapter). The first handshake cycle is completed (in other words wasted) between the input and the lower output, since the latter possesses the token initially, as shown in fig. 3.25. This cycle represents the consumption of the first data flit by the consumer. At this point, the token is handed over to the other output channel, which then waits for the second handshake cycle, i.e., consumption of the second data flit. Notice that the T-FF is enabled with the falling transition of $rout1$, which corresponds to the falling $Ra\_receiver$. The output of the T-FF is the required credit bundle, which indicates to the sender the availability of two buffer places. We chose to use a transition signaling protocol here to further save dynamic power.

3.7.3 Timing Assumptions

In our general context of delay insensitive designs it is important to point out the timing requirements that are still essential for the correct operation. Although all of the requirements that we mention in the following may be completely eliminated, we opt for trading off delay insensitivity for area, dynamic power dissipation, and simplicity, since the associated timing requirements can be satisfied quite conveniently.

Pulse Generation Circuit

The delay element in this circuit must ensure that the pulse generated is long enough to properly drive the MC in the credit FIFO.

Toggle Flip-Flop in the Credit Generation Unit

Since the T-FF is not within the timing closure formed by the split module (a_out1 does not wait for $credit$ to alter), it is essential to make sure that the credit link has settled down to its new state well before the next event on $Ra\_receiver$. Meeting this requirement is definitely easy due to two reasons: a) Ra_FIFO has to drive an MC, which then requires the consumer to complete the handshake cycle. In practice, this procedure already takes sufficiently longer than the T-FF propagation delay. b) $rout1$ will not receive the next event until $Ra\_receiver$ has completed a handshake cycle with the second output channel. This adds a further safety margin to the timing of the circuit.

In addition, when implementing the toggle split and toggle merge blocks, we encountered a few further timing assumptions. Those, however, were very similar to the one discussed above for the T-FF.

3.8 Evaluation

3.8.1 Simulation Results

In this section we present the pre-layout Modelsim simulations of the proposed MCFC mechanism, more specifically its implementation with $M = 2$, as described above. The simulated network comprises two pairs of nodes, which communicate over two VCs. In the first simulation, fig. 3.26, we have presented the case of eager producer-consumer pairs on both VCs. For a more meaningful analysis, we connected both the producers with 32-bit counters, the first of which counts the even numbers, and the other the odd numbers.

The signals must be interpreted as follows: $Lr\_sender1$, for example, is the input request to the sender of the first VC, which is the output of the first counter that counts the even numbers, starting from zero. $Ra\_receiver1$ is the output acknowledgment for the receiver of the first VC. Note that we have a buffer that simply feeds back $Rr\_receiver1$ to $Ra\_receiver1$ mimicking an eager consumer. In this symmetric setting of two eager VCs, each of them gets hold of the SPL in turn. This case corresponds to the maximum bandwidth utilization, as may be conveniently observed. Finally, observe the propagation of credits $credit1\_r2s$ and $credit2\_r2s$ corresponding to VC1 and VC2 respectively. Both make alternating transitions following consumption of two data tokens.

Fig. 3.27 presents two interesting scenarios where one of the VCs is non-eager, i.e. slow, either due to a slow producer, or a slow consumer. The *solid* arrows between $Rr\_receiver2$ and $Ra\_receiver2$ indicate a slow response from the consumer associated with the second VC. On the other hand, the *dashed* arrows between $La\_sender2$ and $Lr\_sender2$ highlight the slow response of the producer associated with the second VC. The purpose of presenting these scenarios is to demonstrate how efficiently an eager VC (VC1 in this case) is able to utilize the

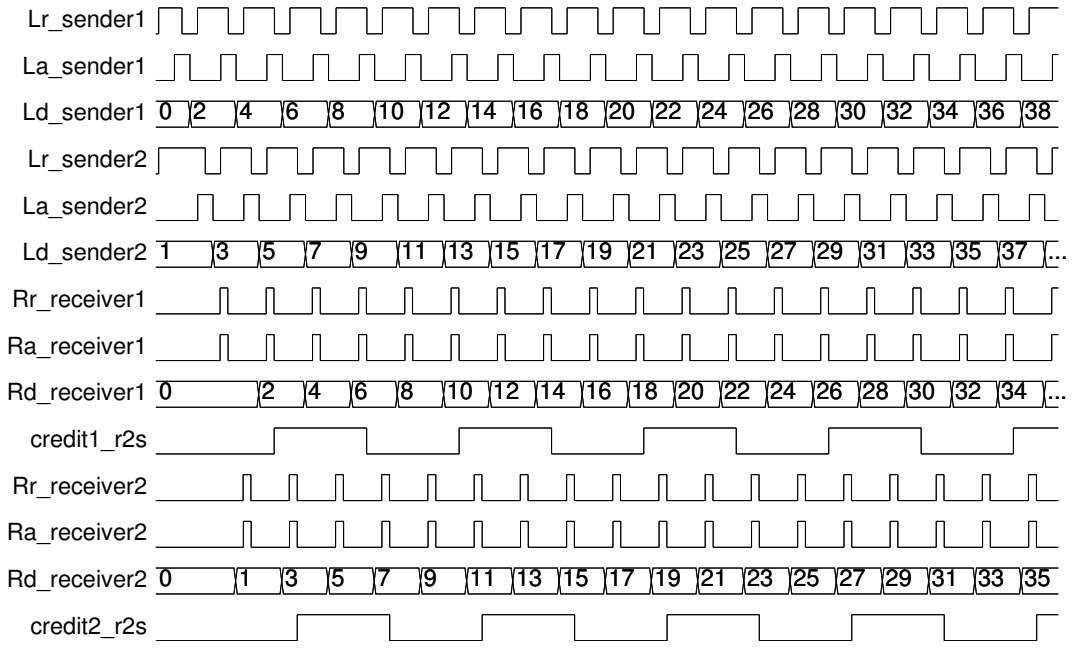


Figure 3.26: Operation of MCFC in an eager producer-consumer environment

SPL, no matter how slow (or completely blocked due to traffic congestion) the other VC might be.

3.8.2 Analysis and Comparison

The most appropriate candidates to compare our MCFC mechanism with are the implementations from [15], that are, the share-unshare and credit-uncredit schemes. Our comparison is based on three metrics: *Number of transitions on the credit link per flit transfer*, *area utilization*, and *throughput*. For a fair comparison we implemented both these circuits along with our own methodology and synthesized all these designs for the same $90nm$ standard cell library.

Number of transitions on the credit link

The credit-uncredit approach, having two wires in the upstream channel, requires four transitions per flit transfer to complete its handshake cycle. The share-unshare scheme, on the other hand, requires just one transition per flit. The proposed MCFC method merely requires half a transition per flit on average, i.e., one transition per two flit transfers. Considering the case of v VCs with all eager producer-consumer pairs, the credit-uncredit scheme would require $4vf$ transitions in total for f flit transfers per VC. For the share-unshare, and the MCFC schemes it should be vf and $vf/2$ transitions respectively.

In an NoC the downstream and upstream channels represent the global, and hence the longest interconnects. It is known that the dynamic power consumed by those long interconnects tends to dominate dynamic power consumption, therefore reducing the number of transitions by a

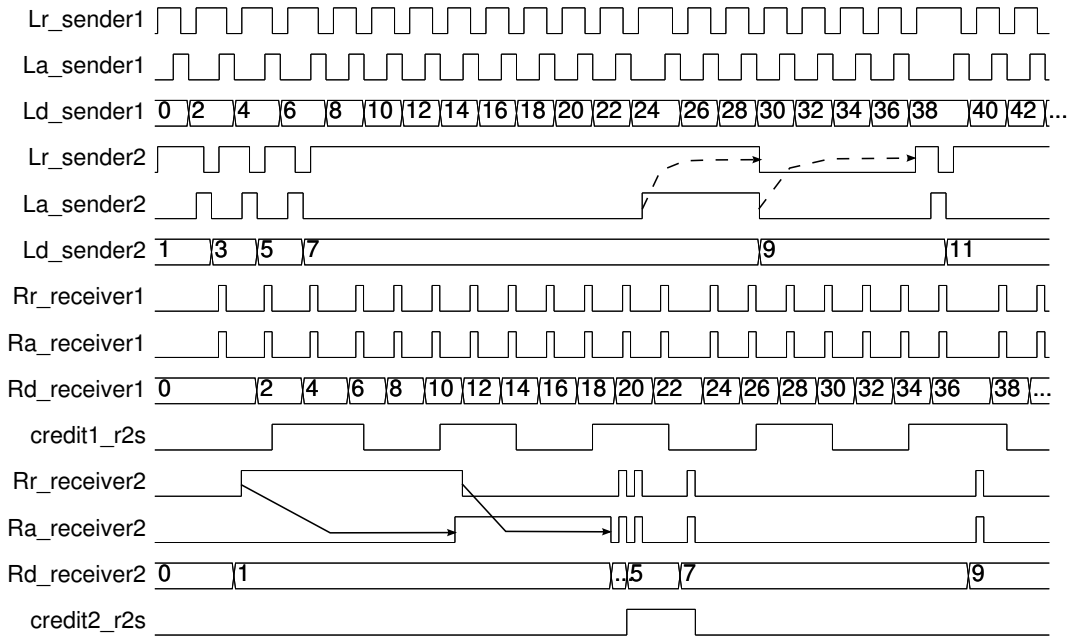


Figure 3.27: Operation of MCFC in a mixed environment

factor of 8 maps to an equally large saving of dynamic power on the upstream channel. The downstream channel remains unchanged for all solutions.

Area utilization

Table 3.3 presents the area utilization of the sender and receiver modules for each of the methodologies we implemented. The share-unshare scheme is of course the most area efficient, requiring the lowest number of storage elements. The creditbox requiring a storage element consumes more area as compared to our proposed sender, which is, however, balanced on the receiver side where we have employed an additional one (recall, however, that we are pessimistic here by not assuming a buffer is available in the receiver anyway). Overall we observe only a marginal difference in the area utilization of the two latter mechanisms.

Table 3.3: Area Utilization (μm^2) of the sender and receiver modules

No	Flow Control Scheme	Sender	Receiver	Overall
1	Share-Unshare	85.5	490.8	576.3
2	Credit-Uncredit	635	1096	1731
3	Proposed (MCFC)	135.6	1675.4	1811

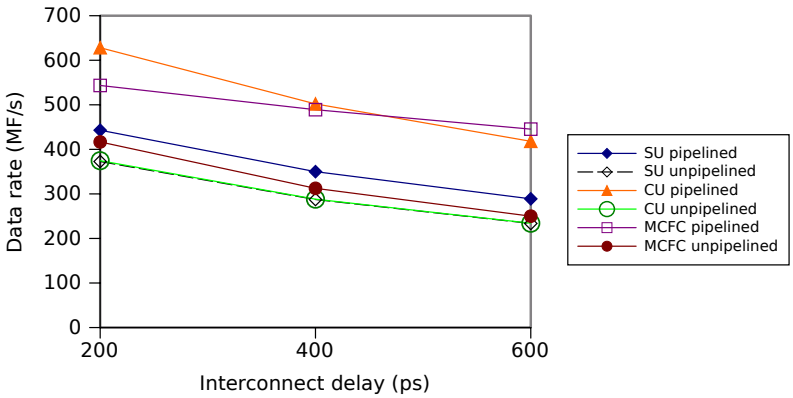


Figure 3.28: Comparison of throughputs of the three schemes

Throughput

In order to compare the throughput of the candidate schemes, we synthesized and simulated a sender-receiver pair connected over interconnects with delays ranging from 200 to 600 ps, for both up- and downstream links. Since pipelined interconnects can significantly affect the throughput, we also decided to compute values using a single stage pipelined interconnect, where the latter was carefully installed in the middle of the SPL. Fig. 3.28 presents the corresponding data rates in million flits per second (MF/s). As can be seen, all the three compared schemes perform equally well with direct interconnects. When pipelining was employed, not suprisingly, the credit based shemes (CU, MCFC) performed much better than the share-unshare scheme (SU). It should be pointed out that CU generates a credit with $Rr\_receiver+$ (see fig. 3.24) in comparison to MCFC's receiver which does the same later with $Ra\_receiver-$, which increases the cycle time. However we have verified in our simulations that a credit forwarded to the sender with $Rr\_receiver+$ can lead to indefinitely blocking the SPL in case the consumer delays $Ra\_receiver+(-)$, making our approach an essential requirement [89].

3.9 Summary

The design of a fully asynchronous router was presented. Our VC allocation methodology allowed us to minmize the number of VCs per IO port, yet satisfying the minimum requirements of avoiding any slowly progressing packet from blocking the SPL. We also proposed a framework to systematically treat some fundamental properties of the handshake controllers for ANoCs, which allowed us to pinpoint a deficiency in one of the most widely adopted VC access control methodologies. Our novel flow control mechanism satisfied all the requirements set within our framework, and minimized the bandwidth requirements by transmitting credits in bundles. The reduction in the number of transitions needed to transmit a given number of credits reduced the dynamic power dissipation on the SPL by a factor of eight, and the area overhead incurred as compared to the state-of-the-art approaches was found to be negligible when synthesized with the UMC 90nm library.

CHAPTER 4

High Speed Resource Sharing

In this chapter we begin by explaining the operation and design of an arbiter, and go on to present a novel tree arbiter cell that allows a pipelined processing of requests. In this way it can achieve significantly lower delay in the critical case of frequent requests coming from different clients. We elaborate the necessary extension to facilitate a cascaded use of this cell in a tree-like fashion, and we show by theoretical analysis that in this configuration our cell provides better fairness than the standard approach. We implement our approach and quantitatively compare its performance properties with related work in a gate-level simulation. Our new cell proves to increase the throughput of three different designs available in literature by approximately 61.28%, 69.24%, and 186.85% respectively.

4.1 Background and Related Work

The purpose of an arbiter is to control the access of several clients to a shared resource. As shown in fig. 4.2 for the case of 2 clients, it has one pair of req/grant signals per client, and one additional req/grant signal pair for the shared resource.

The arbiter's task becomes difficult as soon as two or more clients concurrently request the resource; here the common rule is that the first one will win. However, if the contending requests arrive at (nearly) the same time, this decision becomes particularly problematic, and it is known to involve the risk of metastability then. A dedicated mutual exclusion (MUTEX) circuit is therefore employed for this decision: It has one pair of req/grant inputs per client, and its only duty is to assert the grant associated with the request that arrived earlier. In case of concurrent (very close) requests the MUTEX is free to make a choice; any decision will work, as long as only one grant is asserted at a time. Fig. 4.1(a) illustrates the operation of a MUTEX using a 4-phase protocol: The activation of the earliest request is acknowledged by a grant that remains activated until the request is withdrawn. At that point the second request, if activated, will get serviced in the same way.

Although in principle a multi-way MUTEX can be built [63], the simpler two-way MUTEX is often preferred, and the remaining arbitration logic is arranged in several levels, so as to extend

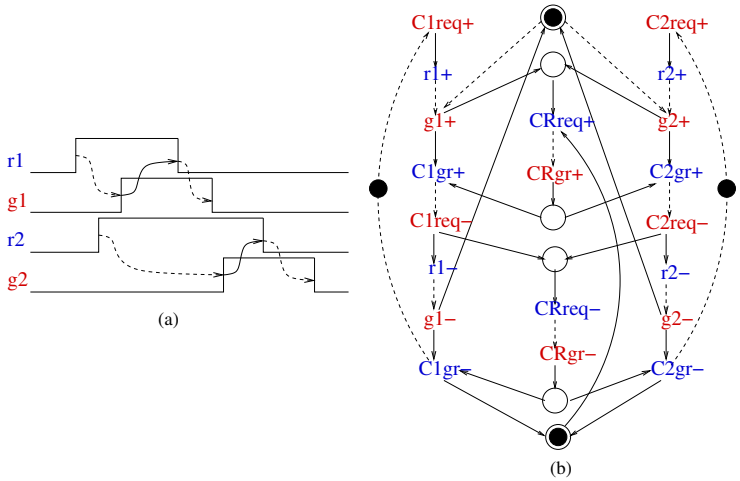


Figure 4.1: STG of a 4-phase Two Input TAC

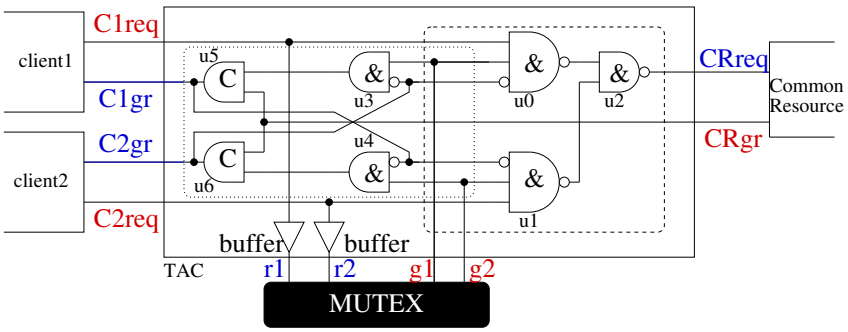


Figure 4.2: Gate level netlist of a 4-phase Two Input TAC

a two-way (n-way) arbiter to a four-way (2n-way) arbiter. The circuit used for this purpose is the *tree arbiter cell* (TAC), an implementation example of which is shown in fig. 4.2. This low latency 4-phase two input TAC has been proposed by Yakovlev et al. [149], and we will use it throughout the rest of the chapter to illustrate our approach. Its key function is to properly convert the req/grant signals from clients, shared resource and MUTEX. The corresponding signal transition graph (STG) is presented in fig. 4.1(b). While the letters *C* and *CR* in both the figures stand for *client* and *common resource* respectively, *req* and *gr* stand for *request* and *grant*. Finally *r* and *g* refer to the in(out)-puts to(from) the MUTEX respectively.

A close analysis of the TAC's STG reveals a possible window of improvement (in terms of its latency): On the one hand, the *down* transitions on clients' requests ($C1req-$, $C2req-$) release the MUTEX ($r1-$, $r2-$) and the common resource ($CRreq-$) simultaneously. On the other hand, however, the requests to lock the arbiter ($C1req+$, $C2req+$) first acquire the MUTEX ($r1+$, $r2+$), followed by reserving the common resource ($CRreq+$) in series. Ghiribaldi et. at [50] recently proposed an efficient 4-way arbiter that exploits the same window to achieve

higher performance in their arbitration scheme. A MUTEX forms the common resource which is reserved and released in parallel with the local MUTEXs. This roughly saves delay of a MUTEX and that of a standard gate.

Some *fast* fixed priority programmable arbiters already exist in literature that work perfectly in synchronous environment (routers) [35], [57]. All those arbiters, in addition to the arbitration circuitry, require scheduling logic to dynamically generate new priority vectors every clock cycle. Although such priority vectors may be generated for asynchronous environments as well, the arbitrary arrival time of the input requests in that case makes the choice of the winner tedious, and usage of the MUTEX, in any case, becomes mandatory. Furthermore, the scheduling logic itself incurs area, power, and performance penalties. All these observations make such arbiters irrelevant to asynchronous design styles.

Felicijan proposed a low latency static priority asynchronous arbiter in [42]. The design did not need any explicit scheduling logic, since it comprised a linear priority module, which allowed one of the n clients, c_k, to block the rest $(k-1)$ having lower priority. The drawback associated with this approach is the number of MUTEXs that scales linearly with clients, and the number of Muller C-elements also scales badly.

In this chapter we propose an arbitration logic for fair and nondeterministic decision (where all clients have the same priority). Like [42] we assume that the arbiter is not the slowest component of the design, and that the clients, once received the grant, would keep the arbiter reserved at least for the duration longer than the latency of the arbiter. These assumptions are not so optimistic, especially for switch controllers in NoCs, where the header flit reserves the shared resources, and the tail flit releases them. Although our methodology is equally applicable to Ghiribaldi's approach [50], we only emphasize on the TAC because of its simple architecture that allows systematic modifications that are relatively easy to illustrate.

4.2 Proposed Tree Arbiter Cell

4.2.1 Window of Improvement

Refer to the STG of the TAC, fig. 4.1(b). Once the common resource (further called CR) is acquired ($CRgr+$) and the grant to one of the clients is set (e.g., $C1gr+$), then the control waits for the corresponding client's request to go low ($C1req-$) before the MUTEX and the CR may be given to the other client (if already active). Now assume that $C1$ takes indefinitely long to remove its request, which forces $C2$ to wait indefinitely as well. Let's denote this *waiting time* ($C1gr+ \rightarrow C1req-$) as wt_{C1}. Once $C1$ has set its request low, a sequence of events must take place before $C2gr+$ could happen (we refer to the sequence $Cxreq- \rightarrow Cygr+$ as the *handoff* H_{CxCy} in the following). These include the release of the MUTEX and the CR (propagation delay of the arbiter while a grant switches from high to low, *tphl*), which happen simultaneously, followed by the events needed to reserve both of them for $C2$ (propagation delay when a grant switches from low to high, *tplh*). While it is not possible to optimize wt_{C1} (a client can keep the arbiter locked for as long as it needs it), the latter two, being local to the TAC, however, can be reduced.

Now refer to fig. 4.2. Gates $\{u3, u5\}$, and $\{u4, u6\}$ form a mutual interlocking mechanism,

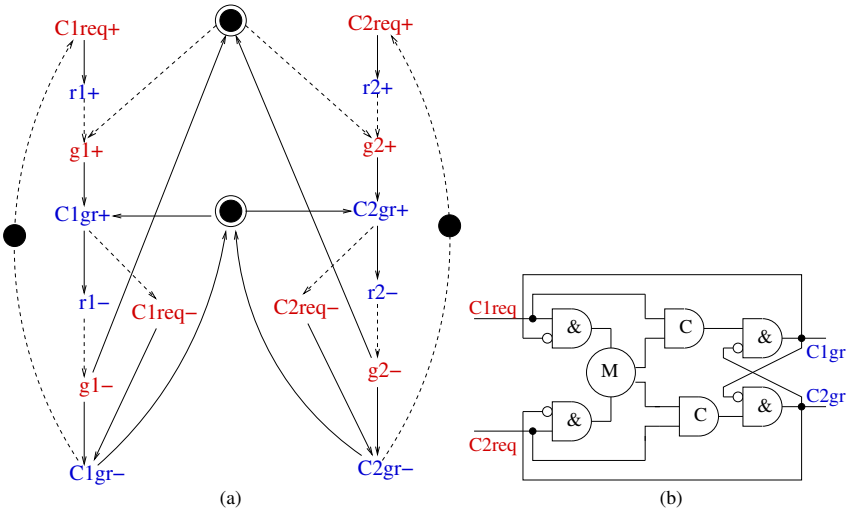


Figure 4.3: Proposed 2-way Arbiter: (a) STG, (b) conceptual schematic

through which $C1gr+$ prevents $C2gr+$ to happen, and vice versa. Consider if, in our example, $g2$ would be already active, and the CR reserved when $C1gr-$ happens, then $C2gr+$ only sees a delay of two gates, thereby completely eliminating *tphl*, and significantly optimizing *tplh*.

4.2.2 Design Concept

For simplicity we begin describing our methodology for the case of two clients. Fig. 4.3(a) presents the partial STG of the original TAC with all the instances relevant to the CR removed. A small difference that we have brought in into this mechanism is the release of the MUTEX ($r1-$, $r2-$) without waiting for the client's request going down: since the mutual interlocking already prevents the pre-mature handoff, the MUTEX is free to be allocated to the other client. In simple words, $Cygr+$ will not happen until $Cxreq-$ has happened, however Cy already possesses the MUTEX.

Fig. 4.3(b) presents a possible implementation of the STG. Although our strategy clearly allows the waiting client to acquire the MUTEX while the winner is yet to release it knowingly, one still might argue that the delay introduced through the C-gates and the cross coupled AND-gates, would be larger than that for releasing/acquiring the MUTEX itself. If so, there is no obvious benefit in terms of performance, and we unnecessarily introduced a number of gates in the system. The benefit, however, will become visible in the following section when we apply the same strategy to the TAC.

4.2.3 Adaptation to TAC

The STG shown in fig. 4.4 illustrates how the TAC adapts to the modifications we had previously described. A client may acquire the arbiter by following the identical sequence of events as in case of the standard TAC, i.e., the requesting client first acquires the local MUTEX, followed

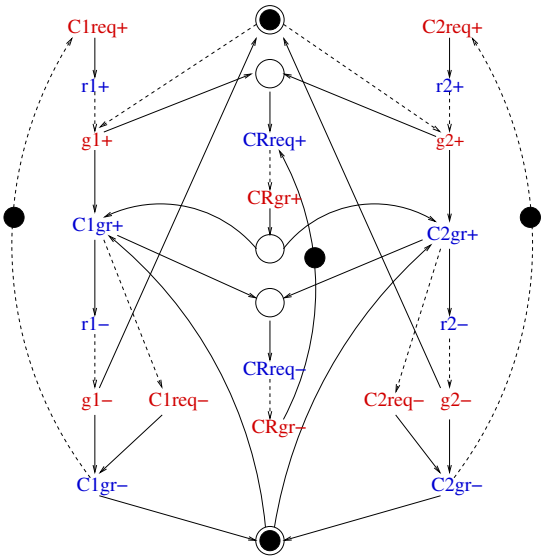


Figure 4.4: STG of the proposed TAC

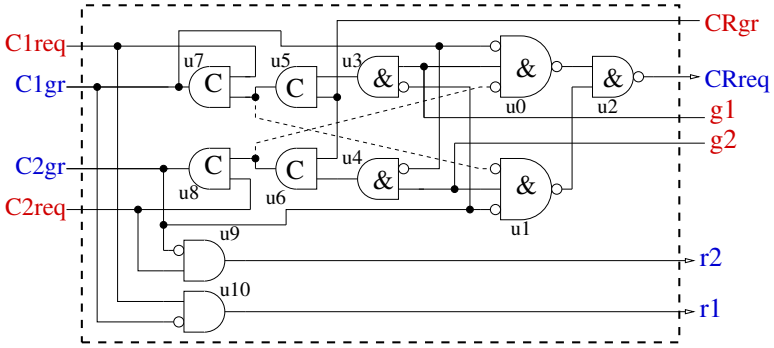


Figure 4.5: Circuit of the proposed 2-way Arbiter

by acquiring the common resource (like [50] we use a MUTEX here as well, further called CRM). The difference that we propose shows up in the other phase: As soon as one of the clients receives the grant, the CRM and the local MUTEX are simultaneously released, and the other client is allowed to lock both of them. The latter type of locking may be termed as *virtual locking*, since the first client is still in charge of the CRM until it literally lowers its request.

This STG when synthesized generates the schematic shown in fig. 4.5. Apparently it seems to have incurred a further overhead in terms of area and latency; the benefit, however, is hidden in the fact, as mentioned earlier already, that *tphl* no more depends on wt_{Cx}; it is initiated as soon as the grant is given. So far we have simply generated a circuit that meets our criterion. However, there are some other challenges that we address in the following in turn.

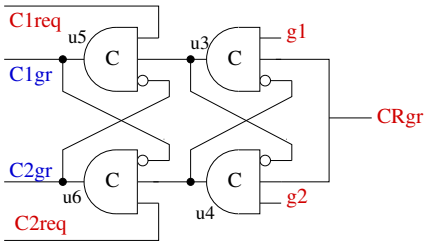


Figure 4.6: Proposed rapid interlocking within 2-way Arbiter

Rapid local clients' interlocking

As may be conveniently observed in fig. 4.5, the handoff H_{CxCy} still sees a delay of three C-gates and an AND-gate[1]. Since this interlocking is essential, it cannot be avoided, but this latency may be reduced to two C-gates by adopting the interlocking shown in fig. 4.6. This solution was achieved by manual optimization of the circuit. Note that all the C-gates used in the method are asymmetric, i.e., the inverted inputs are only relevant during *low to high* transitions of the gates. With symmetric C-gates in place, each of them can easily end up in a deadlock state. While *u5* and *u6* interlock each other to prevent the premature handoffs, *u3* and *u4* do the same to guarantee a safe handshake protocol with the CRM.

A small improvement in terms of performance may be brought into the circuit by adopting the methodology proposed in [50]: The local MUTEX and the CRM may be reserved simultaneously with a client's request. This only requires the gates *u0* and *u1* to have *C1req* and *C2req* as inputs, instead of *g1* and *g2* respectively.

Interlocking multiple TACs

In a tree structure arbitrating multiple clients ($n = 2^k$) requires multiple TACs (2^{k-1}) to be employed. In case of $n = 4$, a CRM may be placed as the common resource for both the TACs, as already mentioned. Interestingly, as soon as one of the TACs, say *T1*, has released the CRM, while one of its grants, say *C1gr* is still high, the other TAC, say *T2* may acquire the CRM, and one of its grants, say *C3gr* may also go high, thus violating the most fundamental property of an arbiter. This happens because our pipelined scheme relies on the mutual interlocking between the two grant outputs *within* a TAC, which is not effective across different TACs. Just like *C1* prevents *C2* within *T1* from getting a grant in parallel via interlocking, it must also do the same to *C3* and *C4* in *T2*. An SI solution will require complete handshaking between the two TACs, which may be conveniently inserted in the STG of fig. 4.4: The winning TAC, *T1*, sends out a *lock_others_request* signal right after $CRgr+$. On the receiving TAC, *T2*, the same signal appears as *lock_me_request*, which must be mutually exclusive with $CRgr+$, since the CRM already belongs to *T1*. From this point onwards, *T2* sends the acknowledgment, and does not allow any of its grants to go high, until the unlock signal has arrived from *T1*. This completes the handshake protocol between the contending TACs. Fig. 4.7 presents the STG described above

[1]Recall that we assume wt_{Cx} is sufficiently large to allow the other client to acquire the local MUTEX and virtually reserve the CRM in the meantime.

(for simplicity we have shown the case of a single client). This STG synthesizes to an SI circuit. However, this solution may incur a significant performance penalty since *T1* has to wait for the acknowledgment before its $Cxgr+$ could happen. This may be slightly relaxed by making the following timing assumptions.

Timing assumptions

We insert a (complex) AOI-gate between the output grants of the TAC that generates the *lock_others_request* signal, and a pair of AND-gates (one for blocking each client) that implements the required functionality of the *lock_me_request* signal, as shown in fig. 4.8. Recall from the previously discussed scenario that *T1* and *T2* can generate simultaneous grants if *lock_others_request* from the former does not reach the latter timely. The condition for safe interlocking is evaluated as follows: *T1* while releasing the CRM roughly sees the delay of two NAND-gates ($u0+$, $u2-$), handoff at the CR ($CRgr_{T1}-$, $CRgr_{T2}+$), and a wire delay[2] between the TAC and the CRM, refer to fig. 4.5. On the other hand, *T2* between reserving the CR and generating a grant sees two C-gates ($u5+ \rightarrow u7+$, or, $u6+ \rightarrow u8+$) and a wire delay. *T1* can block *T2* within a delay of one AOI-gate. For the TAC interlocking to be safe, the following condition must hold true, which we believe is quite simple to achieve;

$$\delta(AOI-) + \delta(wire) < \delta(u5+) + \delta_i(u7+) + \delta(wire) + \delta(u0+) + \delta(u2-) + \delta(H_{CRM})$$

Here $\delta(X)$ and $\delta_i(Y)$ refer to the switching and inertial delays of gates *X* and *Y* respectively. H_{CRM} refers to the handoff at the CRM. For simplicity, if we assume similar wire delays on both sides of the equation, then everything boils down to assuming that the AOI-gate will be faster than a C-gate, two NAND-gates and a MUTEX connected in series, which is very safe.

4.2.4 Unfairness Window

Fair arbitration demands that the grant be given to the first amongst all the clients that requested the CR. Neither the design of the TAC, nor of that proposed in [50], fulfil this demand. For example, consider a scenario where clients *C1* and *C2* request for the CRM simultaneously through TAC *T1*, and *C1* wins the grant while *C2* is put to wait. Now after a while *C3* makes its request through *T2*, which will immediately reach the other input of the CRM, since *C4* is still inactive on the same TAC. Although *C2* made its request before *C3*, the latter would unfairly win the grant as soon as *C1* has released it. This unfairness happened because the request from *C2* to the CRM still had to go through two NAND-gates within *T1*, and this time would always be sufficient for *T2* to acquire the CRM. In fig. 4.9 we have presented the maximum length of this unfairness window, i.e., the period during which a client can unfairly reserve the CR. The abbreviations *CR* and *M* refer to the delays inserted due to the CRM and the local MUTEX respectively, and *FL* and *BL* refer to the forward and backward latencies of the TAC ($Cxreq+/- \rightarrow CRreq+/-$, $CRgr+/- \rightarrow Cxgr+/-$). Note that the events shown in

[2]We assume that wires within the same TAC have zero communication delay, while those connecting with other cells cost some non-negligible delays.

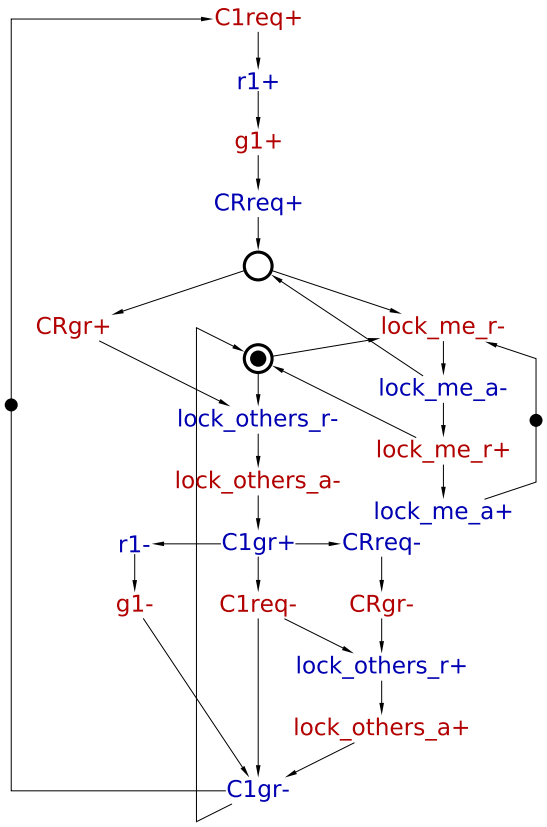


Figure 4.7: STG of the proposed TAC with multiple TACs interlocking

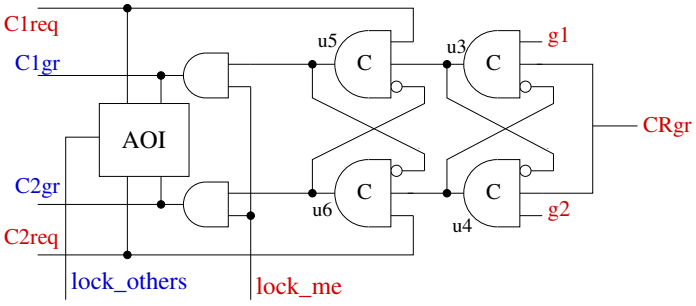


Figure 4.8: Schematic incorporating the interlocking logic

part (a) of the figure correspond to the response of the standard TAC setup, and (b) presents the behavior of the proposed circuit.

The request of *C2*, which occurred at time instance t_2, reached the CRM at t_{12} due to long wt_{C1}. *C3*'s request, meanwhile, reached and acquired the CRM at t_{11} and t_{13} respectively. Without losing generality, the following condition, if true, will lead to unfair grants;

$$t_{11} < t_{12}$$

By substituting the values for t_{11} and t_{12} as depicted, and making few simplifications, such as, FL and BL of *T1* are approximately equal to that of *T2*, and rise and fall times of each gate are identical, the condition leading to unfairness becomes;

$$t_8 < t_1 + 2(FL + BL) + wt_{C1}$$

In the worst case, *C2req* may occur at the same time with *C1req* and still lose the grant, i.e., $t_1 = t_2$;

$$t_8 < t_2 + 2(FL + BL) + wt_{C1}$$

where wt_{C1} may be substantial.

In contrast to above, the condition leading to unfairness using the proposed circuit becomes;

$$t_{11} < t_{17}$$

which upon simplification becomes;

$$t_8 < t_2 + 2(FL + BL)$$

Clearly, the unfairness window in the latter case is restricted to releasing and acquiring the arbiter in quick succession, and therefore guarantees relatively fairer arbitration.

4.3 Implementation and Evaluation

We implemented four different designs for a four-clients setup, (a) standard TAC, (b) Ghiribaldi's 4-way arbiter [50], (c) SPA proposed in [42], and (d) the pipelined high speed TAC proposed in this chapter. All of the designs were synthesized for $90nm$ technology.

4.3.1 Worst and Best Case Latencies

Table 4.1 presents the latency of each arbiter when just one client is active. Note that the Felicijan's design [42] presents a range of latencies; this is due to the different sized C-elements associated with each client: the lowest priority client has the largest C-element, and therefore, is the slowest as well. Something that is not apparent in the table is the fact that the cycle time for each design may be interpreted differently: For the first two, the given cycle times correspond to the best-case since this evaluation does not consider any delays due to the environment. In case the environment delays were considered, the cycle time for each of them would linearly

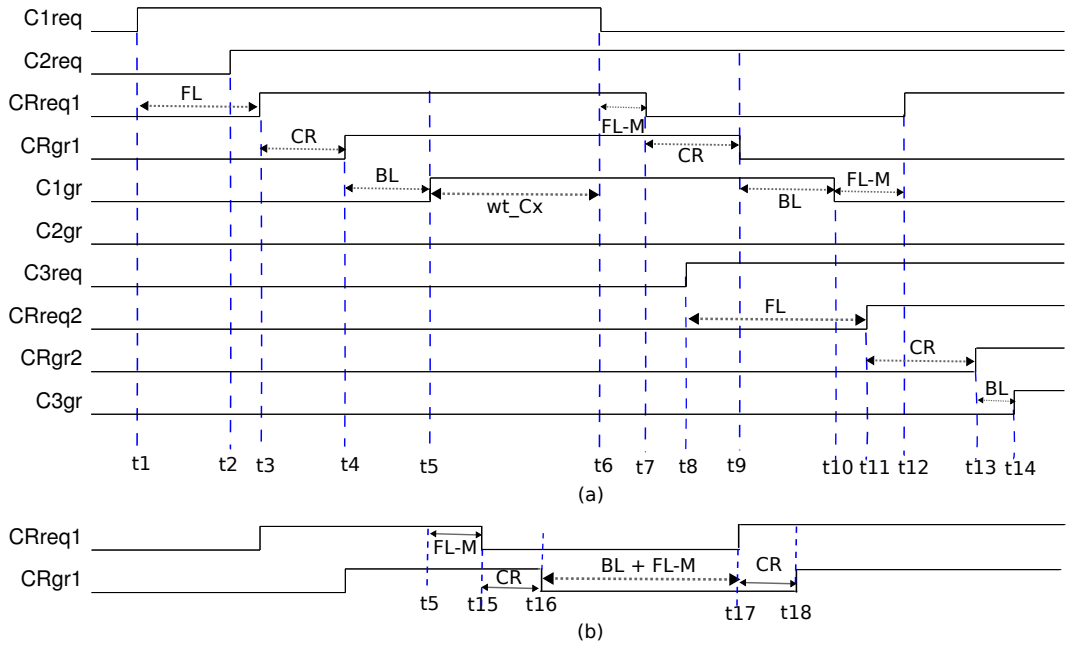


Figure 4.9: Worst-case Unfairness Window: (a) TAC, (b) Proposed Circuit

grow. On the other hand for the remaining two designs, this time corresponds to the worst-case analysis, since both of them have some (virtual) pipelining employed, which would only become visible when multiple clients were active and the delays of the environments were also considered. In simple words, increase in the cycle time (due to the environment) for one client, reduces *tphl* for itself, and *tplh* for the other client. This complementary behavior of the clients is obviously not visible in our evaluation. For Felicijan's work, this shall have an impact only on $tphl$, and that also would be marginal, roughly equivalent to saving a delay of a cascaded AND-gate and C-element pair. Refer to [42] for details.

Table 4.1: Arbiters' latencies for only one active client

Design	tplh (ps)	tphl (ps)	Cycle time (ps)
STD TAC	401	266	667
Ghiribaldi	324	343	667
Felicijan	278 — 536	217 — 336	495 — 872
Proposed	400	548	948

Finally for our design, the given values hold true if only one client is active, and wt_{Cx} is as small as the delay of an inverter, which, as we have already argued, is extremely pessimistic. In that case there is no slack time for our proposed pipelining to become effective. Fig. 4.10 depicts the improvement in $tphl$ with increasing wt_{Cx}. Starting from the delay equivalent to an inverter (corresponding to the worst case latency) up to a point around $360ps$, $tphl$ falls almost linearly

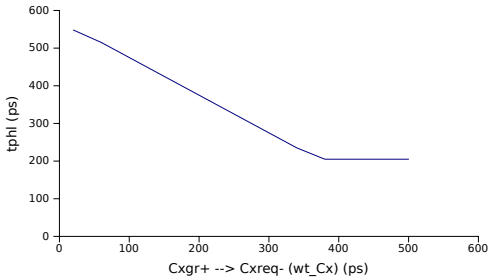


Figure 4.10: Impact of increasing wt_{Cx} on latency of the proposed arbiter

with wt_{Cx}. From this point onwards, the latency becomes constant (206ps), and is governed by the logic that we have added on top of the standard TAC. This adds to $tplh$ to represent the best case cycle time of 606ps for our design.

4.3.2 Handoff Latencies

Once again we assume that clients *C1* and *C2* share a common local MUTEX, and *C3* and *C4* do the same with each other. This means that a handoff between *C1* and *C2* (in any direction) will be much slower than their handoffs with *C3* or *C4* for all the designs except for the Felicijan's SPA in which H_{C1C2} shall be the fastest, and H_{C1C4} shall be the slowest. Similarly for all the designs except Felicijan's, handoffs between *C3* and *C4* will be slower than their handoffs with *C1* or *C2* on the other arbiter.

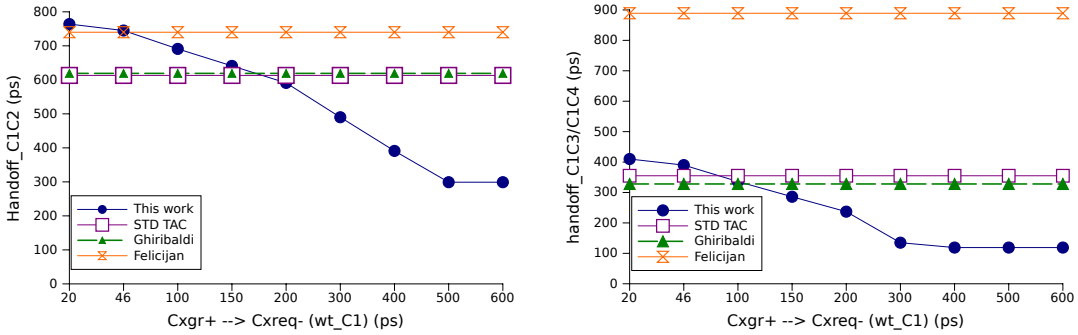
Table 4.2 summarizes these handoff latencies. For designs 1,2 and 4, the worst case latencies were computed by placing an inverter between *Cxgr* and *Cxreq* that would minimize wt_{Cx}. On the other hand, the best case latencies were computed by making wt_{Cx} longer than the arbiters' internal latencies. Note that the first two designs have identical best and worst case values. Because of its dependence upon wt_{Cx}, the results obtained for the proposed solution are so diverging. Given an environment satisfying our assumptions, the proposed work can result in significantly faster arbitration, especially with all eager clients.

As far as design 3 is concerned, it is rather difficult to estimate the worst case latencies. In their design, the authors have used two variable environments: right hand side (rhs), and left hand side (lhs) of the arbiter, the former of which must be slower than the arbiter's internal latency in order for the design to work correctly. In our evaluation, that is how we computed the best case latency for this design. As a result, the worst case is determined by the upper bound on the rhs logic, which is obviously design specific.

To evaluate the threshold client's delay essential for the proposed methodology to allow high speed resource sharing, we have observed the behavior by gradually increasing wt_{Cx} from 20ps to 600ps, and plotting it against the average case latencies of other designs. Fig. 4.11 presents the handoff latencies between the clients on the same and different TACs. It may be conveniently observed, that beyond the 200ps mark, the proposed methodology achieves the best throughput, which comes to saturation around 500ps mark. The saturation occurs since the elapsed time is

Table 4.2: Handoff Latencies

#	Design	best case $C1 \rightarrow C2$ (ps)	worst case $C1 \rightarrow C2$ (ps)	best case $C1 \rightarrow C3$ $C1 \rightarrow C4$ (ps)	worst case $C1 \rightarrow C3$ $C1 \rightarrow C4$ (ps)
1	STD TAC	613	613	355	355
2	Ghiribaldi	619	619	328	328
3	Felicijan	740	-	814 889	- -
4	Proposed	299	764	119	410



(a) (b)

Figure 4.11: Impact of wt_{Cx} on handoff latency: (a) same TAC, (b) different TACs

sufficient for the virtual pipelining to have effectively completed its task in the background, and further delay of the client should not bring any further improvements.

One advantage that the design 3 enjoys over the rest is its guarantee to not generate overlapping grants. In all other circuits, due to different rise and fall times of the standard gates, it may be possible that shortly before the grant to client 1 or client 2 has been removed, the grant to client 3 or client 4 is already set (this equally applies vice versa). Therefore all those designs require the clients to have some decoupling logic with the CR (that forces a null into the protocol no matter the grants are overlapping) to ensure safe handshaking. This additional logic will add a small performance overhead on designs 1, 2 and 4, which is not included in our analysis.

4.3.3 Throughput Estimation

To have a fair estimate of the best case throughput for each of the four designs considering equal priority traffic, we simulated two different orders of arbitration. In the first one, called *alternating* order, we made sure that the requests from the clients arrived simultaneously, leading to a *round* of arbitrations, in the order of grants G1, G3, G2, and G4 (therefore only observing the handoffs between the clients on different TACs). In the second case, called *sequential*, the requests from the clients arrived strictly in the order C1, C2, C3, and C4 (with sufficient delays

in between, so that the effect of handoffs between the clients on the same TAC could also be observed), leading to the grants being given in the same manner as well (G1 $\rightarrow$ G4). Table 4.3 presents the throughput for each design corresponding to the two orders of arbitration, measured in Mega rounds (of arbitration) per second (Mrps). It is evident from the results that on average, the proposed work promises around 61.28%, 69.24%, and 186.85% higher throughput than the designs 1,2, and 3 respectively.

Table 4.3: Comparison of Throughput

#	Design	Throughput (Mrps)	
		Alternating	Sequential
1	STD TAC	400	330
2	Ghiribaldi	384	312
3	Felicijan	206	206
4	Proposed	666	515

4.4 Summary

In this chapter we have proposed a novel tree arbiter cell that allows a pipelined processing of requests, i.e. arbitrating for the next request while the current one is still ongoing. The extra logic required for this feature initially increases the arbiter delay; however, in the relevant case of frequent requests from different clients our scheme yields a considerable speed-up. We have introduced an inter-TAC communication path for cascaded use of our TAC cell that not only enforces exclusive activation of a single grant at a time, but also improves the fairness of the arbitration process. Our simulation results clearly indicate that in most realistic cases our scheme provides superior performance; in an example NoC application we gained a speed-up of 61.28%, 69.24%, and 186.85% as compared to three different designs from literature. In environments where one client is more eager than the rest, designs 1 and 2, having the smallest cycle times, shall prove more useful than the proposed methodology.

CHAPTER 5

Protection of FIFO Control Path

While it is well understood how to efficiently protect the data path in an asynchronous transmission channel against transient faults, as we shall discuss in one of the coming chapters, much less is known about protecting the handshake signals along with their associated logic – mostly a Muller Pipeline – although these are equally critical for the proper function. In this chapter we analyze the possible failure scenarios in the handshake of a 4-phase bundled data protocol that can arise from transient faults and systematically elaborate mitigation techniques for the resulting effects, namely SETs and SEUs. By simulated fault injection we show the effectiveness of the proposed extensions for protecting the channel. We take care to make these extensions themselves immune against transient faults, and we prove their proper and deadlock-free operation under fault conditions by means of model checking. Finally we show that, while providing superior coverage, our approach is in line with comparable approaches with respect to the area overhead.

5.1 Background and Related Work

Recall from the background chapter that each stage of the Muller pipeline consists of a single MC with two inputs. The left input is connected to the request signal arriving from the previous stage, whereas the right input serves as acknowledge signal from the subsequent stage. The MC acts as a synchronization element which relays request transitions, as soon as a new input request is available *and* the previous handshake transition has been acknowledged by the successor stage. As can be seen in fig. 2.15 the output signal of the MC has multiple functions: It is used to drive the request signal for the successive stage and at the same time serves as acknowledge signal for the predecessor. Furthermore it is used as control signal which opens and closes the latches of the pipelined data path. As discussed previously, the MC's function implies waiting for a transition on each of its inputs to occur and changing its output afterwards. So the transition that arrives earlier just arms the MC, while only the transition on the other input will then actually create an effect on the output. Since the purpose of the Muller pipeline is to co-ordinate the flow of data tokens through the pipeline, its malfunction will eventually lead to data errors.

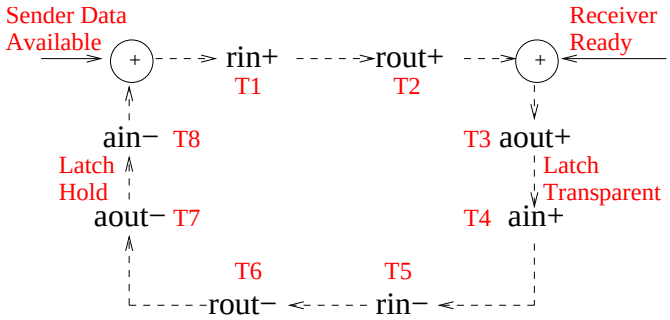


Figure 5.1: Handshake Protocol

In [82] a method is presented to increase robustness of control circuits. The general idea is to use a dual-rail encoding for all handshake signals, where the second rail represents the negated value of the first rail. Consequently the gates of the control circuit have dual-rail inputs and outputs. This redundancy is expected to increase resilience against SETs and SEUs to some extent. The main focus of this method, however, is to increase robustness against delay-variations, e.g., when operating with extremely low supply voltages.

Another method for protecting delay-insensitive circuits against soft-errors is presented in [62]. For this approach all the gates, which are already implemented as dual-rail gates to achieve delay-insensitive operation, are duplicated. Then all duplicated dual-rail signals (data and control) are *double-checked* by MCs. This effectively masks faulty transitions on one of the duplicated rails since the MCs only forward transitions which occur on both rails. However, duplication of the complete data path, already built with dual-rail components, significantly increases the circuit area.

A related line of research is concerned with the protection of asynchronous communication channels against transient faults. In [23, 98], e.g., delay-insensitive data transmission schemes are used and data words are encoded with a suitable error-correcting code.

5.2 Robust Asynchronous Muller Pipeline (RAMP)

5.2.1 Assumptions and Fault Model

For the elaboration of our protection scheme we make the following assumptions on faults as well as the operation of the pipeline:

A1) We assume, and aim to tolerate just one fault per handshake cycle.

A2) We consider only transient faults, i.e. pairs of opposing transitions on the same signal, whose distance defines the fault length. As will become clear later on, we will only tolerate faults with a length of less than a value δT representing the (best case) delay between the transitions *T1* and *T3*, fig. 5.1. This assumption is not very restrictive in practice, considering that, e.g., radiation induced voltage pulses are typically in the order of $100ps$, while the cycle time of the protocol is in the nanosecond range. For transients caused by crosstalk or power supply noise a similar argument can be made and our method applies as well. Later, we relax this assumption slightly.

A3) In this work we do not address metastabililty issues. Although these may emanate from transient faults, their occurrence may be considered sufficiently improbable, and, moreover, metastability mitigation is a topic of its own.

On a global level, the possible effects of an error in the operation of the Muller pipeline can be derived from its function to control the data flow through the latches:

C1) correct data tokens are lost,

C2) incorrect data tokens are erroneously inserted,

C3) the circuit ends up in a deadlock.

As far as *C3* is concerned, we have formally verified that the Muller Pipeline never runs into a deadlock for a single fault scenario (please refer to Sec. 5.3 for our formal verification methodology). Therefore we only need to consider the first two cases. In essence both cases resemble a mismatch in the sequence and/or timing between control path and data path, with the only interface between them being the pipeline signal *aout* that is also used to control the datapath latch operation, in this way synchronizing the two concurrent paths.

For a more detailed analysis we have to consider a glitch on every signal in every possible state of the protocol. This analysis can be simplified by considering the following general issues:

- As already mentioned earlier the MC operates on *pairs of transitions* that arrive on its two inputs. The earlier of the two transitions (say at input *A*) just arms the MC, and only the one arriving later (say at input *B*) will fire it. This principle masks all glitches on input *A* until the transition on *B* arrives. Depending on the actual sequence of transitions this may provide for some degree of inherent fault masking.

- Similarly, inherent fault masking may become effective when an erroneous data token (that may have been captured due to a glitch) is simply overwritten by the correct one before being captured by the successor stage.

- In this context it is very important to mention that the act of switching a latch to transparent is uncritical in the local pipeline stage, an error is manifested by capturing the data, i.e. with the falling edge of *aout*. It will be a key principle of our approach to prevent the latter in case of a fault (while we will not be able to prevent the former). However, a latch becoming transparent because of a fault, can let the latch in the successor stage capture an incorrect data; we have addressed this problem later in this section.

- The timing of an SET may be such that it does not show up as an isolated glitch but rather lines up back to back with an intended transition such that the timing of the latter is changed. In this way a correct transition may be moved to an earlier or later point in time (by the amount of the fault length at most). Due to the delay insensitive nature of the Muller pipeline this simply shifts all subsequent transitions of the control path accordingly without causing any errors there. Usually in a 4-phase design there will be enough safety margins in the relative timing between the two transitions to tolerate such shifts.

Considering these arguments only the following failure situations remain to be studied:

F1) Up/down pulse on rin: The leading up-transition can initiate a complete new handshake cycle without a new data token actually being available. The extra token inserted in this way corresponds to case *C2*.

F2) Down/up pulse on rin: Here the trailing up-transition of the transient can have the same effect as *F1)*.

F3) Up/down pulse on aout: The leading up-transition of this pulse has the potential to prematurely acknowledge a token to the sender which may therefore get corrupted (case *C1*).

F4) Down/up pulse on aout: Here the trailing up-transition may cause case *C1*, in the same way as *F3)*.

F5) Up/down pulse or down/up pulse on rout: As the receiving stage cannot distinguish whether the fault occurred on the predecessor's output (*rout*) or its own input (*rin*), these cases coincide with *F1)* and *F2)*, respectively.

F6) Up/down pulse or down/up pulse on ain: For the same reason these cases coincide with *F3* and *F4*.

F7) Bit-flips at MCs: If the MC is affected by a transient while being in the armed state (one transition already arrived) it will change its state, otherwise it will just produce a glitch at the output and then return to its correct state. In any case there will be an erroneous transition on both handshake signals that are sourced by its output, namely *rin* and *aout*.

These examples illustrate that, although the 4-phase bundled-data handshake primitive is quite a small circuit, yet it is prone to several types of faults and its protection is definitely essential.

5.2.2 Operation Principle

In the first place, it is obviously not possible for the circuit to distinguish between a correct and an incorrect transition, even more so since the execution of the protocol is done jointly by a pair of neighboring MCs which, therefore, locally only have a partial view of the protocol state. However, even with its restricted local view one stage can perform some sanity checks: From fig. 5.1 it is clear that a new input transition on *rout* (or *ain*) is not allowed to occur before the previous one has been acknowledged[1] by issuing *aout* (or *rin*). While this still leaves room for the leading transition of a transient to be accepted, the trailing transition will definitely be too early and hence be easily recognized as erroneous (recall assumption *A2*). Our aim is to detect this situation and suspend the acknowledgment even for the leading transition until the next, correct transition (recall *A1*) arrives. Although it is (fundamentally) not possible to mask the fault's leading transition, we can in this way block the handshake cycle from being completed. The subsequent correct data token will therefore overwrite the potential effects of the fault and retain correct operation of the pipeline (including the data path). Fig. 5.2 illustrates RAMP's envisioned operation principle, the dashed circle highlights a faulty up/down transition on *rin*.

Note that this blocking is enabled by the fact that the 4-phase protocol involves *two* transitions on each signal for completing a handshake cycle. Although this redundancy is sometimes criticized as inefficient, it can be conveniently leveraged here for fault tolerance.

[1] In this context the term "acknowledge" also applies to the sender issuing *rin* as an "acknowledge" for *ain*.

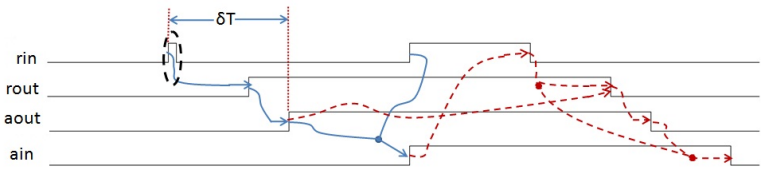


Figure 5.2: The operation principle of RAMP

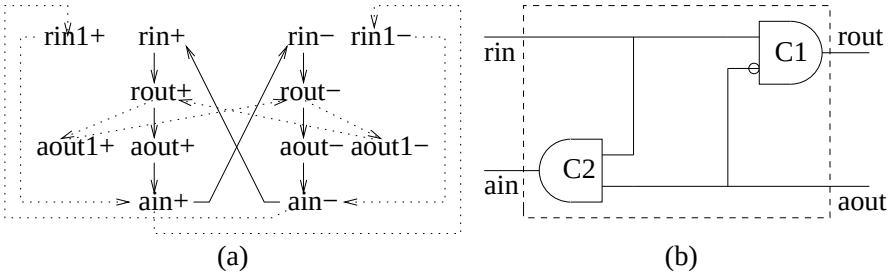


Figure 5.3: RAMP circuit for F2 and F4, (a) STG, (b) Equivalent circuit

It is clear from the functional principle that this approach only works if the trailing transition of the fault is recognized before the leading one has already been acknowledged. Therefore, as a necessary condition, the transient must be shorter than the regular delay between *rout* and *aout* as well as that between *ain* and *rin*.

5.2.3 Initial Circuit Design

For modeling the desired behavior of the RAMP circuit we used STGs, which can be conveniently synthesized into gate-level circuit implementations using the tool *Petrify* [28]. As a starting point we synthesized an STG identical to fig. 5.1, which resulted in simple wire connections between *rin* and *rout*, and *aout* and *ain* respectively. In this section we will now present, step by step, what modifications of this initial circuit were required to design a circuit able to tolerate all the seven fault scenarios *F1* to *F7* we have introduced in Sec. 3-A.

The simplest faults to be tolerated are *F2*, *F3* and *F4*. To be able to mitigate these faults, *rout* must fire only if there was a valid *rin*+ transition, and *aout* is still set to logic-0. Similarly, *ain* must fire only if *aout*+ has arrived following a valid *rin*+ transition. These requirements can be conveniently described in the STG as shown in fig. 5.3(a), where the *dotted* arcs indicate the modifications. Note that the new signals *rin1* and *aout1* are simply copies of the original signals *rin* and *aout* respectively. This trick was required to bypass the logic optimization performed by *Petrify*. Obviously the new arcs just add redundant dependencies to the STG for the sake of fault-tolerance. *Petrify*, however, does not consider faulty executions and therefore would simply ignore these dependencies. In the synthesized netlist we then manually replace the signal names *rin1* and *aout1* again with *rin* and *aout*. The resulting circuit, which now is able to maintain correct operation in case of *F2*, *F3* and *F4* faults, can be seen in fig. 5.3(b).

The next fault scenario to be considered is *F1*: A glitch on *rin* (*T1*) is supposed to force *rout*

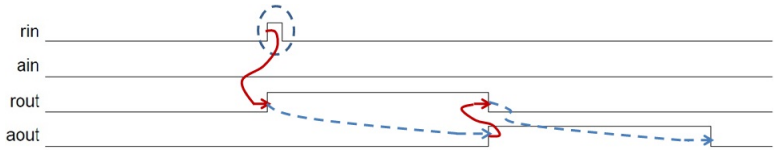


Figure 5.4: Simulation of up/down transient on "rin"

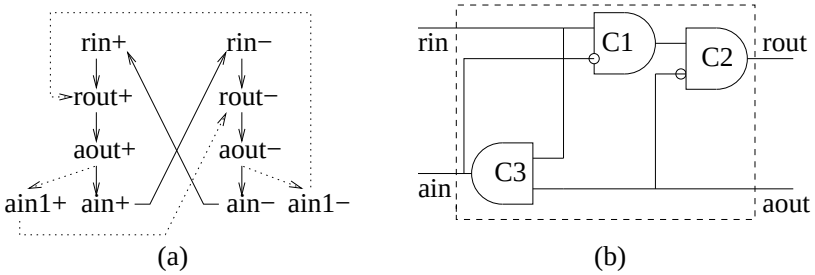


Figure 5.5: RAMP circuit for F1, (a) STG, (b) Equivalent circuit

to fire (*T2*), and then wait for *aout+* to occur (*T3*). According to the operation principle (please refer to Sec. 3B), once *aout* (*T3*) has occurred, the circuit must ensure that neither *ain* (*T4*) nor *rout* (*T6*) makes its subsequent transition until the next valid *rin* (*T1*) has happened. As long as our primary assumption *A2* holds, the circuit given in fig. 5.3(b) guarantees the former. However, the simulation given in fig. 5.4 shows that the circuit fails to prevent *rout* from making its falling transition (*T6*), and as a result there is a spurious completion of the handshake on the receiver's end. Obviously, this property is not a threat for the local pipeline stage, say n, but the successor stage $n + 1$ will definitely consume an invalid token, which completely destroys the operation of the pipeline. To cope with this situation we need to add another layer of protection between *rin* and *rout*. Since *ain* is supposed to remain stable until the valid *rin* arrives, it can be used to "sanitize" transitions on *rout*. This behavior is modeled with the STG shown in fig. 5.5(a). Note that we modified the *original* STG again for the sake of simplicity. After synthesis this STG was translated into a single MC with the input *rin* and $\neg ain$. The result of manually merging this MC into the circuit from fig. 5.3(b) can be seen in fig. 5.5(b). As required, the additional MC prevents *rout* from making its falling transition (*T6*) until a valid *rin* (*T1*) happens. The new circuit now is able to withstand all fault classes from *F1* to *F4*.

The remaining faults from the list are *F5*, *F6* and *F7*. Concerning *F5* and *F6* it can be argued that they are identical to the faults discussed above, in case of a closed loop RAMP stage (see fig. 5.6(a)). However, if we connect two successive stages as shown in fig. 5.6(b), these faults possess a completely different interpretation and effect. Consider, e.g., the case when both stages are in reset state ($rin2 = aout1 = ain2 = 0$), and an *up/down transient* occurs on *rout1*. This obviously forces the MC *CJ* to change its logic state. The key problem here is that *a short pulse is converted into an apparently valid signal*. For the left-hand stage (SS1) this does not have any effect since the correct *rin* signal will not allow the fault to propagate to *ain*. The same fault, however, becomes a valid *rin2+* (*T1*) for stage SS2 and will be forwarded to the subsequent

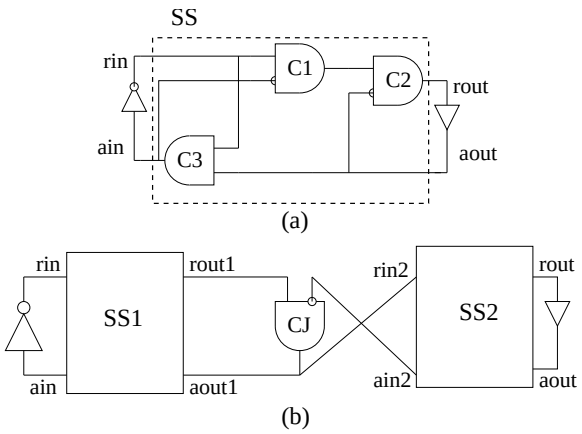


Figure 5.6: Closed-loop RAMP: (a) 1-stage, (b) 2-stage

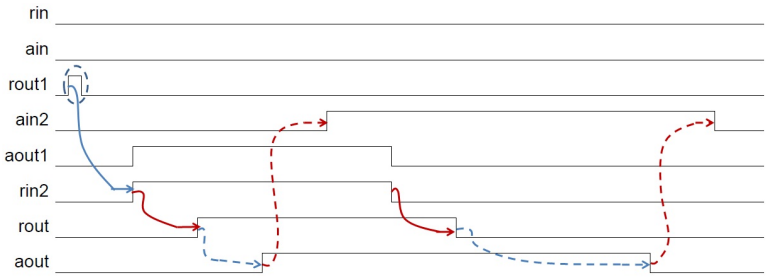


Figure 5.7: Effect of an up/down fault on "rout1" in a two-stage RAMP

stages. The corresponding waveform is presented in fig. 5.7. Please observe how a glitch on *rout1* leads to stable (not just glitches) transitions on *aout1* and *rin2* (*T1*). While *aout1+* is successfully prevented to propagate to *ain* of stage SS1, the faulty transition on *rin2* leads to an invalid activation of *rout* (*T2*). The receiver reacts on this request with an acknowledge on *aout* (*T3*), which subsequently completes this spurious handshake cycle.

Exactly the same situation occurs when there is a bit-flip/SEU in MC *C1* of SS1, MC *C3* of SS2 or MC *CJ*, which connects both stages. In all these cases the stages are on their own, i.e., they cannot mutually protect each other. While SS1 is able to detect and block the faulty transition on *aout*, SS2 cannot distinguish the fault from a valid input request. The solution to this problem is to duplicate the MCs and the associated signals. In this setup bit-flips in a single gate can be tolerated since the duplicated version of the gate still holds the correct value.

Fig. 5.8 shows the modified versions of the RAMP circuits with duplicated gates. Obviously the single MC, which was previously used to connect two stages, now needs to be duplicated as well (see gates *CJ1* and *CJ2* in fig. 5.8(b)). In order to provide a single request signal to the second stage and a single acknowledge signal to the first stage, the outputs of *CJ1* and *CJ2* then are joined by a third MC *CJ3*. Note that this gate is always driven with identical inputs (except for short transition times). Therefore it is resilient to SEUs as the correct output value would be immediately recovered. The worst thing that can happen is a short glitch at the

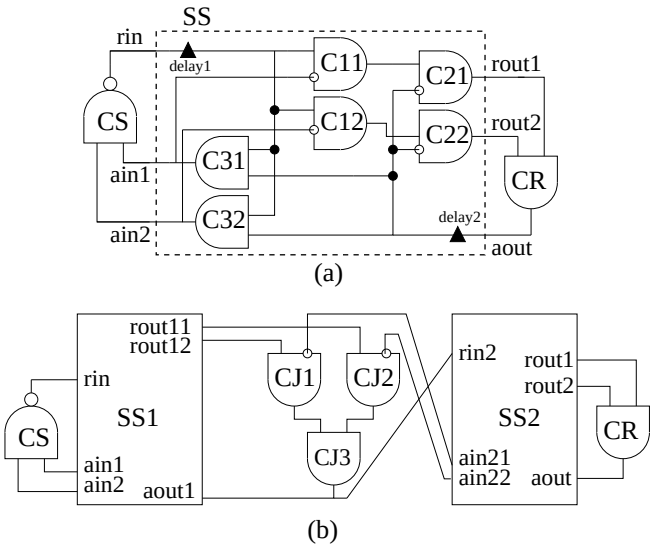


Figure 5.8: Closed-loop RAMP with duplicated gates: (a) 1-stage, (b) 2-stage

output. While in the control path, this is not a problem since glitches can be tolerated by SS1 and SS2 with the mechanisms discussed above, the same problem, however, can lead to data overriding in the datapath. Consider a down/up pulse on *CJ3* between stages *N* and *N+1*, call it *CJ3_N*, when *rout11* and *rout12* are already set to low. The falling transition of the fault would immediately lead to an *up* transition on *CJ3_N-1* making the associated latch, *latch_N-1*, go transparent, thereby overriding its previous data which still would not have been latched by the successor stage, i.e., *latch_N*, because the trailing *up* transition of the fault would have already put the *latch_N* into the transparent mode once again. Now once the correct *down* transition has arrived, *latch_N* would capture the new data while the previous has been completely lost. This problem can be solved without modifying *SS1* and *SS2*: *CJ3_N* is duplicated, call the duplicated version *CJ3_N'*, which now becomes the third (inverted) input of *CJ3_N-1*. This way, even if one of *CJ3_N* or *CJ3_N'* is faulty, the other one should be able to maintain the correct state, and probihits the latch from making an erroneous mode transition. These changes are presented in fig. 5.9 for a case of 2-stage pipeline. In case *CJ3* erroneously lets *ain1* and *ain2* to fall, *CJ3'* still shall be able to hold *CS* from generating a new *rin*.

The duplication ensures that the presented circuit now is able to mitigate the faults *F5* and *F6*, and for the same reason bit-flips can be tolerated in all gates as well. Therefore the last missing fault class *F7* is also covered. Note that we have assumed that both the sender and receiver add their own delays which are greater than the maximum length of the transients. These delays are marked as black triangles on *rin* and *aout* in fig. 5.8(a).

5.3 Formal Verification

Although we have derived our implementation in a systematic manner, it is mandatory to provide evidence for its coverage of all target faults. We used the UPPAAL model checker [72] for this

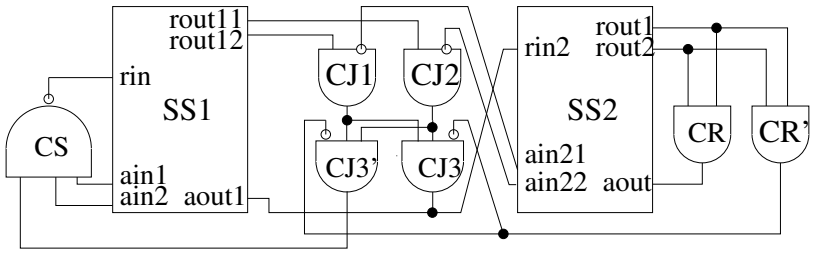


Figure 5.9: Closed-loop RAMP with protected datapath: 2-stage

purpose. Starting from suitable UPPAAL models for every used elementary gate, i.e., the 2-input MCs, inverter and buffer gates (see Appendix), we derived a gate-level representation of our circuit that we subjected to the model checking. Since UPPAAL models are based on networks of timed automata, the switching behavior of these gates could be nicely incorporated: Each gate model contains two stable states ($s0$, $s1$) and two transition states ($wait\_s0$, $wait\_s1$), where the gate is about to change from one stable state to the other within certain time bounds. The latter effectively model the possibility of input glitches being masked: If the input is not stable throughout the transition state, an ongoing transition can be reverted and the gate settles again in the old stable state.

5.3.1 Fault Simulation Methodology

For simulating faults we inserted saboteur units at each and every node of the model. These can produce a single transient pulse on the associated signals, whose length is restricted in order to satisfy assumption (A2). However, the exact point in time when the pulse is generated is completely arbitrary. In order to model soft-errors (bit-flips) of the state-holding elements of the RAMP circuit as shown in fig. 5.8(a), the MCs' models are able to spontaneously change from one stable state to the other. Again, such a fault can be triggered at an arbitrary point in time.

Table 6.3 lists the expressions used for checking the desired properties of the RAMP circuit. Note that the query language used in UPPAAL is a subset of TCTL (timed computation tree logic). The first formula verifies that the number of requests issued by the sender matches with the number of requests that arrived at the receiver. Obviously, the difference of these two numbers must be less than or equal to 1, as the sender can only be one request ahead of the receiver. By this check we can ensure that no correct token is lost (C1). The second and the third expressions are responsible to verify that no superfluous data token can be inserted by a fault (C2). Note that the second expression allows for the receiver to have seen *at most one* request more than has actually been issued by the sender. This can obviously be the case if there is a faulty transition at the *rin* input of the receiver. Recall that the expected behavior of RAMP in this case is to stall the handshake protocol until the sender issues a real request, which turns the faulty data token into a correct one. This behavior is nicely captured by the third property, which checks that if the receiver has processed a faulty request, the sender will eventually catch up. Then the two request counters have to be equal again.

The final formula in Table 6.3 verifies that there are no deadlocks on all possible execution

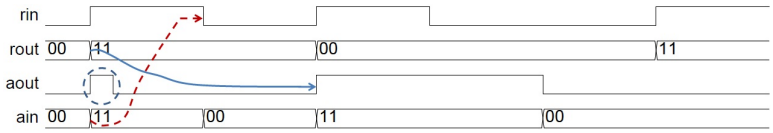


Figure 5.10: Faulty behaviour of the initial circuit

paths. Since we use a closed-loop RAMP model for verification, the execution should be infinite: After a handshake cycle is successfully completed, the sender immediately issues another request. Therefore the check for "no deadlocks" should evaluate to true. Note that by using this setup we also verify that every handshake cycle is correctly acknowledged. Otherwise the execution would stop, which in turn would violate the "no deadlock" property.

Table 5.1: Verified properties

	CTL-Expression	Check
1.	$A\Box$ (sender.requestCount - receiver.requestCount) ≤ 1	C1
2.	$A\Box$ (receiver.requestCount - sender.requestCount) ≤ 1	C2
3.	(receiver.requestCount - sender.requestCount) $== 1$ $-->$ (receiver.requestCount $==$ sender.requestCount)	C2
4.	$A\Box$ not deadlock	C3

5.3.2 Observations and Post-verification Modifications

The observations made during the verification process led us to the following two modifications:

Circuit's faulty behavior

The verification of the circuit of fig. 5.8(a) yielded one problem with an up/down pulse on *aout* in case of pronounced timing asymmetry between forward and backward paths:

Let's assume that an up/down pulse on *aout* coincides with a valid *rin*+ (*T1*) at *C31* and *C32*, and that the receiver is slower than the sender. So *rin* makes its falling transition (*T5*) before *aout* can make its rising transition (*T3*). Also according to (A2), the fault has supposedly settled down already. This allows *ain* to make its falling transition (*T8*), followed by a new *rin*+. This scenario is presented in fig. 5.10. Note that, when the receiver responds with the correct *aout*+, the second *rin*+ has already been issued. Similarly, the next *aout*+ arrives after the third *rin*+, which means the second valid data token is not latched properly; this violates the first property in Table 6.3, and corresponds to the case *C1* described in Sec. 3-A.

To remove this weakness, the circuit must prevent the fault from reaching *C31* and *C32* somehow. We can place another pair of MCs (*C41* and *C42*) that allows *aout* to reach *C31* and *C32* only if *rout1* and *rout2* have already fired correctly, see fig. 5.11. It is obvious that the fault on *aout* can still coincide with *rout* (even before the correct *aout* has arrived) and manage to reach *C31* and *C32*, and subsequently force *rin* to make the opposite transition. This time however, *C41* and *C42* having maintained their logic states, do not allow *ain*–, and therefore a

new *rin+* cannot be issued. Once the correct *aout+* has arrived, *rout–*, followed by *aout–* can occur. Only then *ain–* is allowed, which leads to the next valid *rin+*. This makes the feedback path completely symmetric to the forward one, and allows the receiver to be arbitrarily faster or slower than the sender. The correct behavior of the circuit against the discussed case is presented in a waveform in Sec. 5

Improvement in latency

Refer to fig. 5.8(a): By varying the upper and lower bounds on gate delays and repeatedly running the verification tests, it could be observed that the length of the transient to be tolerated on *aout* should be smaller than the delay of the feedback loop:

$$\begin{aligned}\delta t_{max,aout} < \quad & max(d(C31), d(C32)) + d(CS) + delay1 + \\ & max(d(C11), d(C12)) + d(wire)\end{aligned}$$

where *d(xy)* is an abbreviation for *delay of gate xy*. Similarly, for the transients on *rin*:

$$\begin{aligned}\delta t_{max,rin} < \quad & max(d(C11), d(C12)) + d(CR) + delay2 + \\ & max(d(C21), d(C22)) + d(wire)\end{aligned}$$

Interestingly, $max(d(C11), d(C12))$ is common to both the cases, which implies that it is possible to remove delay1 and delay2, and keep one implicit delay at the outputs of the *C11* and *C12* gates. This significantly improves the performance of the circuit, since one handshake cycle only needs two times the delay to complete, whereas, previously it was four times. In an ideal situation, where we can assume that all the MCs operate at the same speed, and *d1* and *d2* insert identical delays, this new circuit needs no modification at all. However in reality it is hardly possible to achieve such precision in digital circuits. Our *no deadlock* property indicated that the circuit would run into a deadlock if the two inputs of *CR* are at different logic levels (because of the difference in speeds of the two paths) when an SET strikes *CR*, resulting in a bit-flip. Solving this requires us to force *aout* to wait until both *C21* and *C22* have made their respective transitions. Therefore, we insert another small delay *d_prec*, acronym for *delay_precision*, as shown in fig. 5.11:

$$\begin{aligned}d\_prec >= \quad & |d1 - d2| + |d(C11) - d(C12)| + max(d_i(C21), d_i(C22))\end{aligned}$$

where d_i*(C2x)* refers to the inertial delay[2] of gate *C2x*. This ensures that no pulse is suppressed, irrespective of the relative speed of the two paths.

Note that *d_prec* does not depend upon *d(CR)*, which allows the sender and the receiver to run independently at their own speeds. Secondly, the difference in gate delays is some *picoseconds* for $90nm$ technology; therefore, *d_prec* merely incurs a negligible performance overhead

[2]An input pulse that can successfully reach the output of a component *A* must be longer than a specific delay called the *inertial delay* of *A*. Otherwise, the output of *A* will remain unchanged.

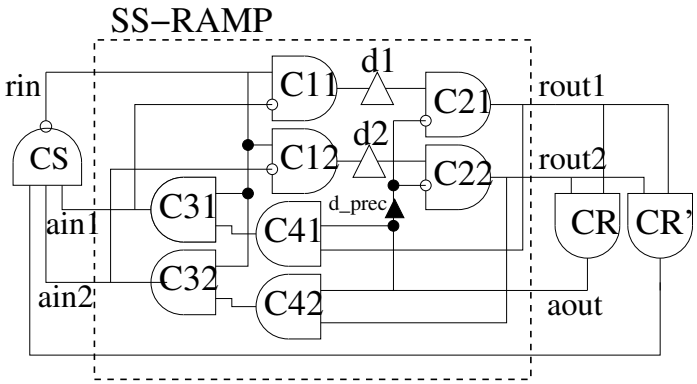


Figure 5.11: Modified closed-loop single stage RAMP

as compared to the case where we had two explicit delays on the sender and the receiver sides. Since, all the other faults are already tackled through duplication, this change of delays does not affect them at all. Fig. 5.11 presents the completely verified closed loop single stage RAMP, where *d1* and *d2* control the length of the transient faults to be tolerated.

5.4 Simulation Results and Discussions

In this section, we present the experimental evaluation of RAMP in terms of fault tolerance, and later we briefly comment on its performance and area overhead in comparison with the solutions based on duplication. All the simulations are done using Modelsim, in compliance with assumption *A2*: the delays *d1* and *d2* are fixed to $5ns$, and we vary the length of the transients between $1ns$ and $3ns$.

5.4.1 Simulation Results

To verify the correct operation of RAMP, we built a two stage RAMP circuit, identical to that of fig. 5.9 except that "SS" was replaced with "SS-RAMP" of fig. 5.11. We extensively tested the circuit for each and every fault described in Sec. 3-A . Fig. 5.12 presents a snapshot of a few simulations for faults 1 to 4, where *rin*, *rout1*, *aout1*, and *ain* are the handshake signals of stage-1, and *rin2*, *ain2*, *rout*, and *aout* correspond to the handshake signals of stage-2. The different circles highlight the injected faults on *rin* and *aout*. The important thing to note is that the number of valid input requests *rin* (total cycles less the encircled transitions) remains equivalent to the number of output requests *rout*, which verifies the fault-tolerance at *rin* since all the faulty transitions are mitigated before they could reach the respective outputs. Similarly, the faults injected at *aout* do not have an impact on the number of transitions on *ain* which verifies the fault-tolerance on *aout* as well.

Likewise, fig. 5.13 presents the fault injection and operation verification against *F5*, *F6*, and *F7*, i.e., faults at *rout1*, *ain2*, *CJ1*, and *CJ2* of fig. 5.9. If any of the applied faults manages to alter the state of the join MC *CJ3*, it would lead to consumption of an invalid token at SS2, case

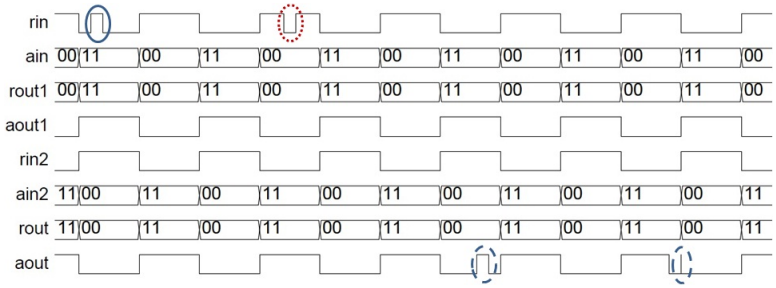


Figure 5.12: Simulation of faults 1 to 4

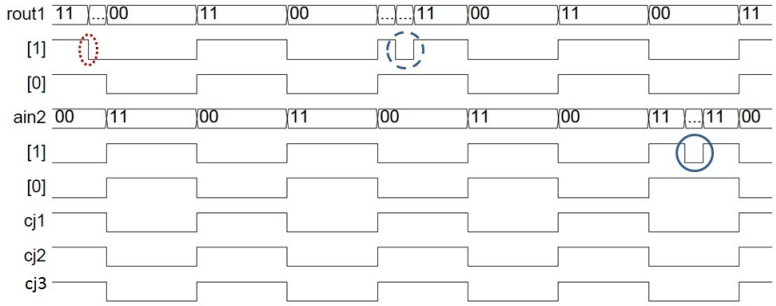


Figure 5.13: Simulation of faults 5, 6 and 7

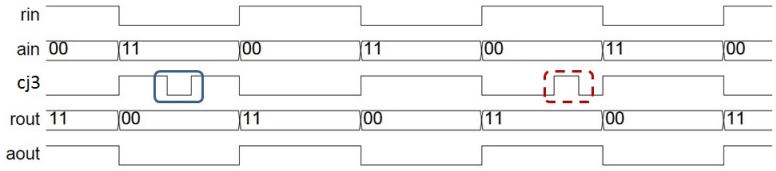


Figure 5.14: Simulation of faults at Join MC

C2. However, fig. 5.13 confirms that none of them manages to do that because the duplicated instance of the same gate maintains its valid state and prohibits the false transition to reach the output of *CJ3*. The fault in dotted circle represents a case of bit-flip, *F7*, while the dashed and the bold circles correspond to transient faults on one of the two wires of *rout* and *ain* respectively.

Finally, consider the case when the join MC *CJ3* is affected by an SET. Since its inputs are identical, this MC is not supposed to end up in a bit-flip. The only two possibilities, i.e., down/up and up/down are both presented in fig. 5.14. Importantly, none of the transients is able to affect either *ain* or *rout*, which confirms the fault-tolerance characteristic of RAMP against the faults occurring at the merge modules.

Table 5.2: Pattern of Area Utilization with Number of Pipeline Stages

Mechanism	Single Stage				Two Stage Pipeline			
	CE*	Data Path	SED/ SEC	Area (μm^2)	CE*	Data Path	SED/ SEC	Area (μm^2)
Non-FT	1	1	—	573	2	2	—	1147
Martin_FD	4	2	2	2289	8	4	2	3453
Martin_PD	3	1	1	2106	6	2	1	2697
RAMP	8	1	1	2333	20	2	1	3186
*CE — Number of Control Elements								

Table 5.3: Pattern of Area Utilization with *N* Pipeline Stages

Mechanism	N Stage Pipeline			
	CE	Data Path	SED/SEC	Area (μm^2)
Non-FT	N	N	—	573N
Martin_FD	4N	2N	2	1165N+2x562
Martin_PD	3N	N	1	591N+1x1515
RAMP	8N+4(N-1)	N	1	818N+36(N-1)

5.4.2 Comparison and Discussion

Inarguably, the proposed solution incurs a substantial area overhead to the original (non fault-tolerant) control logic. Notice, however, that compared to the data path with its wide latches and potentially complex logic function units, the area of the control logic is usually very small. In order to evaluate the area overhead, we compared RAMP with two possible solutions using Martin's *Duplicated Double-checking (DD)* [62] approach: In the first of the two solutions, called *Martin_FD* (FD for full duplication), each MC of the pipeline was replaced with a DD-gate[3], and the accompanying datapath was also duplicated. On the other hand, in the second solution, each MC was duplicated, and checked with a single MC, thus resulting in 3 MCs in total. Furthermore, in the datapath, the storage elements were not duplicated. The latter is termed as *Martin_PD* (PD for partial duplication). In order to protect the data paths, we made use of *Single Error Correction* (SEC) Hamming codes [51] for RAMP and Martin_PD. On the other hand, since Martin_FD already had duplicated storage elements, it just needed to detect an error using *Single Error Detection* (SED), and it should be able to correct it without any complex logic, and area penalty, since the latter is significantly more area efficient as compared to SEC. We have presented a comparison between the two schemes in terms of area and latency in [94]. In chapter 7, we have proposed our own error detection and correction mechanism based on *Double Error Detection with Retransmission* (DED) hamming codes, which could also be used

[3]A DD-gate is a four gate circuit that protects a plain gate by first duplicating it, and then double checking their states using two MCs [62].

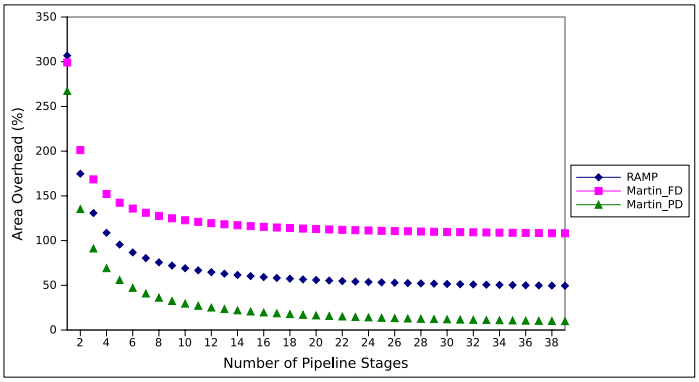


Figure 5.15: Area Overhead incurred by (1) RAMP, (2) Martin_FD, (3) Martin_PD

here in this work. However the circuits required for retransmission would unnecessarily increase the complexity without having any impact on the protection of the control path, we decided to make use of more commonly known algorithms for this work.

We varied the number of pipeline stages between 1 and 39 to observe the impact of the size of the datapath on the overall area utilization. The number of stages comes from HERMES NoC [85] in which the maximum number of flits per packet was 39, and the length of each flit (34 bits) was assumed from [93]. All the designs were synthesized for the $90nm$ technology. Table 5.2 presents the pattern of area utilization that each methodology follows with single and double pipeline stages, and Table 5.3 does the same for *N* stages. The area is measured as the sum of individual areas of the control path, datapath, and the error- detection/correction scheme (where applicable). In case of RAMP, this also includes the number of buffers needed to form the additional delays assuming the maximum length of the transient fault to be $1ns$.

In case of the non-FT version, the area of the *N* stage pipeline is simply the area of a single stage times *N*. For Martin_FD, the overall area is the sum of *N* times the area of control- and datapath, and two SED modules. The areas for Martin_Pd and RAMP are also computed identically, except for the area of SEC module, which is counted only once. It may be conveniently observed that Martin_FD causes the most overhead relative to the Non-FT version. Fig. 5.15 presents the comparison between the overheads of the three methods. With just a single pipeline stage, RAMP stands out the worst because of the expensive control path, along with SEC methodology which already utlizes significantly higher area as compared to the SED mechanism. Note that, with the increasing number of storage elements, Martin_PD turns out to be the most efficient approach; however, with a single checker MC in the control logic, it cannot guarantee protection against all transient faults. Martin_FD, as expected, incurs significant overhead, but guarantees tolerance against both SETs and SEUs. Notice, however, that the pipeline input signals are not duplicated and hence not protected. Their duplication, on the other hand would require adding redundancy to the source, which may become quite costly in terms of area. As can be seen in the figure, RAMP stands between the two alternatives, incurring around 50 percent area overhead, and guarantees full tolerance for all transient faults conforming to (A2), including the input signals. This clearly gives RAMP a profound edge on the state-of-the-art fault-tolerance

methodologies for asynchronous control circuits.

The performance overhead of the individual methods can be estimated by the number of MCs in one handshake round: For the non-FT version it is just 2 MCs, for the Martin_PD we have 4 MCs lined up. The Martin_FD approach requires more MCs overall, but still there are 4 MCs lined up sequentially along the path. Finally, RAMP requires 8 MCs[4]. For a $90nm$ CMOS technology our simulations yielded a handshake cycle time of $174ps$ for the two Martin-methods and $348ps$ for RAMP, which exactly confirms this estimation. While in this comparison RAMP obviously incurs a much higher performance penalty, one should, however, consider that we have looked at a very specific case that is particularly unfavorable for RAMP. In practice, a bundled-data pipeline contains combinational logic between the registers, and delay elements need to be inserted between the MCs in the control path to match the timing of data path and control path. Here the data path dictates the performance limit, and in case of complex combinational blocks in the data path the higher delay of RAMP in the control path will simply become invisible.

5.5 Summary

In this chapter we have motivated the need for protecting the control path of an asynchronous pipeline against transient faults, complementary to existing approaches for data path protection. For the very popular case of a Muller pipeline in a bundled data architecture we have analyzed the potential fault scenarios and, based on this analysis, systematically developed an extension to attain fault tolerance. Unlike mere duplication approaches, our design exploits the redundancy immanent to the 4-phase protocol for mitigating transient faults, thereby eliminating all potential single points of failure. By means of model checking we have proven that our solution can withstand all transient faults in the control logic, including the proposed extension itself.

[4] 4 MCs in a stage, and 2 on it's each side to connect with the neighbor stages. This does not include the redundant MCs.

CHAPTER 6

Fault-Tolerant Switch Allocation

As stated earlier, due to their inherently indeterministic behavior arbiters cannot simply be made fault tolerant by replication. In this chapter we present an in-depth analysis of a tree arbiter cell with respect to possible faults and failure modes. Based on these results we devise a fault tolerant implementation of this cell that carefully avoids all single points of failure and can hence withstand transient faults as well as bit flips in its stateful elements. We verify the fault tolerance of our implementation by means of model checking and compare its overheads and performance penalties with a TMR-based solution. While the validation confirms that our approach is indeed suitable for use within an overall fault tolerance concept, the penalties turn out to be lower than for a comparable TMR approach.

6.1 Related Work

In context of fault-tolerance, it is important to note that under fault free circumstances the sequence of req+ / grant+ / req- / grant- is strictly followed in the arbiter's operation, fig. 4.1. Any deviation in the protocol would spoil the timing closure, and hence indicates an error. The need for a fault tolerant arbiter is well understood in the literature, but not always correctly addressed.

Most of the existing works rely on modular redundancy with a majority voting mechanism. For example, in [27] the authors have used n-modular redundancy (NMR) [131] to guarantee reliability. Similarly, triple modular redundancy (TMR) has been widely adopted for components and links to ensure fault tolerance [130, 150]. The basic principle of replication-based schemes is to provide a sufficient number of correctly operating units whose majority can then mask the behavior of a minority of potentially erroneous units. Besides the significant (namely N-fold) area and power penalty those NMR schemes incur, they further suffer from several problems with the voter implementation: Obviously, a single, unprotected voter forms a single point of failure. Replicating the voter, on the other hand, causes a substantial further overhead, since for a thorough approach not only the voter circuit itself but in essence the whole design needs to be replicated. Finally, some authors protect the (simplex) voter by other means [90, 152], which again incurs an area penalty and makes the voter more complex.

Complexity of the voter is another issue by itself: In the simplest approach each outgoing signal X is derived by a purely combinational majority voting over the outputs X_1, X_2, X_3 of the associated replicated function blocks. In the context of arbiters, however, this raises a fundamental problem: The operation of an arbiter (specifically of the MUTEX inside it) is not fully deterministic. As outlined in Section 6.1, its specification allows the arbiter to make an arbitrary decision in case of simultaneous requests. Due to this operation principle two arbiters that operate in parallel may exhibit contradicting behavior even in the fault free case. Unfortunately this inhibits the above-mentioned masking principle of the NMR approach: Assume a set of N simple two-way arbiters, each with two request inputs (R1, R2) and two grant outputs (G1, G2) on the client side. For coincident requests R1, R2, one half of the components may correctly decide for G1, and the other half (also correctly) for G2. This "tie" situation can only be resolved by choosing an odd number of components (say N=2k+1) along with a threshold of k+1. Now imagine the case where k components decide for G1 and another k decide for G2, while the remaining one is faulty. Obviously the faulty component's behavior is now relevant for the output, rather than being masked. In this way the single faulty component can cause incorrect output behavior like granting both requests simultaneously, no matter how large N is chosen.

In principle, such types of invalid output can be identified and blocked by an intelligent voter that not just has a local view of one output, but a global view of all outputs, which allows the identification and prevention of illegal patterns. Such an approach necessarily involves a mutual interlocking of the grant outputs (to prevent the case of multiple grants) – which ultimately requires MUTEX properties from the voter. To some extent this moves the original problem into another box without actually solving it.

A solution of this kind is presented in [61] for a MUTEX circuit. Here duplicated input and output signals are used, and a simplex MUTEX is cleverly augmented with additional circuitry to attain a high degree of fault tolerance. At the output, however, special 'mutual excluders' are required for mutual interlocking. Another solution proposed by Pontes [108] made use of ${}^{N}C_2$ MUTEXs to arbitrate N clients, in such a way that each client competes with every other. Finally an MC is used to cross check if the client that has been given the grant actually made a valid request. Unfortunately, this mechanism only mitigates one simple fault on the client's request, and ignores every other possibility.

The decision problem with non-deterministic inputs, as mentioned above with the simple voters, is well understood in the distributed algorithms community, and the common solution there is to use consensus algorithms [3]. In the general case, however, consensus requires at least 4 components (more generally 3f+1 components to tolerate f faults), and it involves round based communication among those. Overall, it thus causes prohibitive area overhead and timing penalty and is therefore not practical for an arbiter.

Finally there are some existing solutions that make use of inherent information redundancy instead of the hardware, and have been shown to be much more efficient than TMR [35,101,154]. Since none of the papers actually presented the gate-level designs, it is impossible to verify whether those solutions would successfully eliminate all single points of failures. Radiation hardening by transistor sizing is another common approach to protect circuits from single event transients (SETs). As shown, e.g, in [143], this involves a huge (up to 100x) area overhead.

In this work, rather than replicating or hardening the TAC cell from fig. 4.2, we modify its

internal circuit to make it resistant against transient faults on all nodes and inputs. To this end we will exploit knowledge about the 4-phase protocol to detect and mitigate faults wherever possible. In the remaining cases we will employ hardware replication, and we will ensure that all single points of failure are carefully resolved as well. Note that this relieves us from all the mentioned problems associated with the voter.

6.2 Arbiter Failure Modes and Causes

In this section we will study the effects of single transient faults on an arbiter, more specifically the TAC from fig. 4.2. As a single fault we consider a bit flip in a single stateful element (MC in our case) or the transient inversion of the logic state on a single signal line. This will also include single faults in the MUTEX element. The term "single" refers to our assumption of at most one fault per client handshake cycle. Concerning the fault duration we envision pulses of width less than $1ns$, as they are observed with radiation particle hits[1] [54].

6.2.1 Failure modes on the client interface

Considering the description of the arbiter operation given in Section 4.1 we can state for the client interface the following requirements (*R*) along with failure modes (*F*) that result from their violation:

(R1): *The arbiter issues only one grant at a time.*

(F1) Multiple grants: More than one grant is active at a time.

(R2): *A grant is issued (eventually) iff the corresponding client request is active.*

(F2) Deadlock: The arbiter simply does not produce any grants even in presence of active requests. Since we excluded permanent faults, the TAC must have been driven to a deadlock state by a transient fault[2].

(F3) Undue activation of grant: A grant is activated without being caused by a preceding request.

(R3)*: A grant is deactivated (eventually) iff the corresponding request is deactivated.*

(F2) Deadlock: Like above the failure to remove the grant can only be explained by the TAC being in a deadlock state.

(F4) Premature Removal of Grant: A grant is deactivated although the request that caused it is still active.

6.2.2 Failure modes at the interface to the common resource

(R4): *A CR request is issued (eventually) iff a client request is active.*

(F2) Deadlock: The arbiter does not produce any CR request even in presence of active client requests.

[1] In rare cases those short pulses can also cause metastability issues. However, we generally do not consider this issue here, as that constitutes a topic of its own.

[2] We do not consider the case of starvation (only one requester gets the grant all the time) here as it only occurs if (1) one requester continuously activates its request and (2) we have a continuous sequence of transient faults with very specific, adverse timing.

(F3) Undue activation of CR request: A CR request is activated without being caused by a preceding client request. Under our fault assumptions the client will perceive this as an undue activation of grant (already covered by F3).

(R5): *The CR request is deactivated (eventually) iff the client request that caused it is deactivated.*

(F2) Deadlock

(F4) Premature removal of CR request: A CR request is deactivated while the client request that caused it is still active. In consequence this leads to a premature removal of grant, which is already covered by F4.

(R6): *The client grant is activated (eventually) iff the CR grant is activated.*

(F2) Deadlock

(F5) Premature activation of grant: A client request is answered by a grant before the CR is actually ready.

(R7): *The client grant is deactivated (eventually) iff the CR grant is deactivated.*

(F2) Deadlock

(F6) Premature removal of client grant: A client grant is deactivated while the CR grant associated with it is still active. The problem here is that the timing closure with the CR may get broken.

So in summary the failure modes of the TAC cell can be described by the cases (F1)...(F6). These favorably match with the failure list given in [35] for an arbiter circuit in an NoC router.

6.2.3 MUTEX failures

As outlined above it is not possible to protect the MUTEX by replication due to its indeterministic behavior in borderline decisions. At the same time, however, the circuit of a MUTEX is highly optimized to eliminate the potential for metastability, and any changes intended to harden it against transient faults would certainly compromise that important property. Consequently the best strategy seems to be hardening the TAC in a way to withstand erroneous behavior of the MUTEX. To investigate whether this is possible, let us examine the very common implementation of the MUTEX element from [63] that is shown in fig. 2.12.

Table 6.1 summarizes the effects of transient faults on all the 6 nodes it comprises[3] in all states of the circuit. Note in the table that the symbols $\downarrow\uparrow$ and $\uparrow\downarrow$ indicate faulty down-up, and up-down pulses respectively on the corresponding nodes. Similarly, the symbols $\downarrow$ and $\uparrow$ indicate the bit-flips from high to low, and low to high logic states respectively. In each line the signal transition(s) marked with a "*" correspond(s) to the fault origin, whose effects are visible in the columns to the right.

Note that entries 1-5 of Table 6.1 correspond to faults occurring at the input request *r1*, which not only covers potential MUTEX faults, but also faults originating from the TAC. Cases 2 and 5 do not lead to any further faults, since the victim of the transient fault is still waiting for its turn to acquire the MUTEX, so the request (including the fault) is masked by the NAND gate. Faults 1, 3, 6, 8, 11, and 12 are propagated as short pulses on the corresponding output grant *g1*, without disturbing *g2*.

[3] As the circuit is symmetric we can restrict our analysis to faults on half of the nodes.

Table 6.1: Possible Faults at the Outputs of MUTEX

No	r1	r2	a1	a2	g1	g2
1	$\uparrow\downarrow *$	0	$\downarrow\uparrow$	1	$\uparrow\downarrow$	0
2	$\uparrow\downarrow *$	1	1	0	0	1
3	$\downarrow\uparrow *$	0	$\uparrow\downarrow$	1	$\downarrow\uparrow$	0
4	$\downarrow\uparrow *$	1	$\uparrow$	$\downarrow$	$\downarrow$	$\uparrow$
5	$\downarrow\uparrow *$	1	1	0	0	1
6	0	0	$\downarrow\uparrow *$	1	$\uparrow\downarrow$	0
7	0	1	$\downarrow\uparrow *$	$\uparrow\downarrow$	$\uparrow\downarrow$	$\downarrow\uparrow$
8	1	0	$\uparrow\downarrow *$	1	$\downarrow\uparrow$	0
9	1	1	$\uparrow *$	$\downarrow$	$\downarrow$	$\uparrow$
10	1	1	$\downarrow *$	$\uparrow$	$\uparrow$	$\downarrow$
11	0	X	1	X	$\uparrow\downarrow *$	X
12	1	X	0	1	$\downarrow\uparrow *$	0

The remaining cases lead to effects on both grant outputs: while case 7 leads to short pulses on both grants, cases 4, 9 and 10 result in soft-errors (bit-flips) in the RS-latch formed by the cross-coupled NAND gates that inversely change both *g1* and *g2*. These latter cases resemble snatching the grant from the original winner of the arbiter and present it to the other client. In the design of our TAC protection scheme we will definitely have to consider all these cases.

6.2.4 Fault effects on the TAC

Knowing the failure modes of the MUTEX and the internal structure of the TAC we are now in the position to analyze how transient faults in those components map to the TAC failure modes (F1)...(F6) as identified above. We have done this by means of model checking (for details see Sec. 6.4). Table 6.2 lists the applied faults and their outcomes.

Table 6.2: Verification Results of Faults at TAC

Fault	C1req	C2req	CRgr	g1	g2	g1 and g2	Internal (node outputs)						
							u0	u1	u2	u3	u4	u5	u6
F1	$\surd$	$\surd$	$\surd$	X	X	X	$\surd$	$\surd$	$\surd$	X	X	X	X
F2	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
F3	X	$\surd$	$\surd$	X	$\surd$	X	$\surd$	$\surd$	$\surd$	X	$\surd$	X	$\surd$
F4	X	$\surd$	$\surd$	X	X	X	$\surd$	$\surd$	$\surd$	X	X	X	X
F5	X	X	X	X	X	X	X	X	X	$\surd$	X	X	X
F6	X	X	X	X	X	X	X	X	X	$\surd$	X	X	X

The table should be read as follows: A $\surd$ symbol in the row *F1* and column *C1req*, for example, indicates that no faulty pulse on *C1req* leads to behavior *F1* (multiple grants). Similarly,

the *X* symbol in the column *g1 and g2* and row *F4* indicates that if *g1* and *g2* flip simultaneously, then they can snatch the grant from *C1gr* and incorrectly present it to *C2gr* – corresponding to the scenario *F4*, and so on for the rest. In other words, all the *X* symbols refer to the faults that the TAC presented in fig. 4.2 cannot tolerate by itself, and it will generate incorrect outputs to the environment.

Throughout our simulation and verification activities we made the following additional observations that are not fully reflected in Table 6.2:
(O1): The circuit by default does not result in *F2* against any fault, so it guarantees deadlock freedom. However, this property will have to be checked again after our modifications.
(O2): In case of *F1*, one of the client grants will settle down eventually (since both being high will force the CR request to fall, and as a result CR grant must also fall). However, the length of the pulse generated at the client grant cannot be restricted, since it is determined by the CR, which might take arbitrarily long to release its grant.
(O3): The gates *u0*, *u1* and *u2* need no protection, since their purpose is to generate $CRreq$ whose faulty behavior is, if not detected and ignored by the common resource anyway, simply reflected to $CRgr$ as a transient fault.

6.3 Proposed Fault Tolerant Tree Arbiter Cell

6.3.1 Architectural Considerations

We have already laid out why it is not possible to protect the MUTEX (or the complete TAC including the MUTEX, resp.) from faults by means of replication. Consequently, in our attempt to design a fault tolerant TAC we will, no matter which solution we finally choose, have to take care of handling all types of erroneous behaviors of the MUTEX as listed in Table 6.1. Another invariant will be to use unprotected (i.e. single rail, non-redundant) input and output signals. This decision distinguishes our approach from [61] and avoids the need for replicating CR and clients (i.e. allows protecting them by other means if necessary). So no matter in which way we finally protect the logic internal to the TAC, the interface lines will remain subject to faults, and we will have to handle that. However, faults *originating* on those signals (or on their combinational source gates) can, according to our fault hypothesis, only be transient; so they can be handled by glitch filtering in the receiving stage. So we finally end up with the following constraints:

- use of a single MUTEX only
- TAC must be capable of handling erroneous inputs from MUTEX, CR and client
- client must be capable of handling erroneous inputs from TAC
- for CR we assume it handles transient input faults by exhibiting transient faults of the same length at its output (which then need to be handled by the TAC). Otherwise it also needs specific protection.

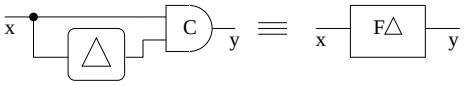


Figure 6.1: Glitch filter implementation

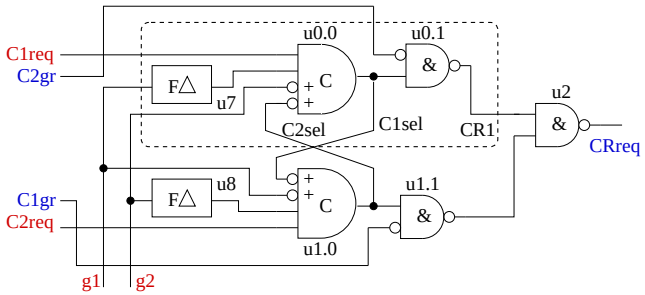


Figure 6.2: Modified Part of the Circuit protecting from *g1* and *g2* flips

The remaining choice is how to cope with TAC internal faults. Considering the issues with voting in a replication-based approach mentioned in Section 6.1, we chose to rather pursue a solution involving a fault tolerant implementation of the simplex TAC in the following.

6.3.2 Hardening the generation of *CRreq*

Let us start with the part of the circuit that is responsible for generating *CRreq*, which is comprised of gates *u0*, *u1* and *u2*. Obviously *u0* governs the activation of *CRreq* consequent to *C1req* under the following conditions[4]: (U0.A) *C1req* is active, (U0.B) *g1* has been received, and (U0.C) *C2gr* is inactive, ensuring that client 2 is given sufficient time to safely finish its access cycle to the CR. Gate *u2* only merges the requests from the two clients. A transient fault (glitch) on *C1req* may propagate to *CRreq* if client 2 is currently idle (U0.C), and if *g1* is received before the fault has vanished (U0.B). We can safely prevent the latter by adding sufficient delay to *g1*. More specifically we introduce a glitch filter, proposed in [5], as shown in fig. 6.1 into the signal line leading *g1* from the MUTEX to *u0*. In the dimensioning of the delay element we can exploit the knowledge that the fault length is in the order of some $100ps$ for radiation induced transients. A large delay will allow us to tolerate faults with longer duration, while at the same time performance of the TAC suffers. The choice of the delay must therefore be done with care and consistently with the expected fault duration.

This solution will also be effective against all glitches originating in the arbiter. From Table 6.1 we know, however, that we must also expect bit flips at the MUTEX. In that case U0.B may immediately be invalidated, thus *u0* will prematurely withdraw *CRreq* (F5). To safely prevent this we need to replace *u0* by an MC (*u0.0*) as shown in fig. 6.2. Now changing its output requires both *g1* and *C1req* to change.

The two inverted inputs of the MC turned out to be necessary due to subtle timing effects we discovered in our verification runs: Due to the propagation delay of the involved gates, the

[4]For brevity we will refer our explanations to client 1 only, but they equally hold for client 2, as well.

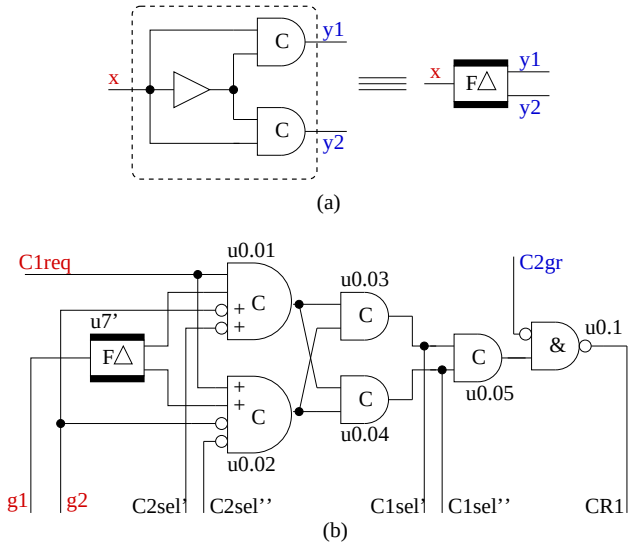


Figure 6.3: Modified Gates: (a) u7, (b) u0

interlocking established by U0.C needs some time to become effective. The specific case of a MUTEX bit flip occurring right during that stabilization phase can therefore become problematic.

We counteract this by two means: (I) We additionally establish a second, faster interlocking loop (that does in particular not depend on the response time of the CR), namely between *u0* and *u1*. (II) To protect this loop during its stabilization phase (where our verification has shown it is prone to deadlocking), we use *g2* as an additional input to *u0*. At the first glance using the mutually exclusive signals *g1* and *g2* as inputs seems redundant, but notice that we use the *non-delayed* version of *g2* here. As a consequence a bit flip at the MUTEX will first disable both gates and only after the delay safely enable one of them. This works reliably as long as the delay is larger than the stabilization time of the interlocking, which is easy to achieve. Preventing *CRreq* from being generated before *C2gr* has been deactivated is still required but does not have critical behavior, so we can still rely on an AND function for this task (*u0.1*).

It remains to be checked whether the changes we performed in the original circuit introduced any further fault potential. In particular bit flips of *u0* and *u1*, the glitch filter, as well as fault pulses on their inputs and outputs need to be considered.

Not surprisingly, our verification has indicated that the newly introduced MCs are prone to bit flips that cannot be handled by the circuit. Therefore we need to duplicate *u0.0* as shown in fig. 6.3(b), with *u0.0.5* as the merging element. Note that the latter does not require duplication, as in the fault free case its inputs will match, thus a potential state flip will immediately be restored (non-matching inputs plus a state flip represents a double fault). For the glitch filter we do not need the merging, since we can use both its outputs individually (see fig. 6.3(a)).

This circuit could finally be proven in our verification to reliably generate *CRreq* under transient faults.

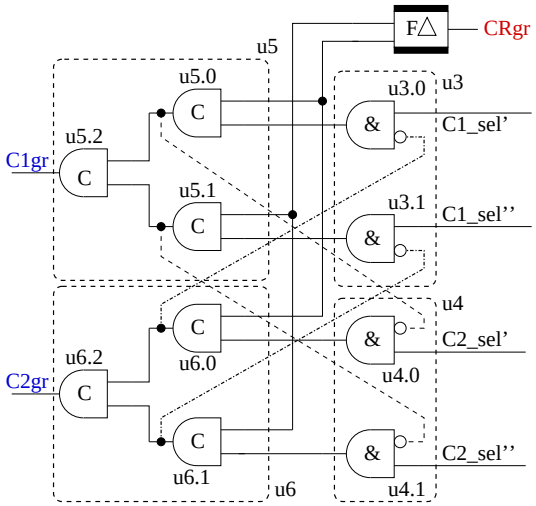


Figure 6.4: Modified Part of the Circuit with Duplicated Gates

6.3.3 Hardening the generation of *C1gr* and *C2gr*

The other part of the TAC circuit, constituted by *u3...u6* is responsible for generating the client grants *C1gr* and *C2gr*. More precisely, *u3* is responsible for arming *u5* to activate *C1gr* when (U3.A) *g1* has been received from the MUTEX and (U3.B) *C2gr* is inactive. The latter is used to interlock the two client grants. The actual activation (and deactivation) of *C1gr* is under control of *CRgr*, which finally makes *u5* fire.

Like above we have to take care of a state flip of *u5*, which we accomplish by duplication and merging, as shown in fig. 6.4. Still a transient at *u3* could consistently upset the state of both replica of *u5*, therefore we duplicate *u3* as well. Finally we need to prevent a transient on *CRgr* from upsetting *u5*. Here we can use our fault tolerant version of the glitch filter. This also covers glitches on *CRreq* that originated in the logic generating the latter and have been propagated by the CR.

A further important change in the circuit is to use the internal signal *C1sel* (that we derived to safely trigger the activation of *CRreq* as outlined above) instead of *g1* for arming *u5*. This is again to provide protection against glitches and state flips of the MUTEX. Since, due to the duplication of *u0*, we have a duplicated version of *C1sel* available, we use that to supply the inputs of the duplicated gate *u5* individually.

This completes our protection strategy for the TAC cell, and, as we will see in Section 6.4, provides protection against all hypothesized faults. The only uncovered faults are transients affecting the output nodes directly. Here we assume the connected clients (or the CR) to incorporate suitable protection measures at their inputs, like the glitch filtering.

6.4 Formal Verification

We once again made use of UPPAAL to formally verifiy the correct operation of our derived implementation. Table 6.3 lists the expressions used for checking the desired properties of the FT-TAC circuit. The first formula verifies that the two output grants are never active simultaneously, i.e., *F1* never occurs. The second formula in Table 6.3 verifies that there are no deadlocks on all possible execution paths. Since we use a closed-loop model for verification, the execution should be infinite: After a handshake cycle is successfully completed, the sender immediately issues another request. Therefore, if the check for "not deadlock" evaluates to true we have also verified that every handshake cycle is correctly acknowledged.

While verifying the third expression (*C2gr* == 0 for example), we made sure that the corresponding request (*C2req*) was connected to ground. This formula checks for a possible *wrong* grant, i.e., if a client did not request access to the common resource, the arbiter must *never* reserve the latter for that specific client. This equally applies to *C1req*.

Expression 4 verifies that *u5.2* and *u6.2* never enter the transition state $wait\_s0$ unless this was due to the removal of the corresponding client request. This checks for a premature removal of a client grant $(F4)$.

Expression 5 checks for the early grant fault, *F5*. In case of the unprotected TAC, *C1gr* (*C2gr*) may go high as a consequence of a transient on *CRgr*. However, once the transient fades off, this property would be invalidated indicating that the client grant was activated due to a fault. Similarly, the next property corresponds to the premature removal of the client grant, case *F6*, caused by a transient fault on *CRgr*. The check is essentially the same as above; in this case, however, we only need to consider the faults that occur once the client grant has already been activated following a correct common resource grant. The operator –> ensures that the property on its right hand side is tested only if the parameter on the left hand side holds true.

Table 6.3: Verified properties

No	CTL-Expression	Faults
1	$A\Box$ (C1gr && C2gr) == 0	F1
2	$A\Box$ not deadlock	F2
3	$A\Box$ (C2gr == 0) $A\Box$ (C1gr == 0)	F3
4	$A\Box$ (u5.2.wait_s0 imply !C1req) $A\Box$ (u6.2.wait_s0 imply !C2req)	F4
5	$A\Box$ ((C1req && C1gr) imply d_CRgr) $A\Box$ ((C2req && C2gr) imply d_CRgr)	F5
6	C1gr –> ((!C1req && !C1gr) imply !d_CRgr) C2gr –> ((!C2req && !C2gr) imply !d_CRgr)	F6

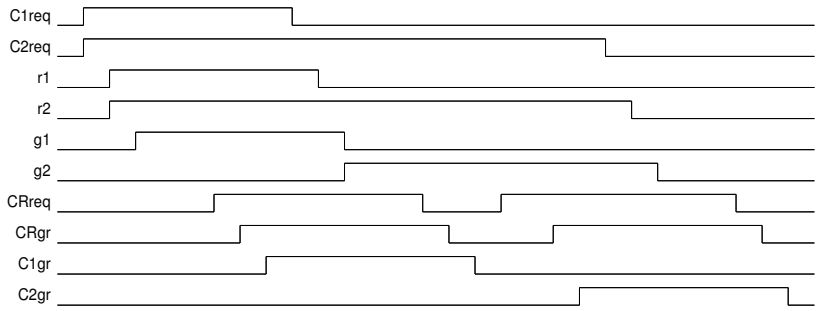


Figure 6.5: Operation of the FT-TAC in a Fault-free Scenario

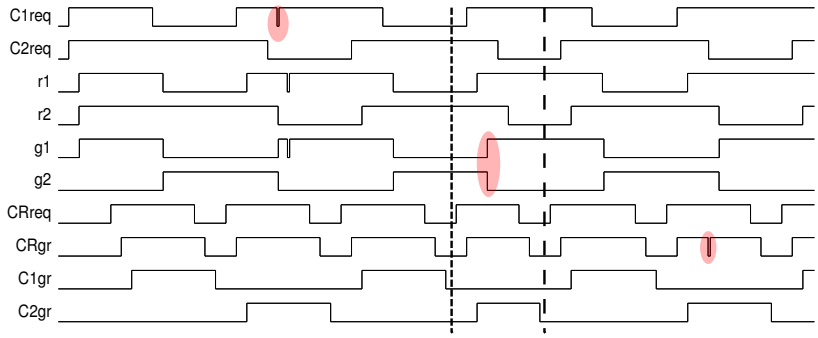


Figure 6.6: Mitigation of Faults Applied at *C1req*, *g1*, and *CRgr*

6.5 Simulation Results

Fig. 6.5 presents the operation of the FT-TAC in a fault-free scenario. For this simulation, as well as for the one to follow, we use a buffer in place of the common resource, which eventually generates grants to every incoming request, *CRreq*. We apply input requests from the two clients simultaneously, which, as can be seen in fig. 6.5, correctly win the arbitration in turn.

Fig. 6.6 demonstrates three critical fault cases which our FT TAC successfully mitigates: In the first case (left of the dotted line) *C1req* is hit with a down-up pulse (highlighted with a shaded circle). In the case of a standard (non fault tolerant) TAC, this would lead to *F4*; however, in this case, the pulse has been successfully mitigated by the short pulse filter at *g1*. As a result, $C1gr\downarrow$ only occurs once the correct $C1req\downarrow$ and $CRgr\downarrow$ have occurred.

The second case (between the two lines) corresponds to the bit-flip fault at the outputs of the MUTEX. Note that $g2\downarrow$ happens before $C2req\downarrow$, which means the standard TAC, in this case, would have either incorrectly snatched the grant from client2, and presented it to client 1 (*F4*), or given a grant to the latter simultaneously (*F1*), of course depending upon $CRgr\uparrow$. In our FT-TAC however, the situation is tackled by disallowing $CRreq\downarrow$ until the client request correctly makes the falling transition. This property is ensured by the MC in the forward path.

The last case corresponds to a simple down-up pulse on *CRgr* which is mitigated by the pulse filter. All the simulations and fault injections were done using Modelsim macro files.

6.6 Analysis and Discussion

Table 6.4 presents the comparison of four different designs (left column) that we synthesized for 90nm technology. For those we compare the delays for *acquiring* (column 2: tplh), and *releasing* (column 3: tphl) the grant by the client, assuming it immediately wins the arbitration and the CR is immediately available (i.e. CR is replaced by a buffer). Finally we also compare the area utilization (right column).

Table 6.4: Area and Latency Comparison

No.	Design	tplh (ps)	tphl (ps)	Area (μm^2)
1	STD_TAC	172.88	97.96	53.32
2	TMR_TAC	3488.83	2432.32	795.56
3	IFT_TAC	2305.8	2234.9	359.41
4	FFT_TAC	2426.32	2374.65	447.19

The first design (line no 1) is the unprotected standard tree arbiter cell from fig. 4.2 that we use as a reference. The second design is the triplicated (TMR) version of the standard TAC with a common MUTEX, five combinational voter circuits (for client grants, one for the CR request, and two for the MUTEX requests), each voter with a glitch filter at its output (which is still needed to cover faults in the voter circuit). For this circuit we observe an enormous timing penalty for the activation that is obviously due to the fact that three glitch filters and voters have been added in series to the critical path, each introducing $1ns$ delay for the filter and some $100ps$ for the voter. For deactivation the situation is somewhat more relaxed, since MUTEX and CR now operate in parallel rather than in series (see fig. 4.1), thus saving one filter delay. The considerable area overhead is also dominated by the glitch filters (implemented as long inverter chains), each of which contributes an overhead of $91.06\mu m^2$ to the overall area. Notice that this solution cannot reliably handle bit flips in the MUTEX, since that would require an interlocking between signals across the replica.

The next implementation, termed *Input-only Fault-Tolerant (IFT) -TAC*, still does not employ any duplication[5]; the circuit merely comprises glitch filters to protect its inputs against transient faults. Furthermore we have used MCs in place of the NAND gates, such that the protection also includes the bit-flip faults of MUTEX. The three input filtering circuits that are now employed, result in a massive area and latency overhead. However, as we only need 3 glitch filters, the situation is not as bad as with the TMR approach.

The final design is the *Fully Fault-Tolerant (FFT) -TAC* that additionally has all the internal gates duplicated, ensuring that no internal transient or bit-flip fault can lead to an incorrect behavior. As can be seen in Table 6.4 the duplication does not incur a remarkable overhead. Furthermore, it is clearly visible that this fully fault tolerant solution outperforms the TMR approach with respect to both delay penalty and area. Still the penalties are substantial. While one might argue that the area penalty, albeit large on a relative scale, is, due to the small size of the TAC in general, acceptable in terms of absolute area, the delay penalty seems prohibitive. In

[5]This may be a useful assumption if the TAC implementation is radiation hardened (albeit the resulting area increase is not considered here)

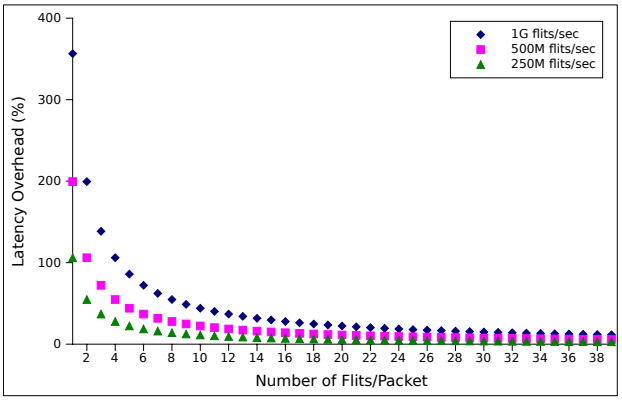


Figure 6.7: Affect of Packet Size on the Latency Overhead

comparison with [61] our approach buys its better overall area efficiency with a delay penalty that is determined by the maximum length of the transient faults to be tolerated (while there is no such restriction in [61]). However, the impact of this delay penalty on the overall system performance is often mitigated through an infrequent use of the TAC. In the case of a switch allocator [35] for the ANoC the TAC is only involved when the header flit reserves the output port, and when the latter is subsequently released by the tail flit. Consequently the throughput penalty caused by the TAC decreases with rising number of flits per packet. Fig. 6.7 illustrates this trend for three different data rates (the values have been chosen according to those reported for asynchronous NoCs in [15], [11], [36], [4]).

6.7 Summary

In this chapter we have shown that the usual technique of replication does not work for the specific problem of arbitration. We have analyzed possible fault sources and failure modes of a TAC, and subsequently we have proposed our novel fault tolerant implementation which successfully mitigates all anticipated transient faults. We have taken specific care to resolve all potential single points of failure. The fault tolerant operation has been verified by means of model checking. We have given a comparison between the original circuit, a TMR-based solution, and (different versions of) our approach concerning delay penalty and area overhead. The results have suggested that our approach is indeed useful, and comparable to an equivalent TMR solution in terms of area and performance overhead.

CHAPTER 7

Fault-Tolerant Inter-switch Communication

In this chapter we compare different redundancy schemes with the aim of providing a reliable communication service where error detection and correction is transparent to the higher system levels, and we also propose a new scheme for data (flit) protection. Our comparison is supported by results from simulation and synthesis. While all solutions are designed to withstand at least one transient fault, they largely differ in their area and delay penalty as well as potential extra coverage that some of them offer.

7.1 Related Work

Resilience to *cross-talk* and SEUs in NoCs is a widely discussed topic. Data protection mechanisms are either implemented on a *hop-by-hop* (HBH) basis, or at *end-to-end* (E2E) level. The choice of the approach strictly depends on the structure of the flit and the switching technique. The HBH approach becomes mandatory in case each flit of the packet contains the routing information (or some other control signals such as header-, body- and tail- flit indicator). On the other hand, if the header flit alone carries the routing information, then it is the only flit of the packet to be decoded on every hop; whereas, the payload can be protected using an E2E level mechanism.

Tamhankar et al. [135] provide data protection against soft errors, on HBH basis, using *delayed twice sampling*; a mismatch results in a retransmission. Pipelined buffers, each comprising a control circuit and two flip-flops (FFs), are inserted on interconnects. This leaves the NoC links to be synchronous, requiring stringent timing assumptions, and leading to increased dynamic power dissipation.

Franz et al. [46] carry the same approach to *delayed multiple sampling* (to handle cross-talk errors), and *single error correction and double error detection* (SEC-DED) (to address SEUs at input buffers on HBH basis). They base their approach on three times data sampling at different

points in time, and majority voting decides the correct data. As an extension to their own work, Franz et al. [45] propose an E2E packet protection by appending a *cyclic redundancy checksum* (CRC) to the payload. This proves to be an efficient approach as far as the throughput is concerned, but leads to several orphan messages on the network, which continue to utilize network resources and increases traffic congestion.

Dutta et al. [40] make use of *store and forward* (SAF) switching scheme and address soft-errors on links and input buffers based on HBH protection. They heavily equip each link and buffer with an encoder-decoder pair, and adopt *unequal error protection code* for multiple error correction capability. The SAF scheme itself requires a large storage space on each tile to hold the entire packet before it is forwarded; this is another significant drawback of this approach besides being slow in nature.

Although the E2E level encoding schemes normally lead to high throughput, they suffer severely from the errors in the routing information [101], which results in several packets being misdirected or lost. This further requires a complete packet retransmission from the source adding a significant performance overhead, and an increase in average packet latency. Gag et al. [77] present an attractive HBH header flit protection scheme, which reduces the number of erroneous header flits, and hence misrouted packets, and assume the payload to be protected at E2E level.

One of the hardest tasks while designing fault-tolerance is to choose the appropriate mechanism, i.e., to make a choice between hardware, information and time redundancy. Each one of these mechanisms has its own merits and demerits. With respect to *information redundancy*, there are a lot of error detecting and correcting codes in the literature. The most prominent codes are Hamming, Reed-Solomon, Berger, BCH, Cyclic, m-out of-n codes, etc. [29, 110]. Of these codes we choose the Hamming code [51] for our system as it is very efficient and has the lowest possible hardware overhead. *Time redundancy* mechanisms [52], are often augmented with diversity enhancements like shifting [102], rotating, inverting [114], or swapping operands, etc. [113]. We will consider these as well. For our system, however, we do not choose hardware redundancy as it increases the area and, in result, the power of the ANoC. In this work we implement, and present a comparative analysis of four different existing coding schemes, namely: *single error detection* (SED), DED, SEC, and *time redundant transmission with voting* (TRV). For TRV, we combine the time redundancy with swapped operands and inverting logic. The design and implementation of a novel approach, called *Adaptive Delayed Twice Sampling with Double Error Detection* (ADTS-DED), is also presented for our ANoC. The working principle and implementation of all these algorithms are presented in Sec. 7.3.

7.2 Baseline Interconnection Network

In this work we adopt our own ANoC presented in chapter 3, as the benchmark on-chip interconnection network. Since each flit contains the flit type that must be read on every node, our approach requires an HBH encoding scheme. For simplicity, we only consider one VC per IO port on every node. This simplification is reasonable since all the VCs are symmetric, and any data protection mechanism would operate on them in the same manner.

Amongst all the encoding schemes that we present in the next section, most require a retransmission mechanism to be able to retransmit the previously sent flit in case it was identified as incorrect upon decoding at the receiver's end. Furthermore, an input latch needs to be placed before the decoder at every input port irrespective of the encoding mechanism. Before going to discuss the encoding schemes, in the following we present a brief overview of both these enhancements we made.

7.2.1 Retransmission Module

The retransmission module, fig. 7.1, is placed between the router-encoder pair. Primarily, it comprises a two-input pull channel multiplexer (mux), and a *capture-pass* latch proposed in [133]. The selection line of the mux is the retransmission request line (error flag), which comes from the decoder of the neighbor tile. If this selection line carries a logical 0, it indicates that the previous flit has correctly been decoded by the neighbor tile. As a result, the mux forwards a new flit available at the router's output. The new data from the router is latched only once the mux requests for a new flit, i.e., error flag carries a 0. The request signal from the mux for a new flit makes the latch transparent, and the input acknowledgment makes it opaque, or capture the new flit. Note that the same capture signal on its way to the MC, is sufficiently delayed to allow the data to be safely latched. The delay element is not shown in the figure.

It is to be highlighted that all the control signals between each pair of tiles are duplicated, so that they can be protected against possible SETs as well. These include data validity signals (*req* and *ack*) associated with 34-bit flits, the retransmission request flag, and also the validity signals associated with the flag. In short, four wires carry the control signals for data flits in a specific direction, and the number of wires carrying the retransmission request flag and its associated control signals is six. The number of wires required to transmit a flit solely depends upon the encoding scheme being employed.

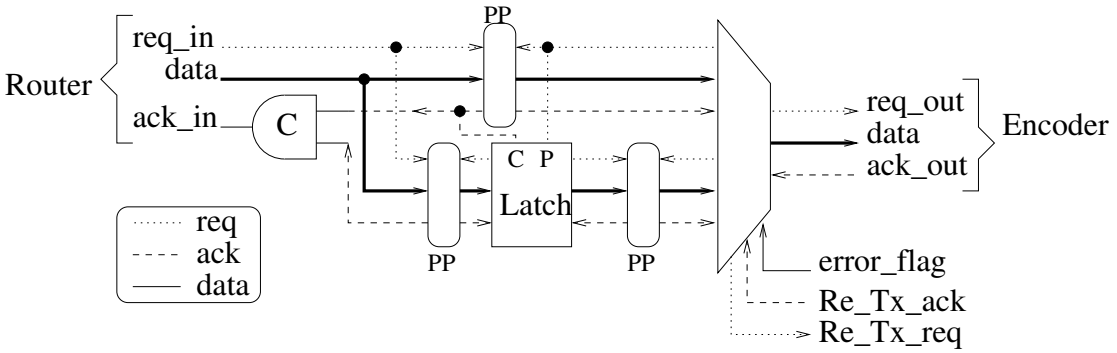


Figure 7.1: Flit Retransmission Logic

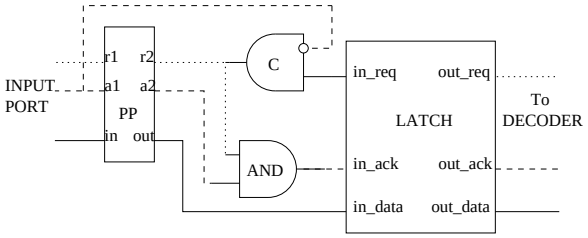


Figure 7.2: Input Module

7.2.2 Input Module

A conventional latch [133] operates in one of the two modes at a time, *capture* or *pass*. In the former, the contents are preserved by means of an internal loop in the latch (any faulty transition at the input is automatically mitigated); however in the latter, the latch behaves as transparent, thereby enabling any transition in the input to be visible at the output. Hence, it is essential that the pass time must be minimal to minimize the probability of transients reaching the output of the latch and subsequently the other combinational circuits that may react to every event, no matter valid or not. Fig. 7.2 illustrates our mechanism to open and close the latch in a manner as fast as possible. Once the *ack* signal makes a rising transition (forcing the latch into pass mode), the *req* signal is supposed to make a falling transition, and it alone is sufficient to set the *ack* signal low once again by means of an AND-gate. In this way, the latch does not need to wait for a round trip cycle of the control signals. In the same figure, the MC makes sure that the circuit does not violate its SI property. The box labeled as "PP" is identical to CWP described in Sec. 7.3.4.

7.3 Encoding Schemes

As a baseline for the comparison later on, we define the requirements we put on the transmission scheme as follows:

1. We assume, and aim to tolerate only one SET per flit per channel at any time.

2. We compromise the real-time performance of the ANoC by employing an HBH retransmission mechanism facilitated by the retransmission module described in Sec. 7.2.1. This allows us to save area and power.

3. Performance in terms of message delay and throughput (specifically in the fault free case, but also in case of faults) is not the primary concern, since our key focus is fault tolerance. However a relative comparison of performance will help in identifying a more efficient solution.

4. Given the increasing integration density, we do not consider hardware overhead too critical. However, more transistors consume more static power and are more prone to faults, therefore we include area (at least rough estimates) in our comparison.

7.3.1 Single Error Detection with Retransmission (SED)

As a first step in providing fault-tolerance for the interconnection network we use a Hamming SED mechanism [51] with retransmission. We need just one parity bit for this mechanism, so the datapath of the interconnection network increases by 1 bit and it is 35 bits. When a fault is detected a retransmission of the flit is requested.

While the encoding mechanism for SED is quite simple to understand, decoding on the other hand requires three combinational blocks, namely: *receiver*, which separates the received parity from the payload, *encoder*, which computes a new parity using the received payload, and *comparison* circuit, which compares the received and the computed parities with each other, and generates the error signal.

7.3.2 Double Error Detection with Retransmission (DED)

In order to tolerate both single and double faults in the interconnection network we used a Hamming DED mechanism [51] with retransmission. It requires a little more hardware than the SED mechanism. We need six parity bits for this mechanism, so the datapath of the interconnection network increases by 6 bits and it is 40 bits. Any single and double faults in the interconnection network and the decoder can be detected with the DED mechanism. If a fault is detected, a retransmission of the flit is requested. Any fault in the control signals can also be tolerated as these are duplicated. The control logic is similar to the one used in SED which has been explained previously.

7.3.3 Single Error Correction (SEC)

With the Hamming SEC code [51] we can correct all single faults in the interconnection network. Due to the correction ability offered by SEC there is no need for retransmission, so we gain performance relative to the above error detection mechanisms. The correction hardware consumes a lot of gates; it is, however, more or less equivalent to the retransmission block used in the other mechanisms. We need six parity bits for this mechanism, so the datapath of the interconnection network increases by 6 bits and it is 40 bits. The control logic for this code is almost identical to the previous two; the only difference is that it requires a fourth block which corrects the fault. Therefore, another control logic circuit has to be inserted. Moreover, the retransmission path is not required for SEC, which makes the overall design less complicated.

7.3.4 Time Redundant Transmission with Voting (TRV)

Time redundancy is an efficient approach to tolerate transient faults, because with time the transient faults fade off. To get some additional protection by diversity, we carefully combined recomputing with swapped and inverted operands, as the required hardware overhead is low compared to the other variants, like shifted and rotated operands. The idea is to transmit the flit three times at different instants t, $t + \delta t$ and $t + 2\delta t$. At time t we transmit the flit without any encoding. At time $t + \delta t$ we invert 25% of the bits (i.e. 8 bits) and swap the positions of MSBs with *least significant bits* (LSBs) and vice versa before transmission. The decoding is done at the receiver's end accordingly. Similarly at time $t + 2\delta t$ we invert 50% of the bits (i.e.

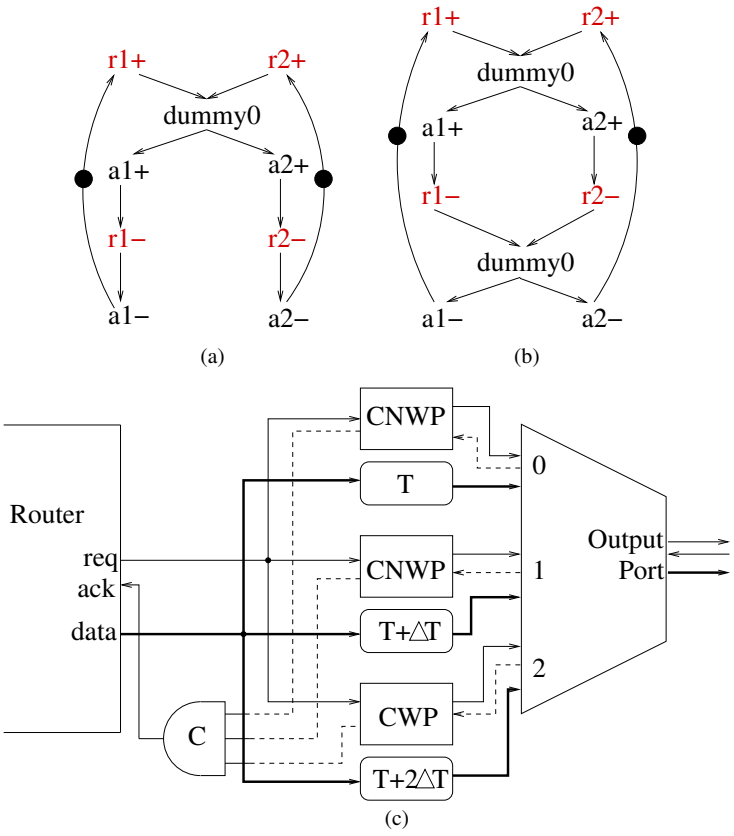


Figure 7.3: (a) CNWP, (b) CWP, (c) Block Diagram of the transmitter for TRV

17 bits) and swap the positions of MSBs with LSBs and vice versa. The first two transmissions are stored in a latch and when the third transmission arrives we do a voting of all the three and eliminate a potential fault. As these three transmissions occur in any (even the fault free) case, this scheme incurs a substantial performance degradation. This could be improved by performing the comparison already after the second transmission and requesting the third one only if the first two do not match. However, the control logic required to implement this conditional strategy is quite complex, which will finally again degrade performance. The control logic used for the unconditional time redundancy mechanism is explained below.

Transmitter

The transmitter for this encoding scheme, fig. 7.3(c), requires the router be forced to wait until all the three versions of a flit have been acknowledged. To achieve that we design a circuit called *customized non-waiting passivator push* (CNWP), fig. 7.3(a). *r1* and *r2* in the STG correspond to the inputs of the CNWP from the router and the mux respectively. When both of these requests have made their positive transitions, acknowledgments in both the directions are forwarded.

Since *ack* to the router is blocked through an MC, *r1* cannot make a falling transition until all the three versions of the flit have been forwarded. Whereas, *a2+* is alone sufficient to let *r2* make its falling transition, leading to the completion of the handshake at the right hand side of the CNWP, and hence the mux proceeds with the second version of the flit corresponding to $t + \delta t$.

The CNWP, as it is, does not work for the third copy of the flit. The reason is that, the moment *a1+* occurs, the MC's output goes high, and subsequently the router sets its *req* low. Obviously, the router, being on the same tile, is faster than the decoding taking place at the neighbor tile. As a result, the handshake on the left hand side of the CNWP completes first, and the router is allowed to place the next flit on its output even before the previous flit has been correctly decoded at the neighbor tile, resulting in data overriding. To tackle this issue, CNWP is modified to wait for both requests *r1* and *r2* to go low, and only then set the acknowledgments *a1* and *a2* to low. This way, the router can only place the new flit on the output once the previous flit has been correctly decoded at the neighbor tile. The modified circuit, fig.7.3(b), is called *customized waiting passivator push* (CWP).

Receiver

The control circuit for the receiver allows each of the first two versions of a flit to reside in a separate latch, from where they are forwarded to the majority voting circuit; the third version is directly forwarded to the voter. The 4-phase bundled data protocol still applies between both control-latch, and control-voter pairs.

7.3.5 Adaptive Delayed Twice Sampling with Double Error Detection (ADTS-DED)

In this work we slightly modify two encoding schemes [135], [46] to achieve better performance and higher robustness by reducing the number of retransmissions, and employing the DED mechanism respectively. The idea is to follow the double sampling approach [135], where a flit is sampled for the second time if the DED decoder circuit indicates a fault. If the decoder persists with an indication of a faulty reception even after the second time sampling, it is assumed that the fault has occurred during encoding at the transmitter's end, and the flit needs to be retransmitted. As a result, a retransmission request is sent to the previous tile.

The transmitter for the approach remains the same as before for the DED scheme, however the receiver is much more complicated. The waveform of the ADTS receiver and the block diagram of the entire mechanism are presented in fig. 7.4 and fig. 7.5 respectively. The process starts with the input request *rin*, in response to which the ADTS unit sends out a request *dec_in_r* to the decoder. Subsequently, the DED circuit generates an error signal *error_out_r*, which goes back to the ADTS unit. The latter then inspects the error flag *error_out_d*, and either forwards the flit to the router, *router_r*, in case of correct decoding, or goes for a re-sample by forwarding another request to the decoder, in case the first sample is detected as erroneous. Following the second faulty detection, the error signal *re_tx_d* is submitted to the retransmission module, with a validity signal *re_tx_r*. The SI property of the entire design is once again consistent through the RTZ bundled data protocol.

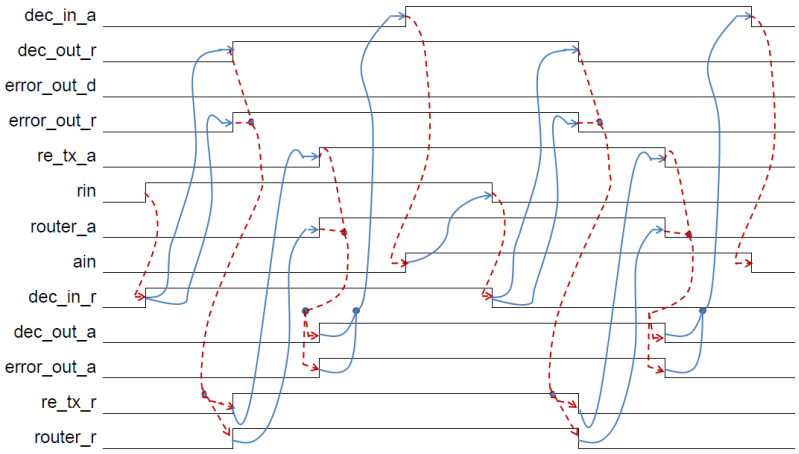


Figure 7.4: Waveform of the ADTS Receiver

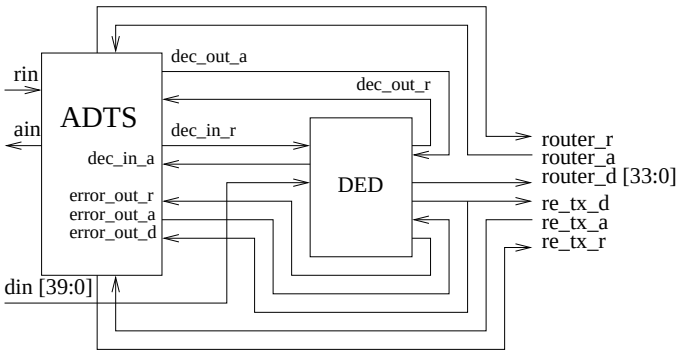


Figure 7.5: Block Diagram of ADTS-DED Mechanism

Unlike previous approaches [135], [46], a correct transmission does not require unnecessary multisampling in our scheme, and costs approximately the same latency as in the normal DED mechanism. In case of a transient fault on a specific channel that fades out with time, our scheme avoids a retransmission that would require a complete RTZ handshake protocol between the two ends, in addition to the long path delay between the two tiles, and obviously exposed to further SETs.

This mechanism adds a slight device utilization overhead, and the *maximum combinational path delay* (MCPD), which is only visible in case re-sampling has to be done, is approximately twice the normal DED operation. However, the probability of transient faults occurring on links is much higher than *stuck-at* wire faults, and *hard* faults within the encoder circuits, since the transient faults make up 80% of the total faults [136]. This indicates that our algorithm will outclass the other schemes, which base their efficiency solely on retransmissions.

All the methodologies based on retransmission require a simple interface (demux) between

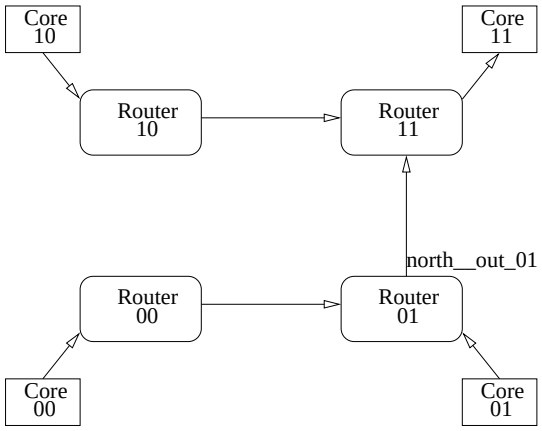


Figure 7.6: Fault Injection and Testing Methodology

the decoder and the router called *Decoder-Router Interface* (DRI). Its responsibility is to decide whether the decoded data needs to be forwarded to the router or a retransmission has to be requested depending upon the error signal received from the decoder circuit. For instance, the signals *router_r* and *re_tx_r* in fig. 7.5 have to be mutually exclusive. The demux merely forwards a valid data token in the right direction.

7.4 Simulation Results

We tested all the encoding schemes for a 2x2 2D mesh ANoC. We used Modelsim script files to inject 50 packets from each of the three tiles *00*, *01* and *10* towards the same destination tile *11* (please refer fig. 7.6[1]). Each packet consists of three flits (header, body, and tail), and we randomly injected a fault, in one of the flits of every packet. As mentioned in Sec. 7.2.2, the pass time of the input latch is the most critical point since any transition that occurs during this time would be visible at the output of the latch, and could further propagate to other modules. In order to make sure that all such critical points are tested, we carefully injected faults at both rising and falling edges of the data validity signals of the input latch.

7.4.1 Simulation Results of ADTS-DED Mechanism

For the case of ADTS-DED, an example is shown in fig. 7.7 (we focus on this one mechanism here for illustration). A transient is injected on bit *39* of *north__out_01d*. In this label, *01* indicates the *xy* coordinates corresponding to the position of the tile in the mesh, and the flit is being forwarded to the neighbor tile connected to its north output port. *lin_0a* refers to the data validity signal of the input latch, which operates in the *pass* mode between the rising and the falling edges of this signal. If the transient settles down well before[2] the falling edge, then

[1]The figure only contains the arrows indicating the intended direction of flit traversal.

[2]We do not consider any timing (setup/hold time) violation in this work.

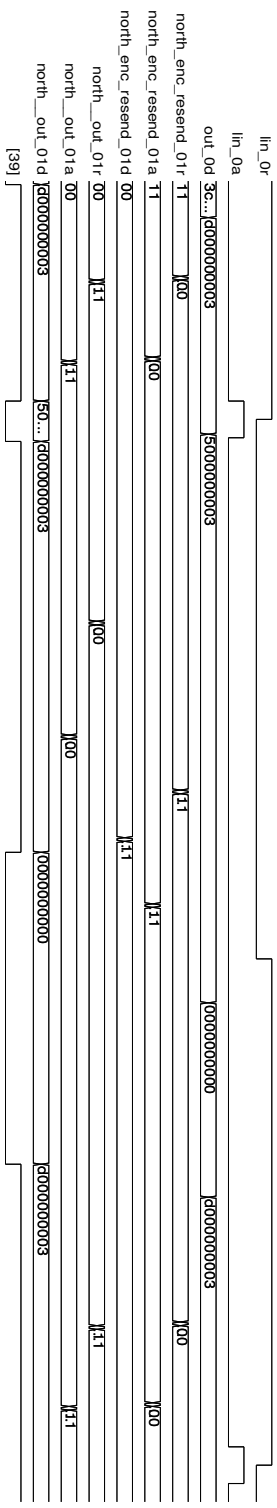


Figure 7.7: Fault Detection and Retransmission Waveform

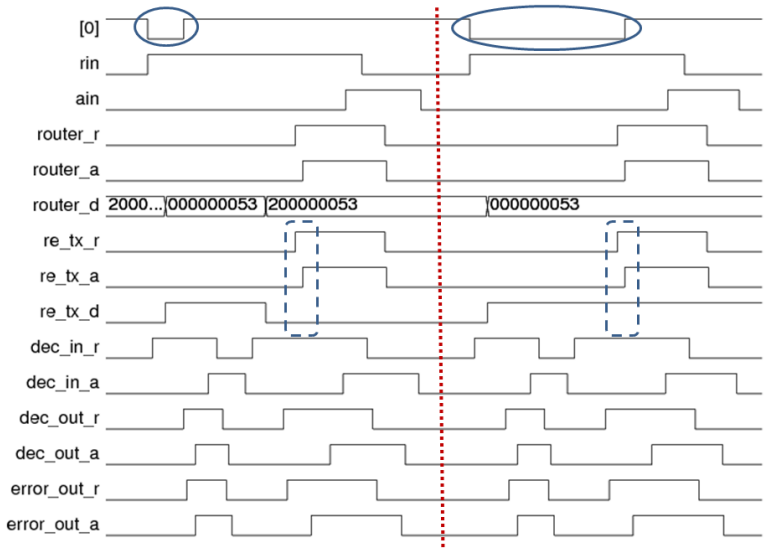


Figure 7.8: Working of ADTS in Presence of Faults

Table 7.1: Device Utilization Summary of the Common Modules in μm^2

	Router	Retransmission	DRI	Input Module
Area	3765.6	854.6	73.7	591.9
MCPD (ps)	1552	1215	724	1431

the fault is masked automatically. On the other hand, if the fault overlaps the falling edge, then it must be caught followed by a retransmission. Fig. 7.7 depicts the latter case. The signal *north_enc_resend_01d* indicates the retransmission request flag received from the neighbor tile, and *north_enc_resend_01a* indicates its validity. When both of these signals are at state *11*, a retransmission takes place. Subsequently, the correct flit *d000000003* is retransmitted. Fig. 7.8 explains the working of the ADTS module in the two possible scenarios: i) the transient fades out before the second sample, ii) retransmission has to be requested. The *dotted* line separates the two cases from each other. The transients are highlighted with two bold ovals on top in both cases. On the left side of the line, the two rising transitions on the signal *dec_in_r* indicate the instructions from ADTS to DED to sample the flit. Since the fault has already settled down, the retransmission request flag *re_tx_d* is low when its corresponding request signal goes high. The flag is highlighted with a dashed rectangular box. Similarly, for the latter case, *re_tx_d* remains high since the error has prolonged over the second sample as well. As a result, retransmission proceeds. In this case, the signal *router_d* indicates the consistent faulty flit on the right side of the dotted line.

Table 7.2: Comparison of Different Fault-Tolerance Mechanisms

Mechanism	Transmitter		Receiver		Tx/flit	Fault Coverage
	Area (μm^2)	MCPD (ps)	Area (μm^2)	MCPD (ps)		
SED	231.3	527	255.6	651	1	Single
DED	453.9	579	519.0	715	1	Double
SEC	453.9	579	1288.1	976	1	Single
Time-Red	902.4	1199	1656.6	932	3	Multiple
ADTS	453.9	579	689.9	814 1440	1	Double

7.4.2 Area Overhead Comparison

We synthesized the design for $90nm$ technology. While Table 7.1 shows the post layout device utilization summary for each module that is independent of the coding mechanism, the post layout device utilization summary for each of the coding schemes is presented in Table 7.2. Not surprisingly, SED requires the least overhead. The transmitters for DED, SEC and ADTS-DED are identical and therefore incur the same overhead, but the receiver is significantly simpler for DED, followed by ADTS-DED and then SEC. TRV requires by far the highest overheads.

7.4.3 Performance Penalty

Performance can be measured along several lines: Of course, the delay penalty resulting from the flit having to pass additional logic stages is relevant here. These figures are shown as MCPD in Tables 7.1 and 7.2. Precisely, MCPD corresponds to the latency of the handshake protocol, from input request↑ to input acknowledgment↓. Again SED is the winner, followed by DED, ADTS-DED, and SEC. TRV appears to be much slower due to the high penalty of the transmitter.

Even more important than the path delay, however, is the number of transmissions in the fault free case as well as in case of a fault. For the fault free case the column *Tx/flit* in Table 7.2 gives this number. For a rough estimation it can be multiplied with the MCPD value, which instantly seems to make TRV the slowest option. Note, however, that SEC and TRV do not require retransmission and DRI modules. Also, TRV does not need an input module either, since it already makes use of two latches to store the flits.

Furthermore, recall from Sec. 7.4.1 that there are two single fault scenarios for ADTS-DED: A fault that can be mitigated just by re-sampling and one that requires retransmission. The value *814* under MCPD for the receiver circuit corresponds to the fault free case, where no resampling is needed, and *1440* is the worst case latency corresponding to the resampling case.

7.4.4 Discussion

Considering the above criteria, and the amount of resources available on-chip in state-of-the-art VLSI designs where area utilization is not a major concern any more, the DED and ADTS-DED mechanisms are the most attractive choices: Their fault coverage is better than SED (see

Table 7.2) with only a marginal increase in performance overhead. Their coverage properties even surpass that of TRV which cannot tolerate faults in more than one replica of the same flit.

A brief comparison can be made between DED and ADTS on two aspects: Area utilization, and performance penalty in case of a faulty reception (note that MCPDs for the two are almost identical for a fault-free reception). The DED approach clearly outclasses ADTS as far as the area is concerned. On the other hand, in case of a faulty reception, a complete retransmission is the only way out for DED, which roughly adds an overhead of *4.01 ns* corresponding to the delay $\{DRI + Retransmission + Transmitter + Receiver + Inputmodule\}$, in addition to two long path delays between the two communicating tiles, one for the error flag, and the other for the retransmission of the actual flit. Relative to this figure, ADTS-DED mechanism adds a net overhead delay of *2.87 ns* corresponding to the worst case MCPD of *1.44 ns* and an additional Input Module delay. Only in such circumstances where a retransmission is inevitable, DED proves to be more efficient than the ADTS-DED approach, since the latter must have already made a useless attempt to resample the faulty flit, incurring an overhead of approximately *2.245 ns* corresponding to one resampling latency of *814 ps*, and an Input Module delay of *1.43 ns*. As stated previously, however, the dominant fault class is represented by the transient faults that fade out by themselves, *most likely within one computation of the decoder*; and therefore the number of retransmissions needed must be minimal, making ADTS-DED approach the most optimal choice.

7.5 Summary

Simply implementing a DED mechanism combined with on-demand retransmission for the protection of the communication channels of our ANoC may severely degrade throughput and hence system performance in the presence of frequent transient faults. In this chapter we have described a more efficient error recovery mechanism called ADTS-DED, which does not require retransmission for single faults and hence reduces the performance degradation of the system by a considerable measure. The pitfalls with this mechanism are the hardware overhead and the *worst case* delay overhead (approximately 57.62% more propagation delay than the DED in case of retransmission) which will also increase the power dissipation. Still we feel that in future technology extra transistors will readily be traded for increased reliability, and hence the approach is very useful.

CHAPTER 8

Fault-Tolerant Router: The Complete Design

In the previous chapters we have proposed fault-tolerance concepts for components that we considered most crucial because they are the building blocks of almost every ANoC available in literature, and their performance significantly contributes to the overall performance of the router. In this chapter we present the complete design of the fault-tolerant router built using those components, namely RAMP and FT-TAC. Fig. 8.1 highlights the blocks that are the main contribution of this chapter: We focus on the colored and shaded modules of the circuit. The former already exist and need to be made fault-tolerant, whereas the shaded components are to be introduced to allow compatible interconnection between various components.

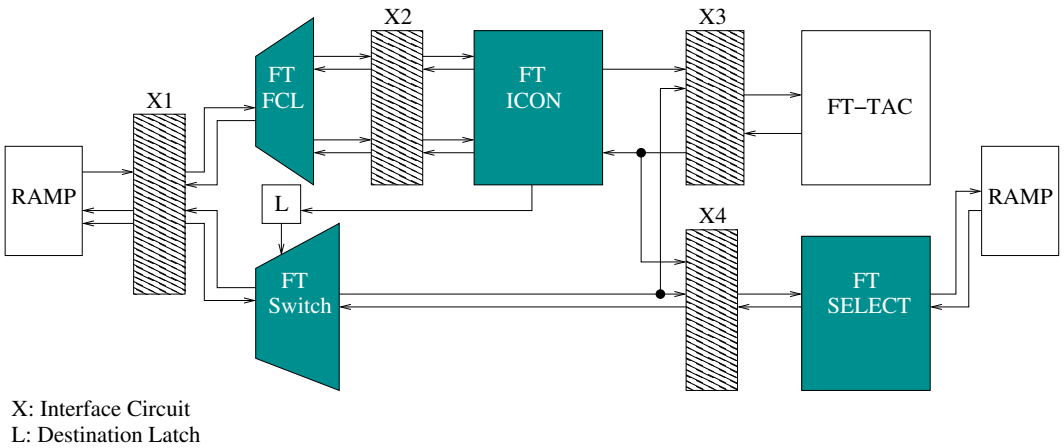


Figure 8.1: Main focus of this chapter

8.1 Preliminaries

Before proceeding with the description of the design and its evaluation, we have to make a few assumptions and simplifications as follows.

8.1.1 Assumptions

A few radiation hardened, and SEU tolerant latches have been proposed, and are already available in standard cell implementations [2, 118]. In our work, we simply assume to have such a latch to store the destination bits used in the crossbar, and therefore, do not intend to take further provisions for protecting the latch contents. However, we do allow SETs on the write-enable signal of the latch (grant from the switch allocator), and propose to tackle those.

8.1.2 Simplifications

Considering the complexity of the entire router, we make a few simplifications regarding the design and functional verification to begin with:

1. Since the IO ports are symmetric, we confine our analysis to just one input - one output pair. We ensure that building the entire router shall simply require replicating the circuit presented here.

2. Because of the huge complexity of the overall design, verifying the entire router by means of model checking is not a viable option; the number of traces grows in millions. As a result, we rely on model checking for individual function blocks, and apply functional verification based on Modelsim simulations for the overall design composed of these pre-verified blocks.

8.1.3 Prior Knowledge

We already possess some prior knowledge from the designs of RAMP and FT-TAC circuits, and we must take that into consideration to avoid any unnecessary logic overhead, and to guarantee the correctness of data exchange. For instance, we already know:

1. *PK1*: that the RAMP and FT-TAC rely on the broad data validity protocol, i.e., the data must remain valid throughout the handshake cycle. The entire router requires the same protocol in place.

2. *PK2:* that each stage of the RAMP inserts a delay element in the control path that is longer than the length of a fault that we wish to tolerate, and the FT-TAC inserts another couple of such delays. This means that the activation of the control signals, specifically the grant from the FT-TAC, is separated from the valid data at least by three times the maximum length of the expected fault. This information is extremely critical in protecting the destination bits from being corrupted, as shall be explained later in the chapter.

3. *PK3:* that the RAMP and FT-TAC circuits are capable of tackling SETs on their inputs, so we do not need duplicated logic to drive these modules.

4. *PK4:* that the outputs of FT-TAC may produce faults as pulses of a limited length, but they will never have their states flipped.

5. *PK5:* that upon a faulty input request ($\uparrow\downarrow$) the RAMP module will issue an early resource request. If this signal propagates to the FT-TAC, it will appear as a valid request, and may acquire the switch incorrectly, which would only be released with a valid tail flit – until then everything else would remain blocked. We need some logic between the RAMP and the ICON to address this problem.

8.1.4 Design Methodology

1. In this chapter, we generally rely on the conventional *duplicate and double check* strategy, i.e., mostly using the DD-gates proposed in [62]. This proven approach does not require a formal proof of correctness.

2. Without altering the design and requirements of the two important circuits, RAMP and FT-TAC (for the latter there is an exception that we explicitly highlight later), we ensure that the rest of the components, which we propose to harden in this chapter, are conveniently integrable with those.

8.2 Hardening the Components

8.2.1 Interface X1

As stated above, *PK5*, the output of the RAMP cannot be immediately used to acquire the FT-TAC, an intermediate circuit is needed to wait for the falling transition, and only then the request may be forwarded to the input handler. However, even such a circuit may acquire the arbiter with a faulty $\downarrow\uparrow$ pulse, the difference in this case is that the previous up transition (which was followed by this fault) must have been a correct request, and the reservation of the FT-TAC, therefore, will not result in indefinite blocking. Fig. 8.2 presents the resulting circuit. Note that now there are two separate requests, one going to the FCL and the other to the switching demux. Since the circuit already has the required redundancy available, it needs no further protection.

8.2.2 Fault Tolerant Input Controller (FT-ICON)

Recall that the request from a header flit acquires an arbiter, and an $\uparrow\downarrow$ pulse on the tail request releases it. If the arbiter is acquired as a result of a faulty header request ($Hr\uparrow$), it may block the operation until a correct flit really arrives and its tail flit releases the earlier acquired arbiter. This may be undesired, but not as devastating as an incorrect tail flit releasing the arbiter ($Tr\downarrow$), leaving behind several orphan flits in the network. Therefore, protecting a tail request is relatively more important than protecting a header flit.

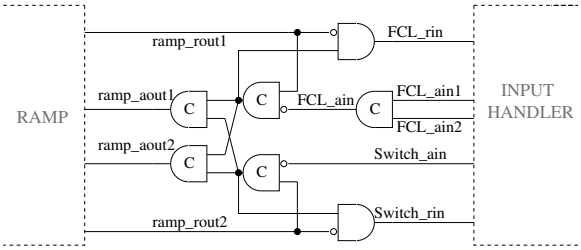


Figure 8.2: Interface Circuit between RAMP and Input Handler

Property	Remarks
A[] not deadlock	No deadlock in any execution path
A[] (u6.wait_s0 && u7.wait_s0) imply (!Tr1 && !Tr2)	At least one Tail request ↓ must be correct
A[] (u6.wait_s1 && u7.wait_s1) imply (Hr && Switch_rin)	Hr or Switch_rin must have a correct ↑

Table 8.1: ICON Verified Properties

An easy solution to protect this simple circuit is duplication. However a careful analysis of its operation reveals that the part of the circuit responsible to handle Hr could rather be protected using a much simpler way. Recall that when the input request of the switching demux (*Switch_rin*, see fig. 8.2) makes a rising transition along with $Hr\uparrow$, it corresponds to a header flit, and in response the output of FT-ICON (*to_arbiter*↑) is produced. The following rising events on *Switch_rin* (corresponding to body and tail flits) do not have any impact on the *to_arbiter* signal since the latter is already set (high). This means that if the event $Hr\uparrow$ happens simultaneously with *Switch_rin*↑, either it really is a correct header request, or it is a fault that will have no impact on the value of the *to_arbiter* signal. Therefore we propose to cross check the *to_arbiter* signal (of fig. 3.3) with *Switch_rin* to make sure that no faults on Hr disturb the output signal.

Note that this scheme cannot work for the other part of the circuit responsible to carry the tail flit request to the output, since the switching demux may have simultaneously received a correct body flit request (*Switch_rin*↑), which would help the faulty tail request ($Tr\uparrow\downarrow$) at ICON bypass the protection logic and subsequently release the FT-TAC. Therefore, duplication of that part of the circuit, including the Tr signal, seems to be the only infallible solution in that scenario. The resulting circuit is depicted in fig. 8.3. Note the difference between the internal loops formed by the output OR-gate (generating Ta) and the AND-gate feeding it in fig. 3.3 and fig. 8.3; the 2-input AND-gate has been replaced with a 3-input one. This ensures that a fault on $Tr1$ does not manifest itself in the loop, thereby generating an incorrect Ta – leading to a deadlock. This deficiency was pointed out by our model checking verification. The final circuit, presented in fig. 8.3, now passes a verification of all the properties given in Table 8.1.

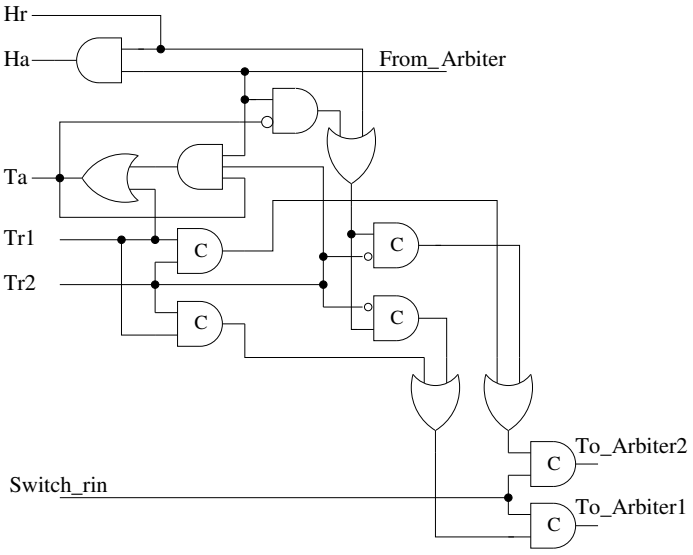


Figure 8.3: FT Input Controller

8.2.3 Fault Tolerant Flit Categorization Logic (FT-FCL)

Since the FCL is a simple demux without any storage element, its state cannot be upset. The only gate that needs to be duplicated is the AND-gate responsible to generate the tail output request as required by the FT-ICON, described above. All other faulty transitions on output requests and input acknowledgments are already mitigated by double checking them with $Switch\_rin$ in FT-ICON and $Switch\_ain$ in X1 respectively. Its schematic is given in fig. 8.4(a).

8.2.4 Interface X2

The X2 interface only comprises a single MC in fig. 3.3, which must be duplicated now to allow the two tail requests from the FCL to reach the FT-ICON.

8.2.5 FT-Switch Demux

Just like the FCL, this demux does not comprise any storage element, however, its outputs are joined with the *to_arbiter* signal from the FT-ICON through an MC before reserving the FT-TAC as was shown in fig. 3.4. It is obvious that a faulty $\uparrow\downarrow$ pulse can conveniently flip the MC, and reserve the arbiter in an incorrect direction. This requires all the gates generating output requests in the switching demux to be duplicated. In addition to this, the two demuxs may be driven by two different input requests, i.e., FCL_rin and $Switch\_rin$. This ensures no single faulty input request can transmit two transient faults on the output requests.

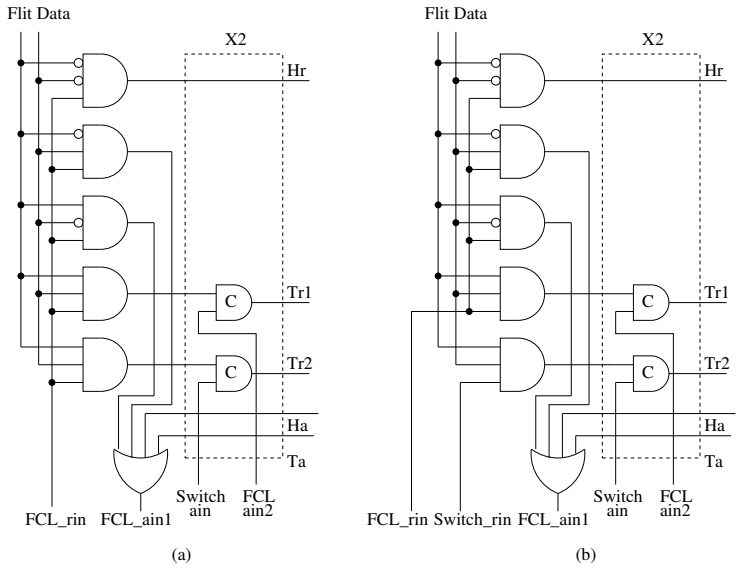


Figure 8.4: Schematics of FT-FCL and X2: (a) Initial design, (b) Passed all the verification tests, Sec. 8.3.2

8.2.6 Interface X4

X4 is the collection of AND-gates within the output generator in fig. 3.4. Since the outputs of the switching demux have been duplicated, all the AND-gates in X4 must also be duplicated, each driven by a different output request of the demux. This further forces the FT-Select module to have duplicated inputs.

8.2.7 Interface X3

This module initially had a single MC that joined the output request of the switching demux with the *to_arbiter* signal. Since the outputs of the demux have been duplicated, this MC must also be duplicated. The resulting two MCs are then joined by another MC, since the FT-TAC does not need duplicated input requests. The schematics of the interfaces X3 and X4 are presented in fig. 8.5.

8.2.8 FT-Select Module

The schematic of FT-Select module is presented in fig. 8.6. Because of duplication it requires two output acknowledgments, which are already provided by the RAMP connected next to it. However, the RAMP requires a single input request, that is why the duplicated outputs of the gates are joined using an MC. Secondly, the input acknowledgments have the potential to flip to the incorrect logic value thereby spoiling the broad data validity protocol required by the RAMP. This is prevented by double checking the input acknowledgments generated by the FT-Select module.

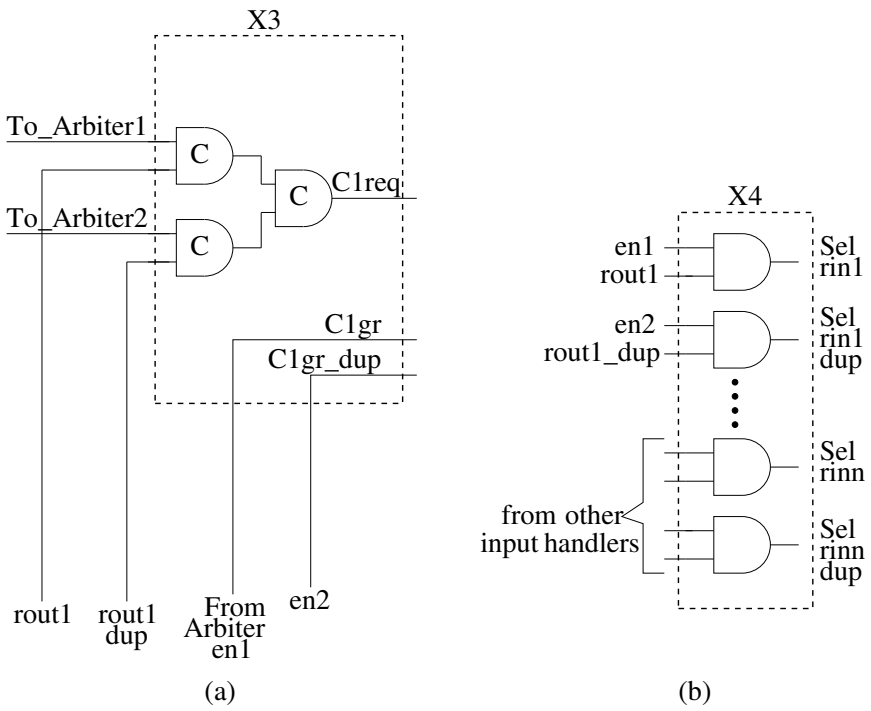


Figure 8.5: Schematics: (a) X3, (b) X4

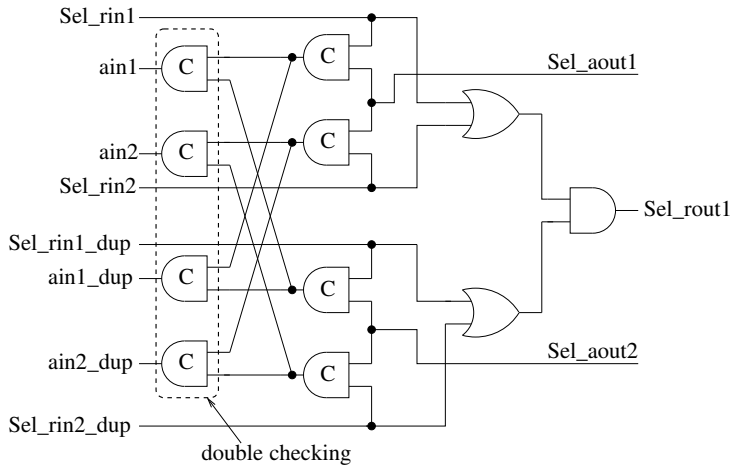


Figure 8.6: FT_Select Module

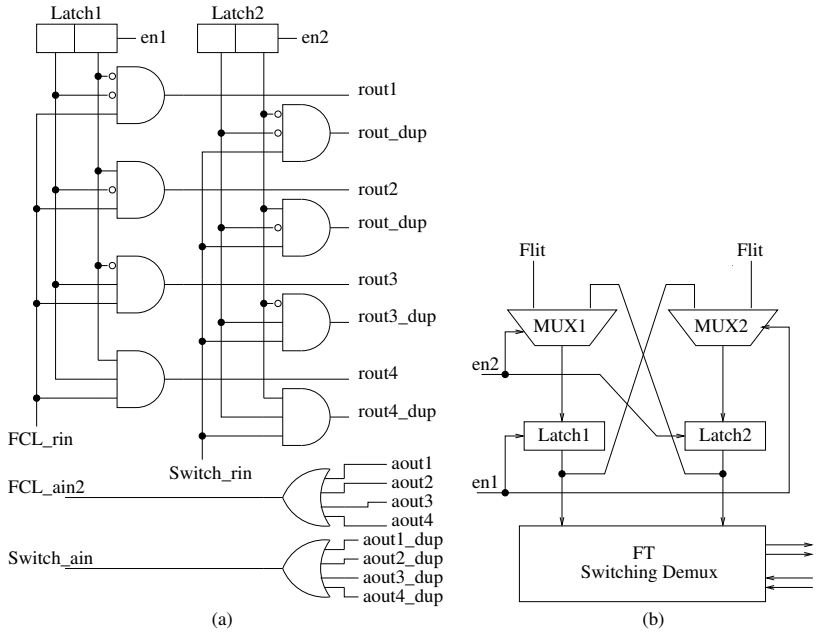


Figure 8.7: FT Switching Demux and FT Latch Enable

8.2.9 FT-Latch Enable Signals

Recall that the selection line of the switching demux comes from the 2-bit SEU hardened latch that does not need further protection. However, if the latch enable signal, which in fact is the *from_arbiter* signal, is faulty, it can replace the correct destination information with any corrupt data available on the flit data bus. Therefore it needs to be duplicated, which forces the output of the FT-TAC, gates *u5.2* and *u6.2* shown in fig. 6.4, to be duplicated in turn. This is one exception we indicated in Sec. 8.1.4(2). Note that this does not require alteration in the FT-ICON, since any one of the two outputs of FT-TAC can still be used.

Using duplicated latch-enable signals to control a single latch will always result in a single point of failure. This can only work with the latch being duplicated as well, and then each enable signal controls one of them. Since all the gates in the switching demux are already duplicated, the output of each latch can control one demux, as shown in fig. 8.7(a).

This situation in particular is more complicated than it looks. Consider one of the enable signal being faulty, which forces the latch it controls to capture some corrupt data. From this point onwards, these bits are not going to recover to their original state, and therefore will result in two different output requests of the demux going high. This will certainly lead to a deadlock at the duplicated MCs where the outputs of the demux are joined with the *to_arbiter* signal in X3. This undesired situation can only be prevented by overriding the corrupted destination bits with the correct bits stored in the other latch. Our proposed protection mechanism is depicted in fig. 8.7(b). We allow the destination bits to enter a latch either directly from the flit on the data bus, or from the other latch through a mux. This mux is controlled by the latch-enable signal of the other latch. This scheme applies to both the latches to ensure we leave no single point of

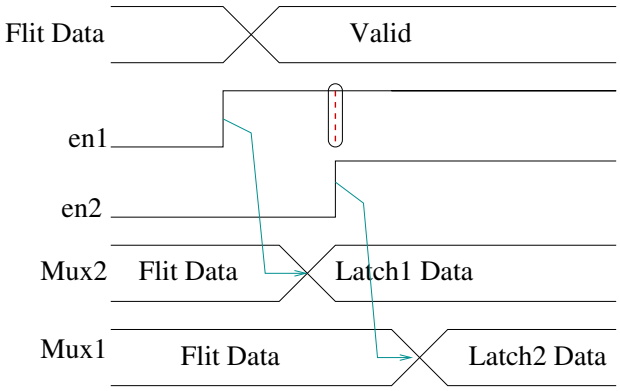


Figure 8.8: A critical fault with FT-Latch enable signals

failure.

A ↓↑ fault simply does not create any problems: 1) The latch that becomes transparent, forwards the correct data from the other latch, 2) The multiplexer that receives the fault at its selection line, and consequently forwards the corrupted data from the flit to its corresponding latch, only finds the latter opaque, and cannot make it store the corrupted destination bits. Therefore one type of fault is always mitigated.

An ↑↓ fault on the other hand is relatively complicated to tackle. Consider the situation depicted in fig. 8.8: A faulty ↑ happens on *en1* when the destination bits available on the flit data are still invalid, and the fault does not settle down until the flit becomes valid, and the other *en2* makes its valid transition. It may be possible that *en1* completely overlaps with *en2* and simply does not go back to low. Since *en1* has already made its corresponding latch opaque with the faulty rising tansition, its contents are guaranteed to be invalid. At the same time Latch2 never saw the valid destination bits since MUX2 was already forwarding the corrupted data from Latch1. This way both the latches may store invalid values, thereby leading to a deadlock or flit misrouting.

This particular problem requires resorting to some timing assumption. Recall *PK2* that valid data and the correct enable signals are separated by at least three times of an expected fault length, which means any fault that happens when the data were still invalid, would already be settled down at the time the other enable signal has made its legal transition. And we know that the 4-phase bundled data protocol already provides a sufficient safety margin, which makes the delay between the data and the enable signals conveniently longer than our requirement. In short, with our timing assumptions, the above sketched scenario should never happen, and the proposed mechanism will work perfectly under other circumstances. The results of our fault injection experiments using Modelsim scripts shall be given in the next section.

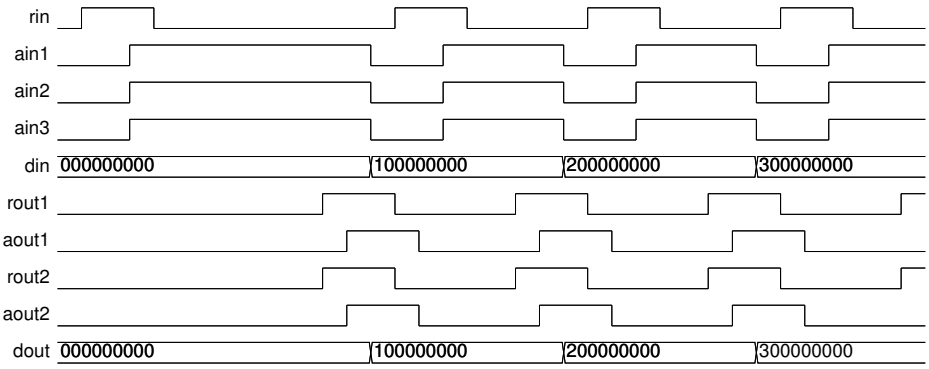


Figure 8.9: Operation of the router in a fault-free environment

8.3 Fault Injection and Simulation Results

8.3.1 Fault-free Operation

For functional verification of our router, we described the system shown in fig. 8.1 in Verilog HDL. In our testbenches we joined the three input acknowledgments, *ain1*, *ain2*, and *ain3* (of the RAMP – input buffer) using a three input MC, and inverted its output to automatically generate new input requests *rin*. Similarly the output requests, *rout1*, and *rout2* (of the output buffer RAMP) were joined using duplicated MCs to automatically generate the output acknowledgments *aout1*, and *aout2*. Each packet comprises four flits. For simplicity we have kept the payload constant, however, the flit categorization bits already distinguish the flits from each other ($header \rightarrow 00, body \rightarrow 01, 10, tail \rightarrow 11$). Fig. 8.9 presents the simulation waveform in a fault-free environment. Note that the delays between the second, third, and fourth handshake cycles are significantly smaller than the delay between the first two cycles – the header flit reserves the FT-TAC, which is a relatively slower component. Beyond giving evidence that the overall design actually works, this waveform clearly depicts the correct protocol, both on the input and the output sides. In the following experiments when we randomly inject faults (of limited lengths) on various signals, the same protocol must be observed to guarantee successful mitigation of faults. In case an SET manages to spoil this protocol, we have to propose appropriate modifications in the overall circuit.

8.3.2 Fault Injection and Verification

We have carefully selected a subset of signals in the entire router to verify its fault-tolerance capability by injecting a series of transient faults on those. Although a thorough verification would require all the signals to be tested, it is not viable to present the detailed fault injection process and analysis of the results, as mentioned already. To make a selection of the signals to be tested, we ignore the internal nodes of the components that we have already model checked, and emphasize on the new components presented in this chapter. In the following we present a few simulation results, in each of which we have injected a series of 100,000 transient faults (one per handshake cycle) on the selected signals. We carefully observe the outputs of the router

to ensure with high confidence that all of the faults are successfully mitigated before reaching any of the outputs. Precisely, we expect our proposed extensions to guarantee three properties:

1. ***Deadlock*:** No fault must lead to a deadlock in any execution path.
2. ***MF*:** No fault must result in a flit being misrouted. By keeping the payload constant, we ensure that all the flits are directed to the same FT-TAC, forcing all the flits to go to the same output port of the switching demux, *rout1*, in fig. 8.7. In case at any point in time, *rout2* goes high, it would indicate a misrouted flit.
3. ***TI/TL*:** Neither an incorrect flit must be inserted, nor a correct one be lost. This can be easily detected by comparing the currently received flit on the output port with the last received flit: The flit identification bits of the current are always supposed to be one more than the last received flit. If this is not true, then either a flit has been lost, or a superfluous token has been incorrectly inserted.

Table 8.2 presents the list of nodes where the faults were injected, and the properties that they satisfied. Except for a single case, all of the tests initially satisfied all the properties without requiring modification in the design. One exception was the fault applied on the input request of FCL, FCL_rin in fig. 8.4(a), in which the same faulty signal was supposed to generate both the tail requests. By chance, the falling transition of the fault on the tail requests coincided with the $Switch\_ain \downarrow$ for a body flit, and prematurely released the FT-TAC, which resulted in a deadlock. This fault was mitigated by generating original tail request with FCL_rin, and its duplicated version with $Switch\_rin$ as shown in fig. 8.4(b).

8.3.3 Simulation Results

In the following we present few of the critical fault cases that we simulated:

- The first simulation, fig. 8.10 corresponds to the faults injected on the input signal, *rin*, of the RAMP (input buffer). This test is critical since a single input fault may diverge on both the output requests and cause the entire circuit to malfunction. The faults have been highlighted using colored ovals. It may be clearly observed that no fault has managed to reach any of the output signals.

In the following figures, we have omitted the data bus from the waveform.

- Fig. 8.11 presents the case when faults were injected on one of the outputs of RAMP, i.e., an input of the interface X1. Once again, due to duplicated request signals, no fault has been able to reach the output.
- Fig. 8.12 corresponds to the faults injected on one of the input requests of the FCL. Note that this experiment was conducted with duplicated input requests, each of which is responsible to generate a different tail request. The router now satisfies all the tested properties against this fault.

8.3.4 Discussion

As Table 8.2 indicates, the final design can withstand all injected faults. In contrast, the non-FT version of the router failed in all the test runs. We synthesized both the non FT and FT

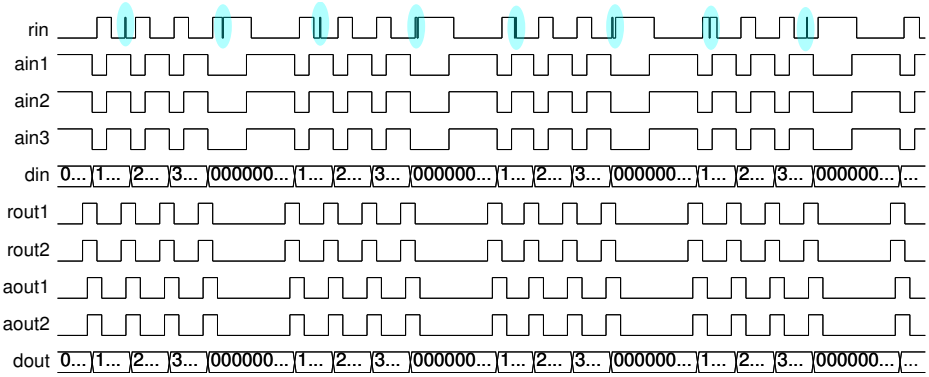


Figure 8.10: Operation of the Router in presence of faults on input request

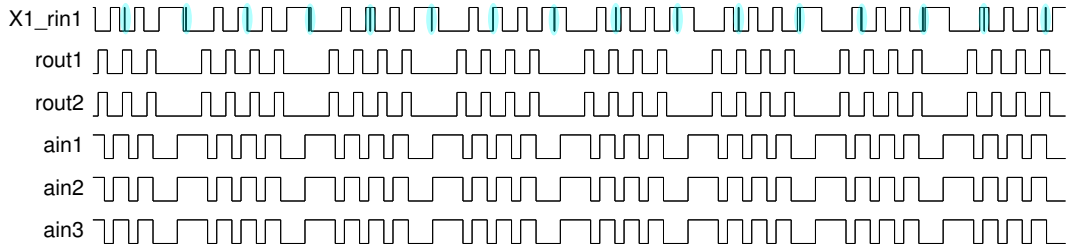


Figure 8.11: Faults applied on input request of Interface X1

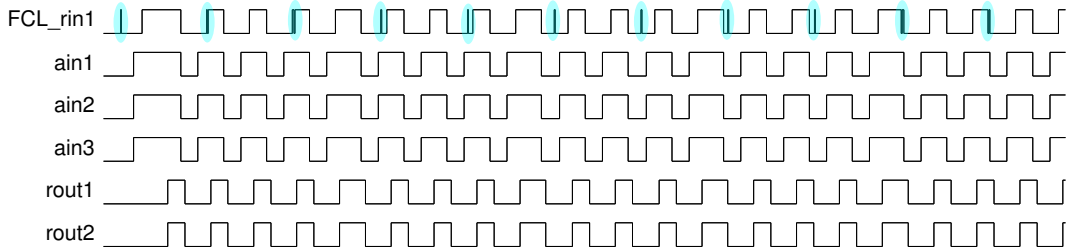


Figure 8.12: Faults applied on input request of FCL

Table 8.2: Verified Properties

Faulty Signal	FT Version			Non FT Version		
	MF	TI / TL	Deadlock	MF	TI / TL	Deadlock
RAMP *rin*	√	√	√	–	–	–
RAMP *rout1*	√	√	√	–	–	–
FCL *rin*	√	√	√*	√	X	X
Switch *rin*	√	√	√	√	X	X
To_arbiter1	√	√	√	X	X	X
From_arbiter1	√	√	√	X	X	√
Sel_$rout$	√	√	√	√	X	√
Sel_$aout$	√	√	√	X	X	√
FCL *ain1*	√	√	√	√	X	√
Switch *ain*	√	√	√	√	X	√

MF: Misrouted Flit
TI / TL: Token Inserted/Lost
*Failed initially, but satisfied when duplicated input requests were used.

routers for 90*nm* technology using Synopsis. The FT version obviously costs more in area utilization, $2250.1\mu m^2$, in comparison to $1208.9\mu m^2$ of the non FT version. The performance has been degraded significantly as well: $31MHz$ in comparison to $644MHz$. However, the reason behind this restricted throughput of the FT version is the small number of flits per packet. As we have argued already in chapter 6, the greater the number of flits per packet, the better the throughput would be. The primary objective of this work was to conduct a thorough fault analysis of an asynchronous router, and to propose ways to mitigate their effects. Of course, a few conventional techniques, such as pipelining, may be employed internally to increase the overall router's performance. This is something we plan to investigate in near future.

8.3.5 Summary

We have proposed an FT router, based on RAMP and FT-TAC circuits previously proposed. In this chapter our focus has been on hardening the components that bind everything together, and we have ensured a safe and FT integration of those. We have verified a few properties by injecting a large number of transient faults on several nodes within the circuit using Modelsim simulations. To make a fair comparison, we injected the same number of faults on the non FT version of the router, and not surprisingly, most of the properties failed. On the other hand, our proposed extensions have degraded the performance of the router, and have caused some overhead in terms of area utilization. While in the future technology extra transistors will be readily traded for increased reliability, making the area overhead a less significant issue, we plan to improve the performance of the FT router by investigating conventional pipelining mechanisms in future.

CHAPTER 9

Conclusion and Prospective Directions

9.1 Overview of Research Contributions

Networks-on-Chip (NoCs) have proven to be an efficient replacement of the shared bus architecture, addressing the scalability limitations of the latter for the future multiprocessor platforms. However, tolerance against transient faults and delay variations emerge as crucial challenges for future CMOS logic cells, and this applies to NoCs as well. In this work we have contributed to making design of NoCs fit for these future challenges by;

1. Elaborating an efficient fault-tolerant implementation for its most fundamental components.

2. Adopting asynchronous implementation to cope with PVT variations.

3. Proposing a framework for systematic check of a protocol implementation with respect to potential blocking behavior to support Quality of Service (QoS).

In the following we summarize the contributions of this research study.

9.1.1 VC Access Control Framework

We have used the notations of signal transition graphs and state graphs as presented in [142] to introduce a systematic treatment of some fundamental properties of handshake controllers for NoCs with virtual channels. In particular we have formulated requirements for safe data transmission (with an option for bufferless implementation) and decoupling of sender and receiver. Furthermore, we have expressed the role of flow control within this framework and analyzed two basic schemes from the literature in this context. This allowed us to apply our rules for a systematic analysis of handshake controllers, and indeed we were able to pinpoint some deficiencies concerning their nonblocking property that is crucial for Quality of Service. At the example of

the credit-uncredit scheme we illustrated how the exact identification of a weakness can be used for a systematic improvement of the controller.

Our framework also allowed us to come up with a novel flow control scheme. Starting from the observation that the messages conveyed over the credit channel are usually just data-less synchronization tokens, we proposed to give those credit tokens a higher significance by associating each with M credits instead of a single one, as usual. We have investigated the mutual relations between number of pipeline stages in the upstream and downstream link, number N of tokens in the loop, number of buffers in the receiver, and size M of those credit bundles. In our implementation example we have shown the choice of $N = 3$ credits along with bundles of $M = 2$ to be very efficient, in conjunction with a 2-phase protocol on the credit channel. Furthermore, we have been able to classify some solutions from the literature as specific instantiations of our generic scheme (namely with $M = 1$), and we have assessed the properties of our MCFC scheme in comparison with those baseline solutions. The bottom-line is that MCFC is in line with most of those schemes with respect to area and performance, while reducing the transmission rate on the upstream channel by a factor of 8, thus substantially saving dynamic power on that global interconnect, and at the same time allowing to choose a slower, more power efficient implementation.

Relevant Publications

1. S. R. Naqvi, R. Najvirt, and A. Steininger. A Multi-Credit Flow Control scheme for asynchronous NoCs. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2013 IEEE 16th International Symposium on*, pages 153-158, 2013.

2. R. Najvirt, S. R. Naqvi, and A. Steininger. Classifying Virtual Channel Access Control Schemes for Asynchronous NoCs. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 115-123, 2013.

3. S. R. Naqvi. AÁRAF: An Asynchronous Router Architecture using Four-Phase Bundled Handshake Protocol. In *Computing in the Global Information Technology (ICCGI), 2012 The Seventh International Multi-Conference on*, pages 200-205, 2012.

9.1.2 Robust and Efficient Resource Sharing Mechanisms

We have proposed a novel tree arbiter cell that allows a pipelined processing of requests, i.e. arbitrating for the next request while the current one is still ongoing. The extra logic required for this feature initially increases the arbiter delay; however, in the relevant case of frequent requests from different clients our scheme yields a considerable speed-up. We have introduced an inter-TAC communication path for cascaded use of our TAC cell that not only enforces exclusive activation of a single grant at a time, but also improves the fairness of the arbitration process. Our simulation results clearly indicate that in case of a four client router in an all-eager environment, our scheme provides superior performance; we gained a speed-up of 61.28%, 69.24%, and 186.85% as compared to three different designs from literature.

Given that an arbiter, due to its inherently non-deterministic operation, cannot be appropriately protected by simple replication and voting on the module level, we have enhanced an

existing implementation of a TAC to make it fault tolerant on circuit level. To this end we have carefully analyzed the failure modes of all involved components and introduced protection at selected locations. Our protection schemes comprise pulse filtering, signal interlocking, and duplication of gates. In this way we have tried to establish seamless protection with minimum overheads. We have verified the tolerance against all single transient faults within the proposed TAC cell as well as at the interfaces by means of model checking. An assessment of area and performance has shown that the overheads are considerable, but still lower than those of a TMR-based solution that, however, cannot provide full coverage.

Relevant Publications

1. S. R. Naqvi, and A. Steininger. A TAC for High Speed Resource Sharing. submitted to *Design, Automation, and Test in Europe (DATE)*, 2014.

2. S. R. Naqvi, A. Steininger, and J. Lechner. An SET Tolerant Tree Arbiter Cell. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 31-39, 2013.

9.1.3 Transient Fault Tolerant Channels and Input Buffers

We have motivated the need for protecting the communication channels and the input buffers in an asynchronous router against transient faults. For the very popular case of a Muller pipeline in a bundled data architecture we have analyzed the potential fault scenarios and, based on this analysis, systematically developed an extension to attain fault tolerance. Unlike mere duplication approaches, our design exploits the redundancy immanent to the 4-phase protocol for mitigating transient faults, thereby eliminating all potential single points of failure. Once again by means of model checking we have proven that our solution can withstand all transient faults in the control logic, including the proposed extension itself.

As far as protection of the channels is concerned, simply implementing a DED mechanism combined with on-demand retransmission for this purpose severely degrades throughput and hence system performance in the presence of frequent transient faults. Therefore we came up with a more efficient error recovery mechanism namely ADTS-DED, which does not require retransmission for single faults and hence reduces the performance degradation of the system by a considerable measure.

All of our proposed solutions are conveniently integrated to form an efficient fully asynchronous routing node that is capable of mitigating a single transient fault in the control path, and two faults on the global communication channel per flit transmission.

Relevant Publications

1. S. R. Naqvi, J. Lechner, and A. Steininger. Protection of Muller-Pipelines from Transient Faults. To appear in *Quality Electronics Design (ISQED), 2014 International Symposium on*, 2014.

2. S. R. Naqvi, V. S. Veeravalli, and A. Steininger. Protecting an Asynchronous NoC against Transient Channel Faults. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 264-271, 2012.

9.2 Prospective Directions

In what follows, we briefly state a few open issues concerning the modeling, functional verification, and synthesis of asynchronous circuits. All of these are in fact based on our observations during this research work. Each of the following forms a separate area of research, addressing a few of which is our primary objective, and the rest we simply point out.

9.2.1 Limitation of Model Checking

Although being bullet proof for the properties it verifies, the model checking cannot guarantee coverage of all possible functional behaviors of the circuit under test. In other words, it checks whatever it is asked to. As a result, it is eventually up to the designer to provide all properties that are essential for a thorough functional verification of the circuit. Linking these properties with the specification in a rigorous way is certainly an essential next step, not only for this work, but for the verification community in general.

9.2.2 Multiple Fault Tolerance

The rapidly increasing soft error rates (SER) in digital circuits suggest that it shall be soon when we witness multiple errors during a handshake cycle, and this concerns both the data path and the control path. While the existing methodologies for the former, such as *Adaptive Delayed Twice Sampling with Double Error Detection* (ADTS-DED), proposed in this work, shall still be applicable, it is almost certain that our contribution in this work of mitigating a single fault per handshake cycle in the control path, is not going to hold valid for too long. As a result, there shall be a need to extend our design to tolerate multiple (two faults to begin with for instance) faults.

9.2.3 Fault Tolerance: A Quality of Service Metric

The constant rise in SER motivates the fault-tolerance capability to be considered a QoS metric. To the best of our knowledge, there is no single work that promised the guaranteed services and fault-tolerance together in one design. We believe this specific area deserves a thorough investigation, and a possible starting point could be proposing a fault-tolerant Static Priority Arbiter. Also in the context of QoS, the use of virtual channels (VC) is common albeit their overhead on area and energy. We have commented on the same in our work as well, however, we believe that a quantitative analysis of their optimal number per routing node against area and energy efficiency, and their application to provide fault-tolerance capability, may prove beneficial for the future many core systems.

9.2.4 Modeling Fault-Tolerance Behavior

During our research we have realized that the modeling and synthesis tools available for asynchronous circuits do not provide much flexibility to the designer. As we have indicated in chapter 5, whenever we attempted to add redundancy in a signal transition graph (STG), *Petrify* would always optimize the redundant arcs during synthesis. We believe, the behavior of a circuit in response to a single event transient (SET) must be given a logical representation while modeling the circuit using an STG. In simple words, the designer must ensure that the STG he has built considers each and every possible transition, including the ones resulting from an SET. This way every circuit, by default, would be able to mitigate the input and local faults. However, the existing tools restrict the designers from building such STGs, and there is a need for a platform that accommodates such behavior.

9.2.5 Design for Testability

Despite their vastly acknowledged potential, the asynchronous circuits and systems struggle to find a place in the electronics industry. It has been reported several times already that the primary reason behind this is the challenge of designing them for testability, which does not follow the conventional approaches, and especialy due to the lack of quality asynchronous EDA tools. Our research group is particularly interested in automatic synthesis of asynchronous circuits. We believe, together with modeling of the fault-tolerance behavior, the automatic synthesis and designing for testability forms a truly prospective direction of research.

APPENDIX A

UPPAAL Models

The models presented here are extracted from the joint work conducted at our department [91, 92], and correspond to chapters 5 and 6. We are going to describe only a few elementary models here; the remaining may be easily understood since all of them are built in the same way.

A.1 NOT Gate

Fig.A.1 presents the model of an inverter. Starting from the initial state upon system reset *rstComb*, depending upon the initial value the control may jump to one of the two possible states *s0* or *s1* respectively corresponding to the output logic levels low and high. At the same time, setting the output variable *y* accordingly. The control then waits in the current state until an event on the input *a* happens, and subsequently jumps to one of the waiting states *w0* and *w1*. The control must stay in the waiting state for a certain duration (minimum delay), depicted by *N_delay*, before settling into the other logic state. If the input variable makes another transition before *N_delay* has elapsed, the control returns to the previous state – this nicely emulates the inertial delay of the inverter. Once the inertial delay has elapsed, the control ignores all further input transitions, waits for the maximum delay, *X_delay* (depicting the switching time of the gate), to elapse, generates the synchronizing event, *y_c*! (which notifies an event to any other component it is connected to), and moves into the opposite state. At this point, the control once again starts checking the input signal.

A.2 AND Gate

In principle, the AND gate, shown in fig. A.2, is modeled exactly like the inverter. However, since there are two inputs, and a transition on any one of them may alter the output, events on both the channels (a_c?, b_c?) must be detected, and the condition, a && b or !(a && b), must be checked before entering the wait state *w1* or *w0* respectively.

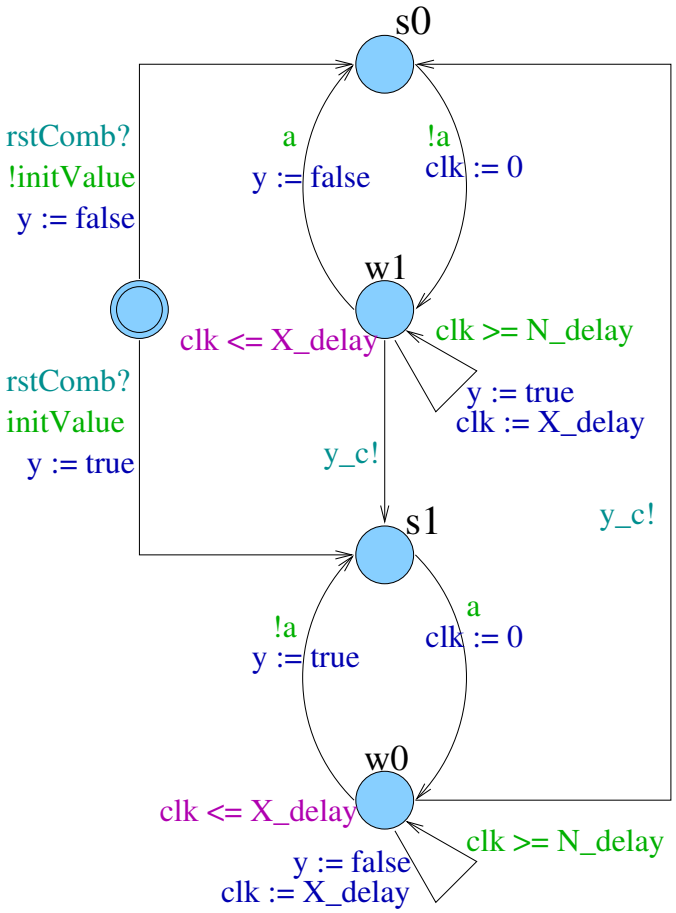


Figure A.1: Model of a NOT Gate

A.3 SET Injector

The fault injector, shown in fig. A.3, comprises two loops: one (*normal*) that simply forwards the input to the output whenever an event occurs on the input, the second loop (*SET*) is executed only if a fault is *enabled* at an arbitrary point of time. During its execution, this loop inverts the input value, keeps the control in the faulty state, *SET*, for the duration equivalent to the maximum length of the fault *fault_M*, and then returns the control to the correct state.

A.4 Muller C-element

The model for an MC, fig. A.4 is almost identical to the AND-gate, except for the guard to leave the state *s1*, and its reset signal is synchronized with the clock, *rstSeq*. Just like any other combinational gate, an MC may also be connected to the SET injector, which could randomly upset the output of the MC for a limited duration. However the SET injector does not model the

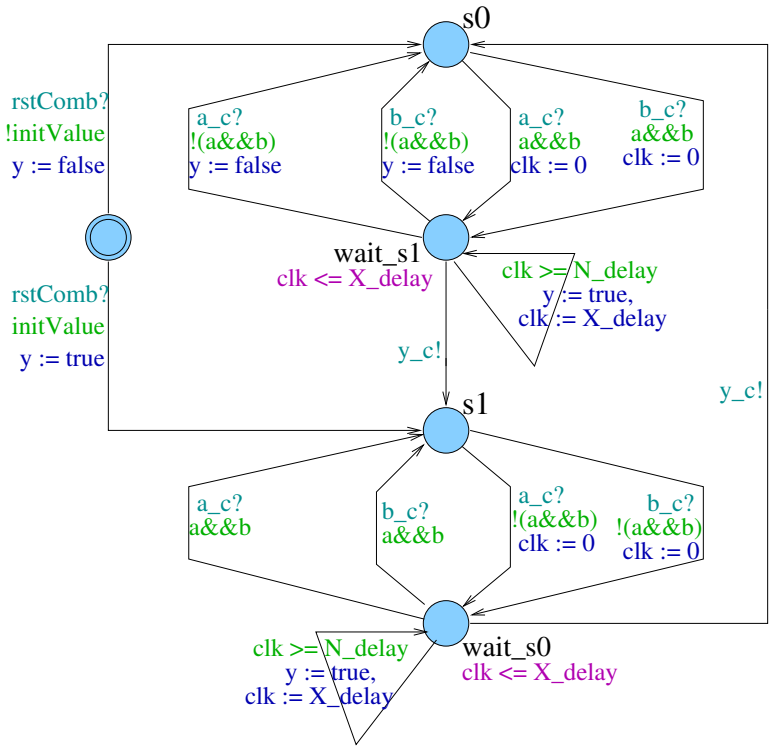


Figure A.2: Model of an AND Gate

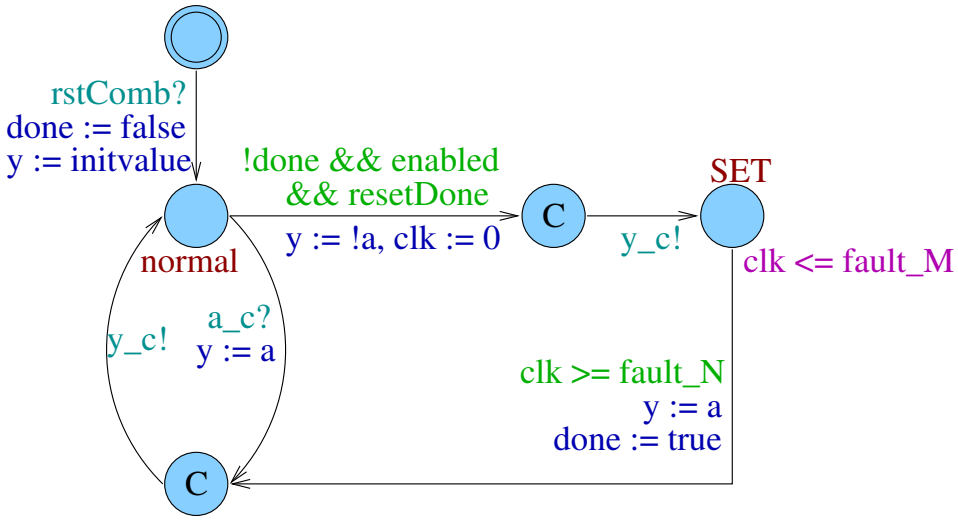


Figure A.3: Model of an SET injector

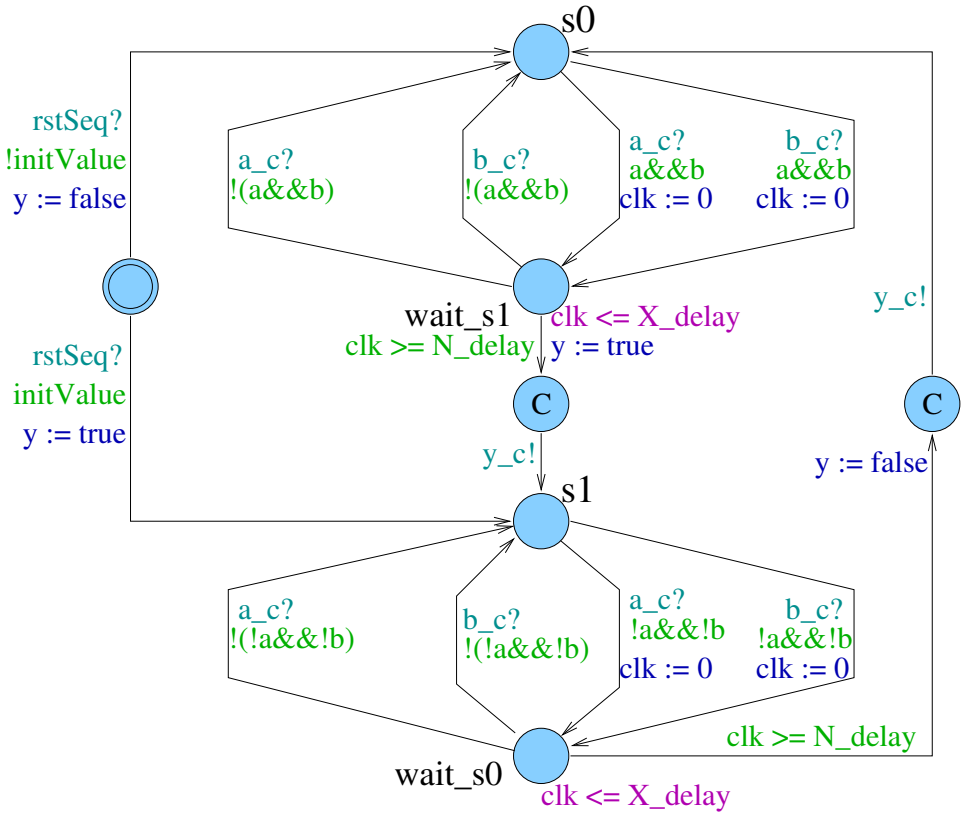


Figure A.4: Model of an MC

bit-flip faults in which case the state of the MC remains inverted until one of the inputs makes a transition. We incorporated this feature into the model of the MC as described next.

The two highlighted edges in fig. A.5 depict how we have modeled the MC's ability to flip its state in response to an SET. Whenever the inputs are not at the same logic level, and an SET is *enabled*, the control may leave the stable state and directly enter the committed state from where it cannot return to its original state. That is why we introduced a *committed* state in the state holding elements, and not the other combinational gates. Note that we keep a fault *count* variable since we have alwas assumed a single fault scenario. Thus an MC flips its state if the fault count is currently zero.

A.5 MUTEX

The MUTEX is modeled as an entirely combinational circuit. Our model makes sure that the two grants are not activated simultaneously, however, all the faults presented in Table. 6.1 are incorporated: The SET injector modules, which may be placed at the two output grants, make sure that all the transient faults are covered, whereas, the faulty inputs to the MUTEX always snatch the grant from one client and give it to the other, thereby covering the bit-flip cases as

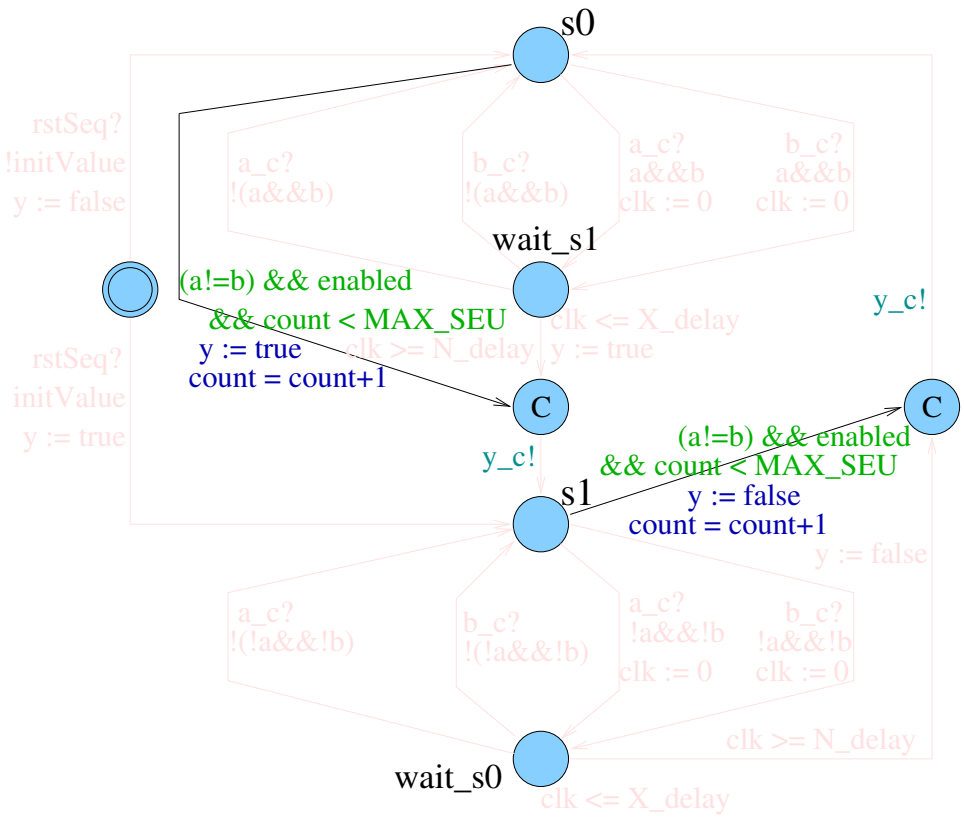


Figure A.5: Model of an MC with bit-flip Fault Support

well. In this case, we allow two faults simultaneously.

All other gates are modeled similarly, therefore we are ommitting their details from the thesis.

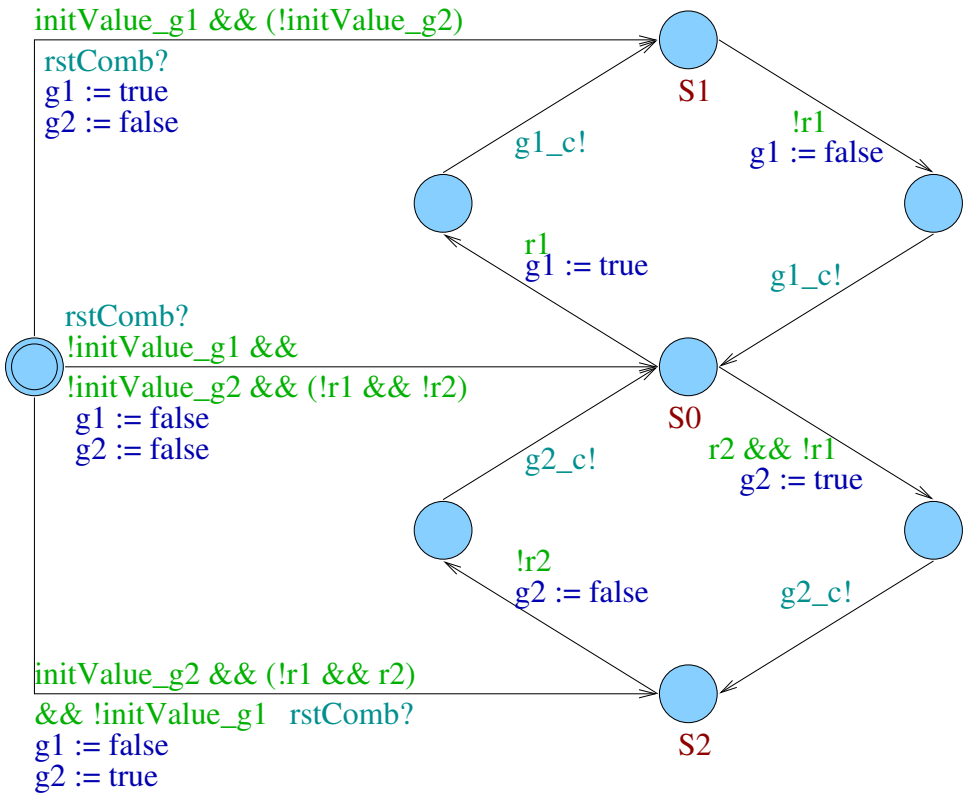


Figure A.6: Model of a MUTEX

Bibliography

[1] A. Agarwal, C. Iskander, and R. Shankar. Survey of NoC Architectures and Contributions. *Engineering, Computing and Architecture*, 3(1), 2009.

[2] O.A. Amusan, A.L. Steinberg, A. F. Witulski, B.L. Bhuva, J.D. Black, M. P. Baze, and L.W. Massengill. Single event upsets in a 130 nm hardened latch design due to charge sharing. In *Reliability physics symposium, 2007. proceedings. 45th annual. ieee international*, pages 306–311, 2007.

[3] Hagit Attiya and Jennifer Welch. *Distributed Computing*. John Wiley & Sons, 2nd edition, 2004.

[4] J. Bainbridge and S. Furber. Chain: a delay-insensitive chip area interconnect. *Micro, IEEE*, 22(5):16–23, 2002.

[5] John Bainbridge and Sean Salisbury. Hardening of self-timed circuits against glitches. Patent Application, 03 2009. WO 2009/030997 A3R4.

[6] William John Bainbridge and University of Manchester. School of Computer Science. *Asynchronous System on Chip Inteconnect*. University of Manchester, 2000.

[7] A. Bardsley and University of Manchester. Dept. of Computer Science. *Balsa: an Asynchronous Circuit Synthesis System*. University of Manchester, 1998.

[8] R. P. Bastos. *Transient-Fault Robust Systems Exploiting Quasi-Delay Insensitive Asynchronous Circuits*. University of Grenoble and TIMA Lab, France, 2010.

[9] E. Baydal, P. López, and J. Duato. Avoiding network congestion with local information. In HansP. Zima, Kazuki Joe, Mitsuhisa Sato, Yoshiki Seo, and Masaaki Shimasaki, editors, *High Performance Computing*, volume 2327 of *Lecture Notes in Computer Science*, pages 35–48. Springer Berlin Heidelberg, 2002.

[10] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. An asynchronous NOC architecture providing low latency service and its multi-level design framework. In *Proc. 11th IEEE Symposium on Asynchronous Circuits and Systems (ASYNC 2005)*, pages 54 – 63.

[11] E. Beigne and P. Vivet. Design of on-chip and off-chip interfaces for a gals noc architecture. In *Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, ASYNC '06, pages 172–, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Luca Benini. Networks on chip: a new paradigm for systems on chip design. In *In Proceedings of Conference on Design, Automation and Test in Europe*, pages 418–419. IEEE Computer Society, 2002.

[13] G. Birtwistle and K.S. Stevens. The family of 4-phase latch protocols. In *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, pages 71 –82, april 2008.

[14] T. Bjerregaard and Sparso. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1226 – 1231 Vol. 2, march 2005.

[15] T. Bjerregaard and J. Sparso. Virtual channel designs for guaranteeing bandwidth in asynchronous network-on-chip. In *Norchip Conference, 2004. Proceedings*, pages 269 – 272, nov. 2004.

[16] T. Bjerregaard and J. Sparso. Scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, pages 34 – 43, march 2005.

[17] R.V. Boppana and S. Chalasani. Fault-tolerant routing with non-adaptive wormhole algorithms in mesh networks. In *Supercomputing '94., Proceedings*, pages 693–702, 1994.

[18] C. Brej, J. Garside, and University of Manchester. School of Computer Science. *Early Output Logic and Anti-tokens*. University of Manchester, 2005.

[19] W.P. Burleson, M. Ciesielski, F. Klass, and Wentai Liu. Wave-pipelining: a tutorial and research survey. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 6(3):464–474, 1998.

[20] Suresh Chalasani. Adaptive fault-tolerant wormhole routing algorithms with low virtual channel requirements. In *In Proc. Int'l Symp. on Parallel Architectures, Algorithms and Networks*, pages 214–221, 1994.

[21] Kok-Leong Chang, J.S. Chang, Bah-Hwee Gwee, and Kwen-Siong Chong. Synchronous-logic and asynchronous-logic 8051 microcontroller cores for realizing the internet of things: A comparative study on dynamic voltage scaling and variation effects. *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, 3(1):23–34, 2013.

[22] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Standford University, 1984.

[23] Fu-Chiung Cheng and Shuen-Long Ho. Efficient systematic error-correcting codes for semi-delay-insensitive data transmission. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 24 –29, 2001.

[24] Ge-Ming Chiu. The odd-even turn model for adaptive routing. *IEEE Trans. Parallel Distrib. Syst.*, 11(7):729–738, July 2000.

[25] Wesley A. Clark. Macromodular computer systems. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 335–336, New York, NY, USA, 1967. ACM.

[26] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mw noc-based digital baseband for mimo 4g sdr. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 278–279, 2010.

[27] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. Bulletproof: a defect-tolerant cmp switch architecture. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 5 – 16, feb. 2006.

[28] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers, 1996.

[29] D. K. Pradhan. *Fault-Tolerant Computing: Theory and Techniques*, volume II. Prentice Hall, Englewood Cliffs, New Jersey, 2003.

[30] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[31] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689, 2001.

[32] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical report, THE ENCYCLOPEDIA OF COMPUTER SCIENCE AND TECHNOLOGY, 1997.

[33] Mark E. Dean, Ted E. Williams, and David L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (ledr). In *Proceedings of the 1991 University of California/Santa Cruz conference on Advanced research in VLSI*, pages 55–70, Cambridge, MA, USA, 1991. MIT Press.

[34] A. DeOrio, D. Fick, V. Bertacco, D Sylvester, D Blaauw, Jin Hu, and G. Chen. A reliable routing architecture and algorithm for nocs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(5):726–739, 2012.

[35] Giorgos Dimitrakopoulos and Emmanouil Kalligeros. Low-cost fault-tolerant switch allocator for network-on-chip routers. In *Proceedings of the 2012 Interconnection Network Architecture: On-Chip, Multi-Chip Workshop*, INA-OCMC '12, pages 25–28, New York, NY, USA, 2012. ACM.

[36] R. Dobkin, R. Ginosar, and I. Cidon. Qnoc asynchronous router with dynamic virtual channel allocation. In *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, page 218, may 2007.

[37] Rostislav (Reuven) Dobkin. Credit-based communication in nocs. Lecture on introduction to networks on chips: Vlsi aspects, Technion, 2007.

[38] Rostislav (Reuven) Dobkin, Ran Ginosar, and Avinoam Kolodny. Qnoc asynchronous router. *Integr. VLSI J.*, 42(2):103–115, February 2009.

[39] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[40] A. Dutta and N.A. Touba. Reliable network-on-chip using a low cost unequal error protection code. In *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on*, pages 3 –11, sept. 2007.

[41] Karl M. Fant and Scott A. Brandt. Null convention logic. Technical report, Available online, http://www.theseusresearch.com/Downloads/NCL.PDF, 1997.

[42] F. Feliciian and S.B. Furber. An asynchronous on-chip network router with quality-of-service (qos) support. In *SOC Conference, 2004. Proceedings. IEEE International*, pages 274 – 277, sept. 2004.

[43] T. Felicijan, J. Bainbridge, and S. Furber. An asynchronous low latency arbiter for quality of service (qos) applications. In *Microelectronics, 2003. ICM 2003. Proceedings of the 15th International Conference on*, pages 123 – 126, dec. 2003.

[44] Jose Flich and Davide Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall/CRC, 2010.

[45] Arthur Pereira Frantz, Maico Cassel, Fernanda Lima Kastensmidt, Érika Cota, and Luigi Carro. Crosstalk- and seu-aware networks on chips. *IEEE Des. Test*, 24(4):340–350, July 2007.

[46] Arthur Pereira Frantz, Fernanda Lima Kastensmidt, Luigi Carro, and Erika Cota. Dependable network-on-chip router able to simultaneously tolerate soft errors and crosstalk. In *Test Conference, 2006. ITC '06. IEEE International*, pages 1 –9, oct. 2006.

[47] Robert M. Fuhrer, Steven M. Nowick, Michael Theobald, Niraj K. Jha, Bill Lin, and Luis Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines, 1999.

[48] S.B. Furber, D.A. Edwards, and J.D. Garside. Amulet3: a 100 mips asynchronous embedded processor. In *Computer Design, 2000. Proceedings. 2000 International Conference on*, pages 329–334, 2000.

[49] Lake Hiawatha NJ Gar Moy. Majority vote circuit, 11 1984.

[50] Alberto Ghiribaldi, Davide Bertozzi, and Steven M. Nowick. A transition-signaling bundled data noc switch architecture for cost-effective gals multicore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 332–337, 2013.

[51] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950.

[52] H. H. Hana and B. W. Johnson. Concurrent Error Detection in VLSI Circuits Using Time Redundancy. In *Proc. of the IEEE Southeastcon '86 Regional Conf.*, pages 208–212, March 1986.

[53] O. Hauck and S.A. Huss. Asynchronous wave pipelines for high throughput datapaths. In *Electronics, Circuits and Systems, 1998 IEEE International Conference on*, volume 1, pages 283–286 vol.1, 1998.

[54] Tino Heijmen. Radiation-induced soft errors in digital circuits – a literature survey, 2002.

[55] R. Holsmark, S. Kumar, M. Palesi, and A. Mejia. Hira: A methodology for deadlock free routing in hierarchical networks on chip. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 2–11, 2009.

[56] M.N. Horak, S.M. Nowick, M. Carlberg, and U. Vishkin. A low-overhead asynchronous interconnection network for gals chip multiprocessors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):494–507, 2011.

[57] Wei Huang, Mircea R. Stan, Kevin Skadron, Karthik Sankaranarayanan, Shougata Ghosh, and Sivakumar Velusam. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 878–883, New York, NY, USA, 2004. ACM.

[58] D.A. Huffman and Massachusetts Institute of Technology. Research Laboratory of Electronics. *The Synthesis of Sequential Switching Circuits*. Technical report (Massachusetts Institute of Technology. Research Laboratory of Electronics). Research Laboratory of Electronics, Massachusetts Institute of Technology, 1954.

[59] M. Imai and T. Yoneda. Improving dependability and performance of fully asynchronous on-chip networks. In *Asynchronous Circuits and Systems (ASYNC), 2011 17th IEEE International Symposium on*, pages 65–76, 2011.

[60] M.H. Jabbar, D. Houzet, and O. Hammami. 3d multiprocessor with 3d noc architecture based on tezzaron technology. In *3D Systems Integration Conference (3DIC), 2011 IEEE International*, pages 1–5, 2012.

[61] W. Jang and A. J. Martin. A soft-error-tolerant asynchronous microcontroller. In *13th NASA Symposium on VLSI Design*, June 2007.

[62] Wonjin Jang and A.J. Martin. Seu-tolerant qdi circuits [quasi delay-insensitive asynchronous circuits]. In *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, pages 156 – 165, march 2005.

[63] David J Kinniment. *Synchronization and Arbitration in Digital Systems*. Wiley, 2007.

[64] A. Kohler and M. Radetzki. Fault-tolerant architecture and deflection routing for degradable noc switches. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 22–31, 2009.

[65] A. Kohler, G. Schley, and M. Radetzki. Fault tolerant network on chip switching with graceful performance degradation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(6):883–896, 2010.

[66] M. Koibuchi, H. Matsutani, H. Amano, and T.M. Pinkston. A lightweight fault-tolerant mechanism for network-on-chip. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 13–22, 2008.

[67] T. Krishna, A Kumar, P. Chiang, M. Erez, and Li-Shiuan Peh. Noc with near-ideal express virtual channels using global-line communication. In *High Performance Interconnects, 2008. HOTI '08. 16th IEEE Symposium on*, pages 11–20, 2008.

[68] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *Computers, IEEE Transactions on*, 48(9):866–880, 1999.

[69] A Kumar, Li-Shiuan Peh, P. Kundu, and N.K. Jha. Toward ideal on-chip communication using express virtual channels. *Micro, IEEE*, 28(1):80–90, 2008.

[70] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express virtual channels: towards the ideal interconnection fabric. *SIGARCH Comput. Archit. News*, 35(2):150–161, June 2007.

[71] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express virtual channels: towards the ideal interconnection fabric. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 150–161, New York, NY, USA, 2007. ACM.

[72] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[73] Jakob Lechner and Robert Najvirt. A generic architecture for robust asynchronous communication links. In JoséL. Ayala, Delong Shang, and Alex Yakovlev, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 7606 of *Lecture Notes in Computer Science*, pages 121–130. Springer Berlin Heidelberg, 2013.

[74] Yun-Tae Lee. Low power soc in deep-submicron era. In *SOC Conference, 2003. Proceedings. IEEE International [Systems-on-Chip]*, pages 421–, 2003.

[75] D.H. Linder and J.C. Harden. An adaptive and fault tolerant wormhole routing strategy for k -ary n-cubes. *Computers, IEEE Transactions on*, 40(1):2–12, 1991.

[76] Andrew M Lines. Pipelined asynchronous circuits. Technical report, Pasadena, CA, USA, 1998.

[77] M. Gag, P. Gorski, T. Wegner, D. Timmermann. Evaluation of Switch-to-Switch Header Flit Protection Schemes in Networks-on-Chips. In *Zuverlässigkeit und Entwurf - 5. GI/GMM/ITG-Fachtagung*, sept. 2011.

[78] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, AUSCRYPT '90, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.

[79] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979.

[80] D.G. Messerschmitt. Synchronization in digital system design. *Selected Areas in Communications, IEEE Journal on*, 8(8):1404–1419, 1990.

[81] A. Mitra, W.F. McLaughlin, and S.M. Nowick. Efficient asynchronous protocol converters for two-phase delay-insensitive global communication. In *Asynchronous Circuits and Systems, 2007. ASYNC 2007. 13th IEEE International Symposium on*, pages 186–195, 2007.

[82] Andrey Mokhov, Victor Khomenko, Danil Sokolov, and Alex Yakovlev. On dual-rail control logic for enhanced circuit robustness. Research report, Newcastle University, 2010.

[83] C.E. Molnar, I.W. Jones, W.S. Coates, J.K. Lexau, S.M. Fairbanks, and I.E. Sutherland. Two fifo ring performance experiments. *Proceedings of the IEEE*, 87(2):297–307, 1999.

[84] Y. Monnet, M. Renaudin, and R. Leveugle. Asynchronous circuits sensitivity to fault injection. In *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International*, pages 121–126, 2004.

[85] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integr. VLSI J.*, 38(1):69–93, October 2004.

[86] M.T. Moreira, R.A. Guazzelli, and N.L.V. Calazans. Return-to-one protocol for reducing static power in c-elements of qdi circuits employing m-of-n codes. In *Integrated Circuits and Systems Design (SBCCI), 2012 25th Symposium on*, pages 1–6, 2012.

[87] K. Morgan, M. Caffrey, P. Graham, E. Johnson, B. Pratt, and M. Wirthlin. Seu-induced persistent error propagation in fpgas. *Nuclear Science, IEEE Transactions on*, 52(6):2438–2445, 2005.

[88] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proc. Int'l Symp. Theory of Switching, Part 1*, pages 204–243. Harvard Univ. Press, 1959.

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

[90] Ali Namazi. *Design-for-reliability techniques for nanometer VLSI*. PhD dissertation, THE UNIVERSITY OF TEXAS AT DALLAS, USA, 2010.

[91] Rameez Naqvi, Andreas Steininger, and J. Lechner. Protection of muller-pipelines from transient faults. *International Symposium on Quality Electronics Design 2014*, Mar. 2014.

[92] S.R. Naqvi, A. Steininger, and J. Lechner. An set tolerant tree arbiter cell. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 31–39, 2013.

[93] Syed Rameez Naqvi. A'araf: An asynchronous router architecture using four-phase bundled handshake protocol. *ICCGI 2012*, June 2012.

[94] Syed Rameez Naqvi, Varadan Savulimedu Veeravalli, and Andreas Steininger. Protecting an asynchronous noc against transient channel faults. *15th Euromicro Conference on Digital System Design*, Sep. 2012.

[95] Steven M Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical report, 1995.

[96] J. Nurmi. Network-on-chip: A new paradigm for system-on-chip design. In *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pages 2–6, 2005.

[97] Jabulani Nyathi, S. Sarkar, and P.P. Pande. Multiple clock domain synchronization for network on chip architectures. In *SOC Conference, 2007 IEEE International*, pages 291–294, 2007.

[98] Simon Ogg, Bashir Al-Hashimi, and Alex Yakovlev. Asynchronous transient resilient links for noc. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, pages 209–214, New York, NY, USA, 2008. ACM.

[99] R.O. Ozdag and P.A. Beerel. High-speed qdi asynchronous pipelines. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 13–22, 2002.

[100] Eustace Painkras, Luis A. Plana, Jim D. Garside, Steve Temple, Simon Davidson, Jeffrey Pepper, David M. Clark, Cameron Patterson, and Steve Furber. Spinnaker: A multi-core system-on-chip for massively-parallel neural net simulation. In *CICC*, pages 1–4. IEEE, 2012.

[101] Dongkook Park, C. Nicopoulos, Jongman Kim, N. Vijaykrishnan, and C.R. Das. Exploring fault-tolerant network-on-chip architectures. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 93–104, 2006.

[102] J. Patel and L. Fung. Concurrent Error Detection in ALUs by Recomputing with Shifted Operands. *IEEE Trans. on Computers*, 31(7):589–595, July 1982.

[103] A. Peeters and K. Van Berkel. Single-rail handshake circuits. In *Asynchronous Design Methodologies, 1995. Proceedings., Second Working Conference on*, pages 53–62, 1995.

[104] Song Peng and Rajit Manohar. Self-healing asynchronous arrays. In *Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, ASYNC '06, pages 34–, Washington, DC, USA, 2006. IEEE Computer Society.

[105] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[106] Matthew Pierson. Self timed, self tuned state machines using low power pass transistor logic.

[107] Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. Workcraft – a framework for interpreted graph models. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, pages 333–342. Springer Berlin Heidelberg, 2009.

[108] JULIAN JOSÉ HILGEMBERG Pontes and Faculty of Informatics PUCRS. *Soft Error Mitigation in Asynchronous Networks on Chip*. August.

[109] N. R. Poole. Self-timed logic circuits. *Electronics Communication Engineering Journal*, 6(6):261–270, 1994.

[110] D. K. Pradhan. *Fault-Tolerant Computing: Theory and Techniques*, volume I. Prentice Hall, Englewood Cliffs, New Jersey, 2003.

[111] Jan M. Rabaey, Anantha P. Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits : a design perspective*. Prentice Hall electronics and VLSI series. Pearson Education, 2 edition, January 2003.

[112] Amir-Mohammad Rahmani, Khalid Latif, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. A stacked mesh 3d noc architecture enabling congestion-aware and reliable interlayer communication. In *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '11, pages 423–430, Washington, DC, USA, 2011. IEEE Computer Society.

[113] D. A. Rennels, A. Avizienis, and M. Ercegovac. A Study of Standard Building Blocks for the Design of Fault-Tolerant Distributed Computer System. In *Proc. of the 8th Int'l. Conf. on Fault-Tolerant Computing Systems*, pages 208–212, June 1978.

[114] D. A. Reynolds and G. Metze. Fault Detection Capabilities of Alternating Logic. *IEEE Trans. on Computers*, 27(12):1093–1098, December 1978.

[115] Dobkin (Reuven) Rostislav, Victoria Vishnyakov, Eyal Friedman, and Ran Ginosar. An asynchronous router for multiple service levels networks on chip. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, ASYNC '05, pages 44–53, Washington, DC, USA, 2005. IEEE Computer Society.

[116] S. Rusu. Clock generation and distribution in high-performance processors. In *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, pages 207–, 2004.

[117] Maitham Shams, Jo C. Ebergen, and Mohamed I. Elmasry. Asynchronous circuits.

[118] X. She and N. Li. Low-overhead single-event upset hardened latch using programmable resistance cells. *Computers Digital Techniques, IET*, 4(5):420–427, 2010.

[119] A. Sheibanyrad and A. Greiner. Two efficient synchronous ↔ asynchronous converters well-suited for networks-on-chip in gals architectures. *Integr. VLSI J.*, 41(1):17–26, January 2008.

[120] A. Sheibanyrad, A. Greiner, and I. Miro-Panades. Multisynchronous and fully asynchronous nocs for gals architectures. *Design & Test of Computers, IEEE Transactions on*, 25(6):572 – 580, Dec. 2008.

[121] Y. Shi, S. Furber, and University of Manchester. School of Computer Science. *Fault-tolerant delay-insensitive communication*. University of Manchester, 2010.

[122] Yebin Shi, S.B. Furber, J. Garside, and L.A. Plana. Fault tolerant delay insensitive inter-chip communication. In *Asynchronous Circuits and Systems, 2009. ASYNC '09. 15th IEEE Symposium on*, pages 77–84, 2009.

[123] Lai Yit Siang, B.A. Bin Rosdi, Teh Eng Keong, Teh Chai Forng, and C.L.S. Xying. An automated clock distribution topology in soc designs. In *Industrial Electronics and Applications (ISIEA), 2011 IEEE Symposium on*, pages 454–458, 2011.

[124] M. Singh and S.M. Nowick. Mousetrap: ultra-high-speed transition-signaling asynchronous pipelines. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 9–17, 2001.

[125] G.E. Sobelman and K. Fant. Cmos circuit design of threshold gates with hysteresis. In *Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on*, volume 2, pages 61–64 vol.2, 1998.

[126] W. Song, D. Edwards, and University of Manchester. School of Computer Science. *Spatial Parallelism in the Routers of Asynchronous On-chip Networks*. University of Manchester, 2011.

[127] Jens Sparso and Steve Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[128] Kenneth Stevens. Automatic synthesis of fast, compact self-timed control. Technical report, In IFIP Working Conference on Design Methodologies, 1992.

[129] Kenneth S. Stevens. Practical verification and synthesis of low latency asynchronous systems, 1994.

[130] A. Strano, A. Ghiribaldi, M. Favalli, and D. Bertozzi. Power efficiency of switch architecture extensions for fault tolerant noc design. In *Sustainable Computing and Computing for Sustainability. The Third International Green Computing Conference (IGCC'12)*, jun. 2012.

[131] C.E. Stroud. Reliability of majority voting based vlsi fault-tolerant circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):516 –521, dec. 1994.

[132] Mishell J. Stucki, Severo M. Ornstein, and Wesley A. Clark. Logical design of macromodules. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 357–364, New York, NY, USA, 1967. ACM.

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

[134] Ivan Sutherland and Scott Fairbanks. Gasp: A minimal fifo control. In *Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, ASYNC '01, pages 46–, Washington, DC, USA, 2001. IEEE Computer Society.

[135] Rutuparna Ramesh Tamhankar, Srinivasan Murali, and Giovanni De Micheli. Performance driven reliable link design for networks on chips. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 749–754, New York, NY, USA, 2005. ACM.

[136] Teijo Lehtonen, Pasi Liljeberg, Juha Plosila. Online Reconfigurable Self-Timed Links for Fault-Tolerant NoC. *VLSI Design*, 2007.

[137] Navid Toosizadeh. *Enhanced Synchronous Design Using Asynchronous Techniques*. The University of Toronto, 2010.

[138] Wen-Chung Tsai, Ying-Cherng Lan, Yu-Hen Hu, and Sao-Jie Chen. Networks on chips: structure and design methodologies. *JECE*, 2012:2:2–2:2, January 2012.

[139] K. Van Berkel, R. Burgess, J. Kessels, A. Peelers, M. Roncken, and F. Schalij. A fully-asynchronous low-power error corrector for the dcc player [cmos technology]. In *Solid-State Circuits Conference, 1994. Digest of Technical Papers. 41st ISSCC., 1994 IEEE International*, pages 88–89, 1994.

[140] Kees van Berkel. Beware the isochronic fork. *Integr. VLSI J.*, 13(2):103–128, June 1992.

[141] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008.

[142] S.N. Varanasi, K.S. Stevens, and G. Birtwistle. Concurrency reduction of untimed latch protocols - theory and practice. In *Asynchronous Circuits and Systems (ASYNC), 2010 IEEE Symposium on*, pages 26 –37, may 2010.

[143] Varadan Savulimedu Veeravalli and Andreas Steininger. Performance of radiation hardening techniques under voltage and temperature variations. In *2013 IEEE Aerospace Conference*, Mar. 2013.

[144] T. Verhoeff and Technische Universiteit Eindhoven. *A Theory of Delay-insensitive Systems*. CIP-Gegevens Koninklijke Bibliothek, 1994.

[145] P. Vivet, D. Dutoit, Y. Thonnart, and F. Clermidy. 3d noc using through silicon via: An asynchronous implementation. In *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, pages 232–237, 2011.

[146] B.D. Winters and M.R. Greenstreet. A negative-overhead, self-timed pipeline. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 37–46, 2002.

[147] Dong Xiang, Yueli Zhang, and Yi Pan. Practical deadlock-free fault-tolerant routing in meshes based on the planar network fault model. *Computers, IEEE Transactions on*, 58(5):620–633, 2009.

[148] Yongfeng Xu, Jianyang Zhou, and Shunkui Liu. Research and analysis of routing algorithms for noc. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 2, pages 98–102, 2011.

[149] A. Yakovlev, A. Petrov, and L. Lavagno. A low latency asynchronous arbitration circuit. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(3):372 –377, sept. 1994.

[150] A. Yanamandra, S. Eachempati, N. Soundararajan, V. Narayanan, M.J. Irwin, and R. Krishnan. Optimizing power and performance for reliable on-chip networks. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 431 –436, jan. 2010.

[151] Chengmo Yang and Alex Orailoglu. Light-weight synchronization for inter-processor communication acceleration on embedded mpsocs. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 150–154, New York, NY, USA, 2007. ACM.

[152] Keun Soo Yim, V. Sidea, Z. Kalbarczyk, Deming Chen, and R.K. Iyer. A fault-tolerant programmable voter for software-based n-modular redundancy. In *Aerospace Conference, 2012 IEEE*, pages 1 –20, march 2012.

[153] Young Jin Yoon, Nicola Concer, Michele Petracca, and Luca Carloni. Virtual channels vs. multiple physical networks: a comparative analysis. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 162–165, New York, NY, USA, 2010. ACM.

[154] Qiaoyan Yu, Meilin Zhang, and P. Ampadu. Exploiting inherent information redundancy to manage transient errors in noc routing arbitration. In *Networks on Chip (NoCS), 2011*, pages 105 –112, may 2011.