

University of Newcastle upon Tyne
School of Electrical, Electronic and Computer Engineering



Multi-resource Approach to Asynchronous SoC: Design and Tool Support

by

Stanislavs Golubcovs

PhD Thesis

December 2011

Abstract

As silicon cost reduces, the demands for higher performance and lower power consumption are ever increasing. The ability to dynamically control the number of resources employed can help balance and optimise a system in terms of its throughput, power consumption, and resilience to errors. The management of multiple resources requires building more advanced resource allocation logic than traditional 1-of- N arbiters posing the need for the efficient design flow supporting both the design and verification of such systems.

Networks-on-Chip provide a good application example of distributed arbitration, in which the processor cores needing to transmit data are the clients; and the point-to-point links are the resources managed by routers. Building fast and smart arbiters can greatly benefit such systems in providing efficient and reliable communication service.

In this thesis, a multi-resource arbiter was developed based on the Signal Transition Graph (STG) development flow. The arbiter distributes multiple active interchangeable resources that initiate requests when they are ready to be used. It supports concurrent resource utilization, which benefits creating asynchronous Multiple-Input-Multiple-Output (MIMO) queues.

In order to deal with designs of higher complexity, an arbiter-oriented design flow is proposed. The flow is based on digital circuit components that are represented internally as STGs. This allows designing circuits without directly working with STGs but allowing their use for synthesis and formal verification. The interfaces for modelling, simulation, and visual model representation of the flow were implemented based on the existing modelling framework. As a result, the verification phase of the flow has helped to find hazards in existing Priority arbiter implementations.

Finally, based on the logic-gate flow, the structure of a low-latency general purpose arbiter was developed. This design supports a wide variety of arbitration problems including the multi-resource management, which can benefit building NoCs employing complex and adaptive routing techniques.

Contents

List of Figures	viii
List of Tables	xii
Acknowledgements	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Main Contributions	4
1.3 Organisation of Thesis	5
1.4 Bibliography	7
2 Background	8
2.1 Asynchronous Circuits	8
2.2 Digital Circuits	10
2.3 Petri Nets	12
2.3.1 Pre-set and Post-set	13
2.3.2 Enabling and Firing	13
2.3.3 Other PN Properties	14
2.3.4 Signal Transition Graphs	14
2.4 Asynchronous Circuit Primitives	16
2.4.1 C-element	16
2.4.2 Toggle Component	17
2.4.3 Decision Wait Element	17

2.5	Asynchronous Signalling	18
2.5.1	Handshake Protocols	18
2.5.2	Channel Types	19
2.5.3	Delay-insensitive Encoding	20
2.5.4	Dual-rail	20
2.5.5	Bundled data Encoding	22
2.6	Example of Logic Synthesis Using Petrify	22
3	Review on Asynchronous Arbiters	27
3.1	Introduction	27
3.2	Arbiter-specific Properties	28
3.3	Metastability	31
3.4	Analogue arbiters	32
3.4.1	The MUTEX Element	32
3.4.2	Analogue 1-of-3 arbiter	33
3.4.3	Analogue 2-of-3 Arbiter	34
3.5	Two-way Arbiters	34
3.5.1	4-phase Arbitration (RG)	35
3.5.2	2-phase Arbitration (RGD)	36
3.5.3	“Nacking” Arbiter	36
3.6	1-of- N Multi-way Arbiters	38
3.6.1	Mesh-based Implementation	39
3.6.2	Cascaded Tree Arbiters	40
3.6.3	Token Ring	41
3.6.4	Ordered Arbiters	43
3.6.5	Priority Arbiters	44
3.7	Multi-resource Arbiters	47
3.7.1	Multi-token Arbiters	47
3.7.2	Patil’s Arbiter	49
3.7.3	Committee arbiter	50

3.8	Conclusions	51
4	Concurrent Multi-Resource Arbiter: Design and Applications	53
4.1	Introduction	53
4.2	Design Method	55
4.3	2×2 Arbiter Design	56
4.3.1	Functionality	56
4.3.2	Resolving the Conflict	59
4.3.3	Implementation	63
4.3.4	Verification of the Circuit	66
4.3.5	Latency Estimation	66
4.3.6	Simulation in Spectre	68
4.3.7	Cost of the Parallelism	69
4.4	Extending up to $N \times M$ Arbiters	69
4.4.1	Column/row Blocking	71
4.4.2	Ring-based Blocking	73
4.4.3	Latency Estimation	75
4.4.4	Simulation in Spectre	75
4.4.5	Fairness of the Arbiter	76
4.5	Multi-resource Arbiter for Passive Resources	76
4.5.1	Task Specification	76
4.5.2	Implementation of the Ring Cell	78
4.5.3	Implementation of the Client Controller and Token Controller	80
4.5.4	Latency Estimation	80
4.5.5	Comparison with Patil's Arbiter	81
4.6	Designing MIMO Queues	81
4.6.1	MIMO Performance Comparison	83
4.7	Conclusions	83

5	Gate-level Design Flow	85
5.1	Introduction	85
5.2	Features of the Gate-level Design Flow	87
5.2.1	Basic Plugin Components	87
5.2.2	Gates of High Complexity	89
5.2.3	Delay-Insensitive Circuits	90
5.2.4	Circuits with Timing Assumptions	91
5.2.5	High-level Models	93
5.2.6	State Space Exploration	94
5.2.7	Circuits with MUTEX Elements	96
5.3	Analysis of Priority Arbiter	98
5.4	Conclusions	101
6	Design of Generalized Arbiter	102
6.1	Introduction	102
6.2	Arbiter Design	103
6.2.1	Design Method	103
6.2.2	Basic Structure	104
6.2.3	Decomposition	108
6.2.4	High Performance	109
6.2.5	Avoiding Deadlocks	109
6.2.6	Circuit Verification	110
6.3	Possible Extensions	113
6.4	Performance Estimations	114
6.4.1	Priority 2-of-3 Arbitration	114
6.4.2	Pipelined Arbiter Scaling	115
6.5	Conclusions	116
7	Conclusions	118
7.1	Summary of Contribution	119

7.2	Future Work	119
A	Summary on Asynchronous Arbiters	120
B	Workcraft Interface	123
B.1	Main Window	123
B.1.1	Basic Mouse Controls	124
B.2	Common Operation Modes	124
B.3	STG Plugin Operation Modes	124
B.4	Digital Circuit Plugin Operation Modes	126
B.5	Conversion to the Circuit STG	126
B.6	Simulation	126
B.7	Verification	127
	References	129

List of Figures

1.1	GALS evolution	3
2.1	Petri net firing transition	14
2.2	STG example of a C-element	15
2.3	C-element examples	17
2.4	Toggle component	17
2.5	Decision-wait element	18
2.6	Basic handshake protocols	18
2.7	Channel types	20
2.8	Dual-rail encoding	21
2.9	Bundled data, 4-phase push channel	22
2.10	Channel converter structure	23
2.11	Control logic STG	24
3.1	Naive arbiter implementation and metastability	32
3.2	MUTEX element [47]	33
3.3	MUTEX with standard gates [35]	33
3.4	Tri-flop arbiter	34
3.5	2-of-3 analogue arbiter	35
3.6	4-phase two-way arbiter	36
3.7	2-phase two-way arbiter	37
3.8	Nacking arbiter	37
3.9	Multi-input arbiter	38

3.10	1-of-3 mesh	39
3.11	Generic mesh structure	40
3.12	Tree structure	40
3.13	Cascaded arbiter STG and implementation	41
3.14	Token ring high-level models	42
3.15	Ring implementations	43
3.16	Ordered FIFO arbiter structure	44
3.17	Ordered 3-way arbiter implementation	44
3.18	Three-way static priority arbiter	46
3.19	Eight-ways dynamic priority arbiter	47
3.20	Multi-token arbiters	48
3.21	Multi-resource “Forward acting” arbiter	49
3.22	Committee problem	51
4.1	Synopsis of an arbiter [35]	54
4.2	Arbiter design flow	56
4.3	2×2 arbiter interface	57
4.4	2×2 arbiter STG	58
4.5	Additional exclusion places added	60
4.6	State graph of the modified STG	61
4.7	STG with MUTEX elements	62
4.8	Arbiter structure	64
4.9	2×2 arbiter implementation	64
4.10	Timing diagram	67
4.11	4×3 arbiter implementation (shows active requests on $r2g$ and $c2g$)	70
4.12	Arbiter with blocking tiles	71
4.13	Column/row block tile STG	72
4.14	Tile implementation for the C/R blocks	73
4.15	Ring-based tile STG	74
4.16	Tile implementing ring-based approach	74

4.17	Asymmetric multi-resource arbiter structure	77
4.18	Busy token ring cell	78
4.19	The STG of a ring cell	79
4.20	Implementation of the ring cell	79
4.21	Ring cell structure	80
4.22	Structure of the sequential 2×2 MIMO queue [80]	82
4.23	8×8 MIMO queue	82
4.24	MIMO performance	83
5.1	Toggle component	88
5.2	MUTEX element	89
5.3	High complexity gates	90
5.4	Modelling asymmetric forks	91
5.5	Counter with timing assumptions	92
5.6	3-input arbiter (high level model)	93
5.7	High-level view on 2×2 arbiter	93
5.8	Generated MUTEX STG	95
5.9	Mpsat verification flow	96
5.10	C-element formed of NAND gates	96
5.11	NAND-based C-element STG	97
5.12	Modelled priority arbiter	99
5.13	<i>LOCK</i> decomposition	100
6.1	Arbitration example in 2D routing grid	103
6.2	Arbiter design flow	104
6.3	Generalized arbiter high-level PN	105
6.4	High-level circuit structure	105
6.5	Decomposition into simple gates	108
6.6	<i>SYNC_OR</i> decomposition	111
6.7	Pipelining	113

6.8	Employing RGD interface	114
6.9	Decoupling synchronizer and grant controller	115
6.10	Pipelined arbiter performance	116
B.1	Workcraft Interface	125
B.2	Opening generated STG	127

List of Tables

2.1	Dual rail 4-phase codes	21
4.1	2×2 performance estimation in Spectre	68
4.2	4×3 performance estimation in Spectre	76
6.1	Priority 2-of-3 arbitration	114
A.1	Analogue arbiters	120
A.2	Two-way arbiters	121
A.3	1-of- N arbiters	121
A.4	Other arbiters	122

Acknowledgements

I am grateful to my supervisor, Prof. Alex Yakovlev, for his incredibly wise guidance and motivating talks. I am also grateful to Dr. Alex Bystrov and Dr. Fei Xia for their fruitful discussions on various related topics.

I would like to thank my friends and colleagues Dr. Danil Sokolov, Dr. Andrey Mikhov, Dr. Robin Emery and Mr. Ashur Rafiev for contributing their ideas and making my student life full of fun and excitement. Special thanks to Mr. Arseny Alekseyev and Dr. Ivan Poliakov for their great technical help with the Workcraft source code. In addition, I wish to thank Dr. Victor Khomenko for his advice and work on the tools Punf and Mpsat which were of great help with many complex design decisions.

A very special thanks to my parents Alexander and Antonina for their constant encouragement and moral support.

This work was supported by EPSRC grant GR/E044662/1 (STEP).

Chapter 1

Introduction

1.1 Motivation

The increasing scale of modern designs is accompanied by the increased risk of failure; as a result large multi-functional designs are more difficult to test and implement. To address the problem, the larger systems are composed of smaller well tested design blocks dedicated to particular tasks and combined into a single System-on-a-Chip (SoC) [66]. The result is a system with processors, memory components, encoders, floating point units, and other blocks (also called the *intellectual property* (IP) cores), whose complexity is constrained allowing combining solutions from independent vendors.

The greater requirements for high-performance systems have led designers to replace large complex cores with multi-core systems composed of simpler cores [49]. The repeated use of the same core effectively creates a pool of resources (or service providers), which allows greater control over balancing between performance and power consumption.

Another question is durability; one or more cores may become temporarily or permanently inactive because of a fault. With the redundant resources, the “healthy” cores could take over the job and prevent the system from failure. Examples of such resource redundancy are *RAID* (Redundant Array of Independent Disks) and *RAIM* (Redundant Array of Independent Memory) data storage architectures. Both technologies improve

resilience and performance of storing information on disk or in memory. In general, such a redundant resource approach can be applied to any type of resource to achieve a trade-off between performance and reliability.

Nowadays the *Network-on-Chip* (NoC) architecture is increasingly popular, where devices communicate through a set of Point-to-Point (P2P) links that form network structures. A network is distributed over multiple link segments in order to allow concurrent communication between multiple devices and to reuse the same links for propagating data to different destinations. Essentially, a NoC provides and manages a distributed set of communication resources. Various topologies, depending on the routing protocol, provide different opportunities on what data propagation paths can be taken. For instance; on a regular 2D NoC mesh there are multiple shortest paths possible for the data travelling between two cores located in the opposite corners of the mesh. Theoretically, such a choice of a particular path can be regarded as a redundant/replaceable resource, which can be used for improving NoC performance, durability, and power consumption.

Asynchronous NoCs

With the increasing clock skew on smaller transistor sizes, planning multiple unrelated clocks independently driving separate time domains becomes a necessity. The system built on the concept of multiple clocks is called *GALS* (Globally Asynchronous, Locally Synchronous) system, initially proposed by D. M. Chapiro in [15]. It assumes splitting the design into individual modules with independent clock rates, which simplifies the task of timing closure and enables design reuse by allowing multiple designs with independent clock rates on the same chip. The communication latency, however, may have a significant impact on the overall system performance [30].

Essentially, GALS is a compromise between fully synchronous and fully asynchronous systems which may have various degrees of asynchrony in its communication layer. The basic GALS designs communicate via dedicated links (Figure 1.1a) where the communication is built by directly connecting these domains through special synchronous-to-synchronous (sync-sync) interfaces (Figure 1.1).

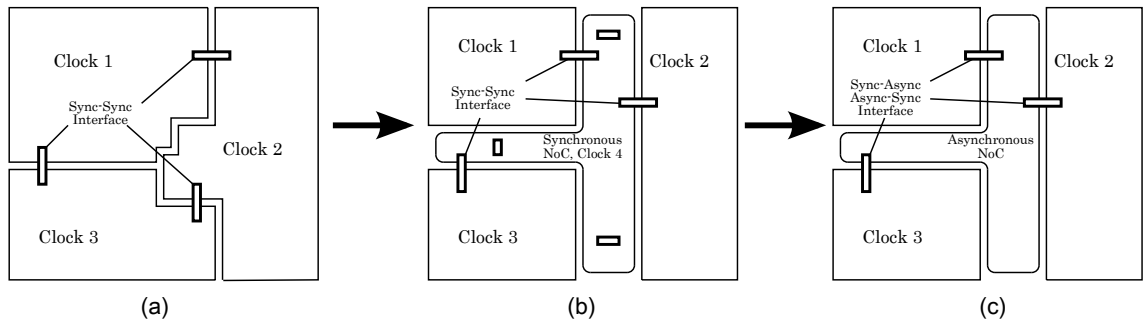


Figure 1.1: GALS evolution

As the number of independent cores grows, dedicated communication networks are created (Figure 1.1b). At this stage, the network itself may still be clocked; however, since this clock needs to cover the whole NoC area, the design had complications similar to those in systems with a global clock skew. The clock branches with high skew require special *mesochronous synchronizers* [42] that synchronize skewed clock branches belonging to the same clock domain, this may also result in extra latency for the inter-core communication. It was shown that with the smaller transistor size even more synchronization cycles may be needed, leading to the increased latency overhead [8], thus motivating the fully asynchronous networks on chip.

The complete removal of global clock from the NoC environment having fully asynchronous NoC implementation (Figure 1.1c) is the next step of the GALS evolution. Such a network does not have the issue of timing closure and only needs synchronizers to transfer data from asynchronous environment to clocked domains. As a result, the asynchronous NoC designs do not suffer as much the penalty of extra synchronization latency caused by synchronizers.

Smart Arbiters

Because NoC communication resources are reused by multiple clients, additional arbitration circuitry is needed to manage resource utilization. *Arbiters* are designed to prevent multiple client accesses when the resource cannot serve more than one client at a time. Based on some form of communication, the arbiter forbids client accesses when the resource is not ready to be used. Alternatively, by granting a resource, the arbiter guaran-

tees that the resource can be used without the risk of being interfered. This simple form of 1-of- N arbitration can be also extended to multiple resources. The M -of- N arbitration featuring N clients and M resources forms a double-sided conflict where multiple clients compete for the acquisition of a resource, and multiple resources also compete to access the clients (supposedly because a client cannot be engaged with more than one resource at a time) [78].

The asynchronous environment is difficult for arbitration because there are no timing expectations about when the requests may arrive, and the only way the arbiter can decide which request to grant is by serving requests based on their relative arrival time. Simple digital logic used in synchronous arbiters is not sufficient in this case because the arriving events may be too close leaving the arbiter in a metastable state [37]. The metastability in asynchronous arbiters has to be contained within known boundaries by using specialized circuitry; as a result, asynchronous arbiters are not so easy to design and scale.

The well known 1-of- N asynchronous arbiters are still being actively studied; however, the more general M -of- N arbiters also need to be considered in order to address the problems of the multi-resource utilization. This is the primary goal of this work.

Currently, there are no integrated design flows supporting asynchronous arbiters. Instead, these designs are often made “by hand” with no guarantees of correct operation in regards to deadlocks and hazards. For reliable solutions, the explicit support for verification is needed.

It is believed that smart asynchronous arbiters can help building the asynchronous routers, which in turn will allow creating sophisticated asynchronous NoCs with optimal utilization of available communication resources.

1.2 Main Contributions

The first contribution of this thesis is a design of a multi-resource arbiter supporting M -of- N arbitration among active resource components. It directly serves the idea of providing interchangeable resources which improves the overall performance and reliability of a system. As an example, the Multiple-Input-Multiple-Output queue is created, which in

practice can be used for load balancing among a number of processors.

The second contribution is the design of gate-level EDA flow and its implementation as a plugin for the *Workcraft* [64] modelling environment. It supports high-complexity gates, timing assumptions, and non-deterministic components (such as MUXes), which is useful in designing small-to-medium sized asynchronous arbiters. The flow supports formal verification for deadlocks and hazards, which helps the design of new and the verification of existing circuits. This flow is evaluated through a complete design, modelling, and verification of the *priority arbiter*.

The third contribution is the design of a generalized arbiter based on the proposed flow. This arbiter can be used to solve a large variety of resource allocation tasks. It is presented at different degrees of abstraction, at high-level and at the level of decomposition. Timing estimations are made for the pipelined version of this arbiter, demonstrating its performance for the increasing number of arbiter clients.

1.3 Organisation of Thesis

Chapter 1: Introduction motivates the necessity of the asynchronous multi-resource arbitration and briefly outlines the scope and contribution of the thesis.

Chapter 2: Background describes the asynchronous circuits and their advantages and disadvantages in comparison with the clocked designs. It defines Petri nets, STGs and overviews their roles in specifying digital circuits at various degrees of abstraction.

Chapter 3: Review on Asynchronous Arbiters studies typical features characterising arbitration circuits. It reviews existing arbiter designs both analogue and digital. In this chapter multiple conventional arbiters are presented in order of increasing complexity starting from various 1-of-2 arbiters and advancing into scalable 1-of- N and M -of- N designs.

Chapter 4: Concurrent Multi-Resource Arbiter: Design and Applications presents the initial solution to multi-resource arbitration with active resources. A number of

practical applications are also presented along with performance estimations comparing the design against existing alternative implementations in the area of MIMO queues.

Chapter 5: Gate-level Design Flow presents the gate-level EDA flow supporting arbiter models, which was implemented as part of the Workcraft modelling environment in an attempt to ease the STG based design flow. It also demonstrates how existing arbiter circuits can be modelled and verified based on the example of priority arbiter.

Chapter 6: Design of Generalized Arbiter presents the structure of the generalized arbiter developed with the flow proposed in previous chapter. The arbiter is able to tackle a large variety of allocation problems making it an invaluable circuit for designing advanced asynchronous NoC routers ensuring a high degree of link resource utilization.

Chapter 7: Conclusions summarises the major results achieved in this work and suggests the areas for future research.

1.4 Bibliography

- Stanislavs Golubcovs, Delong Shang, Fei Xia, Andrey Mokhov, and Alex Yakovlev. Concurrent Multi-Resource Arbiter: Design and Applications
Accepted for *IEEE Transactions on Computers*
- Stanislavs Golubcovs and Alex Yakovlev. *Low Power Networks-on-Chip*, chapter 4: Asynchronous Communications for NoCs, pages 71–109.
Springer Verlag, 2010.
- Delong Shang, Fei Xia, Stanislavs Golubcovs, and Alex Yakovlev. The magic rule of tiles: Virtual delay insensitivity.
In *PATMOS*, pages 286–296, 2009.
- Stanislavs Golubcovs, Delong Shang, Fei Xia, Andrey Mokhov, and Alex Yakovlev. Multi-resource arbiter decomposition.
Technical Report NCL-EECE-MSDTR-2009-143, Newcastle University, February 2009.
- Stanislavs Golubcovs, Delong Shang, Fei Xia, Andrey Mokhov, and Alex Yakovlev. Modular approach to multi-resource arbiter design.
In *Asynchronous Circuits and Systems*, 2009. ASYNC '09. 15th IEEE Symposium on, pages 107–116, May 2009.
- Stanislavs Golubcovs, Andrey Mokhov, and Alex Yakovlev. Multi-resource arbiter design.
Technical report, UK Asynchronous Forum, Manchester, 2008.

Chapter 2

Background

2.1 Asynchronous Circuits

Asynchronous or self-timed circuits operate without a global clock sequentially updating the state of the system. They are formed of multiple control signals causing partial system state updates and operate in the continuous time domain, which provides certain advantages over the synchronous designs.

Modularity The asynchronous circuits are composed of modular blocks that intercommunicate through handshakes. It is possible to develop these blocks independently with different techniques and simply compose them together afterwards. Because the communication protocols usually do not have specific timing expectations, modules can work at different speeds without the risk of violating timing constraints. In contrast, clocked systems have to be designed with the specific clock signal in mind and a slight variation on the signal delay may have global implications.

Variability

The shrinking of the layout feature sizes causes an increase in interconnect delay variability for both traditional die-to-die and emerging intra-die variations [83]. The delay variability also causes clock skew variations. For the $0.25\mu m$ process the reported variability was already 25% [39]. And finally, the crosstalk capacitance can cause up to $1.5\times$

delay variation [29]. All these factors require much more sophisticated timing analysis. Safety margins associated with the variability have to be increased at the cost of system performance.

Similar issues occur in self-timed designs. However, the number of timing assumptions is significantly less. The robust *delay-insensitive* designs allow the circuits to operate without explicit timing expectations on the delay of wires or gates, hence, variations in these delays do not prevent correct functionality.

The asynchronous circuits are highly adaptive to power supply variation. With less power, transistors would react slower, but with no strict timing constraints, such a design would still continue to work.

Power Consumption

In synchronous systems, the clock signal needs to be propagated across the whole area of the chip, which makes it a major power consumer. The asynchronous design is based on more or less localized handshake communications. As a result, it often requires much less power to operate.

Another important aspect is event-based power consumption. In a synchronous system, the clock signal drives each clocked element of the circuit irrespective of real data changes. On the contrary, the asynchronous design is driven by the “on demand” philosophy. It dissipates power only when it is activated by the incoming request signal.

Static power consumption is related to the technology used and the area of the circuit. For CMOS technology of 130nm and above, static power consumption is a minor contributor. As it scales down to 90nm and beyond, the static (leakage) power becomes a greater concern. Again, asynchronous techniques allow applying power gating in a more flexible event-based way than in clocked circuits [38].

Performance

The rate of computation in synchronous systems is dependent on the *critical path*. It is the longest time, which takes a signal to propagate between two clocked components.

The critical path determines the clock rate at which components can be latched and has to include variations caused by different input data, high temperature, and low voltage supply. As a result, the synchronous circuit is as fast as the slowest path in its worst performance variation.

The asynchronous circuits operate through handshakes, which introduces overhead into the computation process. Each computation transaction has to explicitly end with the acknowledgement signal sent back to the requester, making the asynchronous computation less attractive. On the other hand, the asynchronous handshakes work as fast as the computation lasts. This leads to significant performance advantages, if the worst case delays far exceed the delay of an average computation.

2.2 Digital Circuits

Throughout this work, the concept of an asynchronous *digital circuit* is inherited from D. Muller’s switching circuits [55]. A digital circuit is a set of binary signals (also called components or gates), where each signal is associated with a binary “current value”. The signal is called *active*, when the signal value is “1” or *inactive*, when the signal value is “0”. A signal may also have an associated Boolean function $f(x_1, \dots, x_n)$ specifying its “next value” depending on the *inputs of the signal*. For instance, the AND gate can be specified as $a = b \cdot c$, where a is the name of the signal, and b and c are the inputs for a . For convenience, the signal a may be written in its set/reset form:

$$a = \begin{cases} \uparrow & b \cdot c \\ \downarrow & \bar{b} + \bar{c} \end{cases}$$

A signal is *excited* when its next value function becomes different from its current value. This may happen when one of its inputs x_1, \dots, x_n has changed. The excited signal may eventually align its value with the value of the associated function making the signal *stable* again. When the signal settles its value from excited to the stable state, it is said that the signal *fires* a transition. The timing it takes for this sort of transition to

complete is called the delay of the signal (or the gate delay), which may be arbitrarily long in the general case.

All signals of a circuit are subdivided into the *input*, *internal*, and *output signals*. The internal and the output signals represent implementation of a circuit, while the input signals represent the environment of the circuit (it is usually a simplification describing the behaviour of an environment on the signals, but not its implementation).

In contrast with the non-digital circuits (called the *analogue circuits*), signal changes from the excited to the stable state are instantaneous and never hold values outside the binary set $\{0, 1\}$. It is also assumed that these digital signals never produce glitches due to the complexity of their function: for any single change among a signal inputs x_1, \dots, x_n , the signal will either remain stable or become excited and eventually change its value (i.e., it will never make more than one transition for each input change).

Under the assumption of non-zero gate delays, it is also possible that the function change happens after the signal was excited already. The signal change from its excited state to the stable state denotes its *non-persistency*. The non-persistency of any of the internal or output signals is considered a *hazard* and wrong circuit operation. The non-persistency of an input signal is not considered a problem because that is outside the scope of circuit responsibility.

Classification

Asynchronous digital circuits fall into the following groups [75]:

- *Delay-insensitive circuits* (DI) assume that the circuit is expected to work “correctly” regardless of its wire and gate delays. This class of DI circuits is considered the most robust with respect to process and environmental variations. In practice the class of delay-insensitive circuits is rather small [46].
- *Speed-independent circuits* (SI) assume that the circuit is expected to work “correctly” regardless of its gate delays. This class of circuits assumes there is no delay in wires and the change of one signal value immediately affects all of the functions in other signals.

- *Quasi-delay-insensitive* (QDI) circuits assume the presence of *isochronic forks* [46], where the wire delay branches are considered to have no difference. If it is assumed that a delay on a wire before its branching is part of the delay of a gate driving the wire, then QDI=SI. This class of circuits may also make assumptions about one-sided fork delays, where one branch of a fork is either the same or a slower branch, which are called the *asymmetric forks* (see Chapter 5 for more details). These forks impose fewer constraints on the design and are easier to enforce.
- *Self-timed circuits* (also called circuits with timing assumptions), where the circuit is expected to work “correctly” if certain relative gate delay timing conditions are satisfied. In practice, such circuits are less reliable with respect to timing variations and need special attention during their layout. These circuits are fairly practical and help to greatly simplify the logic, when the *timing assumptions* are reasonable and easy to enforce by layout or additional delay elements.

2.3 Petri Nets

The *Petri net* model was introduced by Carl Adam Petri in his PhD thesis in 1962 [62, 56]. It is a mathematical formalism describing concurrent events, causalities between these events (also called transitions), and a dynamic state of a certain system. As opposed to the Finite State Machine (FSM) with a global state, the Petri net is described by a multiple of distributed states, which better corresponds to the nature of the asynchronous circuit events.

Definition 2.1. A Petri net is a quadruple, $PN = (P, T, F, m)$ where:

P is a finite set of places

T is a finite set of transitions: $(T \cap P = \emptyset)$

$F : (T \times P) \cup (P \times T) \rightarrow \mathbb{N}$ is a flow relation

$m : P \rightarrow \mathbb{N}$ is the marking of the net (also called the state of the net)

The definition means there are places and transitions interconnected with each other. A connection is only possible from a transition to a place or from a place to a transition.

The marking is used to represent the dynamic state of a system. An example of a Petri net is shown in Figure 2.1a on the following page. Here $F(p1, t1) = 1$, $F(p0, t1) = 2$, ..., $F(t2, p5) = 1$, in the diagram it is shown by the number or directed arcs connecting places and transitions. Marking for places $m(p1) = 1$, $m(p2) = 2$, e.t.c., or in other words: $p1$ has 1 token and $p2$ has 2 tokens.

2.3.1 Pre-set and Post-set

Definition 2.2. The *pre-set* of a transition t , denoted $\bullet t$, is the set of places $p \in \bullet t \subseteq P$ such that $p \in \bullet t \Rightarrow F(p, t) > 0$. Similarly, a pre-set of a place p , denoted $\bullet p$, is the set of transitions $t \in \bullet p \subseteq T$ such that $t \in \bullet p \Rightarrow F(t, p) > 0$. Symmetrically, a *post-set* of a transition (or a place) a , denoted a^\bullet , is the set of places (or transitions) b , such that $b \in a^\bullet \Rightarrow F(a, b) > 0$.

In the example on Figure 2.1a: $\bullet t1 = \{p0, p1, p2\}$, $t1^\bullet = \{p3, p4\}$, $\bullet p2 = \emptyset$, and $p2^\bullet = \{t1, t2\}$.

2.3.2 Enabling and Firing

Definition 2.3. A transition $t \in T$ is *enabled* at a marking m_1 if for any place $p \in \bullet t$, $m_1(p) \geq F(p, t)$. The enabled transition may *fire* producing a new marking m_2 :

$$\forall p \in P : m_2(p) = m_1(p) - F(p, t) + F(t, p),$$

where each $+$ and $-$ are defined component-wise. In other words, firing t subtracts $F(p, t)$ tokens from $\bullet t$ for each $p \in \bullet t$ and adds $F(t, p)$ tokens to t^\bullet for each $p \in t^\bullet$.

In the example (Figure 2.1a): the transition $t1$ is enabled because the marking $m(p0) = 2$, $m(p1) = 1$, $m(p2) = 2$ satisfies the flows $F(p0, t1) = 2$, $F(p1, t1) = 1$, and $F(p2, t1) = 1$. The transition $t2$ is not enabled because $m(p4) < F(p4, t2)$.

Figure 2.1b demonstrates the result of firing transition $t1$. The tokens from $\bullet t1 = \{p0, p2\}$ have been removed and added to $t1^\bullet = \{p3, p4\}$ according to the flow function F . After this transition, there are tokens in $p2, p4$ enabling transition $\bullet t2$.

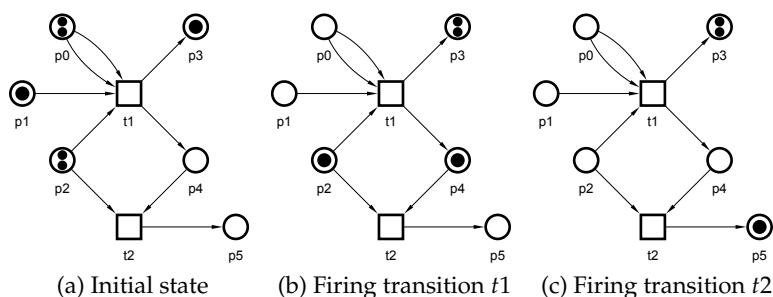


Figure 2.1: Petri net firing transition

2.3.3 Other PN Properties

Traces

A *trace* is a sequence of fired transitions in a Petri net; it is used to depict the path from its initial marking to any reachable marking. For instance, the trace from the marking in Figure 2.1a to the marking in Figure 2.1c is depicted with the trace $t_1 \rightarrow t_2$.

Boundedness The *K-bounded Petri nets* are the nets that for any reachable marking (any trace from the initial marking) have at most K token in any place: $\forall p \in P : m(p) \leq K$. Such Petri nets are interesting for the automated analysis tools because their reachable marking set is finite.

Most Petri nets presented in this thesis are 1-safe. A Petri net is called the *1-safe* when it is 1-bounded (for any reachable marking, $\forall p \in P : m(p) \leq 1$).

Deadlocks The *deadlock state* is a special PN marking, where none of the transitions is enabled. A Petri net is *deadlock-free*, if there are no reachable markings leading to the deadlock state.

2.3.4 Signal Transition Graphs

The idea to describe digital circuits with Petri nets was independently proposed in [17] and in [67], it extends the Petri net model with a function mapping PN transitions to particular circuit signal transitions.

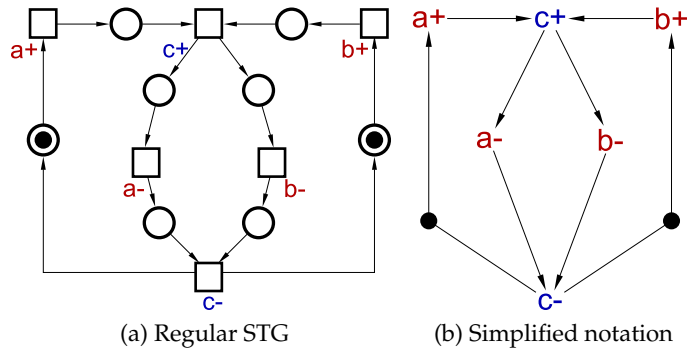


Figure 2.2: STG example of a C-element

Definition 2.4. *Signal Transition Graph*(STG) is a pair, $STG = (PN, L)$ where:

PN is a Petri net $PN = (P, T, F, m)$, and

L is a labelling function: $L : T \rightarrow (S \in C, op \in \{+, -, \sim\}) \cup dummy$

The labelling function associates events of the Petri net with circuit signal transitions. Transitions denoted with “+” are the raising signal transitions from “0” to “1”. Transitions denoted with “-” are the falling signal transitions from “1” to “0”. Transitions denoted with “~” are the toggle transitions switching signals from “0” to “1” or from “1” to “0” depending on the current value of the related signal.

Transitions that are not associated with any signals are called *dummy transitions*. Such transitions can be useful for shifting the marking of an STG without affecting states of the modelled signals.

Simplified Notation

Throughout this thesis the simplified notation of the STG models is used. This notation sometimes omits the “~” sign for the toggle transitions and the labelled transition boxes are replaced by labels. Finally, places in diagrams are often omitted when $|\bullet p| = |p\bullet| = 1$, which as a result makes the model more compact and easier to read.

Figures 2.2a and 2.2b present the STG of a 2-input *C-element*. The model presents two input signals a and b (shown in red), these signals form the environment of the model. The output signal c represents the output of the C-element (shown in blue).

The diagram shows that the output can raise or fall only after both inputs have

transitioned to 1 or to 0 correspondingly. In this example the arcs from c to a and from c to b introduce an assumption that the environment signals, once transitioned, would not change until the output signal has settled.

Complete State Coding

The *Complete State Coding* property holds when an STG does not have a *complete state coding conflict*.

The Complete State Coding conflict occurs when an STG has two reachable states in which the values of all the signals coincide but the sets of enabled output and internal transitions are different. In practice it means that these conflict states have ambiguous *next state* functions and cannot be directly implemented as a digital circuit, hence, the absence of CSC conflict is a necessary condition for a circuit to be synthesized.

Later in this chapter an example of CSC conflict will be considered as well as ways to resolve it.

2.4 Asynchronous Circuit Primitives

2.4.1 C-element

The *C-element* introduced by Muller [55] is a latched component that changes its value when all of its inputs make a single transition. The element depiction in diagrams is as shown in Figure 2.3. The basic 2-input C-element can be described by the equation $q = a \cdot b + (a + b) \cdot q$, or in the set/reset form:

$$q = \begin{cases} \uparrow & a \cdot b \\ \downarrow & \bar{a} \cdot \bar{b} \end{cases}$$

It means that the output of the gate is set when both inputs are active, it is reset when both inputs are inactive, and it stays *unchanged* otherwise. The 3-input C-element is correspondingly defined for three inputs: $q = a \cdot b \cdot c + (a + b + c) \cdot q$.

The *asymmetric C-element* has different sets of inputs for its set and reset functions. An

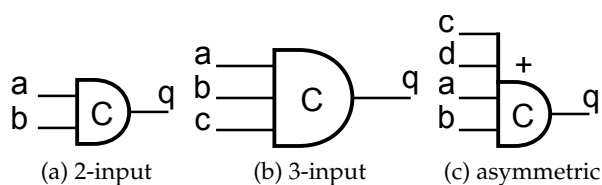


Figure 2.3: C-element examples

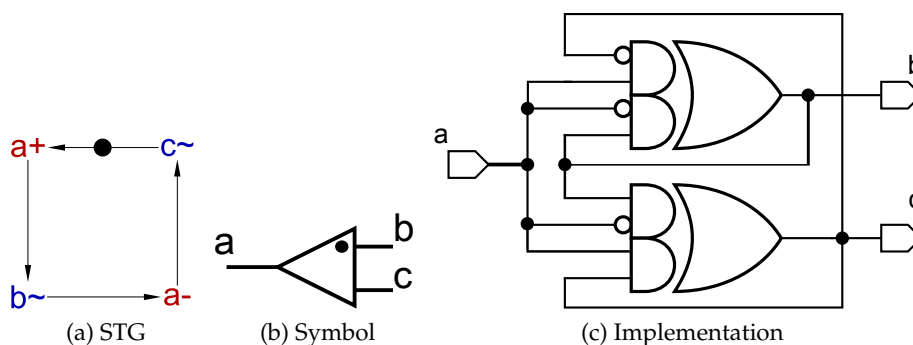


Figure 2.4: Toggle component

asymmetric C-element with all four inputs activating it and only two inputs deactivating it is shown in 2.3c. It is described by: $q = a \cdot b \cdot c \cdot d + (a + b) \cdot q$ or the set/reset notation:

$$q = \begin{cases} \uparrow & a \cdot b \cdot c \cdot d \\ \downarrow & \bar{a} \cdot \bar{b} \end{cases}$$

so it is reset when both $a = 0$ and $b = 0$, regardless of the c and d values.

2.4.2 Toggle Component

The *toggle component* is a latch that has one input a and two outputs b and c (Figure 2.45.1). It toggles b on every raising transition of a and toggles c on every falling transition of a . The initial state in this element can be either “0” or “1”. When not mentioned otherwise, the initial state of all the STGs with toggle transitions “~” is initiated with “0”.

2.4.3 Decision Wait Element

The decision-wait (DW) element (also known as the JOIN component [32]) is a latch with a number of rows and columns. The outputs are formed for each intersection between

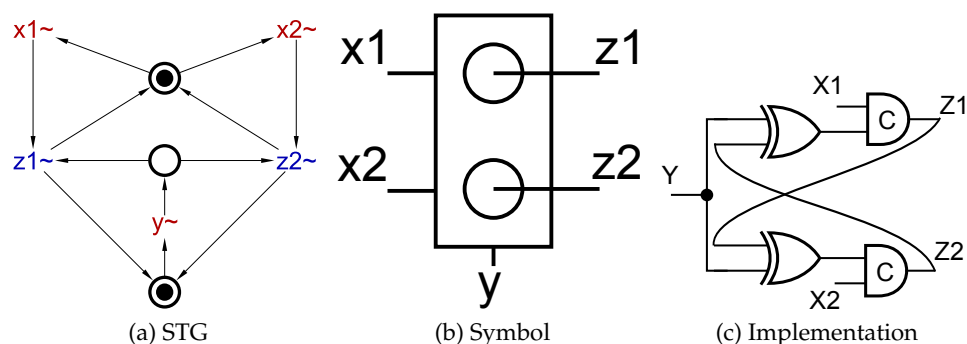


Figure 2.5: Decision-wait element

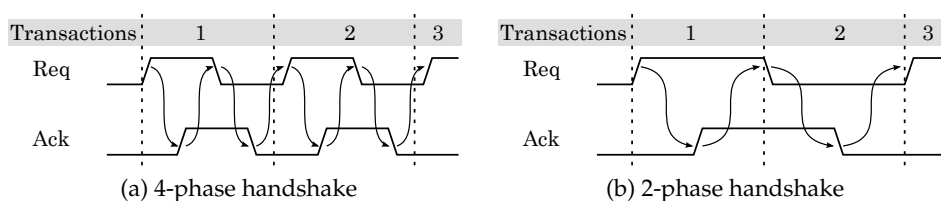


Figure 2.6: Basic handshake protocols

columns and rows. When there is a single change of one column and one row input, the corresponding intersection also toggles its value. An example of DW element with two rows and one column is shown in Figure 2.5.

2.5 Asynchronous Signalling

2.5.1 Handshake Protocols

Handshakes provide asynchronous systems flexibility of composing independent modules into communicating systems through well defined interfaces. A handshake is the two-way request-acknowledge communication transaction featuring the active participant initiating the handshake with a request for a service, and the passive side, responding to the initial request with the acknowledgement stating that the work requested has been done [75]. Depending on the signalling scheme used, a handshake can be either the 4-phase (Figure 2.6a) or the 2-phase (Figure 2.6b).

The *4-phase handshake* is related to the level-based signalling. It is also commonly known as a return-to-zero (RTZ) method. It forms the sequence of the following events:

Phase 1 $Req \uparrow$ – A request is issued, new communication cycle has started.

Phase 2 $Ack \uparrow$ – The request is acknowledged. The requester may proceed with the transaction.

Phase 3 $Req \downarrow$ – The requester is resetting to the initial state and waiting for the responder to reset as well.

Phase 4 $Ack \downarrow$ – The initial state of the handshake is restored, can process to Phase 1.

The *2-phase handshake* can be related to the transition or the pulse based signalling. It has only two phases per handshake:

Phase 1 Either $Req \uparrow$ or $Req \downarrow$ – A request is issued and the new communication cycle has been started.

Phase 2 Either $Ack \uparrow$ or $Ack \downarrow$ – The request is acknowledged, can proceed to Phase 1.

When both Req and Ack signals are used, the event detection on both sides can use these signals as a reference. If $Req \neq Ack$ the responder knows that there was an event on the request line. If $Req = Ack$ then the requester knows that the acknowledgement has been sent.

2.5.2 Channel Types

The type of the channel differs depending on which side of a handshake is transmitting the data. When the transmitter is making the initial request, it forms the so called *push channel* as data being sent is associated with the request event (Figure 2.7a). In contrast, the *pull channel* transmits data with the acknowledgement phase (Figure 2.7b).

In a more general view, the handshake with no data transition associated is called the *nonput channel*. It does not transport the data, but can still be used to synchronize the communicating modules. Finally, if both the transmitter and receiver attach some information to the request and acknowledge signals, such a communication is called the *biput channel* [75].

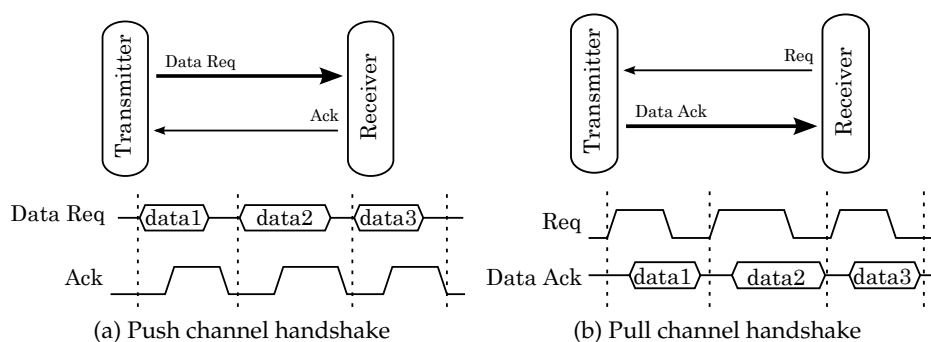


Figure 2.7: Channel types

Because of the variety of data transmission directions, it is convenient to call the participants the *active side* (or the master) initiating the request and the *passive side* (or the slave) responding to the initial request with the acknowledgement.

2.5.3 Delay-insensitive Encoding

Delay-insensitive codes [82] allow the signal propagation time for each separate wire and for each separate communication to be different. It is only required that within a finite amount of time each signal arrives to the destination.

Delay-insensitive techniques encode both the event (an indication that the input represents fresh and valid data) and the associated data into the same wires. Special *completion detection* circuitry is used at the data receiver side to identify when the data event has occurred.

2.5.4 Dual-rail

Dual-rail logic is the DI encoding that uses two separate wires ($x.false$ and $x.true$ or $x.0$ and $x.1$) to transmit zeros and ones. In a traditional 4-phase handshake only one wire can be active at a time. Hence, there are three allowed combinations (Table 2.1):

Activating one of the signals will effectively transmit either 0 or 1. Activating both signals is not allowed and when a sequence of data symbols is transmitted using a 4-phase protocol, they must be separated by *spacers* (a special signal state separating different consequent values).

Table 2.1: Dual rail 4-phase codes

$x.0$	$x.1$	meaning
0	0	spacer
1	0	send 0
0	1	send 1
1	1	not allowed

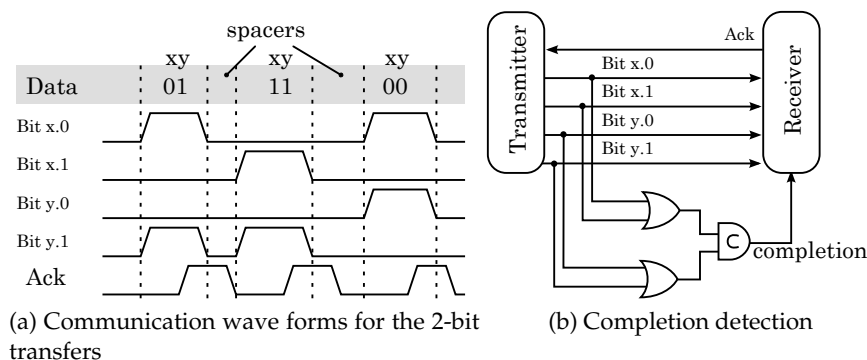


Figure 2.8: Dual-rail encoding

Multiple dual-rail pairs can be grouped to form a multi-bit channel. The n -bit value requires using $2n$ wires and can accommodate 2^n different codewords.

Figure 2.8a demonstrates an example of push channel communication for two bits using dual-rail codes. The completion detection is organised by using n 2-input OR gates. They detect a valid data on each of the bit channels. Then, all validity signals are combined into one completion signal using a *C-element* [70] (Figure 2.8b). This component is a latch that outputs the value of the inputs when these inputs match and preserves its value otherwise. In general, for the n -bit channel a tree of 2-input *C-elements* can be used.

Note that the concept of spacers does not always imply the use of 0 values on the dual-rail channel lines. The negative logic optimization might provide a better circuit by inverting one or both ($x.0$ and $x.1$) rails. Also, the dual-rail channel may have alternating spacer values, thus providing a more balanced power consumption in security aware applications [71].

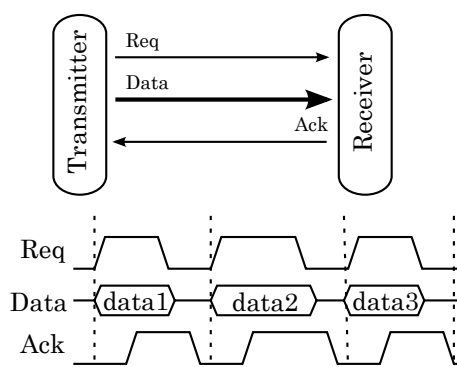


Figure 2.9: Bundled data, 4-phase push channel

2.5.5 Bundled data Encoding

The *bundled data* encoding is one of the most popular *delay-sensitive encoding* techniques. The term bundled data refers to a setup when the data symbol is encoded using a conventional binary number notation, and an additional event signal is used to signify when the data is valid. It means that the n -bit encoding requires only $n + 1$ wires. The timing assumption is applied here, stating that by the time the request signal approaches the receiver, all of the data signals will be stabilised and usable. As a result, the completion detection circuitry also becomes trivial – it needs to check what the request signal value is active (for the 4-phase communication).

Figure 2.9 demonstrates the data wires bundled with *Req* and forming the 4-phase push channel. The associated timing assumption is that the data wires manage to settle by the time the request transition reaches the receiver.

There may be various combinations of different handshake and channel types. For instance, the bundled data can also be 2-phase (e.g. to reduce latency and transition overheads in the channel) [77].

2.6 Example of Logic Synthesis Using Petrify

Consider an example of designing a simple 4-phase controller for an asynchronous link interface based on the logic synthesis technique. It is a structure of the converter transceiving messages (packets) from the on-chip to the off-chip link (Figure 2.10) proposed

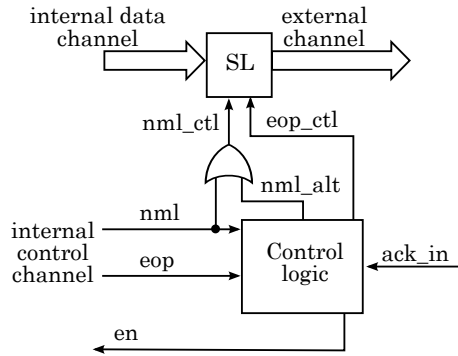


Figure 2.10: Channel converter structure

in [73]. Packets are formed of the data flits that arrive via the *internal data channel* pipeline into the Send Logic (*SL*) unit. To separate different packets, the *internal control channel* provides additional information about the data flit types. When a normal flit is sent, the *nml* signal is activated (*nml+*) by the control channel. Alternatively, when the last flit of the packet is sent, *eop* is activated (*eop+*).

The *SL* (Send Logic) unit reacts to the incoming requests *nml_ctl+* or *eop_ctl+*. For *nml_ctl+* it propagates the flit from the internal data channel to the *external channel* link. For *eop_ctl+* its task is to generate special “end-of-packet” flit and also send it across the off-chip link.

The Control Logic unit controls sending data over the external channel by activating either *nml_ctl+* or *eop_ctl+*. Then, following the 4-phase protocol, it resets the request and acknowledges the control channel with *en+*. For the normal flit, after *nml+* it only has to propagate the handshake: $nml+ \rightarrow ack_in+ \rightarrow en+$ and reset $nml- \rightarrow ack_in- \rightarrow en-$.

When the last flit activates *eop+*, the logic controller should, first, activate sending the flit via $nml_alt+ \rightarrow nml_ctl+ \rightarrow ack_in+$. Then, after the reset phase $nml_alt- \rightarrow nml_ctl- \rightarrow ack_in-$, it needs to send the “end-of-packet” flit and acknowledge the environment: $eop_ctl+ \rightarrow ack_in+ \rightarrow eop_ctl- \rightarrow ack_in-$. The whole logic is modelled using the STG shown in Figure 2.11a. A single token placed in the initial state *M0* enables two mutually exclusive transitions *eop+* and *nml+*. It is now possible to follow a trace of events. For instance, when *eop+* fires, the token will propagate to the arc $eop+ \rightarrow$

$M5$ is also eliminated, however, $M1$ is still in conflict with $M3$. To resolve this conflict, one can increase the number of signals used to encode the states. For instance, if a new signal csc is added as shown in Figure 2.11b, it will split the diagram into two regions, one with $csc = 0$ and the other with $csc = 1$. Then the main task is to achieve that the conflicting markings appear in different regions. In the presented case the new unique coding for $M1$ and $M3$ are: $M1 = 0100000$ and $M3 = 0100001$.

By running Petrify with the new model, it is possible to find the implementation for each of the signals, which is:

$$\begin{aligned} en &= ack_in \cdot (\overline{eop} + eop_ctl) \\ eop_ctl &= csc \cdot (eop_ctl + \overline{ack_in}) \\ nml_alt &= \overline{csc} \cdot eop \\ csc &= eop \cdot (csc + ack_in) \end{aligned}$$

The process of adding new signals and reducing concurrency can also be done automatically by Petrify. One may find that there are also other solutions resolving the conflicts without sacrificing concurrency, in the presented example it could be done by adding two additional signals instead of one. Whenever there is a choice as to which approach to use, there are certain trade-offs related, for example, to the performance of the overall system, including both the controller and the controlled logic. Concurrency reduction makes the controller implementation simpler, due to having more “don’t cares” in the circuit state space, but may sometimes degrade the system’s performance, depending on the mean value and distribution of the delays associated with the initially concurrent branches. On the other hand, the insertion of additional signals implies a more complex controller implementation, which may reduce the performance if the gains from the concurrent branches in the controlled circuit are relatively small. In the latter case the gains are absorbed by the delays introduced by the additional cells implementing csc signals.

Yet another powerful technique for resolving state encoding conflicts is based on applying *timing assumptions*. It can greatly simplify the implementation without the additional performance penalty. It is similar to concurrency reduction, however, the events are not specifically constrained by adding new arcs. Instead, it is assumed that certain events happen in a predefined order. For instance, it is known that the external link channel is slow, much slower than any gate in the control logic. This allows the events to be modelled as sequential, i.e. $nml+ \rightarrow ack_in+$ and $nml- \rightarrow ack_in-$. This, in turn, simplifies the implementation for the signal *en* because it does not have to sense the *nml* signal at all.

Chapter 3

Review on Asynchronous Arbiters

3.1 Introduction

The necessity for asynchronous arbitration can be found in applications where limited resources (service providers) are distributed among multiple clients (service users). Typical examples are shared communication channels, multi-port memories, shared data processing components, to name but a few. When a shared resource is not able to support multiple clients at a time, the arbitration phase is needed to avoid multiple clients accessing a resource concurrently. Instead of directly using the resource, clients communicate with the arbiter to gain the permission (or get arbiter grant) to use the resource. Correspondingly, the arbiter monitors whether the resource can be used and, when the resource becomes available, grants access for an incoming client request.

Synchronous arbiters operate in time domains different from asynchronous arbiters. The synchronous arbiters are clocked and only compute their decision based on stable input values and stable internal states. Issues related to their design are limited to the arbitration fairness and priority management, which is easy to address with traditional synchronous design techniques. The asynchronous arbiters activate with the incoming requests without explicit timing assumptions as to when these requests may arrive. In other words, the arbiter will try to grant a request arriving first in the continuous time domain. While it is easy for requests separated in time, the arbiter latch splitting these

requests may become metastable when two clients request the resource simultaneously. Because of this problem, the asynchronous arbiters have a dedicated interest among other topics of asynchronous circuits.

The problem of arbitration includes understanding the operations of typical arbiters and the set of rules governing the arbiter interaction with clients. The formal specification is needed to describe this behaviour and enable formal verification of the design. Convenient models for describing asynchronous arbiter functionality are the Petri nets and the Signal Transition Graphs. Petri nets can easily model concurrent events, non-deterministic choice, and unbounded component delay at various levels of abstraction. Each model capturing arbiter behaviour contains dedicated states denoting arbiter availability and the events igniting transitions between these states. The key arbitration transitions are the ones transferring the arbiter in and out of its critical section.

This chapter reviews a number of existing asynchronous arbiter solutions from the perspective of various arbitration properties such as communication rules, fairness, and scalability. It starts by presenting various simple two-way arbiters, and then progresses into the more complex problem of multi-way, multi-client, and multi-resource arbitration.

3.2 Arbiter-specific Properties

Communication Protocol

A communication protocol determines the rules by which clients request and release resources and the rules by which arbiters grant client access. The protocols are broadly split into one of the signalling schemes: either 4-phase signalling or 2-phase signalling.

The 4-phase signalling (also called the *return-to-zero* (RTZ) protocol) assumes that each of the communication signals transitions at least two times per single arbitration transaction. It initiates with all communication signals being equal to logical “0”, and is eventually reset back to the “all zero” state, before the next transaction is started.

The alternative 2-phase signalling (also called the non-return-to-zero (NRZ) protocol) does not include the phase of returning signals to the initial zero state. Hence, each new

arbitration transaction can be started by any transition on the client request line. This type of arbiter requires an additional 2-phase “done” signal, which tells the arbiter when the resource is released.

As will be shown, quite often the communication protocol of an arbiter can be identified by the presence of particular signals. For instance, the RG arbiters use two signals (request-grant) per client and support the 4-phase request-grant protocol. Alternatively, the 2-phase communication can be identified by the RGD (request-grant-done) signals, with the communication participants toggling their signal states to communicate.

Arbiter Channels

Arbiter channels are the two-way communication links connecting the arbiter with clients (and sometimes with resources). Usually these channels are the *nonput* channels. Each client always uses its own channel, which is independent from other clients, so that the potential conflicts between the clients were resolved within arbiter boundaries. In this review, the channel types do not vary within the scope of the same arbiter. However, it is also possible that some arbiters use mixed channel types for different clients.

Order of Arbitration

The *order of arbitration* identifies the number of resources that are shared and the number of clients that may need the resource. The most basic arbiter type manages two clients accessing one resource and form a particular range of two-way arbiters, hence, it presents a 1-of-2 arbitration problem (also called the two-way arbiter).

A more general arbitration problem is the 1-of- N arbitration (also called the multi-way arbiter). It is an extended arbitration problem supporting requests from N clients and inherits most of the design aspects that are related to the basic 1-of-2 arbitration such as signalling scheme and the signals used for communication.

Similarly, an arbiter may manage multiple resources distributed among clients. For N clients and M resources it extends the allocation task to the M -of- N arbitration, which also inherits concepts from the 1-of- N arbiters, such as scalability and fairness.

Fairness

Fairness is an important property applicable for any arbiter with two or more *pending requests*. Any request arriving at the arbiter while the resource is occupied becomes a pending request. When there are multiple pending requests, the additional arbitration may be needed to decide which of the pending requests will be the next assigned the resource. Different arbiters may have alternative policies regarding this conflict. The most popular policies are the *fair* and the *priority-based* arbitration.

The arbiter is considered fair, if each of client requests is guaranteed to receive a grant after a limited number of grants issued to other clients. Or, in other words, an infinite number of client requests will result in infinite number of grants.

The priority arbiter favours the “most important” request first. Hence, the discriminated clients may sometimes starve as a result.

The *statistical fairness* of an arbiter is another type of fairness sometimes considered by designers. It estimates grant probability distribution for even distribution of requests.

Scalability

Arbiter scaling considers the relation of arbiter costs in power, area, latency, interconnect complexity, or throughput against the number of channels communicating with clients. These circuit properties are governed by arbiter internal organization and have different optimal implementations depending on how many clients need to be supported.

Analogue vs Digital

Every asynchronous arbiter has to contain metastability. *Analogue arbiters* are modelled at the level of transistors and include metastability filters that deal with non-digital signal levels. They are more difficult to design and scale, but, when used right, provide great performance benefits.

Digital arbiters are constructed of usual logic gates and use analogue arbiters as basic building blocks to contain metastability. In digital arbiter implementations all of the signals are always binary, either “0” or “1”. The metastable values are hidden within the

analogue arbiter components and do not have to be considered in digital circuit models. The main advantages of the digital arbiters are their scalability and flexibility in solving arbitration conflicts of high complexity.

Topology

The notion of topology is applicable for the scalable arbiters composed of a certain number of *arbitration cells*. It specifies how the cells are connected together when the arbiter is scaled. The commonly known topologies are ring, tree, or mesh.

3.3 Metastability

The asynchronous nature of an arbiter assumes that its inputs can be changed at any time, even when the circuit's internal state is not stable.

For a simple 4-phase two-way arbiter with request channels r_1, r_2 and grant channels g_1, g_2 the functionality can be specified with the following equations: $g_1 = r_1 \cdot \overline{g_2}$ and $g_2 = r_2 \cdot \overline{g_1}$ (Figure 3.1a).

The arbiter always grants the first client request by lowering the corresponding grant signal when the client issues a request, and the resource is not granted the second client.

The problem occurs when two clients request it at the same time. Because every gate has certain speed limitations, there will be a period of time during which the output is already greater than logical "0" but is still less than logical "1". When two request signals arrive within this time interval, both gates start changing their outputs and mutually try to lock out their neighbouring gate. As a result, the half-way blocked gates will be outputting metastable values for a prolonged period of time (Figure 3.1b).

The occurrence of metastability is rare in reality; however, theoretically, its duration is unbounded and can be arbitrarily long. By the time the metastability is resolved, the metastable output can be misinterpreted by the subsequent gates and create a hazard propagating through the circuit, hence, special mutual exclusion circuit is required.

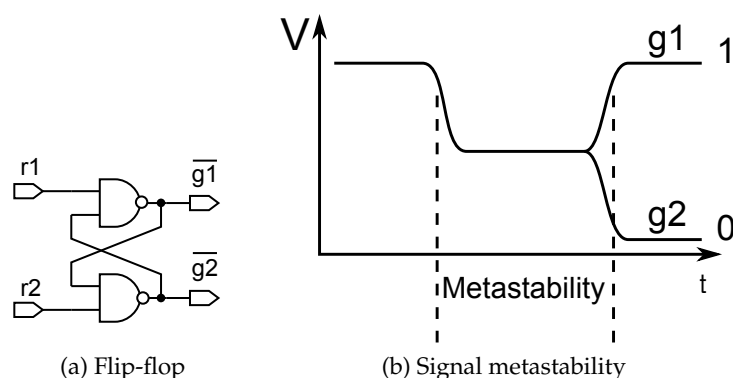


Figure 3.1: Naive arbiter implementation and metastability

3.4 Analogue arbiters

3.4.1 The MUTEX Element

The problem of metastability in arbiters was noticed long ago. An attempt was made to find a solution of the mutual exclusion arbitration based on standard gates [63]. However, it was later realised that the circuit resolving metastability has to take into account the non-digital nature of the signals [36, 14]. Later, the MUTEX component was implemented using threshold filters by S. Patil in [60]. The faster implementation based on the NMOS *analogue difference circuit* was proposed by C. Seitz in [68]. Later, this component was refined for the CMOS technology by A. Martin in [47]. It consists of an SR-latch followed by the metastability filter, which hides the metastable values until one of the gates wins the arbitration. The component is now known as the MUTEX element (Figure 3.2). It is actively used as a basic building block when designing more complex arbiters. Internally the MUTEX component manages analogue signal values, while externally it behaves as a digital component with an unbounded latency.

In practice, if the asynchronous design is only allowed to use standard library cells, the MUTEX component can be built of the cross-coupled NAND gates followed by the low-threshold inverters (Figure 3.3). It is also a useful technique for prototyping cir-

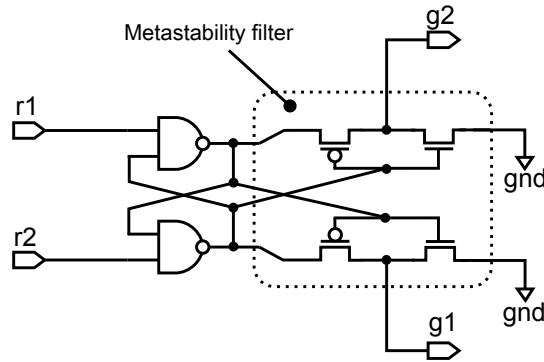


Figure 3.2: MUTEX element [47]

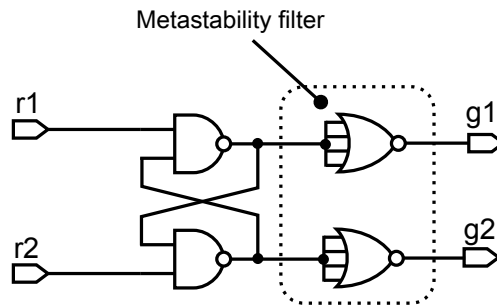


Figure 3.3: MUTEX with standard gates [35]

circuits in *Field Programmable Gate Array* (FPGA) structures; however, the drawback is the increased latency of the component.

3.4.2 Analogue 1-of-3 arbiter

The tri-flop is a generalization of the MUTEX element; it supports three clients accessing a single resource (Figure 3.4a). In other words, it is a 4-phase implementation of the 1-of-3 arbitration conflict.

The implementation is straightforward with the cross-coupling NAND gates followed by the metastability filter (Figure 3.4b). When all of the n inputs are held at logic “0”, the gate outputs x_1, x_2, x_3 are equal to “1”. When one of the requests arrives, the corresponding x signal falls, blocking the other gates and raising the corresponding output grant signal g to “1”.

The detailed analysis of this arbiter has shown that multiple request signals r_1, r_2, r_3 arriving at the same time may cause the *ternary metastability* with oscillating voltage

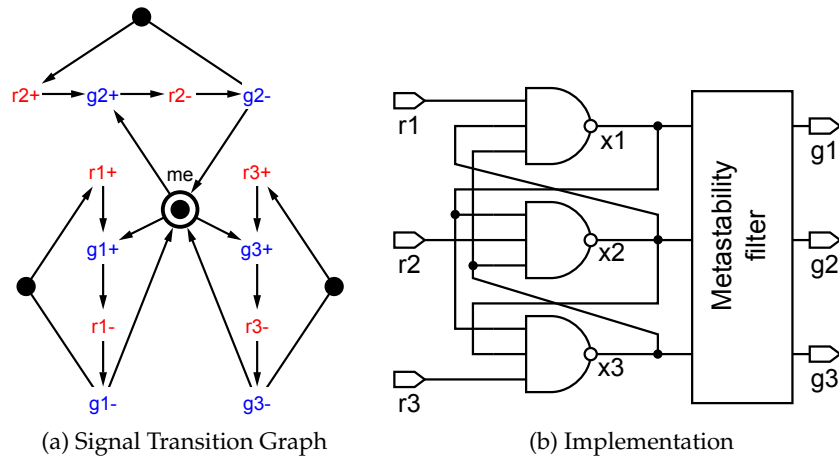


Figure 3.4: Tri-flop arbiter

on the x outputs [10, 43]. Additionally, it was noted that this arbiter does not have pre-defined behaviour when two requests are pending. Right after the resource is released, the pending requests are in danger of starting another metastability. The resolution to this conflict will not be fair as the outcome will likely depend on the transistor threshold values, which will always favour one of the clients.

Since the arbiter tries to provide the grant in one computation step, its main advantage is low latency; however, its drawbacks of the oscillatory response, unfairness and complicated design process often lead designers to use other 1-of-3 implementations.

3.4.3 Analogue 2-of-3 Arbiter

A small extension to the tri-flop was proposed in [10] (Figure 3.5) where it implements the 2-of-3 allocation with the arbiter granting two first client requests. As noted in the paper, this arbiter does not suffer the oscillatory behaviour and there is no more than one pending request at a time. Hence, this solution presents a highly efficient 2-of-3 arbiter implementation.

3.5 Two-way Arbiters

The digital two-way arbiters assume there are only two clients that share a single resource. This section considers various communication protocols and does not consider

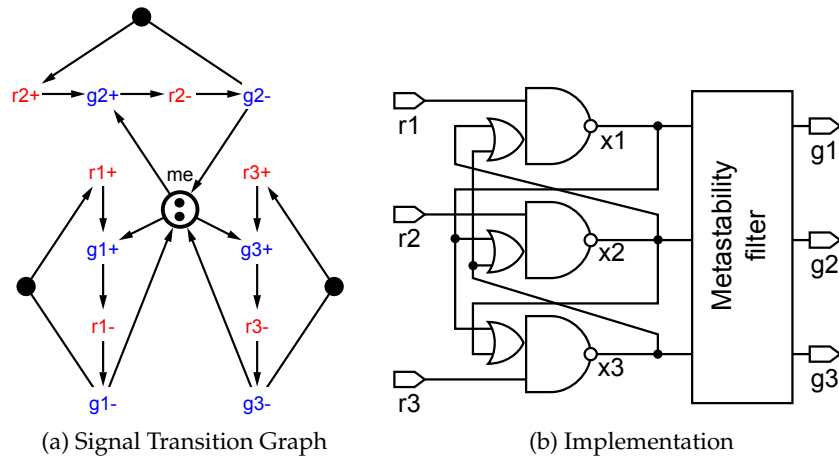


Figure 3.5: 2-of-3 analogue arbiter

arbiter scaling.

3.5.1 4-phase Arbitration (RG)

The 4-phase arbiter is based on *4-phase handshakes*. The protocol for two inputs can be regarded as the simplest type of arbiter, which is implementable as a single MUTEX element presented before. The protocol behaviour is defined with the STG diagram shown on Figure 3.6.

When client 1 requests the resource, the event $r1+$ occurs. Eventually, the arbiter responds with $g1+$ if the second client is not using the resource and the me place has a token. While the token is on the path $g1+ \rightarrow r1- \rightarrow g1-$, the arbiter will prevent the second client from using the resource. The transition $g1+$ grants the resource to the client 1 may safely be used. To release the resource when it is no longer required, the client issues transition $r1-$. This is subsequently followed by $g1-$, enabling the resource to be granted to other clients. To obey the 4-phase communication, client 1 must wait until $g1-$ takes place, and only then it is allowed to issue the next request $r1+$ which would start a new arbitration transaction.

Functionally, MUTEX implements the E. W. Dijkstra’s binary semaphore with its “wait” state located on the arcs $r_i+ \rightarrow g_i+$ and the “signal” state located on the arcs $r_i- \rightarrow g_i-$. As it can be seen, the “wait” state blocks the client until the me place receives a token. The “signal” state is not blocked by anything and immediately returns

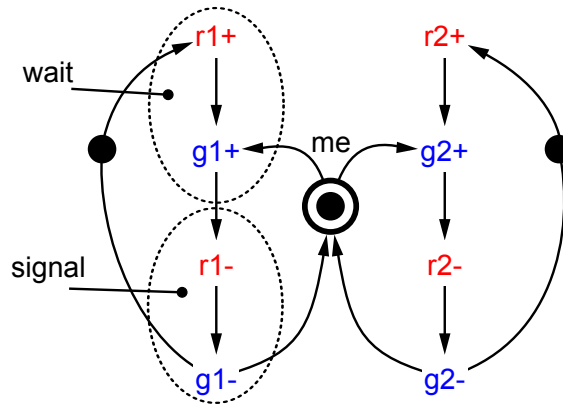


Figure 3.6: 4-phase two-way arbiter

the resource.

3.5.2 2-phase Arbitration (RGD)

The 2-phase arbiter is a binary semaphore implementation based on the 2-phase handshake communication, which was proposed by I. Sutherland in [77]. The arbiter has two inputs and one output for each individual client. According to the signals used, it is called the RGD (Request–Grant–Done) arbiter denoting three signal transitions required per one arbitration for each client.

First, a request for arbitration is sent via one of the request lines r_j . Eventually it is followed by the transition of g_j acknowledging that the resource was granted. After the resource was used, the client transitions d_j which finalizes the transaction. There is no explicit reset phase and every transaction will be started with the alternating signal values.

The implementation of the 2-phase semaphore consists of two XOR gates, two toggle components, and two C-elements. The STG description is simplified to only three toggle transitions per client.

3.5.3 “Nacking” Arbiter

The “nacking” arbiter is designed to avoid client waiting [57, 20]. It always responds with either “acknowledged” when the resource is free or with the “not acknowledged”

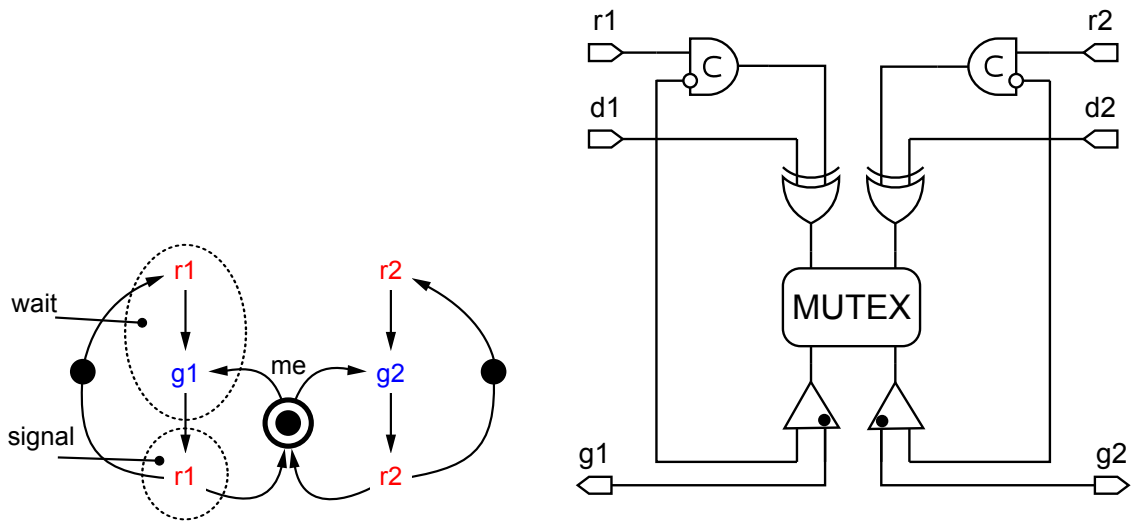


Figure 3.7: 2-phase two-way arbiter

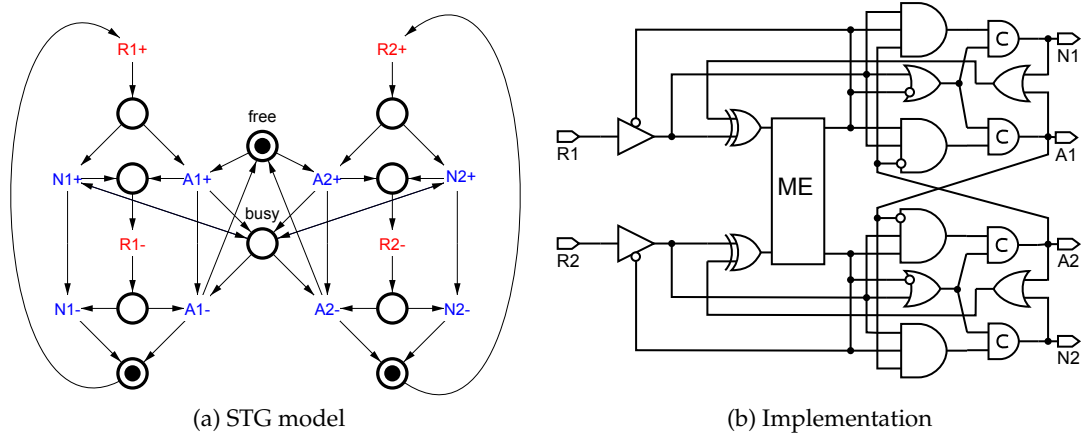


Figure 3.8: Nacking arbiter

when the resource is not available. Such an immediate response prevents the client from waiting and allows it to perform other useful tasks, perhaps repeating the request at a later time.

The arbiter STG is presented on Figure 3.8a. As the requests arrive, they always propagate to either granting response signals $A1/A2$ or the nacking responses $N1/N2$. This selection of the response is based on whether the token is located in the *free* or *busy* place of the arbiter.

There are three conflicts possible:

- $A1+$ with $A2+$ mutually disable each other (the regular arbitration between two

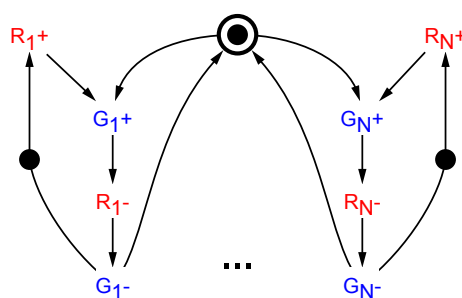


Figure 3.9: Multi-input arbiter

concurrent requests);

- $A2-$ with $N1+$ or $A1-$ with $N2+$ (the arbiter does not know whether to “nack” a request, that arrives simultaneously with the resource being released and explicit arbitration is needed to make this decision).

Luckily, all these conflicts never occur concurrently. This means that a single MUTEX element is sufficient to resolve the conflict. When the MUTEX element is used for the first time, the feedback loop propagates the signal back which, XOR-ed with the initial request, releases MUTEX for the next arbitration (Fig. 3.8b).

3.6 1-of- N Multi-way Arbiters

From a more general view, the problem of 1-of-2 arbitration can be extended to a greater number of clients requiring access to a common resource, posing the 1-of- N arbitration problem (Figure 1-of- N Multi-way Arbiters). All of the 1-of- N arbiters still support the 1-of-2 arbitration, but their structure allows sharing a resource among the increasing number of clients.

The multi-way arbiters can be built by combining the *arbitration cells* (or tiles) that are treated as the basic building blocks and are simply reused when more clients are needed. The evaluation of a particular implementation can be done by estimating the parameters such as area, latency, power, e.t.c., against the increasing number of clients. This section provides an insight on various 1-of- N arbiter types and the common topologies used.

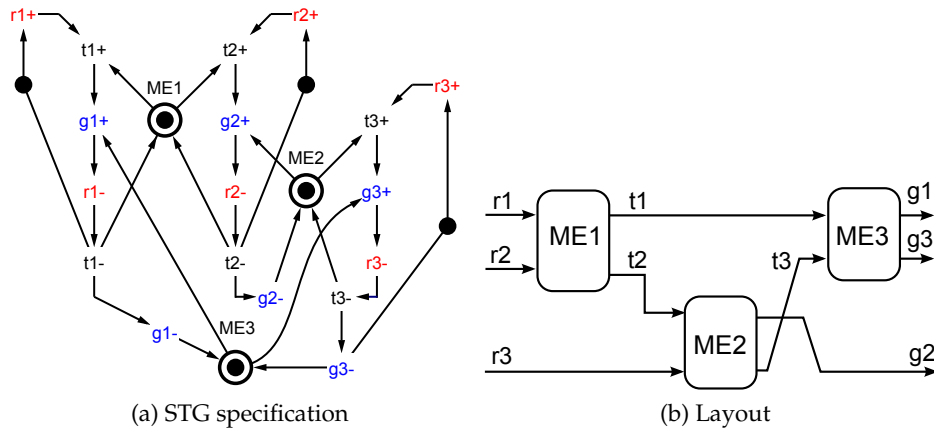


Figure 3.10: 1-of-3 mesh

3.6.1 Mesh-based Implementation

One way of arbitrating N clients is by interleaving multiple 2-input MUX elements in a mesh-like structure. An example of a 1-of-3 arbiter is shown in Figure 3.10. It arbitrates three inputs by arbitrating each pair of clients. The result is that only one grant signal will propagate at a time.

While the resource is occupied by one of the clients, the requests from the two other clients may become pending, which raises the question about arbiter fairness. Each time a client releases its request, it unblocks the propagation of two other client requests, hence, the resource will be granted to each of the clients after a finite number of resource grants.

The mesh-based arbiter can be extended to support N requests. A convenient layout without over-crossing interconnects is shown in Figure 3.11. With the increasing N its response latency increases linearly. It is a reasonably good performance in comparison with other topologies. Statistically, it is even better, because the implementation of each arbiter cell is so simple (just the MUX elements). The significant drawback of this arbiter is the quadratic growth of its area, rendering it impractical for the large number of clients:

$$C(N) = \frac{N(N-1)}{2}$$

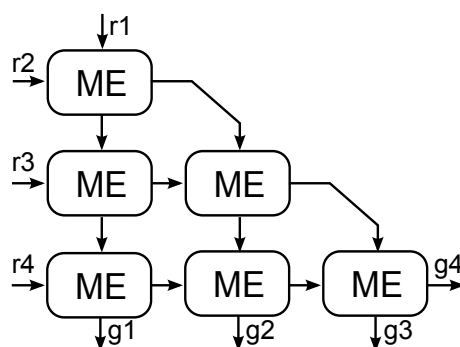


Figure 3.11: Generic mesh structure

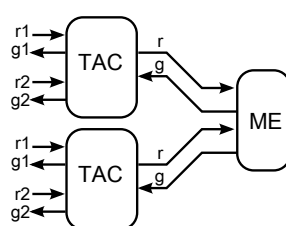


Figure 3.12: Tree structure

3.6.2 Cascaded Tree Arbiters

The cascaded *tree arbiter cells* (TAC) can be connected together in a tree-like structure, where each cell has two channels communicating with clients $r1/g1$, $r2/g2$ and the outgoing channel r/g communicating with the next level or arbitration.

When either of the requests wins the local arbitration, the outgoing signal r propagates the request further. Depending on the total number of clients, there may be multiple layers of arbitration, where the signal is propagated in a similar fashion until it reaches the root of the tree. At the top cell a simple MUTEX element may be used to arbitrate between the two root branches.

As the root node responds with a grant signal, the acknowledgement propagates back to the only client that has won all the arbitrations.

The optimal structure of the arbiter assumes a balanced tree with the total number of clients being $N = 2^x$. It is easy to see that each additional TAC component increases the total number of supported clients by 1: $C(N) = N - 1$ rendering the area increase linear.

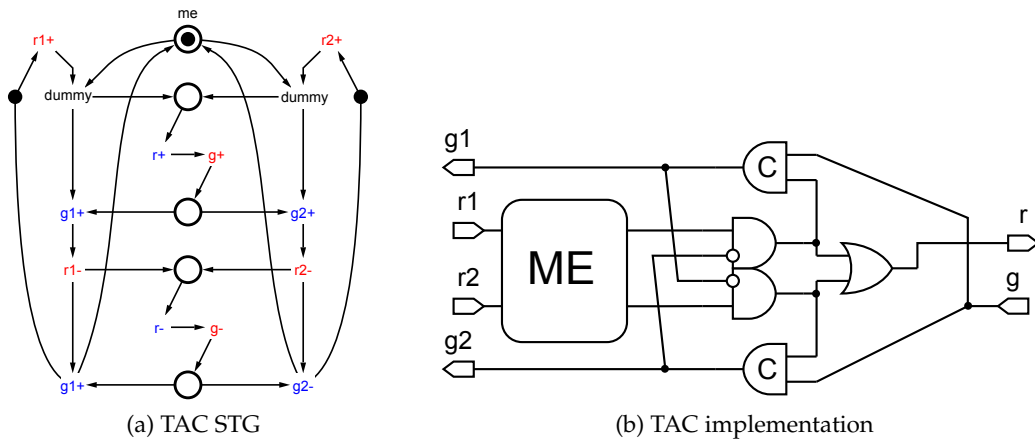


Figure 3.13: Cascaded arbiter STG and implementation

For comparison, the 1-of-8 tree arbiter needs seven cells in total, while the mesh arbiter would already require 28 MUTEX components.

The response time of the arbiter for a balanced tree is dependent on the number of layers, which is:

$$L(N) = t \cdot \log_2 N = t \cdot \log_2 2^x = t \cdot x$$

This means that the arbiter has a very good latency overhead for the increased number of clients. At the same time, the propagation of requests has to pass the distance until the root of the tree and back to the branch, which in comparison with other topologies is a significant overhead for low numbers of clients.

It can also be shown that this arbiter is fair because each layer of arbitration will be interleaving request grants. At each level of arbitration, there will be an equal distribution of grants for each of the branches, and the bottom level the client grants will be also evenly distributed.

3.6.3 Token Ring

Arbiters can also be organized in a ring-like structure. A chain of cells would have neighbouring cells from its left and right sides (the next cell and the previous cell). A *privilege*

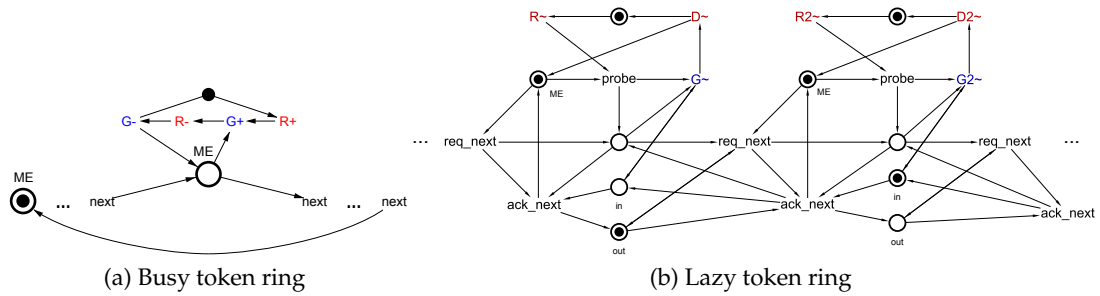


Figure 3.14: Token ring high-level models

*token*¹ is passed between the cells in one direction. Once the token is inside the arbitration cell, it can be captured by the associated client. One token in the network ensures that only a single client can capture it, effectively resolving the 1-of- N allocation.

High-level Petri net specifications show basic behaviour of the cells (Figure 3.14). Each cell creates an access point to one additional client, hence, the area of implementation increases linearly and N request channels can be implemented by using $C(N) = N$ arbitration cells. It is easy to see that the fairness is guaranteed because each resource release always brings the token closer to each of the pending requests.

Depending on how the arbiter works, it can be regarded as “busy” or “lazy” token ring. The busy token ring is fairly simple, each cell constantly passes the token to its next neighbour as long as there is no associated client request. The drawback of the arbiter is the constant activity causing dynamic power consumption while there are no active client requests.

The lazy token rings propagate the token only when there is demand from a pending request. The idea is that a request from a client keeps propagating through other cells until it reaches the cell holding the token. At that point, by means of handshake communication, the request “grabs” the token and pulls it back into its own cell. Once the token is in the cell, the resource is used until it is no longer needed. Once released, other cells around the loop can grab it again and drag it away.

The 4-phase busy ring and the 2-phase lazy ring implementations are shown in Figure 3.15. The advantage of the first implementation is in its simplicity and low latency.

¹Here the “privilege token” is referred to the abstract privilege token and not necessary the Petri net token identifying a state of the model.

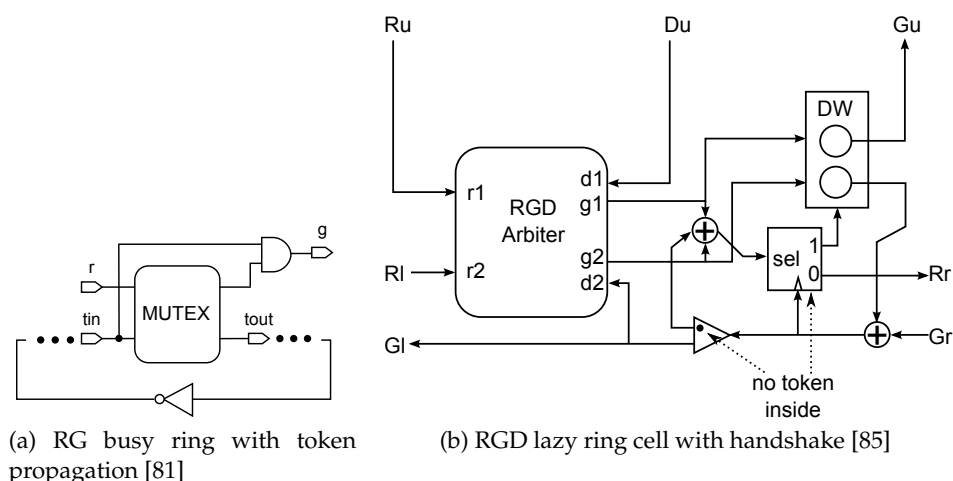


Figure 3.15: Ring implementations

The advantage of the second circuit is the low power consumption since it does not do anything until a client request arrives.

Each ring cell implements two channels of communication. The first channel grants the token to the associated client, the second channel propagates the token around the loop. The propagation around the loop can be done in two styles, either a single signal is being pushed through all the cells and then negated as in Figure 3.15a; or an explicit handshake (4-phase or 2-phase) can take place between the consequent arbitration cells.

As for latency estimation, the average response time is dependent on the total number of cells and the speed at which the token is being passed from one cell to another. If the token propagates one cell within time L_{Cell} , then in worst case it needs to pass $N - 1$ cells making the overall response time linear: $L(N) = L_{Cell} \cdot (N - 1)$.

Finally, it should also be noted that because the token propagation on Figure 3.15a creates slightly unbalanced chances for each of the clients. As the falling edge propagation takes roughly half the time of the full propagation cycle, clients located closer to the inverter will have greater chances of receiving the resource.

3.6.4 Ordered Arbiters

It was shown in previous arbiters that the order of grants for pending requests can vary depending on implementation. The *ordered arbiters* proposed by A. Bystrov [12] are de-

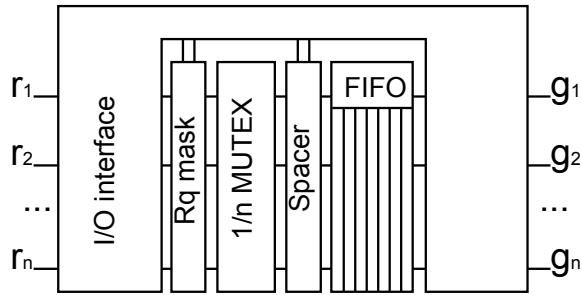


Figure 3.16: Ordered FIFO arbiter structure

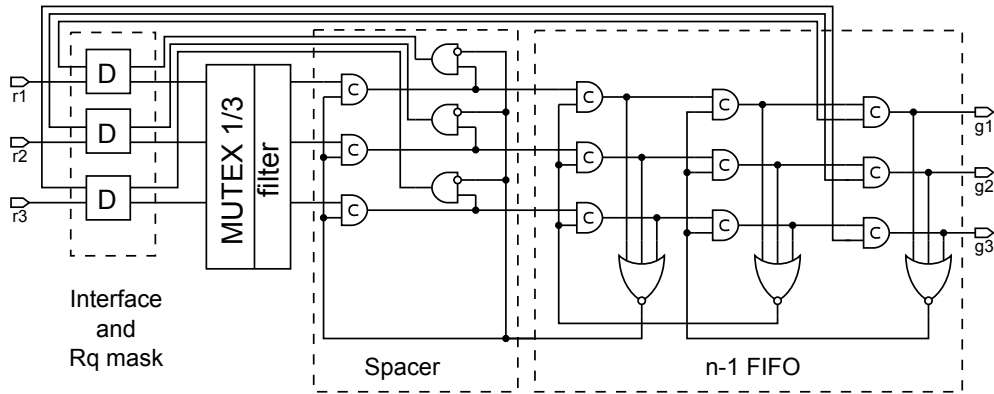


Figure 3.17: Ordered 3-way arbiter implementation

signed to preserve the order of incoming pending requests. The basic idea is to free the arbitration element as soon as possible, so that it arbitrates the next requests while the resource is in use by the first client.

The storage of won arbitrations is organized through FIFO (First In First Out) component as shown in Figure 3.16. Once the resource is released, the next client in queue immediately receives the grant.

The three-way ordered arbiter is shown on Figure 3.17. The interface and masking functionality is performed by the D-elements [45]. The scaling of this arbiter is quadratic as the pipeline size is increased both in depth and in breath. The arbiter is fair because it repeats the sequence of requests in the same order.

3.6.5 Priority Arbiters

Priority arbiters radically differ in the way they treat pending requests. Based on fixed or dynamically set priorities, certain clients are discriminated making this arbiter delib-

erately “unfair”. Such arbiters are used in the asynchronous NoCs in order to support prioritized traffic, which ensures the guaranteed latency for data delivery [22].

Static Priority Arbiter

The *static priority arbiter*, as the name suggests, uses statically set priorities in its granting logic. Its idea was mostly developed by A. Bystrov in [11] while a variation based on additional timing assumptions can be found in [24].

The arbiter splits the arbitration into two stages. At the first stage, it waits for one or more requests propagating through an array of set-dominant latches (Figure 3.18). When at least one request is present, the *lock* signal formed by the reset-dominant latch locks the column of MUTEX elements inside the lock register. Then the second stage of arbitration begins. The locked MUTEX elements form the 4-phase dual-rail input for the *priority module*, where the final arbitration decision is made based on the active inputs and the specified priorities. Three propagation scenarios are possible for the requests:

- The request has won MUTEX arbitration and is of the highest priority → the priority module issues the grant to the corresponding output port.
- The request has won MUTEX arbitration, but there is another signal with higher priority → the request is ignored for the time being and is reassessed at the next arbitration.
- The request did not manage to pass the arbitration, or the client did not issue the request → regardless of its priority, this request has no effect on the outputs until the next transaction.

As the priority module has finished its computation, one of the grant signals is issued and eventually the falling edge of the *lock* signal unlocks all of the MUTEX elements. At this stage the arbiter will only wait and accumulate new pending requests as they arrive.

As the granted client has finished using the resource, its request signal is reset. This also releases the priority module and results in the lock signal being unblocked for the next arbitration transaction. If the time of using the resource is large, multiple pending

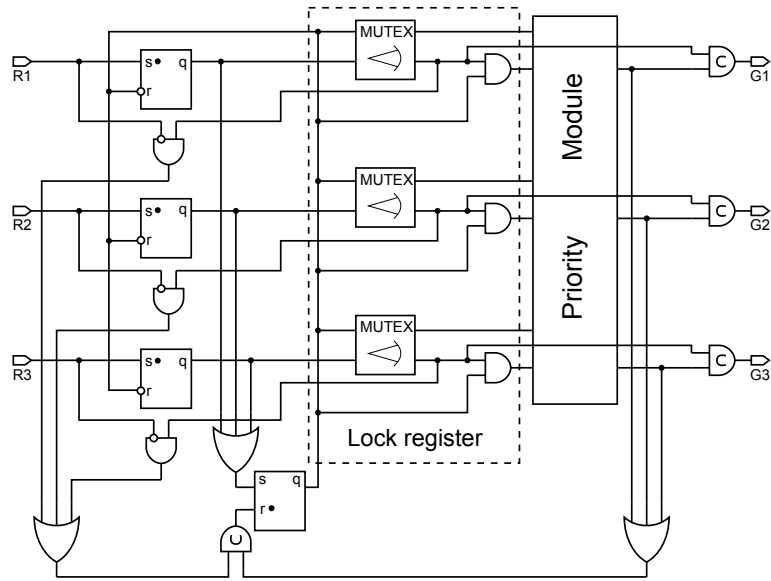


Figure 3.18: Three-way static priority arbiter

requests may occur, eventually resulting in the sequence of grants issued in order of their priorities.

For each additional client the arbiter structure remains mostly flat with slight overhead of the latency in the growing OR gates and the additional computational overhead in the priority module. This makes the arbiter particularly fast and scalable when dealing with large number of inputs.

The complexity for each new client grows linearly $C(N) = N$ (as it was shown in [11], the priority module is implementable as a tree with the logarithmic latency increase). It means, that such implementation can be easily scaled and adapted to a large number of clients.

Dynamic Priority Arbiter

The *dynamic priority arbiter* (DPA) has a similar structure to the static priority arbiter (Figure 3.19). The difference is that this arbiter uses additional data lines P_0, \dots, P_7 providing the priority values for each of the requests. The priority module compares these values in order to determine the highest priority request. Hence, an arbitrary priority function can be implemented.

The more detailed discussion of the dynamic priority arbiter can be found in [11].

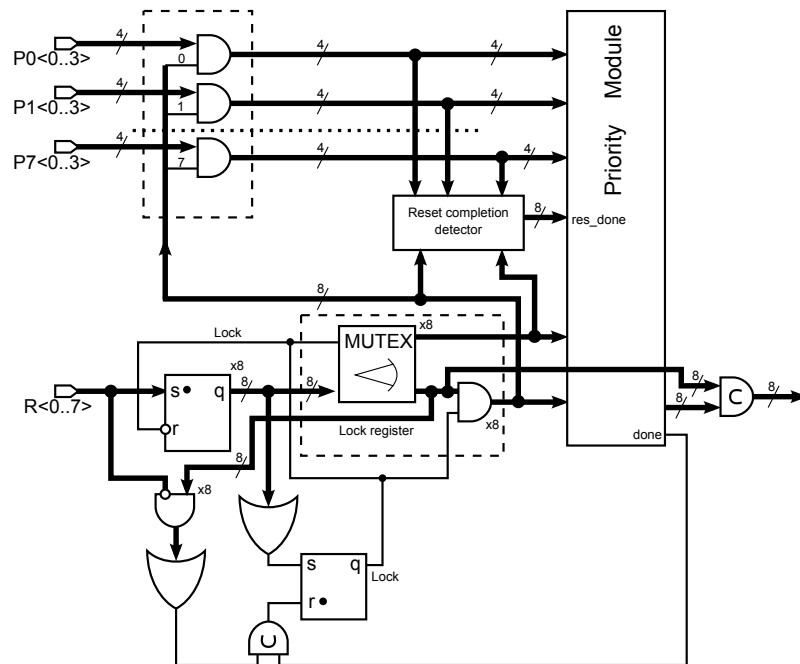


Figure 3.19: Eight-ways dynamic priority arbiter

However, it is worth noting that the arbiter has similar scaling characteristics with the SPA implementation.

3.7 Multi-resource Arbiters

The concept of multi-resource arbitration assumes there are multiple resources that need to be allocated among a number of clients. In addition to the fairness and scalability of the multi-way arbiters, for certain multi-resource designs it makes the conflict double-sided since each client can only be served by one resource at a time. This section considers the designs using multi-resource allocation.

3.7.1 Multi-token Arbiters

The multi-token arbiter is needed when there is a resource supporting more than one client at a time. The shared resource with capacity M (or M tokens) permits concurrent use by a maximum of M clients. For instance, an arbiter managing three clients with the capacity of two would be able to give grant signal to the first two client requests while

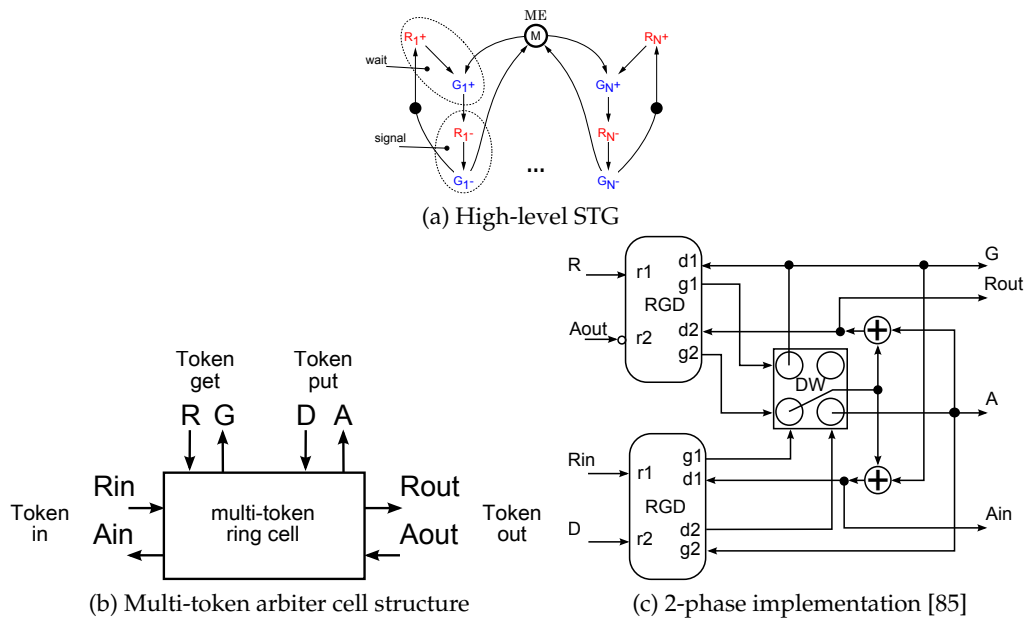


Figure 3.20: Multi-token arbiters

the third request would become pending until one of the first two clients releases the resource. On the high-level STG diagram in Figure 3.20a this behaviour is represented by placing M tokens into the mutual exclusion place ME . As a request is granted, it decreases the token count by one and when there are no tokens left, any new client requests become pending. This is a more general example of Dijkstra’s semaphore, decreasing its counter while value is greater than “0”.

Such a behaviour changes the concept of resource availability because more than one client is able to use it at a time. From a practical perspective, this arbiter creates an artificial bottleneck of performance when the resource is in danger of being overloaded. It helps balance out irregularities of the bursty request environment by spreading the incoming load over time.

A ring-based topology can be used to construct this arbiter out of ring cells as shown in Figure 3.20b. The movement of the token is ensured through four communication channels. The “token in” and “token out” channels receive the tokens from previous cell and send it to the next one. The channels “token get” and “token put” capture the tokens for use by a client, and inject used tokens back into the loop. This arbiter is similar to the token-ring arbiter presented before. However, because there are more than one token in

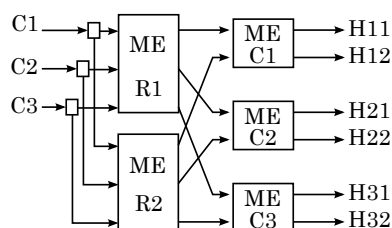


Figure 3.21: Multi-resource “Forward acting” arbiter

the loop, two different conflicts may occur: the conflict between the channels “token out” and “token get” competing for the token, and the conflict between channels “token in” and “token put” competing for the right to transmit their tokens through the “token out” channel. The implementation shown in Figure 3.20c contains two arbiters dealing with these conflicts.

The arbiter is fair because each pending client request eventually receives a resource. The latency is improved in comparison with the token-ring arbiter, as the greater number of tokens reduces the worst waiting time. The arbiter area scales linearly with the increasing number of clients: $C(N) = N$.

3.7.2 Patil’s Arbiter

The problem of multi-token arbitration can be extended to the problem of multi-resource arbiter or the tokens with identifiers. It is a problem of M -of- N arbitration with N clients, M identifiable resources, and $M \times N$ different grant signals. It has a double-sided conflict, where multiple available resources mutually compete for a client, which means that an arbitration among resources is also needed.

The solution was first found by Patil in [59] with the so called “forward acting” arbiter. The structure of such an arbiter for 3 clients and 2 resources is presented in Figure 3.21 where the arrows connecting components are the two-way communication links. It consists of two columns of arbitration logic, the first column contains M 1-of- N arbiters representing the resources, the second column contains N 1-of- M arbiters representing the clients.

The conflict is resolved in two columns of arbitration. When a client i requests a resource, the client broadcasts its request to all of the resource arbiters in the first column.

Each resource arbiter correspondingly grants the request and propagates it to the next column. If a resource is not available at this point it simply waits until it becomes available, or receives the “nack” signal from the i -th client arbiter in the next column. The i -th client arbiter waits for the first request from the resource arbiter j , and then propagates the “nack” response to all of the resource arbiters apart from the j -th arbiter. In the resource arbiter column, this “nack” signal forces “grants” for the client i and also releases the resources for other client requests. After receiving all M requests, the client arbiter issues a grant on the channel H_{ij} .

The arbiter latency depends of the latency of the 1-of- N and 1-of- M arbiter cells. Based on the experience from existing scalable arbiter implementations, it can be either linear (as in ring arbiters) or logarithmic (as in priority arbiters). Similarly, fairness of the arbiter is dependent on the fairness of each individual arbiter cell, which can be either fair or priority based. The area, however, grows quadratically because M 1-of- N arbiters and N 1-of- M arbiters in total are needed: $C(N, M) = N \cdot M$.

3.7.3 Committee arbiter

The *committee arbiter* described in [9, 13] is useful for solving a large variety of resource allocation problems, including the multi-resource arbitration. Informally, the idea of committee arbitration states that there are professors organising themselves into committees. Each professor can be a member of one or more committees and each committee is associated with one group of professors (Figure 3.22). The main condition for starting a committee meeting is the presence of all professors in the group associated with a committee. When multiple committee groups of professors are available, an arbitrary group may start the committee meeting. Correspondingly, while the professors are occupied in one committee, they are not available for any other committee to start.

It is possible to map existing allocation tasks to the committee arbiter. A generic 1-of- N arbiter corresponds to a single professor that attends any of the N committees on schedule. While the professor is busy with one committee, other committees will not start during that time. Another example is the priority arbiter. It also features only one

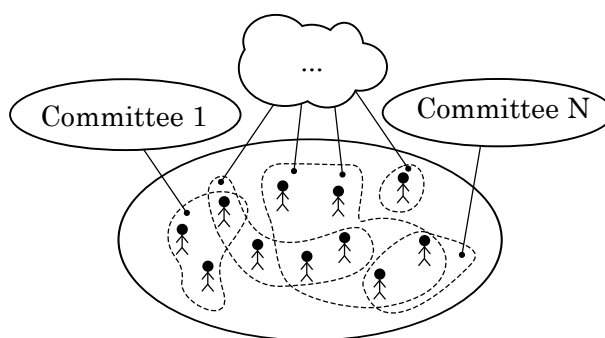


Figure 3.22: Committee problem

professor, but the scheduled committees will be attended in order of their importance.

In the “Dining Philosophers” arbitration with 4 philosophers, there will be 4 committees (being the philosophers) and 4 professors (being the forks). Each of the scheduled committees requires 2 professors to begin. If the event of assigning professors to a committee is not atomic, each committee may “capture” a single professor each and as a result stall in a deadlock state (see the problem of Dining Philosophers [1]).

3.8 Conclusions

The chapter presents different arbiter implementations, starting with simple 1-of-2 arbiters, fast non-scalable analogue arbiters, and progresses into more complex 1-of- N and M -of- N allocation schemes. The order of arbitration increases the number of questions regarding arbiter implementations. The simple two-way arbiters in general consider the communication rules between a client and an arbiter. Multi-way arbiters retain the concepts of communication protocol and add the concepts of topology, fairness and scalability, as those are the decisive factors identifying which arbiter is the best for a particular task. Finally, a range of multi-resource arbiters was presented, demonstrating that this allocation scheme inherits the concepts from the multi-way arbiters and also raises additional questions about resource utilization concurrency and the double-sidedness of the conflict, i.e., the multi-resource arbiters also need to grant resources, so that no more than one resource is accessing a client at a time. All these questions are important when building new arbiters, and will be considered in the next chapters. For further reading,

more information on asynchronous arbiters can be found in reviews [85, 41, 35]. Also, a summary of various other arbiter implementations is provided later in Appendix A.

Chapter 4

Concurrent Multi-Resource Arbiter: Design and Applications

4.1 Introduction

From a general point of view, there may be various situations requiring the multi-resource arbitration. In the simple scenario resources are interchangeable and can serve clients equally well, but the conflict of utilization is still there because there are not as many resources as there are clients. Examples of multi-resource arbiters were mentioned in the non-FIFO buffers [78] and the multibus solutions [54]. Instead of distributing a single resource, the arbiter selects one of the M available resources and grants it to one of the N clients. With more resources, more client requests can be satisfied concurrently, which effects in improving the overall performance of a system. Some of the resources may be released faster than others, resulting in their capture by new client requests sooner, but this only means that their next service will be granted sooner and the overall service time of each resource would still be balanced.

The view on the multi-resource arbiters can be generalized further by assuming that the resources also actively report their availability (Figure 4.1). Every client requests when a resource is needed, and also every resource indicates that it has become available for clients. The distinction is made here between active and passive resources. The *active*

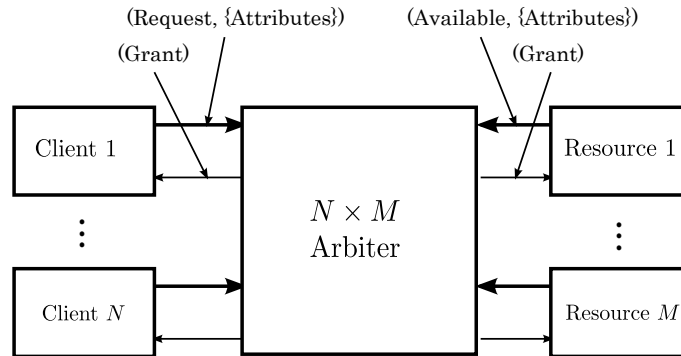


Figure 4.1: Synopsis of an arbiter [35]

resources independently indicate availability to the arbiter. If a resource has not made a request, it is not considered available by the arbiter and it will not be offered to the clients. In comparison, a *passive resource* does not make the initial request, instead, the arbiter assumes the resource is available when it is not in use by any of the clients. However, in this case, there may still be an additional handshake occurring between the arbiter and the resource to check the moment when the resource is ready (e.g. the tree arbiter cell).

Active resource solutions may be useful when the readiness of a resource may be postponed indefinitely after being released by a client. For example, this can occur if a resource is damaged or turned off to save power. For the damaged resource, all healthy resources can act as the “spare parts”, while switching off resources allows trading performance over power consumption. Yet another example of the active resource is a “product producer”, where consumption of its previous product (e.g. a result of a computation) does not imply the readiness of the next product.

The active resource arbiter problem can also be mapped to the so called *committee arbiter* [13]. There are committees that can be started by certain defined groups of professors. The restrictions are as follows: (1) a committee can start only when all of the professors are available, (2) Each professor can be engaged in not more than one committee at a time. When mapped to active resources, there will be $N + M$ professors arranging committees in pairs. In each pair, one professor must come from the group of N and the other from the group of M .

Multiple solutions for the committee arbiter were proposed in [9], which are based on the CSP 2-phase model specifications. This arbitration problem can also be interpreted

for our case when the initial requests are considered to be the professors and the granted pairing is the starting event of a committee. The solutions provided in [9] use either explicit or implicit polling mechanisms and/or multi-way arbiters based on 2-phase signalling and as a result lead to slower and larger hardware implementations.

This chapter presents the hardware design of asynchronous active multi-resource arbiters, that accept requests from both the client and the resource sides. The arbiter design is the speed-independent implementation based on 4-phase handshakes. First, the solution is provided for simple 2×2 arbitration, then multiple scalable $N \times M$ implementations are presented. The performance of the arbiter is estimated with transistor-level analogue simulations. The ability to activate concurrent utilization is important and is ensured to secure a better circuit performance when a single resource utilization time is sufficiently large. The implementation is formally verified to be Speed-Independent making it a more reliable circuit.

4.2 Design Method

Currently, asynchronous arbiter design is not fully supported by existing synthesis tools such as *Petrify* [4]. But traditionally an arbitration conflict is resolved by outlining critical section boundaries and using MUTEX elements in combination with supporting circuitry to manage requests entering and exiting these critical sections.

In this chapter the STG-based workflow is used, where the arbiter is initially specified by an STG, which is a convenient modelling formalism supporting concurrency and mutually exclusive events.

Figure 4.2 shows the design flow. First, a high-level *STG specification* is formed. It depicts the desired behaviour of the arbiter in an abstract way without specifying the details of how the conflicts are resolved.

Then, during the factoring phase one or more MUTEX element STGs are added to the model in such a way that the arbitration conflicts disappear from the original STG and only occur inside the inserted MUTEX element STGs. The result is a refined STG, where the MUTEX elements are modelled and *factored out* as a part of environment. This

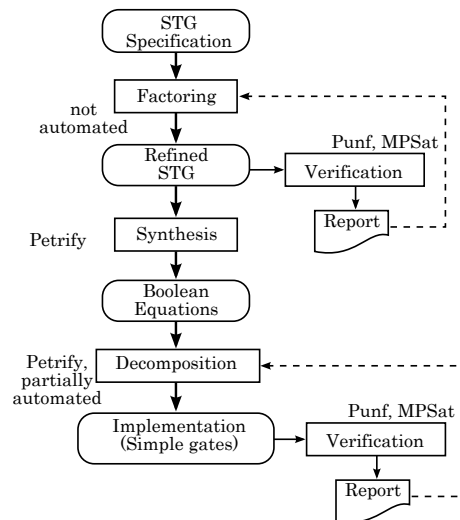


Figure 4.2: Arbiter design flow

STG can be verified for deadlocks, hazards, as well as other model states representing wanted or unwanted behaviour. If an error is found at this stage, it needs to be fixed before proceeding to the next flow phase.

The *refined STG* is synthesized with Petrifly. The tool produces the implementation in Boolean equations and is also capable of decomposing the circuit into simpler logic gates. This approach is useful when no particular structure is required. The *semi-automated decomposition* (through experimentation) may also help to find a structurally divided implementation, which is helpful for searching scalable design implementations.

At the end, the verification procedure is launched again to make sure the circuit works as expected and no hazards were introduced.

4.3 2×2 Arbiter Design

4.3.1 Functionality

Structurally, an active resource arbiter is symmetric from both sides with the same communication interface. Its task can be rephrased as the activation of available pairings between left and right neighbouring circuits effectively pairing outstanding requests from the opposite sides (Figure 4.3). All clients and resources actively participate by issuing requests (clients request when they need a resource and resources make requests

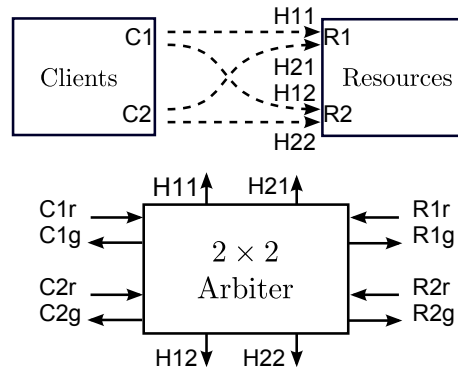


Figure 4.3: 2×2 arbiter interface

to inform the arbiter about their availability). The communication is organized in 4-phase handshakes where each client (and resource) releases the pairing by deactivating its *request* signal. The arbiter acknowledges each request with its associated *grant* signal. According to the protocol, the requester is only allowed to make a new transition after its previous transition has been acknowledged.

A pairing is possible when at least one resource and at least one client have made their requests. It forces the arbiter to activate the corresponding pairing and acknowledge the corresponding client and resource pair by issuing grants on both sides. Each client-resource pairing is identified by one of the complementary signals $H11, H12, H21, H22$ which can be used on both sides in combination with the grant signals. For instance, the signal $H12$ will signify a pairing of the client $C1$ and the resource $R2$. For the 2×2 case there are four different pairings in total. The removal of the requests will eventually result in the arbiter deactivating the pairing as well.

The grant signal issued by the arbiter is persistent. The arbiter waits until both sides remove their requests, and only then simultaneously removes their grants.

The basic model of the arbiter is presented in Figure 4.4. The red (input) transitions represent the environment and the blue (output) transitions represent the outputs from the arbiter. The initial token placement enables request transitions from both sides. As shown in the diagram, places $p1, \dots, p4$ render neighbouring pairings to be mutually exclusive. If client $C1$ is paired with $R1$, the pairing $H11$ takes away tokens from $p1$ and $p2$, effectively disabling $H12$ and $H21$. Pairings in the opposite corners do not share a

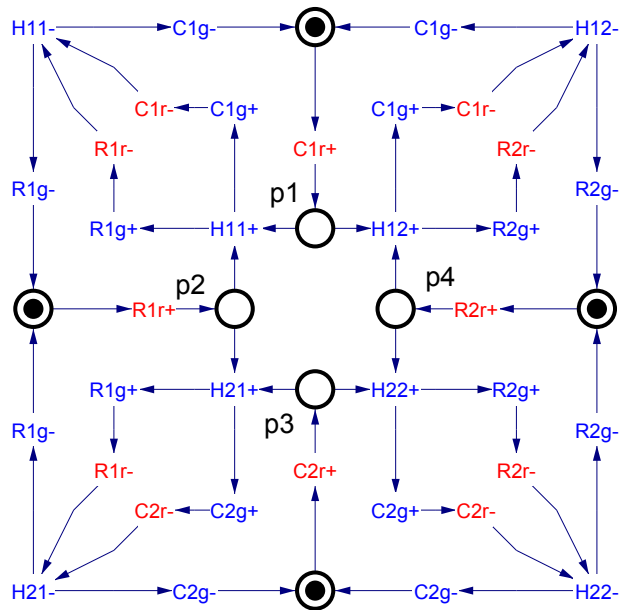


Figure 4.4: 2×2 arbiter STG

common place, and, according to the STG, can be activated concurrently.

It is important that the circuit does not prevent the non-conflicting pairings. When all four requests come at the same time, the arbiter makes a decision and connects requests by activating either $H11$ and $H22$ forming parallel connections or $H12$ and $H21$ forming the over-crossing connections (Figure 4.3).

To summarize, the arbiter functionality is shown in the following steps:

1. Any of the request lines are allowed to activate at any time. The arbiter waits until at least one pair of requests can be matched;
2. When a new pairing is possible, the arbiter activates it with one of the H signals and confirms requests with corresponding grant signals on the client and resource sides;
3. If all four requests arrive at the same time, the arbiter chooses to activate either $H11$ and $H22$ or $H12$ and $H21$;
4. When there are three requests active, only one pair will be formed and the unpaired request (either resource or client) will wait to be matched afterwards;
5. When both requests of a pairing are released, the pairing signal is released as well, which is subsequently acknowledged on the grant signal lines.

4.3.2 Resolving the Conflict

The STG in Figure 4.4 contains 8 conflicts (4 conflict pairs): $H11 \longleftrightarrow H12$, $H11 \longleftrightarrow H21$, $H22 \longleftrightarrow H12$, and $H22 \longleftrightarrow H21$. It is not clear how such a complex conflict set can be resolved in the analogue circuit domain. It is also known from the literature [35, 10] that building an arbitration structure more complex than a simple MUTEX element at the transistor level is problematic because such structures may be prone to oscillatory behaviour and pose additional challenges in the fabrication process. Hence, the aim here is to create an architecture for resolving this resource allocation task by means of MUTEX elements used as basic arbitration components.

It is possible to resolve the 2×2 conflict by reducing concurrency of the requesting parties. If client requests $C1r$, $C2r$ are exclusive and resource requests $R1r$, $R2r$ are exclusive, then only one pairing can be activated at a time. To constrain the concurrency of the requests, places $p5$ and $p6$ are introduced as shown in Figure 4.5. This modification places pairing signals $H11+$, \dots , $H22$ into a critical section where conflicting pairing signals are never enabled at the same time. After firing, these events immediately return tokens to $p5$ and $p6$ to allow the next pairing to commence.

The state graph produced from the STG diagram in Figure 4.5 shows that the concurrent pairings are still possible although the activation of the pairings was made sequential (Figure 4.6).

Because the signals $C1r$, $C2r$, $R1r$, $R2r$ are inputs of the arbiter (Figure 4.5), the additional restrictions such as mutual exclusiveness cannot be enforced without affecting the behaviour of the circuit environment (Figure 4.4). Hence, new internal signals must be introduced, which would simulate the environment requests while supporting the desired behaviour of exclusiveness.

The exclusiveness of the environment requests can be ensured by introducing MUTEX elements as in Figure 4.7. The path from the request transition $C1r+$ to the place $p1$ now also contains transitions $rc1+ \rightarrow gc1+$, which implies MUTEX arbitration. Similarly, all of the pairings are now activated through the arbitration of MUTEX STGs incorporated into the model (ignore the dashed arcs at this point). The *local request trans-*

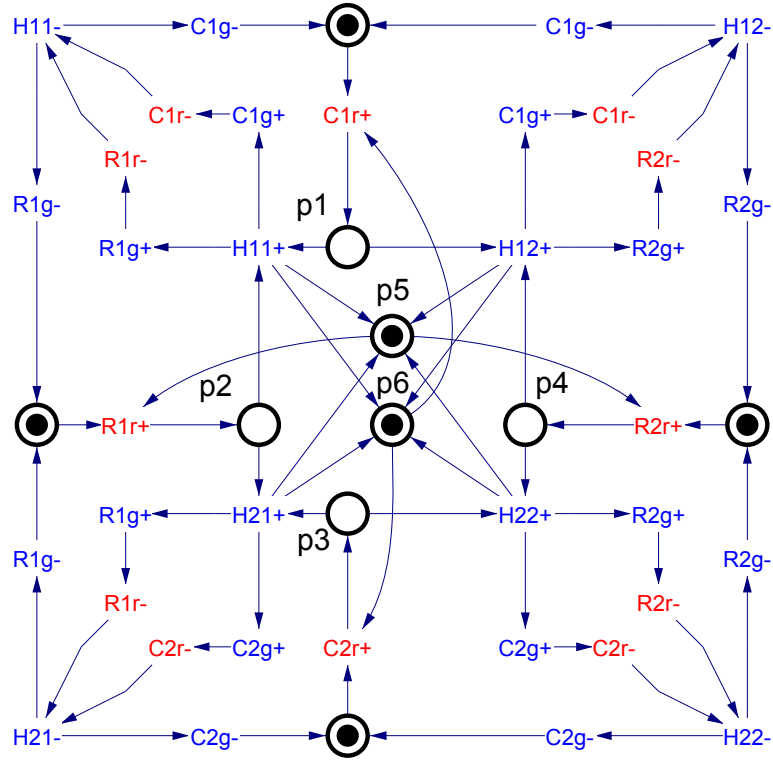


Figure 4.5: Additional exclusion places added

itions $rc1+, \dots, rr2+$ receive requests from the environment and propagate them to the MUTEX component. The *local grant transitions* $gc1+, \dots, gr2+$ implement the constraint of mutual exclusiveness and forward the environment requests to the pairing activation events $H11+, \dots, H22+$. While the MUTEX components work independently, their combination ensures that there are no more than two tokens in the places activating the pairings. This ensures that no more than one pairing activation will happen at any time, thus resolving the conflict completely (no additional arbitration is needed). Right after the first pairing has been activated, the MUTEX request signals are released, enabling the subsequent activation of the second pairing in the opposite corner if the requisite requests were asserted.

In the STG with MUTEXes, there is a *complete state coding conflict* [19]. Consider the trace: $C1r+ \rightarrow rc1+ \rightarrow gc1+$. In this state only the signals $C1r, rc1, gc1$ will evaluate to 1. The $rc1-$ is not allowed to fire yet, it waits for $H11+$ or $H12+$ to fire first. Then, if transition $rc1-$ is sufficiently slow, the events may produce the trace $R1r+ \rightarrow \dots \rightarrow$

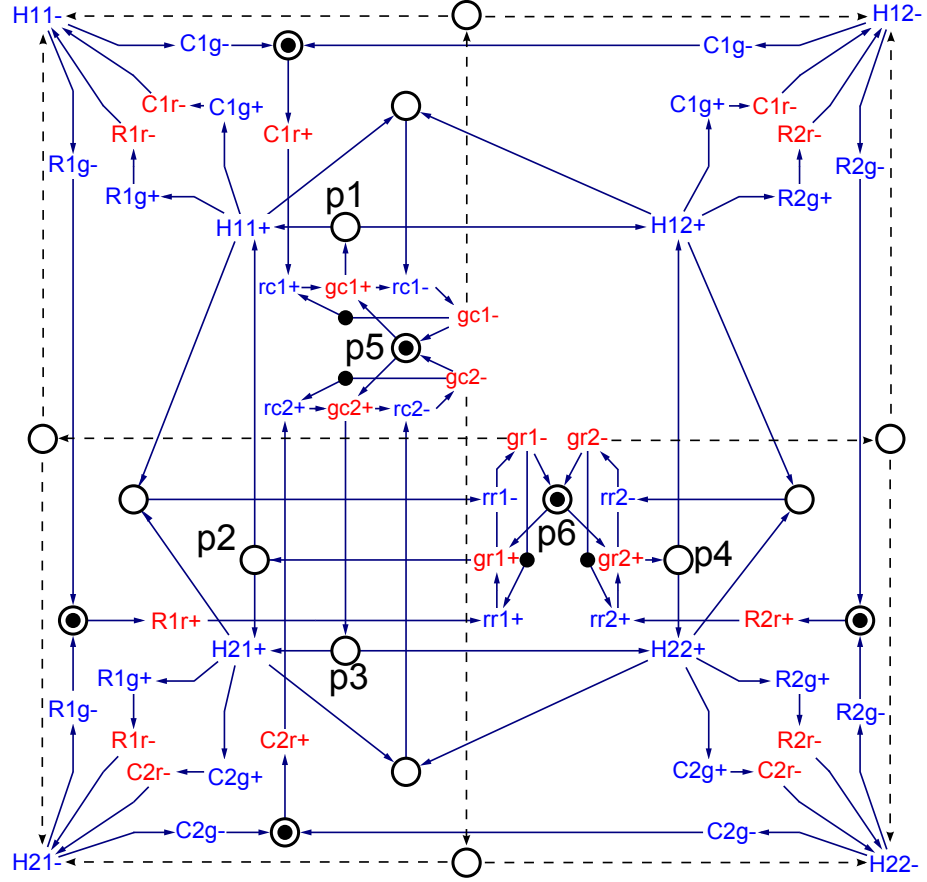


Figure 4.7: STG with MUTEX elements

$$H11 = gc1 \cdot gr1 \cdot \overline{H12} \cdot \overline{H21} \\ + H11 \cdot (gc1 + gr1 + C1r + R1r)$$

$$H12 = gc1 \cdot gr2 \cdot \overline{H11} \cdot \overline{H22} \\ + H12 \cdot (gc1 + gr2 + C1r + R2r)$$

$$H21 = gc2 \cdot gr1 \cdot \overline{H11} \cdot \overline{H22} \\ + H21 \cdot (gc2 + gr1 + C2r + R1r)$$

$$H22 = gc2 \cdot gr2 \cdot \overline{H12} \cdot \overline{H21} \\ + H22 \cdot (gc2 + gr2 + C2r + R2r)$$

$$rc1 = \overline{H12} \cdot \overline{H11} \cdot C1r \quad rc2 = \overline{H22} \cdot \overline{H21} \cdot C2r$$

$$rr1 = \overline{H21} \cdot \overline{H11} \cdot R1r \quad rr2 = \overline{H22} \cdot \overline{H12} \cdot R2r$$

The equations $H11, \dots, H22$ represent 6-input gates with memory and need to be

decomposed into smaller gates. If decomposed with Petrify, each of the pairing signals will be formed of two 5-input gates:

$$\begin{aligned}
 [2] &= gc1 \cdot (\overline{H12} \cdot \overline{H21} \cdot gr1 + H11) \\
 H11 &= H11 \cdot (gr1 + R1r + C1r) + [2] \\
 [4] &= gc1 \cdot (\overline{H11} \cdot \overline{H22} \cdot gr2 + H12) \\
 H12 &= H12 \cdot (gr2 + R2r + C1r) + [4] \\
 [6] &= gc2 \cdot (\overline{H11} \cdot \overline{H22} \cdot gr1 + H21) \\
 H21 &= H21 \cdot (gr1 + R1r + C2r) + [6] \\
 [8] &= gc2 \cdot (\overline{H12} \cdot gr2 \cdot \overline{H21} + H22) \\
 H22 &= H22 \cdot (gr2 + C2r + R2r) + [8]
 \end{aligned}$$

The decomposition is correct and is one way to implement these signals. From the equations above one can see the conditions for the set and reset events. Each pairing is only set when both local requests arrive and none of the conflicting pairings is active. The reset of the pairing is only allowed when both of the associated requests are reset and the local grant signals released. Based on this knowledge, it is possible to find another structurally better organised decomposition.

4.3.3 Implementation

Arbiter Structure

The structure is presented in Figure 4.8. By introducing new *internal pairing* signals $h11$, $h12$, $h21$, $h22$ it is possible to subdivide the implementation of signals $H11$, $H12$, $H21$, $H22$ into the *grant controller* and the *request controller* parts, which splits arbitration into two simpler problems.

Each of the requests from either client or resource side can arrive at any moment. Suppose client C1 has issued a request $C1r+$. The signal first propagates through the request mask: $rc1+$ and then is arbitrated with the neighbouring request from C2 (Figure 4.9a). Both MUTEX elements ensure there are at most one client and one resource entering the request controller part. The conflict is completely resolved for all request combinations, even when all clients and all resources issue requests at the same time.

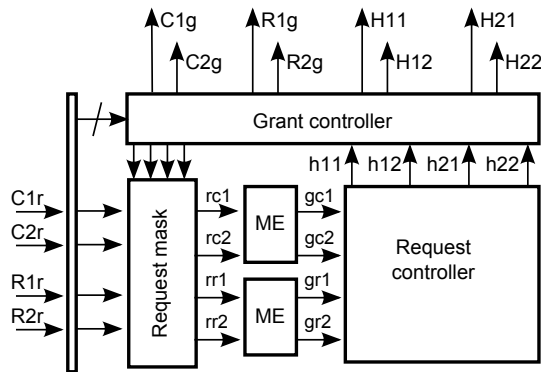
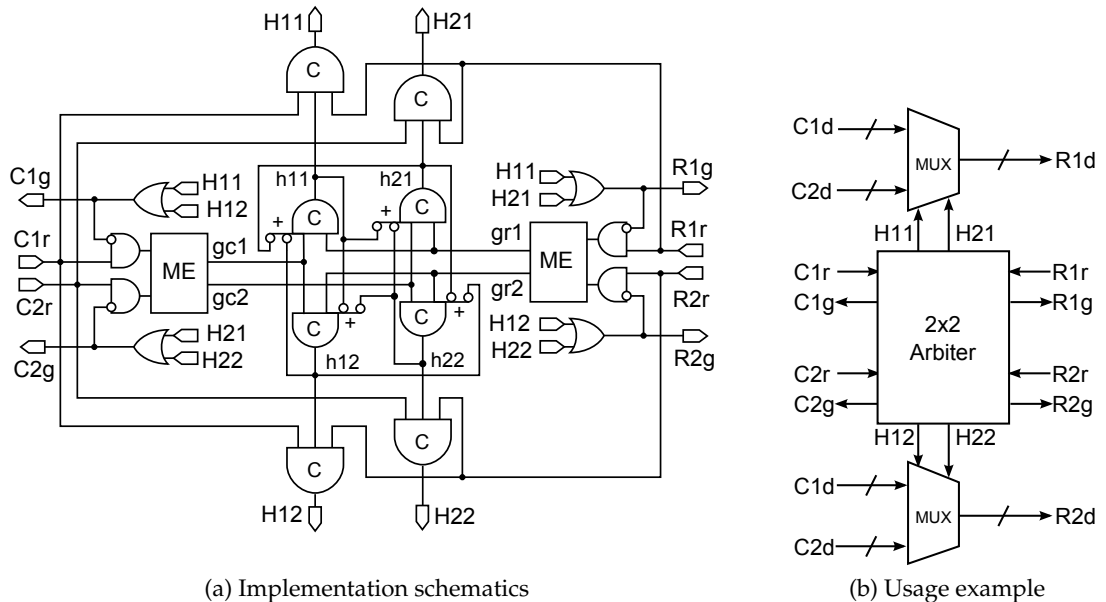


Figure 4.8: Arbitrer structure



(a) Implementation schematics

(b) Usage example

Figure 4.9: 2×2 arbiter implementation

Suppose eventually client C1 and resource R1 issue requests $C1r+$, $R1r+$ (Figure 4.9a). The requests propagate through the request mask: $rc1+$, $rr1+$. Then they propagate through MUTEXes and win the arbitration: $gc1+$, $gr1+$. The local grants $gc1+$ and $gr1+$ eventually activate the pairing $H11+$ and the grant controller activates the arbiter grant signals: $C1g+$, $R1g+$. The request mask consisting of a column of 2-input AND gates hides the initial requests: $rc1-$ and $gr1-$ which in turn release both MUTEX elements and allows new requests to propagate into the request controller part. The mask is manipulated by the arbiter grant signals $C1g$ and $R1g$. So, when the client or resource request is removed, its corresponding grant signal will also be deactivated, which in turn

would open the mask for new requests.

Request Controller

The speed-independent implementation of the request controller consists of four asymmetric C-elements producing the pairing request signals h_{11}, \dots, h_{22} . It asserts the pairing request (i.e., changes one of its outputs from 0 to 1) when a pair of local grant signals arrives. Later, when both local grants are removed, the pairing request signal is also removed (i.e., changes from 1 to 0).

Because the request masks may release the left and right MUTEX elements at different times, there may be a situation when a new local grant has propagated through one MUTEX element with the other MUTEX still holding the old local grant value. This situation could trigger the activation of the pairing in conflict with the one that had been activated. To prevent this from happening, each of the asymmetric elements receives dedicated block signals from each of the conflicting pairings.

Grant Controller

The grant controller consists of four 2-input C-elements and four 2-input OR gates. It is activated by the request controller and is responsible for keeping the request active while either of the sides still needs it. While the pairing activation is sequential, the pairing release phase is fully concurrent.

Each pairing is activated by synchronising the requests from the associated participants and the selection of the request controller. The activated pairing then propagates through the OR gates to deliver the grant signal to the original requesters. The pairing signals can be used to select the required data propagation path as shown in Figure 4.9b, which is then followed by the outgoing grant signal to activate the corresponding client and resource.

4.3.4 Verification of the Circuit

The original Petrify synthesis result is expected to be *speed-independent* with all the safety properties this implies. However, the decomposition needs to be verified because it was derived through intuition.

The circuit can be formally verified with command line tools *Punf*, *Mpsat* [5, 33] and *Workcraft* integrated environment as described in [65]. First, the gate-level model of the circuit is created in *Workcraft*. By using the inbuilt functionality of the tool, it is automatically converted into its STG equivalent (the so called *circuit Petri net*). Such a model defines the causality of its outputs. However, it does not have information about the input transitions, which is defined in a separate *interface STG* (also called the environment model of the circuit). The environment model that defines the behaviour of all inputs is derived from the initial STG model shown in Figure 4.4. Then the circuit Petri net, combined with the environment model, is processed with *Mpsat* in collaboration with *Punf* to verify whether the circuit has states producing hazards [65].

The implementation of the circuit was verified using the method described above. The output of *Mpsat* confirmed that there are no reachable states present in the initial STG (Figure 4.4) where one of the signals disables the activation of the others regardless of the speed of each circuit gate. It was also checked that the states activating concurrent non-conflicting pairings ($H11$ and $H22$ or $H12$ and $H21$) are reachable and there are no reachable states activating the pairings in conflict.

4.3.5 Latency Estimation

The arbiter performance can be estimated based on the average time that passes from the moment of the initial request to the moment the circuit grants the pair and propagates this grant to the corresponding client and resource. The circuit response latency depends on the request arrival from both sides. Therefore, it is reasonable to estimate the latency starting from the point when both (one client and one resource) requests have arrived.

The first pairing latency is counted from the moment when at least one pair of requests has arrived, to the moment when one of the pairings have been activated (and the

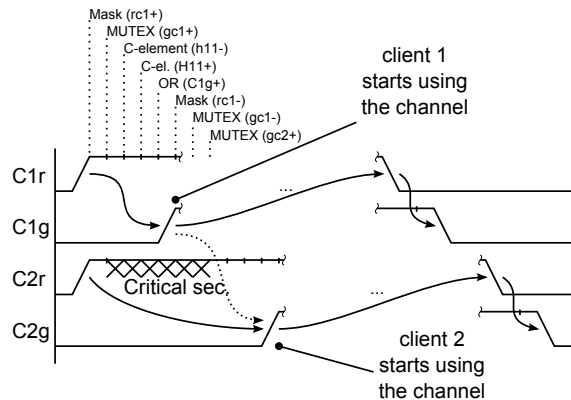


Figure 4.10: Timing diagram

consequent grant signal was send to the client and the resource). Similarly, the second pairing latency is counted starting from the moment the second pair of requests has arrived. If there is no metastability, the latency for the first pairing is always the same. The second pairing latency depends on whether the circuit is occupied activating the first pairing. The worst latency occurs when all requests arrive at the same time. In this case, the circuit has to activate the first pairing before it processes the second one.

Consider the timing diagram of the 2×2 circuit (Figure 4.10). The figure depicts the situation when all four requests have arrived simultaneously (only the client side signals are shown). For simplicity assume that all gate delays are the same for each gate and there is no additional delay due to metastability in the MUTEX elements. As shown on the diagram, both requests propagate through the request mask simultaneously and arrive into the corresponding MUTEX. Once one of the requests passes the MUTEX cell, the circuit enters its critical section and the other request must wait till the MUTEX is released. In our example $C1r$ wins the MUTEX arbitration and receives the pairing first, thus having a better response time. The second client waits until the MUTEX is freed by the request mask, which happens right after the second C-element activates the pairing $H11$. The critical section introduces six gate delays making the response time for the second pairing a constant eleven gate delays. This latency is not related to how long client 1 occupies the allocated resource, which may be of a variable and significant length. This reduction of the critical section to a length not related to the environment is obtained with the internal parallelism of the arbiter.

Table 4.1: 2×2 performance estimation in Spectre

#	$C1r$	$C2r$	$R1r$	$R2r$	$C1g$	$C2g$	$R1g$	$R2g$	1st lat.	2nd lat.	2nd-1st	mW
1	100	—	100	—	479	—	487	—	387	—	—	0.4
2	100	105	100	105	517	938	526	947	426	842	416	0.59
3	100	101	100	101	540	960	549	969	449	868	419	0.58
4	100	300	300	200	579	999	990	570	379	699	320	0.47
5	700	100	700	200	1076	576	1085	585	385	385	0	0.4

4.3.6 Simulation in Spectre

The circuit was modelled with Spectre (Cadence analogue analysis tool) in a 90nm technology library with balanced transistor widths ranging from 500nm to 6 μ m. The rising and falling edges of any of the inputs is set to 50 picoseconds and the supply voltage is 1 Volt. One fan-out of 4 inverters delay (FO4 delay) is measured to be 25ps. No wire delay was included in the model. All of the measurements are taken when the signal value passes the 0.5V voltage level.

The circuit response latency is estimated in Table 4.1. The first group of columns (from $C1r$ to $R2r$) shows the absolute timing of the arriving signals (all latency numbers are shown in picoseconds). The next group shows response times for each signal. The following two column groups show the time that has passed since the moment the request was initiated (when both a client and a resource arrive and the circuit actually can respond with at least one pairing activation).

The response latency for the first example can be estimated as the distance between the request pair arrival ($C1r, C2r$) which is 100ps and the last grant signal: $487 - 100 = 387$ ps.

In the second and third examples requests arrive within 5 and 1 picoseconds thus creating a metastability overhead and increasing the absolute response latency for both pairings. The second pairing is delayed by the first one. It is waiting for the first grant signal to mask the initial request and release MUTEX element, consequently activating the second pairing. After the first pairing was resolved, the additional latency is deterministic and is capped at around 420ps.

In the fourth example, the requests for the second pairing ($H21$) arrive 100 pico-

seconds after the requests for H_{12} . The arbiter is still busy with the first pair of requests; however, the second pair needs to wait 100 picoseconds less. This is because the second pair of requests arrived after the first pairing was well into its activation.

Finally, in the last example requests arrive at such timings that the pairing activation times do not overlap. In this case, the second pairing is activated with the same latency as the first one.

4.3.7 Cost of the Parallelism

Because the implementation of concurrent requests requires additional logic, it is interesting to find out the latency behaviour of fully sequential solutions.

For the sequential implementation there is no need for the request masks and heavy 2-input C-elements. The internal pairing signals will produce pairing activation signals ($\overline{H_{11}}$, $\overline{H_{12}}$, $\overline{H_{21}}$, $\overline{H_{22}}$). Also, the negative logic optimisation can be applied. The OR gates formerly collecting the H signals can now be changed to NAND's. The simulation has shown significant improvement of the activation time, shrinking the latency to just 150ps. However, the worst case waiting for the second pairing now also depends on the utilization time of the first pairing and the deactivation time (the unlocking of the MUTEX element).

Assume initial requests arrive as in the second test case from Table 4.1. The first pairing is granted at 189ps, at which point the requests for the first pairing begin to withdraw. Finally, the grant of the second pairing arrives at 524ps. This result is $842 - 524 = 318$ ps better than it is in the concurrent model. It means that in the current setup, if the pairing is used for less than 300ps, the feature of opening the second pairing concurrently is not justified and a simpler circuit could effectively provide the same basic functionality.

4.4 Extending up to $N \times M$ Arbiters

In the general problem statement, the arbiter needs to support N clients and M resources. A rectangular grid of tiles can be used to implement the functionality of C-elements in the original design in Figure 4.9a. This regularity helps to enhance the layout and scalability

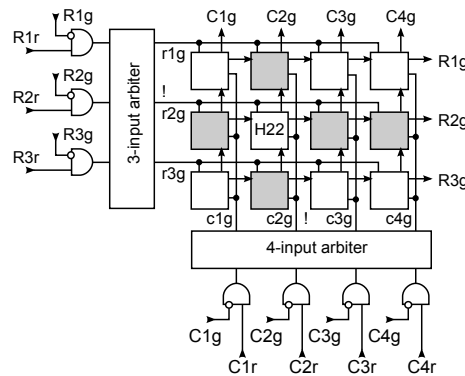


Figure 4.11: 4×3 arbiter implementation (shows active requests on $r2g$ and $c2g$)

of the design.

An example of a layout for 4 clients and 3 resources is given in Figure 4.11. The internal resource grants $r1g$, $r2g$, $r3g$ and client grants $c1g$, $c2g$, $c3g$, $c4g$ form three rows and four columns of the grid. Similarly to the 2×2 arbiter all the conflicting pairings need to be disabled before the new requests propagate through the M - and N -input arbiters. In particular, the pairing row needs to be disabled before new client and resource requests propagate through the local arbiters. As it can be seen from the figure, there are $N + M - 2$ conflicting pairings (or tiles) for each H tile.

If the structure of the 2×2 arbiter (Figure 4.9a) is to be directly used, for the 4×3 arbiter this would require 7-input asymmetric C-elements, which is not practical for increasing N and M .

Another problem with directly extending the solution in Figure 4.9a to $N \times M$ tiles is the fact that this solution is not based on *true tiles* [72]. With increasing N and M , each tile needs to maintain its interface by adding new inter-tile wires (new input wires for each additional pairing in conflict). A truly scalable solution should employ true tiles that do not change shape with the size of the grid and have a constant number of inter-tile wires that does not increase with the size of the grid. With these, a change of N or M will result in the simple addition or removal of rows or columns of tiles.

In order to achieve this, an explicit blocking mechanism may be developed that would disable all the conflicting pairings before the pairing activation signal H is raised. This is described in the following sections.

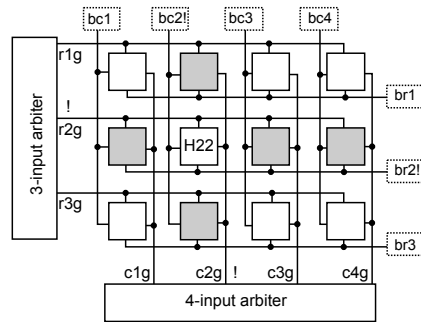


Figure 4.12: Arbitrer with blocking tiles

4.4.1 Column/row Blocking

The explicit blocking can use additional specialised “blocking tiles” associated with each column and row (Figure 4.12). Each of the blocking tiles is simply a tree of OR gates broadcasting the block signal when one of the associated tiles was chosen for a new pairing. Under the assumption of speed-independence, where wire forks are isochronous blocking all of the conflicting tiles simultaneously.

At first, two requests propagate over one column and one row. The tile that receives both requests acknowledges itself as the chosen pairing with $x+$ and sends two requests on its column and row block tiles (Figure 4.13). As a result, the whole column and row become blocked for new requests ($bci+$, $bri+$). The tile chosen understands that because it also receives the same block signals. Consequently, it activates its h signal: $h+$ and waits until both requests cg and rg are withdrawn. The blocks are released ($bci-$, $bri-$) only after both old requests were removed ($cg-$, $rg-$). Hence, there is no risk of a wrong tile being activated.

Since both $cg+$ and $rg+$ signals are going to propagate to all tiles in a certain row and column, there will be tiles that receive only one request signal placing a token in either $p1$ or $p2$ (Figure 4.13). To ensure the 1-safeness of the net, one of the events (either $bri+$ or $bci+$) will remove this token from the place preceding $x+$. The signal sequences $cg+ \rightarrow bci+ \rightarrow cg- \rightarrow bci-$ and $rg+ \rightarrow bri+ \rightarrow rg- \rightarrow bri-$ correspond to the communication inside the disabled tiles. It is important that the disabled tile never reacts

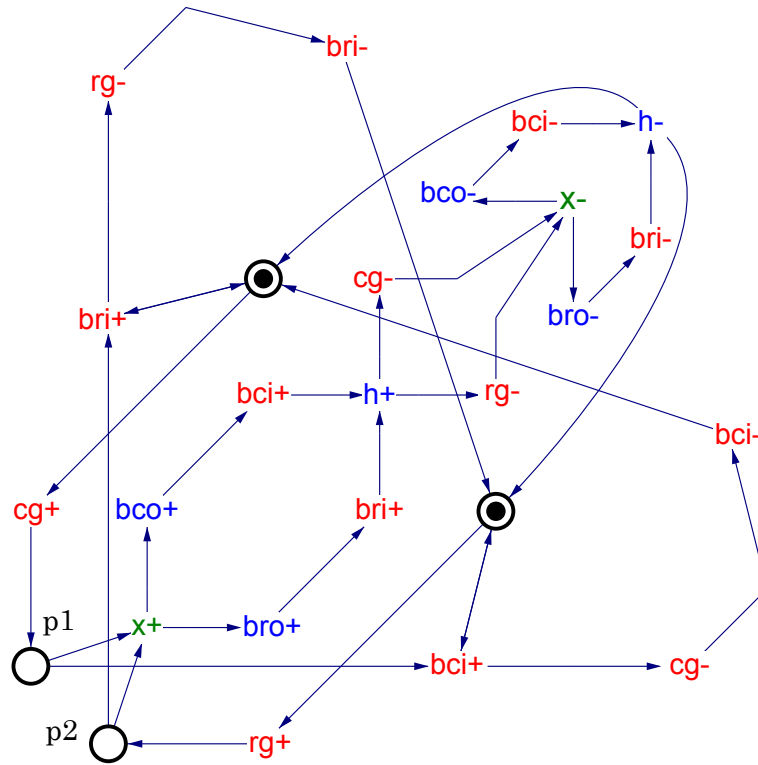


Figure 4.13: Column/row block tile STG

with $h+$ which is ensured in the given STG.

An implementation of the H tile (as found by Petrify) is shown in Figure 4.14. The signals bci are obtained by OR-ing all bco from each H tile on the pairing column. Symmetrically, the bri input is obtained by OR-ing bro from each tile of the row. The arrival of $bci+$ and $bri+$ eventually activates $h+$ and disables all of the conflicting pairings. Because of the existence of dedicated row/column blocking tiles, the gates dealing with blocking and unblocking within the H tile are greatly simplified and the large fan-in problem does not occur.

The shortcomings of using blocking tiles is that the isochronic forks of the scaled N and M may be increasingly difficult to satisfy. So, an alternative implementation is considered in the next section.

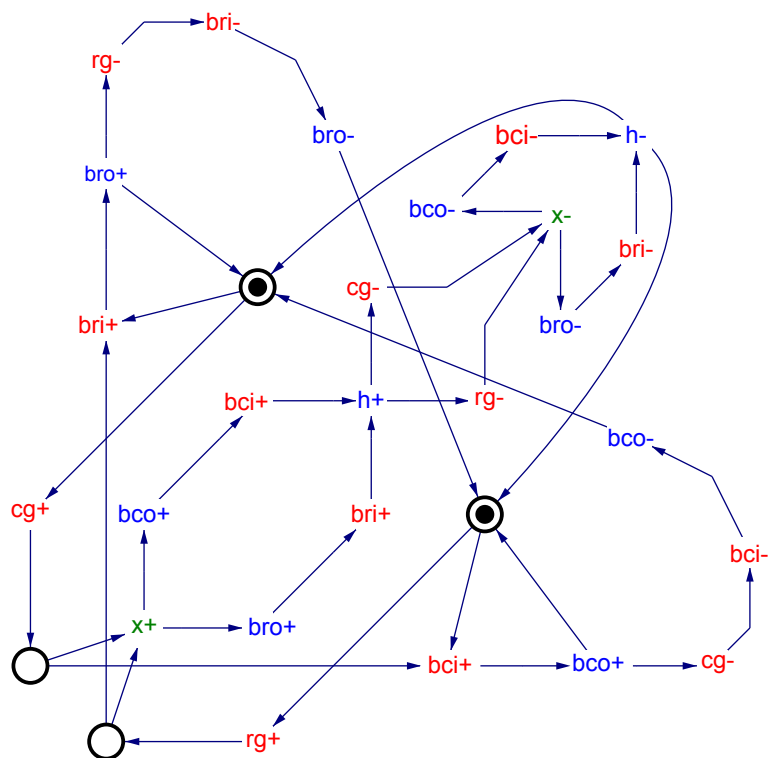


Figure 4.15: Ring-based tile STG

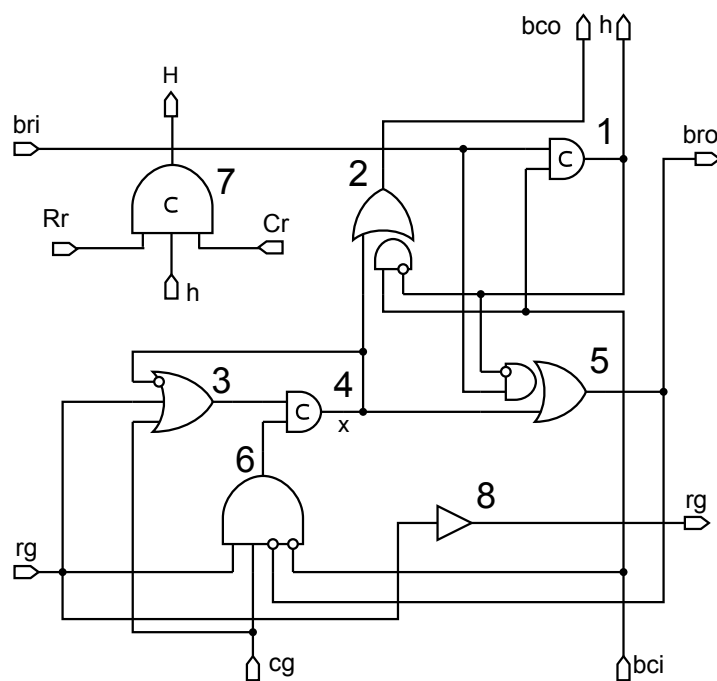


Figure 4.16: Tile implementing ring-based approach

This tile implementation is speed-independent, and the inter-tile blocking and unblocking connections are delay-insensitive because the blocking and unblocking signals are monotonic and the logic along the blocking/unblocking path is fully combinatorial with no forks on outputs. This means that the overall system is speed-independent within tiles of relatively compact size and along longer connections it is delay-insensitive.

4.4.3 Latency Estimation

One disadvantage of the $N \times M$ arbiter is the quadratic growth of the worst case response latency. The worst delay happens when all of the requests on both sides arrive at once, it can be estimated as: $L(N, M) = (\min(N, M) - 1) \cdot \delta$, where δ is the time needed to activate one pairing. Note that δ also grows with the increased N and M . For the column/row block it is logarithmic: $\delta_{cr} = \log(\max(N, M))$; and for the ring-based block it is linear: $\delta_{ring} = \max(N, M)$. This renders the design impractical for the increasing N and M and more efficient methods need to be considered.

An alternative approach would be to use a dedicated *lock* signal taking a snapshot of currently active requests [11, 72]. The lock mechanism formed by a column of the MUTEX elements allows the granting of all the pairings concurrently, thus reducing the circuit response latency. The implementation and the comparison of this approach is the subject for further research.

4.4.4 Simulation in Spectre

The model of the 4×3 arbiter was simulated in Spectre with a setup similar to the 2×2 arbiter. The circuit connects 4 clients with 3 resources and can produce up to 3 concurrent pairings out of 12 possibilities. Table 4.2 presents latency estimation for various request arrival times. The internal arbiters are basic 3- and 4-input mesh-based ones found in [35]. Their behaviour may not be completely fair regarding the actual arrival times of requests due to their topological asymmetry. As a result, there is a slight variation in the additional critical cycle delay depending on which request pair arrives first.

The first three examples show latencies for three pairs of requests. Examples 4 and 5

Table 4.2: 4×3 performance estimation in Spectre

#	C1r	C2r	C3r	C4r	R1r	R2r	R3r	1st lat.	2nd lat.	3rd lat.	Overh.	mW
1	25	—	—	—	25	—	—	509	—	—	—	1.17
2	—	25	—	—	—	25	—	517	—	—	—	1.17
3	—	—	25	—	—	—	25	510	—	—	—	1.16
4	25	30	35	40	25	30	35	556	1138	1710	572,582	1.65
5	25	26	27	28	25	26	27	562	1144	1714	570,582	1.82
6	25	125	225	325	25	125	225	508	1037	1521	529,484	1.78
7	25	325	625	925	25	325	625	509	826	1111	317,285	1.77
8	25	725	1425	2125	25	725	1425	509	516	510	—	1.59

demonstrate the maximum latency overhead along with the metastability due to simultaneous request arrival.

In conclusion, the response latency for each pair of the requests (when there is no conflict) is roughly the same and equates to the critical section delay δ of the circuit (which in this case is approximately $\delta \approx 510\text{ps}$). The probability that there would be a conflict between two request pairs solely depends on their rate. With an increasing number of resources and clients and the rate of requests, the probability of clashing pairs would increase, which is another reason why this design does not scale well.

4.4.5 Fairness of the Arbiter

It is easy to see that the fairness of the $N \times M$ arbiter is dependent on the fairness of its internal arbiters used to select pairings. The fair internal arbiter would ensure fair selection of resource or client; however, depending on the design needs, the client or resource or both can be made prioritised.

4.5 Multi-resource Arbiter for Passive Resources

4.5.1 Task Specification

In the remaining part of the chapter multiple examples of arbiter usage in practice are presented.

The first example presents creating multi-resource arbiter for *passive resources*. For a

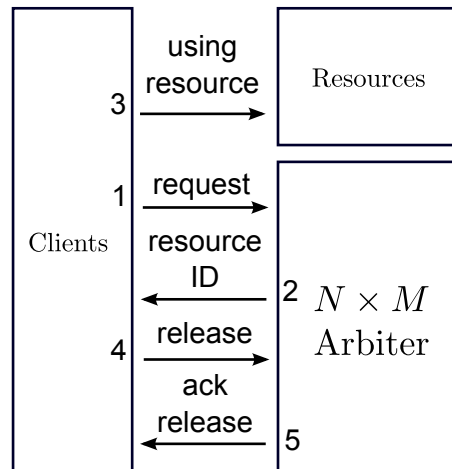


Figure 4.17: Asymmetric multi-resource arbiter structure

given client request, the arbiter would return the available resource and it would make sure that not more than one client is receiving the same resource (Figure 4.17). After the resource is no longer required, the client needs to inform the arbiter that it is releasing the resource, so that the resource would become available to other clients. Depending on the protocol, an explicit signal could acknowledge the client that the resource had been released.

The implementation is based on the idea of multi-token ring arbiters presented in [85]. The token ring would consist of a certain number of separate cells, each cell connecting to one client and two neighbouring cells (Figure 4.18). The tokens propagating inside the ring are considered as available resources and can be captured by clients when they need resources and later inserted back into the ring.

The model of such a cell is similar to the 2×2 arbiter in Figure 4.9a. The former client side requests $C1$ and $C2$ correspond to the propagation channels “Token put” (releasing the token back into the loop) and “Token in” (delivering token from the left neighbour). Correspondingly, the channels “Token get” (capturing the token for the client) and “Token out” (delivering the token to the right ring cell) correspond to the resource side requests $R1$ and $R2$, so the pairings are now formed between the token propagation channels.

The token events are asynchronous and can happen at any moment on any channel.

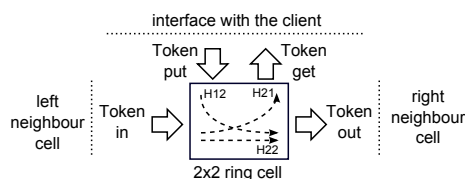


Figure 4.18: Busy token ring cell

The exception is the concurrent propagation on “Token put” and “Token get” because normally a client would not try to get and put the token at the same time. This leaves only three different token propagation scenarios. Firstly, the token can propagate through the cell from the left to the right neighbour (pairing $H22$ in the initial design). Secondly, it can be captured by the client, connecting channels “Token in” and “Token get”, which would be the pairing $H21$. And finally, the token can be released by connecting channels “Token put” and “Token out”, the pairing $H12$. $H11$ is not needed any more, which simplifies the circuit design. It means there would be no simultaneous transfer for $H11$ and $H22$. In addition, since the client either takes a token or puts it back into the system, there would be no simultaneous transfer for $H12$ and $H21$. This leads to a simplified STG diagram (Figure 4.19).

The analysis of the diagram shows that there are no pairs of conflicts: $H22 \longleftrightarrow H12$ and $H22 \longleftrightarrow H21$. The environment makes a choice on either to get the token from the loop ($Tgr+$) or put it back ($Tpr+$) and there are at most three requests activating at a time.

4.5.2 Implementation of the Ring Cell

The cell can be implemented fully sequentially. First of all, the absence of concurrent transfers removes the need for the gates masking initial requests. Additionally, because there are no concurrent transfers, there is no need for C-elements forming the grant controller, as it can be done by using just the internal asynchronous C-elements forming the request controller in the initial design. Additionally, the pairing $H11$ is not needed and can be removed. The pairings still need to be blocked by conflicting pairings in order for the circuit to work, because, according to the protocol, before the arbiter activates a

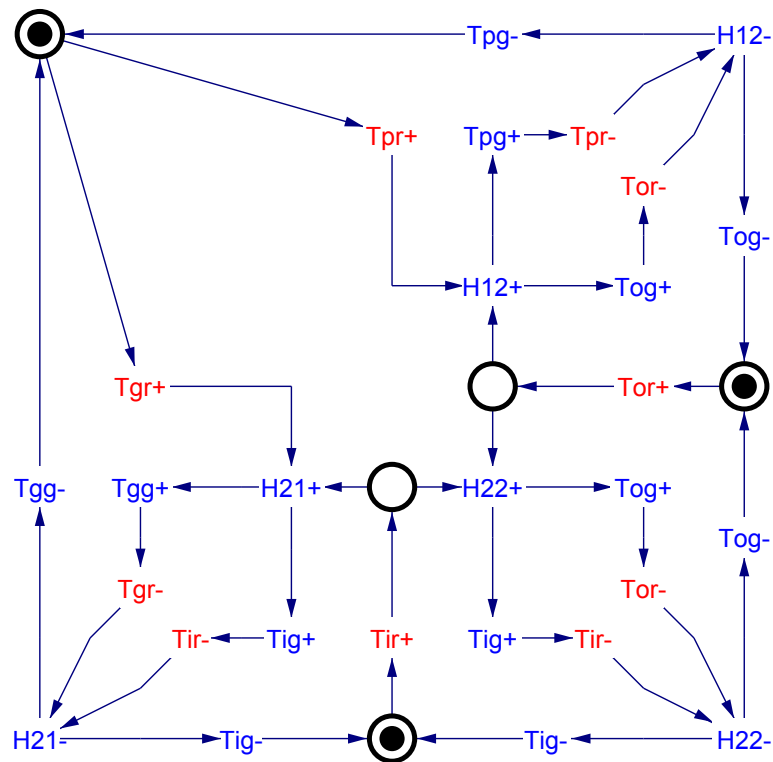


Figure 4.19: The STG of a ring cell

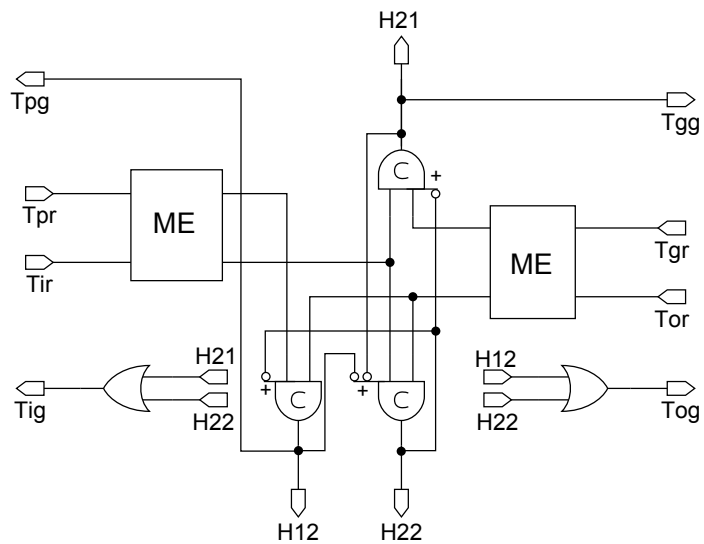


Figure 4.20: Implementation of the ring cell

new pairing, it needs to wait until both parties remove the request from the previous communication (Figure 4.20).

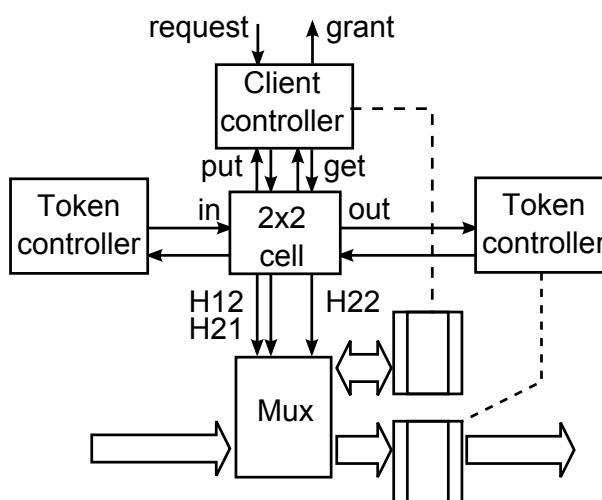


Figure 4.21: Ring cell structure

4.5.3 Implementation of the Client Controller and Token Controller

Apart from the cell providing the pairings, two additional controllers are needed: the token controller acquiring tokens from the left neighbour and delivering them to the right neighbour, as well as the client controller, providing the request/grant interface for a client (Figure 4.21). Each controller is associated with a data register designed to store the identifier of the token.

4.5.4 Latency Estimation

Similarly to the usual token ring response latency, the response time of this arbiter linearly increases as the number of clients N grows. Fortunately, this latency can be reduced when more resource tokens propagate through the ring. The more there are resource tokens in the system, the less a client needs to wait till the moment it receives a token. For the uniform token distribution, the latency can be estimated as N/M , where M is the number of resources propagating over N client cells, although, due to the effect of tokens clumping together [84], the worst case latency may degrade to $N - M$.

4.5.5 Comparison with Patil's Arbiter

A similar kind of arbiter, where multiple passive resources are distributed among clients, was presented earlier by Patil [58, 59]. Both arbiters distribute M passive resources over N clients. The distinct advantage of the Patil's arbiter is that it does not imply busy waiting. When there are no new requests made, the arbiter does not consume dynamic power. The multi-token arbiter presented in this chapter follows the ideas of the busy token-ring arbiter. Tokens always keep propagating around the loop even if the arbiter is not receiving new requests.

The main advantage of the token-based arbiter is that all the tokens are propagating independently around the loop. So, the arbitration is concurrent for each client, which significantly reduces the arbiter latency. Another important advantage is that the design is much easier to scale. For each additional client there would be only one additional client controller and the token controller with no increasing complexity in wiring.

4.6 Designing MIMO Queues

A Multiple-Input-Multiple-Output (MIMO) queue is a type of propagation channel attached to multiple senders and receivers. It functions as a buffer adjusting the mismatches between sender and receiver rates and also provides a choice between the interchangeable receivers. So, it acts as a multi-resource arbiter, where the data flits sent from any of the senders are distributed over multiple receivers.

As demonstrated in [80], MIMO queues (among other designs) can be constructed in order to build applications tolerating variability of task "consumers" and task "producers". For instance, such a queue can be used to perform load balancing of tasks over multiple processors. In [80] authors present the design of a basic 2×2 queue component with a functionality similar to the 2×2 arbiter. It consists of two tree arbiters connected through the *handshake passivator* as shown in Figure 4.22. The multiplexers are managed by the SR-latches which are in turn set up by the MUTEX elements inside the 2-port tree arbiter cells (see [80] for more details). The *handshake activator* is used to request new data

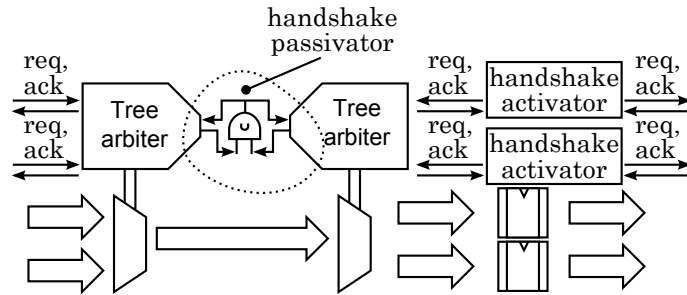


Figure 4.22: Structure of the sequential 2×2 MIMO queue [80]

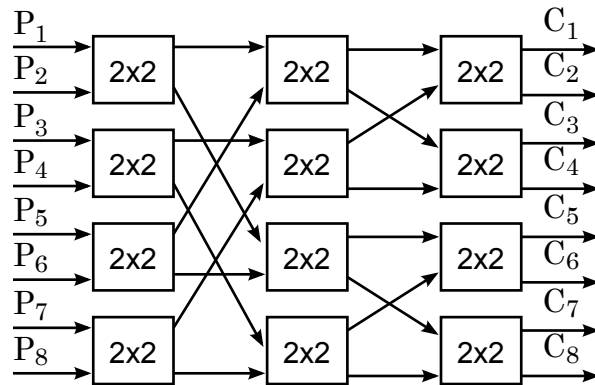


Figure 4.23: 8×8 MIMO queue

from its left neighbour and store it in its data flip-flop. First, it requests the data from the multiplexer on its left side. Then, after the full 4-phase communication cycle is finished, it pushes the data to its right neighbour. Since all MIMO inputs and outputs are push channels, it is possible to attach multiple 2×2 components sequentially in order to build larger queues.

A generic $N \times N$ queue can be constructed by building a butterfly network out of the basic 2×2 queues as shown in Figure 4.23. Thus the design is easily expandable with no increasing fan-in and fan-out overhead with the area cost of $N \cdot \log N$.

MIMO queues can also be constructed by adding handshake activators to the 2×2 arbiter design shown in Figure 4.9b. The basic difference, however, is the non-blocking property of this design, which allows the concurrent use of both handshake activators and has a significant impact on the performance of the whole network.

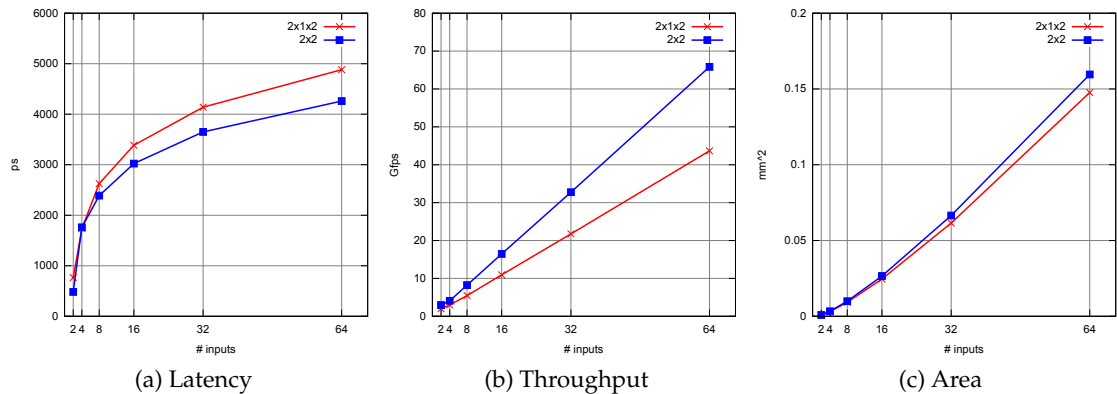


Figure 4.24: MIMO performance

4.6.1 MIMO Performance Comparison

To demonstrate the performance of the presented circuit, the design was simulated in Verilog VCS compiler. It is based on CMOS $90nm$ standard library cells and includes wire delays back-annotated from the place and route step in Cadence. Each channel is constructed to propagate 16-bit flits of data.

Two designs are compared: 1) the original MIMO design based on sequential 2×2 cells recreated from [80] (for clarity, those are called the $2 \times 1 \times 2$ cells), 2) the MIMO design based on parallel 2×2 cells presented here. The experiment is configured in such a way that the consumers and the producers work faster than the MIMO channel, effectively making the MIMO being the bottleneck of performance. The average latency (in picoseconds), average throughput (Giga-flits per second), and the cell area (mm^2) for various numbers of inputs are presented in Figure 4.24. The parallelism of separate cells has a significant impact on the throughput. The parallel MIMO cells have achieved around 10% latency improvement and 50% throughput improvement at a cost of about 8% of area overhead for each of the test cases.

4.7 Conclusions

This chapter describes the asynchronous design of an arbiter managing pairings between two clients and two resources. Each resource actively reports its availability and can be

connected to any of the clients. The initial STG of the circuit demonstrates the complexity of the conflict emerging among four incoming requests. One of the important features implemented is the arbiter's ability to establish parallel, non-conflicting pairings. In order to resolve the conflict, the original model is refined with a number of relatively fast internal arbitrations. The activation of the pairings is therefore sequential but the overall resource utilization is still concurrent.

During the initial design the Petrify tool was used to find implementations for separate circuit parts (such as the request controller and the grant controller), later it was also used to find different versions of the final design. Additionally, the overall circuit implementations were verified to be speed-independent by using the Puf and the Mpsat tool chain while all the STG and gate-level modelling was done in Workcraft.

A general solution for the multi-resource $N \times M$ arbiters is also described. The extended version of multi-resource arbiter was used by other researchers to improve NoC router throughput in [74]. Theoretically, it can be created for an arbitrary number of inputs and is decomposable into simple gate elements. In practice, this design may have tight limits on how many clients and resources it supports, mainly because of the quadratic area growth and the worst case latency.

The chapter further describes the creation of a multi-resource arbiter with passive resources. It is shown how the 2×2 arbiters can be used for creating multiple-input-multiple-output queues that help load balancing over multiple data processors. The results show 50% throughput improvement as compared with the original implementation in [80], the main reason for this improvement is allowing concurrent channel utilization.

Chapter 5

Gate-level Design Flow

5.1 Introduction

Modern circuit designers are challenged by the requirements of high performance solutions and reduced development time. Based on the size and complexity on the intended design, the optimal approach may vary. Devices with low transistor count and high demands for performance are likely to be modelled directly in transistors and use manual mask layout. The much more complex systems (such as hardware encoders) are more likely to be tackled by the behavioural specifications in such languages as TiDE (Haste) [79], Balsa [7, 75], VHDL, Verilog, and System-C. In the medium complexity spectre, there are various signal controllers that react to incoming events and, possibly, carry out basic computations (various schedulers, handshake components, data transceivers, etc.). These devices are quite complex for transistor level but often simple enough to be efficiently designed at the level of binary logic gates.

Gate-level designers both in digital and analogue domain must take into account timing of the signal propagation, in order to be able to tell whether the circuit would work in each of its states for any test case. Since signal timing is increasingly more difficult to predict and control [2], the reduction of timing assumption count is important for robust and flexible solutions. To ease these complications, designers create circuits that are less dependent on the delay of individual gates or wires. In other words, these circuits do not

depend on the timing assumptions and are able to work correctly regardless of how fast their individual components are. Checking that the circuit works correctly for any gate delay can be done by traversing through every reachable signal state. As the number of possible states grows exponentially with each new signal, highly effective methods must be used to perform the state exploration in reasonable amount of time.

The state exploration in a given PN or STG can be done automatically with tools such as *Petrify* [4] or the *Punf* and *Mpsat* [34]. As a result, it is possible to determine whether a given circuit model has deadlocks or whether the given STG represents a circuit with hazards. Additionally, the STG can be synthesized into a digital circuit implementation formed by a set of Boolean equations. The obtained equations are often more complex than the simple cells of a given technology. Thus, they need to be mapped to the given set of basic elements in such a way that no hazards would occur.

Unfortunately, larger STGs make this detailed model of signal transitions more difficult to design, which brings to a thought that such an STG could be formed structurally by combining higher level components. For such a formalism the model of interconnected Boolean logic blocks is proposed. It will be shown that this approach allows viewing the system at various degrees of abstraction and can help composing larger and more complex systems.

Workcraft is a plugin based modelling tool that helps creating various interpreted graph-based models (IGMs) [65, 64]. Any objects created within the graph models can be associated by directed connections between them. A circuit model plugin was implemented as a part of Workcraft development environment in an attempt to support the gate-level design flow. Its features are useful for creating new arbiters and are presented in this chapter.

5.2 Features of the Gate-level Design Flow

5.2.1 Basic Plugin Components

The plugin is based on the digital circuit models described in Chapter 2. For clarity of model presentation and ease of use, the modelling plugin provides the following visual elements:

1. *Circuit components* that group related signals together. Their basic function is to visually present signals and their connections. Their *output contacts* are the actual signals of the circuit. The input contacts of a circuit component are effectively the *placeholders* for other signals. They are used as arguments for the functions within the name space of the component and are later associated with the actual signals by connections.
2. The *input ports* and *output ports* that describe the interface of the circuit. Here, the input ports act as signals, and the output ports are the placeholders (same as inputs to the circuit components).
3. *Connections* (or wires) associating signals with corresponding signal placeholders. There may be many outgoing connections from one signal. But not more than one connection may arrive to a placeholder. Circuit *joint* components can be used to branch connections at convenient places on the diagram.

Each of the circuit signals (both the input ports and the output contacts) is specified by the dedicated set_{v_i} and $reset_{v_i}$ functions, so that $f_{v_i} = \overline{f_{v_i}} \cdot set_{v_i} + f_{v_i} \cdot \overline{reset_{v_i}}$. In other words, set_{v_i} specifies the condition, when signal f_{v_i} activates to 1, and $reset_{v_i}$ when signal resets to 0. To avoid signal oscillation, in any circuit state, set_{v_i} and $reset_{v_i}$ should never be true at the same time: $\forall_{v \in \mathcal{V}, s \in \mathcal{S}} : set_v(s) \cdot reset_v(s) = 0$.

On the diagrams, set and $reset$ functions are denoted with the vertical arrows \uparrow and \downarrow followed by the Boolean formula respectively. When there is a formula f with no vertical arrow, it means the $reset$ function is the negation of a set function: $set = f, reset = \overline{f}$.

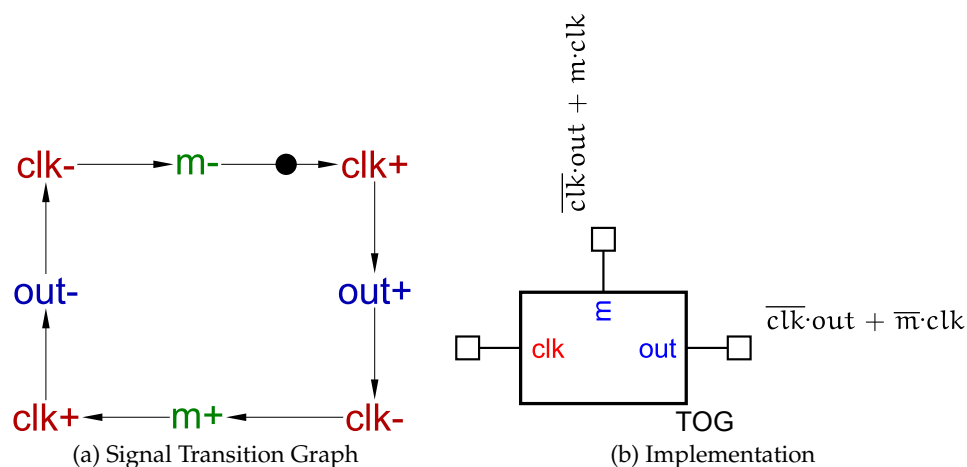


Figure 5.1: Toggle component

Basic Examples

Consider the design of a toggle element (Figure 5.1). The STG diagram shows that it has one input signal clk , one output out , and one internal signal m used as the local memory. It specifies that on every positive clock edge $clk+$ the circuit changes its output to the opposite value. The clock signal itself is not enough to determine whether the signal should rise or fall with the next clk change; therefore, the signal m is used to store that information.

The circuit can be implemented with two complex gates as shown in Figure 5.1b. Gate signal functions are $m = \overline{clk} \cdot out + m \cdot clk$ and $out = \overline{clk} \cdot out + \overline{m} \cdot clk$. The component may have multiple inputs (placeholder contacts) and more than one output (signal contacts). The internal signal m acts as memory and is not driving anything. However, here an assumption is made that the clk signal will not change before m transitions.

In the next example, a model of a MUTEX element is depicted in Figure 5.2.

Both signals $g1$ and $g2$ are specified with corresponding *set* and *reset* functions:

$$g_{1,2} = \begin{cases} \uparrow & r_{1,2} \cdot \overline{g_{2,1}} \\ \downarrow & \overline{r_{1,2}} \end{cases}$$

Figure 5.2b shows how the MUTEX element is connected to its environment. Ports $input1$ and $input2$ are the input signals and their behaviour is constrained by the state of

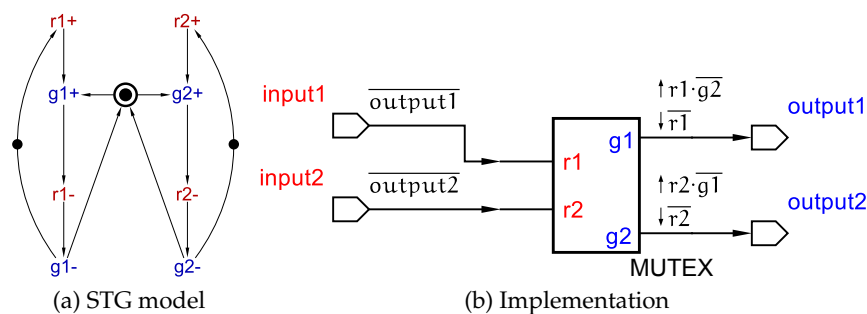


Figure 5.2: MUTEX element

the output ports *output1* and *output2*. Essentially, it is an example of 4-phase handshake executed on both of the MUTEX channels.

5.2.2 Gates of High Complexity

It is possible to model a system from a high level perspective. Its main idea is to capture circuit behaviour without showing its low-level implementation. There may be multiple different decompositions which may or may not work for their own reasons. Meanwhile, if the initial system contains a hazard, its decomposition would still have the same problem. Therefore, launching the verification technique at an early state will allow to catch out hazards at an early design stage.

There are no restrictions on the complexity of gates used in the circuit model. By entering appropriate functions, Workcraft allows elements with arbitrary numbers of inputs. Regardless of a signal's complexity, its value would only change by a single + or – transition. Figure 5.3 shows high complexity gates as an example. The specification of the components is as follows:

- C-element: $c1 = \begin{cases} \uparrow & a \cdot b \cdot c \cdot d \cdot e \cdot f \\ \downarrow & \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot \bar{d} \cdot \bar{e} \cdot \bar{f} \end{cases};$
- complex gate: $c2 = (a \cdot b + c \cdot d + f) \cdot \bar{e};$
- reset-dominant latch: $c3 = \begin{cases} \uparrow & (a + b + c) \cdot \overline{done} \\ \downarrow & done \end{cases}.$

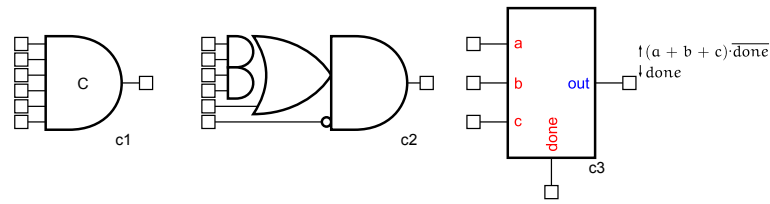


Figure 5.3: High complexity gates

By constructing the system at the level of high complexity gates, the top to bottom design approach is followed. First, the task can be represented with large circuit blocks separated only by their responsibility. Then, gradual refinement is used to approach a particular gate-level implementation. The decomposition can be done by any accessible means such as manual design as well as the automated mapping into particular technology. Once decomposed, the design can be checked for hazards in an automated manner.

5.2.3 Delay-Insensitive Circuits

The main assumption about the Speed-Independent circuit is that wires do not have delay. In real systems where one gate signal is forked to drive two or more gates, the actual arrival of the signal may happen at different times. Such a situation may take place due to multiple reasons: the length of the forked wires is different, each branch of a signal is connected to a transistor with a different size, cross-capacitance in wires slows down or speeds-up the signal propagation, etc. For small connection distances these effects may be negligible and thus lie within the same *equipotential region* [69]. For the longer interconnects in larger designs, all of these factors will have an increasingly noticeable impact and, hence, need to be modelled explicitly. An ideal solution would be to create designs that work for arbitrary wire delays; however, building pure delay-insensitive circuits limits the designer to a fairly small set of implementable circuits [46].

The delay-insensitivity of the circuit can be checked by modelling arbitrary delay on wires. This can be easily modelled by introducing additional buffer components on the wires where wire delay is considered unpredictable. Such buffers are not needed in wires with no forks because this timing unpredictability is already included in the signal driving the wire. For forked wires, that may propagate a signal in interchangeable order,

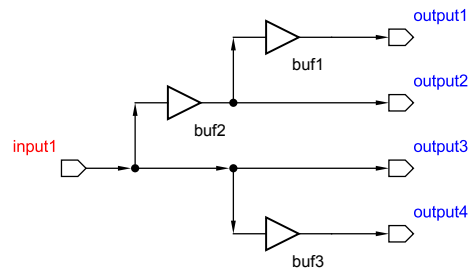


Figure 5.4: Modelling asymmetric forks

a buffer needs to be inserted for each branch.

A more practical assumption could be that one wire branch is not slower than another. These *asymmetric forks* can be modelled by only placing the buffer on branches that are not the fastest. An example on Figure 5.4 shows buffer configurations that partially reduce unpredictability in wires. As with isochronic fork, all of the outputs may receive the *input1* simultaneously. However, the *output3* will never be behind any other outputs (for instance, when this wire is the shortest and connected to the lightest load). Similarly, the *output2* will always be before *output1* implying also *output3* before *output1*. The timing between branches *output2* and *output4* is not known and behaves as in delay-insensitive circuits.

5.2.4 Circuits with Timing Assumptions

Timing assumptions, when adequate, are helpful for improving the complexity and the performance of the circuit. This section describes how *relative timing assumptions* can be used in the model. A relative timing assumption in the logic gate model is an additional signal transition constraint, that assures a Boolean expression is met before firing the associated transition. The main difference, however, is that the constraint does not result in any new inputs of a component, which would increase its complexity.

A simple example of a counter constructed of toggle components is shown in Figure 5.5.

The *clock* signal is not constrained by any signals, therefore its value may be seen as a glitch if the toggle components are not quick enough. The condition that all of the circuit signals have settled can be specified as a constraint for the *clock* to change. As shown in

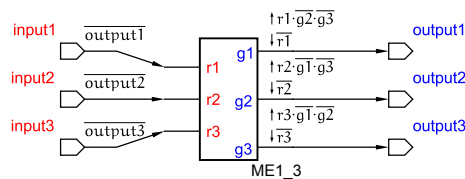


Figure 5.6: 3-input arbiter (high level model)

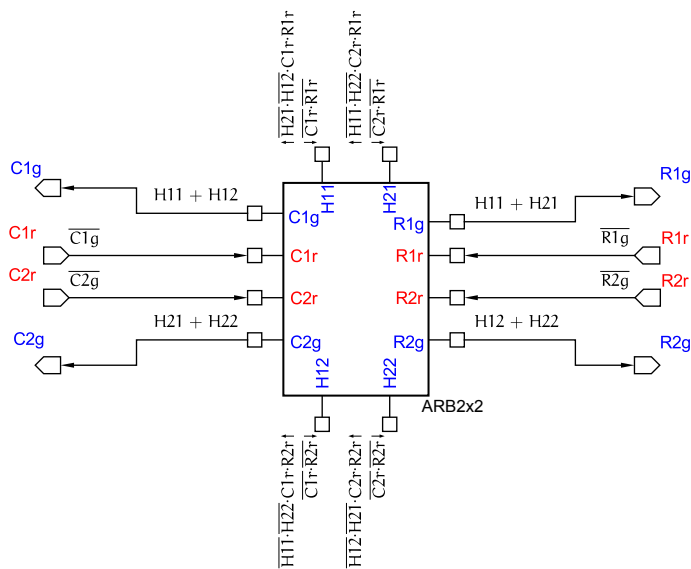


Figure 5.7: High-level view on 2×2 arbiter

5.2.5 High-level Models

An important feature of the logic gate models is the ability to represent component behaviour at various degrees of abstraction. A simple 4-phase 1-of-3 arbitration component may have various implementations (token-ring arbiter, arbitration tree, arbiter mesh, e.t.c. [35]). However, the basic functionality of the component states the same: when there is a request, it should provide at most one grant signal at a time. Hence, this functionality can be modelled directly without introducing the complexity of any specific implementation (Figure 5.6).

The next example of the high-level presentation is the 2×2 arbiter described in Section 4 (Figure 5.7). The model in Figure 4.4 specifies the expected behaviour of the environment and the behaviour of the arbiter without showing the details of implementation, this is an alternative presentation for the STG.

When dealing with larger models, it is also useful to merge multiple simpler gates

into a single complex gate (component) and form a simpler model, which would sustain the same behaviour and have a smaller complexity.

5.2.6 State Space Exploration

The flow presented in this chapter is based on the traversal of Petri net unfoldings. As a designed circuit is converted into its STG model, the tools Puf and Mpsat are used for state space exploration [5, 33].

When the behaviour of each signal is defined, the process of circuit conversion into its STG form is done with the following steps:

- First, a pair of places is created for each signal in the circuit. One place p_1 is used to represent value 1. The other place p_0 would represent the value 0. Exactly one token is placed in either 0- or 1-place (p_0 or p_1) depending on the initial signal state. During the simulation, this token will travel from p_0 to p_1 and back when one of the corresponding *set* or *reset* transition is executed.
- After creating contact places, all the *set* and *reset* functions are converted to their Disjunctive Normal Form (DNF) $set = cs_1 + cs_2 + \dots + cs_n$ and $reset = cr_1 + cr_2 + \dots + cr_m$, where cs_i and cr_j are the conjunctive clauses. Then corresponding rising and falling transitions t_{1+}, \dots, t_{n+} and t_{1-}, \dots, t_{m-} are created for each of the conjunctive clauses.
- At the final stage, all of the generated transitions are constrained by connecting them to the generated place pairs with read arcs. For instance, the set function $set = a \cdot (b + c)$ would be converted to the DNF form $set = a \cdot b + a \cdot c$ and then produce two *set* transitions for clauses $t_{1+} \leftarrow a \cdot b$ and $t_{2+} \leftarrow a \cdot c$, where a , b and c are the places of corresponding signals.

An example of modelling the MUTEX element from Figure 5.2b is shown in Figure 5.8. Here, the transition $g1+$ (shown as *MUTEX_g1+*) is constrained by signal states $input1 = 1$ and $g2 = 0$. Note how the placeholder contacts ($r1$, $output1$, ...) have disappeared from the model, they are now only shown as comments under the generated

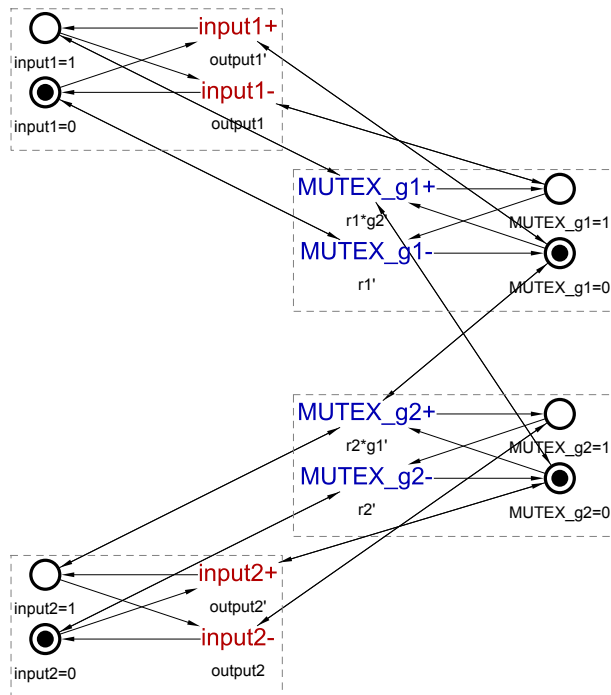


Figure 5.8: Generated MUTEX STG

transition labels for signals $g1$ and $g2$.

Mpsat verification flow

The Mpsat verification flow is shown in Figure 5.9. From a circuit defined by Boolean equations its circuit STG can be generated with the plugin. The generated STG is always 1-safe and can be analysed by Punf [33] in order to create STG unfolding. The unfolded model is used by the Mpsat tool to check the STG for the properties specified separately. Based on the provided STG unfolding and the list of markings being searched, Mpsat either reports a trace that leads to the specified marking or reports that this marking is not reachable. There is a special Mpsat *reach language* that allows the automatic generation of markings for a given unfolding and a property [34]. The programs in this language are able to describe various STG features, including deadlocks and non-persistency.

The next example demonstrates the verification of a C-element decomposition (Figure 5.10). This is a NAND-based C-element implementation proposed by Maevsky. Both $input1$ and $input2$ are constrained by the $output$ as if they were connected through two

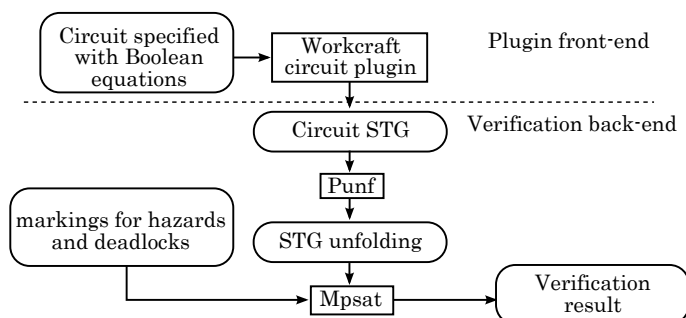


Figure 5.9: Mpsat verification flow

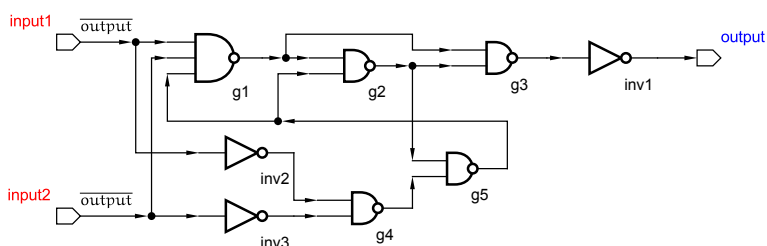


Figure 5.10: C-element formed of NAND gates

inverters. The STG generated from the circuit is shown in Figure 5.11.

When such a model is tested for hazards, the following trace is returned:

$input1+ \rightarrow input2+ \rightarrow g1- \rightarrow inv3- \rightarrow g2+ \rightarrow g4+ \rightarrow g5- \rightarrow g1+ \rightarrow g3- \rightarrow inv1+ \rightarrow input1-$. The hazard occurs because the transition $inv2-$ enabled by $input1+$ is again disabled by $input1-$ at the end of the trace. In practice this inverter is likely to be faster than the sequence of events from $input1+ \rightarrow \dots$ to $\dots \rightarrow input1-$. However, strictly speaking it is not a speed-independent circuit and the timing assumption that $input1+ \rightarrow inv2-$ is faster than $input1+ \rightarrow \dots \rightarrow input1-$ should be explicitly stated as a necessary condition for correct operation.

5.2.7 Circuits with MUTEX Elements

Traversing through the circuit state space with a MUTEX element will always find a hazard because its outputs are non-persistent by design. When both MUTEX requests arrive, both grant signals become excited. Then, after the first grant signal fires, the second is disabled, which creates a hazard.

In digital circuits, the environment is modelled with various high-level techniques,

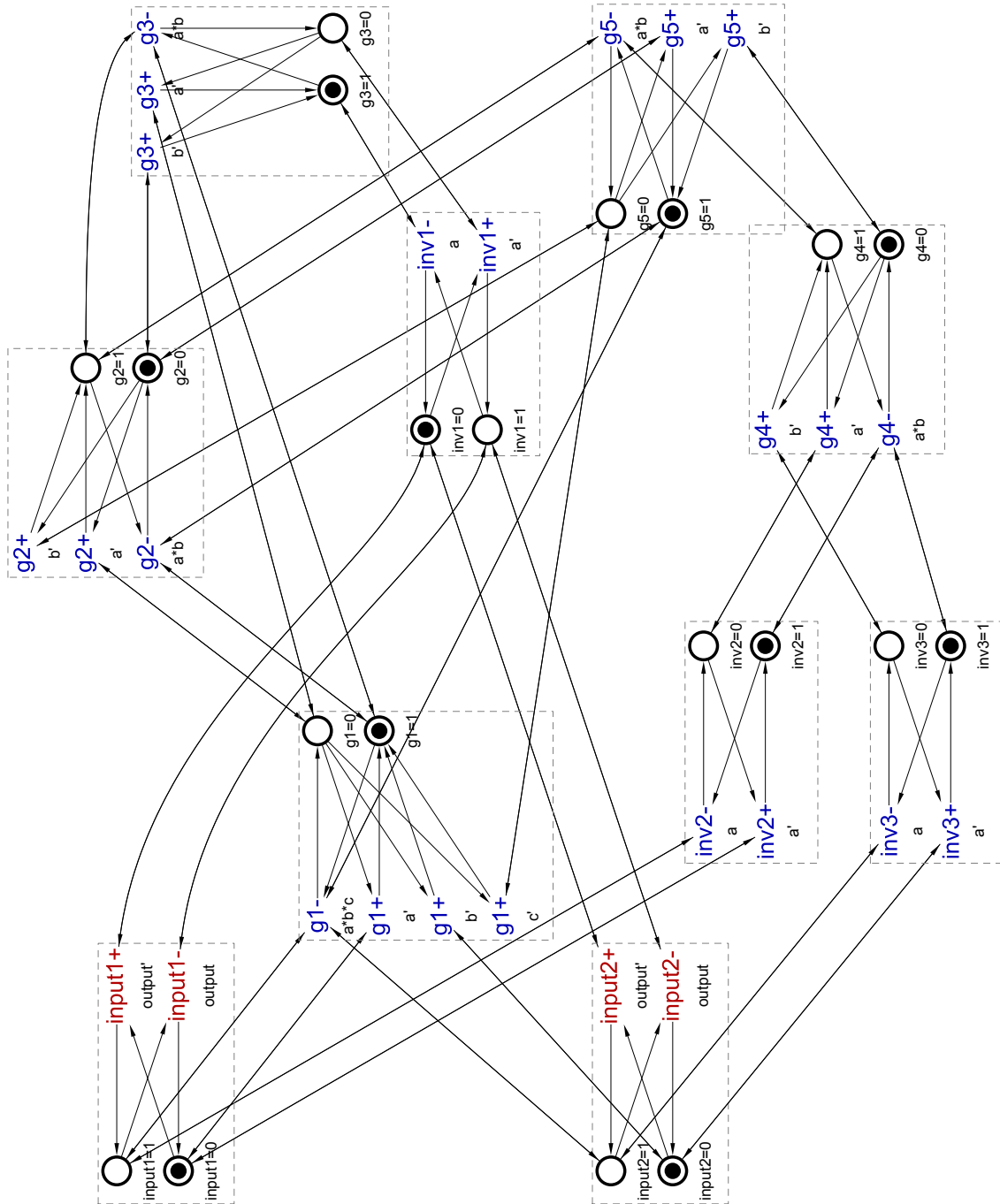


Figure 5.11: NAND-based C-element STG

the only constraint is its obedience to certain well defined communication protocols. It may contain hazards or non-deterministic choices, which do not make a difference until the moment the environment actually changes one of the circuit inputs. In other words, it does not matter whether environment signals are non-persistent; however, if it is not safe from glitches, the internal signals may become non-persistent.

From this point, the logical solution for modelling MUTEXes is modelling them as part of the environment.

5.3 Analysis of Priority Arbiter

This section presents the analysis of the Priority arbiter described in [11], which is also mentioned earlier in the review (Chapter 3).

The arbiter is constructed as shown in Figure 5.12. It's inputs follow the active 4-phase handshake protocol. The set-dominant latches with the inverted reset inputs $SR1$, $SR2$, $SR3$ are represented with corresponding set/reset Boolean equations:

$$q = \begin{cases} \uparrow & s \\ \downarrow & \bar{r} \cdot \bar{s} \end{cases}$$

The MUTEX elements $ME1$, $ME2$, $ME3$ are modelled as part of the environment and are shown with the dashed lines. The priority module $PRI0$ has 6 inputs corresponding to the “won” and “lost” arbitration of the MUTEX elements $w1, l1, w2, l2, w3, l3$. Its task is to wait until three out of six input signals become active, and then activate one of the corresponding grants: $g1, g2, g3$. All basic components such as AND-gates, OR-gates, and C-elements are shown straightforward, with the corresponding logic gate symbols. The $LOCK$ component implementation is shown different from the original design in Chapter 3. It has the 3-input OR gate and the reset-dominant latch combined into a single complex gate. The composition into a single gate was done intentionally to avoid a hazard, which will be shown later in the section.

As soon as the circuit is created within the tool, it can be formally verified with the

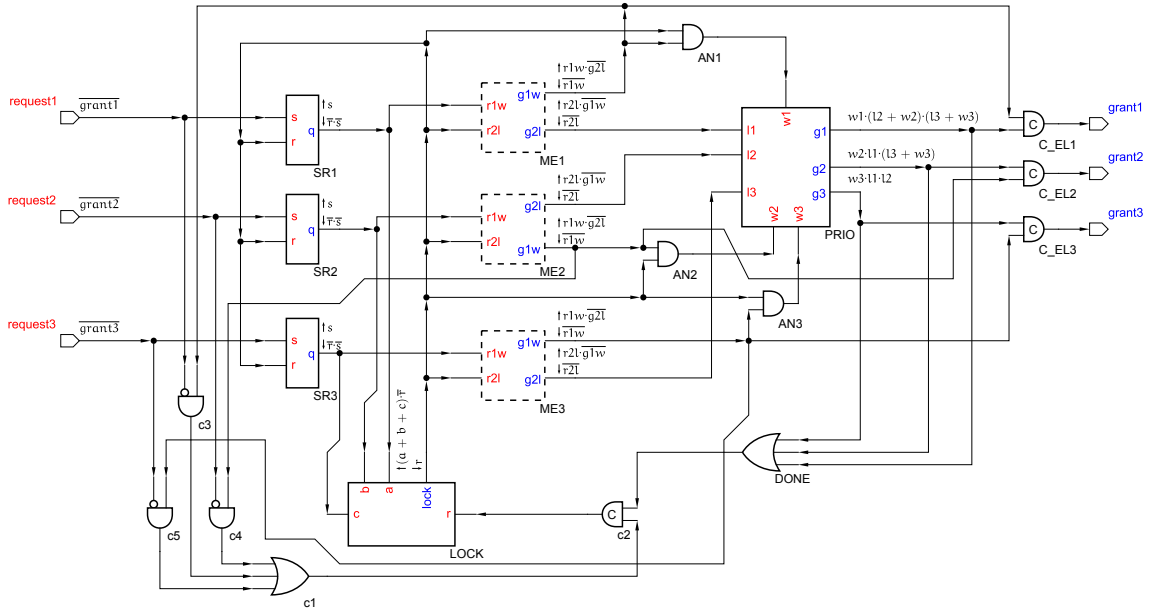


Figure 5.12: Modelled priority arbiter

method described above. However, without specifying the necessary timing assumptions, there will be hazards and deadlocks found as a result.

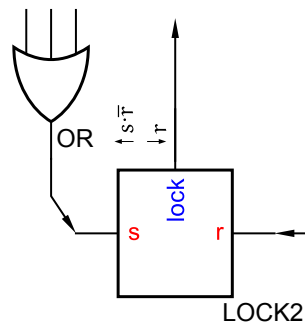
One timing assumption mentioned in [11] was that at least one $g1w$ signal manages to win the arbitration in $ME1$, $ME2$, or $ME3$. In other words, there is no situation, when all of the MUTEX elements grant “lose” signals $g2l$, which is a reasonable timing assumption meaning that the path from each SR component to the corresponding $ME.r1w$ input is shorter than the path $SR \rightarrow LOCK \rightarrow ME.r2l$.

To avoid the state when all MUTEX arbitrations are lost, the relative timing assumptions for the rising transitions on MUTEX elements are added:

$$\begin{aligned}
 ME1.g2l \uparrow & : ME2.g1w + ME3.g1w \\
 ME2.g2l \uparrow & : ME1.g1w + ME3.g1w \\
 ME3.g2l \uparrow & : ME1.g1w + ME2.g1w
 \end{aligned}$$

They suppress the activation of the third $g2l$ signal and in the context of the $PRIO$ module used, avoid the deadlock.

If the model is checked, there will be a hazard found in each of the AN gates. The

Figure 5.13: *LOCK* decomposition

problem occurs because the reset phase of the *AN* gate is not acknowledged by the priority module *PRI0*. For an extremely slow transition $AN1-$, the reset phase of the *lock* signal $lock-$ would be able to propagate through the environment and begin a new transaction with the $request1+$, which, again, may eventually end up with $lock+$ and $ME1.g1w+$. At that point it would disable transition $AN1-$ and can be seen as a glitch produced by gate *AN1*. Based on the number of events that have to happen before the hazard occurs, it is reasonable to assume that $AN1-$ will occur before the next $lock+$. Hence, another timing assumption may be added safely:

$$LOCK.lock \uparrow : \overline{AN1} \cdot \overline{AN2} \cdot \overline{AN3}$$

Now consider the decomposition of the complex *LOCK* component into the 3-input OR gate and the reset-dominant SR-latch *LOCK2* the way it was done in [11] (Figure 5.13). Since the initial request may come at any time on any of the request signals, the positive edge of the request may arrive at the OR gate at any time. This means that whenever the $lock-$ transition is about to disable the OR gate, this may happen simultaneously with another request enabling it at the same time. This hazard cannot be eliminated by adding a timing assumption because by the definition of the arbiter, there are no relative timing assumptions between independent client requests. Hence, to avoid the hazard, the arbiter structure needs to be changed.

5.4 Conclusions

The approach of designing circuits based on Boolean functions allows avoiding complexity of STGs by structurally dividing a circuit into simpler components. The new plugin implemented in Workcraft EDA allows designing digital circuits, where each signal is represented with Boolean equations limiting its behaviour.

The behaviour of each signal can be specified by two separate equations regarding its *set* and *reset* conditions. These equations are not explicitly limited by the number of inputs, therefore, gates of arbitrary complexity can be created.

It is possible to mark functional blocks to be treated as part of the environment, which allows specifying components with internal conflicts such as MUTEX elements.

The method of designing circuits directly with gates is slightly more limited than what STGs can describe. Any digital circuit can be converted into its STG equivalent; however, there are STGs with CSC conflicts, demonstrating that the scope of STGs is wider than the digital circuits. Luckily, the circuits composed of gate-level components never create CSC conflicts, which is an additional reason to prefer using the gate-level flow when dealing with circuits of higher complexity.

Chapter 6

Design of Generalized Arbiter

6.1 Introduction

The multi-resource arbiter presented in Chapter 4 is limited to solving the arbitration conflict among multiple resources providing an identical service. This may be insufficient when certain resources offer a kind of service that is only useful to a subset of clients. At the same time a subset of resources may still be suitable for a particular group of clients, which creates a problem of *semi-interchangeable resource* allocation.

One practical example for such an arbitration is building a network routing component that selects a propagation path depending on the availability of its output ports and the list of accepted ports selected by a client. The client selects multiple suitable resources and the arbiter provides a grant when at least one of the suitable resources becomes available. In practice this may occur with data packets travelling diagonally in a two-dimensional mesh of routers. Depending on the output port availability, each routing component may propagate a packet either vertically or horizontally while reducing the distance towards destination regardless of the path chosen. Other topologies also provide various degrees of *path redundancy*, which can be used to improve performance and reliability of the communication network under development.

The problem of semi-interchangeable resources can be also viewed as the committee arbitration where committees are associated with more than one resource group. An

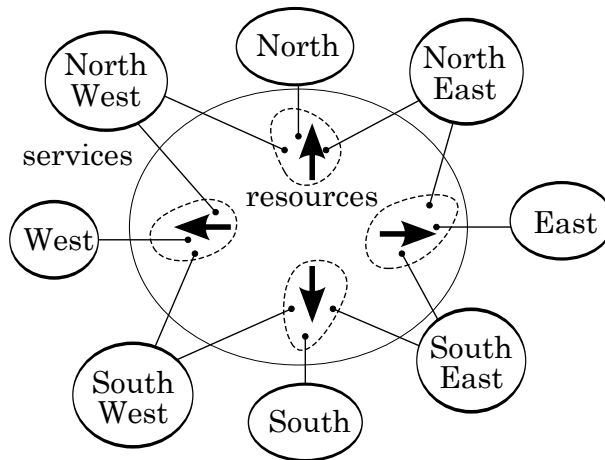


Figure 6.1: Arbitration example in 2D routing grid

example in Figure 6.1 presents a routing component, where each diagonal committee is associated with two interchangeable professors; however, a more general purpose arbiter implementation is presented to address this problem. The scalable, low-latency design outlined in this chapter is based on the ideas of the priority arbiter and the gate-level flow presented in Chapter 5, where the arbitration is split into separate phases of synchronization and grant management.

6.2 Arbiter Design

6.2.1 Design Method

The design method is similar to the one described in Chapter 4. First, using high-level specification, the MUTEX elements are factored out from the rest of the logic. The result is the structural model of high-level circuit components (possibly with timing assumptions) and a number of MUTEX elements that are used to hide metastability.

Then, the MUTEX elements are presented as part of the environment, separating them from the signals that have to be checked for hazards. If at this stage the formal verification procedure detects any deadlocks or hazards, the factoring phase is repeated.

When the high-level model is ready and functioning, the implementations for each of the components and their decomposition into simple logic gates can be found with synthesis tools. This process may be fully automatic or partly done by hand. The manual

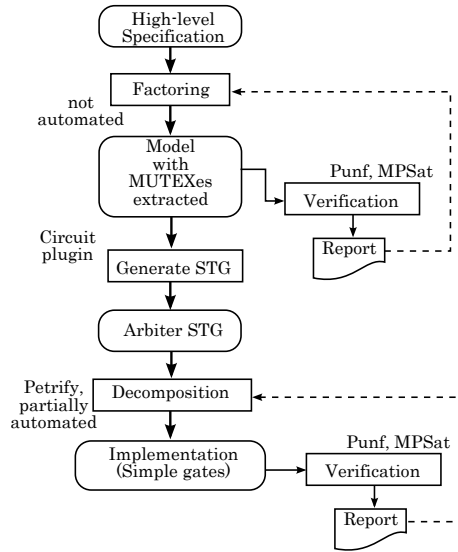


Figure 6.2: Arbiter design flow

decomposition helps to enforce structure among new signals which is important for finding scalable solutions. Since during this phase new hazards may have been introduced, the verification phase is launched again.

6.2.2 Basic Structure

The basic structure of the generalized arbiter is borrowed from the priority arbiter [11] where the *grant phase* is separated from the *synchronization phase*. The Figure 6.3 presents the high-level view of the problem. When at least one request arrives, it passes through the column of MUTEX elements and activates the *lock* signal. Once *lock* has fired it disables further request propagation and activates the arbitration procedure. The outcome of this arbitration may depend on how many requests have arrived. The model does not specify whether signals *grant1*, *grant2*, or *grant3* will fire after being enabled in the same transaction; the arbitration activity depends on the arbitration logic being implemented and is not covered in the diagram. During the arbitration one or more grants may be issued as a result. Eventually, when the arbitration is finished, the *done* signal disables any further grants while returning the arbiter into its initial stage ready to accept more requests. Note how the pairs of events r_1, r_2, r_3 are used in order to keep this diagram 1-safe. The first of the requests moves the token from p_1 to p_2 , which switches the en-

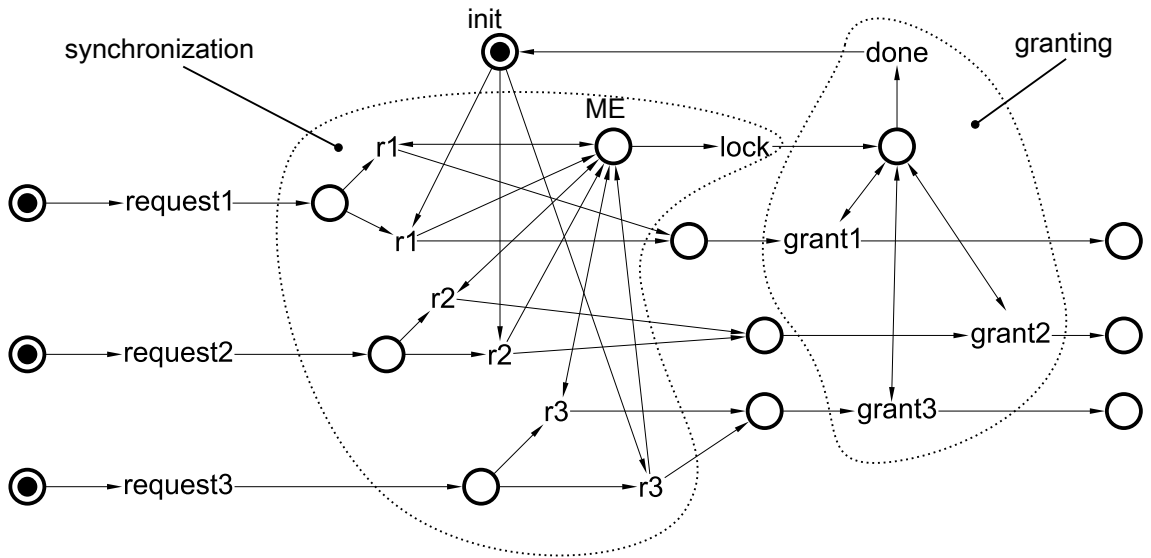


Figure 6.3: Generalized arbiter high-level PN

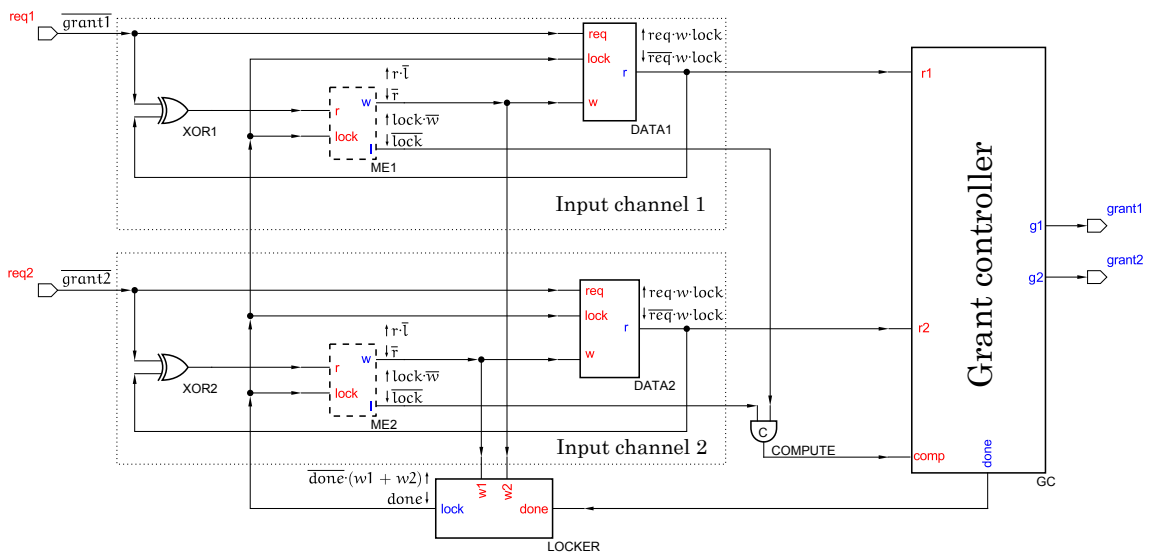


Figure 6.4: High-level circuit structure

abled r_i transitions so the second and the third requests will not add more tokens to p_2 , so there is only one *lock* event per arbitration transaction.

The choice of this structure is motivated by its good scalability as additional client rows can be easily added to the model without increasing its depth.

A high-level circuit model of the generalized arbiter with the MUTEX elements separated from the rest of the logic is presented in Figure 6.4. It is a simple example of an arbiter with only two request signals *req1* and *req2* and two grant signals *grant1* and

grant2. The activation constraints applied to the signals *req1* and *req2* implement the rules of the 4-phase handshakes. The arbiter operates in two stages. At the first stage, the column of MUTEX elements is locked fixing each of the request values. At the second stage the arbitration decision is made based on the state of requests. Communication with the environment is ensured through 4-phase handshakes, where each input request signal *req* from a client corresponds to one output grant signal *grant*. The *req+* and *req-* transitions are used by the environment to request and release a resource. The *grant+* signifies when the resource was granted and *grant-* returns the handshake to the initial state and signals that the client is allowed to start the next transaction.

The main components of the arbiter are the *Input channels*, the *LOCKER*, and the *Grant controller* (Figure 6.4). The input channels are used to store the state of all the requests and synchronize the environment request signals with their internal states. The *LOCKER* component activates synchronization when at least one of the input signals has changed its state. The grant controller is then activated to resolve any conflicts based on the request information provided by the input channels.

Process of Arbitration

Consider how the requests propagate through the design. Initially, all of the circuit signals are low. The XOR gates detect any of the environment requests changes and interpret those changes as a new requests for arbitration by activating the $ME_{j,r}$ lines.

Suppose the basic 1-of-2 arbitration is being considered and the $req1+$ was issued. This request propagates through the MUTEX element: $req1+ \rightarrow XOR1+ \rightarrow ME1.w+ \rightarrow LOCKER.lock+$. At this point, the *LOCKER* component begins synchronization: $LOCKER.lock+ \rightarrow ME2.l+$ preventing propagation of new requests beyond the *ME2* and unlocking the state update on the *DATA1* component. The set/reset expressions of the *DATA1* component will align its output *DATA1.r* with the current value of *req1* whenever both *DATA1.lock* and *DATA1.w* are active:

$$DATA1.r = \begin{cases} \uparrow & req \cdot w \cdot lock \\ \downarrow & \overline{req} \cdot w \cdot lock \end{cases}$$

After the alignment phase is complete, the XOR gate will release the request from the MUTEX component, consequently letting the $ME1.l+$ transition to take place: $LOCKER.lock+ \rightarrow DATA1.r+ \rightarrow XOR1- \rightarrow ME1.w- \rightarrow ME1.l+$.

For any number of input channels, the special state with all the MUTEX elements holding their $ME_j.l$ signal active will signify the end of synchronization. The input signals $r1$, $r2$, and $comp$ in the grant controller GC will form a *bundled data channel*, where $comp$ is the request line, and $r1$, $r2$ are the data lines.

When $GC.comp+$ is triggered, the task of the grant controller is to change the state of the grant signals according to the data provided through the input channels.

The design of the grant controller can be created by modelling a *finite state machine* (FSM) that updates its state according to the request state information provided whenever the $comp$ signal is raised, which acts as a clock signal. The timing constraint here is that the transition $comp+ \rightarrow done+$ happens after all of the grant signals have settled and are presenting the new FSM state. After the arbitration is completed, the signal $LOCKER.lock$ is disabled by $done+$. This finishes the current arbitration transaction, eventually leading to $GC.comp-$ and $GC.done-$.

The important thing to note here is that both requests $req1+$ and $req2+$ may arrive simultaneously and both propagate as “win” through the column of MUTEX elements. In this case, the grant controller will receive both requests and release grants according to its implementation, be it priority arbitration or some other policy.

Another important case study is when new requests arrive after the arbitration was started. These requests will be blocked by the locked component until the arbitration is finished. However, once MUTEXes are released, all of the requests will propagate in the next arbitration transaction.

According to the protocol, the release of resources is eventually followed on the client side: $req1+ \rightarrow grant1+ \rightarrow req1-$. By the XOR components it will be interpreted as a

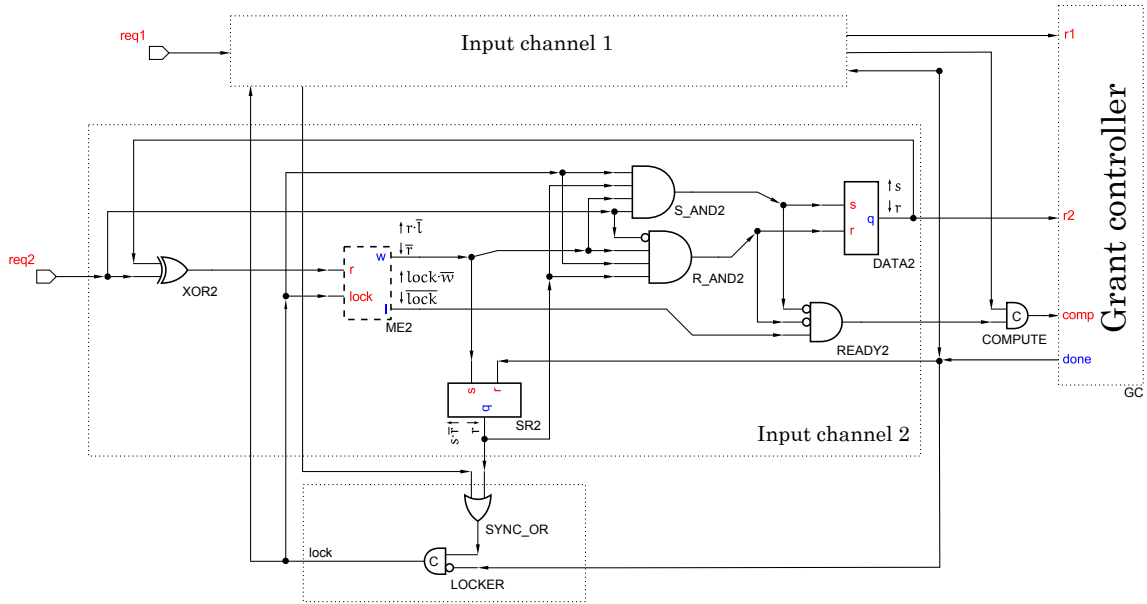


Figure 6.5: Decomposition into simple gates

new request, eventually igniting $ME1.w+$. It means that the arbitration is started every time the resource is requested or released.

6.2.3 Decomposition

The scalable speed-independent decomposition of the arbiter is shown on Figure 6.5. The input channel can be constructed with three AND gates (S_AND2 , R_AND2 , $READY2$), one SR-latch ($DATA2$) and one reset-dominant SR-latch ($SR2$).

The component initiating synchronization consists of an OR-gate ($SYNC_OR$) collecting requests from the input channels and the C-element ($LOCKER$) issuing the $lock$ signal.

Arbiter Scaling

Arbiter scaling up to N clients is straightforward. Since each of the input channels is formed as the *true tile* [72] meaning that it can be reused for any number of inputs without changing its interface and its internal structure.

For N clients there will be N inputs for the $COMPUTE$ C-element and N inputs for the $SYNC_OR$ OR gate. This C-element can be safely decomposed into a tree of smaller

C-elements or have any other implementation used in the *completion detection* circuits.

6.2.4 High Performance

Because of the significant synchronization overhead, the arbitration latency may seem large for the occasional requests scattered over time. However, the variability of arbiter latency is expected to be close to a constant for any number of requests activated (providing there are enough resources to satisfy clients). The high performance of the arbiter is achieved by the fact that all of the pending requests are processed in a single computation transaction concurrently resolving all conflicts in one go. This can be an important feature for high performance designs dealing with bursty request environments.

The task of the grant controller is managing grants upon receiving the $GC.comp+$ transition, which can be viewed as a clock signal triggering computation. It allows designing the grant controller in a flow similar to the synchronous design. At the same time, the computation is only launched when there are requests pending, meaning that dynamic power would not be consumed when the circuit is idle.

6.2.5 Avoiding Deadlocks

The arbiter does not have deadlock states if the grant controller is implemented correctly. Arbiter request signals are causally related to the state of the grant signals by obeying the 4-phase protocol rules. On the other hand, the arbitration can only be started after an activity on the request lines.

In the simple example above, the grant controller may have an incorrect implementation, which is prone to stalling the arbiter. For instance, when the grant controller receives a request from the second client $req2+$, it is consequently granted with $grant2+$. While the resource is being used, the first client may also initiate its request $req1+$, which is not granted and becomes pending because the resource is busy. Eventually, the resource is released with $req2-$. The arbiter logic may have a flaw that will result in ignoring $grant1+$ (because the resource is still busy), and during this transaction only responding by $grant2-$, thus acknowledging the resource release to the second client. The arbiter

now knows that the resource was released; however it still needs transitions on the request lines to begin the new arbitration. Because the first client has already sent its request, it will be waiting for a response. This stalls the arbiter until the next activity on the *req2* line. Hence, to avoid stalling, the grant controller must always provide all of the new grant signals that became available during the same transaction.

6.2.6 Circuit Verification

The generalized arbiter was formally verified by methods described in Chapter 5.

Decomposition

The circuit STG is generated from the decomposed circuit in Figure 6.5. The grant controller is defined in a way that its outputs $GC.grant1$, $GC.grant2$ may arbitrarily align with the inputs $GC.r1$, $GC.r2$ while the condition $GC.comp \cdot \overline{GC.done}$ is true:

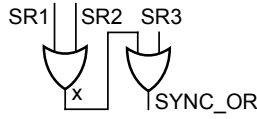
$$GC.grant1 = \begin{cases} \uparrow & comp \cdot \overline{done} \cdot r1 \\ \downarrow & comp \cdot \overline{done} \cdot \overline{r1} \end{cases}$$

and similarly:

$$GC.grant2 = \begin{cases} \uparrow & comp \cdot \overline{done} \cdot r2 \\ \downarrow & comp \cdot \overline{done} \cdot \overline{r2} \end{cases}$$

In other words, any combination of grants is permitted for any combination of requests. This creates an opportunity to run the arbiter into a deadlock state when both requests for resource were ignored ($r1 = 1, r2 = 1$, and $grant1 = 0, grant2 = 0$) or when both resource releases were ignored ($r1 = 0, r2 = 0$, and $grant1 = 1, grant2 = 1$).

The signal $GC.done$ always aligns itself with the $GC.comp$: $GC.done = GC.comp$. By


 Figure 6.6: *SYNC_OR* decomposition

adding the following set constraint, the deadlock states can be avoided:

$$GC.done = \begin{cases} \uparrow & comp \cdot \overline{(r1 \oplus grant1) \cdot (r2 \oplus grant2)} \\ \downarrow & \overline{comp} \end{cases}$$

So, the *GC.done+* transition is only allowed when at least one grant signal is aligned with its input request and more request activity is expected eventually.

After applying these constraints, the automated verification phase was successful, showing no deadlocks or hazards found.

Arbiter Scaling

Another question arises when the arbiter is scaled to support three inputs or more. The problematic element here is the *SYNC_OR* gate receiving the MUTEXed requests for arbitration. For an increasing number of clients, the OR gate eventually will have to be split into smaller gates, forming a tree of OR gates. Consider a decomposition shown in Figure 6.6. Two requests *SR1* and *SR3* may activate simultaneously, igniting the transition of the OR gates. It is sufficient for *SR3* alone to be present in order to activate the *lock* signal followed by the computation in the grant controller. Now, assume the transition *x+* is particularly slow and does not happen up until *GC.done+*. As the *GC.done+* fires, latches *SR1*, *SR2*, *SR3* are reset back to 0 while also disabling the unacknowledged transition *x+*, which creates a hazard condition. In any practical circuit the timing between *x+* and *done+* is easy to achieve because *done+* timing will be postponed by the relatively slow *COMPUTE* signal and the arbiter computation itself.

To model the timing assumption, it is sufficient to additionally constrain the *done* signal:

$$GC.done = \begin{cases} \uparrow & comp \cdot \overline{(r1 \oplus grant1)} \cdot \overline{(r2 \oplus grant2)} \cdot \overline{(SR1 + SR2)} \cdot \bar{x} \\ \downarrow & \overline{comp} \end{cases}$$

Once the timing assumption was ensured, the verification phase was successful, showing no deadlocks or hazards found.

Asymmetric forks

The design is largely speed-independent, meaning that wire delays are assumed to have *isochronic forks*. It is reasonable to assume that the forks located inside the input channels are isochronic as those are only used for the local communication. The two forks following signals *LOCKER.lock* and *GC.done* are more likely to have skewed timing on each of its branches (Figure 6.5). These forks are likely to become part of the global interconnects when the design scales and inverter trees are employed to maximize performance and additional analysis is needed to address these problems.

The *lock* signal can be safely forked into an inverter tree without creating any hazards (assuming the signals become isochronic once they enter the input channel tiles). When the *done* signal is forked, there may be a hazard forming on one of the *SR* signals. Indeed, if a buffer component is placed on the branch disabling *SR2* in the model on Figure 6.5, then, under the assumption of arbitrary buffer delay the arrival of *SR2.r+* may happen just after the *SR2.s+*, which will create a hazard on the *SR2* output.

In order for this hazard to occur, the path $GC.done+ \rightarrow SR2.q-$ has to be longer than the path $GC.done+ \rightarrow LOCKER.lock \rightarrow ME2.l- \rightarrow ME2.w+ \rightarrow SR2.q+$. Avoiding the hazard is ensured by placing the input of the *LOCKER*'s C-element on the longest branch of the *GC.done* fork.

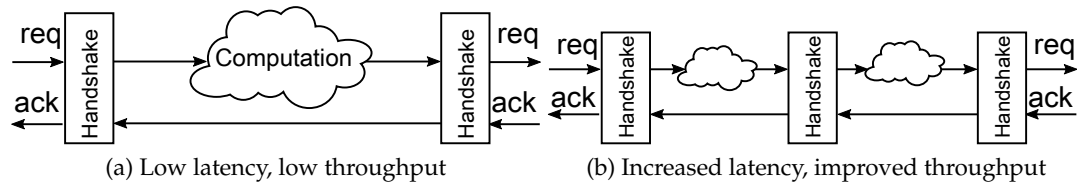


Figure 6.7: Pipelining

6.3 Possible Extensions

Data Lines

The request lines can be bundled with data lines to provide additional information for the arbitration logic. The data lines do not need to pass MUTEX elements and internally, their data can be latched by the requests that won the arbitration.

Accumulate and Fire

The *accumulate and fire* [16] tactics can be enforced by modifying the OR-gate tree in the *LOCKER* component in order to ignore the request combinations that are not useful or interesting. For instance, the arbiter may wait for at least a few requests arriving before it actually starts the arbitration and does not waste energy on lonely requests. Another example is the $M \times N$ arbiter where the arbiter needs to have at least one request from both sides to make a pair, otherwise arbitration would not make sense.

Pipelining

Pipelining is a technique used in long wire interconnects to increase the throughput of a system [6, 28]. It is also used to increase throughput by splitting slow computation into multiple fast computation stages (Figure 6.7). Similar approaches can be used to improve the throughput of the arbiter by splitting its synchronization and grant phases into two independent pipeline stages, the first executing synchronization, and the second producing grant signals.

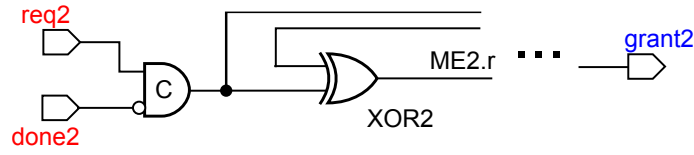


Figure 6.8: Employing RGD interface

Table 6.1: Priority 2-of-3 arbitration

$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g1'$	$g2'$	$g3'$
1	X	X	0	X	X	1	0	0
X	X	1	1	X	X	0	1	0
X	X	0	X	1	X	0	0	1
0	X	1	X	X	X	0	1	0
1	X	1	X	X	0	1	1	0
1	X	1	X	0	X	1	1	0
X	X	X	X	1	1	0	0	1
1	X	0	X	X	X	1	0	0

Supporting RGD

The arbiter can be adapted to support the RGD protocol. It is relatively simple to do because each of the request signal transitions is arbitrated already. Hence, implementing the 2-phase logic is possible by using one additional C-element per channel as shown in Figure 6.8.

6.4 Performance Estimations

6.4.1 Priority 2-of-3 Arbitration

One simple example considered is the priority-based 2-of-3 arbiter. It acts similarly to the 1-of-3 priority arbiter favouring requests $r1, r2, r3$ in that order. However, it is allowed to grant 2 resources at a time. The truth table for the arbiter decision making module is shown in Table 6.1. Here $g1', g2',$ and $g3'$ are the next state values for the current state of $r1, \dots, r3, g1, \dots, g3$. "X" represent the "don't care" values, which can be either "0" or "1".

The circuit as derived with the "Logic Friday" tool [3] is implementable with the following equations:

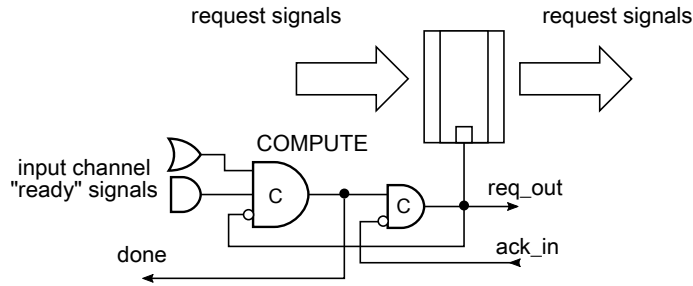


Figure 6.9: Decoupling synchronizer and grant controller

$$\begin{aligned}
 g1' &= r1 \cdot (r2 \cdot (\overline{g3} + \overline{r3}) + \overline{g2} + \overline{r2}) \\
 g2' &= r2 \cdot (r1 \cdot (\overline{g3} + \overline{r3}) + \overline{r1} + g2) \\
 g3' &= r3 \cdot (\overline{r2} + g3)
 \end{aligned}$$

In this example matched delay line was used for the completion detection. The estimated latency based on a CMOS 90nm implementation (including wire delays) of a request propagating from the input port until receiving a grant signal is between 900ps and 1000ps (depending on which requests were issued). The latency is still roughly the same regardless of whether two requests are granted concurrently or some request is granted while another released.

6.4.2 Pipelined Arbiter Scaling

The pipelined version of the arbiter has its grant controller separated through the additional handshake as shown in Figure 6.9. The requests possibly bundled with additional data are propagated further through the data latch controller. The falling edge of the COMPUTE signal is decoupled from the *req_out* signal. The latch is transparent when *req_out* = 0 and opaque when *req_out* = 1.

Figures 6.10a, 6.10b, and 6.10c demonstrate the pipelined arbiter performance based on its implementation in 90nm CMOS cell library including wire delays.

The latency is estimated as the time between the request signal propagates from the change in the request lines to the rising edge of the *req_out* (Figure 6.9). As it would be expected from the structure, the latency increase is logarithmic. The additional input channels occasionally add more layers to the trees of logic gates communicating the in-

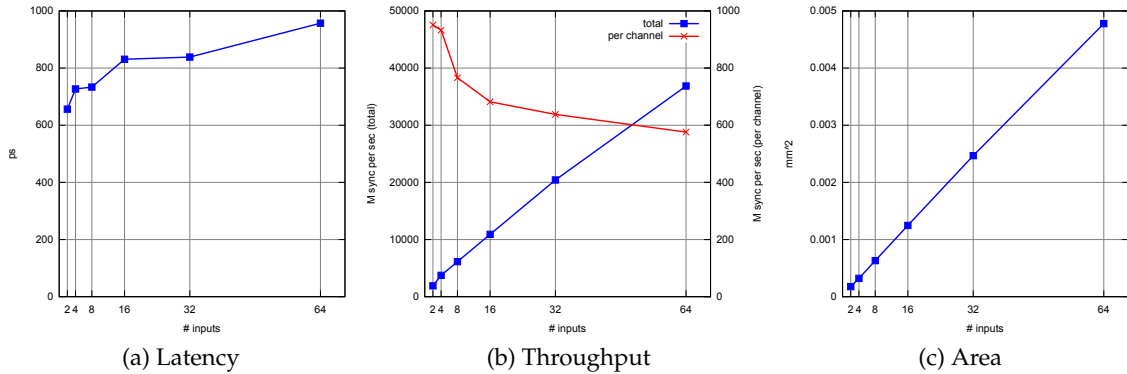


Figure 6.10: Pipelined arbiter performance

put channels. The latency for two inputs is $650ps$ on average. As the number of inputs increases up until 64, the latency increases to approximately $950ps$.

The throughput is measured as the number of input channels multiplied by the number of transactions (measured in millions per second) assuming that the grant controller is faster and the synchronizer is the bottleneck of performance. With the increased input count the total number of arbitrations drops to about 576 million arbitrations per second while the maximum number of requests processed increases to $576 \times 64 = 36864$ million requests per second (Figure 6.10b).

Finally, the estimation in Figure 6.10c shows how much the actual synchronizer area increases with the increasing number of clients. The result is essentially a linear dependence on the number of input channels.

6.5 Conclusions

This chapter presents a structure for an asynchronous generalized arbiter, which is able to tackle a large variety of arbitration problems.

The circuit may be slower than existing arbiters that are structurally dedicated to particular arbitration problems such as the 2×2 or the 1-of- N implementations. However, the generalized arbiter, scaled for multiple inputs, has benefits of resolving the conflicts in an intelligent and highly concurrent manner. For instance, the 4-input arbiter implementing the 2×2 MIMO cell can be made with simultaneous channel allocation for both

clients. At the same time, the channel allocation could be made static, e.g., the configuration of the channel multiplexers would not be configured for each arbitration transaction, and it would only be changed if particular combination of requests requires it.

The grant controller is activated by the locally generated clock signal allowing developing the advanced arbitration logic in synchronous design flow. At the same time, the arbiter operates only when there is a change among its request lines, meaning that no dynamic power is wasted when no arbitration is requested. The arbiter allows pipelining that can improve its throughput. At the same time, the grant computation is pushed into separate logics making the module easier to develop and verify for fabrication faults.

The generalized arbiter was formally verified to be free from deadlocks and hazards while the risk of stalling the arbiter was shown to be dependent on the correctness of the grant controller implementation. The scalable decomposition of the arbiter was presented. All of the necessary timing assumptions used are practical and easy to enforce even without explicit delay elements.

More research is needed to estimate arbiter performance and compare it with existing implementations. The chapter has provided a high-level specification, which may have different decompositions. Other decompositions with more timing assumptions also need more study as such designs may provide greater performance benefits in future.

It is believed that such an arbiter can be useful in future asynchronous NoC routers where sophisticated arbitration logic is needed for designing various adaptive routing protocols in busy request environments.

Chapter 7

Conclusions

This work presents multi-resource arbitration components that distribute interchangeable resources among clients based on the information on resource availability. The high performance of a system is ensured by the concurrency in resource utilization. The robustness is ensured by the fact that each client can use one of multiple interchangeable resources and the client can be serviced while at least one of the resources is available.

The simple multi-resource arbiter is not suitable for the allocation task when the resources are completely or partially interchangeable and more sophisticated arbitration logic is needed for this case. The traditional STG based design flow appears to have difficulties in tackling this problem and a higher level design flow was proposed and implemented as a result. The gate-level design flow is a good alternative, which helped tackling the problem structurally.

With stronger tool support, the more complex arbiter designs can be approached. The generalized arbiter conceptually solves a great variety of arbitration problems. Its effectiveness is achieved through dividing the arbitration process into two separate stages of computation. First of all, the arbiter needs to acknowledge changes in the input signals, which would initiate the second stage – the computation of grant signals based on the current input values. Such a separation has allowed to create an arbiter with low latency. As opposed to most other arbiter implementations, each of the input signals propagates through only a single layer of MUTEX arbitration. The column of MUTEX

elements creates a snapshot of all input signals, which can be used by the grant logic computation. In practice, this arbitration platform can be used to create arbiters with semi-interchangeable resources as a comprehensive solution to the asynchronous NoC routers, effectively implementing the multi-resource utilization.

7.1 Summary of Contribution

- This work introduces the multi-resource arbiter and shows a number of examples, where such a design could be useful;
- To develop more complex arbiters, the gate-level design flow was implemented a plugin of the existing Workcraft modelling environment;
- Finally, the generalized arbiter is developed based on the gate-level design flow.

7.2 Future Work

This work opens up opportunities to address multiple additional issues in future research. The gate flow plugin still needs to be improved to support hierarchical composition of components, so that a finished design it can be encapsulated into a box with multiple inputs and outputs and then reused in other models. Alternatively, the support for abstract components with pre-defined interface and no implementation can be useful for testing out alternative implementations of a chosen component.

Various models supporting the concept of digital signals can be intermixed with the gate-level design, the examples are STGs and Conditional Partial Order Graphs (CPOGs) [51]. When combined, these models can function simultaneously while communicating through the gate-level model connections.

The generalized arbiter has a great potential for further research. Its performance can be compared to other arbiters and its usefulness still needs to be demonstrated through building new highly efficient asynchronous NoCs.

Appendix A

Summary on Asynchronous Arbiters

Further tables provide an overview of existing asynchronous arbiter implementations.

Table A.1: Analogue arbiters

PROTOCOL	DESCRIPTION/FEATURES	REFERENCES
RG	threshold-based MUTEX	[60]
RG	4-phase MUTEX element	[68, 44]
RG	fast multi-flop arbiter	[10, 21, 43]
RG	2-of-3 arbiter	[10]
RGD	5-wire arbiter with “enabling”, the “propellor arbiter”	[53]
RG	Lockable C-element	[26]

Table A.2: Two-way arbiters

HANDSHAKE WIRES	DESCRIPTION/FEATURES	REFERENCES
RGD	2-phase arbiter	[77, 27]
RGD	2-phase with enabling	[76]
RNG	“eager” arbitration	[86]
RGN	non-blocking arbitration, “nacking” arbiter	[57, 20]
RGNDA	2-phase nacking arbiter	[23]

Table A.3: 1-of- N arbiters

TOPOLOGY	HANDSHAKE WIRES	DESCRIPTION/FEATURES	REFERENCES
<i>Mesh-based arbiters</i>			
mesh	RG	linear latency, quadratic scalability	[35]
<i>Cascaded tree</i>			
casc. tree	RG	linear scalability	[63, 61, 86]
casc. tree	RG	fast request propagation	[31]
casc. tree	RG	fast request release	[25, 86]
<i>Busy ring</i>			
busy ring	RG/P	4-phase/propagate	[18, 81, 35]
busy ring	RGD/P	2-phase/propagate	[86]
busy ring	RG/RG	4-phase/handshake	[48]
busy ring	RGDNA(2-phase)/P	2-phase/propagate	[23]
<i>Lazy ring</i>			
lazy ring	RG/RG	4-phase/handshake	[48, 35]
lazy ring	RG/(2-phase)P	uses “pausable” 2-phase token propagation	[26]

Table A.4: Other arbiters

NAME	HANDSHAKE WIRES	DESCRIPTION/FEATURES	REFERENCES
<i>Flat arbiters</i>			
“flat” arbiter	RG	“flat” arbitration producing ordered request state	[52, 40]
<i>Priority-enforcing arbiters</i>			
ordered arbiter	RG	The order of grants repeats the order of requests	[12]
daisy-chain	RG	Priority enforced by topology	[11, 35]
static priority arbiter	RG	Static priority defined in combinational logic	[63, 37, 11, 35]
dynamic priority arb.	RG	Dynamically reconfigurable priority	[11, 35]
<i>Multi-resource arbiters</i>			
Multi-token arbiter		Arbiter grants a resource M times	[85]
Patil’s arbiter	RG	multi-resource arbitration with passive resources	[58, 59]
Committee arbiter	RG	Wide range of arbitration problems	[9]
“soft” arbiter	RG	“soft” arbitration, total number of resources granted converges to M	[50]

Appendix B

Workcraft Interface

B.1 Main Window

Workcraft is a plugin-based computer aided design (CAD) tool allowing to create and interactively simulate Petri nets, Signal Transition Graphs, Digital Circuits and some other model types that fall into the category of the *interpreted graph models* [64]. The main window of the program is shown in Figure B.1. It is split into multiple smaller windows with dedicated responsibilities:

Main menu is used to manage models, configure system, and call various external tools to do additional model processing;

Editor tabs shows all of the opened models, which allows to open several models in the same session;

Editor window is the main window where models are created, viewed, and simulated;

Tool controls is used to manage simulation traces when in simulation mode;

Property editor allows changing properties of objects selected in the editor window when in editing mode;

Editor tools panel allows to select the mode of operation (Figure B.1: Selection tool, Connection tool,). Its contents vary from one model to another providing different sets of operation modes.

Workspace presents the list of opened or imported files;

Utility windows shows additional information such as external tool output, error messages or the progress of launched tasks.

B.1.1 Basic Mouse Controls

Mouse wheel zooms in and out;

Left click selects, connects, creates new objects (depending on the active editor mode);


Right click shows context-sensitive drop-down menu;


Middle button pans view.

B.2 Common Operation Modes

The common operation modes are used in both the STG and the Circuit models.


Select  – selecting objects, moving them around and changing their properties;


Connect  – creating new directed connections between model components;

Simulate  – simulating models interactively.

B.3 STG Plugin Operation Modes

To create a new circuit model, from the menu select: File→Create Work...→Signal Transition Graph.

Dummy transition  – creating a dummy transition;

Signal transition  – creating signal transitions (a transition that is associated with a raising or a falling edge of some signal). Use the selection mode to change its name, transition direction, or signal type (input, output or internal);

Place  – creating new places. Use the selection mode to change its token count.

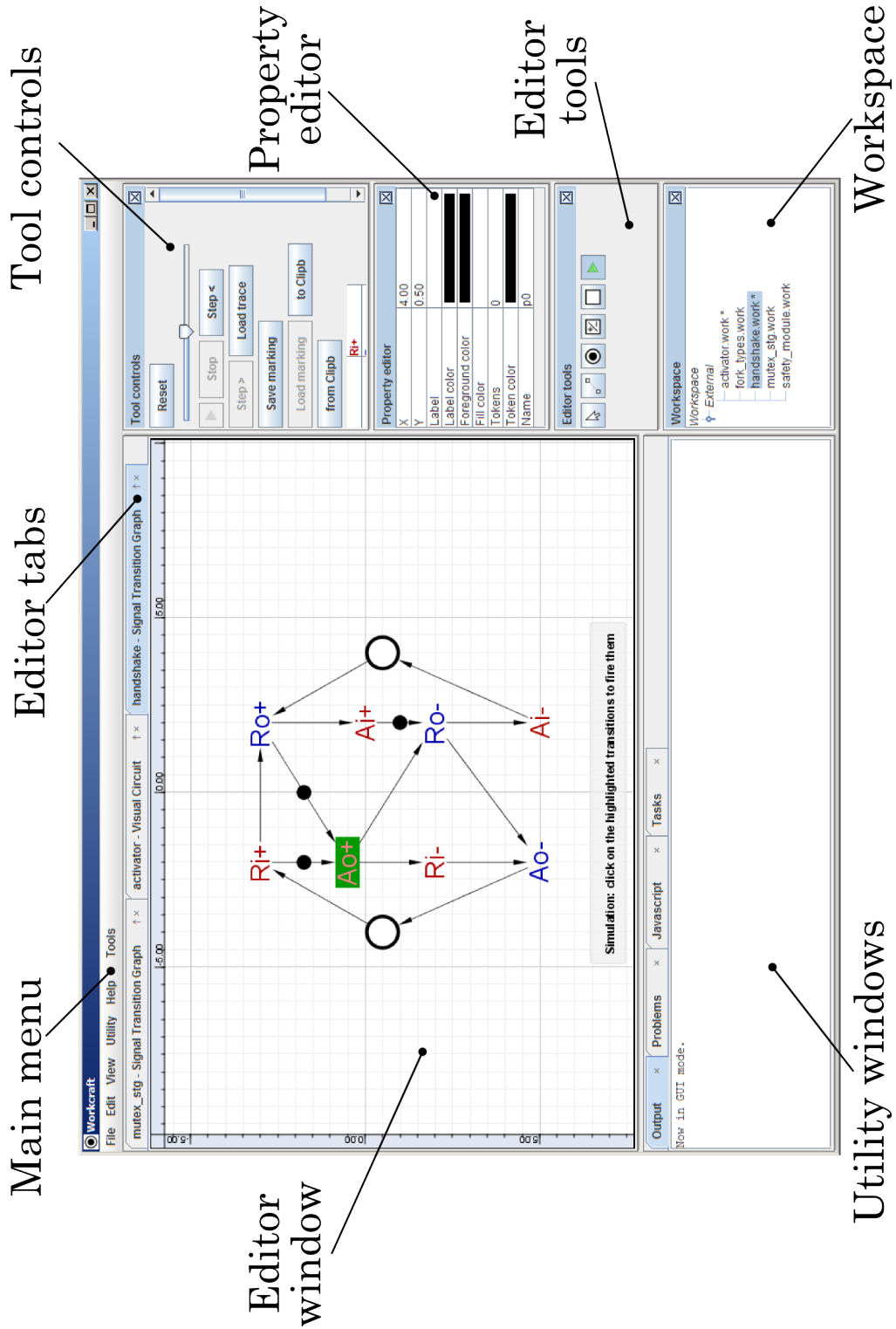





Figure B.1: Workcraft Interface

B.4 Digital Circuit Plugin Operation Modes

To create a new circuit model, from the menu select: File→Create Work→Digital Circuit.

Input/output port  – creating new input or output ports. *Left-click* creates an output port. *Left-click + shift* creates an input port;


Joint  – creating joints that allow branching wires from one source to multiple destinations. The joints are not essential as branching may happen from any connection point such as ports or component contacts. However, joints are still useful for giving the model a better look;

Function  – creating function components. Right-clicking on the function component will bring the pop-up menu where the additional output ports may be added. For instance, it can be used to keep both MUTEX signals inside the same function component.

B.5 Conversion to the Circuit STG

Any circuit model can be converted to its STG equivalent. It is easily done through the menu: Tools→STG→Generate STG. Once the STG is generated, its model occurs in the “Workspace” window (Figure B.2). It can be opened by right-clicking on the new model in the list and selecting “Open editor”.

B.6 Simulation

Both circuit model and STG model allow interactive simulation. The simulation is started by clicking on the  button.

In the STG model, during simulation the enabled transitions are highlighted and can be clicked to see the result of their firing. The “Tool Controls” window will show the trace being generated (Figure B.1). This trace can be saved into clipboard or restored from it by clicking buttons “to Clipb” and “from Clipb”. When clicked on a particular

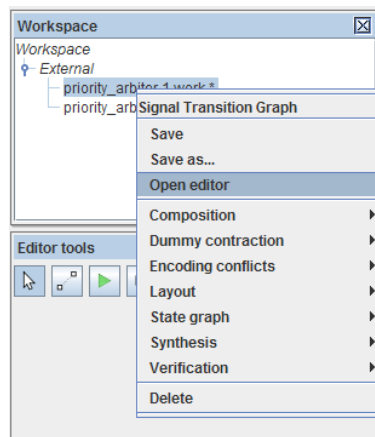


Figure B.2: Opening generated STG

trace transition, the model returns to the state it was at that moment. To restore the state, the simulation tool actually “unfires” the fired transitions (for the transition being unfired it removes tokens from the post-set places and adds tokens to the pre-set places) in the opposite order, which can always restore the initial state.

In the circuit model, the simulation is actually based on the STG generated from the circuit. Once started, the main window shows the circuit with wires painted “active” and “inactive” and the excited signal contacts are marked and can be clicked to change circuit state. The simulation changes the generated STG and then paints the circuit contacts according to the excited STG transitions and paints signal wires according to the corresponding STG signal values. As a result, the simulation tool is inherited from the STG simulation, and can also use the same traces, be “fired” and “unfired”.

B.7 Verification

The circuit verification for deadlocks and hazards has a shortcut. It can be executed through the menu: Tools→Verification→Check circuit for deadlocks and hazards (the long way of doing it would be converting the circuit to its STG form, and then calling explicitly the particular type of verification as it is described in [64]). Once the verification is started, the STG is generated, which is then put through the Puf to find its unfolding. After that, the unfolded model is processed by Mpsat to find whether there are reachable

deadlocks or hazard states. If such a state is found, the plugin returns the corresponding trace, that can be replayed in simulation.

References

- [1] Dining philosophers problem
http://en.wikipedia.org/wiki/Dining_philosophers_problem.
- [2] International technology roadmap for semiconductors: 2005 edition.
- [3] Logic friday: <http://sontrak.com/>.
- [4] Petrify: <http://www.lsi.upc.es/~jordicf/petrify/petrify.html>.
- [5] Punf: <http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/>.
- [6] John Bainbridge and Steve Furber. CHAIN: A delay-insensitive chip area interconnect. *IEEE Micro*, 22:16–23, 2002.
- [7] A. Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, Department of Computer Science, University of Manchester, 2000.
- [8] Salomon Beer, Ran Ginosar, Michael Priel, Rostislav (Reuven) Dobkin, and Avinoam Kolodny. The devolution of synchronizers. In *Proceedings of the 2010 IEEE Symposium on Asynchronous Circuits and Systems, ASYNC '10*, pages 94–103, Washington, DC, USA, 2010. IEEE Computer Society.
- [9] I. Benko and J.C. Ebergen. Delay-insensitive solutions to the committee problem. *Advanced Research in Asynchronous Circuits and Systems, 1994., Proceedings of the International Symposium on*, pages 228–237, November 1994.
- [10] Kees van Berkel and C.E. Molnar. Beware the three-way arbiter. *Solid-State Circuits, IEEE Journal of*, 34(6):840–848, June 1999.

-
- [11] Alex Bystrov, David J. Kinniment, and Alex Yakovlev. Priority arbiters. In *ASYNCR '00: Proc. of the 6th Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 128–137, Washington, DC, USA, 2000. IEEE Computer Society.
- [12] Alex Bystrov and Alex Yakovlev. Ordered arbiters. *Electronics Letters*, 35(11):877–879, May 1999.
- [13] K. Mani Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
- [14] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.
- [15] D.M. Chapiro. *Globally asynchronous locally synchronous systems*. PhD thesis, Stanford University, October 1984.
- [16] Yuan Chen. *High Level Modelling and Design of a Low Power Event Processor*. PhD thesis, Newcastle University, January 2009.
- [17] T.A. Chu. *Sintesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.
- [18] P. Corsini. n-user asynchronous arbiter. *Electronics Letters*, 11(1):1–2, 1975.
- [19] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and Alex Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Springer-Verlag, ISBN: 3-540-43152-7, 2002.
- [20] J. Cortadella, L. Lavagno, P. Vanbekbergen, and Alex Yakovlev. Designing asynchronous circuits from behavioural specifications with internal conflicts. In *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, pages 106–115, Salt Lake City, UT, Nov 1994.
- [21] A.C. Davies. Dynamic properties of a multiway arbiter. *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, 3:221–224, 2000.

-
- [22] Rostislav R. Dobkin, Ran Ginosar, and Avinoam Kolodny. Qnoc asynchronous router. *Integr. VLSI J.*, 42:103–115, February 2009.
- [23] Jo C. Ebergen, P. F. Bertrand, and S. Gingras. Solving a mutual exclusion problem with the rgd arbiter. In *Proceedings of the IFIP WG10.5 Working Conference on Asynchronous Design Methodologies*, pages 137–147, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.
- [24] T. Felicijan, J. Bainbridge, and S. Furber. An asynchronous low latency arbiter for quality of service (QoS) applications. *Microelectronics, 2003. ICM 2003. Proceedings of the 15th International Conference on*, pages 123–126, December 2003.
- [25] Hartmann J. Genrich and Robert M. Shapiro. Formal verification of an arbiter cascade. In *Proceedings of the 13th International Conference on Application and Theory of Petri Nets*, pages 205–223, London, UK, 1992. Springer-Verlag.
- [26] Ganesh Gopalakrishnan. Developing micropipeline wavefront arbiters. *IEEE Design & Test of Computers*, 11(4):55–64, Winter 1994.
- [27] M.R. Greenstreet and T. Ono-Tesfaye. A fast, asP*, RGD arbiter. In *Advanced Research in Asynchronous Circuits and Systems, 1999. Proceedings., Fifth International Symposium on*, pages 173–185, 1999.
- [28] Ron Ho, John Gainsley, and Robert Drost. Long wires and asynchronous control. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 240–249. IEEE Computer Society Press, 2004.
- [29] Ron Ho and Mark Horowitz. Lecture 9: More about wires and wire models. *Computer Systems Laboratory*, 2007.
- [30] Anoop Iyer and Diana Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 158–168, Washington, DC, USA, 2002. IEEE Computer Society.

-
- [31] M.B. Josephs and J.T. Yantchev. CMOS design of the tree arbiter element. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 4(4):472–476, December 1996.
- [32] R.M. Keller. Towards a theory of universal speed-independent modules. *Computers, IEEE Transactions on*, C-23(1):21–33, January 1974.
- [33] Victor Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, University of Newcastle upon Tyne, February 2003.
- [34] Victor Khomenko. A usable reachability analyser. Technical report, Newcastle University, 2009.
- [35] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. John Wiley & Sons, Ltd, 2007.
- [36] David J. Kinniment and Doug Edwards. Circuit technology in a large computer system. In *Proceedings of the Conference on Computers–Systems and Technology*, pages 441–450, October 1972.
- [37] David J. Kinniment and Viv Woods. Synchronisation and arbitration in digital systems. *Proc. IEEE*, 123(10):961–966, October 1976.
- [38] Tong Lin, Kwen-Siong Chong, Bah-Hwee Gwee, and Joseph S. Chang. Fine-grained power gating for leakage and short-circuit power reduction by using asynchronous-logic. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 3162–3165, May 2009.
- [39] Ying Liu, S.R. Nassif, L.T. Pileggi, and A.J. Strojwas. Impact of interconnect variations on the clock skew of a gigahertz microprocessor. In *Design Automation Conference, 2000. Proceedings 2000. 37th*, pages 168–171, 2000.
- [40] Yu Liu, Xuguang Guan, Yang Yang, and Yintang Yang. An asynchronous low latency ordered arbiter for network on chips. In *Natural Computation (ICNC), 2010 Sixth International Conference on*, volume 2, pages 962–966, August 2010.

-
- [41] Kia Seng Low and Alex Yakovlev. Token ring arbiters: an exercise in asynchronous logic design with Petri nets, 1995.
- [42] Daniele Ludovici, Alessandro Strano, Davide Bertozzi, Luca Benini, and Georgi N. Gaydadjiev. Comparing tightly and loosely coupled mesochronous synchronizers in a NoC switch architecture. In *3rd ACM/IEEE International Symposium on Networks on Chip*, pages 244 – 249, May 2009.
- [43] O. Maevsky, D.J. Kinniment, Alex Yakovlev, and Alex Bystrov. Analysis of the oscillation problem in tri-flops. *EECE*, 1:381 – 384, May 2002.
- [44] Alain J. Martin. On Seitz’s arbiter. Technical Report 5212:TR:86, Caltech Computer Science, 1986.
- [45] Alain J. Martin. Collected papers on VLSI design. In *Caltech-CS-TR-90-09*, Dept. of Computer Science, Caltech, 1990.
- [46] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.
- [47] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.
- [48] Alain J. Martin. Synthesis of asynchronous vlsi circuits. Technical Report CaltechCSTR:1991.cs-tr-93-28, California Institute of Technology, 1991.
- [49] T.G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –11, nov. 2008.
- [50] A. Mokhov and Alex Yakovlev. Soft arbiters. Technical Report NCL-EECE-MSD-TR-2009-149, Newcastle University, 2009.

-
- [51] Andrey Mokhov. *Conditional Partial Order Graphs*. PhD thesis, Newcastle University, September 2009.
- [52] Andrey Mokhov, Victor Khomenko, and Alexandre Yakovlev. Flat arbiters. In *ACSD'09*, pages 99–108, 2009.
- [53] C.E. Molnar and I.W. Jones. Simple circuits that work for complicated reasons. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 138–149, 2000.
- [54] T.N. Mudge, J.P. Hayes, and D.C. Winsor. Multiple bus architectures. *Computer*, 20(6):42–48, 1987.
- [55] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proc. Int'l Symp. Theory of Switching, Part 1, Harvard Univ. Press*, pages 204–243, 1959.
- [56] Tadao Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, April 1989.
- [57] S.M. Nowick and D.L. Dill. Practicality of state-machine verification of speed-independent circuits. *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on*, pages 266–269, November 1989.
- [58] Suhas S. Patil. n-server m-user arbiter. Technical report, Computation Structures Group Memo 42, M.I.T., 1969.
- [59] Suhas S. Patil. Forward acting n x m arbiter. Technical report, Computation Structures Group Memo 67, M.I.T., 1972.
- [60] Suhas S. Patil. Synchronizers and arbiters. Technical Report Memo 91, MIT Press, October 1973.
- [61] R. C. Pearce, J. A. Field, and W. D. Little. Asynchronous arbiter module. *IEEE Trans. Comput.*, 24(9):931–932, 1975.
- [62] Carl Adam Petri. *Kommunikation mit Automaten (Communicating with automata)*. PhD thesis, 1962.

-
- [63] W.W. Plummer. Asynchronous arbiters. *Computers, IEEE Transactions on*, C-21(1):37–42, January 1972.
- [64] Ivan Poliakov. *Interpreted Graph Models*. PhD thesis, Newcastle University, May 2011.
- [65] Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. Workcraft — a framework for interpreted graph models. In *PETRI NETS'09: Proc. of the 30th Int. Conf. on Applications and Theory of Petri Nets*, pages 333–342, Berlin, Heidelberg, 2009. Springer-Verlag.
- [66] Jan M. Rabaey and Alberto Sangiovanni-vincentelli. System-on-a-chip - a platform perspective.
- [67] Leonid Rosenblum and Alex Yakovlev. Signal graphs: from self-timed to timed ones. *Int. Workshop on Timed Petri Nets*, pages 199–206, July 1985.
- [68] C. L. Seitz. Ideas about arbiters. *Lambda*, 1:10–14, 1980.
- [69] Charles L. Seitz. System timing. In Mead Conway, editor, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, Reading MA, 1980.
- [70] Maitham Shams, Jo C. Ebergen, and Mohamed I. Elmasry. Modeling and comparing CMOS implementations of the C-element. *IEEE Transactions on VLSI Systems*, 6(4):563–567, December 1998.
- [71] D. Shang, A. Yakovlev, A. Koelmans, D. Sokolov, and A. Bystrov. Dual-rail with alternating-spacer security latch design. Technical Report NCL-EECE-MSD-TR-2005-107, Newcastle University, 2005.
- [72] Delong Shang, Fei Xia, Stanislavs Golubcovs, and Alexandre Yakovlev. The magic rule of tiles: Virtual delay insensitivity. In *PATMOS*, pages 286–296, 2009.
- [73] 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, May 2009.

-
- [74] Wei Song and Doug Edwards. Improving the throughput of asynchronous on-chip networks with sdm. In *Proc. of the UK Electronics Forum*, June 2010.
- [75] Jens Sparsø and Steve Furber. *Principles of Asynchronous Circuit Design*. Kluwer Academic Publishers, ISBN: 978-0-7923-7613-2, Boston/Dordrecht/London, 2002.
- [76] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. The counterflow pipeline processor architecture. *Design Test of Computers, IEEE*, 11(3):48, 1994.
- [77] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.
- [78] Y. Tamir and H.-C. Chi. Symmetric crossbar arbiters for vlsi communication switches. *Parallel and Distributed Systems, IEEE Transactions on*, 4(1):13–27, 1993.
- [79] Alexander Taubin, Jordi Cortadella, Luciano Lavagno, Alex Kondratyev, and Ad M. G. Peeters. Design automation of real-life asynchronous devices and systems. *Foundations and Trends in Electronic Design Automation*, 2(1):1–133, 2007.
- [80] C.H. van Berkel and T. van Roermund. Scalable multi-input-multi-output queues with application to variation-tolerant architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(7):920–923, July 2009.
- [81] Victor I. Varshavsky, M. A. Kishinevsky, V. Marakhovsky, and Alex Yakovlev. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.
- [82] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, March 1988.
- [83] Chandu Visweswariah. Death, taxes and failing chips. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 343–347, New York, NY, USA, 2003. ACM.

- [84] A.J. Winstanley, A. Garivier, and M.R. Greenstreet. An event spacing experiment. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 47 – 56, April 2002.
- [85] Alex Yakovlev. Designing arbiters using Petri nets. *VLSI Systems Research Center, Israel Institute of Technology, Haifa, Israel*, pages 179–201, 1995.
- [86] Alex 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, Sep 1994.

Index

- 2-phase signalling, 28
- 4-phase, 45
- 4-phase handshake, 35, 57, 82, 89, 106
- 4-phase signalling, 28
- accumulate and fire, 113
- active, 10
- active resource, 54
- analogue arbiter, 30
- analogue circuit, 11
- analogue difference circuit, 32
- arbiter, 3
- arbiter channels, 29
- arbitration cell, 31, 38
- asymmetric fork, 12, 91
- asynchronous circuit, 8, 87
 - delay-insensitive, 11
 - self-timed, 12
 - speed-independent, 11
- bundled data, 22, 107
- C-element, 15, 16
 - asymmetric, 16
- CAD, 123
- channel
 - active side, 20
 - passive side, 20
- channel type, 19
 - biport channel, 19
 - nonport channel, 19
 - pull channel, 19
 - push channel, 19
- circuit
 - input signal, 11
 - internal signal, 11
 - output signal, 11
- circuit STG, 95
- committee arbiter, 50, 54, 102
- complete state coding, 16, 60
 - conflict, 16
- completion detection, 20, 22, 109
- critical path, 9
- decision wait, 17
- delay-insensitive, 9, 75, 91
- delay-sensitive encoding, 22
- digital arbiter, 30
- digital circuit, 10, 87
- dual-rail, 45
- dummy transition, 15

- dynamic priority arbiter, 46
- equipotential region, 90
- excited signal, 10
- fair arbitration, 30
- FPGA, 33
- GALS, 2
- generalized arbiter
 - grant controller, 106
 - input channel, 106
 - locker, 106
 - pipelining, 113
- grant controller, 63
- grant phase, 104
- handshake, 18
 - 2-phase, 19
 - 4-phase, 18
- handshake activator, 81
- handshake passivator, 81
- hazard, 11
- inactive, 10
- interpreted graph model, 123
- IP core, 1
- isochronic fork, 12
- mesh arbiter, 39
- mesochronous synchronizers, 3
- metastability, 28
- Network-on-Chip, 2
- next state function, 16
- non-deterministic choice, 28
- non-persistence, 11
- nonput channel, 29
- order of arbitration, 29
- ordered arbiter, 43
- passive resource, 54
- path redundancy, 102
- pending requests, 30
- Petri net, 12, 28
 - 1-safe, 14
 - boundedness, 14
 - deadlock, 14
 - post-set, 13
 - pre-set, 13
 - trace, 14
 - transition enabling, 13
 - transition firing, 13
- priority arbiter, 5
- priority module, 45
- priority-based arbitration, 30
- privilege token, 41
- push channel, 82
- RAID, 1
- RAIM, 1
- reach language, 95
- read arc, 94
- relative timing assumption, 91

request controller, 63
return-to-zero, 28
RGD arbiter, 36

semi-automated decomposition, 56
semi-interchangeable resources, 102
signal inputs, 10
signal transition, 10
spacer, 20
stable signal, 10
static priority arbiter, 45
statistical fairness, 30
STG, 15, 28
 simplified notation, 15
synchronization phase, 104

ternary metastability, 33
timing assumption, 26
timing assumptions, 12
toggle component, 17
toggle element, 88
tree arbiter cell, 40
true tiles, 108

unbounded delay, 28

Workcraft, 5, 86