Design of an asynchronous communication network for an audio DSP chip

Master of Science Project

Informatics and Mathematical Modelling (IMM)
Computer Science and Engineering, division

Technical University of Denmark(DTU)

15 August 2005

Supervisor: *Prof. Jens Sparsø*
External supervisor: *Johnny Halkjær Pedersen*
[*William Demant Holding*]
Co-supervisor. Ph.D student: *Tobias Bjerregaard*

Mikkel Bystrup Stensgaard (*s001434*)

Abstract

This project investigates the replacement of the communication network in a multi-configurable DSP-core developed by *William Demant Holding*. The existing network is implemented as a subset of a fully connected network which contains many long wires that consume power and complicates routing.

The existing network is replaced by 3 different *packet-switched*, *source-routed* asynchronous networks, which solve many of the problems in the current network implementation. The size of the networks are linear with the number of communicating blocks which makes it very scalable, the networks are 'plug-and-play' and can be ported to other applications, there are no restrictions on which blocks that can communicate as in the current solution, and the networks decouple the connected blocks which allows them to run in their own clock domain.

As the needed bandwidth is very low the networks are designed with area and power in mind, and simple solutions are chosen for all design issues. The networks are implemented as a binary tree of *merger* and *router* blocks, and both bundled data and a *1-of-5* delay-insensitive data encoding are implemented and compared.

This report documents the design, implementation, synthesis, and verification of the networks. It also discusses the design choices in a number of different areas such as data-encoding, network topology and how to implement multicasting. As the networks are designed as asynchronous circuits, part of the report documents the implementation of these and how to handle asynchronous circuits in a synchronous design flow.

Acknowledgements

This master of science project has been carried out at the Technical University of Denmark in close cooperation with *William Demant Holding*. I would like to thank *William Demant Holding* for the hospitality that you have showed me, for giving me inside information about the Aphrodite chip, for letting me use your design flow and for letting me work at your facilities. I am grateful to Johnny Halkjær Pedersen from *William Demant Holding* for the time you spent telling me about the Aphrodite chip, for integrating the developed NoCs into the existing system, and helping me out on the design flow. There has been an increasing amount of work, that turned up during the project. It has not been an easy task for you to find time to help me in your busy schedule. Thanks a lot Johhny! Without your help the project would have been impossible.

In particular I would like to thank to my supervisor Jens Sparsø and Co-supervisor, Ph.D student Tobias Bjerregaard. Your burning interest in Asynchronous circuits is a big inspiration, and I would like thank you for always begin ready to share your experiences into the asynchronous world and for the interest you both showed in this project.

Contents

1 Introduction **1**
1.1 Network-on-chip . . . 1
1.2 Previous work . . . 3
1.3 Project description . . . 3
1.4 Report structure . . . 4

2 Background: Network-on-chips **6**
2.1 Overview . . . 6
2.2 Network type . . . 6
2.3 Packets and flits . . . 7
2.4 Switching techniques . . . 7
2.5 Routing . . . 8
2.6 Guaranteeing bandwidth . . . 9
2.7 Topology . . . 9

3 Background: Asynchronous circuits **10**
3.1 Overview . . . 10
3.2 The C-element . . . 11
3.3 Handshake protocols . . . 12
3.3.1 Bundled data . . . 12
3.3.2 Delay-insensitive encoding . . . 13
3.3.3 Comparison . . . 14

4 Design methodology **15**
4.1 Overview . . . 15
4.2 Standard cells and drive-strengths . . . 15
4.3 Basic asynchronous components . . . 17
4.3.1 Mutex . . . 17
4.3.2 C-elements . . . 18
4.4 Complex asynchronous controllers . . . 19
4.5 Bundled data design and asymmetric delay . . . 22
4.6 Initializing asynchronous circuits . . . 23

5 The Aphrodite DSP **24**
5.1 Overview . . . 24
5.2 Configurable network . . . 24
5.3 Multicast . . . 25
5.4 Clocks, dataflow and Lego2 protocol. . . . 25
5.5 Sample addition . . . 26

6 Specification of the Network interface **28**
6.1 Overview . . . 28
6.2 Adders . . . 28
6.3 Network ports . . . 29
6.4 Configuration . . . 30
6.5 Final NoC interface . . . 32
6.6 Integration into Aphrodite . . . 32

7 Network design **33**
7.1 Overview . . . 33
7.2 Topology . . . 34
7.3 Data encoding . . . 37
7.4 Multicast . . . 38
7.5 Summary . . . 41

8 Implementation **42**
8.1 Overview . . . 42
8.2 Common network platform . . . 43
8.2.1 NA, Network Adapter . . . 44
8.2.2 AN, Network adapter . . . 45
8.2.3 Serializer . . . 47
8.2.4 De-serializer . . . 49
8.3 Specific network blocks . . . 51
8.3.1 Bundled data network blocks . . . 51
8.3.2 *1-of-5* network blocks . . . 51
8.4 The networks . . . 52
8.4.1 NoC1: Bundled data, multicast in NA . . . 52
8.4.2 NoC2: Bundled data, shared multicast blocks . . . 52
8.4.3 NoC3: *1of5* encoding, multicast in NA . . . 53

9 Verification **57**
9.1 Overview . . . 57
9.2 Main testbench . . . 58
9.2.1 Verification modules . . . 60
9.2.2 Tests . . . 60

10 Logic synthesis and simulation **62**
10.1 RTL simulation . . . 62
10.2 Logic synthesis . . . 62
10.3 Gate-level simulation . . . 63
10.4 Place and Route . . . 63
10.5 Area and power estimates . . . 63

11 Results and discussion **64**
11.1 Overview . . . 64
11.2 Results . . . 64
11.2.1 Bundled data networks . . . 65
11.2.2 *1-of-5* network . . . 65
11.3 Discussion . . . 67

12 Conclusion **70**

Bibliography **73**

A Synchronization **74**

B Cell library **76**

C CD contents **78**

D Network building blocks **79**
D.1 Common blocks . . . 79
D.1.1 AM_multicast . . . 79
D.1.2 AM_unicast . . . 80
D.1.3 AN, network adapter . . . 81
D.1.4 de_serializer . . . 82
D.1.5 Multicaster . . . 83
D.1.6 NA, network adapter . . . 84
D.1.7 serializer . . . 85
D.1.8 Sequencer . . . 86
D.1.9 Sequencer_en . . . 87
D.1.10 Sequencer2 . . . 88
D.2 Bundled data blocks . . . 89
D.2.1 P_merge . . . 89
D.2.2 P_merge_tree . . . 90
D.2.3 P_multicast . . . 91
D.2.4 P_network . . . 92
D.2.5 P_router . . . 93
D.2.6 P_router_tree . . . 94
D.2.7 P_sink . . . 95
D.3 *1-of-5* blocks . . . 96

D.3.1 PC_bundled_1of4 . . . 96
D.3.2 PC_1of4_bundled . . . 97
D.3.3 S_latch . . . 98
D.3.4 S_merge . . . 99
D.3.5 S_merge_tree . . . 100
D.3.6 S_network . . . 101
D.3.7 S_router . . . 102
D.3.8 S_router_tree . . . 103
D.3.9 S_sink . . . 104
D.3.10 S_source . . . 105

E Verilog Code **106**
E.1 Cell library . . . 106
E.1.1 cell_library.v . . . 106
E.1.2 cell_library_at58000.v . . . 113
E.2 Networks . . . 121
E.2.1 Converter . . . 121
E.2.2 Converter_P2 . . . 124
E.2.3 NoC . . . 128
E.2.4 NoC_P2 . . . 132
E.2.5 NoC_S1 . . . 138
E.3 Common blocks . . . 144
E.3.1 global.v . . . 144
E.3.2 AM_multicast . . . 146
E.3.3 AM_unicast . . . 148
E.3.4 AN . . . 149
E.3.5 de_serializer . . . 151
E.3.6 Multicaster . . . 154
E.3.7 NA . . . 157
E.3.8 serializer . . . 160
E.3.9 Sequencer . . . 163
E.3.10 Sequencer_en . . . 164
E.3.11 Sequencer2 . . . 165
E.4 Verification . . . 167
E.4.1 bfm_lego2master . . . 167
E.4.2 bfm_lego2slave . . . 171
E.4.3 Configuration . . . 175
E.4.4 mutex . . . 178
E.4.5 noc_top_testbench . . . 179
E.5 Bundled data blocks . . . 192
E.5.1 P_merge . . . 192
E.5.2 P_merge_tree . . . 194
E.5.3 P_multicast . . . 197
E.5.4 P_network . . . 199

E.5.5 P_router 201
E.5.6 P_router_tree 204
E.5.7 P_sink 207
E.6 *1-of-5* blocks 208
E.6.1 PC_bundled_1of4 208
E.6.2 PC_1of4_bundled 210
E.6.3 S_latch 212
E.6.4 S_merge 214
E.6.5 S_merge_tree 217
E.6.6 S_network 220
E.6.7 S_router 222
E.6.8 S_router_tree 225
E.6.9 S_sink 228
E.6.10 S_source 230

Chapter 1

Introduction

As CMOS technology advances, it becomes possible to design very large and complex circuits on a single chip. Because the designs are so large and complex, the current trend is to combine a number of predesigned reusable blocks such as microprocessors, digital signal processors (DSPs), memories, input/output controllers, and special purpose data processing blocks. Some of these blocks could be bought from other companies as "black boxes", while others might be designed in-house. One of the major challenges for the designer is to create a communication structure which allows the different blocks to exchange data.

A shared bus is one of the possible solutions the designer can choose from. A problem with the shared bus is that the bandwidth becomes a possible bottleneck when many blocks are using the same bus. Also, the capacitance of the bus raises dramatically with an increasing number of connected blocks and length of the bus. This increases the power usage and decreases the speed of the bus.

Another possibility is the fully connected network, where all blocks are directly connected. The number of wires in a fully connected network is a second order function of the number of communicating blocks, which makes it infeasible for a large number blocks. Even for a small number of blocks the large number of wires complicates routing and each wire might require a bus driver depending on the distance it spans on the chip.

Common for the shared bus and a fully connected network is that the designer faces a growing problem as more and more blocks are embedded on the same chip. As the same clock has to be distributed over the entire chip, timing closure is an ever increasing problem. Because of this, the *Semiconductor Industry Association* roadmap predicts that by 2007 many designs will be Globally-Asynchronous Locally-Synchronous (GALS) where each block is running in its own clock domain while communicating asynchronously. This can be accomplished by incorporating a small routing network on the chip, denoted a Network-on-Chip (NoC).

1.1 Network-on-chip

A NoC consists of a number of router nodes connected by point to point links. Figure 1.1 shows a simple example of a NoC where the router nodes are connected as a mesh topology. This means, that the network can be expanded by adding new router nodes to the network,

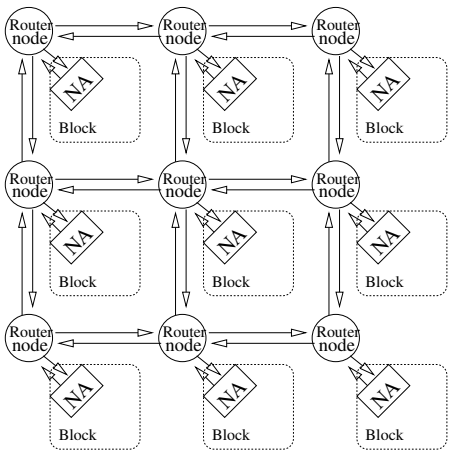


Figure 1.1: Example of simple homogeneous NoC. Each block is connected to a router node through a network adapter (NA), and the router nodes are connected in a mesh topology using bi-directional links.

which makes the network extremely scalable. Because the router nodes are connected with short point to point links, the need for large drivers are minimized, and it is possible to pipeline the communication and thereby increase the bandwidth for a certain link width. One can say, that the long wires are segmented into smaller pipeline stages, which increases the bandwidth for a very small cost because the need for large drivers is no longer present. By sharing the same links, the number of wires on the chip decreases significantly, and the homogenous structure of the mesh topology makes routing a relatively easy task. By separating the blocks from each other by means of the network, it is possible for the different blocks to run in separate clock domains, such that timing closure can be done for each individual block instead of the entire system.

The blocks are connected to the NoC through a network adapter, which could e.g. use the Open Core Protocol (OCP) [1]. OCP defines a common standard for the interface between the blocks and the network. In theory, this makes it possible to facilitate "plug and play" System-on-Chip (SoC) designs, where any Intellectual Property (IP) block can communicate as long it uses the OCP.

A block communicates by means of its network adapter, which sends data into the actual network. The data is passed from router to router node until it reaches its destination. The topology of the network does not need to be a mesh, and can for example be chosen such that the number of wires to be routed for the specific application is minimized. A more in depth overview of NoCs is given in chapter 2.

The NoC can be implemented as both synchronous, asynchronous or a mixed solution. In this project an asynchronous implementation is chosen. Some of the advantages are implicit flow-control, no dynamic power consumption when idle, no clock to be routed in the network, decreased electromagnetic emission, robust to process variations and battery voltages, and decreased electro migration. A short introduction to asynchronous circuits is to found in chapter 3.

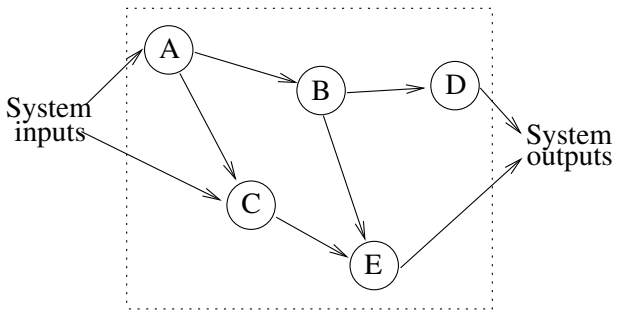


Figure 1.2: Illustration of the dataflow through the 'Aphrodite DSP'. The circles illustrates individual audio processing blocks and the arrows illustrates how data flows between the different blocks.

1.2 Previous work

Currently, many universities are doing research in both synchronous and asynchronous NoCs. Some of these NoCs are 'Nostrum' from the Royal Institute of Technology in Stockholm [12], 'Xpipes' from University of Bologna [5], 'Mango' from the Technical university of Denmark [6], and 'Chain' from the university of Manchester [3]. The first three use the Open Core Protocol (OCP), which relies on Read/Write transactions and the mesh topology as illustrated in figure 1.1. As the router nodes implement 5x5 switches they are relatively large and contain a considerable amount of buffers as they supply advanced features such as virtual channels and guaranteed services[1]. The OCP is not used in this project because this specific application does not rely on Read/Write transactions as will be explained in the succeeding section. The network designed in this project does not need to be this flexible and feature rich, thus the design philosophy is to keep the network as simple as possible. The 'Chain' network, which consists of narrow asynchronous links, has such characteristics, and will be used to implement one of the NoCs designed in this project.

1.3 Project description

In this project three simple asynchronous NoC solutions are designed and implemented for an existing special purpose DSP, denoted the 'Aphrodite DSP' or just 'Aphrodite'. The goal is to replace the existing network with a NoC and compare these in terms of power and area.

'Aphrodite' is a multi-configurable DSP-core for audio applications developed by *William Demant Holding*. It consists of a number of audio processing blocks which are connected by a small network. The network is used to set up a circuit-switched dataflow between the blocks as shown in figure 1.2. The circles illustrate individual audio processing blocks, and the arrows illustrate how data flows between the different blocks. As the chip is to be used in a number of different applications, the dataflow can be changed by reconfiguring the network. The network used to configure the dataflow is currently implemented as a subset of a fully connected network

[1] Chapter 2 goes into more detail about these terms

which has a number of disadvantages as already mentioned. In addition, the network is not scalable as it is tailored to this specific application and must be redesigned if blocks are added or removed. Also, as it is not a fully connected network, some of the blocks cannot communicate at all. Even though design effort are used to design this network, there still are potential routing problems due to the large number of wires. If the number of blocks are increased in future versions of the chip, the size of the network would increase dramatically, making the current network solution infeasible. In contrast a NoC is fully scalable and all blocks can communicate which eliminates the need for any ad-hoc networks solutions. In theory, the NoC is 'plug and play' which decreases the development of new chips besides making it easier to do timing closure, because the individual blocks are decoupled by the network.

Since the audio chip is a real application, and because *William Demant Holding* has helped integrate the new NoCs into the original 'Aphrodite DSP', it is possible to compare the existing network solution with the suggested ones in terms of power and area. To my knowledge, NoC has only been tested in academic applications or very small application with only 3-4 blocks. This is therefore an exceptional opportunity to see how NoCs compare to a traditional network solution and hopefully make some interesting and usable observations. Even though the size of the network is small with only 12 communicating blocks, the needed bandwidth is very limited, and the network utilization is low, this small application provides an example that asynchronous NoCs are usable in real applications. If the NoCs turn out to use more power and area than the existing network, it might still be a good solution in future generations of the audio chip.

The challenge in this project is not to design a large complex NoC, but instead to design a very simple NoC which fulfill the needs in this specific application. The implementation is kept as simple as possible and does not include huge amount of buffers, virtual channels or guaranteed services. Design decisions are discussed in a number of different subjects which include data encoding, network topology and how to handle multicasting. In order to implement the NoCs, a design flow which allows the implementation of asynchronous circuits must be established. A large part of this report is therefore about implementing the network using the cell library used in the original 'Aphrodite DSP' and how to handle asynchronous circuits in the synchronous design flow used at *William Demant Holding*. Besides the actual network many things such as network adapters, multicast controllers, and synchronization units must be designed.

The report documents all the steps needed to design an asynchronous NoC using a standard cell library, the implementation of 3 different NoCs, the integration of the NoCs into the existing design, and a discussion of the results. The designs are not 'Place & Routed', but mapped to gate-level in a 0.18 μm technology upon which estimates of the power and area are made.

1.4 Report structure

The report is structured such that chapter 2 and 3 contain background information about NoC and asynchronous circuits. Chapter 4 introduces the design methodology and how to design asynchronous blocks. 'Aphrodite' is introduced in chapter 5, while chapter 6 defines a new interface to the network such that the existing network can be substituted by a NoC. The actual network designs are discussed in chapter 7 and implemented in chapter 8. Verification is dis-

cussed in chapter 9 and notes about the logic synthesis and simulation flow are given in chapter 10. The results are presented and discussed in chapter 11 and finally chapter 12 concludes what has been archived in this project.

Gate-level implementations of all designed blocks can be found in appendix D and the code for the blocks are included on the CD-ROM and in appendix E. A short description of the CD content is included in appendix C.

Chapter 2

Background: Network-on-chips

This chapter gives a general overview of NoCs and the different terms which are used to describe them. Even though comments are made through the chapter concerning the specific application, it can safely be skipped if such an introduction is not necessary.

2.1 Overview

Network-on-chip is a very broad term which simply states that some kind of communication network is implemented on the chip. When designing the network, many choices and tradeoffs must be made and the optimal network depends on e.g. the expected workloads, power constraints, physical constraints, number of communicating blocks, scalability, performance, and ease of wire routing. This also means, that there is no network design which is perfect in all applications and designs. The information used to write this section is mainly found in [11].

2.2 Network type

A network can be classified as a *shared-medium network*, an *indirect network*, or a *direct network*. Each type will be introduced in the following.

A shared bus is an example of a *shared-medium network*, where the network can only be used by one block at a time. Due to the high number of communicating blocks the shared network is not an option in this context. The bandwidth would probably suffice, but the capacitance of the bus would be very large because of the distance it spans and the number of connected blocks.

Figure 2.1 shows an example of a *direct network*, where each block interfaces the network through a network adapter which is connected to a router node. The router nodes are connected using either uni- or bi-directional links which allows data to be transferred between any of the connected blocks. In a *direct network*, each router node must be connected to a block, and router nodes are considered part of the blocks. This means that the blocks are considered to be directly connected, hence the term *direct network*. When a block wants to communicate, it sends data to its network adapter which handles the actual communication. The router nodes do not need a direct link to the destination router node, since data is transferred through intermediate router

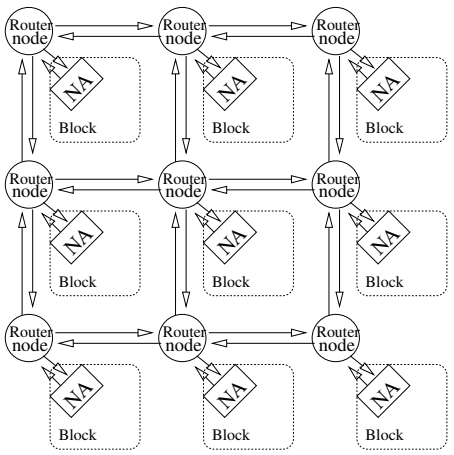


Figure 2.1: Example of simple homogeneous NoC. Each block is connected to a node through a network adapter and the nodes are connected in a mesh topology using bi-directional links.

nodes. Because the blocks communicate with the network through a network adapter, they do not need any information about the network implementation.

In contrast to the *indirect network* a *direct network* also contains independent router nodes which are not connected to any block.

2.3 Packets and flits

The data which is communicated between the different blocks are encapsulated into *packets*. Depending on the used switching technique the *packet* can contain a header with information such as addresses of the destination nodes or the route to be used[1]. Besides a header, the *packet* also contains a payload which is the actual data. If the size of the *packet* is larger than the width of the point-to-point links between two router nodes, the *packet* is partitioned into a stream of flow control digits (*flits*) which are sent over the link one at a time. The size of a *flit* is the number of bits which can be sent concurrently on a link and of course depends on the width of the link. The width of the links do not need to be constant and can for example be varied in different areas of the network depending on the bandwidth need for the specific link. Depending on the implementation the number of *flits* in a *packet* can be constant or special *tail flit* can be used to indicate the end of the *packet*.

2.4 Switching techniques

Communication is performed by forwarding *packets* between the different router nodes till it reaches its destination. This means that a router node must decide how to handle each received *packet* as it can be sent on any of the outgoing links. This is denoted the route of a *packet* and

[1]Switching techniques are introduced in the succeeding section

must be controlled using one of a number of different switching techniques. One possibility is to apply *circuit-switching*, where a path is reserved from the source block to the destination block before sending any data. It takes some time to reserve the path but it is very fast to send data when the reservation is complete. *Circuit-switching* is especially useful for infrequent communication of large lumps of data which is not the case in this application. This switching technique also locks the router nodes such that other communication is blocked.

Instead, the data can be divided into small *packets* which are sent one at a time and individually routed. This is called *packet-switched* communication, because each *packet* is routed individually instead of being sent using an already established route. *Packet-switched* routing exists in 3 different variants: The first is denoted *store-and-forward*, since a route node receives and stores an entire *packet* before forwarding it to the next router node. This requires that the buffers in the router nodes are large enough to contain an entire *packet*, thus increasing both the size of the router nodes and their latency.One major advantage is that *packets* can be interleaved through a router node and that deadlock cannot occur if the buffers are large enough. If the *packet* only consists of a single *flit* the entire *packet* can be sent concurrently on the *links*, making the switching inherently *store-and-forward*. The second switching technique is *virtual cut-through switching* which basically works the same way as a *store-and-forward* except that a router node starts forwarding the *packet* before it has been received entirely. The buffers in the router nodes are still large enough to contain an entire *packet*, but the latency through the network is decreased compared to a *store-and-forward* network. The last switching technique is *wormhole switching* which is the exact same thing as *virtual cut-through switching*, except that the buffers are so small that they cannot contain an entire *packet*. This means that a *packet* always spans several router nodes and links. If the *packet* is blocked for some reason, it can easily result in a deadlock. In order to avoid deadlocks special routing techniques can be applied or *virtual channels* [8] can be introduced. A number of *virtual channels* share the bandwidth of a single physical link using for example time division or other sharing techniques. Each *virtual channel* needs its own separate buffer in the router node and circuitry must be added to implement the sharing of the psychical link. Both increase the size of the router node.

2.5 Routing

The route of a *packet* can be either *deterministic*, that is, determined before the *packet* is sent, or *adaptive*, where the route is determined dynamically on a per router node basic. When *adaptive* routing is applied, a central routing controller or the individual router nodes determines the route of each *packet* based on the current traffic load in different parts of the network. In theory this dynamically balances the load on the network and thereby reduces possible bottlenecks. If some of the links suddenly start to malfunction, these links could be avoided. Since communication between two specific blocks do not always take the same route the *packets* may arrive out of order which further complicate things. All in all, *adaptive* routing leads to very complex, large, and slow router nodes and is not an option in this project.

When the route of a *packet* has been decided the router nodes must know how to route the *packet*. This can be done as *network routing* where the *packet* simply contains a unique address of the destination block. The router node then determines the route by looking in a routing table

which can be changed dynamically by e.g. a central routing controller. This solution requires large routing tables in each router node as well as circuity to look up the route. Also, the size of the routing tables depends on the number of communicating blocks. Instead, the route can be determined at the source block and contained in each *packet*. This is denoted *source routing* and makes the router nodes very simple, as they do not take any route decisions. *Source routing* is currently used in all the NoC articles that I have encountered because of the simple router node implementation.

2.6 Guaranteeing bandwidth

Most NoC implementations use Best Effort (BE) routing where data is sent as fast as currently possible. The time it takes for a *packet* to arrive at the destination depends on the current network load and is therefore dependent on other communicating blocks. Some applications require the introduction of Guaranteed Services (GS) where 2 communication parties are guaranteed a certain amount of bandwidth. This is the case in e.g. multimedia and audio applications where guaranteed continuous streaming of data is required. Research has also been done in combining best effort routing with guaranteed services. One approach, which is presented in [6], is to provide GS by a *virtual circuit-switched* network by reserving a certain amount of bandwidth on each link on the communication path. Instead of guaranteing bandwidth, one could also imagine that network traffic is prioritized depending on the importance, thereby providing Quality of Service (QoS).

2.7 Topology

The choice of topology depends on many different aspects such as number of communicating blocks, scalability, ease of routing etc. A mesh structure, which is illustrated in figure 2.1, is the most used topology because it extremely very scalable. The number of blocks can be increased by adding new nodes without altering the existing layout. Also, the routing of wires can be done very easily. Some of the disadvantages in this topology is that the nodes are quite complex as they contain a 5x5 crossbar and a large amount of buffers. Other topologies include hyper-cubes, binary trees, fat trees, hierarchical structures, hybrid solutions, and many more. A discussion of which topology to use in this project is presented in chapter 7.2.

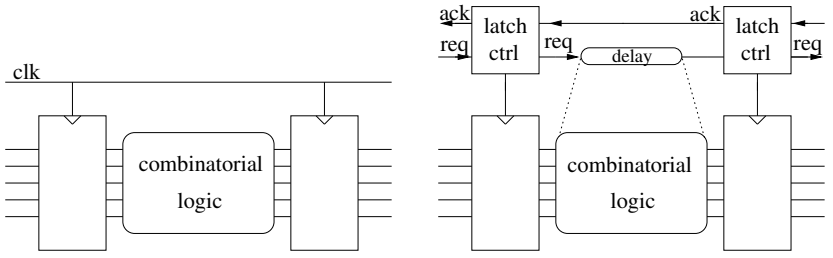
Chapter 3

Background: Asynchronous circuits

This chapter gives a short introduction to asynchronous circuits with emphasis on handshake protocols and advantages over synchronous circuits. It is by no means a complete introduction as the ones which can be found in textbooks as for example [13].

3.1 Overview

Traditional synchronous design consists of combinatorial logic separated by latches or registers as illustrated in figure 3.1a. The slowest path through the combinatorial logic determines the highest clock frequency at which the circuit can be clocked. Since all registers/latches are clocked at the same time there will be a surge of power every time the clock ticks. These surges lead to increased electromigration which decreases the lifetime of the chip and is an increasing problem as technology size decreases. The power spectrum is highly non-uniform and contains spikes at the clock-ticks which give rise to electromagnetic emission that can disturb analog devices in the product. The non-uniformity also leads to lower battery time if the product is battery driven due to the nature of batteries. If parts of the chip are idle for periods of time, as is the case with a NoC, clock-gating must be explicitly applied to ensure that the registers/latches

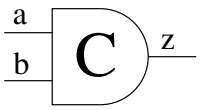


(a) Synchronous circuit.

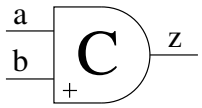
(b) Asynchronous circuit. The delay must be larger than slowest path in the combinatorial logic.

Figure 3.1: In asynchronous circuits the clock is substituted with handshake controllers.

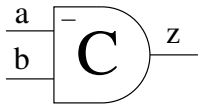
a	b	z
0	0	0
0	1	no change
1	0	no change
1	1	1



(a) Truthtable and symbolic representation of a 2 input C-element with function $z = ab + z(a + b)$.



(b) Asymmetric C-element with function $z = ab + za$.



(c) Asymmetric C-element with function $z = b + z(a + b)$.

Figure 3.2: Truthtable and symbolic representation for different Muller C-elements.

are not clocked during idle periods.

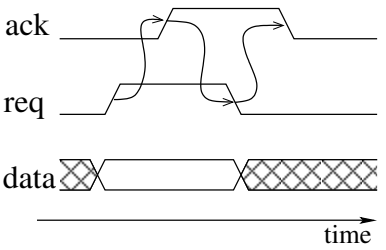
In contrast, different parts of asynchronous circuits run at their own pace as registers/latches are not clocked by a common clock. This is done by exchanging the clock with handshake circuitry as illustrated in figure 3.1b. Asynchronous circuits do not have any dynamic power consumption during idle periods, and since no clock has to be distributed, the increasing problem of clock skew and large clock trees are eliminated. As wires are getting taller, narrower, and placed closer together, crosstalk is also an increasing problem in synchronous circuits. If a delay-insensitive one-hot encoding is used, the problem with crosstalk is decreased because wires which are routed together do not make transition at the same time.

There is no such thing as a free lunch. First of all the well-proven synchronous designflow which is known by thousands of designers cannot be used directly, and commercial asynchronous design tools are almost non-existing. As technology decreases the leakage current increases heavily which means that the static power consumption is being a larger and larger part of the total power consumption. As asynchronous circuitry tend to be larger than the equivalent synchronous circuit, one of the major advantages might no be valid for future technologies.

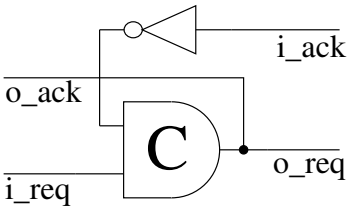
3.2 The C-element

The Muller C-element plays a central role in the construction of asynchronous circuits. The truthtable of a C-element with 2 inputs as well as its symbolic representation is shown in figure 3.2. The C-element implements the logic function $z = ab + z(a + b)$ and is a state-holding device. In contrast to an AND gate which indicates when the inputs are all 1, and an OR gate which indicates when the inputs are all 0, the C-element indicates both. This is also known as a join or rendezvous.

C-elements can also be asymmetric which means that not all inputs need to be the same for the C-element to change state. For example the C-element in figure 3.2b implements the function $z = ab + za$. The b input is denoted "plus" because it is only used in the rising transition. Both



(a) Illustration of the 4 phases.



(b) Implementation using a single C-element and an inverter.

Figure 3.3: 4-phase bundled data handshake. The data is valid whenever request is high which is denoted the *extended early* data-validity scheme.

inputs still need to be '1' for the output to change to '1', but only input *a* needs to be '0' for the output to go low. The C-element in figure 3.2b implements the function $z = b + z(a + b)$. The "minus" indicates that the *a* input is only used in the falling transition. Both inputs still need to be '0' for the output to change to '0' but only input *b* is needed for the output to go high.

3.3 Handshake protocols

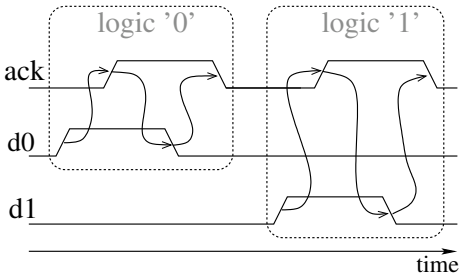
Asynchronous circuits can be constructed using either bundled data or using a delay-insensitive encoding. The 2 different possibilities are introduced in the following subsections.

3.3.1 Bundled data

All bundled data handshake protocols substitute the clock with handshake controllers, but keep the combinatorial logic as illustrated in figure 3.1b. A delay which is larger than the slowest path in the combinatorial logic must be inserted in the request wire.

The simplest and widely used handshake protocol is the 4-phase (Return-to-Zero) bundled data protocol as illustrated in figure 3.3a. As the name '4-phase' indicates, the handshake consists of 4 phases: 1) the sender raises the *request* wire to indicate that data is valid, 2) the receiver raises the *acknowledge* wire to indicate that the data has been received and latched, 3) the sender lowers the *request* wire, 4) the receiver lowers the *acknowledge* wire which completes the handshake cycle. Figure 3.3b shows an implementation of a latch controller which is known as a Muller pipeline[1]. Each stage implements such an un-decoupled 4-phase latch control circuit using a single C-element and an inverter. The controller is denoted un-decoupled because the incoming and outgoing handshakes of the controller are strictly coupled. This means, that two succeeding latches cannot contain data at the same time. The two handshakes can also be fully decoupled but this increases the complexity of the latch controller as well as the propagation delay. Details about the implementation of different 4-phase latch controllers can be found in [9]. In the Muller pipeline from figure 3.3, the sender starts the handshake cycle which is known as a *push* scheme because the data is pushed by the sender. In contrast the handshake is initiated by the receiver in the *pull* scheme by raising the *request* wire to indicate that data can safely be

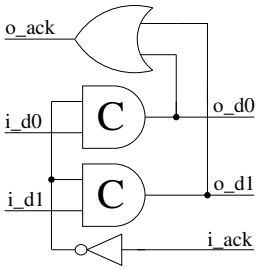
[1]Named after the inventor



(a) Illustration of the 4 phases in a logic '0' and logic '1'.

d0	d1	meaning
0	0	Empty
0	1	1
1	0	0
1	1	Not used

(b) Truthtable.



(c) A dual-rail latch implemented using C-elements.

Figure 3.4: 4-phase dual-rail handshake. The request signal is implicitly given as the data lines are using a one-hot encoding where they are mutual exclusive.

sent. As indicated on figure 3.3a, the data is expected to be valid when *request* is high which is denoted the *extended early* data-validity scheme. Different *data-validity schemes* exists, which defines in which part of the handshake data is valid [13].

The handshake can also consist of 2 phases (non-Return-to-Zero) instead of 4. This decreases the number of transitions in the handshake cycle but complicates the handshake circuitry. It is also possible to combine the *request* and *acknowledge* wires into a single wire. As the wire is driven by both the sender and receiver, it must have high impedance to keep its value when it is not driven.

3.3.2 Delay-insensitive encoding

Another possibility is to use delay-insensitive encoding where the data is encoded using a one-hot scheme. The simplest example is called dual-rail where each bit is encoded into two wires as illustrated in figure 3.4. The truthtable for the encoding and a pipeline stage which employs a 4-phase protocol is shown. The 4 phases of the handshake are: 1) the sender raises $d0$ to indicate a logic '0' or $d1$ to indicate a logic '1', 2) the receiver raises the *acknowledge* wire to indicate that the data has been received and latched, 3) the sender lowers $d0$/$d1$, 4) the receiver lowers the *acknowledge* wire which completes the handshake cycle. Note that the 2 wires are mutual exclusive and the request signal is implicitly given. It takes 4 transitions to communicate 1 bit independent of the data value.

Several one-hot lines can be combined into a bus by using a special 'completion detection' unit which detects when data is present on all lines and when all lines have returned to zero. This is normally implemented using a C-element or a tree of these if necessary. When several one-hit lines are combined into a bus they are using a single *acknowledge* wire.

Instead of encoding a single bit into 2 wires, a higher order encoding could also be chosen. As an example *1-of-4* encoding could be employed where 2 bits are encoded into 4 wires. The advantages are that 2 bits are transferred using the same number of transitions as it takes to transfer 1 bit in a dual-rail implementation. The size of the 'completion detector' also decreases compared to the dual-rail as the number of lines for a N-bit word decreases. If the number

Protocol	**Wires**	**Transitions**
Bundled data (Return to zero)	N+2	avg(N)+4
Dual-rail	2N+1	2N+2
1-of-4	2N+1	N+2

Table 3.1: Number of wires and transitions for 3 different data encodings using a 4-phase protocol where N is the number of bits in a data word. The probability of '0' and '1' are assumed to be 50% for both.

of wires do no contribute to the power consumption, 8 bits of data could be sent using just 4 transitions using a *1-of-256* encoding.

As with the bundled data protocol, a 2-phase protocol could also be chosen but this complicates the circuitry.

3.3.3 Comparison

Table 3.1 lists the number of wires and transitions used to transfer a single word of N bits for a selection of 4-phase protocols. The number of wires includes the *acknowledge* and *request* wires. Note, that the number of transitions is constant for the one-hot encodings while it depends on the actual data for the bundled data. I assume that the probability is 50% for both '0' and '1' and that the data lines returns to zero after each handshake. This might not always be the case.

The advantages of one-hot encoding are that there is no need for matched delays and the circuits are truly delay-independent. This means that the circuitry will work no matter how large the wire and gate delays are. After the chip has been manufactured it will work independent of the temperature, process variations, and even supply voltage. The speed of the chips will differ but will work as expected. A delay-insensitive implementation is used in some high bandwidth network chips because it is possible to make them operate at very high speeds. It might also be an advantage that the number of transitions is independent of the actual data as this makes the power usage predictable.

Some of the disadvantages are that at least two wires are needed for each bit, that normal combinatorial circuits cannot be used, and that the corresponding one-hot implementation is potentially much larger and slower. In this project the network is not doing any computation which means that a one-hot encoding might be a good solution.

Chapter 4

Design methodology

This chapter describes how the asynchronous blocks are designed and implemented at gate-level.

4.1 Overview

As it will be explained in chapter 8 the networks are constructed by connecting a number of small carefully designed building blocks. These block consists of a mix of speed-independent control circuits and bundled data circuits. Logic synthesis tools are specialized in synthesis of synchronous designs and cannot be uses in the synthesis of asynchronous circuits. In this project, the asynchronous circuits are designed by deriving a set of speed-independent boolean expressions which are implemented as netlists of standard cells. The designs are marked as 'do not touch', such that the logic synthesis tool does not optimize the circuits.

In the following sections the design of asynchronous controllers are explained in a bottom-up fashion starting from the use of standard cell libraries and all the way to the finished blocks. This includes including complex asynchronous controllers, matched delays in bundled data circuits, and how to handle initialization.

4.2 Standard cells and drive-strengths

In this project asynchronous controllers are designed as a netlist of standard cells. Since the delay through each cell is carefully timed, we cannot use automatic drive-strength optimization. Instead, each standard cell is implicit instantiated including the drive-strength. This allows us to carefully control the delay through each block as well as the capacitance on the inputs and drive-strength of the outputs.

Many cells in a standard cell library exist in 2-5 different versions with different drive-strengths. Increasing drive-strength means that the cell has larger fanout and thereby can drive more cells, but the size of the cell as well as the typical propagation delay increases. In some standard cell libraries the input capacitance of the cell increases as well. The standard cell library which is used in this project has almost constant input capacitances for all drive-strengths except inverters, buffers, and high-performance gates. The input capacitance for a cell with drive-strength 1 are denoted *unit input capacitances* through the rest of the report. In the 0.18 μm

process used in this project, the unit input capacitance is 40-60 nF. A cell with drive-strength 1 can drive approximately 4 inputs with unit capacitance at maximum speed. If the fanout is larger, a cell with a larger drive-strength must be used, or the signal must be buffered to a larger drive-strength. The buffering is normally done in a number of stages with an increase in drive-strength by a factor 3-4 in each stage as this gives a good performance.

It would be nice to have a tool which could automatically choose the optimal driven-strengths of each instantiated standard cell, but unfortunately no such tool exists for asynchronous circuits at this point of time. An automatic tool could also identify the longest paths in the circuit and slow down other paths to decrease the used power and area.

Instead, the drive-strengths are chosen manually based on some simple rules of thumb which gives a good, but not optimal, solution. There is room for optimization in the size, power-usage and speed of the circuit by choosing more optimal drive-strengths. Circuit optimization is not important in this project since the purpose is not to produce a highly optimized solution, but to show the concepts of an asynchronous NoC. Doing this kind of optimization by hand takes a long time and the drive-strength must be recalculated every time the circuit is changed, or the standard cell library is replaced.

Generally, the blocks are designed such that the outputs have a drive-strength of 1 and the inputs have unit capacitances. While this might not be optimal in terms of power, speed, and area, it is a good comprise that makes it easier to connect the blocks as all inputs have the same capacitance and all outputs have the same drive-strength. Inside the individual blocks, cells with drive-strengths 1 are used as a cell seldom drives more than 4 other cells. If a cell drives more than 4 inputs a cell with a larger drive-strengths is used or a buffer is inserted. Since most cells in the used standard cell library have unit input capacitance, independent of the drive-strength, a cell with larger drive-strength is generally used in this project. By ensuring that all cells have unit input capacitances, the drive-strength of a cell is only dependent on the number of cells that it drives. If this wa not the case the drive-strength of a cell would be dependent on the number of cells that it drives **and** the input-capacitances of these. Since this blows up the complexity of the problem, it is ensured that inputs always have unit capacitances. If a cell library is used where the input capacitances increase with the drive-strength, a buffer should be used at the output of a cell with drive-strength 1.

In some of the small asynchronous controllers it might be beneficial to use standard cells with drive-strength $\frac{1}{2}$ which are both faster and use less power.

The following summarizes how to choose the drive-strengths of the cells:

- Outputs of blocks have drive-strength 1 and inputs have unit capacitance.
- Generally cells with drive-strength 1 are used
- If a drive-strength larger than 1 is needed, a cell with this drive-strength is used if:
  1. Such a gate exists
  2. The inputs to the cell still have unit capacitance

  If this is not the case a buffer where each stage increases the drive-strength by 3-4 is inserted instead

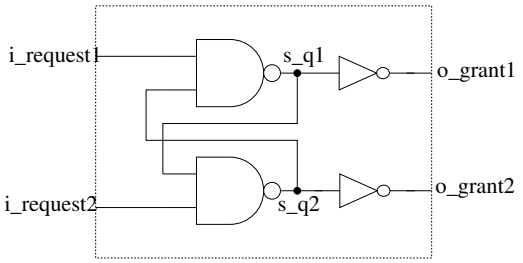


Figure 4.1: Implementation of a mutex.

- Inside the asynchronous controllers, cells with drive-strength 1/2 are used in some cases. (but outputs of the block must still have drive-strength 1).

4.3 Basic asynchronous components

In order to design asynchronous controllers a few asynchronous elements, which do not exist in a ordinary cell library, must be created. This is the mutex and a collection of different C-elements. Since custom cells are hard to implement and must be re-implemented if a new technology is used, it is an advantage to construct these from available standard cells.

4.3.1 Mutex

The mutex is a component which ensures that two signals are mutually exclusive. This is used to control access to shared blocks and is used when 2 busses are merged into one. It consists of two inputs and two outputs, and its function is to ensure that at most one of the outputs is high at any point of time. Figure 4.1 shows how this is implemented using two crosscoupled NAND gates and 2 inverters. The 2 NAND gates handle the actual arbitration while the 2 inverters act as a metastability filter to ensure that the outputs are never high at the same time. In the initial state both inputs are low, the two intermediate nodes *s_q1* and *s_q2* are high, and both outputs low. If *i_request1* becomes high, *s_q1* goes low which ensures that *s_q2* stays high independent of *i_request2*, and that *o_grant1* becomes high. The behavior is similar if *i_request2* becomes high. The arbitration comes into play if the two inputs become high at the same time. First, the voltage at *s_q1* and *s_q2* will drop to about half of the supply voltage and enter a metastability phase where the two NAND gates are trying to drive their respective outputs low. Eventually one of them "wins" the race and either *s_q1* or *s_q2* goes high while the other goes low. During this metastability phase it is extremely important that none of the outputs becomes high as both of NAND gates could turn out to be the "winner" and create a hazard on one or both outputs. The two inverters work as a metastability filter which makes sure that none of the outputs go high when the intermediate nodes are in the metastability phase. The threshold voltage of the inverters is therefore important and must be well below half of the supply voltage. The shown metastability filter is just one of many possible implementations, but common is that a detailed analysis must be made at transistor level using the parameters from the used cell library.

There are several problems during simulation with the illustrated implementation. Both problems are due to the fact, that simulators only do binary simulation on logic-levels 0 and 1. First, the simulator enters an infinite loop when both inputs become high at the same time, which deadlocks the simulation and makes both outputs infinitely alternate between 0 and 1. Second, the metastability filter does not work at all. Both problems are simulator specific and can be considered as *false errors* because they will never happen in the produced chip.

One way to get around this is to do synthesis as normal and replace the mutex with a behavioral model during simulation. This means that the area estimates are made with the real mutex, while the delay and power estimates are made with a behavioral model. The SDF file which contains the timing of the mutex must therefore be changed to contain the estimated propagation delay of the mutex.

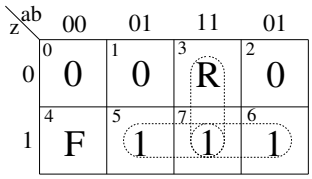
In this project, the behavioral version is used when simulation on RTL level while at netlist version is used when simulation on gate-level. Simulation on the mutex, shows that the it works as expected but that it sometimes produce a glitch on one of the outputs. This is not a problems, since the blocks which contain the mutex do not malfunction because of a small glitch. If the mutex was used in other blocks, it might has to be replaced by its behavioral version.

4.3.2 C-elements

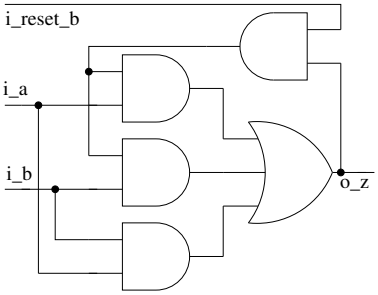
The C-element is a state holding component which indicates when all its inputs are either 0 or 1. C-elements can be implemented in a number of different ways which all capture the correct functionality. The number of inputs often determines which method that takes up the least area. Ont method is to implement the C-elements using complex gates. Since standard cell libraries do not always have the same types of complex gates, the C-elements probably have to be re-implemented if the cell library is replaced. Figure 4.2b shows a possible implementation of a 2 input C-element using a complex gate containing a feedback loop, such that it implements the function $z = ab + z(a + b) = ab + zb + za$. The C-element can be reset to 0 by setting all the inputs to 0. This might not always be possible during the reset phase and by inserting an AND gate in the feedback loop, it is possible to reset it to 0 by setting just one of its input to zero. One could insert the reset gate at the output instead, but this would increase the propagation delay of the cell.

Figure 4.2c shows the implementation of a 3 input asymmetric C-element with the function $z = abc + z(a + b) = abc + za + zb$. The *i_c* input is a "plus" input which must be 1 for the output to go high, but does not need to be 0 for the output to go low.

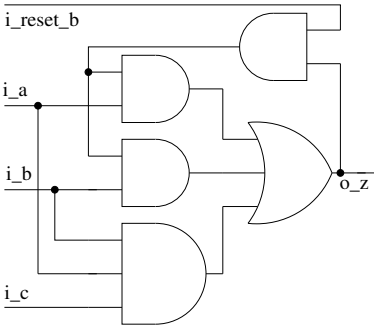
Since the C-element is not an atomic cell but created of a complex gate with a feedback loop, some assumptions must be made concerning the environment and the routing of the feedback loop in order to avoid hazards. This is best illustrated by inspecting the karnaugh map of the 2 input C-element implementation which is shown in figure 4.2a. The dotted areas represent the min-terms, *F* indicates that the output is doing a falling transition and *R* that the output is doing a rising transition. A dynamic hazard can occur if both inputs are 1 and the output is making a rising transition from state 3 to state 7. Just as the output changes to 1, the environment changes both *a* and *b* to 0 before the two min-terms have taken over. This means that the output might change to 0 and afterwards become 1 for at short period due to one of the other min-terms. The problem is that one min-term is "taking over" from another and is an important issue when



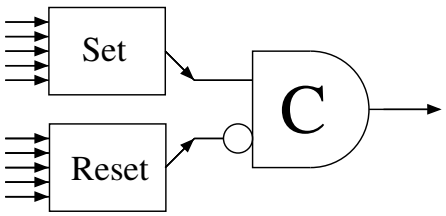
(a) Karnaugh map of 2 input C-element with the logic function z=ab+z(a+b).



(b) 2 input C-element with logic function: z=ab+z(a+b).



(c) 3 input asymmetric C-element with logic function: z=abc+z(a+b).



(d) Larger C-elements can be constructed from a 2 input C-element and a set and reset function.

Figure 4.2: Implementation of different C-elements.

designing asynchronous components.

In order to avoid this hazard the feedback connection must be stabilized before both of the input changes. I presume that the feedback loop is routed locally, and, as I only include the delay of an OR gate in the feedback loop, this should be the case. The C-elements can also be implemented using simple gates, but this increases the problem with hazards and demands further assumptions about the routing.

If C-elements containing many inputs are needed, it might not be possible to design them using a single complex gate. Instead a 2-input C-element can be used as a state-holding device with a set and reset input as illustrated in figure 4.2d. A latch with asynchronous set and reset input can also be used. This method might take up less area for large C-elements. Note, that the set and reset logic must be designed such that it does not produce any dynamic or static hazards.

4.4 Complex asynchronous controllers

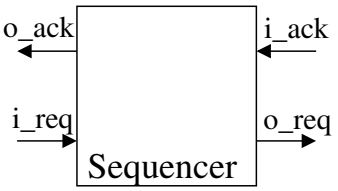
When designing complex asynchronous controllers a tool is needed to ensure a hazard free implementation. In the project I have used Petrify [7] which can be used to synthesize Petri nets

and asynchronous controllers. Petrify takes a Signal Transition Graph (STG) which describes the behavior of the asynchronous controller and generates speed-independent boolean expressions. The output can be implemented using either complex gates, C-elements, or technology mapping. I have not looked into Petrify's ability to do technology mapping and have instead concentrated on complex gates and C-elements. When using C-elements, Petrify produces a set and reset function as illustrated in figure 4.2d, while it produces complex boolean expressions when requesting a complex gate implementation. For this project I have used the complex gate option as it produced the smallest circuits. This is because the controllers are quite small. If Petrify gives a solution which requires a complex gate that does not exist in the standard cell library, the C-element option must be used instead. The graphical tool, Visual STG Lab (VSTGL)[2], which is developed at DTU was used to design the STG's.

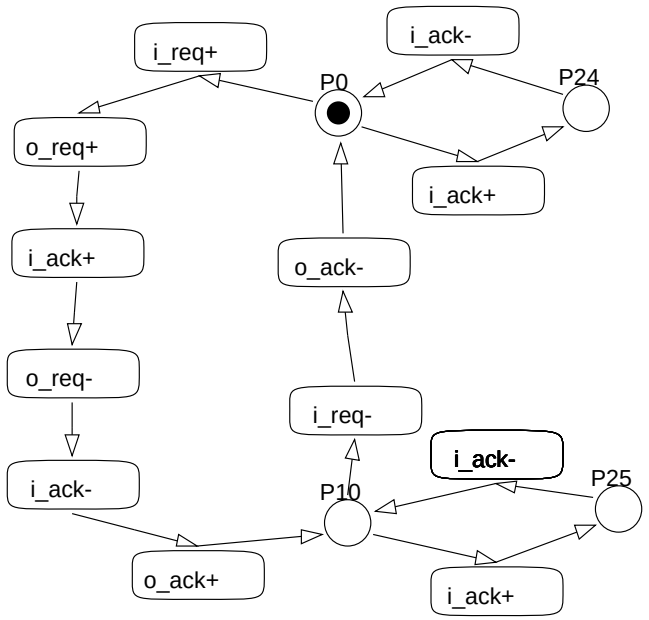
To illustrate the design of a complex asynchronous controller I have chosen to go through the design of a *sequencer* which is a simple 4-phase handshake generator. Figure 4.3a shows the symbol of the *sequencer* and its inputs and outputs. Basically, it accepts a handshake on the left hand side and generates a handshake on the right hand side before completing the handshake on the left hand side. In addition to this functionality the `i_ack` line can alternate when the *sequencer* is currently not performing a handshake. This is because a number of *sequencers* are handshaking on the same *request* and *acknowledge* wires, why the *i_ack* wire must be ignored when the *sequencer* is not currently performing a handshake.

The STG, which describes the order of events for the *sequencer*, is shown in figure 4.3b. Even though the STG captures the wanted behavior, the functionally is best understood by going through the order of events: 1) `i_ack` can make a number of alternations if other *sequencers* are performing a handshake. 2) `i_req` goes high to indicate the a handshake must start. 3) A 4-phase handshake is performed on `o_req` and `i_ack` 4) `o_ack` is driven high to indicate that the handshake has been completed on the right side. 5) `i_ack` can make a number of alternations if other *sequencers* are performing a handshake. 6) `i_req` goes low and `o_ack` is driven low to finish the handshake.

Figure 4.3c shows the output of petrify using complex gates. The boolean expressions for `o_ack` and `csc0` can be identified as asymmetric C-elements and 2 possible gate-level implementations of the controller is shown in figure 4.3d and 4.3e. One very important note is that Petrify assumes that the complex gates exists with both inverted and non-inverted inputs. As it was not possible to design C-elements with inverted inputs using the complex cells in the used standard cell library, inverters are inserted manually. Petrify produces speed-independent boolean expressions which assume that wire delays are zero. Wire delays can be lumped into the gates, except when there is a fork as for example the `s_1` and `i_req` signals in figure 4.3d. The delays from the fork to all end-points should be identical which in asynchronous literature is denoted an *isochronic* fork. As the designed circuits are normally very small, it is ok to assume that this is the case except when inverters are inserted. This is the case for the implementation in figure 4.3d and instead the inverters are removed from the fork as shown in figure 4.3e.



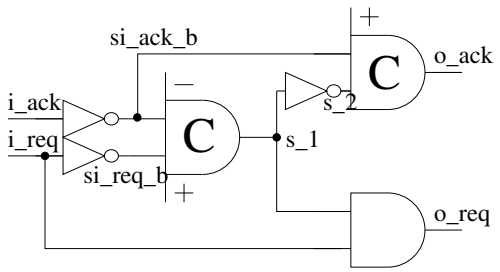
(a) Symbol.



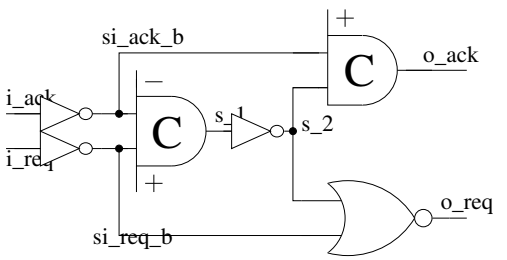
(b) STG specification.

```
# EQN file for model sequencer3
# Generated by petrify 4.2 (compiled 5-Jul-04 at 11:55 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 8.00

INORDER = i_req i_ack o_req o_ack csc0;
OUTORDER = [o_req] [o_ack] [csc0];
[o_req] = i_req csc0;
[o_ack] = csc0' (o_ack + i_ack');
[csc0] = i_ack' csc0 + i_req';

# No set/reset pins required.
```

(c) Output from petrify.

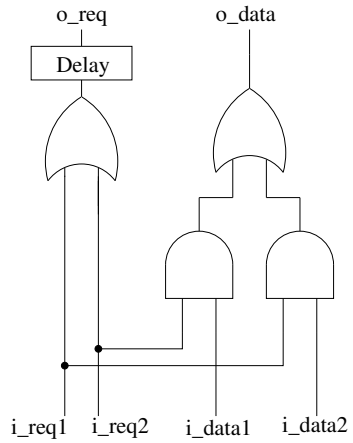


(d) Gate level implementation 1. The forks at `s_1` and `i_req` cannot be considered *isochronic* because of the inverters.

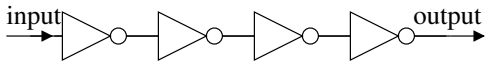


(e) Gate level implementation 2. All forks can be considered as *isochronic*.

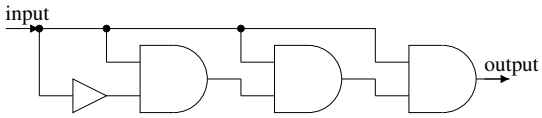
Figure 4.3: The *sequencer* circuit which performs a 4-phase handshake.



(a) The delay must be matched such that request does not go high till *o_data* is stable.



(b) Symmetric delay implementation.



(c) Asymmetric delay implementation.

Figure 4.4: Delay must be inserted in the request path in the bundled data design.

4.5 Bundled data design and asymmetric delay

When designing a component which uses the bundled data protocol, a matched delay must be inserted in the request line as described in chapter 3.1. The matched delay must be larger than the worst case latency of the functional block. Figure 4.4a illustrates a typical scenario which is encountered when designing a component for a bundled data network. The circuit takes 2 request lines and 2 data lines as input and outputs a single data value and request. The input request lines are assumed to be mutual exclusive and control which of the 2 data inputs that are to be outputted. According to the protocol the *o_data* line must be stable before *o_req* goes high, why a delay must be inserted before the output. This delay must be large enough to account for the extra gate-delay which is contributed by the AND gate, but also include the delay which are caused by wires and cross capacitances. In this case the data is a single bit, but it might be a bus, which means that the request is driving several AND gates. It might even be necessary to insert buffers to increase its drive strength. All these delays must be accounted for in the matched delay and is a good example that we want to be in control of the used gates such that we are sure to insert enough delay. As it is hard to predict the exact delay of the circuit, the matched delay must be quite conservative. On the other hand the delay should not be too large, as this will tslow down the circuit and the delay element will be larger and consume more power. In order to validate that the delay is large enough, the circuit has to be *place and route*'ed and the delay back-annotated.

Figure 4.4b shows a simple delay implementation which consists of a chain of inverters. This delay is symmetric as the *low→high* and *high→low* transitions takes the same amount of time. In a 4-phase protocol an asymmetric delay is preferable as the *high→low* transition only decreases the speed of the circuit. One possibility is to use non-balanced buffers, since their *high→low* propagation delay is roughly twice the size of their *low→high* propagation time. An inverter must be inserted before and after the buffers, such that the *low→high* transition that

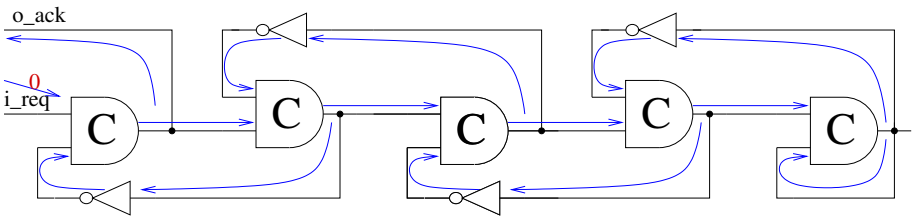


Figure 4.5: Initialization ripples through the circuit.

has the largest propagation delay. Figure 4.4c shows another possible implementation of an asymmetric delay where a *low→high* transition has to propagate through the entire chain of AND gates. In contrast, a *high→low* transition are propagating through all the AND gates in parallel and therefore has a propagation delay of a single AND gate. In complex bundled data circuits which contain large portions of combinatorial circuits, more advanced delay techniques could be used to improve performance. E.g the delay could depend on the data values as these might influence the longest path. This not an issue in this project as the longest path are always constant in the implemented network blocks. Also it is beyond the scope of this project to make a study of asymmetric delay implementations.

4.6 Initializing asynchronous circuits

Before an asynchronous circuit can be used it must be brought into a known state. That is, it must be initialized properly. One way to achieve this is by adding controllability to the outputs of all asynchronous cells. Since this controllability is implemented by adding a number of gates it increases the area, power usage, and propagation delay of the circuit. A better way is to insert controllability in a few places and make sure that the initialization will ripple through the circuit. Figure 4.5 illustrates how a Muller pipeline is initialized by setting *in_req* to 0. Since the only input to the pipeline is the *in_req* and the other input to the first C-element is in an unknown state, the C-elements must contain a reset signal. This allows it to be reset when just one of its inputs are set to 0.

When an asynchronous circuit is designed, the properties which ensures a proper initialization must be noted. This includes which inputs that must be set to a certain logic value and the time it takes for the reset to propagate.

Chapter 5

The Aphrodite DSP

This chapter gives an overview of the Aphrodite chip for which a NoC is designed. Focus is kept on the configurable network which is to be replaced.

5.1 Overview

'Aphrodite DSP' is a multi-configurable DSP-core for audio applications developed by *William Demant Holding*. It consists of a number of audio processing blocks which can be connected different ways using a circuit-switched configurable network. An example of such a dataflow was shown in figure 1.2 on page 3. The DSP blocks include microphone inputs, headphone outputs, audio processing blocks, and an interface to a digital microprocessor. A block does not necessarily contain one input and one output but can contain any number of inputs and outputs.

The network is configured using a special configuration bus which enables the same DSP to be used in many different audio applications. The configuration can even be changed at run-time without resetting the system.

There is a total of 16 input ports and 12 output ports in the network, and generally samples of 18 bits are communicated between the different blocks with a few cases of 16 bits. In the case of 16 bits, the sample is appended 2 bits to make it 18 bits as well.

5.2 Configurable network

The current network is implemented as a subset of a fully connected network. In a fully connected network each input port in the network is connected to all possible output ports. This gives a total of $18n_on_i$ wires, when the data is 18 bits, n_i is the number of input ports, and n_o is the number of output ports. Since many of these connections are not used in any feasible configuration they are removed from the design. The result of this process is a subset of a fully connected network, where each input port is only connected to a subset of the output ports as illustrated in figure 5.1.

Since several input ports are connected to the same output port, each output port contains a multiplexor which enables each port to "choose" which input port to receive data from. The

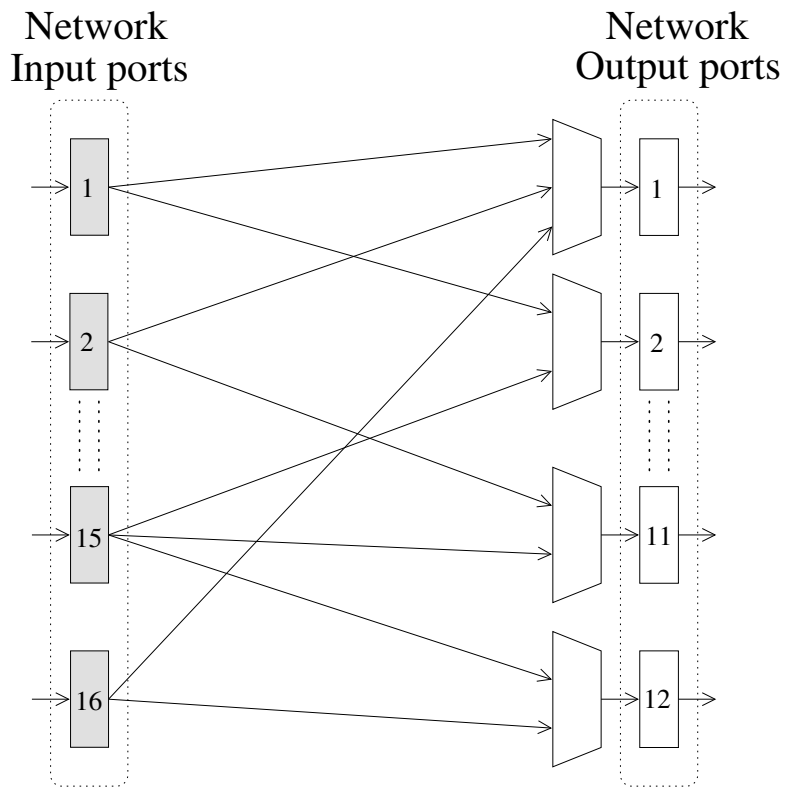


Figure 5.1: The current network implementation is a subset of a fully connected network. Each input port is connected to a subset of the output ports.

multiplexors are controlled by a number of registers which is accessed through a separate configuration bus.

5.3 Multicast

The current network design supports multicasting as each input port is routed to all feasible output ports. This means, that data arrives at all output ports and multicasts are handled by configuring several multiplexors to receive data from the same input port. In theory some of the DSP-blocks in the 'Aphrodite DSP' can send a *packet* to 6 different blocks at a time. It is very unlikely that this many destinations are used at the same time and in the current configurations only 2 destinations are used simultaneously.

5.4 Clocks, dataflow and Lego2 protocol.

Two different clocks are used in the DSP. The *sample clock* is the slowest, controlling the frequency at which new samples are feed into the system and taken out. The *main clock* is running 96 times faster than the *sample clock* and controls the flow of samples through the network and

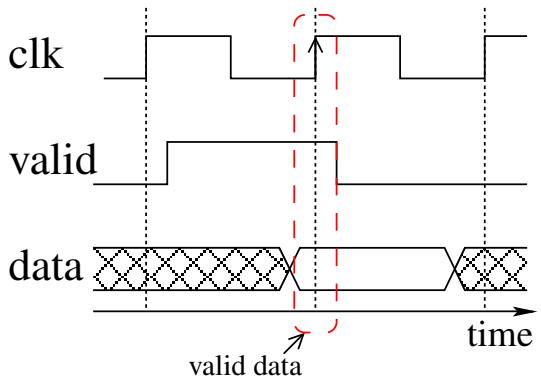


Figure 5.2: The Lego2 protocol which is used to communicate data between the different blocks. The *valid* signal indicates that data is valid at the corresponding positive edge of the clock.

computation inside the individual blocks. The *main clock* is capable of running up to 10 MHz, but is currently configured at 1 MHz. The *sample clock* has a maximal operating speed of 106 KHz when the *main clock* is at its fastest.

Imagine a scenario where data is sampled at the microphones, processed in the Filter Bank and sent to the digital microprocessor. This communication and audio processing takes a given number of *sample* clock cycles. What is important is not the number of cycles, since a small delay in sound is inaudible, but that the number of cycles are constant from sample to sample. If this is not the case noise and clicks will be heard at the receiving party.

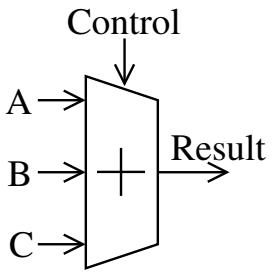
Since the individual blocks have different latencies, and samples are not communicated each *main clock* cycle, the sample is accompanied by an additional *valid* signal to indicate the validity of data. This means that the dataflow is data-driven and a sample can be visualized as a token flowing through the different blocks. The used protocol is denoted the Lego2 protocoland is illustrated in figure 5.2. As the *valid* signal is sampled each clock period it must be high for one clock period only.

5.5 Sample addition

Three of the multiplexors in the current design contain an adder such that several incoming samples can be added. These are called *MUXADDers* to distinguish them from a normal multiplexor. One of these *MUXADDers* can add two samples while the two remaining can add up to three samples.

Besides being able to work as a normal multiplexor the *MUXADDer* can be set up to add several of the incoming signals. In this case the *MUXADDer* waits for data on the inputs which are to be added. That is, it waits for a valid token on the respective inputs. When all data have arrived the samples are added and the result sent to the block.

One could imagine a setup where the user listens to music and an alarm suddenly starts. In this setup a sinusoid is added to the music, even when the alarm is silent. This is controlled by disabling the sinusoid generator such that it does not produce any samples, and thereby not



(a) Illustration of a *MUXADDer* with 3 inputs.

Control	Result
0	A
1	B
2	A+**B**
3	**A**+B+**C**

(b) Configuration table. Bold inputs indicate *master* inputs.

Figure 5.3: Example of a *MUXADDer* which have 4 different functions. Some of these include the addition of the incoming samples.

a valid signal. Since no sample is sent from the sinusoid generator, the *MUXADDer* will stall as it waits for data on this input. In order to avoid this situation some of the inputs to the *MUXADDer* are marked as *masters*. The *MUXADDer* will do the addition as soon as all the *master* signals have arrived which ensures that the stall will not occur.

Figure 5.3 shows the symbol of a *MUXADDer*and an example of its functionality. It should be noted that this example is fictive and is not used in the Aphrodite DSP. Whenever the *MUXADDer* is used to add incoming signals the *master* signals are marked bold in the configuration table.

Chapter 6

Specification of the Network interface

This chapter introduces a new general network interface which allows any network to be inserted and tested. The new network interface is integrated into the 'Aphrodite DSP', which requires some work due to the existence of computational units in the existing network.

6.1 Overview

In order to replace the current network in the Aphrodite DSP with different NoC implementations, a clear and simple network interface must be defined. The term interface is used to denote the protocol at the network input and output ports and how the network is configured. The interface must be designed such that any network can be inserted and configured without changing the environment. When such an interface has been created it is possible to insert, simulate, and synthesize different network implementations and compare them in terms of area and power. Instead of creating a testbench for each designed NoC, the common interface also allows the creation of a common testbench, which makes it possible to verify the network without being integrated in the Aphrodite DSP.

One of the major problems with the existing network is that it contains computational units which do not belong in a network. If the network and computational units are not totally decoupled, some of the advantages of a scalable, reusable NoC are no longer present. It is therefore important that this computation is removed from the network. *William Demant Holding* has been so kind to help implement the computational units outside the network, such that the new network can be integrated into the Aphrodite DSP.

In the following sections, the *MUXADDers* are removed from the network, a protocol at the input and output ports are defined, a universal configuration method is created which is able to configure any inserted network, and the network interface is integrated into the 'Aphrodite DSP'.

6.2 Adders

As explained in chapter 5.5 three of the multiplexors in the current design can add several incoming samples, in addition to the normal multiplex functionality. In order to implement a NoC

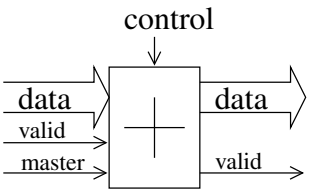
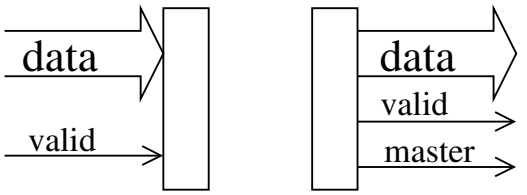


Figure 6.1: The new addition block accumulates samples received on its input port.



(a) Network input port.

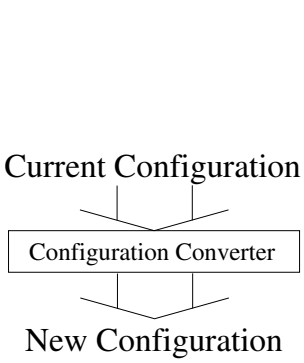
(b) Network output port.

Figure 6.2: Illustrations of the input and output ports to the network.

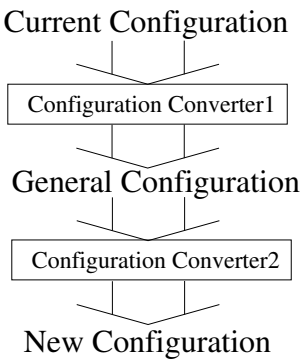
this functionality must be moved away from the network. A straight forward solution would be to create a network output port for each possible operand in the addition. This simplifies the adder implementation, but increases the number of output ports as well as area of the network itself. Instead, a single output port is created which receives samples from all operands as illustrated in figure 6.1. The adder accumulates the received samples until the expected number of samples has been received. Since the adders can be configured to add a different number of samples, a control signal is used to indicate the number of operands in the current addition. As explained in chapter 5.5 some of the samples are denoted as *master* samples. Since the adder receives all samples on the same port, there is no way to distinguish the *master* samples from the ones which are not. One solution would be to create two output ports. One for the *master* samples, and one for the *non-master* samples. Again, this is not a feasible solution due to the area overhead of the network. Instead an extra *master* signal is inserted as shown in figure 6.1. The *master* signal is not constant for a given input port and can even be different for different receivers of a multicast. Therefore, it must be included in the configuration introduced in section 6.4.

6.3 Network ports

The current network uses the Lego2 protocol at both the input ports and output ports. The Lego2 protocol was introduced in section 5.4 and simply consists of 18 data bits and a *valid* signal to indicate the presence of data. The new network input and output ports are illustrated in figure 6.3. As is seen, the network input port is unchanged, while the output ports also contain a *master* signal used in the addition.



(a) The current configuration is converted directly into the new configuration.



(b) The current configuration is first converted into a general configuration which is then converted into the new configuration.

Figure 6.3: The current configuration must be converted into a new configuration. Here 2 possible implementations are shown.

6.4 Configuration

In the current implementation the network is configured by controlling the multiplexors at the network output ports as explained in chapter 5.2. That is, each network output port decides which network input port to receive data from. An example of such a configuration is shown in figure 5.3 on page 27. In the new network it is the network input port who determines where to send the data. Also, the current configuration is only usable in this specific implementation and if an additional output or input port is added to the design, the number of inputs to the multiplexors changes and so do the configuration.

The current configuration could be converted directly to the new configuration as illustrated in figure 6.3a. This means that a configuration converter should be implemented for each new network as they are configured differently. If the original configuration was changed, all configuration converters should be updated to reflect this change.

Instead, the existing configuration is converted into a general configuration as illustrated in figure 6.3b. By creating a general configuration only one converter must be implemented from the existing configuration. It is much easier, and less error prone, to implement a converter from a specific configuration to a general configuration than between two specific ones. Also, a general configuration makes the configuration independent of the Aphrodite Configuration and would make it much easier to port the NoCs to other applications.

The network is configured using a general configuration matrix as illustrated in table 6.1. Each row represents an input port and each column an output port. A '1' means that the input port is sending data to the output port while a '0' means that no data is sent. As explained in chapter 6.2, a *master* signal must be specified for in each *packet*. Therefore a second matrix is created which determines the value of the *master* signal for the specific connection.

The two configuration matrices allow the network to be configured in a general and simple way, it supports multicast, and the *master* signal can be changed for each destination in a

Input \ Output	1	2	....	11	12
1	0	0		0	0
2	0	0		0	0
....					
15	0	0		0	0
16	0	1		0	1

Table 6.1: Configuration matrix. Each row represents an input port and each column represents an output port. In the shown configuration input port 16 sends data to output port 2 and 12.

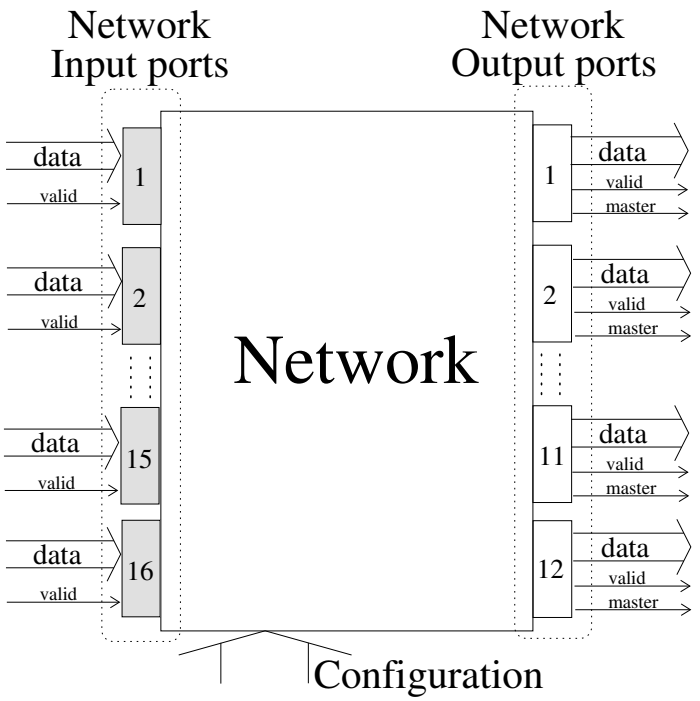


Figure 6.4: Illustration of the new network interface.

multicast. Besides, the configuration can quite easily be translated to the configuration which is used in each specific NoC implementation. If a NoC was to be used in a real application, the configuration would of course be specified directly for the used NoC. The indirect configuration is only meant to create a general configuration which is independent of the inserted NoC.

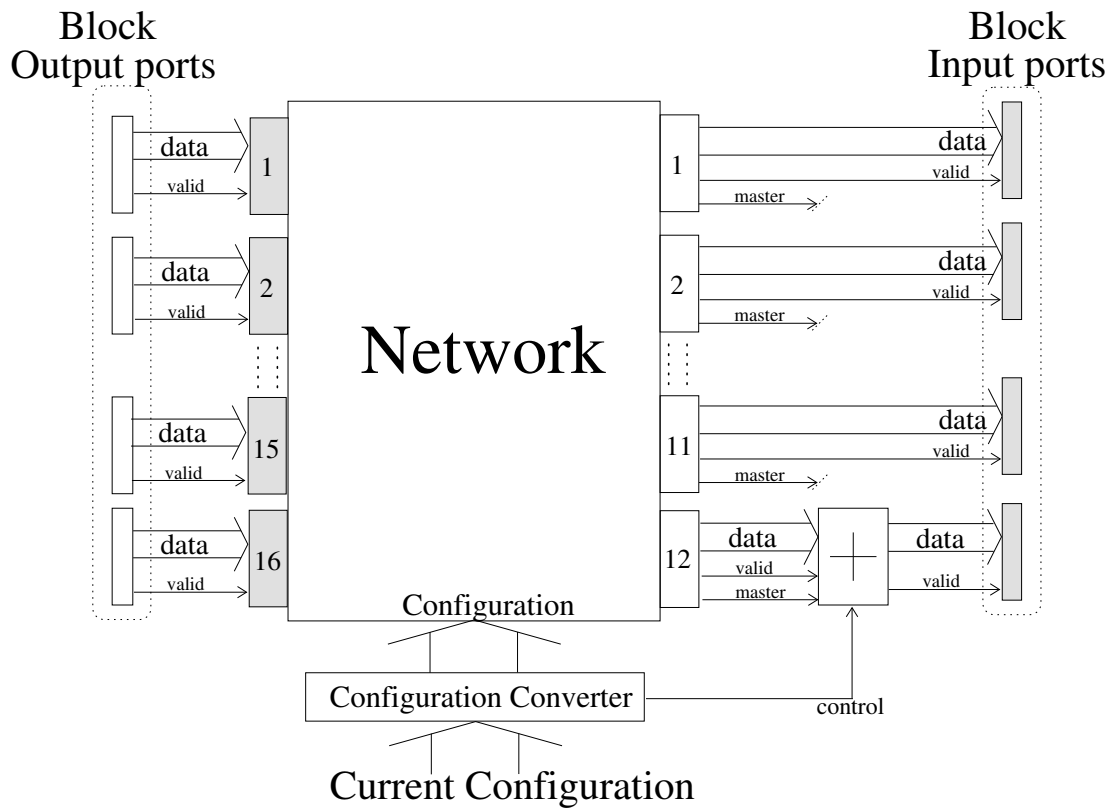


Figure 6.5: Integration of the new network interface into the Aphrodite DSP.

6.5 Final NoC interface

Figure 6.4 shows the final NoC interface. It consists of 16 input ports and 12 output ports using the Lego2 protocol. The input ports are 18 bit wide, while the output ports are 19 bit wide because of the additional *master* signal which is used in the addition. The network is configured by providing the general configuration matrix.

6.6 Integration into Aphrodite

In order to integrate this network interface into the Aphrodite DSP, the functionality of the *MUXADDers* must be implemented and inserted between the current network interface and the new network interface. *William Demant Holding* has implemented these new adder blocks. The *configuration converter* which converts the current configuration into a general configuration matrix and control the signals to the adders has not been implemented. Instead, the configuration is specified directly as a general configuration matrix for the needed configurations. Figure 6.5 shows how the new interface is integrated into the aphrodite DSP. The *master* signal is only used in the adders and is discarded if no adder is inserted at the specific output port.

Chapter 7

Network design

In this chapter the network design is discussed. Design decisions are taken in a number of different areas such as topology, data encoding, and how to handle multicast. Many of the terms used in this chapter was introduced in chapter 2.

7.1 Overview

In chapter 6 a new interface to the network was introduced as illustrated in figure 6.4 on page 31. This chapter discusses the actual network design which is to replace the existing network. Basically, the network can be viewed as a switch with 16 inputs and 12 outputs as illustrated in figure 7.1. Network adapters are inserted at input and output ports to interface between the

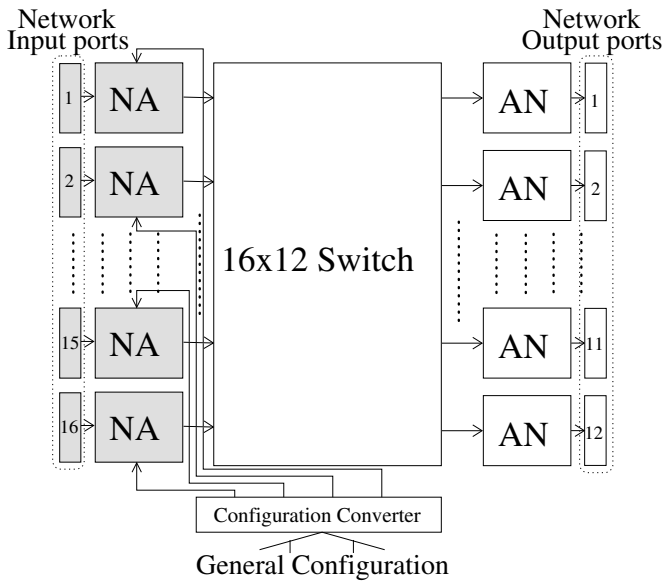


Figure 7.1: The network can be viewed as a switch with 16 inputs and 12 outputs.

	Route	Master	Data	
msb	bit 23-19	bit 18	bit 17-0	lsb

Table 7.1: The basic *packet* format. The route is assumed to be 4 bit but this is not always the case.

ports and the network. The network adapters at the input ports, denoted NA, encapsulate the data received on the input ports in *packets* and handles the sending of these *packets* into the network. In figure 7.1, multicasts are also handled in the NA, network adapters, but this might not be the case, as discussed later. At the output ports the network adapters are denoted AN. Their function are to receive *packets* from the network and forward the data from the *packets* to the output ports in compliance with the Lego2 protocol.
There are several important design decisions to discuss in this chapter:

- Choice of topology.
- Data encoding: Bundled data or delay-insensitive encoding.
- Should the links be wide enough to contain an entire *packet* or should the *packets* be serialized into a stream of *flits*.
- How are multicasts implemented.

Many networks can be implemented having different characteristics in terms of area, bandwidth, latency, power usage, and supply of advanced features. Since the bandwidth need is very low in this application, the networks are designed with focus on low area and power. This also means that no advanced features such as *Guaranteed services* or *virtual channels* are needed,c as they complicate the network circuitry. In other words, the network is kept as simple as possible.

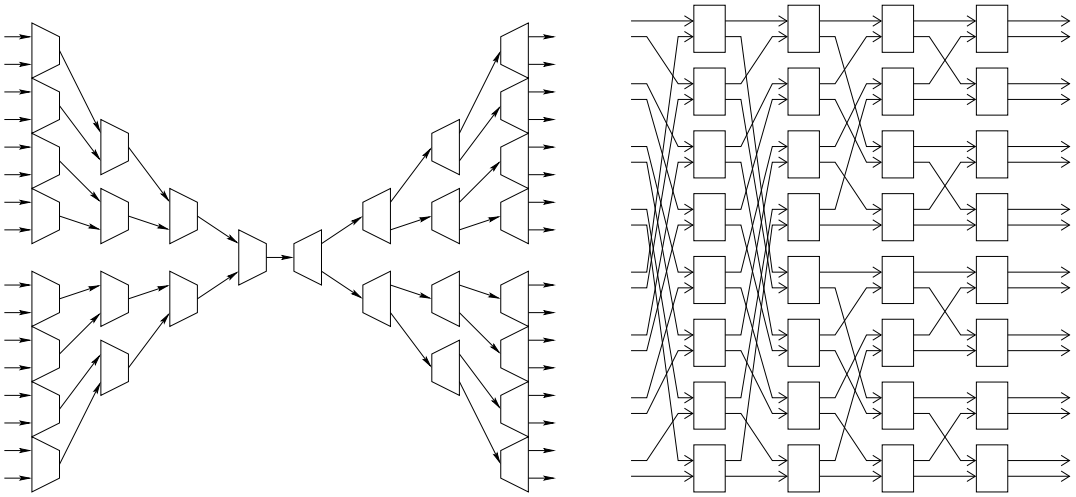
Only *source-routed* networks are considered, where the route is determined by the sender and contained in each *packet*. This is to make the router nodes as simple as possible. Table 7.1 illustrates the basic format of a *packet*. The 19 least significant bits contain the data and *master* bit while the most significant bits determine the route to the destination block. When the *packet* reaches a router node the most significant bit is used to determine the route at this specific router. The entire route is then shifted one bit to the left while the data and *master* bits are kept untouched. The network is implemented using 4-phase handshake protocols since 2-phase protocols are more complicated to implement.

In the following sections the different design decisions are discussed.

7.2 Topology

The topology of the network is very important in terms of area, power dissipation, bandwidth, and latency. The following lists some of the possible topologies:

Crossbar At one extreme, one could make a 16x12 crossbar which is a non-blocking switch having 16 inputs and 12 outputs. In a crossbar, communication between two ports does



(a) Balanced binary tree network.

(b) The Baseline network. An example of an unidirectional multistage 16x16 network.

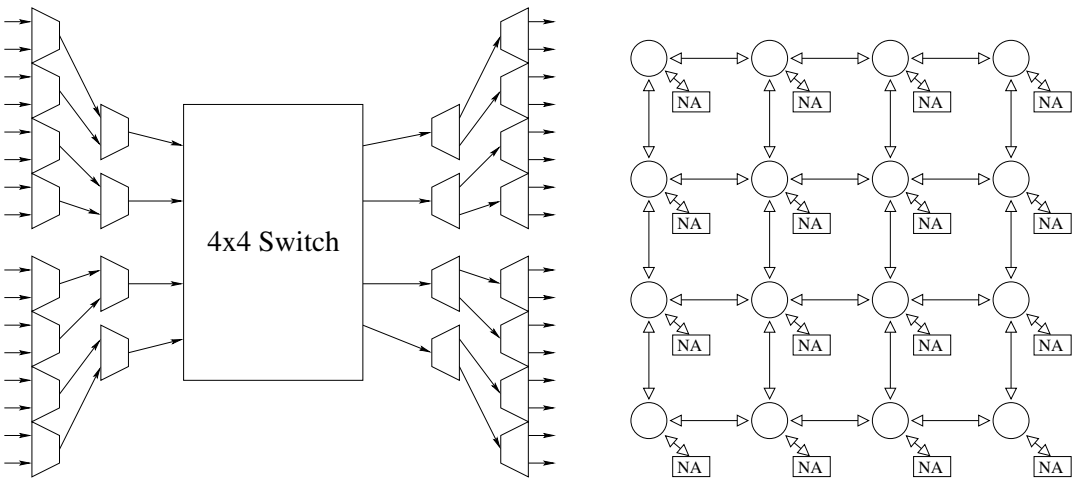
Figure 7.2: 2 examples of network topologies.

not influence the communication between other ports. An example of a crossbar is a fully connected network. It is also possible to restrict the crossbar such that some ports cannot communicate at all, as it is the case for the current network implementation in Aphrodite. Even for a low number of communicating block a crossbar is prohibitive big and is out of the question for this project.

Binary tree: At the other extreme, one could design a binary tree as illustrated in figure 7.2a. The inputs are merged into a single line using a tree of 2 input *merger* blocks and are routed to the outputs using a tree of 2 output *router* blocks. In this topology data always passes through the entire depth of the tree and all communication is *blocking* as it passes through the root of the tree. The root thereby becomes a bottleneck, but this might not be a problem due to the small bandwidth requirements.

Multistage network: Another possible topology is a multistage interconnection network as illustrated in figure 7.2b. A multistage network is constructed using a number of small switches (or crossbars), which are connected in a specific pattern. The illustrated network is called a *baseline* network and consists of 4 stages of each 4, 2x2 switches. As a switch can be implemented using one *merge* block and one *router* block, the latency through the network is the same as for the binary tree. In contrast to the binary tree there is not a single point in the network where all communication must pass. On the other hand the network uses a far larger amount of transistors and wires. It should be noted that this topology is not a crossbar, as there are restrictions on which ports that can communicate in parallel.

General routers: The fourth option is to connect a number of general routers by either uni- or bidirectional links. Figure 7.3b shows an example where the general routers are connected in a mesh structure using bidirectional links. A network adapter, which handles



(a) Hybrid. (b) Mesh.

Figure 7.3: 2 examples of network topologies.

both input and output, is connected to each router node and each router node implements a 5x5 switch or crossbar. This topology is interesting because it is extremely scalable and because there is no central point through which all communication must pass. For example, locality can be exploited by placing blocks that create high traffic loads close to each other. The general router nodes can be connected in many different ways as for example toruses, hypercubes, or a hierarchical structure with increasing bandwidth for central router nodes [11].

This topology takes up a lot of area because of the large router nodes and the needed number of wires. Care must also be taken to avoid deadlock, and techniques such as *virtual channels* might have to be applied which complicate the router nodes even more. Due to the limited bandwidth need and relatively small number of communicating blocks this topology is not relevant for this application.

Hybrids: It is also possible to construct hybrid solutions of the mentioned topologies. One which could be interesting for this application is a 4x4 switch which connects a number of binary trees as shown in figure 7.3a. In this solution there is no longer a single point in the network where all data must pass, thereby allowing parallel communication. This, of course, requires that the 4x4 switch is implemented such that it allows parallel communication as e.g a 4x4 crossbar or a multistage network.

The binary tree topology has been chosen due to the small number of wires and routing circuits. Due to the small bandwidth requirement, there is no reason to employ a more complex topology. Some of the disadvantages with the binary tree are that packets are always passing through the entire depth of the tree and locality of communicating blocks is not exploited. The binary tree is still the best topology for this application, as the number of communicating blocks are so small, that even the smallest hybrid solution would require far to much circuitry.

As the network contains 12 output ports, 4 layers of *routers* are needed. Each routing decision needs 1 bit and a *packet* therefore needs 4 bits for routing. It should be noted that some of the output ports only needs 3 bits for routing.

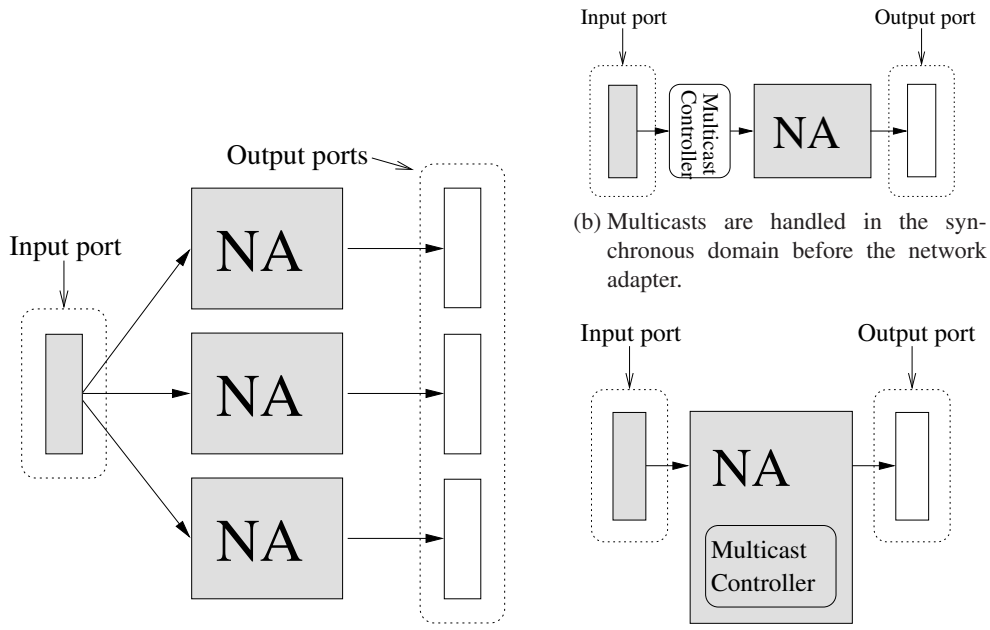
7.3 Data encoding

Both bundled data and one-hot delay-insensitive encoding can be used in the network. If a one-hot encoding is used, the need for matched delays are no longer present. Since the matched delays in a bundled data solution must be conservative in order to ensure validity of data under all operating conditions and process variations, delay-insensitive communication tends to be faster than bundled data. As stated in chapter 3.3.2 and summarized in table 3.1 on page 14, *1-of-4* encoding uses less transitions than *dual-rail* encoding. The problem with *1-of-4* encoding in this context is that each routing decision requires 1 bit, while each *1-of-4* lines encodes 2 bits. This can be solved by making routers with 4 outputs instead of 2, or by re-encoding the *packet* inside the router. These solutions are not used as they complicate the router circuitry and the fanout of the *router* will increase to 4. Instead, each routing decision is encoded into 2 bits which increases the number of route bits to 8. Also, an extra unused bit is appended, such that the *packet* contains an even number of bits. All in all, the size of a *packet* increases from 23 to 28 bits.

Besides choosing how to encode the data, the width of the links must also be decided. Either the links are wide enough to contain an entire *packet*, or the *packet* is be divided into a stream of *flits* and sent using for example *wormhole switching* as discussed in chapter 2.4. This reduces the number of wires and size of the router circuitry. On the other hand the bandwidth is lower, and the *packets* must be serialized into *flits* at the input ports and de-serialized at the output ports.

I have chosen to implement two different data encodings in order to compare them in terms of power and area. The first is a parallel bundled data encoding using 25 wires divided into 19 wires for data (including the *master* signal), 4 wires for routing, and 2 wires for *request* and *acknowledge*. The second is a *1-of-5* delay-insensitive encoding sent as *wormhole switching* using 6 wires divided into 5 wires for data and 1 for *acknowledge*. The reason to use *1-of-5* encoding instead of *1-of-4* is to be found in the article about the 'CHAIN' network [3]. The authors made an experiment where the number of *flits* in a packet was constant, such that the routers simply have to count the number of *flits*. However, experimental results show that this has the disadvantage of complicating the router circuitry and the packet must always contain the same number of *flits*. Instead, a special *end of packet*(EOP) wire is asserted by the last *flit* to indicate that the wormhole can be closed, thereby resetting the *router* and *merge* blocks. This also means that the routers do not need to know the size of the *packet*, and the size could even differ depending on the data payload. I denote this a *1-of-5* encoding. A *packet* is serialized into 15 *flits*, 4 for routing, 10 for data (including the *master* signal) and 1 for EOP.

Since the bandwidth of the bundled data network is much larger than the one using *1-of-5* encoding, the bundled data network is made without any buffers to decrease its size. This means that the network adapters at the input ports perform handshakes directly with the network adapters at the output ports.



(a) Multiple network adapters, where each handles one packet to one destination.

(b) Multicasts are handled in the synchronous domain before the network adapter.

(c) Multicasts are handled by a multicast controller inside the network adapter.

Figure 7.4: 3 different proposals for implementing multicast support.

7.4 Multicast

In the current implementation multicasts are implicitly supported because data arrives at all possible destinations[1]. This is not the case in a *packet-switched* NoC implementation and instead a *packet* must be generated for each destination. The following paragraphs go through different possibilities, adding support for multicasts.

Multiple network adapters A simple solution is to create a network adapter for each possible destination. If a port in the original network can send to 3 different destinations this requires 3 separate network adapters as illustrated in figure 7.4a. The input port is connected to all three network adapters, which each handles the generation of a single *packet*. The individual network adapters can be disabled/enabled by a configuration controller.

The disadvantages of this solution are the sizes of the additional network adapters and that the number of input ports in the network increases significantly. This leads to an increase in network size, power usage, and latency. This increase in network size makes this solution infeasible.

Blocks handles multicast If the connected DSP blocks generate a *packet* for each destination, the network only has to support unicast. This simplifies the network as it does not have to worry about multicasts at all. If the DSP blocks are general purpose processors or similar

[1] Chapter 5.3

devices, they can generate the *packets* in software. The DSP blocks in the 'Aphrodite DSP' are not general purpose processors and the idea in this project is to substitute the current network with a new network having the same functionality. Therefore, this proposal is out of the question.

Multicast before the Network Adapters Instead of letting the blocks handle multicasts a special multicast controller can be inserted between the input ports and the network adapters as illustrated in figure 7.4b. This allows the multicast controller to be implemented in the synchronous domain and the network adapter only needs to handle unicasts

In this solution the networks adapters need a way to tell the multicast controllers that the previous *packet* has been successfully sent. This indication is asynchronous and must be synchronized in the multicast controller which takes at least 4 clock cycles if a 4-phase protocol is used[2]. Even though this might not be a problem this solution has not been chosen because of the latency.

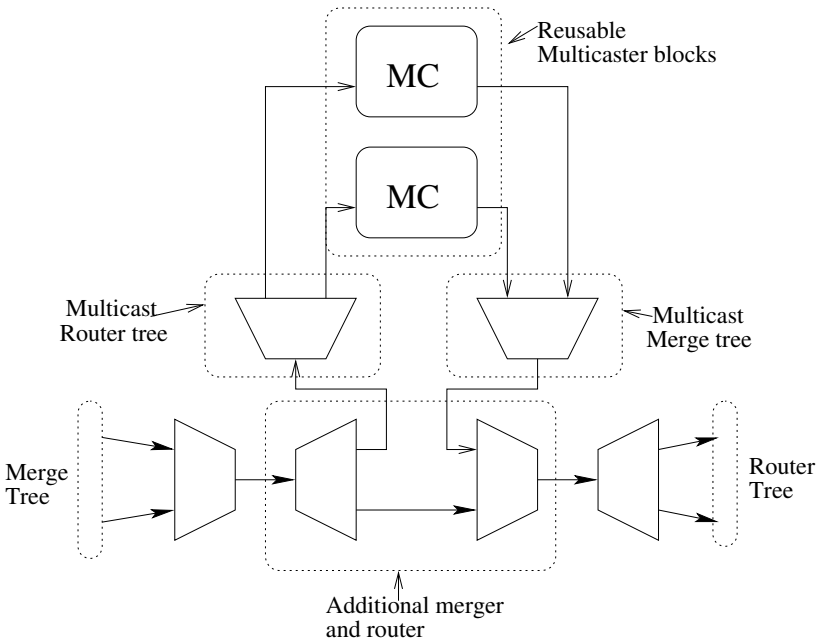
Multicast in the Network Adapters The route of a *packet* is determined in the network adapter and another possibility is to implement the multicast functionality inside the network adapter. Figure 7.4c illustrates how the network adapter contains a multicast controller.

When multicasts are handled in the network adapter, the multicast circuitry can be made both in the synchronous domain or the asynchronous domain. A synchronous solution is analogue to placing the multicast controller before the network adapter and has already been considered. By implementing the multicast controller in the asynchronous domain, no synchronization is necessary between the *packets*. This also makes it possible to send several *packets* within the same clock-cycle.

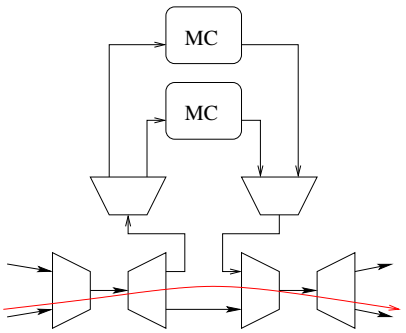
Reusable Multicast blocks In the solutions mentioned so far, one of the major disadvantages is that the number of possible destinations must be known when doing the synthesis and that the multicast circuitry must be included in all network adapters which is foreseen to do multicast. Instead of implementing multicast in the network adapters, a shared multicast block is created. This multicast block receives a single *packet* on its input and sends two *packets* on its output. If a block wants to multicast it does not send the *packet* directly to the destinations. Instead it does an indirect multicast by sending a single *packet* to one of the shared multicast blocks, which handle the actual multicast. A number of these shared multicast blocks are instantiated according to the needs in the specific application. Since the multicast blocks are shared, only a subset of the number of multicast blocks are needed and the complexity of the network adapters decreases as they only need to support unicast. The needed number of multicast blocks are the largest number of simultaneously multicasts for all input ports. In this application at most 2 input port are multicasting at the same time.

One question arises when talking about multicast blocks: Where are they to be placed in the network? The placement of the multicast blocks affects both the area and dynamic

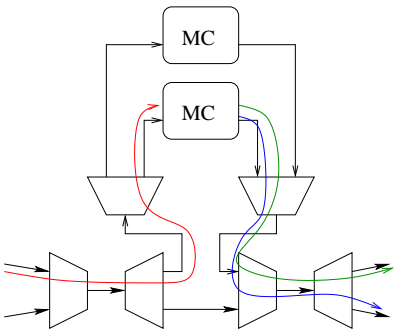
[2]Appendix A goes into detail about synchronization between two different domains



(a) An additional router and merger are inserted to control access to the multicast network which consists of a router tree, multicast blocks and a merger tree.



(b) Example of a unicast.



(c) Example of a multicast. A single packet is sent to one of the multicast blocks which generates two new packets.

Figure 7.5: The reusable multicast blocks are placed at the root of the tree to decrease the latency of multicasts. This example only contains two multicast blocks, but any number of multicast blocks could be inserted.

power consumption. In this project a binary tree topology is implemented and one solution is to increase the size of the network with one additional input and output for each multicast block.

Another solution is to place the multicast blocks near the root of the tree, thereby decreasing the latency of multicasts. Figure 7.5a illustrates a possible implementation of this. An additional *router* and *merge* block are inserted independent of the number of multicast

blocks. Figure 7.5b illustrates a unicast which does not use the multicast blocks, while figure 7.5c illustrates a multicast.

If general a router topology is used, the multicast blocks could be distributed across the router nodes or placed at special router nodes which was reserved to serve multicast.

Two different multicast implementations are compared. In the first, multicasts are handled in the network adapters while the second solution handles multicasts in two shared multicast blocks.

7.5 Summary

Three different *packet-switched*, *source-routed* networks will be implemented and compared. All are using a binary tree topology, which is constructed by a number of binary *merge* and *router* network blocks. The networks have 16 input ports and 12 output ports which means that each *packet* has to be routed at most 4 times. All asynchronous circuits implemented using a 4-phase handshake protocol.

The bundled data networks are transparent to handshakes while the *1-of-5* network are pipelined.

The networks differ in their data encoding, their link width, and the way multicasts are handled.

NoC1 is using a bundled data encoding and the link width is 25 bits including *request* and *acknowledge* wires. Multicasts are implemented in the network adapters at the network input ports.

NoC2 is using a bundled data encoding and the link width is 25 bit including *request* and *acknowledge* wires. Multicasts are implemented as two reusable multicast blocks which are placed in the root of the binary tree.

NoC3 is using *1-of-5* encoding and the link width is 6 bits including an *acknowledge* wire. Multicasts are implemented in the network adapters at the network input ports.

Chapter 8

Implementation

This chapter goes through the actual implementation of the networks.

8.1 Overview

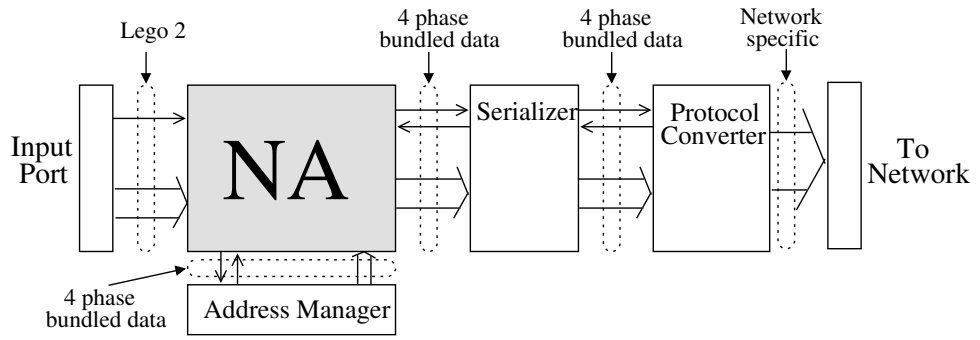
In order to design three networks and avoid code redundancy, a hierarchical design structure is employed. By creating a number of small Network Building Blocks (NBBs) many kinds of networks can be created by instantiating and connecting a number of NBBs . Many NBB's use other NBB's to implement their functionality and some even recursively instantiate themselves. The NBBs are designed as templates, which allow for example the width of a bundled data channel or the number of inputs and outputs to be specified for each instance.

In order to further decrease the redundancy, a "Common network platform" is created. It consists of a number of NBBs which can be used in all networks, independent of the data encoding and network topology. The "common network platform" contains network adapters, which converts between the synchronous and asynchronous domains, and blocks, which serialize a *packet* into a stream of *flits* and vise versa.

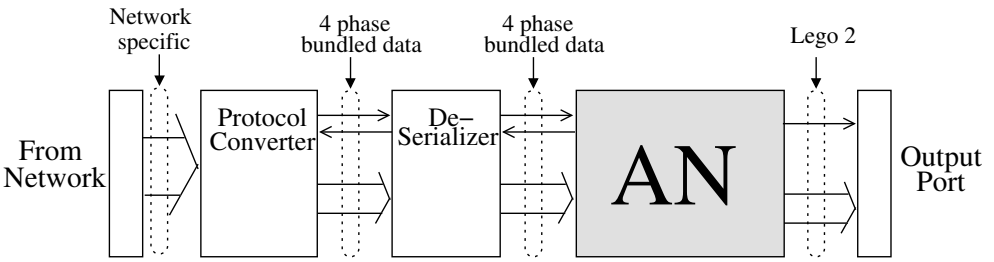
The implementation is done using the 'Verilog' Hardware Description Language (HDL) which is widely used within the IC industry. Since 'Verilog' is also used at *William Demant Holding* it is an obvious choice. Many of the NBBs are made as templates by using the 'parameter' statement in 'Verilog', which can be specified for each instance of a module. Inside the modules, 'generate' statements are used to utilize the specified parameters to change the creation and behavior of the module. 'generates' are used to make structural code and is included in the 'Verilog' 2001 standard. It is of course required that the used tools support 'generates' which is not always the case.

The NBBs are implemented by instantiating cells from a virtual cell library, such that the implementation is independent of the used standard cell library. The virtual cell library also inserts delay in the behaviorial version of the cells. Appendix B gives a short introduction to the cell library.

In the following sections the 'common network platform' is introduced and the NBBs are implemented. Then, specific network blocks for the 2 data encodings are implemented and at last the three networks are designed using the introduced NBBs.



(a) The data is encapsulated in a single *packet*, serialized into a stream of *flits* and converted to the proper protocol before sent into the actual network.



(b) *flits* are received from the network and converted into 4-phase bundled data, de-serialized into a single *packet* and converted to the Lego2 protocol.

Figure 8.1: The "Common network platform" is used by all networks to connect the input and output ports to the network.

Appendix D contains a complete list of NBBs,including their gate-level implementation, while appendix E contains all 'Verilog' code which can also be found on the attached CD-ROM.

8.2 Common network platform

The networks are designed using a "common network platform" which consists of a number of standard NBBs that can be used by all networks. The NBBs within the "common network platform" are used to convert from the Lego2 protocolto an asynchronous *packet* and the other way around.

Figure 8.1a illustrates how the NA, network adapter is connected to the network input port and the actual network such it can be used in all network implementations. It accepts data using the Lego2 protocol and creates a number of *packets*, which are sent into the network. The *Address Manager* block is connected to the network adapter, such that the same network adapter can be used for both unicast and multicast. The output port, which is connected to the actual network, uses a 4-phase bundled data protocol where the entire *packet* is sent in parallel. An optional *seralizer* block, which serializes the *packet* into *flits*, can be inserted if needed. At last, a *protocol converter* is used to convert from 4-phase bundled data to the protocol which is used in the actual network. This structure makes the *network adapter* and *serializer* reusable for all

network implementations. The *protocol converter* is not considered as a part of the "common network platform" as it is specific for each data encoding.

A similar construct is used to receive *packets* from the network and output the data to the network output port. This is illustrated in figure 8.1b. First, an optional *protocol converter* converts from the protocol used inside the actual network to a 4-phase bundled data protocol, if these are not the same. If the *packet* is sent using several *flits*, a *de-serializer* block is inserted to convert the *flits* into a single *packet*, before it is connected to the AN, network adapter which outputs the data using the Lego2 protocol. Both the AN, network adapter and *de-serializer* are reusable for all network implementations.

In the following subsections, the implementation of the network blocks which are part of the "common network platform" are implemented.

8.2.1 NA, Network Adapter

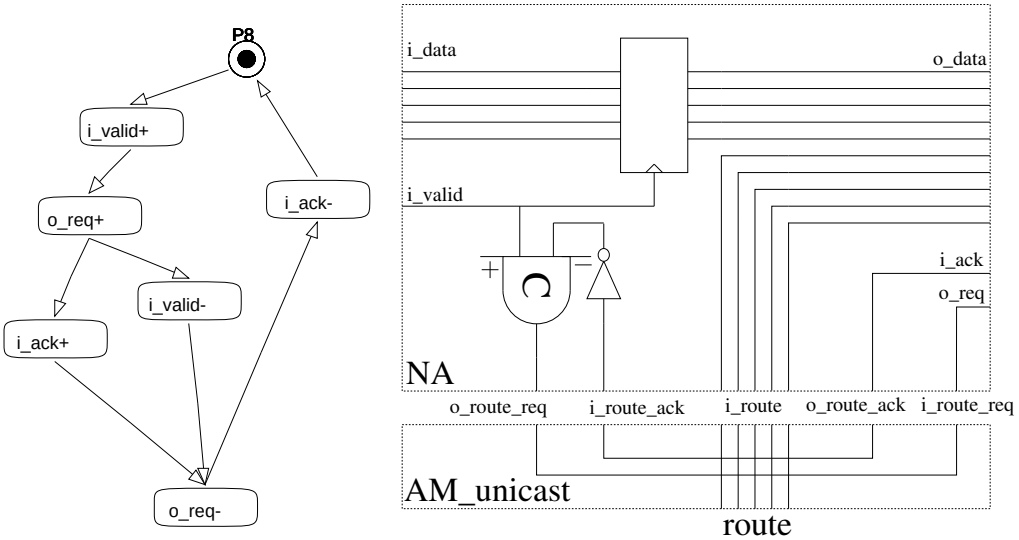
The NA, network adapter receives data using the Lego2 protocol, encapsulates the data in a *packet*, and sends the *packet* into the network using a 4-phase bundled data protocol. As the Lego2 protocol does not contain an *acknowledge* wire, there is no flow control at the input port. This means that the network adapter does not have any means to indicate that it is not ready to receive data. Therefore, it must always be able to receive data. If this are not the case, data might be lost. In this application it is assumed, that the delay between succeeding data to the network adapter is large enough for the network adapter to handle the sending of a *packet*. This is fulfilled because the DSP blocks communicate at most one sample each *sample period*, as it was explained in chapter 5. If this is not the case, buffers must be inserted such that no data is lost.

Figure 8.2a shows an STG which captures the wanted behavior of the NA, network adapter. 1) `i_valid` goes high which indicates that data has arrived at the input port. 2) `o_req` is asserted to send a *packet* 3) The *packet* is acknowledged by `i_ack` and, at some point, the environment lowers `i_valid`. (In parallel) 4) `o_req` is driven low and when `o_ack` goes low the cycle is complete.

Note that `o_req` is not lowered until `i_valid` has gone low. This means that the outgoing handshake is coupled with the `i_valid` signal. I was not able to design a simple STG which allows `i_valid` to go low at any point of time. Petrify needs some timing assumptions that I do not know how to provide. It is possible to design an STG which decouples the handshake from the `i_valid` signal, but the produced circuit was relatively large and is not needed in this application. A de-coupled handshake controller [9] could also be inserted between the generated handshake controller and the outputs.

Figure 8.2 shows the gate-level implementation of the NA, network adapter. As it is seen, the generated handshake is not sent directly to the output port, but it instead sent to a so called *Address Manager*. The idea is, that the same network adapter should be used for both uni- and multicasts and that the *Address Manager* handles the handshaking and generation of routes. The *Address Manager* shown in figure 8.2 handles a unicasts by connecting the in-going handshake with the out-going handshake and supplying a single route. The AM_multicast, which handles multicasting, can be found in appendix D.1.1.

Data is saved in a D flip-flop on positive edges of the `i_valid` signal. It would have used



(a) STG for which Petrify outputs the boolean expression: o_req = i_valid+ o_req i_ack‘.

(b) Gate-level implementation.

Figure 8.2: Implementation of the NA, network adapter.

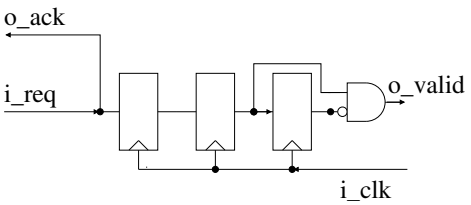
less area if a level-sensitive latch could have been used instead, but this is not possible because data is only valid on the rising edge of the `i_valid` wire.

8.2.2 AN, Network adapter

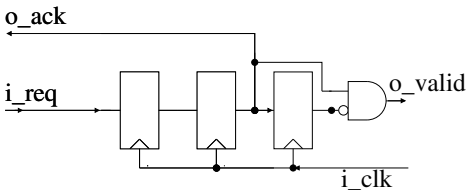
The AN, network adapter receives *packets* from the network using a 4-phase bundled data protocol and outputs data using the Lego2 protocol. Since the direction of data is from the asynchronous to the synchronous domain, the Lego2 protocol must be synchronized using the clock signal from the block to which it is connected. When data is transferred from one clock domain to another, or from an asynchronous to a synchronous domain, safe synchronization must be applied. Appendix A explains the basics of such synchronization.

Figure 8.3 illustrates 4 different ways to synchronize from the asynchronous domain to the Lego2 protocol. The latch which stores the data is not shown, but must be included in the actual implementation. Note that the signal after the first flip-flop is never used, because it can be in a state of metastability and thereby create hazards.

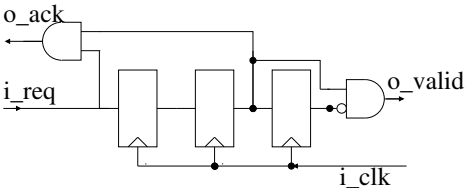
Figure 8.3a illustrates a solution which will fail, because the handshake can complete within a single clock cycle. If this is the case, the synchronous part will never see the data and this solution is not to be used. At the other extreme, figure 8.3b shows the classic two-flop synchronizer which takes at least 4 clock-cycles as the handshake waits for the *request* signal to be synchronized on both its rising and falling edge. The solution in figure 8.3c improves this by completing the handshake before the synchronization of the falling transition of *request*. This means that the handshake finishes much faster, but this solution will not work, if a new handshake starts



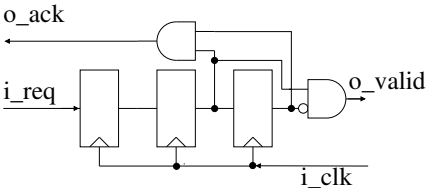
(a) This solution will fail if the asynchronous part is to fast. It is extremely dangerous and should be avoided.



(b) Fool proof two-flop solution which takes at least 4 clock-cycles to complete.



(c) Solution where only the first part of the handshake is synchronized. Will fail if another handshake starts before the entire synchronization is done.



(d) Solution which takes at least 5 cycles but avoid a latch in the data path because the data is first acknowledges after the pulse on `o_valid`.

Figure 8.3: 4 different implementations of the asynchronous to Lego2 protocol synchronization. The last flip-flop and the AND gate with inverted input is added to make sure that `o_valid` is only high for one clock-cycle. A latch for storing the data is not shown on the figures.

before the previous handshake has been completely synchronized. As some of the ports in the 'Aphrodite DSP' can receive more than one *packet* each *sample period*, this solution is not a possibility for this application. The solution in figure 8.3d avoids the need of a data-latch, because the data is first acknowledged after the data has been sent using the Lego2 protocol. The penalty is one extra clock-cycle for the synchronization of the rising *request* signal.

In summary, only two of the four solutions are usable. The standard two-flop synchronizer in figure 8.3b is used because it completes the handshake in 4 clock-cycles. In order to decouple the handshake between the network and the synchronization to the Lego2 protocol, buffers can be inserted between the network and the network adapter.

As the clock frequency in Aphrodite is at most 10 Mhz, it would also be possible to clock the first register by the negative clock edge instead of the positive clock edge. This would decrease the number of clock periods needed for the synchronization, but I have not investigated this option further and one always have to be careful when playing around with the clock and synchronization.

The final gate-level implementation of the NA, network adapter is included in figure 8.4. A flip-flop is used as state-holding device even though a level-sensitive latch would suffice. This is because there was problems during the integration into 'Aphrodite' when the data returned to zero after the handshake has completed.

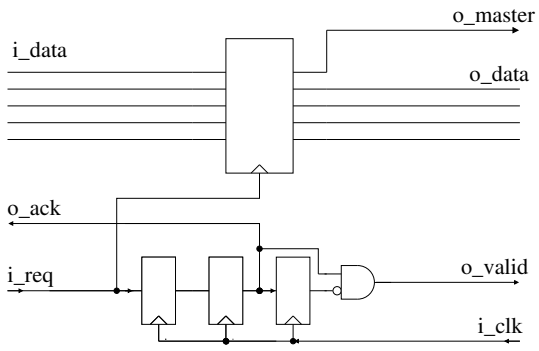


Figure 8.4: Gate-level implementation of the AN, network adapter.

8.2.3 Serializer

This network block serializes a *packet* into a stream of 2 bit *flits*. Both the input and output use a 4-phase bundled data protocol. After the last *flit* has been sent, a special "End Of Packet" (EOP) wire is asserted to indicate that there is no more *flits* in the *packet*. The EOP wire works like the request wire and a 4-phase handshake must be performed.

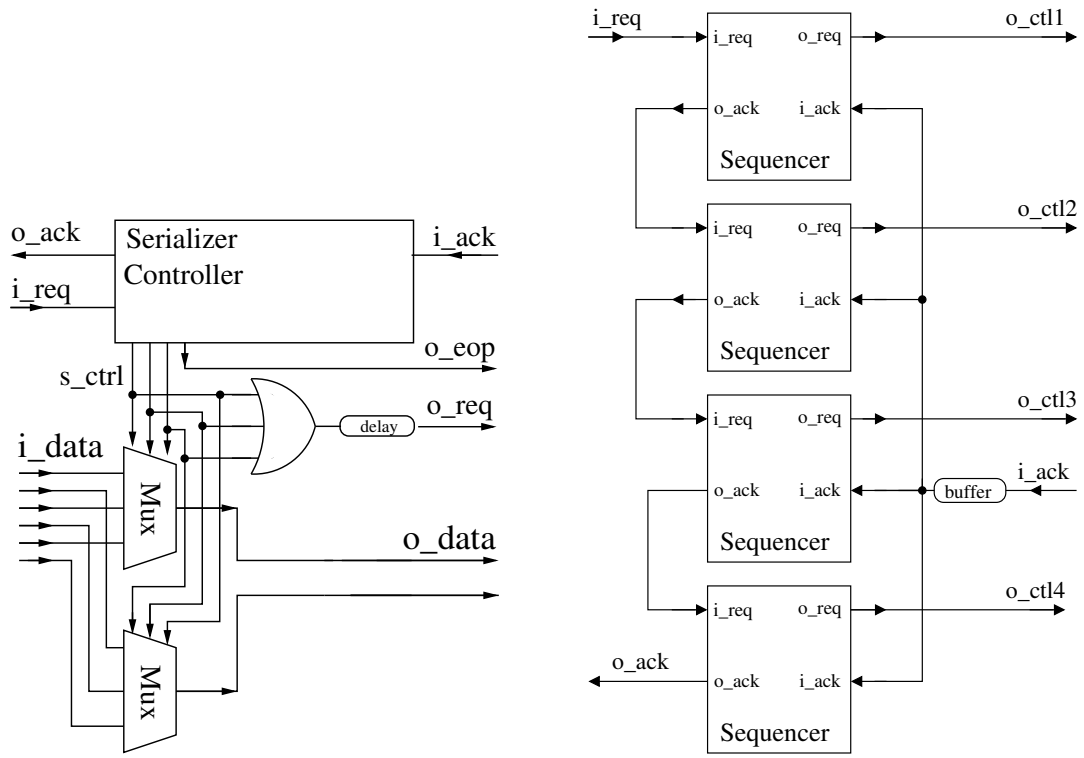
The *serializer* can be implemented in many ways with different speed, area and power characteristics. An obvious possibility is to employ a shift register but this would consume a lot of unnecessary power and is not considered an option.

Instead, the bits are selected 2 at a time using multiplexors as illustrated in figure 8.5a. The block is hard-coded to output *flits* of 2 bits but this could very well have been selectable by a 'parameter'. The 'brain' of the *serializer* is the controller which handles all handshakes and generates control signals for the two multiplexors. The control signals also act as the outgoing request signal. The request is generated by OR'ing the control signals. A matched delay is inserted on the *request* wire such that the data is stable before the *request* wire is asserted. Note that the controller is instantiated to perform one more handshake than the number of data *flits*. The last control wire is forwarded as `o_eop` to perform the EOP handshake.
The functionality of the controller is as follows:
1) `i_req` goes high to indicate that new data has arrived at the input. 2a) One of the control signals is asserted. This is used to control the multiplexors and generate a request to the succeeding stage. 2b) The succeeding stage acknowledges the input 2c) The control signal is lowered 2d) The succeeding stage lowers `i_ack` 3) Step 2 is repeated till all data has been send. In this case 4 *flits* are sent. 3 for data and 1 for EOP. 4) The 4-phase handshake to the preceding stage which started the conversion is completed.

The controller can be implemented in many different ways:

- The entire controller can be specified as an STG which is made into a speed-independent asynchronous circuit using Petrify. The STG can be auto-generated by a script depending on the number of *flits*.

- The controller can be designed as an ordinary synchronous state-machine. After the circuits which determines the next state and output have been synthesized it must be turned



(a) Overview of the *serializer* network block.

(b) Implementation of the *serializer controller*.

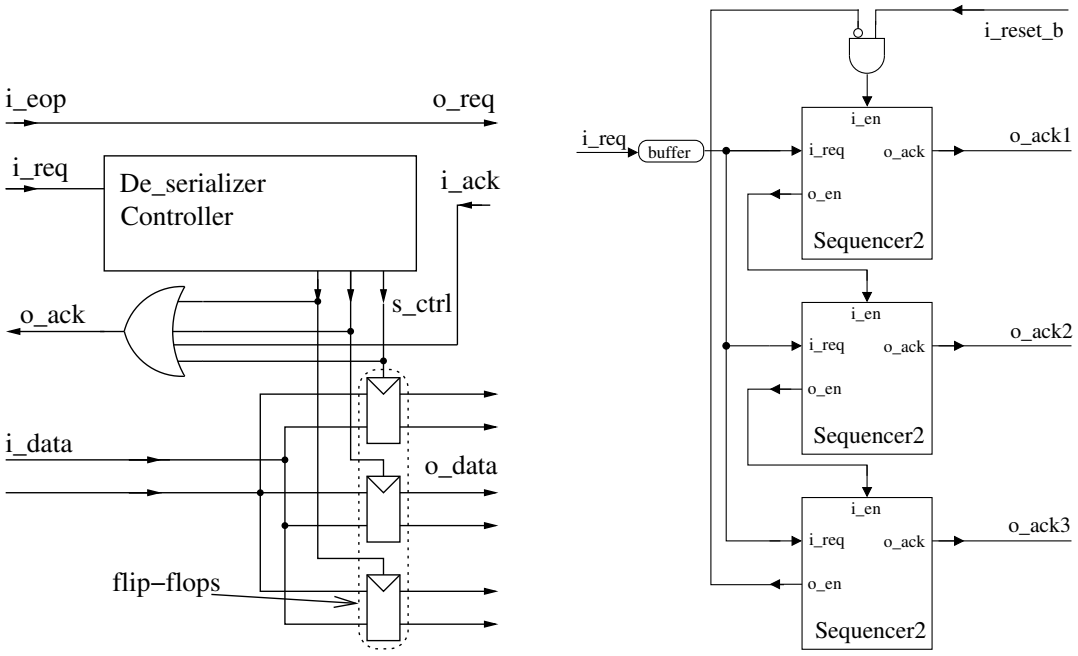
Figure 8.5: Implementation of the *serializer* block which sends 3 *flits* of each 2 bit and an EOP *flit*. Note that the controller is using 4 *sequencers* because the EOP *flit* must be generated and acknowledged.

into an asynchronous state-machine by inserting matched delays. It should be noted, that this option has not been investigated thoroughly.

- The controller can be decomposed into smaller blocks which each handle one handshake and the setup of one control wire. The needed blocks can be designed as STG's and realized using Petrify.

The first two options need to be re-implemented each time the number of *flits* changes, which is not the case for the third option. Because the design is decomposed into smaller circuits, the circuits are also easier to design and implement.

Figure 8.5b shows an implementation of the controller which can handle 4 handshakes. 3 handshakes for data *flits* and one for the EOP handshake. The controller is constructed by connecting 4 simple *Sequencer* blocks which each carries out a single handshake. The *Sequencer* block was designed in chapter 4.4 and basically accepts a handshake on the left hand side, generates a handshake on the right hand side, and completes the handshake on the left hand side. In addition to this functionality, the `i_ack` wire can alternate when the *sequencer* is not involved in an outgoing handshake. This is needed because the same acknowledge wire is connected to



(a) Overview of the *de-serializer* network block.

(b) Implementation of the *de-serializer controller*.

Figure 8.6: Implementation of the *de-serializer* which handled 3 *flits*.

all the *sequencers*. When a *sequencer* has completed the handshake on its right hand side it acknowledges the handshake on the left hand side which starts the succeeding *sequencer*.

The big advantage of this construction is that it is very easy to design controllers which handle a different number of *flits*. Only the buffer which is inserted such that the incoming acknowledge can drive all *Sequencers* is dependent on the number of *Sequencers*. Note that the controller is instantiated one stage larger than the number of *flits* such that the last control signal can be used as *EOP*.

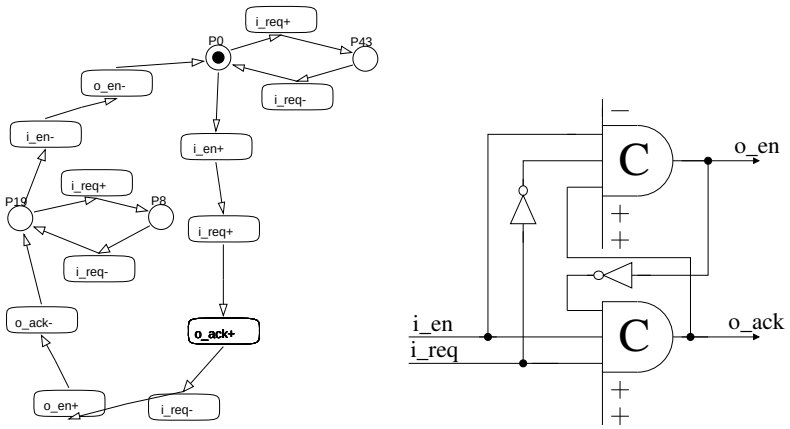
If the latency of the *serializer* turns out to slow down the sending of data, the serialization can be divided into a number of pipeline stages to improve the latency of each stage. This is only an advantage if the succeeding blocks are able to receive data fast enough.

8.2.4 De-serializer

This network block de-serializes a stream of 2 bit *flits* into a single data value. Both input and output uses a 4-phase bundled data protocol. After the last *flit* has been received, a special "End Of Packet"(*EOP*) wire is asserted to indicate that there are no more *flits* in the *packet*.

The *de-serializer* is very similar to the *serializer* and the implementation suggestions and comments made in the previous subsection applies for the *de-serializer* as well. Again, a shift register is avoided due to the unnecessary power consumption.

The chosen solution is illustrated in figure 8.6a. The block is divided into a controller which handles all handshakes and control signals to a number of latches. The 2 incoming data wires



(a) STG describing the behavior. (b) Gate-level implementation.

Figure 8.7: Implementation of the *sequencer2*.

are connected to 3 flip-flops which are controlled by individual signals from the controller. The control signals also act as acknowledge to the preceding stage why they are OR'ed. The basic idea is that one of the flip-flop control signals is asserted when a *flit* arrives. This makes sure that one of the flip-flops stores the data, while the others stay unchanged. A solution using latches was also tried out, but the complexity and size of the controller increased. This is because the latches must be in *opaque* mode except when they are receiving data, thus a pulse must be made independent of the *acknowledge*.

The controller are implemented in a similar way to the controller in the *serializer*. A small block, denoted *sequencer2*, handles one handshake and controls one flip-flop. A number of these are instantiated and connected as illustrated in figure 8.6b which makes it very easy to construct controllers of different size. In the initial state all inputs, outputs, and internal wires are '0' except for the `i_en` input to the first *sequencer2*. This means that the first *Sequencer2* is enabled while the rest are disabled. When `i_req` makes a rising transition the first *sequencer2* performs a 4-phase handshake using `i_req` and `o_ack1` before it asserts `o_en` which enables the next *sequencer2*. In this fashion the *sequencer2* blocks perform a handshake one by one. When the last *sequener2* is done, the feedback resets the construct and the cycle is complete. This construction assumes that the number of *flits* is constant for all *packets*.

Figure 8.7 shows an STG capturing the behavior of the *sequencer2* as well as its gate-level implementation. It has many similarities to the *sequencer* STG which is used as an example in chapter 4.4. The sequence of events are as follows: 1) `i_req` can make a number of transitions if other controllers are handshaking, 2) `i_en` goes high to indicate that the controller is activated, 3) when `i_req` goes high a 4-phase handshake is completed using `o_ack` and `i_req`, and the next controller is activated by rising `o_en`, 4) `i_req` can make a number of transitions if other controllers are handshaking, 5) when `i_en` is lowered `o_en` is set to '0' which completes the cycle.

8.3 Specific network blocks

Both bundled data and *1-of-5* delay-insensitive encoding are used in the network implementations. In order to implement a network using a specific encoding, a *merge* and *router* block must be implemented. The *merge* block merges two input port onto a single output port. As the input ports are not mutual exclusive, the *merge* blocks must contain some sort of arbitration. The *router* block receives a *packet* ont its input port and routes it to one of its output ports depending on the route. The route is contained in the *packet*.

Besides these two elementary blocks, a number of behavioral blocks are created for use in the testbenches. For example a *source* which sends data, and a *sink* which receives and acknowledges data.

8.3.1 Bundled data network blocks

The 4-phase bundled data blocks do not contain any buffers or latches and are transparent to handshakes. This means, that the network adapters at the input and output port are performing a handshake directly with each other. Gate-level implementations of all bundled data blocks can be found in appendix D.2.

Merge block (Appendix D.2.1)
: The *merge* block consists of a handshake arbiter and a multiplexor[1]. The handshake arbiter grants one of the inputs access to the output port and locks the arbiter until the handshake is complete. The multiplexor is implemented using an complex AND-OR gate.

Router block (Appendix D.2.5)
: As explained in chapter 7.1, the most significant bit is used to determine the route of the *packet* and the route is shifted left by one. The *packet* is sent to one of the output ports depending on the route. A new route is first accepted when the handshake cycle is complete. Note, that AND gates are inserted such that data are only sent to one of the output ports. The data could safely be routed to both output ports since only one of the port receives a handshake. This would cause the data to shift through the entire network and would contribute heavily to the power consumption.

8.3.2 *1-of-5* network blocks

The *1-of-5* blocks encode 2 bits of data into 4 wires, while it uses a fifth wire to indicate the End-Of-Packet(EOP) as explained in chapter 7.3. The *router* and *merge* blocks are almost an exact implementation from the 'Chain' network [3]. Gate-level implementations of all *1-of-5* network blocks can be found in appendix D.3.

Merge block (Appendix D.3.4)
: When one of the input ports are granted access to the output port, a controller blocks the other input port until an EOP *flit* has been received and acknowledged.

[1]The arbiter is implemented is chapter 5.8.2 in [13].

Destination	1	2	3	4	5	6	7	8	9	10	11	12
Enable	0	0	**1**	0	0	0	0	0	0	**1**	0	0

Table 8.1: Configuration vector which represents a multicast to destination 3 and 10.

Router block (Appendix D.3.7)

When the first *flit* arrives, a controller determines the route of the packet and locks the router. This means, that succeeding *flits* are routed to the same output port. The controller is reset when the EOP *flit* is received.

In order to incorporate the *1-of-5* blocks into the 'common network platform', two protocol converters are implemented. *PC_bundled_1of4* converts 2 bit of bundled data into a *1-of-4* delay-insensitive encoding while *PC_1of4_bundled* converts the other way around.

8.4 The networks

The 'Common Network platform' and network blocks for the 2 encodings have now been presented and it is time to construct the 3 network solutions. The network design was discussed in chapter 7 and the design decisions summarized in chapter 7.5.

An overview of the three network implementations is presented in the following subsections. In order to configure the networks, the general configuration matrix presented in chapter 6.4 must be converted into a local configuration for each network. This conversion is discussed for each of the three networks.

8.4.1 NoC1: Bundled data, multicast in NA

This network uses a 4-phase bundled data protocol through the entire network and handles multicasts in the network adapter. Figure 8.8 illustrates how the network is constructed using the network building blocks. The 'Verilog' code can be found in appendix E.2.3.

A configuration converter is instantiated inside each *AM_multicast* block, as multicasts are handled in each NA, network adapter. The converter takes a vector from the general configuration matrix as input. This vector describes which output ports the *packet* must be sent to. The format of the input vector is illustrated in table 8.1 where each bit represents a possible output port. The converter converts the vector into a number of routes and enable signals which are used by the *Multicaster* block. If an input port is only sending to a single destination, only a single route is configured and enabled. The code for the converter is included in appendix E.2.1.

8.4.2 NoC2: Bundled data, shared multicast blocks

This network uses a 4-phase bundled data protocol through the entire network and handle multicasts using two shared multicast blocks. Figure 8.9 illustrates how the network is constructed using the network building blocks. The 'Verilog' code can be found in appendix E.2.4.

The configuration converter must specify a single route for each NA, network adapter and two routes for each multicast block. This is done by counting the number of destinations for each input port and reserving one of the shared multicast blocks if a multicast is required.

1 destination Assign destination route directly to the *AN_unicast* block.

2 destinations Reserve a shared multicast block, assign the route of the multicast block to the *AN_unicast* block and assign the 2 destination routes to the multicast block.

The behavioral implementation of the converter is included in appendix E.2.2.

8.4.3 NoC3: *1of5* encoding, multicast in NA

In this network data is sent as a stream of 2 bit *flits* using a *1-of-5* delay-independent data encoding. Multicasts are handled at the NA, network adapters. Figure 8.10 illustrates how the network is constructed using the network building blocks. The 'Verilog' code can be found in appendix E.2.5.

The configuration converter developed for NoC1 is also used for this network as the topology and routing decisions are the same. The 4 route bits must be divided into 4 *flits* as an entire *flit* is used for each routing decision. As each *flit* encodes 2 bits, this is done in the *NoC_S1* block by appending a '0' to each route bit.

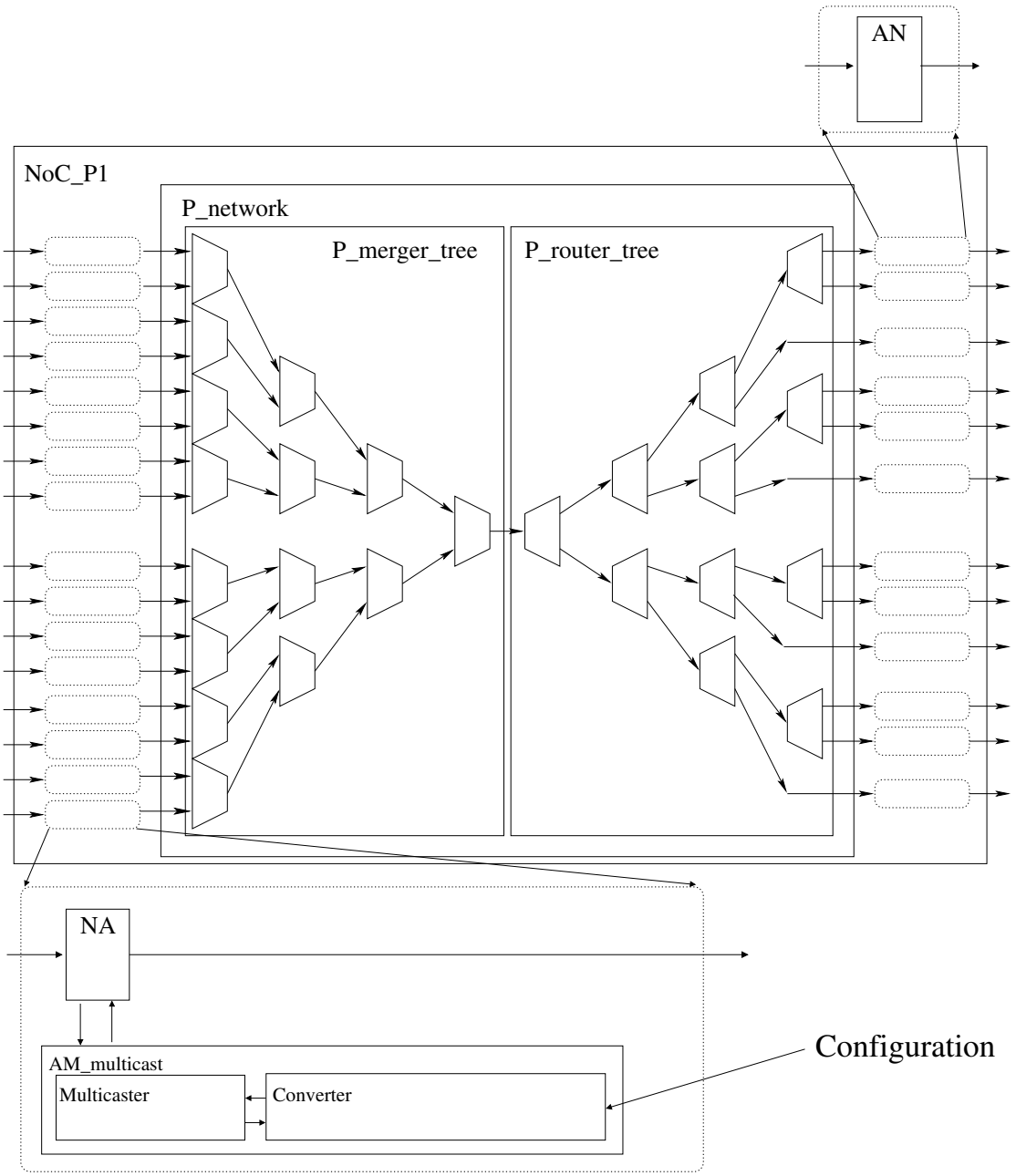


Figure 8.8: NoC1: 4-phase bundled data network where multicasts are handled in the NA, network adapter. Gate-level implementations of the different blocks can be found in appendix D.

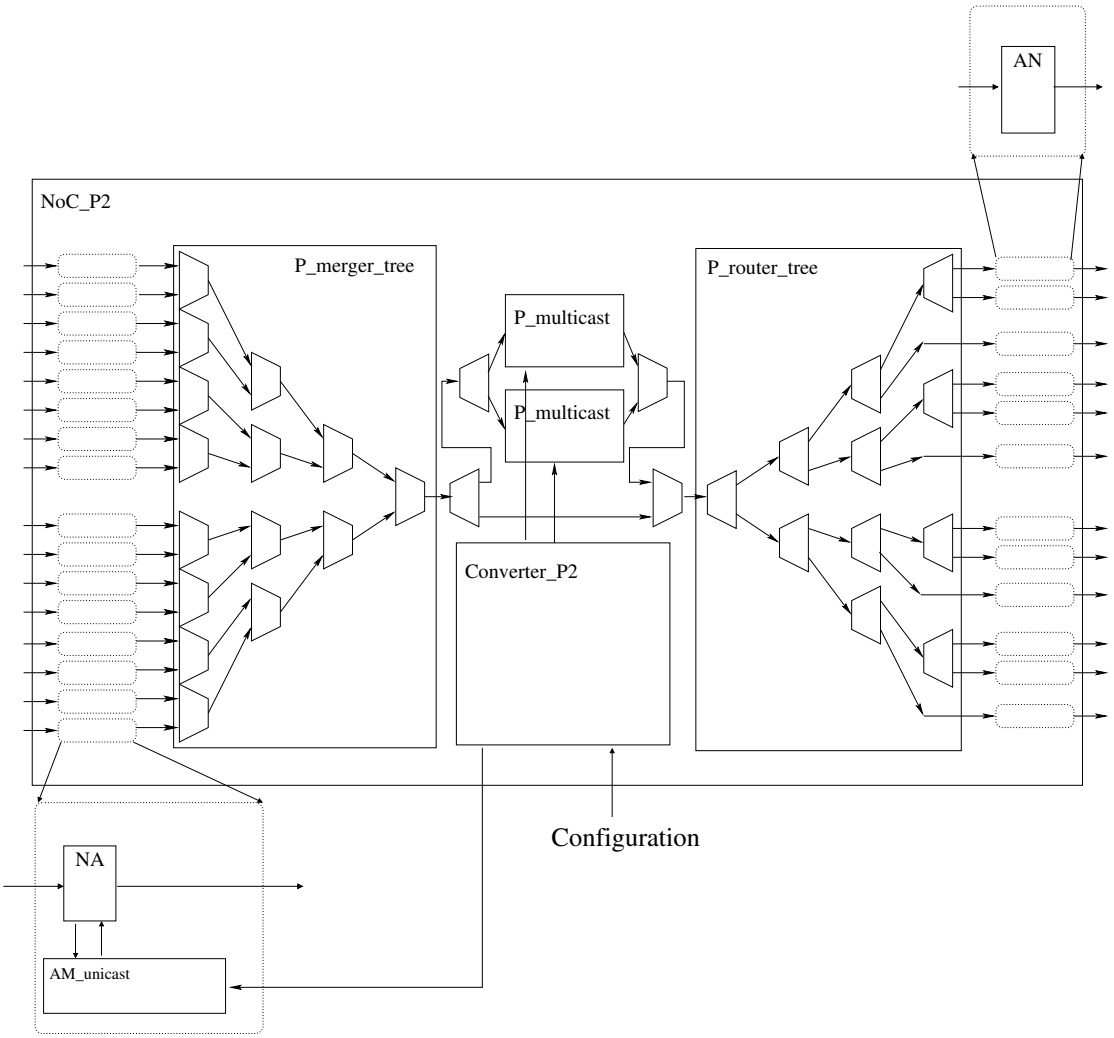


Figure 8.9: NoC2: 4-phase bundled data network where multicasts are handled in share multicast blocks. Gate-level implementations of the different blocks can be found in appendix D.

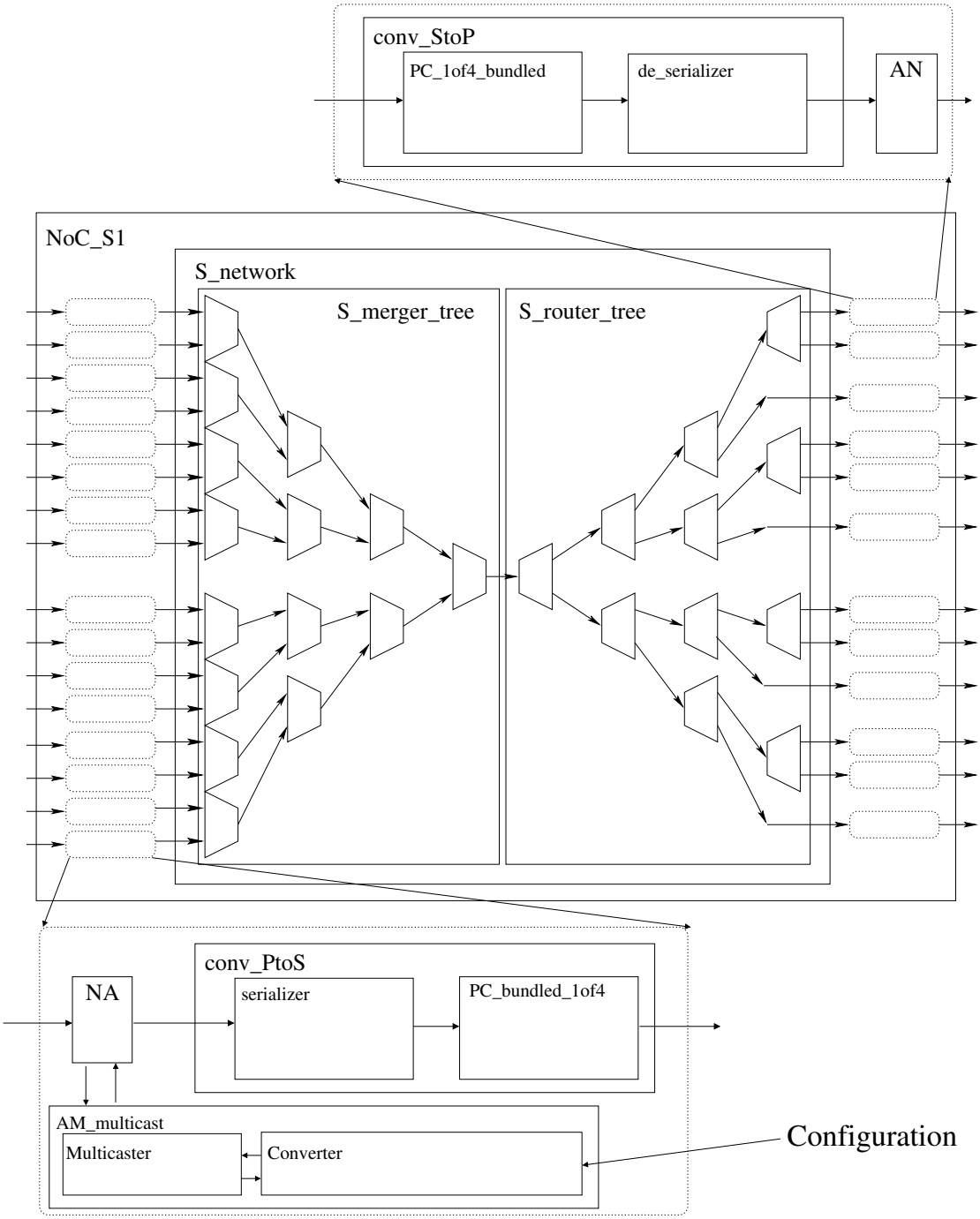


Figure 8.10: NoC3: *1-of-5* delay-insensitive network where multicasts are handled in the NA, network adapter. Gate-level implementations of the different blocks can be found in appendix D.

Chapter 9

Verification

This chapter introduces the techniques which are used to verify the behavior of the NoCs and the individual network blocks.

9.1 Overview

When designing a large and complex system, the behavior of the system must be properly verified. This is typically done by creating a testbench which controls the inputs to the system and monitors the outputs. The testbench should expose the system to different scenarios and test if it behaves as expected in all imaginable situations. Random tests are not likely to detect all possible errors, and instead every system part and feature should be tested separately and in different combinations. It is also necessary to include tests which stress the system to its maximum.

In this project, two types of testbenches are created. A 'main' testbench which verifies the functionality of the entire network and a number of small 'individual' testbenches which test each network block individually. The 'main' testbench is able to test all networks as they are accessed and configured the same way. The common network interface was introduced in chapter 6. The construction of the 'main' testbench is gone through in the succeeding sections.

The 'individual' testbenches make it much easier to thoroughly verify the behavior of the individual network blocks, because the inputs and outputs can be directly controlled and observed, respectively. Some input situations rarely occur when the network blocks are integrated into a NoC. As an example, the *mutex*, which was introduced in chapter 4.3.1, is ensuring that 2 signals are mutual exclusive. The correct behavior of the *mutex* is hard to verify when it is integrated into other network blocks. The testbench in appendix E.4.4 tests all possible situations that can occur. Many, but not all, of the network blocks have individual testbenches. It should also be noted that the testbenches are not documented and that the testbenches are not included in the Appendix, but can be found on the CD-ROM.

Besides testbenches, many of the network blocks contain behavioral code which monitors the signals inside the block. This makes it possible to report if an unsuspected situation occur. In ordinary sequential programming languages, such as C and C++, this is known as 'assertions'. As an example the NA, network adapter must be idle when new data arrives. This is verified by the 'Verilog' code in figure 9.1 by checking if `o_route_req` or `i_route_ack` is high when

```
`ifdef ERROR_CHECKING
always @(posedge i_valid)
begin
  if(o_route_req!=0 || i_route_ack!=0)
  begin
    $display ("ERROR!. Valid signal came through when
     network adapter was already busy.\n");
    $display ("  req: %b",o_route_req);
    $display ("  ack: %b",i_route_ack);
    $stop;
  end
end
`endif
```

Figure 9.1: Example of verification code which monitors the signals in the NA, network adapter. The code writes an error message and stops the simulation if an unsuspected situation occurs.

new data arrives. Note that ERROR_CHECKING must be defined for the verification code to be enabled. This allows all verification code to be enabled or disabled by means of a single definition. The file, *global.v*, in appendix E.3.1, is used to specify which network to instantiate. Error checking and debug information is also enabled/disabled in this file.

9.2 Main testbench

Figure 9.2 illustrated how the main testbench is constructed by attaching special verification modules to the input ports, output ports, and the configuration port of the network interface which was specified in chapter 6. Each verification module contains a number of *tasks*[1] which is used by the **Testbench controller** to perform a number of tests. The main testbench code instantiates the different verification modules, the network under test, and contains the code which implements the **Testbench controller**. It can be found in appendix E.4.5.

In the succeeding subsection the verification modules are explained in more detail, but here I briefly mention their basic functionality. The **Configuration controller** module gives easy access to the network configuration, including multicast and control of the *master* signal. The **Lego2 master** module is responsible for sending data into the network using the Lego2 protocol, while the **Lego2 slave** module is responsible for receiving data and check if the data is correct. A separate slave module is instantiated for each output port while a single master module is connected to all input ports. The reason that a single master module is used to control all input ports, is that data must be sent into the network on several input ports at the same time instant. As it is not possible to call several *tasks* in parallel, this behavior requires that the sending of data is implemented in a single module.

When a *packet* is sent into the network, it must be verified that it arrives at the expected output ports, and that the data has not be altered inside the network. A problem is that it is not possible to tell from which input port the *packet* was sent when it arrives at an output port. One solution is to encode a unique ID for the input port into the data. Also, when several *packets* are

[1] A *task* is a behavioral procedure call

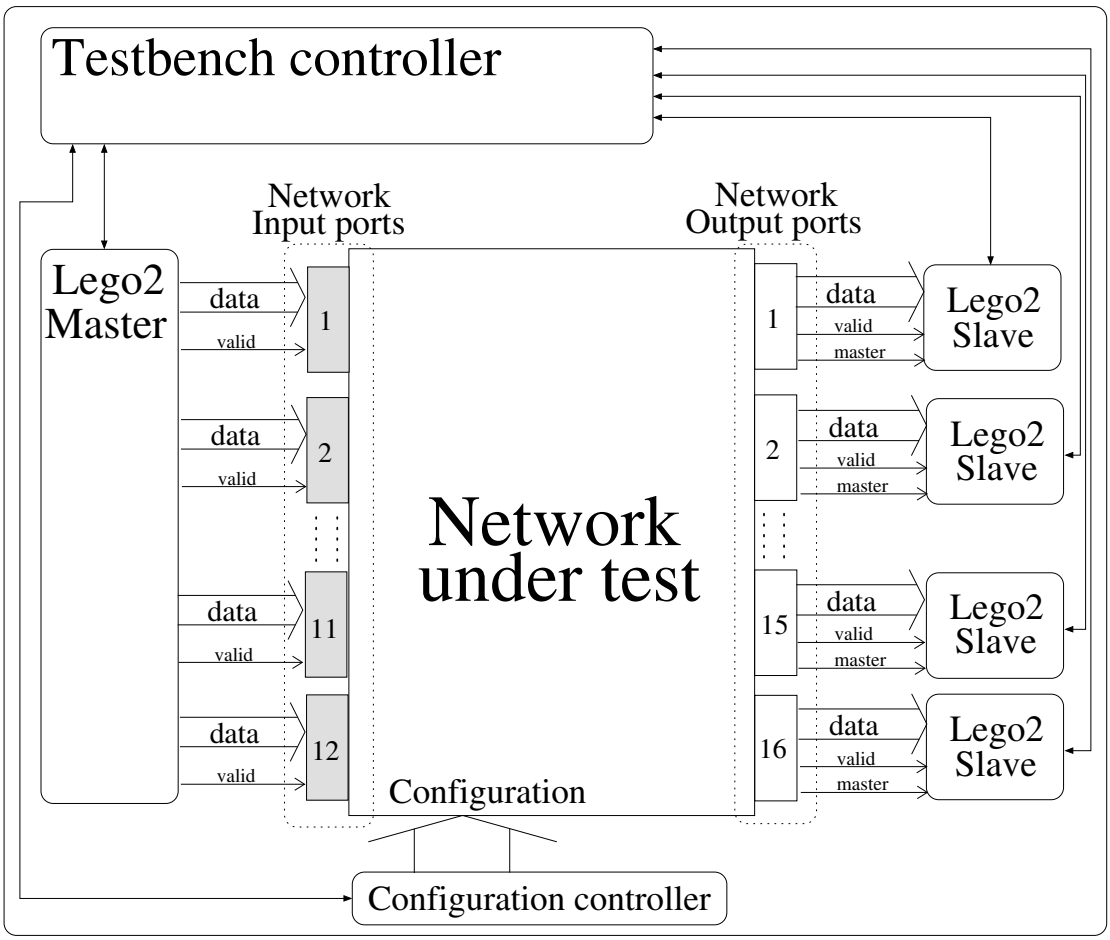


Figure 9.2: The testbench is constructed by attaching special verification module to the input ports, output ports and configuration.

sent into the network at the same time, it is very important that *packets* do not contain the same data. If this was the case it would not be possible to distinguish the *packets* from each other, and the *packets* could in theory have arrived at the wrong output port. Random data is therefore not a possibility, and instead the data is encoded as shown in table 9.1. The *packet* contains the input port who sent it, as well as the addresses of the output ports to which it was sent. It should be noted that when a *packet* is multicasted, the same *packet* is sent to several output ports and there is no way to distinguish the *packets*, except for the *master* signal which is specified for each destination output port. If a *packet* is sent to more than 2 output ports it is not possible to distinguish the *packets*.

In order to test that several *packets* addressed for the same output port has been received correctly, the **Lego2 slave** is implemented such that it expects a *packet* from each input port. The module contains an array of data values, one for each input port. As the input port is encoded into the *packet*, it is possible to use the input port as an index into this array.

	Unused	Output port3	Output port2	Output port1	Input port	
msb	bit 17-16	bit 15-12	bit 11-8	bit 7-4	bit 3-0	lsb

Table 9.1: Encoding of the data which is sent into the Noc. The encoding makes it possible to distinguish all *packets*.

9.2.1 Verification modules

In the following, a short description of the functionality is given for each verification module.

Configuration controller This module eases the configuration of the network. It consists of *task* to enable a certain route, **enableRoute**, by specifying the input and output port for the route and the value of the *master* signal. It also contains a *task* to disable a certain route, **disableRoute**, and a *task* to disable all routes, **clear**. The code can be found in appendix E.4.3.

Lego2 master This module sends data into the network using the Lego2 protocol. The module is first configured by specifying which input ports that are to receive data. This is done using the *task* **setup_txs** which configures the module to send data to single input port. **setup_txs** can be called several times if data is to be sent to several input ports. When the module has been configured, the *task* **txs** is called to send the configured data to the inputs ports. Data is only sent to the input ports which have been configured, and all data are sent at the same time. Code can be found in appendix E.4.1.

Lego2 slave This module counts the number of *packets* that arrives at each output port and checks if the received data is correct. It contains a single *task*, **setExpectedData** which tells the module which data and *master* signal to expect from a certain input port. When it receives a *packet*, it checks if a *packet* was expected from the input port which sent it, and checks if the data is correct. Then it notes that no further data is expected from the specific input port. This is to make sure that the arrival of the same *packet* twice is treated as an error. If an error occurs, it makes a pulse on a wire which is read by the **testbench controller**. The Code can be found in appendix E.4.2.

9.2.2 Tests

A number of different tests are performed to test different aspects of the network. Each test is conducted by performing the following actions in sequence:

1. The wanted routes are setup by using the **Configuration controller**.
2. It is specified which input ports that are to receive data and which data they are to receive. This is done using the **Lego2 master**.
3. Each **Lego2 slave** module is told which data to expect from which input ports.
4. **Lego2 master** is instructed to send the requested data to the input ports.

5. Wait for a number of clock-cycles which should ensure that the network has delivered all data on the output ports.

6. Check if the **Lego2 slave**s received the expected number of *packets*, and if these packets were all correct.

The following lists the different tests, their purpose and notes on their implementation.

Unicast The purpose of this test is to test unicast. Each input port sends data to each output port, one at a time. All combinations are tested.

Multicast The purpose of this test is to test the multicast functionality. Each input port sends data to two output ports at a time. All combinations of input and output ports are tested.

Multicast2 This is another multicasting test. Two input ports are doing multicast at the same time. Many, but not all, combinations are tested due to the large number of combinations.

Chaos Test This test is stressing the network. All inputs are sending data to the same output port at the same time instance. All output ports are tested.

Adder Tests the basic functionality of the adders which are inserted when integrating the NoC interface into the 'Aphrodite DSP'.

Chapter 10

Logic synthesis and simulation

In this chapter the logic synthesis and simulation flow is briefly discussed, and some notes are given on the power and area estimates.

10.1 RTL simulation

When a design has been implemented as an RTL-description, it is simulated to verify the behavior of the implementation. The purpose of this simulation is to check the functionality of the behavioral code, not the timing. In many synchronous design flows, such as the one used at *William Demant Holding*, the default simulation settings do not include gate-delays. This is a problem as the designed networks are specified directly at gate-level by instantiating and connecting specific cells from the standard cell library. If there is no delays through the cells, the asynchronous circuits will behave unexpected. Actually, the behavioral version of the used standard cell library does not include delays at all. Therefore, new cells are created which wraps the cells from the standard cell library to insert delays. The new cells are presented in appendix B.

10.2 Logic synthesis

After the RTL-description has been verified, a logic synthesis tool is used to translate the RTL-description into a netlist of standard cells. Among many things, the logic synthesis tool optimizes the design, chooses drive-strengths and insert buffers. The logic synthesis tools are designed to handle synchronous design and cannot be used on asynchronous design as the design styles are very different. If the optimization step is not removed from the synthesis flow, one can be absolutely certain that the circuit will not work as expected. This is easily fixed by stopping the logic synthesis after the design is mapped to gate-level.

10.3 Gate-level simulation

When simulating at gate-level, it is important to turn off X-propagation on the registers which do synchronization. If X-propagation is enabled, the simulator will propagate 'X' if a timing-violation occurs due to a setup or hold error on the register input. This will stall the asynchronous circuit and the simulation stops. In this project, the registers in the NA, network adapter discussed in chapter 8.2.1 contains synchronization registers.

10.4 Place and Route

After the gate-level implementation has been verified, the design is 'placed and routed'. This allows to estimate the wire capacitances and insert buffers which are needed to drive these. Unfortunately, the network designs are not 'placed and routed' due to the lack of time. It could be very interesting to do this in the future and it will, hopefully, show a positive impact on the total length of wires and their power consumption.

10.5 Area and power estimates

Area estimates of the networks are extracted from the gate-level implementation and only include area of the standard cells. If the area of wires should be included, the design should have been 'placed and routed'.

An estimate of the power consumption can be performed on the gate-level implementation or after 'place and routing'. The former only contains the power consumption in the standard cells, while the latter includes the power consumption of wire transitions as well.

The plan was to estimate the power by integrating the networks into 'Aphrodite' and use the original testbench. This makes it easy to compare the original network and the networks designed in this project. The networks were successfully integrated and simulated on the RTL-description. We did not manage to extract power estimates due to tool issues that are solved at the time of writing.

Chapter 11

Results and discussion

This chapter presents and discusses the results.

11.1 Overview

The designed networks are successfully integrated into the 'Aphrodite DSP' and mapped to gate-level by a logic synthesis tool. As the designs are not 'placed and routed', the presented area estimates are based on the gate-level implementations and do not contain area of wires. The area of the original Aphrodite network is also estimated from its gate-level implementation to enable a fair comparison between the designed networks and the existing netowkr solution. The area of the original network is estimated as the area of the multiplexors at the output ports, minus the area of the computational units in the *MUXADDers*. The original Aphrodite network is estimated to consist of 7395 gate equivalents which takes up roughly 0.092 mm^2 in a 0.18 μm process[1]. This corresponds to 1.2% of the total chip area.

Throughout the report it has been stated that both area and power estimates would be presented. Unfortunately it has not been possible to extract power estimates due to unforseen difficulties with the design and verification tools.

In the following sections the area and bandwidth estimates are presented and commented, and the results are discussed.

11.2 Results

Table 11.1 shows a comparison between the area and bandwidth of the different network implementations. The bandwidth is extracted from the gate-level simulation using worst case timing parameters. It is a measure of the number of bits that can be sent through the network, without synchronization or multicasting. The listed bandwidths are only possible if the synchronization in the network adapters are decoupled from the network communication, which is currently not the case. A detailed list of the area usage for each network are shown in table 11.2.

[1]There are approximately 80.000 gate equivalents pr mm^2

Network	Area (mm^2)	Bandwidth (MBit/s)	% of original network	% of chip
Original	0.093		100%	1.19%
NoC1	0.084	358	91%	1.08%
NoC2	0.078	253	85%	1.00%
NoC3	0.19	100	203%	2.41%

Table 11.1: Area usage and bandwidth of the different networks.

11.2.1 Bundled data networks

The first two networks use a 4-phase bundled data protocol, and the network blocks are transparent to handshakes. The difference between the two networks is that NoC1 handles multicasting in the network adapters, while NoC2 handles multicasting in two shared multicast blocks. This decreases the area of multicasting from 14% to 8%, but increases the latency for unicasting, as an additional *merge* and *router* block are inserted at the root of the network.

The latency for all 4-phases of the handshake is 5.2 ns for the *merge* block and 8 ns for the *router* block. This is a total latency of 53 ns and 66 ns for the longest paths through the two networks. As 19 bits are transferred in each *packet*, the bandwidths are 358 MBit/s and 253 MBit/s, respectively.

11.2.2 *1-of-5* network

The third network employs narrow links using a *1-of-5* delay-insensitive encoding, and handles multicasting in the NA network adapter. At first sight it seems odd, that this network is twice the size of the other networks. The main reason is that the *serializer* and *de-serializer* blocks use roughly 50 % of the area, but I believe that there are a number of other reasons as well:

- The two other networks contain no buffers at all. This makes all blocks in these networks extremely simple. In contrast, the *router* and *merge* blocks in NoC3 contains one and two latches, respectively. Some of these latches could be removed without decreasing the bandwidth, as the *serializer* and *de-serializer* are currently the bottlenecks in this design.

- The *1-of-5* blocks use a large amount of C-elements with both 2 and 3 inputs. These C-elements use an area of 5-6 gate equivalents which is almost as much as a flip-flop. If more effective implementations were used, this area could be decreased. For example inverting C-elements could be used in many situations. If possible, the C-elements could even be designed as custom cells.

- Each block uses a number of OR gates with 5 and 8 inputs. These OR gates are initially implemented as an binary tree of 2 input OR gates. This is large and slow, and should be implemented using NOR-NAND constructs or other inverting multi-input gates.

- The *serializer* and *de-serializer* takes up 45% of the total area. Around half of the area in the *de-serializer* is used by flip-flops which could be exchanged by latches if the con-

Block	**Number**	**Area/block**	**Area**	**Percent**
NA, network adapter	16	115	1840	27 %
AN, network adapter	12	135	1620	24 %
AM_multicast	16	60	960	14 %
Merger	15	87	1305	19 %
Router	11	92	1012	15 %
Total			6637 (0.084 mm^2)	

(a) NoC1: 4-phase bundled data network where multicasting is handled in the NA, network adapters.

Block	**Number**	**Area/block**	**Area**	**Percent**
NA, network adapter	16	115	1840	29 %
AN, network adapter	12	135	1620	26 %
Merger	15	87	1305	21 %
Router	11	92	1012	16 %
Multicast part			(478)	(8%)
Merger	2	87	174	3%
Router	2	92	184	3 %
P_Multicast	2	60	120	2 %
Total			6255 (0.078 mm^2)	

(b) NoC2: 4-phase bundled data network where multicasting is handled in share multicast blocks.

Block	**Number**	**Area/block**	**Area**	**Percent**
NA, network adapter	16	115	1840	12 %
AN, network adapter	12	135	1620	11 %
AM_multicast	16	60	960	6 %
Merger	15	145	2175	14 %
Router	11	103	1133	8 %
Serializer	16	272	4352	29 %
De-Serializer			2960	20 %
normal	8	241		
discards one flit	4	258		
Total			15040 (0.19 mm^2)	

(c) NoC3: *1-of-5* delay-insensitive network where multicasting is handled in the NA, network adapters.

Table 11.2: Area usage of the different blocks in the three networks.

trollers were modified. In the *serializer*, 30% of the area in used by 3 OR gates with 14 inputs. As already explained, these large OR-gates are made as trees of 2 input non-inverting OR gates, which is far from an optimal implementation. At last, other implementations of the controller in both the *serializer* and *de-serializer* should be considered, as the current implementation is large compared to the rest of the network.

I believe that the listed changes could decrease the implementation of NoC3 by at least 2-3000 gate equivalents, thereby using approximately 2% of the total chip area.

As the *1-of-5* networks consist of 4 different network blocks with different latency, it is difficult to calculate the bandwidth. The *serializer* is the slowest block and therefore the bandwidth of the network is determined by this. It takes 20 ns to transfer a *flit* of 2 bits, which gives a bandwidth of 100 Mbit/s. From the gate-level implementation I have measured that it takes 240 ns to send 20 bits of data from an input to an output port, including serialization and de-serialization of the *flits*. This gives a bandwidth of 83 Mbit/s. This is lower than 100 Mbit/s but it includes the 5 *flits* used for routing and EOP, and handshakes to start the *serializer*. The individual *router* and *merger* blocks can transfer approximately 500 MBit/s.

11.3 Discussion

The first thing to notice is that the two bundled data networks are actually 9 and 15% smaller than the original network, which must be considered a success. In addition to this, the area of the original network is expected to increase further if the designs were 'place and routed'. This is because area of the gates in the networks are only one part of the story. The original network in Aphrodite contains many long wires which complicate routing and require bus-drivers. Some of the network input ports are connected with up to 6 different output ports, which require even larger bus-drivers. These bus-drivers are not included in the area estimates. In contrast, all the networks designed in this project consists of short wires, which drive at most two gate-inputs. Also, the *router* and *merge* blocks can be distributed among the communication blocks, making routing easier for the 'place and route' tool. The GALS methodology even allows timing-closure to be performed for each communicating block instead of the entire design, thereby making 'place and routing' easier.

It is also noticeable that the network adapters take up more than 50% of the total network area in the two bundled data solutions. This is a quite a surprise, as they practically contains nothing more than a flip-flop for the data. It illustrates that the *router* and *merge* blocks in the bundled data networks are very small. If buffers are inserted into the a network to increase its bandwidth, the area of the network will raise, making the network infeasible for the 'Aphrodite DSP'.

The difference between NoC1 and NoC2 is that NoC2 implements multicasting in shared multicast blocks, which decreases the area of multicasting from 14% to 8%. This slight decrease in area corresponds to less than 0.1% of the total chip area. If each network input port sends *packets* to more than 2 destinations, the blocks which handles multicast, will take up far more area, and the gain of shared multicast blocks will increase. If the number of simultaneous multicasts increases, the advantage might no longer be present, as each shared multicast block requires an additional *router* and *merge* block in the network. These additional blocks are

connected with wires which must be routed on the chip, and included in the area estimate. In concerns of power in NoC2, each *packet* sent from the NA, network adapters pass through an additional *router* and *merge* block, thus increasing the power consumption of unicasts. On the other hand, multicasts use less power in NoC2, because they are handled at the root of the tree. In summary, the number of possible multicasts for each block and the number of simultaneous multicasts affects the areas of the two bundled data networks. The power consumption is dependent of the distribution of unicasts and multicasts in the communication, and the networks must be 'place and routed' and power simulated before choosing between these two networks.

The NoC3 network takes up more area than the other networks, but it might not be out of the question. The network uses 2.4% of the total chip area, which is not an unreasonable amount. As the width of the links is 6 wires, less wires are to be routed than in the bundled data networks. Also, the use of *1-of-5* encoding decreases the problem with crosstalk, because only one of the wires make a transition when transferring data. As crosstalk is increasing with decreasing technology ,this might be important in future chips. *1-of-5* encoding doe not need matched delay and the circuitry can be made very fast. Especially in processes with large variations, as the matched delay in a bundled data solution must be conservative and therefore slow. The *router* and *merge* blocks can handle approximately 500 Mbit/s, which makes it a good choice for bandwidth demanding applications. A number of links can be routed in parallel if more bandwidth is needed.

Even though power consumption has not been estimated, it is still possible to make some remarks concerning the expected tendency. The dynamic power used at at node is given by

$$P = CV^2 f$$

where C is the capacitance of the node, V is the voltage, and f is the switching frequency. As mentioned, the fanout and length of the wires in the designed networks, results in a reduction in capacitance compared to the existing network solution. Concerning switching activity, the original network uses roughly 11 transitions for each *packet*, which is 2 for the valid signal and 9 transtions for half of the data-bits[2]. In the bundled data solutions, a *packet* is transferred using approximately 27 transitions, which is 4 for the handshake and half of the 23 data bits including the return to zero. The *1-of-5 solution* always uses 60 transitions as it takes 4 transitions to transfer a *flit*, and a *packet* consists of 15 *flits* including routing. The switching activity for the *1-of-5* network is increased by a factor of 6, and the power consumption of this network will probably increase. For the bundled data networks, the number of transitions are almost tripled. On other hand I postulate that the capacitance of the wires are reduced by more than a factor of 3, thereby reducing the power consumption. When technology decreases, wires becomes taller but thinner which increase their resistance and the coupling between the wires [4]. This tendency promotes short wires even more as the capacitance of wires, and thereby power consumption, increases.

In the 'Aphrodite DSP' only a subsection of the input and output ports can communicate. If a larger subset can communicate, the advantages over the existing network solution are increased, as the designed networks do not increase in size. The area of the networks are linear

[2]I assume that data is un-correlated

dependent on the number of inputs and outputs, making the network very scalable. The networks could very well be used in other applications with different number of inputs, outputs and data bits. The network could be decoupled from the synchronization at the network adapters, if needed. This depends on the bandwidth and latency requirements for the application. It should be noted that the bandwidth in the two bundled data networks decreases with the number of inputs and outputs, because the networks do not include any buffers. It is possible to trade area for bandwidth by inserting buffers between the *router* and *merge* blocks such that communication is pipelined. As all *packets* pass through the root of the network, the power consumption raises with an increasing number of inputs and outputs. For very large applications and bandwidth demanding systems, a more general topology might be beneficial such that locality can be exploited by placing blocks that create high traffic loads close to each other. On the other hand, the router nodes in such a general topology are much larger.

Chapter 12

Conclusion

This chapter concludes on the work done in this project and the results which was discussed in the previous chapter.

I have successfully designed and implemented three asynchronous *packet-switched*, *source-routed* networks. The networks have 16 input ports ,12 output ports, and supports multicasting. Two of the networks use a 4-phase bundled data protocol, while the third uses a *1-of-5* delay-insensitive encoding. The networks are integrated into the 'Aphrodite DSP' and mapped to standard cells in a $0.18\mu m$ technology. The estimated areas are extracted from the gate-level implementations, as the designs are not 'placed and routed'.

All three networks show promising results and the smallest bundled data network takes up 0.084 mm^2, which is 15% less than the existing network solution. Still, it provides sufficient bandwidth and is able to communicate 358 Mbit/s, using estimates from the gate-level implementation. If the designs are 'place and routed', the area of the designed networks are expected to decrease even more, relative to the existing network solution.

The power consumption of the networks are not estimated due to difficulties regarding the design and verification tools. Still, I have argued that the power consumption decreases for the bundled data network, due to the shorter wires. If time permitted, it would be very interesting to 'place and route' the designs, such that the area and power could be properly estimated and compared.

The network which uses *1-of-5* encoding takes up 0.19 mm^2, which is twice as much as the original network. Still, this is only 2.4% of the total chip area. I expect this network to use more power than the bundled data network, and it is not a good choice for this application. It might be an option for other applications who need more bandwidth, as it provides the largest bandwidth per wire and is delay-insensitive.

The designed networks are 'plug-and-play' and can easily be ported to future generations of the 'Aphrodite DSP' or other applications with different number of inputs, outputs and bandwidth requirements. The size of the networks are linear dependent of the number of inputs, outputs, multicasts and number of data bits. This makes the networks very scalable. The bandwidth in the two bundled data networks decrease with the number of inputs and outputs, because

these networks do not include any buffers. It is possible to trade area for bandwidth by inserting buffers between the *router* and *merge* blocks, such that communication is pipelined. As the networks decouple the communicating blocks, the chip can be designed using the GALS methodology, which eases timing-closure and allows each block to run in its own clock domain.

Even though the networks are not 'placed and routed', the results illustrates that it is possible to design small *packet-switched* networks for applications with limited bandwidth requirements.

Bibliography

[1] Open core protocol(ocp) homepage. http://www.ocpip.org/.

[2] Visual stg lab (vstgl) homepage. http://vstgl.sourceforge.net/.

[3] John Bainbridge and Steve Furber. CHAIN: A delay-insensitive chip area interconnect. *IEEE Micro*, 22:16–23, 2002.

[4] W. J. Bainbridge and S. B. Furber. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 118–126. IEEE Computer Society Press, March 2001.

[5] Davide Bertozzi and Luca Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):1101–1107, 2004.

[6] Tobias Bjerregaard and Jens Sparsø. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *Proc. Design Automation and Test in Europe (DATE'05), ACM sigda, 2005*, pages 1226–1231, 2005.

[7] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[8] William J. Dally. Virtual-channel flow control. *IEEE Transactions on Applied Superconductivity*, 3(2):194–205, 1992.

[9] Stephen B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[10] Ran Ginosar. Fourteen ways to fool your synchronizer. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 89–96. IEEE Computer Society Press, May 2003.

[11] Sudhakar Yalamanchili José Duato and Lionel Ni. *Interconnection Networks: An enginerring approach*. Morgan Kaufmann, 2003. ISBN 1-55860-852-4, Revised printing.

[12] Rikard Thid Shashi Kumar Mikael Millberg, Erland Nilsson and Axel Jantsch. The nostrum backbone - a communication protocol stack for networks on chip. In *Proceedings of the IEEE International Conference on VLSI Design*, pages 693–696, 2004.

[13] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

Appendix A

Synchronization

When transferring data from one clock domain to another or from an asynchronous to a synchronous domain, safe synchronization must be applied. As described in [10] it is extremely dangerous to optimize the synchronization circuits as it is quite easy to make fatal mistakes which makes the circuit malfunction. The article describes some of the many mistakes that has been made in the past when the designer thinks that he has done something really clever.

Figure A.1 shows the basic two flop synchronizer which is a safe and widely used synchronization technique. In this example the two flop synchronizer uses a push scheme to transfer data between two different clock domains. As seen the receiver synchronizes the request and the sender synchronizes the acknowledge.

If the first flops get metastable because the request line changes just as *clk_2* ticks, the *r1* signal will be metastable for a unknown period of time. Instead of using *r1* directly it is feed into a new flop and has a whole clock period to stabilize.

[10] notes the following equation for Mean Time Between Failures (MTBF) for the two flop synchronizer

$$MTBF = \frac{e^{\frac{T}{\tau}}}{T_W f_A f_D} \tag{A.1}$$

where τ is the settling time constant of the flop, T_W is a parameter related to its time window of susceptibility, f_A is the clock frequency of the flops and f_D is the frequency of which data is

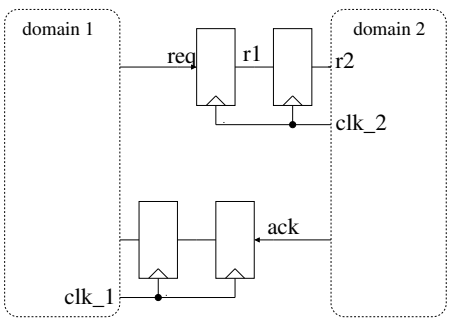


Figure A.1: 2 flop synchronization.

pushed across the clock domains. As seen, the MTBF is closely related to T. In a 18 μm process, conservative parameters are τ =10ps, T_W = 50ps, f_A = 200Mhz and $f_D = 20Mhz$ the MTBF the two-flop synchronizer is 10^{240} years. Compared to this a single flop will enter metastability at a rate of $\frac{1}{T_W f_A f_D} = 5\mu s$ which can hardly be considered safe.

Appendix B

Cell library

Instead of instantiating cells from the cell library directly, a new virtual cell library is created which wraps the cells in the used standard cell library. A cell in the virtual cell library starts with the letter C_.
There are several reasons for creating this template cell library.

- To insert a propagation delay in the behavioral simulation. As explained in section 10.1, there is no delay in the used cell library.
- To take advantage of the complex gates in the used standard cell library which has to be implemented by simple gates if they do not exist.
- To implement the asynchronous cells such as the Mutex and different C-elements using complex gates.

In addition to the virtual cell library, a small number of 'template cells' has also been created. They all start with the letter TC_ and are created to ensure unit capacitance of the inputs as the design rule in section 4.2 indicates. The template cells are used whenever a gate needs a drivestrength larger than one. This is for example the case if an enable signal is feed to a number of latches. There is also template cells with a variable number of inputs. For example, a multiplexor and an N-input OR gate which are constructed using a number of simple or complex cells. It should be noted the selection of template cells are far from complete. This is because I only implemented the ones which were needed for this project.

The use of template cells makes the actual design almost independent of the actual standard cell library. If the cell library is exchanged, only the virtual cells and template cells must be re-implemented. The delay through the cell does however differ from each cell library and the matched delay must therefore be recalculated based on the used standard cell library.

Figure B.1 shows an inverter template cell which parameter is the FANOUT, that is the standard unit capacitance that it can drive. The verilog 'generate' statement is then used to select an appropriate cell from the used standard cell library. The verilog code for virtual cells and template cells can be found in appendix E.1.2 and E.1.1, respectively.

```
module TC_INV(a,z);
  parameter FANOUT = 2;

  input     a;
  output    z;
  wire      s_z;

  generate
    if(FANOUT<=1)
      inv0d0 inv_d0(.a(a), .z(s_z));
    else
    if(FANOUT<=4)
      inv0d1 inv_d1(.a(a), .z(s_z));
    else
    if(FANOUT<=8)
      inv0d4 inv_d4(.a(a), .z(s_z));
    else
    if(FANOUT<=16)
      inv0d7 inv_d7(.a(a), .z(s_z));
    else
    if(FANOUT<=32)
      inv0da inv_da(.a(a), .z(s_z));
  endgenerate
  assign #`GATE_DELAY z = s_z;
endmodule
```

Figure B.1: Example of wrapper cell which inserts a gate delay into the behavioral model.

Appendix C

CD contents

The attached CD-ROM includes all source code from appendix E, divided into 3 directories:

blocks/ All network blocks.

include/ Template cell library and *global.v* which contains global defines such as routes, which network to use, debug_level etc.

noc_top/ Contains the tree networks as well as the main testbench.

Appendix D

Network building blocks

D.1 Common blocks

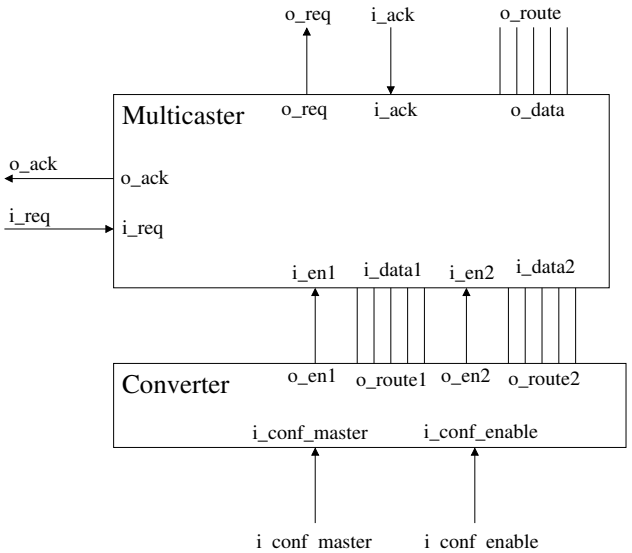
D.1.1 AM_multicast

Function:

This is an *Address Manager* which is used to provide multicast for the AN network adapter. The Multicaster module (appendix D.1.5) is used to provide the multicast functionality while the converter module converts from the general configuration matrix into a number of routes and enable signals. The converter is included in appendix E.2.1.

Note that even though the illustration below only shows a multicast with 2 destinations, any number of destinations can be provided.
Code can be found in Appendix E.3.2

Gate-level Implementation:



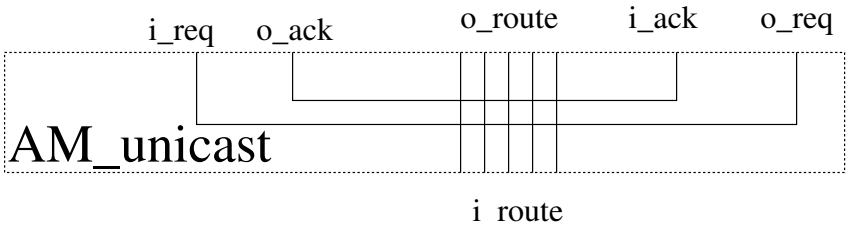
D.1.2 AM_unicast

Function:

This is an *Address Manager* which is used to provide unicast for the AN network adapter. It is simply implemented by connecting the ingoing handshake with the outgoing handshake and as it only has one route the incoming route is also connected directly to the outgoing.

Note that even though the illustration below only shows a multicast with 2 destinations, any number of destinations can be provided.
Code can be found in Appendix E.3.3

Gate-level Implementation:



D.1.3 AN, network adapter

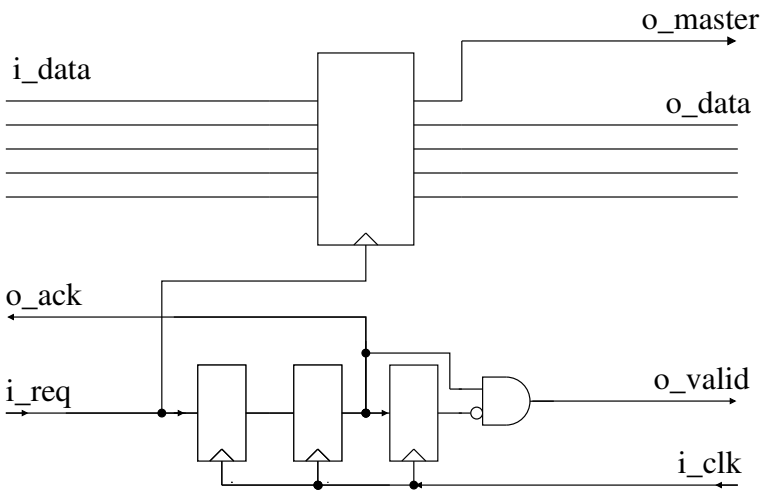
Function:

The AN network adapter inputs a *packet* using a 4-phase bundled data protocol and outputs data using the Lego2 protocol. The design and implementation was discussed in chapter 8.2.2

Data is latched in a flip-flop on the positive edge of `i_req`. It could also be implemented by connecting a level-sensitive latch to `o_ack`, but this gave rise to problem when integrating the NoC's into the 'Aphrodite DSP'.

Note that the handshake is synchronized to the clock domain of the output port. This means that it takes at least 4 clock cycles from for the entire handshake to complete and buffers should be inserted before the network adapter to decouple if from the network.
Code can be found in Appendix E.3.4

Gate-level Implementation:



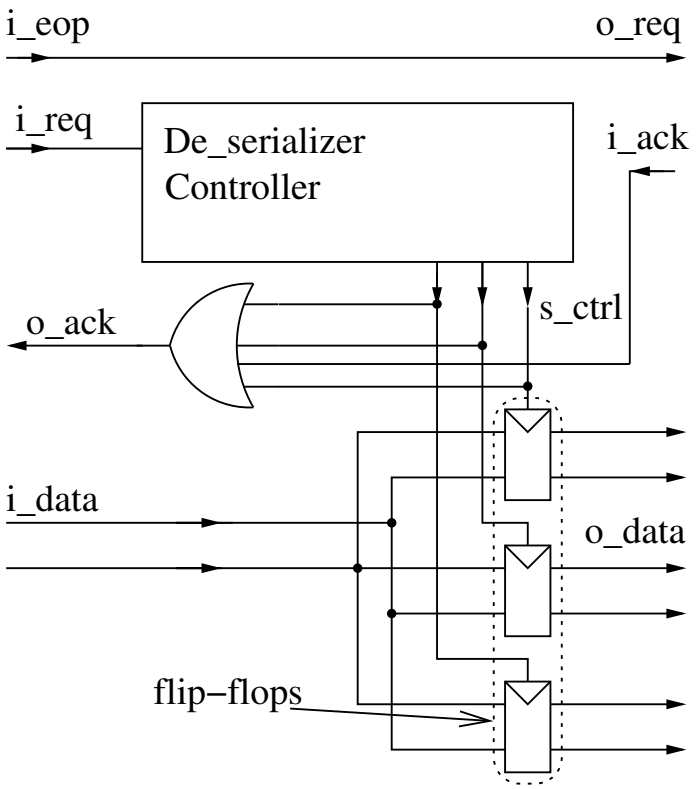
D.1.4 de_serializer

Function:

This network block de-serializes a stream of 2 bit *flits* into a single data value. Both input and output uses a 4-phase bundled data protocol. After the last *flit* has been received, a special "End Of Packet"(*EOP*) wire is asserted to indicate that there are no more *flits* in the *packet*.

Details about the implementation can be found in chapter E.3.5
Code can be found in Appendix E.3.5

Gate-level Implementation:



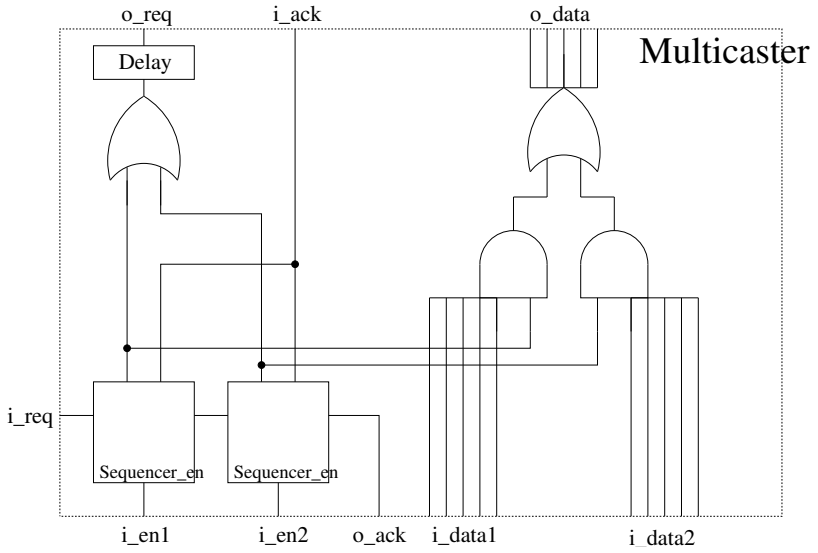
D.1.5 Multicaster

Function:

The multicaster is used to generate any number of multicasts. The illustrated gate-level implementation only shows 2 multicasts, but the code is made such that the *multicaster* can provide any number of multicasts.

When `i_req` is asserted the first *sequencer* handshakes on the output port while outputting the data from `i_data1`. When the first *sequencer* has finished its handshake the succeeding *sequencer* is activated and so forth. At last the incoming handshake is completed. Each sequencer can be disabled which is used in for example the multicast *Address Manager* from appendix D.1.1.
Code can be found in Appendix E.3.6

Gate-level Implementation:

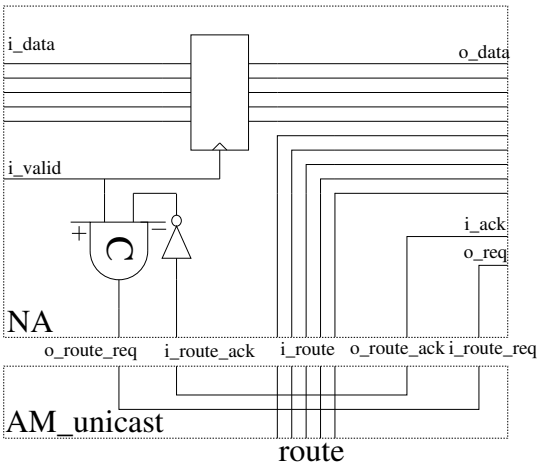


D.1.6 NA, network adapter

Function:

The NA network adapter inputs data using the Lego2 protocol and generates *packets* using a 4-phase bundled data. It is assumed that the delay between data on the input port is large enough for the network adapter to have sent the last *packet*. An *Address Manager* must be connected to the network adapter which provide the route for the *packet*. The current *Address Managers* are AM_unicast and AM_multicast. AM_unicast is shown in the figure to illustrate the basic behavior of the adapter. Chapter 8.2.1 contains more in depth information about the implementation.
Code can be found in Appendix E.3.7

Gate-level Implementation:



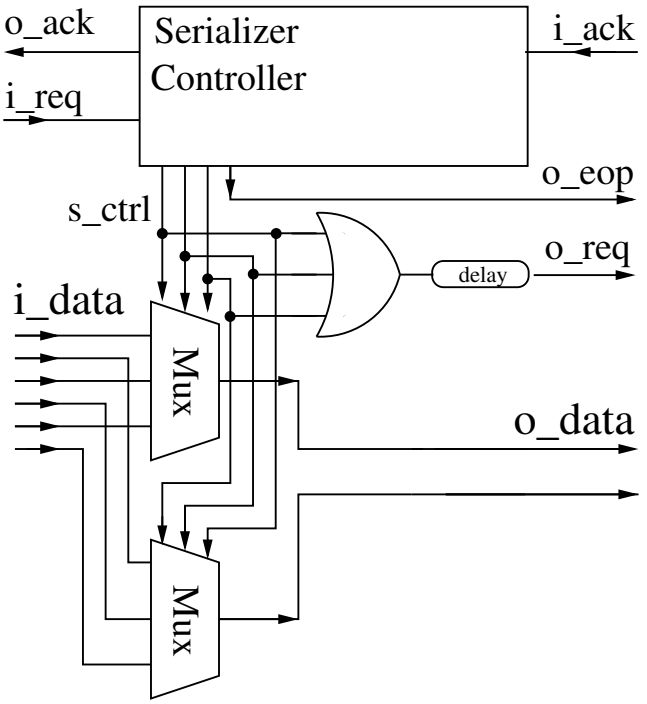
D.1.7 serializer

Function:

This network block serializes a *packet* into a stream of 2 bit *flits*. Both the input and output uses a 4-phase bundled data protocol. After the last *flit* has been sent a special "End Of Packet" (EOP) wire is asserted to indicate that there are no more *flits* in the *packet*. The EOP wire works like the request wire and a 4-phase handshake must be performed using the EOP wire as *request*.

Details about the implementation can be found in chapter 8.2.3.
Code can be found in Appendix E.3.8

Gate-level Implementation:

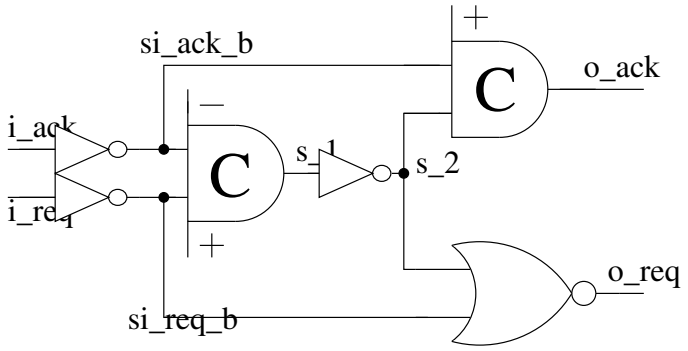


D.1.8 Sequencer

Function:

The *sequencer* accepts a 4 phase handshake on the left hand side and generates a 4-phase handshake on the right hand side before completing the handshake on the left hand side. A more in depth explanation of its functionality and the implementation can be found in chapter 4.4. The *sequencer* is for example used in the serializer which can be found in appendix D.1.7.
Code can be found in Appendix E.3.9

Gate-level Implementation:

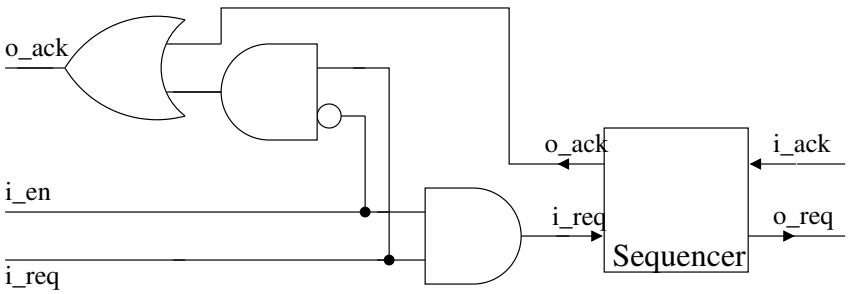


D.1.9 Sequencer_en

Function:

This module wraps the *sequencer* from appendix D.1.8 such that it can be disabled. If `i_en` is '1' is functions as the *sequencer*. If it is disabled, by lowering `i_en`, it does not produce any outgoing handshake and simply completes the incoming handshake right away.
Code can be found in Appendix E.3.10

Gate-level Implementation:

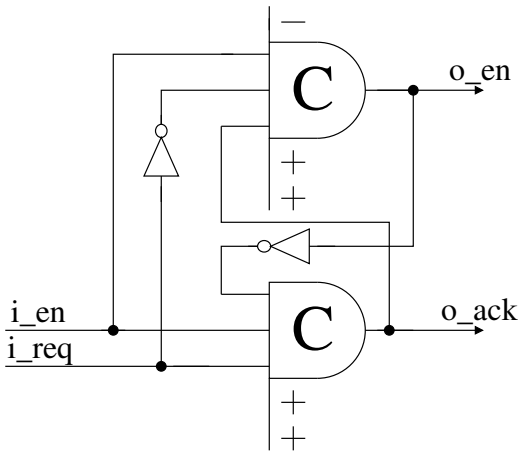


D.1.10 Sequencer2

Function:

This *sequencer* is used in the *de-serializer* in appendix E.3.5. The functionality and implementation was gone through in chapter 8.2.4.
Code can be found in Appendix E.3.11

Gate-level Implementation:



D.2 Bundled data blocks

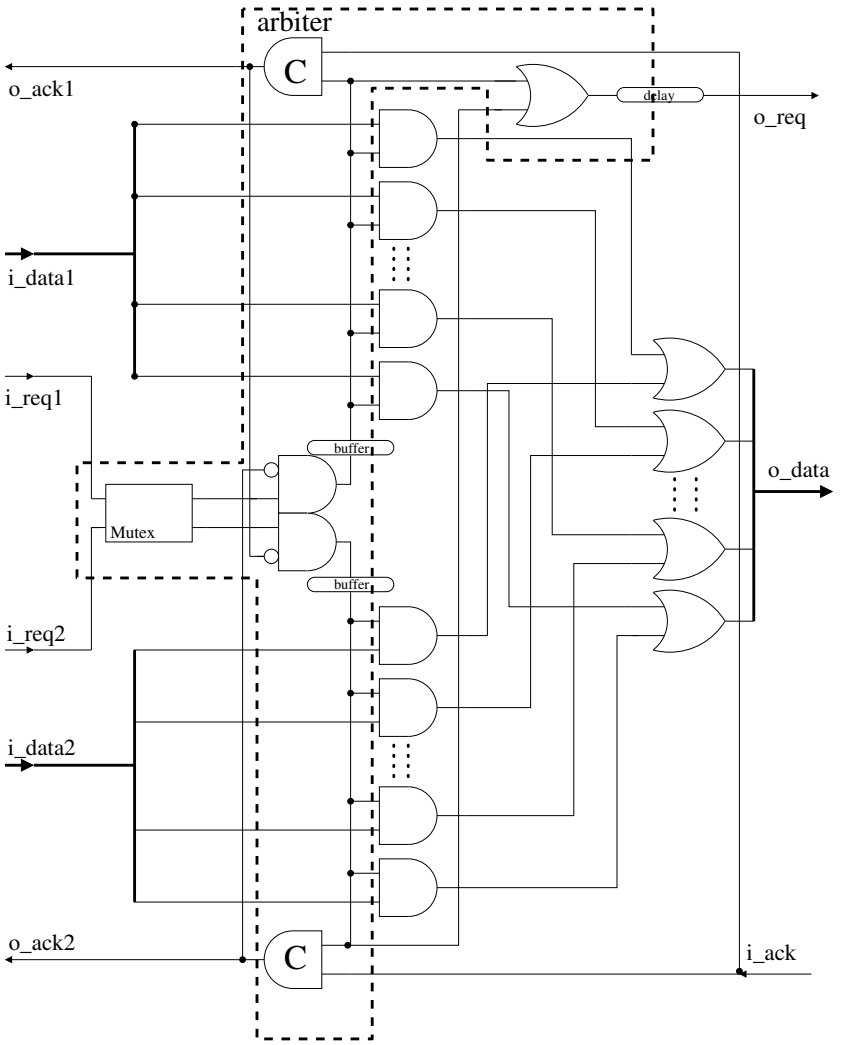
D.2.1 P_merge

Function:

This module merges 2 input ports onto a single output port. The protocol on both input and output ports are 4-phase bundled data. The 2 input ports do not have to be mutual exclusive and arbitration is done inside the merge module.

The merger consists of a handshake arbiter and a multiplexor. The handshake arbiter grants one of the inputs access to the output ports and locks the arbiter until the handshake is complete. The multiplexor is implemented using an complex AND-OR gate.
Code can be found in Appendix E.5.1

Gate-level Implementation:

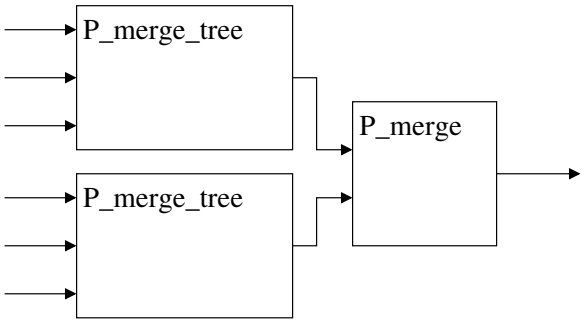


D.2.2 P_merge_tree

Function:

This module instantiates a binary tree of P_merge elements. The code is recursively defined as shown in the figure below by instantiating two smaller P_merge_tree's and connecting them by a single P_merge element.
Code can be found in Appendix E.5.2

Gate-level Implementation:



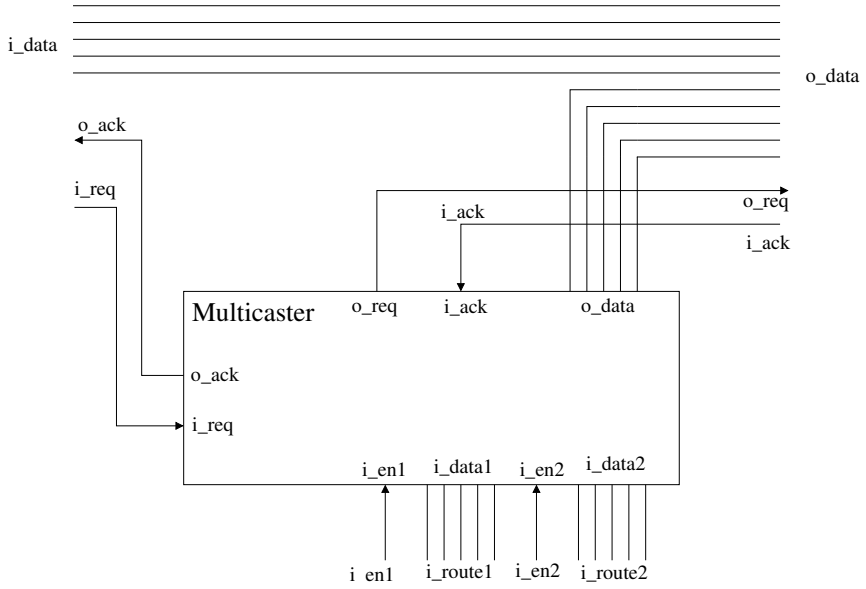
D.2.3 P_multicast

Function:

This module implements a 4-phase bundled data multicaster. It accepts a *packet* on its input port and generates a number of *packets* on the output ports. The routes of the *packets* must be provided.

Note that even though the illustration below only shows a multicast with 2 destinations, any number of destinations can be provided.
Code can be found in Appendix E.5.3

Gate-level Implementation:

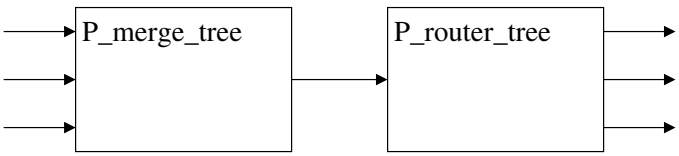


D.2.4 P_network

Function:

This module instantiates a binary tree of P_merge elements and a binary tree of P_router elements and connects them as illustrated below.
Code can be found in Appendix E.5.4

Gate-level Implementation:



D.2.5 P_router

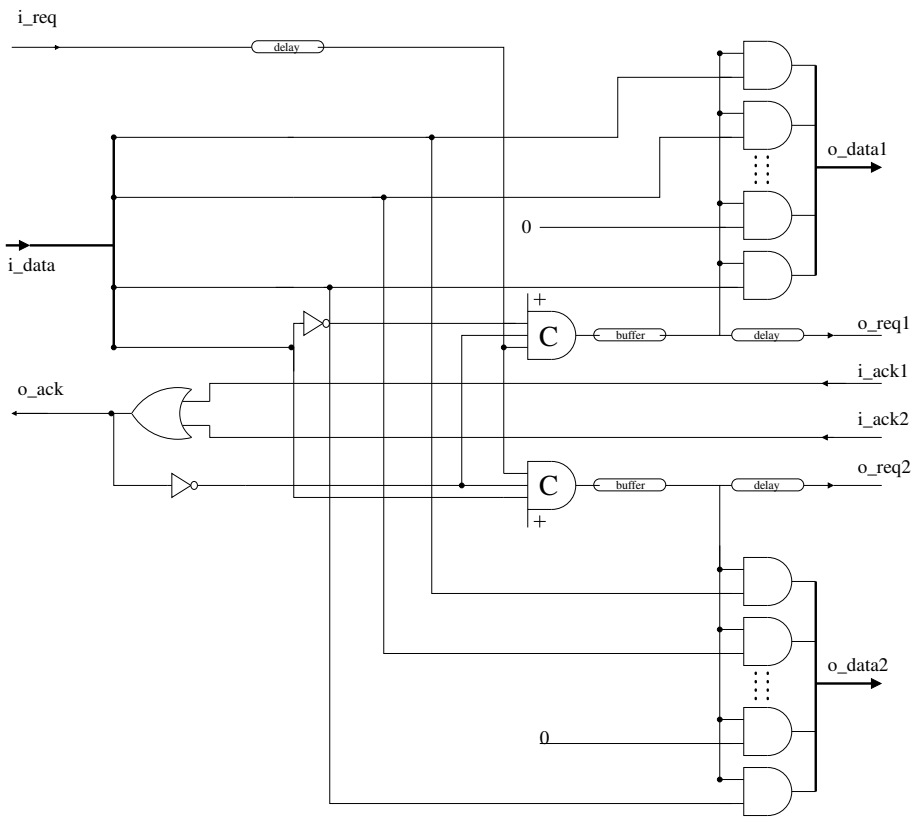
Function:

This module routes a *packet* from the input port to one of the two output port using a 4-phase bundled data protocol. As was explained in chapter 7.1, the most significant bit is used to determine the route of the *packet* and the route is shifted left by one. This also means that the least significant route bit is set to 0.

Note that AND gates are inserted such that data are only sent to one of the output ports. The data could safely be routed to both output ports since only one of the port receives a handshake. This would cause the data wires to shift through the entire network and would contribute heavily to the power consumption.

In the illustration below the route is 2 bits, but both the number of bits used for the route and for the data can be specified.
Code can be found in Appendix E.5.5

Gate-level Implementation:

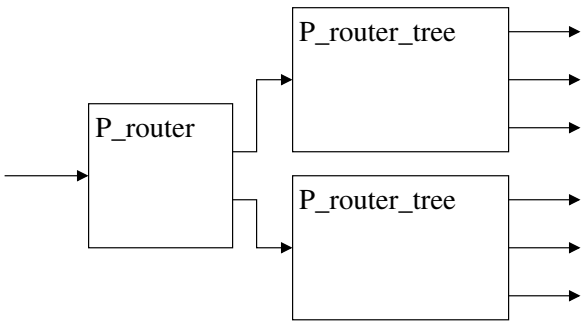


D.2.6 P_router_tree

Function:

This module instantiates a binary tree of P_router elements. The code is recursively defined as shown in the figure below by instantiating two smaller P_router_tree's and connecting them by a single P_router element.
Code can be found in Appendix E.5.6

Gate-level Implementation:



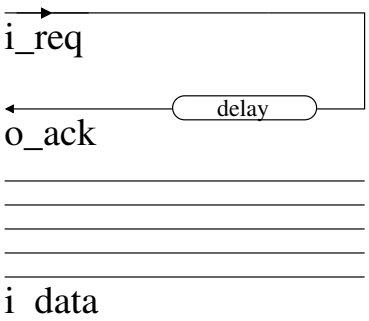
D.2.7 P_sink

Function:

This is a simple sink for the 4-phase bundled data protocol. The sink is a eager consumer which acknowledges the input as soon as it arrives and is always ready to receive new data.

The module also contains behavioral code which displays the received data.
Code can be found in Appendix E.5.7

Gate-level Implementation:



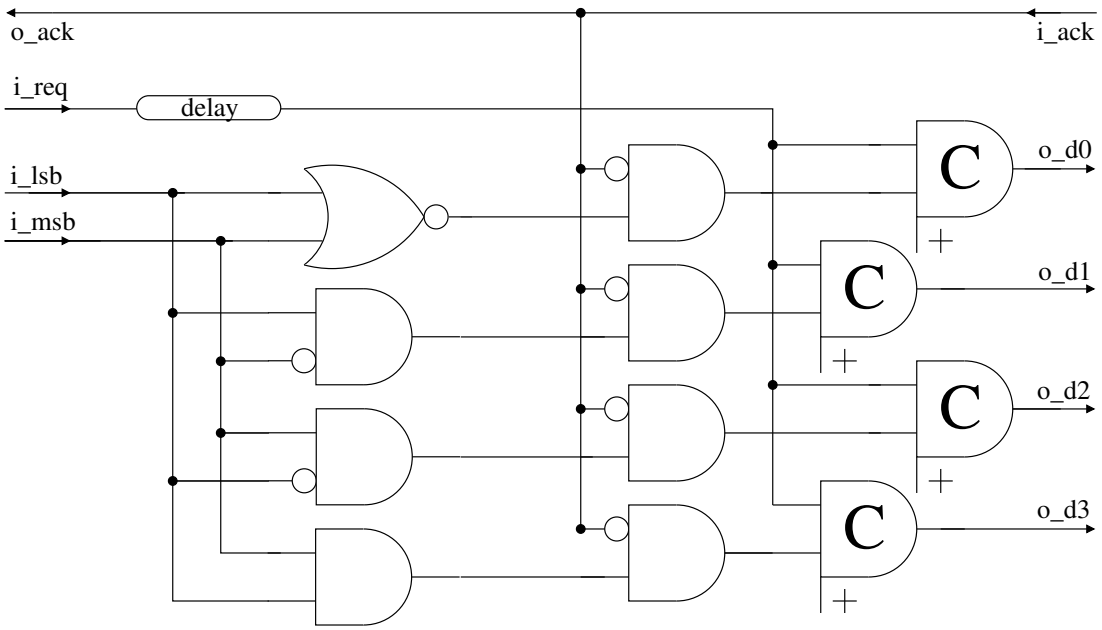
D.3 *1-of-5* blocks

D.3.1 PC_bundled_1of4

Function:

This 2 bit *Protocol converter*, converts a 4 phase bundled data protocol into a 4 phase *1-of-4* delay insensitive encoding. Note the AND gates with inverted inputs which disables all outputs while the *acknowledge* wire is asserted. If these gates were not inserted the output might change before *i_req* goes low.
Code can be found in Appendix E.6.1

Gate-level Implementation:

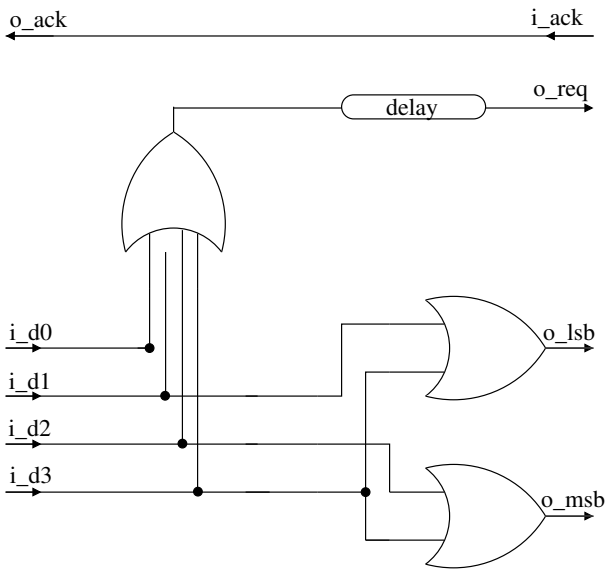


D.3.2 PC_1of4_bundled

Function:

This 2 bit *Protocol converter*, converts a 4 phase *1-of-4* delay insensitive encoding into a 4 phase bundled data protocol. The *request* signal is simply generated when either of the incoming data wires goes high and the 4 one-hot wires converted into a 2 bit representation. A small delay is inserted in the *request* wire to make sure that the data is stable when the `o_req` wire is asserted. Code can be found in Appendix E.6.2

Gate-level Implementation:

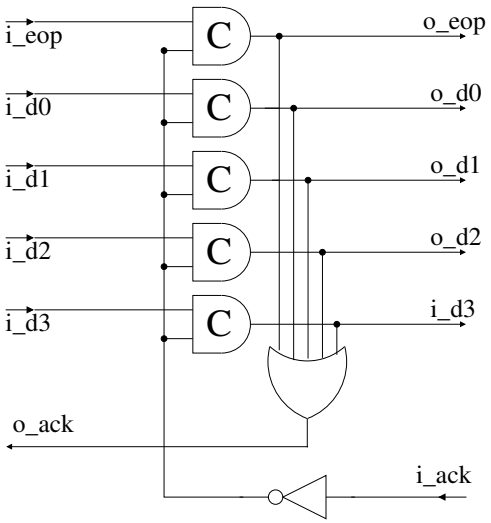


D.3.3 S_latch

Function:

This is a simple *1-of-5* latch using a 4-phase handshake protocol. The latch is inspired from the CHAIN network [3].
Code can be found in Appendix E.6.3

Gate-level Implementation:

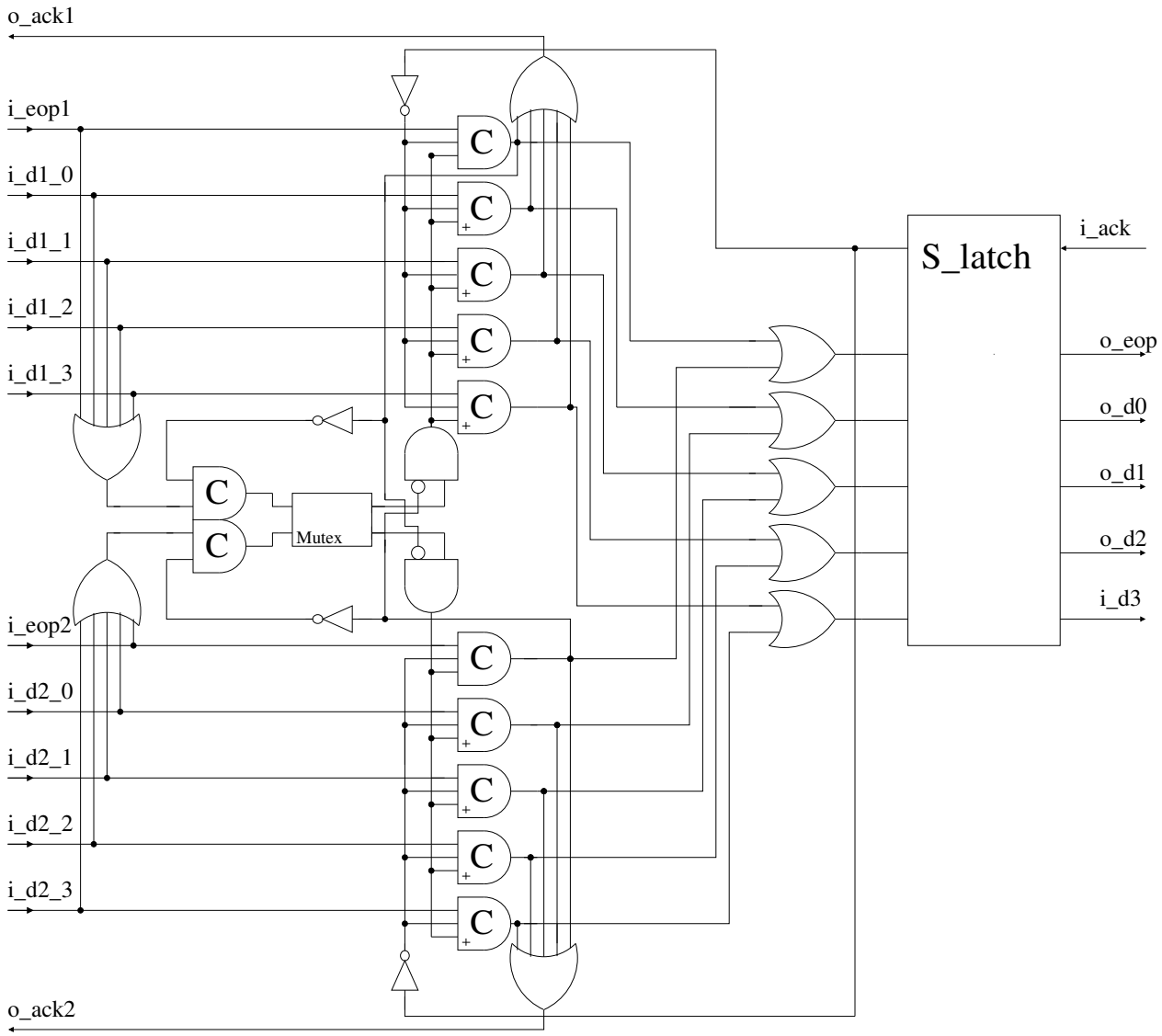


D.3.4 S_merge

Function:

This module merges 2 input ports using a *1-of-5* protocol onto a single output port. The 2 input ports do not have to be mutual exclusive and arbitration is done inside the merge module. The merger is inspired from the CHAIN network [3] and the only difference is that C-elements are used as state-holding instead of set/reset latches.
Code can be found in Appendix E.6.4

Gate-level Implementation:

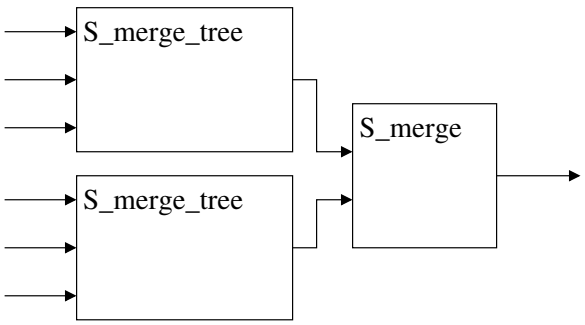


D.3.5 S_merge_tree

Function:

This module instantiates a binary tree of S_merge elements. The code is recursively defined as shown in the figure below by instantiating two smaller S_merge_tree's and connecting the by a single S_merge element.
Code can be found in Appendix E.6.5

Gate-level Implementation:

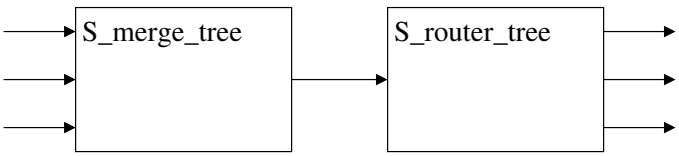


D.3.6 S_network

Function:

This module instantiates a binary tree of S_merge elements and a binary tree of S_router elements and connects them as illustrated below.
Code can be found in Appendix E.6.6

Gate-level Implementation:

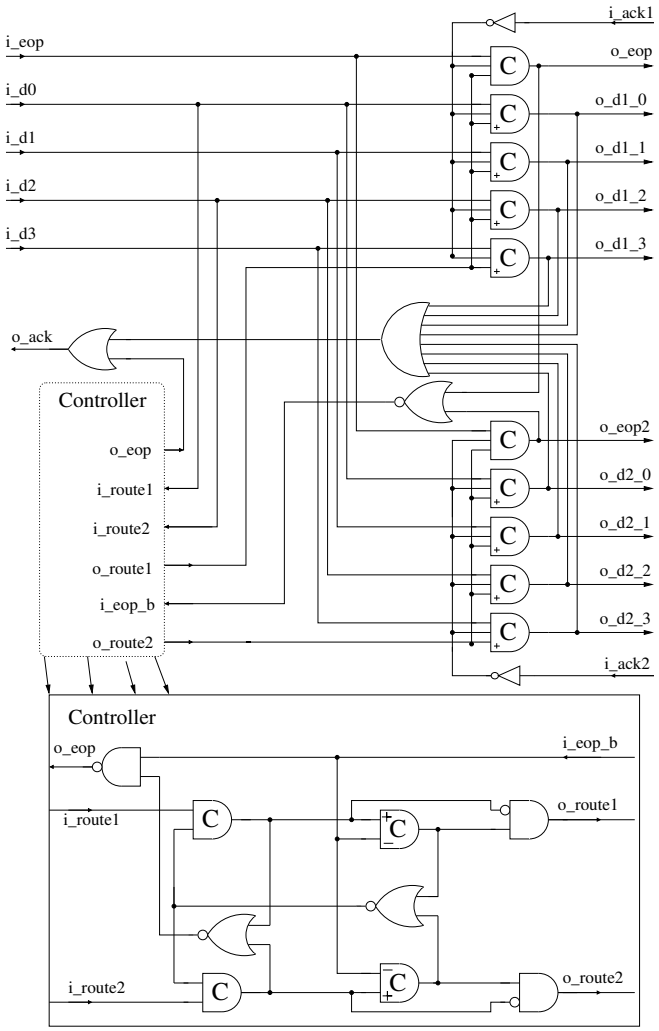


D.3.7 S_router

Function:

This module inputs a *packet* using a *1-of-5* protocol and routes it onto one of two possible output ports based on the values of the first *flit*. The value '0' routes the remaining *flits* to output port 1 while the value '2' routes remaining *flits* to output port 2. The router is inspired from the CHAIN network [3] and the only differences are that C-elements are used as state-holding instead of set/reset latches and that only the values '0' and '2' routes the data.
Code can be found in Appendix E.6.7

Gate-level Implementation:

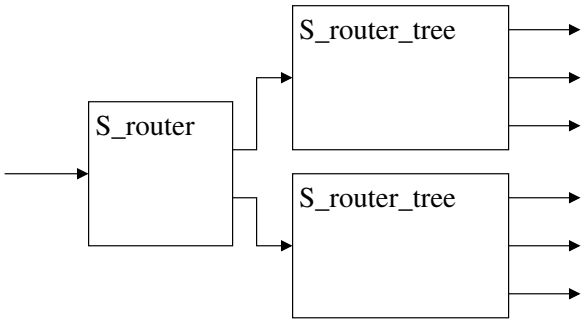


D.3.8 S_router_tree

Function:

This module instantiates a binary tree of S_router elements. The code is recursively defined as shown in the figure below by instantiating two smaller S_router_tree's and connecting the by a single S_router element.
Code can be found in Appendix E.6.8

Gate-level Implementation:



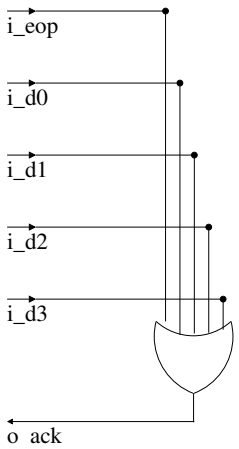
D.3.9 S_sink

Function:

This is a simple sink for the *1-of-5*, 4-phase handshake protocol. The sink is a eager consumer which acknowledges the input as soon as it arrives and is always ready to receive new data.

The module also contains behavioral code which displays the received data.
Code can be found in Appendix E.6.9

Gate-level Implementation:



D.3.10 S_source

Function:

This is pure behavioral source for the *1-of-5*, 4-phase handshake protocol. The source contains *tasks* for both sending a single *flit* and an entire *packet* including the End Of Packet (EOP) *flit*. Code can be found in Appendix E.6.10

Appendix E

Verilog Code

E.1 Cell library

E.1.1 cell_library.v

```
`ifndef _cell_library_v
`define _cell_library_v
/*
  Description:
    Library of cells.


  Naming convention:
    C_"Cell type""Number of Inputs""Options"_D"Strength"

    Options:
      'A': Each 'A' means that a single port is negated.

    Strength:
      The drive strength og the port. If this is 1, it is not written


    e.g:
      C_AND2     : 2 input AND gate
      C_AND2_D3  : 2 input AND gate with drivestrength 3
      C_AND2A    : 2 input AND gate where the first port is negated
      C_OR4AA    : 4 input OR gate where the first 2 ports are negated


    All cells that resets to something ends with "_R0" of "_R1" for
       reset to zero or 1

    C-Elements are a bit special as they can be assymetric
      'M': Each 'M' means that a single port is assymetric "-" (MINUS
         ). Starting from first port
```

```
      'P': Each 'P' means that a single port is assymetric "+" (PLUS)
         . Starting from last port

    e.g:
      C_C2_R0      : C-element with 2 inputs that resets to zero
      C_C3P_R1_D2 : C-element with 3 inputs that resets to one. The
         last port is assymetic "+". The output
                  has the drive strength 2
      C_C2M_R0     : C-element with 2 inputs that resets to one. The
         first port is assymetic "-"
      C_C4MMP_R1   : C-element with 4 inputs that resets to one. The
         first 2 ports are assymetic "-"
                  while the last port is assymetic "+"

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/


//`include "cell_library_behavioral.v"
`include "cell_library_at58000.v"

`timescale 1ns/1ps

module TC_delay(a,z);
  parameter DELAY_PS=200; //Delay in pico seconds

  `define DelayPrStage 200
  `define NumberOfStages (DELAY_PS/`DelayPrStage+1)
  `define NumberOfInverters (`NumberOfStages*2)

  input      a;
  output     z;

  wire [0:`NumberOfInverters] s_tmp;

  //Delay is simply an even number of inverters
  inv0d0 inverter_chain[0:`NumberOfInverters-2] (.i(s_tmp[0:
     `NumberOfInverters-2]), .zn(s_tmp[1:`NumberOfInverters-1]));
  inv0d1 inverter_output (.i(s_tmp[`NumberOfInverters-1]), .zn(s_tmp[
     `NumberOfInverters]));
  //Assign input
  assign s_tmp[0] = a;
  //Assign output
  assign #(DELAY_PS/1000.0) z = s_tmp[`NumberOfInverters];
endmodule // TC_delay

/*
  OR, 5 input
```

```
*/
module C_OR5(a,b,c,d,e,z);
  input a,b,c,d,e;
  output z;

  wire s_1,s_2;

  C_OR2 or11(.a(a), .b(b), .z(s_1));
  C_OR3 or12(.a(c), .b(d), .c(e), .z(s_2));
  C_OR2 or2(.a(s_1), .b(s_2), .z(z));
endmodule

/*
  NOR, 5 input
*/
module C_NOR5(a,b,c,d,e,z);
  input a,b,c,d,e;
  output z;

  wire s_1,s_2;

  C_OR2 or11(.a(a), .b(b), .z(s_1));
  C_OR3 or12(.a(c), .b(d), .c(e), .z(s_2));
  C_NOR2 nor2(.a(s_1), .b(s_2), .z(z));
endmodule

/*
  OR, 8 input
*/
module C_OR8(a,b,c,d,e,f,g,h,z);
  input a,b,c,d,e,f,g,h;
  output z;

  wire s_1,s_2,s_3;

  C_OR2 or11(.a(a), .b(b), .z(s_1));
  C_OR3 or12(.a(c), .b(d), .c(e), .z(s_2));
  C_OR3 or13(.a(f), .b(g), .c(h), .z(s_3));
  C_OR3 or2(.a(s_1), .b(s_2), .c(s_3),. z(z));
endmodule

module C_ORx(inputs,z);
  parameter NUMBER=2;
  input [NUMBER-1:0] inputs;
  output z;

  `define N_UPPER (NUMBER/2)
  `define N_LOWER (NUMBER-`N_UPPER)
```

```
  wire s_z1;
  wire s_z2;

  generate
    if(NUMBER>2)
    begin
      //Upper and lower or groups
      C_ORx #(`N_UPPER) or_upper(.inputs(inputs[NUMBER-1:`N_LOWER]),.
          z(s_z1));
      C_ORx #(`N_LOWER) or_lower(.inputs(inputs[`N_LOWER-1:0]),.z(
          s_z2));
      //Connect the trees
      C_OR2 or2(.a(s_z1), .b(s_z2), .z(z));
    end else
    if(NUMBER==2)
    begin
      //Connect the trees
      C_OR2 or2(.a(inputs[0]), .b(inputs[1]), .z(z));
    end else
      assign z = inputs[0];
  endgenerate
endmodule

module TC_INV(a,z);
  parameter FANOUT = 2;

  input     a;
  output    z;

  wire      s_z;

  generate
    if(FANOUT<=2)
      inv0d0 inv_d0(.i(a), .zn(s_z));
    else
    if(FANOUT<=4)
      inv0d1 inv_d1(.i(a), .zn(s_z));
    else
    if(FANOUT<=8)
      inv0d2 inv_d2(.i(a), .zn(s_z));
    else
    if(FANOUT<=28)
      inv0d0 inv_d7(.i(a), .zn(s_z));
  endgenerate

  assign #(`GATE_DELAY) z = s_z;
endmodule

module TC_BUF(a,z);
```

```
  parameter FANOUT = 2;

  input     a;
  output    z;

  wire      s_z;

  generate
    if (FANOUT<=4)
      buffd1 buf_d1(.i(a), .z(s_z));
    else
    if (FANOUT<=12)
      buffd3 buf_d3(.i(a), .z(s_z));
    else
    if (FANOUT<=28)
      buffd7 buf_d7(.i(a), .z(s_z));
  endgenerate

  assign #(`GATE_DELAY) z = s_z;
endmodule

module TC_INV(a,z);
  parameter FANOUT = 2;

  input     a;
  output    z;

  wire      s_z;

  generate
    if (FANOUT<=2)
      inv0d0 inv_d0(.i(a), .zn(s_z));
    else
    if (FANOUT<=4)
      inv0d1 inv_d1(.i(a), .zn(s_z));
    else
    if (FANOUT<=8)
      inv0d2 inv_d2(.i(a), .zn(s_z));
    else
    if (FANOUT<=28)
      inv0d0 inv_d7(.i(a), .zn(s_z));
  endgenerate

  assign #(`GATE_DELAY) z = s_z;
endmodule

module TC_AND2A(a,b,z);
  parameter FANOUT = 2;
```

```
  input  a, b;
  output z;

  wire   s_z, s_z2;

  generate
    if (FANOUT<=4)
      an12d1 an12d1(.a1(a), .a2(b), .z(s_z));
    else
    if (FANOUT<=8)
      an12d2 an12d2(.a1(a), .a2(b), .z(s_z));
    else
    if (FANOUT<=16)
      an12d4 an12d4(.a1(a), .a2(b), .z(s_z));
    else
    if (FANOUT<=28)
    begin
      an12d4 an12d4(.a1(a), .a2(b), .z(s_z2));
      //Insert big buffer
      buffd7 buffd7(.i(s_z2), .z(s_z));
    end
  endgenerate
  assign #(`GATE_DELAY*2) z = s_z;
endmodule

/*
  Multiplexor with 1-hot capability
*/
module TC_mux(i_data, i_ctrl, o_data);
  parameter SIZE=2;
  // Inputs
  input [SIZE-1:0] i_data, i_ctrl;
  // Outputs
  output o_data;
  //
  wire [SIZE-1:0] s_tmp;

  C_AND2  ands[SIZE-1:0](.a(i_data),.b(i_ctrl),.z(s_tmp));
  C_ORx #SIZE or_tree(.inputs(s_tmp),.z(o_data));


  /*
    Error checking
  */
  `ifdef ERROR_CHECKING
  integer count,i;
  always @(i_ctrl or i_data)
  begin
      count=0;
```

```
        for(i=0;i<SIZE;i=i+1)
        begin
          if(i_ctrl[i]) count = count+1;
          if((i_ctrl[i]!=1'b1 && i_ctrl[i]!=1'b0)|| (i_data[i]==1'b1 &&
              i_data[i]==1'b0))
          begin
            $display("TC_mux : ERROR");
            $display("   Some internal signals is unknown!");
            $display("   i_ctrl: %b",i_ctrl);
            $display("   i_data: %b",i_data);
            $stop;
            end
        end
        if(count>1)
        begin
          $display("TC_mux: ERROR");
          $display("   More than 1 control is high and it is suppose to
              be 'one hot' encoding!");
          $display("   i_ctrl: %b",i_ctrl);
          $stop;
        end
    end
`endif

endmodule


`endif
```

E.1.2 cell_library_at58000.v

```
/*
  Description:
    Cell library

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _cell_library_at58000_v
`define _cell_library_at58000_v

//`uselib dir=../technology/ libext=.v

`timescale 1ns/1ps
`include "global.v"

`define GATE_DELAY 0.150 //150 ps


/*
  AND, 2 input
*/
module C_AND2(a,b,z);
  input a,b;
  output z;

  wire s_z;

  an02d1 an02d1_i(.z(s_z), .a1(a), .a2(b));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
  NAND, 2 input
*/
module C_NAND2(a,b,z);
  input a,b;
  output z;

  wire s_z;

  nd02d1 nd02d1_i(.zn(s_z), .a1(a), .a2(b));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
```

```
  AND, 2 input, A negated
*/
module C_AND2A(a,b,z);
  input a,b;
  output z;

  wire s_z;

  an12d1 an12d1_i(.z(s_z), .a1(a), .a2(b));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
  AND, 3 input
*/
module C_AND3(a,b,c,z);
  input a,b,c;
  output z;

  wire s_z;

  an03d1 an03d1_i(.z(s_z), .a1(a), .a2(b), .a3(c));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
  AND, 4 input
*/
module C_AND4(a,b,c,d,z);
  input a,b,c,d;
  output z;

  wire s_z;

  an04d1 an04d1_i(.z(s_z), .a1(a), .a2(b), .a3(c), .a4(d));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
  OR, 2 input
*/
module C_OR2(a,b,z);
  input a,b;
  output z;

  wire s_z;
```

```
  or02d1 or02d1_i(.z(s_z), .a1(a), .a2(b));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
  OR, 3 input
*/
module C_OR3(a,b,c,z);
  input a,b,c;
  output z;

  wire s_z;

  or03d1 or03d1_i(.z(s_z), .a1(a), .a2(b), .a3(c));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
  OR, 4 input
*/
module C_OR4(a,b,c,d,z);
  input a,b,c,d;
  output z;

  wire s_z;

  or04d1 or04d1_i(.z(s_z), .a1(a), .a2(b), .a3(c), .a4(d));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
  NOR, 2 input
*/
module C_NOR2(a,b,z);
  input a,b;
  output z;

  wire s_z;

  nr02d1 nr02d1_i(.zn(z), .a1(a), .a2(b));
  assign #`GATE_DELAY z = s_z;

endmodule

/*
```

```
  AND-OR
*/
module C_AOR22(a, b, c, d, z);
  input  a,b,c,d;
  output z;

  wire s_z;

  aor22d1 aor222d1_i(.z(s_z), .a1(a), .a2(b), .b1(c), .b2(d));

  //Delay through complex gates
  assign #`GATE_DELAY z = s_z;
endmodule


/*
  C-element, 2 input
    Resets(low) to zero
*/

module C_C2_R0(a,b,z, reset_b);
  input a,b, reset_b;
  output z;

  wire s_z, s_z1,s_z2;

  aor222d1 aor222d1_i(.z(s_z1), .a1(a), .a2(b), .b1(a), .b2(s_z), .c1
      (b), .c2(s_z));

  //Delay through complex gates
  assign #`GATE_DELAY s_z2 = s_z1;
  //Reset gate
  C_AND2 adds1(.a(s_z2),.b(reset_b),.z(s_z));
  //Output
  assign z = s_z2;

endmodule


/*
  C-element, 2 input
    Resets(low) to zero

    Asymetric, one plus

*/
module C_C2P_R0(a,b,z,reset_b);
  input a,b,reset_b;
  output z;
```

```
  wire s_z, s_z1, s_z2;

  aor22d1 aor22d1_i(.z(s_z1), .a1(a), .a2(b), .b1(a), .b2(s_z));

  //Delay through complex gates
  assign #`GATE_DELAY s_z2 = s_z1;
  //Reset gate
  C_AND2 adds1(.a(s_z2),.b(reset_b),.z(s_z));
  assign z = s_z2;

endmodule

/*
  C-element, 2 input
    Resets(low) to zero

    Asymetric, one minus, one plus

*/
module C_C2MP_R0(a,b,z,reset_b);
  input a,b,reset_b;
  output z;

  wire s_z, s_z1, s_z2;

  aor21d1 aor21d1_i(.z(s_z1), .b1(a), .b2(s_z), .a(b));
  //Delay through complex gates
  assign #`GATE_DELAY s_z2 = s_z1;
  //Reset gate
  C_AND2 and_reset(.a(s_z2),.b(reset_b),.z(s_z));
  //output
  assign z = s_z2;
endmodule

module C_C3MPP_R0(a,b,c,z,reset_b);
  input a,b,c,reset_b;
  output z;

  wire s_z, s_z1, s_z2;

  aor22d1 aor22d1_i(.z(s_z1), .b1(a), .b2(s_z), .a1(b), .a2(c));
  //Delay through complex gates
  assign #`GATE_DELAY s_z2 = s_z1;
  //Reset gate
  C_AND2 and_reset(.a(s_z2),.b(reset_b),.z(s_z));
  //output
  assign z = s_z2;
endmodule
```

```
/*
  C-element, 3 input
    Resets to zero

    assymetric, 2 Plusses


*/module C_C3PP_R0(a,b,c,z,reset_b);
  input a,b,c,reset_b;
  output z;

  wire s_z, s_z1, s_z2, s_z3;

  aoi321d1 aor321d1_i(.zn(s_z1), .c1(a), .c2(b), .c3(c), .b1(a), .b2(
      s_z), .a(1'b0));
  //Delay through complex gates
  assign #`GATE_DELAY s_z2 = s_z1;
  inv0d1 inv(.i(s_z2),.zn(s_z3));

  //Reset gate
  C_AND2A and_reset(.a(s_z2),.b(reset_b),.z(s_z));
  //output
  assign z = s_z3;
endmodule



/*
  C-element, 3 input
    Resets to zero
*/
module C_C3_R0(a,b,c,z,reset_b);
  input a,b,c,reset_b;
  output z;

  wire s_set, s_reset_b;

  an03d1 or3(.a1(a), .a2(b), .a3(c), .z(s_reset_b));
  or03d1 and3(.a1(a),.a2(b), .a3(c), .z(s_set));

  C_C2_R0 c2(.a(s_set),.b(s_reset_b),.z(z), .reset_b(reset_b));


/*  reg z;

always @(reset_b)
  z <= 0;
```

```
always @(a or b or c)
  if(a==b && b==c)
    #`GATE_DELAY z <= a;
*/
endmodule

/*
  C-element, 3 input,
    Resets(low) to zero

    Asymetric, one plus
*/
module C_C3P_R0(a,b,c,z,reset_b);
  input a,b,c,reset_b;
  output z;

  wire s_z, s_z1,s_z2;

  aoi322d1 aoi322d1_i(.zn(s_z1), .c1(a), .c2(b), .c3(c), .b1(a), .b2(
      s_z), .a1(b), .a2(s_z));
  //Delay through complex gates
  assign #`GATE_DELAY s_z2 = s_z1;
  //Reset gate
  C_AND2A adds1(.a(s_z2),.b(reset_b),.z(s_z));
  //output
  TC_INV inv_output(.a(s_z2),.z(z));
endmodule

/*
  Mutex, 2 input
    Reset low
*/
module C_MUTEX2(i_req1,i_req2,o_req1,o_req2);
  input i_req1,i_req2;
  output o_req1, o_req2;


`ifdef SYNTHESIS_ON
  // RTL version
  wire   s_q1,s_q2;
  nd02d0 nand1(.zn(s_q1), .a1(i_req1), .a2(s_q2));
  nd02d0 nand1(.zn(s_q2), .a1(i_req2), .a2(s_q1));
  inv0d1 inv1(.i(s_q1), .zn(o_req1));
  inv0d1 inv2(.i(s_q2), .zn(o_req2));

`else
  // Behavioral version
  reg    o_req1, o_req2;
```

```
  always @(posedge i_req1)
  begin
    if(!o_req2)
      o_req1 <= 1'b1;
  end
  always @(negedge i_req1)
    begin
      o_req1 <= 1'b0;
      if(i_req2)
    o_req2 <= 1'b1;
    end
  always @(posedge i_req2)
    begin
      if(!o_req1 && !i_req1)
    o_req2 <= 1'b1;
    end
  always @(negedge i_req2)
    begin
      o_req2 <= 1'b0;
      if(i_req1)
    o_req1 <= 1'b1;
    end
`endif //SYNTHESIS_ON

endmodule

/*
  Latch
    Active low
    Q output only
    Reset (low)
*/

module C_LATCHQL(d,en,q);
  input d,en;
  output q;

  wire s_q;

  lanlq1 lanlq1_i(.q(s_q), .en(en), .d(d));
  assign #`GATE_DELAY q = s_q;

endmodule

/*
  D-flip flop,
    Positive clock edge
    Q output only
*/
```

```
module C_FD1Q(d,clk,q);
  input d,clk;
  output q;

  wire s_q;

  dfnrq1 dfnrq1_i(.q(s_q), .cp(clk), .d(d));
  assign #`GATE_DELAY q = s_q;

endmodule

/*
    Set/Reset module (active low for both set and reset)
    Q and Q_Bar output
*/
module C_SR(set_b, reset_b, q, q_b);
  input set_b,reset_b;
  output q, q_b;

  wire s_q, s_qb;

  //Using a set/reset flip-flop
  labhb1 labhb1_i(.q(s_q), .qn(s_qb), .e(1'b0), .d(1'b0), .cdn(
      reset_b), .sdn(set_b));
  assign #`GATE_DELAY q = s_q;
  assign #`GATE_DELAY q_b = s_qb;

endmodule

`endif
```

E.2 Networks

E.2.1 Converter

```
/*
  Description:
    Converts a number of enabled routes into a route

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _Converter_v
`define _Converter_v

`include "global.v"
```

```
module Converter(i_enable, i_master, o_enable, o_route);

task integerToRoute;
  input  [3:0] address;
  output [3:0] route;

  begin
    case (address)
      0  : route = `ROUTE_1;
      1  : route = `ROUTE_2;
      2  : route = `ROUTE_3;
      3  : route = `ROUTE_4;
      4  : route = `ROUTE_5;
      5  : route = `ROUTE_6;
      6  : route = `ROUTE_7;
      7  : route = `ROUTE_8;
      8  : route = `ROUTE_9;
      9  : route = `ROUTE_10;
      10 : route = `ROUTE_11;
      11 : route = `ROUTE_12;
      12 : route = `ROUTE_13;
      13 : route = `ROUTE_14;
      14 : route = `ROUTE_15;
      15 : route = `ROUTE_16;
      default: route = 0;
    endcase
//    $display("  integerToRoute: %d %b", address, route);
  end
endtask

parameter N_MC=2;
parameter N_ROUTES=`N_OUTPUTS;

// Inputs
input [N_ROUTES-1:0] i_enable;
input [N_ROUTES-1:0] i_master;
// Outputs
output [N_MC*(`ROUTE_WIDTH+1)-1:0] o_route;
output [N_MC-1:0] o_enable;

reg [N_MC*(`ROUTE_WIDTH+1)-1:0] o_route;
reg [N_MC-1:0] o_enable;

integer i;
always @(i_enable or i_master)
begin
//  $display("Converting: %b", i_enable);
  o_enable = 0;
  o_route = 0;
```

```
  for(i=0;i<N_ROUTES;i=i+1)
  begin
    if(((i_enable>>i)&1'b1)==1'b1)
    begin
      o_route = o_route<<5;
      integerToRoute(i,o_route[4:1]);
      o_route[0] = ((i_master>>i)&1'b1);
      o_enable = (o_enable <<1) | 1'b1;
    end
  end
//  $display("Done Converting: en: %b routes: %b", o_enable, o_route)
    ;
end

//ambit synthesis off

/*
Function which tests that the inputs are never acked at the same time
*/
integer i,count;

always @(i_enable)
begin
  count=0;
  for(i=0;i<N_ROUTES;i=i+1)
    if(((i_enable>>(N_ROUTES-i-1))&1'b1)==1'b1)
      count = count+1;
  if(count>N_MC)
  begin
    `ERROR("ERROR!. To Many mulicasts.")
  end
  //`ERROR('"%d mc",count`)
end

//ambit synthesis on

endmodule

`endif
```

E.2.2 Converter_P2

```
/*
  Description:
    Converts a number of enabled routes into a route

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _Converter_P2_v
`define _Converter_P2_v

`include "global.v"

`define ROUTE_MC_1 5'b10000
`define ROUTE_MC_2 5'b11000
`define ROUTE_MC_3 5'b10100
`define ROUTE_MC_4 5'b11100


module Converter_P2(
  i_enable,
  i_master,
  o_route,
  o_mc_enable, //enable for MC modules
  o_mc_route   //route for MC modules
);

/*
  Parameters
*/
parameter N_MC_BLOCKS=2;     //Number of multicast blocks
parameter N_MC_PR_BLOCK=2; //Multicasts pr multicast block
parameter N_INPUTS=10;
parameter N_OUTPUTS=10;

parameter ROUTE_WIDTH=5+1; //route + master


task integerToRoute;
  input  [4:0] address;
  output [4:0] route;

  begin
    case (address)
      0  : route = `ROUTE_1;
      1  : route = `ROUTE_2;
      2  : route = `ROUTE_3;
      3  : route = `ROUTE_4;
```

```
      4  : route = `ROUTE_5;
      5  : route = `ROUTE_6;
      6  : route = `ROUTE_7;
      7  : route = `ROUTE_8;
      8  : route = `ROUTE_9;
      9  : route = `ROUTE_10;
      10 : route = `ROUTE_11;
      11 : route = `ROUTE_12;
      12 : route = `ROUTE_13;
      13 : route = `ROUTE_14;
      14 : route = `ROUTE_15;
      15 : route = `ROUTE_16;
      default: route = 0;
    endcase
  end
endtask

task integerToMCRoute;
  input  [5:0] address;
  output [5:0] route;

  begin
    case (address)
      0  : route = `ROUTE_MC_1;
      1  : route = `ROUTE_MC_2;
      2  : route = `ROUTE_MC_3;
      3  : route = `ROUTE_MC_4;
      default: route = 0;
    endcase
  end
endtask


// Inputs
input [N_INPUTS*N_OUTPUTS-1:0] i_enable;
input [N_INPUTS*N_OUTPUTS-1:0] i_master;
// Outputs
output [N_INPUTS*ROUTE_WIDTH-1:0] o_route;
output [N_MC_PR_BLOCK*N_MC_BLOCKS*ROUTE_WIDTH-1:0]  o_mc_route;
output [N_MC_BLOCKS*N_MC_PR_BLOCK-1:0] o_mc_enable;

reg [N_INPUTS*ROUTE_WIDTH-1:0] o_route;
reg [N_MC_PR_BLOCK*N_MC_BLOCKS*ROUTE_WIDTH-1:0]  o_mc_route;
reg [N_MC_PR_BLOCK*N_MC_BLOCKS-1:0] o_mc_enable;

//Temporary registers
reg [N_OUTPUTS-1:0] current_enable, current_master;
reg [ROUTE_WIDTH-1:0] tmp_route;
//generate veris
```

```
integer i,j,count;
integer currentMC, currentAddress;

always @(i_enable or i_master)
begin
  //First, reset routes and multicasts
  o_route = 0;
  o_mc_route = 0;
  o_mc_enable=0;
  //Reset multicast counter
  currentMC=0;

  //Loop through all inputs
  for(i=0;i<N_INPUTS;i=i+1)
  begin
    current_enable = i_enable>>(i*N_OUTPUTS);
    current_master = i_master>>(i*N_OUTPUTS);

    //First count the number of destinations for this input
    //to determine if it is a unicast or multicast
    count=0;
    for(j=0;j<N_OUTPUTS;j=j+1)
      if(((current_enable>>j)&1'b1)==1'b1)
        count = count+1;

    /*
      Not used
    */
    if(count==0)
    begin
    end else
    /*
      Unicast
    */
    if(count==1)
    begin
      for(j=0;j<N_OUTPUTS;j=j+1)
        if(((current_enable>>j)&1'b1)==1'b1)
        begin
          integerToRoute(j,tmp_route[ROUTE_WIDTH-2:1]);
          tmp_route[ROUTE_WIDTH-1] = 1'b0;//No multicast
          tmp_route[0] = ((current_master>>j)&1'b1);
          o_route = o_route | (tmp_route << i*ROUTE_WIDTH);
        end
    end
    else
    /*
      Multicast
    */
```

```
    if(count==2)
     begin
       //First, send the packet to the MC block
       integerToMCRoute(currentMC,tmp_route[ROUTE_WIDTH-1:1]);
       tmp_route[0] = 0; //does not matter
       o_route = o_route | (tmp_route << i*ROUTE_WIDTH);

       currentAddress=0;
       //Second, Set up the block to the 2 addresses
       for(j=0;j<N_OUTPUTS;j=j+1)
         if(((current_enable>>j)&1'b1)==1'b1)
         begin
           integerToRoute(j,tmp_route[ROUTE_WIDTH-1:2]);
           tmp_route[1] = 0; //don't care
           tmp_route[0] = ((current_master>>j)&1'b1);
           //Setup the route
          o_mc_enable = o_mc_enable | (1'b1<<(currentAddress+
             N_MC_PR_BLOCK*currentMC));
           o_mc_route = o_mc_route | (tmp_route<<(ROUTE_WIDTH*(
              currentAddress+N_MC_PR_BLOCK*currentMC)));
           //Advance current address
           currentAddress = currentAddress+1;
         end
       //Advance currentMC, such that next multicast uses the next MC
          block
       currentMC=currentMC+1;
     end else
    begin
       `ERROR("ERROR!. To Many mulicasts. Only 2 allowed")
     end
   end
end

endmodule

`endif
```

E.2.3 NoC

```
/*
  Description:
    NoC.

    Parallel bundled data version
    Topology: Binary tree

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _NoC_v
`define _NoC_v

`include "global.v"
`timescale 1ns/1ps


module NoC(
  i_data,
  i_valid,
  i_conf_enable,
  i_conf_master,
  o_data,
  o_master,
  o_valid,
  i_clk,
  i_reset_b);

/*
  Parameters
*/
parameter INPUTS=`N_INPUTS;
parameter OUTPUTS=`N_OUTPUTS;
parameter DATA_WIDTH=`DATA_WIDTH;


//This is not really a parameters.. +1 due to master4 bit
parameter ROUTE_WIDTH=`ROUTE_WIDTH+1;
parameter BUS_WIDTH=DATA_WIDTH+ROUTE_WIDTH;

//TBD. Due to modelsim crash
parameter ROUTE_WIDTH2=5;

/*
  Inputs
*/
input [INPUTS*DATA_WIDTH-1:0] i_data;
```

```
input [INPUTS-1:0] i_valid;
input [INPUTS*OUTPUTS-1:0] i_conf_enable,i_conf_master;
input i_clk, i_reset_b;

/*
  Outputs
*/
output [OUTPUTS*DATA_WIDTH-1:0] o_data;
output [OUTPUTS-1:0] o_master;
output [OUTPUTS-1:0] o_valid;

/*
  Internal signals
*/
wire [INPUTS*BUS_WIDTH-1:0] si_data;
wire [OUTPUTS*BUS_WIDTH-1:0] so_data;
wire [INPUTS-1:0] si_req,so_ack;
wire [OUTPUTS-1:0] so_req,si_ack;
wire [INPUTS*ROUTE_WIDTH-1:0] s_route;
wire [INPUTS-1:0] s_route_ack, s_route_req;
wire [INPUTS-1:0] s_req,s_ack;


/*
  Netlist
*/
//The network
P_network network(si_data, si_req, so_ack, so_data, so_req, si_ack,
    i_reset_b);
defparam network.INPUTS=INPUTS;
defparam network.OUTPUTS=OUTPUTS;
defparam network.BUS_WIDTH=BUS_WIDTH; //width+Master
defparam network.DATA_WIDTH=DATA_WIDTH+1;

//The Network adapters
genvar i;
generate
    for(i=0;i<INPUTS;i=i+1)
    begin : NA_generation
      NA naAdapter (
         .i_data(i_data[(1+i)*DATA_WIDTH-1:i*DATA_WIDTH]),
         .i_valid(i_valid[i]),
         .o_data(si_data[(1+i)*BUS_WIDTH-1:i*BUS_WIDTH]),
         .o_req(si_req[i]),
         .i_ack(so_ack[i]),
         .o_route_req(s_route_req[i]),
         .i_route(s_route[(1+i)*ROUTE_WIDTH-1:i*ROUTE_WIDTH]),
         .i_req(s_req[i]),
         .o_ack(s_ack[i]),
```

```
        .i_route_ack(s_route_ack[i]),
        .i_clk(i_clk),
        .i_reset_b(i_reset_b));
        defparam naAdapter.DATA_WIDTH=DATA_WIDTH;
        defparam naAdapter.BUS_WIDTH=BUS_WIDTH;
        defparam naAdapter.ROUTE_WIDTH=ROUTE_WIDTH;

      //AM_unicast am_unicast(s_route1[0:3], s_route_req[i], s_ack[i
          ],s_route2[i*'ROUTE_WIDTH:(1+i)*'ROUTE_WIDTH-1], s_req[i],
          s_route_ack[i],i_reset_b );
       AM_multicast am_multicast(
        .i_conf_enable(i_conf_enable[(i+1)*OUTPUTS-1:i*OUTPUTS]),
        .i_conf_master(i_conf_master[(i+1)*OUTPUTS-1:i*OUTPUTS]),
        .i_req(s_route_req[i]),
        .i_ack(s_ack[i]),
        .o_data(s_route[(1+i)*ROUTE_WIDTH2-1:i*ROUTE_WIDTH2]),
        .o_req(s_req[i]),
        .o_ack(s_route_ack[i]),
        .i_reset_b(i_reset_b));
        defparam am_multicast.DATA_WIDTH=ROUTE_WIDTH;
    end
    for(i=0;i<OUTPUTS;i=i+1)
    begin : AN_generation
      AN anAdapter (
        .i_data(so_data[(i+1)*BUS_WIDTH-ROUTE_WIDTH:i*BUS_WIDTH]),
        .i_req(so_req[i]),
        .o_ack(si_ack[i]),
        .o_data(o_data[(1+i)*DATA_WIDTH-1:i*DATA_WIDTH]),
        .o_master(o_master[i]),
        .o_valid(o_valid[i]),
        .i_clk(i_clk),
        .i_reset_b(i_reset_b));
      defparam anAdapter.SIZE=DATA_WIDTH+1;

    end
endgenerate

/*
Function which tests that the inputs are never acked at the same time
*/

'ifdef ERROR_CHECKING
integer l,count;
always @(so_req)
begin
  count=0;
  for(l=0;l<OUTPUTS;l=l+1)
    if(so_req[l])
      count = count+1;
```

```
    if(count>1)
    begin
      $display ("NoC.v: ERROR!. two request at the same time. %d %b\n",
          count,so_req);
      $stop;
    end
end
`endif
endmodule

`endif
```

E.2.4 NoC_P2

```
/*
  Description:
    NoC.

    Parallel bundled data version
    Topology: Binary tree

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _NoC_P2_v
`define _NoC_P2_v

`include "global.v"
`timescale 1ns/1ps


module NoC_P2(
  i_data,
  i_valid,
  i_conf_enable,
  i_conf_master,
  o_data,
  o_master,
  o_valid,
  i_clk,
  i_reset_b);

/*
  Parameters
*/
parameter INPUTS=`N_INPUTS;
parameter OUTPUTS=`N_OUTPUTS;
parameter DATA_WIDTH=`DATA_WIDTH;

parameter N_MC_BLOCKS=2;
parameter N_MC_PR_BLOCK=2;


//This is not really a parameters.. 5 bits for route and 1 for
    matster bit
parameter ROUTE_WIDTH=5+1;
parameter BUS_WIDTH=DATA_WIDTH+ROUTE_WIDTH;

//TBD. Due to modelsim crash. It is really annoying, but this seems
    to get around it
parameter ROUTE_WIDTH2=(ROUTE_WIDTH+1-1);
```

```
parameter DATA_WIDTH2=(DATA_WIDTH+1-1);
parameter BUS_WIDTH2=(BUS_WIDTH+1-1);


/*
  Inputs
*/
input [INPUTS*DATA_WIDTH-1:0] i_data;
input [INPUTS-1:0] i_valid;
input [INPUTS*OUTPUTS-1:0] i_conf_enable,i_conf_master;
input i_clk, i_reset_b;

/*
  Outputs
*/
output [OUTPUTS*DATA_WIDTH-1:0] o_data;
output [OUTPUTS-1:0] o_master;
output [OUTPUTS-1:0] o_valid;

/*
  Generate variables
*/
genvar i;

/*
  Internal signals
*/
wire [INPUTS*BUS_WIDTH-1:0] si_data;
wire [OUTPUTS*BUS_WIDTH-1:0] so_data;
wire [INPUTS-1:0] si_req,so_ack;
wire [OUTPUTS-1:0] so_req,si_ack;
wire [INPUTS*ROUTE_WIDTH-1:0] s_route;
wire [INPUTS-1:0] s_route_ack, s_route_req;
wire [INPUTS-1:0] s_req,s_ack;


/*
  Netlist
*/


/*
  Wires
*/
wire s_ack_l1, s_ack_l2, s_ack_l3;
wire s_req_l1, s_req_l2, s_req_l3;
wire [BUS_WIDTH-1:0] s_data_l1, s_data_l2, s_data_l3;
wire s_ack_r1, s_ack_r2, s_ack_r3;
wire s_req_r1, s_req_r2, s_req_r3;
```

```
wire [BUS_WIDTH-1:0] s_data_r1, s_data_r2, s_data_r3;
wire [N_MC_BLOCKS*N_MC_PR_BLOCK*ROUTE_WIDTH-1:0]  s_mc_route;
wire [N_MC_BLOCKS*N_MC_PR_BLOCK-1:0] s_mc_enable;

/*
  Converter, which converts configuration matrix into signals for
  Multicastblocks and Network adapters
*/
//parameter N_MC_BLOCKS=2;     //Number of multicast blocks
//parameter N_MC_PR_BLOCK=2; //Multicasts pr multicast block
//parameter N_INPUTS=10;
//parameter N_OUTPUTS=10;

Converter_P2 #(N_MC_BLOCKS,N_MC_PR_BLOCK,INPUTS,OUTPUTS) converter(
  .i_enable(i_conf_enable),
  .i_master(i_conf_master),
  .o_route(s_route),
  .o_mc_enable(s_mc_enable),
  .o_mc_route(s_mc_route)
);

P_merge_tree #(INPUTS, BUS_WIDTH) merge_tree (
  .i_data(si_data),
  .i_req(si_req),
  .o_ack(so_ack),
  .o_data(s_data_l1),
  .o_req(s_req_l1),
  .i_ack(s_ack_l1),
  .i_reset_b(i_reset_b));

P_router #(BUS_WIDTH, (DATA_WIDTH+1)) router(
  .i_data(s_data_l1),
  .i_req(s_req_l1),
  .o_ack(s_ack_l1),
  .o_data1(s_data_l2),
  .o_req1(s_req_l2),
  .i_ack1(s_ack_l2),
  .o_data2(s_data_l3),
  .o_req2(s_req_l3),
  .i_ack2(s_ack_l3),
  .i_reset_b(i_reset_b)
  );

//Connect the lower part directly
assign s_data_r2 =s_data_l2;
assign s_req_r2 =s_req_l2;
assign s_ack_l2 =s_ack_r2;

//wires
```

```
wire [N_MC_BLOCKS-1:0] s_ack_mc1, s_ack_mc2;
wire [N_MC_BLOCKS-1:0] s_req_mc1, s_req_mc2;
wire [N_MC_BLOCKS*BUS_WIDTH-1:0] s_data_mc1, s_data_mc2;
//Multicast router tree
P_router_tree #(N_MC_BLOCKS, BUS_WIDTH, (DATA_WIDTH+1))
    multicast_router_tree (
  .i_data(s_data_l3),
  .i_req(s_req_l3),
  .o_ack(s_ack_l3),
  .o_data(s_data_mc1),
  .o_req(s_req_mc1),
  .i_ack(s_ack_mc1),
  .i_reset_b(i_reset_b));

//Generate and connect the multicast blocks
generate
    for(i=0;i<N_MC_BLOCKS;i=i+1)
    begin : multicast_block_generation
        P_multicast p_multicast(
          .i_routes(s_mc_route[(i+1)*N_MC_PR_BLOCK*ROUTE_WIDTH-1:i*
              N_MC_PR_BLOCK*ROUTE_WIDTH]),
          .i_route_en(s_mc_enable[(i+1)*N_MC_PR_BLOCK-1:i*
              N_MC_PR_BLOCK]),
          .i_data(s_data_mc1[i*BUS_WIDTH+DATA_WIDTH-1:i*BUS_WIDTH]),
          .i_req(s_req_mc1[i]),
          .o_ack(s_ack_mc1[i]),
          .o_data(s_data_mc2[(i+1)*BUS_WIDTH2-1:i*BUS_WIDTH2]),
          .o_req(s_req_mc2[i]),
          .i_ack(s_ack_mc2[i]),
          .i_reset_b(i_reset_b));
        defparam p_multicast.N_MC=N_MC_PR_BLOCK;
        defparam p_multicast.DATA_WIDTH=DATA_WIDTH;
        defparam p_multicast.BUS_WIDTH=BUS_WIDTH;
        defparam p_multicast.ROUTE_WIDTH=ROUTE_WIDTH;
    end
endgenerate

//Multicast merge tree
  P_merge_tree #(N_MC_BLOCKS, BUS_WIDTH) multicast_merge_tree (
  .i_data(s_data_mc2),
  .i_req(s_req_mc2),
  .o_ack(s_ack_mc2),
  .o_data(s_data_r3),
  .o_req(s_req_r3),
  .i_ack(s_ack_r3),
  .i_reset_b(i_reset_b));

P_merge merger (
  .i_data1(s_data_r3),
```

```
  .i_req1(s_req_r3),
  .o_ack1(s_ack_r3),
  .i_data2(s_data_r2),
  .i_req2(s_req_r2),
  .o_ack2(s_ack_r2),
  .o_data(s_data_r1),
  .o_req(s_req_r1),
  .i_ack(s_ack_r1),
  .i_reset_b(i_reset_b)
  );
defparam merger.BUS_WIDTH=BUS_WIDTH;

P_router_tree #(OUTPUTS, BUS_WIDTH, (DATA_WIDTH+1)) router_tree (
  .i_data(s_data_r1),
  .i_req(s_req_r1),
  .o_ack(s_ack_r1),
  .o_data(so_data),
  .o_req(so_req),
  .i_ack(si_ack),
  .i_reset_b(i_reset_b));

//The Network adapters
generate
    for(i=0;i<INPUTS;i=i+1)
    begin : NA_generation
      NA naAdapter (
         .i_data(i_data[(1+i)*DATA_WIDTH-1:i*DATA_WIDTH]),
         .i_valid(i_valid[i]),
         .o_data(si_data[(1+i)*BUS_WIDTH-1:i*BUS_WIDTH]),
         .o_req(si_req[i]),
         .i_ack(so_ack[i]),
         .o_route_req(s_route_req[i]),
         .i_route(s_route[(1+i)*ROUTE_WIDTH-1:i*ROUTE_WIDTH]),
         .i_req(s_req[i]),
         .o_ack(s_ack[i]),
         .i_route_ack(s_route_ack[i]),
         .i_clk(i_clk),
         .i_reset_b(i_reset_b));
         defparam naAdapter.DATA_WIDTH=DATA_WIDTH;
         defparam naAdapter.BUS_WIDTH=BUS_WIDTH;
         defparam naAdapter.ROUTE_WIDTH=ROUTE_WIDTH;

         AM_unicast am_unicast(
         .i_req(s_route_req[i]),
         .i_ack(s_ack[i]),
         .o_req(s_req[i]),
         .o_ack(s_route_ack[i]),
         .i_reset_b(i_reset_b));
```

```
      end
    //AN adapters
    for(i=0;i<OUTPUTS;i=i+1)
    begin : AN_generation
      AN anAdapter (
        .i_data(so_data[(i+1)*BUS_WIDTH-ROUTE_WIDTH:i*BUS_WIDTH]),
        .i_req(so_req[i]),
        .o_ack(si_ack[i]),
        .o_data(o_data[(1+i)*DATA_WIDTH-1:i*DATA_WIDTH]),
        .o_master(o_master[i]),
        .o_valid(o_valid[i]),
        .i_clk(i_clk),
        .i_reset_b(i_reset_b));
      defparam anAdapter.SIZE=DATA_WIDTH+1;

    end
endgenerate

/*
Function which tests that the inputs are never acked at the same time
*/

`ifdef ERROR_CHECKING
integer l,count;
always @(so_req)
begin
  count=0;
  for(l=0;l<OUTPUTS;l=l+1)
    if(so_req[l])
      count = count+1;

  if(count>1)
  begin
    $display ("NoC.v: ERROR!. two request at the same time. %d %b\n",
        count,so_req);
    $stop;
  end
end
`endif
endmodule

`endif
```

E.2.5 NoC_S1

```
/*
  Description:
    NoC

    Serial version
    Topology: Binary tree

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _NoC_S1_v
`define _NoC_S1_v

`include "global.v"
`timescale 1ns/1ps


module NoC_S1(
  i_data,
  i_valid,
  i_conf_enable,
  i_conf_master,
  o_data,
  o_master,
  o_valid,
  i_clk,
  i_reset_b);

/*
  Parameters
*/
parameter INPUTS=`N_INPUTS;
parameter OUTPUTS=`N_OUTPUTS;
parameter DATA_WIDTH=`DATA_WIDTH;


//This is not really a parameters.. +1 due to master4 bit
parameter ROUTE_WIDTH=`ROUTE_WIDTH+1;
parameter BUS_WIDTH=DATA_WIDTH+ROUTE_WIDTH*2;

//TBD. Due to modelsim crash
parameter ROUTE_WIDTH2=5;

/*
  Inputs
*/
input [INPUTS*DATA_WIDTH-1:0] i_data;
```

```
input [INPUTS-1:0] i_valid;
input [INPUTS*OUTPUTS-1:0] i_conf_enable,i_conf_master;
input i_clk, i_reset_b;

/*
  Outputs
*/
output [OUTPUTS*DATA_WIDTH-1:0] o_data;
output [OUTPUTS-1:0] o_master;
output [OUTPUTS-1:0] o_valid;

/*
  Internal signals
*/
wire [INPUTS*BUS_WIDTH-1:0] si_data;
wire [OUTPUTS*(DATA_WIDTH+2)-1:0] so_data;
wire [INPUTS-1:0] si_req,so_ack;
wire [OUTPUTS-1:0] so_req,si_ack;
wire [INPUTS*ROUTE_WIDTH-1:0] s_route;
wire [INPUTS-1:0] s_route_ack, s_route_req;
wire [INPUTS-1:0] s_req,s_ack;


/*
  Netlist
*/
//The network
S_network_with_converters network(si_data, si_req, so_ack, so_data,
   so_req, si_ack, i_reset_b);
defparam network.INPUTS=INPUTS;
defparam network.OUTPUTS=OUTPUTS;
defparam network.BUS_WIDTH=BUS_WIDTH; //width+Master
defparam network.BUS_WIDTH_OUT=DATA_WIDTH+2;

//The Network adapters
genvar i;
generate
    for(i=0;i<INPUTS;i=i+1)
    begin : NA_generation
      NA naAdapter (
         .i_data(i_data[(1+i)*DATA_WIDTH-1:i*DATA_WIDTH]),
         .i_valid(i_valid[i]),
         .o_data(si_data[(1+i)*BUS_WIDTH-1:i*BUS_WIDTH]),
         .o_req(si_req[i]),
         .i_ack(so_ack[i]),
         .o_route_req(s_route_req[i]),
         .i_route({
               //The route
           s_route[(1+i)*ROUTE_WIDTH-1],1'b0,
```

```
            s_route[(1+i)*ROUTE_WIDTH-2],1'b0,
            s_route[(1+i)*ROUTE_WIDTH-3],1'b0,
            s_route[(1+i)*ROUTE_WIDTH-4],1'b0,
            1'b0,s_route[(1+i)*ROUTE_WIDTH-5]
            }),
          .i_req(s_req[i]),
          .o_ack(s_ack[i]),
          .i_route_ack(s_route_ack[i]),
          .i_clk(i_clk),
          .i_reset_b(i_reset_b));
          defparam naAdapter.DATA_WIDTH=DATA_WIDTH;
          defparam naAdapter.BUS_WIDTH=BUS_WIDTH;
          defparam naAdapter.ROUTE_WIDTH=ROUTE_WIDTH*2;

        AM_multicast am_multicast(
          .i_conf_enable(i_conf_enable[(i+1)*OUTPUTS-1:i*OUTPUTS]),
          .i_conf_master(i_conf_master[(i+1)*OUTPUTS-1:i*OUTPUTS]),
          .i_req(s_route_req[i]),
          .i_ack(s_ack[i]),
          .o_data(s_route[(1+i)*ROUTE_WIDTH2-1:i*ROUTE_WIDTH2]),
          .o_req(s_req[i]),
          .o_ack(s_route_ack[i]),
          .i_reset_b(i_reset_b));
          defparam am_multicast.DATA_WIDTH=ROUTE_WIDTH;
      end
      for(i=0;i<OUTPUTS;i=i+1)
      begin : AN_generation
        AN anAdapter (
          .i_data(so_data[(i+1)*(DATA_WIDTH+2)-2:i*(DATA_WIDTH+2)]),
          .i_req(so_req[i]),
          .o_ack(si_ack[i]),
          .o_data(o_data[(1+i)*DATA_WIDTH-1:i*DATA_WIDTH]),
          .o_master(o_master[i]),
          .o_valid(o_valid[i]),
          .i_clk(i_clk),
          .i_reset_b(i_reset_b));
         defparam anAdapter.SIZE=DATA_WIDTH+1;

      end
endgenerate

endmodule

module S_network_with_converters(i_data,i_req,o_ack,o_data,o_req,
    i_ack,i_reset_b);

parameter INPUTS = `N_INPUTS;
parameter OUTPUTS = `N_OUTPUTS;
parameter BUS_WIDTH = `BUS_WIDTH;
```

```
parameter BUS_WIDTH_OUT = `DATA_WIDTH;

// Inputs
input [INPUTS*BUS_WIDTH-1:0] i_data;
input [INPUTS-1:0] i_req;
input [OUTPUTS-1:0] i_ack;
input i_reset_b;

// Outputs
output [INPUTS-1:0] o_ack;
output [OUTPUTS*BUS_WIDTH_OUT-1:0] o_data;
output [OUTPUTS-1:0] o_req;

//internal signals
wire [INPUTS*4-1:0] si_data;
wire [INPUTS-1:0] si_eop;
wire [OUTPUTS-1:0] si_ack;

wire [INPUTS-1:0] so_ack;
wire [OUTPUTS*4-1:0] so_data;
wire [OUTPUTS-1:0] so_eop;

//network
S_network #(INPUTS, OUTPUTS, 4) net(si_data, si_eop, so_ack, so_data,
     so_eop, si_ack, i_reset_b);

//converters
conv_PtoS #(BUS_WIDTH) conv_PtoS_inst[INPUTS-1:0](.i_req(i_req), .
    i_ack(so_ack), .i_data(i_data), .o_ack(o_ack), .o_data(si_data), .
    o_eop(si_eop), .i_reset_b(i_reset_b));
/*conv_StoP #(BUS_WIDTH_OUT) conv_StoP_inst[OUTPUTS-1:0](.i_data(
    so_data), .i_eop(so_eop), .o_ack(si_ack), .o_req(o_req), .i_ack(
    i_ack), .o_data(o_data), .i_reset_b(i_reset_b));
defparam conv_StoP_inst[0].REMOVE_TRAILING_PACKET = 1;
defparam conv_StoP_inst[3].REMOVE_TRAILING_PACKET = 1;
defparam conv_StoP_inst[6].REMOVE_TRAILING_PACKET = 1;
defparam conv_StoP_inst[9].REMOVE_TRAILING_PACKET = 1;
*/
//0
conv_StoP #(BUS_WIDTH_OUT,1) conv_StoP_inst0(.i_data(so_data
    [1*4-1:0*4]), .i_eop(so_eop[0]), .o_ack(si_ack[0]), .o_req(o_req
    [0]),
                                .i_ack(i_ack[0]), .o_data(o_data[1*
                                    BUS_WIDTH_OUT-1:0*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//1
conv_StoP #(BUS_WIDTH_OUT,0) conv_StoP_inst1(.i_data(so_data
    [2*4-1:1*4]), .i_eop(so_eop[1]), .o_ack(si_ack[1]), .o_req(o_req
    [1]),
```

```
                                .i_ack(i_ack[1]), .o_data(o_data[2*
                                    BUS_WIDTH_OUT-1:1*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//2
conv_StoP #(BUS_WIDTH_OUT,0) conv_StoP_inst2(.i_data(so_data
    [3*4-1:2*4]), .i_eop(so_eop[2]), .o_ack(si_ack[2]), .o_req(o_req
    [2]),
                                .i_ack(i_ack[2]), .o_data(o_data[3*
                                    BUS_WIDTH_OUT-1:2*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//3
conv_StoP #(BUS_WIDTH_OUT,1) conv_StoP_inst3(.i_data(so_data
    [4*4-1:3*4]), .i_eop(so_eop[3]), .o_ack(si_ack[3]), .o_req(o_req
    [3]),
                                .i_ack(i_ack[3]), .o_data(o_data[4*
                                    BUS_WIDTH_OUT-1:3*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//4
conv_StoP #(BUS_WIDTH_OUT,0) conv_StoP_inst4(.i_data(so_data
    [5*4-1:4*4]), .i_eop(so_eop[4]), .o_ack(si_ack[4]), .o_req(o_req
    [4]),
                                .i_ack(i_ack[4]), .o_data(o_data[5*
                                    BUS_WIDTH_OUT-1:4*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//5
conv_StoP #(BUS_WIDTH_OUT,0) conv_StoP_inst5(.i_data(so_data
    [6*4-1:5*4]), .i_eop(so_eop[5]), .o_ack(si_ack[5]), .o_req(o_req
    [5]),
                                .i_ack(i_ack[5]), .o_data(o_data[6*
                                    BUS_WIDTH_OUT-1:5*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//6
conv_StoP #(BUS_WIDTH_OUT,1) conv_StoP_inst6(.i_data(so_data
    [7*4-1:6*4]), .i_eop(so_eop[6]), .o_ack(si_ack[6]), .o_req(o_req
    [6]),
                                .i_ack(i_ack[6]), .o_data(o_data[7*
                                    BUS_WIDTH_OUT-1:6*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//7
conv_StoP #(BUS_WIDTH_OUT,0) conv_StoP_inst7(.i_data(so_data
    [8*4-1:7*4]), .i_eop(so_eop[7]), .o_ack(si_ack[7]), .o_req(o_req
    [7]),
                                .i_ack(i_ack[7]), .o_data(o_data[8*
                                    BUS_WIDTH_OUT-1:7*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//8
conv_StoP #(BUS_WIDTH_OUT,0) conv_StoP_inst8(.i_data(so_data
    [9*4-1:8*4]), .i_eop(so_eop[8]), .o_ack(si_ack[8]), .o_req(o_req
    [8]),
```

```
                                .i_ack(i_ack[8]), .o_data(o_data[9*
                                    BUS_WIDTH_OUT-1:8*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//9
conv_StoP #(BUS_WIDTH_OUT,1) conv_StoP_inst9(.i_data(so_data
    [10*4-1:9*4]), .i_eop(so_eop[9]), .o_ack(si_ack[9]), .o_req(o_req
    [9]),
                                .i_ack(i_ack[9]), .o_data(o_data[10*
                                    BUS_WIDTH_OUT-1:9*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//10
conv_StoP #(BUS_WIDTH_OUT,0) conv_StoP_inst10(.i_data(so_data
    [11*4-1:10*4]), .i_eop(so_eop[10]), .o_ack(si_ack[10]), .o_req(
    o_req[10]),
                                .i_ack(i_ack[10]), .o_data(o_data[11*
                                    BUS_WIDTH_OUT-1:10*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));
//11
conv_StoP #(BUS_WIDTH_OUT,0) conv_StoP_inst11(.i_data(so_data
    [12*4-1:11*4]), .i_eop(so_eop[11]), .o_ack(si_ack[11]), .o_req(
    o_req[11]),
                                .i_ack(i_ack[11]), .o_data(o_data[12*
                                    BUS_WIDTH_OUT-1:11*BUS_WIDTH_OUT]), .
                                    i_reset_b(i_reset_b));


endmodule


`endif
```

E.3 Common blocks

E.3.1 global.v

```
/*
  Description:
    global definitions

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/

`ifndef _global_v
`define _global_v

//Synthesis or simulation
`define SYNTHESIS_ON
//ambit synthesis off
  `undef SYNTHESIS_ON
//ambit synthesis on

//`define NOC_INTEGRATED

//`define NoC_Instance NoC
//`define NoC_Instance NoC_P2
`define NoC_Instance NoC_S1

//Debug levels
`define DEBUG_LEVEL1
`define DEBUG_LEVEL2
//`define DEBUG_LEVEL3
//Error checking
//`define ERROR_CHECKING

//Data width definitions
`define DATA_WIDTH 22
`define ROUTE_WIDTH 4
`define BUS_WIDTH (`DATA_WIDTH+`ROUTE_WIDTH)

//These number are seen from the network!..
`define N_INPUTS 16
`define N_OUTPUTS 12

//Route to the different output blocks
//Should match the current network
`define ROUTE_1  4'b0000
`define ROUTE_2  4'b0010
`define ROUTE_3  4'b0011
`define ROUTE_4  4'b0100
```

```
`define ROUTE_5  4'b0110
`define ROUTE_6  4'b0111
`define ROUTE_7  4'b1000
`define ROUTE_8  4'b1010
`define ROUTE_9  4'b1011
`define ROUTE_10 4'b1100
`define ROUTE_11 4'b1110
`define ROUTE_12 4'b1111
//Not used
`define ROUTE_13 4'b1100
`define ROUTE_14 4'b1101
`define ROUTE_15 4'b1110
`define ROUTE_16 4'b1111

/*`define ROUTE_1 4'b0000
`define ROUTE_2 4'b0001
`define ROUTE_3 4'b0010
`define ROUTE_4 4'b0011
`define ROUTE_5 4'b0100
`define ROUTE_6 4'b0101
`define ROUTE_7 4'b0110
`define ROUTE_8 4'b0111
`define ROUTE_9 4'b1000
`define ROUTE_10 4'b1001
`define ROUTE_11 4'b1010
`define ROUTE_12 4'b1011
`define ROUTE_13 4'b1100
`define ROUTE_14 4'b1101
`define ROUTE_15 4'b1110
`define ROUTE_16 4'b1111
*/

`define ROUTE_5_2 8'b10001000

//Convert
`define CONV_1of4_to_2(dat) ({dat[2]|dat[3], dat[1]|dat[3]})
`define CONV_2_to_1of4(dat) ({dat[1]&dat[0], dat[1]&~dat[0], ~dat[1]&
    dat[0], ~dat[1]&~dat[0]})

`define INIT_TESTBENCH  $timeformat(-9, 10, " ns", 10);
`define TIMESCALE `timescale 1ns/1ps
`define ERROR(s) $display("ERROR:"); $display(s); $stop;

`define CHECK(o,val) if(o!=val) begin $display("Value is %b, but
    should be",o, val); $stop; end
`define CHECK_0(o) `CHECK(o,1'b0)
`define CHECK_1(o) `CHECK(o,1'b1)

`endif
```

E.3.2 AM_multicast

```
/*
  Description:
    Multicast module for the "Address Manager"

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/

`ifndef _AM_Multicast_v
`define _AM_Multicast_v

`include "global.v"

module AM_multicast(
  i_conf_enable,
  i_conf_master,
  i_req,
  i_ack,
  o_data,
  o_req,
  o_ack,
  i_reset_b);
/*
  Parameters
*/
parameter N_ROUTES=`N_OUTPUTS;
parameter N_MC=2;
parameter DATA_WIDTH=5;

/*
  Inputs
*/
input i_req, i_ack, i_reset_b;
input [N_ROUTES-1:0] i_conf_enable;
input [N_ROUTES-1:0] i_conf_master;

/*
  Outputs
*/
output o_req,o_ack;
output [DATA_WIDTH-1:0] o_data;

/*
  Internal signals
*/
wire [N_MC-1:0] s_en;
wire [N_MC*DATA_WIDTH-1:0] s_data;
```

```
/*
  Netlist
*/
Multicaster #(N_MC, DATA_WIDTH) multicasters(
  s_en,
  s_data,
  i_req,
  i_ack,
  o_data,
  o_req,
  o_ack,
  i_reset_b);

Converter #(N_MC) converter(
  .i_enable(i_conf_enable),
  .i_master(i_conf_master),
  .o_enable(s_en),
  .o_route(s_data));

endmodule

`endif
```

E.3.3 AM_unicast

```
/*
  Description:
    Unicast module for the "Address Manager"

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/

`ifndef _AM_unicast_v
`define _AM_unicast_v

`include "global.v"

module AM_unicast(
  i_req,
  i_ack,
  o_req,
  o_ack,
  i_reset_b
  );
/*
 Inputs
*/
input i_req, i_ack, i_reset_b;
/*
 Outputs
*/
output o_req,o_ack;

/*
 Netlist
*/
assign o_req = i_req;
assign o_ack=i_ack;

endmodule

`endif
```

E.3.4 AN

```
/*
  Description:
    AN, Network adapter

    Receives a packet using a 4-phase bundled data protocol and
    outputs data using the Lego2 protocol.

    Takes at least 4 clock cycles for the handshake to complete. 2
       for
    rising edge of request and 2 for falling edge.

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _AN_v
`define _AN_v

`include "global.v"

module AN(
  i_data,
  i_req,
  o_ack,
  o_data,
  o_valid,
  o_master,
  i_clk,
  i_reset_b);

/*
  Parameters
*/
parameter SIZE=`DATA_WIDTH; //Datawidth (including master)

/*
  Inputs
*/
input [SIZE-1:0] i_data;
input i_req,i_clk,i_reset_b;

/*
  Outputs
*/
output [SIZE-2:0] o_data;
output o_valid, o_ack, o_master;

/*
```

```
  Wires
*/
wire s_valid1,s_valid2,s_valid3;

/*
  Netlist
*/
//Data

//C_LATCHQL data_latches[SIZE-1:0] (.d(i_data),.en(s_valid2), .q({
    o_master,o_data}));
C_FD1Q data_latches[SIZE-1:0] (.d(i_data),.clk(i_req), .q({o_master,
    o_data}));

// Ack when request arrives
//assign o_ack = s_valid2;
  C_AND2 ack_reset(.a(i_reset_b),.b(s_valid2),.z(o_ack));
//assign o_ack = i_req;

//Synchronize valid token on clock signal
C_FD1Q sync1 (i_req, i_clk, s_valid1);
C_FD1Q sync2 (s_valid1, i_clk, s_valid2);
C_FD1Q sync3 (s_valid2, i_clk, s_valid3);
TC_AND2A #4 oneshoot (s_valid3, s_valid2,o_valid);


/*always @(posedge o_valid)
begin
  $display ("AN!. master: %b - %d",o_master,SIZE);
end
*/
endmodule

`endif
```

E.3.5 de_serializer

```
/*
  Description:
    de-Serializes a packet which consits of flits of 2 bits into
    a single packet

    Both input and output uses a 4-phase bundled data protocol

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/
`ifndef _de_serializer_v
`define _de_serializer_v

`include "global.v"


module de_serializer(
  i_data,
  i_eop,
  i_req,
  o_ack,
  o_req,
  i_ack,
  o_data,
  i_reset_b);
    /*
      parameters
    */
    parameter BUS_WIDTH=`BUS_WIDTH;
    parameter REMOVE_TRAILING_FLITS=0;
    //not actual parameters
    parameter LINE_SIZE=2;
    parameter N_LINES=(BUS_WIDTH/LINE_SIZE);

    /*
      Inputs
    */
    input i_ack, i_reset_b,i_eop;
    input i_req;
    input [1:0] i_data;

    /*
      Output
    */
    output o_ack,o_req;
    output [BUS_WIDTH-1:0] o_data;
```

```
    /*
      Internal signals
    */
    wire [1:0] s_data;
    wire [N_LINES+REMOVE_TRAILING_FLITS-1:0] s_ctrl;
    wire s_req, so_ack, so_ack2;

    /*
      Netlist
    */
    assign s_data=i_data;
    assign o_ack = so_ack2;
    assign s_req = i_req;

    de_serialize_controller #(N_LINES+REMOVE_TRAILING_FLITS)
       controller(
      .i_req(s_req),
      .o_ctrl(s_ctrl),
      .i_reset_b(i_reset_b));

    assign o_req = i_eop;

    C_ORx  #(N_LINES+REMOVE_TRAILING_FLITS+1) or_three(
    .inputs({s_ctrl[N_LINES+REMOVE_TRAILING_FLITS-1:0],i_ack}),
    .z(so_ack2));

    genvar i;
    generate
      for(i=0; i<N_LINES;i=i+1)
      begin : gen_con
        C_FD1Q ff_array[1:0](.d(s_data),.clk(s_ctrl[
            REMOVE_TRAILING_FLITS+N_LINES-1-i]),.q(o_data[i*2+1:i*2]))
            ;
      end
    endgenerate

endmodule

/*
  de_serialize_controller
*/
module de_serialize_controller(
  i_req,
  o_ctrl,
  i_reset_b);
    /*
     Parameters
    */
    parameter SIZE=2;
```

```
    /*
     Inputs
    */
    input i_req, i_reset_b;
    /*
     Outputs
    */
    output [SIZE-1:0] o_ctrl;
    /*
     Internal signals
    */
    wire [SIZE-1:0] si_en, so_en;

    /*
     Netlist
    */
    Sequencer2 sequencers [SIZE-1:0](
     .i_req(i_req),
     .i_en(si_en),
     .o_en(so_en),
     .o_ack(o_ctrl),
     .i_reset_b(i_reset_b)
    );

    C_AND2A anda_reset (.a(so_en[SIZE-1]), .b(i_reset_b),.z(si_en[0])
        );

    assign si_en[SIZE-1:1] = so_en[SIZE-2:0];
endmodule


`endif
```

E.3.6 Multicaster

```
/*
  Description:
    Multicaster

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/
`ifndef _Multicaster_v
`define _Multicaster_v

`include "global.v"

/*
  Addressing Mechanicm
*/

module Multicaster(
  i_en,
  i_data,
  i_req,
  i_ack,
  o_data,
  o_req,
  o_ack,
  i_reset_b);

parameter N_MC=2;
parameter DATA_WIDTH=5;

//Stupid define such that modelsim does not crash
//TBD
`define DATA_WIDTH_FIX_MODELSIM (DATA_WIDTH+1-1)

// Inputs
input i_req, i_ack, i_reset_b;
input [N_MC-1:0] i_en;
input [N_MC*DATA_WIDTH-1:0] i_data;

// Outputs
output o_req,o_ack;
output [DATA_WIDTH-1:0] o_data;

//Wires
wire [N_MC-1:0] si_req, so_next, so_req;
wire [DATA_WIDTH*N_MC-1:0] s_data;
wire [DATA_WIDTH*N_MC-1:0] s_data2;
wire s_req;
```

```
//Construct the sequencer
Sequencer_en sequencers_en[N_MC-1:0] (
  .i_en(i_en),
  .i_req(si_req),
  .i_ack(i_ack),
  .o_req(so_req),
  .o_ack(so_next),
  .i_reset_b(i_reset_b));
assign si_req[0] = i_req;
assign o_ack =  so_next[N_MC-1];
assign si_req[N_MC-1:1] = so_next[N_MC-2:0];

genvar i,j;
generate
    for(i=0;i<N_MC;i=i+1)
    begin : and_tree_generation

//        C_AND2 DATA_ands[DATA_WIDTH-1:0] (i_data[(z+1)*DATA_WIDTH-1:z
    *DATA_WIDTH],so_req[i],s_data[(z+1)*DATA_WIDTH-1:z*DATA_WIDTH]);
       for(j=0;j<DATA_WIDTH;j=j+1)
       begin : and_tree_generation2
         C_AND2 DATA_and (i_data[i*DATA_WIDTH+j],so_req[i],s_data[i*
             DATA_WIDTH+j]);
       end
    end
endgenerate

//Generate OR trees for request and data
generate
  C_ORx #(N_MC) reqor(.inputs(so_req), .z(s_req));

  for(i=0;i<`DATA_WIDTH_FIX_MODELSIM;i=i+1) //TBD: should be
      DATA_WIDTH
  begin : req_tree_generation
   for(j=0;j<N_MC;j=j+1)
    begin : req_tree_generation2
        assign s_data2[i*N_MC+j] = s_data[j*DATA_WIDTH+i];
    end
   C_ORx #(N_MC) data_or(.inputs(s_data2[(i+1)*N_MC-1:i*N_MC]),.z(
       o_data[i]));
  end

endgenerate

//Delay the request line
TC_delay #1000 delay1(s_req,o_req);

endmodule
```

```
`endif
```

E.3.7 NA

```
/*
  Description:
    Network adapter

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _NA_v
`define _NA_v

`include "global.v"

module NA(
  i_data,
  i_valid,
  o_data,
  o_req,
  i_ack,
  o_route_req,
  i_route,
  i_req,
  o_ack,
  i_route_ack,
  i_clk,
  i_reset_b);

/*
  Parameters
*/
parameter DATA_WIDTH=`DATA_WIDTH;
parameter BUS_WIDTH=`BUS_WIDTH;
parameter ROUTE_WIDTH=`ROUTE_WIDTH;

/*
  Inputs
*/
input [DATA_WIDTH-1:0] i_data;
input [ROUTE_WIDTH-1:0] i_route;
input i_valid,i_clk,i_ack,i_reset_b;
input i_req, i_route_ack;

/*
  Outputs
*/
output [BUS_WIDTH-1:0] o_data;
output o_req, o_route_req, o_ack;
```

```
/*
  Internal signals
*/
wire [DATA_WIDTH-1:0] s_data;
wire s_rr;
wire s_reset_b;

/*
  NetList
*/
TC_INV #(1) inv_reset(.a(i_reset_b),.z(s_reset_b));
C_OR2 rr(i_valid,s_reset_b,s_rr);

// Data
C_FD1Q dataflipflops[DATA_WIDTH-1:0] (i_data,s_rr, s_data);
assign o_data = {i_route,s_data};

// Request and ack
assign o_req = i_req;
assign o_ack = i_ack;


// Route request
wire s_route_ack;
TC_INV #(1) inv1 (i_route_ack,s_route_ack);

C_AND2 and_reset(.a(i_valid),.b(i_reset_b),.z(i_valid_reset));
C_C2MP_R0 reqc (s_route_ack, i_valid_reset, o_route_req,i_reset_b);

//ambit synthesis off

/*
    Function which tests that the valid signal never comes true when
       the network adapter is
    currently active
*/
`ifdef ERROR_CHECKING
always @(posedge i_valid)
begin
  if(o_route_req!=0 || i_route_ack!=0)
  begin
    $display ("ERROR!. Valid signal came through when network adapter
       was already busy.\n");
    $display ("  req: %b",o_route_req);
    $display ("  ack: %b",i_route_ack);
    $stop; //masked by synthesis off

  end
end
```

```
`endif
//ambit synthesis on
endmodule
`endif
```

E.3.8 serializer

```
/*
  Description:
    Serializes a parrallel packet into flits of 2 bits

    Both input and output uses a 4-phase bundled data protocol

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _serializer_v
`define _serializer_v

`include "global.v"


module serializer(
  i_req,
  i_ack,
  i_data,
  o_req,
  o_ack,
  o_data,
  o_eop,
  i_reset_b);

  /*
   Parameters
  */
  parameter BUS_WIDTH=`BUS_WIDTH;
  parameter LINE_SIZE=2;
  parameter N_LINES=(BUS_WIDTH/LINE_SIZE);

  /*
   Inputs
  */
  input i_req, i_ack, i_reset_b;
  input [BUS_WIDTH-1:0] i_data;
  /*
   Outputs
  */
  output o_ack, o_req;
  output [1:0] o_data;
  output o_eop;
  /*
  Internal signals
  */
  wire s_req;
```

```
  wire [N_LINES:0] s_ctrl;
  wire [N_LINES-1:0] si_data1, si_data2;


  //Handshake controller
  serialize_controller #(N_LINES+1) controller(
    .o_ctrl(s_ctrl),
    .i_req(i_req),
    .o_ack(o_ack),
    .i_ack(i_ack),
    .i_reset_b(i_reset_b));

  genvar i;
  generate
    for(i=0;i<N_LINES;i=i+1)
    begin : assign_gen
      assign si_data1[i] =i_data[BUS_WIDTH-i*2-2];
      assign si_data2[i] =i_data[BUS_WIDTH-i*2-1];
    end
  endgenerate

  //Mux
  TC_mux #N_LINES mux1(.i_data(si_data1),.i_ctrl(s_ctrl[N_LINES-1:0])
      ,.o_data(o_data[0]));
  TC_mux #N_LINES mux2(.i_data(si_data2),.i_ctrl(s_ctrl[N_LINES-1:0])
      ,.o_data(o_data[1]));

  //eop
  assign o_eop=s_ctrl[N_LINES];
  //the request out is an or of the control lines (except eop)
  C_ORx  #N_LINES or_three(.inputs(s_ctrl[N_LINES-1:0]),.z(s_req));


  TC_delay #400 delay_req(.a(s_req), .z(o_req));
endmodule


/*
  serialize_controller
*/
module serialize_controller(i_req, o_ack, i_ack, o_ctrl, i_reset_b);
  parameter SIZE=2;
  // Inputs
  input i_req, i_ack, i_reset_b;
  // Outputs
  output o_ack;
  output [SIZE-1:0] o_ctrl;

  //Internal signals
```

```
  wire [SIZE-1:0] si_req, so_next, so_req;

  //The controller simply consists of a number of sequencers
  Sequencer sequencers[SIZE-1:0] (.i_req(si_req), .i_ack(i_ack),
                        .o_req(so_req), .o_ack(so_next), .i_reset_b(
                           i_reset_b));
  assign si_req[0] = i_req;
  assign si_req[SIZE-1:1] = so_next[SIZE-2:0];
  assign o_ack =  so_next[SIZE-1];
  //The controller output is the request signals
  assign o_ctrl = so_req[SIZE-1:0];
endmodule



`endif
```

E.3.9 Sequencer

```
/*
  Description:
    Sequencer

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/


`ifndef _Sequencer_v
`define _Sequencer_v

`include "global.v"

module Sequencer(i_req, i_ack, o_req, o_ack, i_reset_b);

  // Inputs
  input i_req, i_ack, i_reset_b;
  // Outputs
  output o_req,o_ack;
  //Internal signals
  wire   s_1,s_2;
  wire   si_ack_b,si_req_b;


  TC_INV #2 inv1(.a(i_req), .z(si_req_b));
  TC_INV #2 inv2(.a(i_ack), .z(si_ack_b));
  C_C2MP_R0 c2mp(.a(si_ack_b),.b(si_req_b),.z(s_1), .reset_b(
      i_reset_b));
  TC_INV #2 inv3(.a(s_1), .z(s_2));
  C_C2P_R0 c2a(.a(s_2),.b(si_ack_b),.z(o_ack), .reset_b(i_reset_b));
  C_NOR2 nor2(.a(s_2),.b(si_req_b),.z(o_req));
endmodule

`endif
```

E.3.10 Sequencer_en

```
/*
  Description:
    Sequencer with enable

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/


`ifndef _Sequencer_en_v
`define _Sequencer_en_v

`include "global.v"

module Sequencer_en(i_en, i_req, i_ack, o_req, o_ack, i_reset_b);

  // Inputs
  input i_en, i_req, i_ack, i_reset_b;
  // Outputs
  output o_req,o_ack;
  //Internal signals
  wire   s_ack1, s_ack2;
  wire   s_req;


  Sequencer sequencer(
    .i_req(s_req),
    .i_ack(i_ack),
    .o_req(o_req),
    .o_ack(s_ack2),
    .i_reset_b(i_reset_b));

    C_AND2 and2(.a(i_en),.b(i_req),.z(s_req));
    TC_AND2A #1 and2a(.a(i_en),.b(i_req),.z(s_ack1));
    C_OR2 or2(.a(s_ack1),.b(s_ack2),.z(o_ack));

endmodule

`endif
```

E.3.11 Sequencer2

```
/*
  Description:

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _Sequencer2_v
`define _Sequencer2_v

`include "global.v"



/*
# EQN file for model StoP4
# Generated by petrify 4.2 (compiled 5-Jul-04 at 11:55 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 8.00

INORDER = i_en i_req o_ack o_en;
OUTORDER = [o_ack] [o_en];
[o_ack] = o_en' (i_en i_req + o_ack);
[o_en] = o_ack i_req' + i_en o_en;

o_ack = o_en` i_en i_req + o_en` o_ack

# Set/reset pins: reset(o_ack)
*/

module Sequencer2(
  i_req,
  i_en,
  o_en,
  o_ack,
  i_reset_b
);

/*
  Inputs
*/
input i_req, i_en, i_reset_b;
/*
  Outputs
*/
output o_en, o_ack;
/*
  Interal signals
```

```
*/
wire so_en_b, si_req_b;

/*
  Netlist
*/
TC_INV #1 inv1(.a(i_req),.z(si_req_b));
TC_INV #1 inv2(.a(o_en),.z(so_en_b));

//o_ack
C_C3PP_R0 c1(.a(so_en_b),.b(i_en),.c(i_req),.z(o_ack),.reset_b(
    i_reset_b));
//o_en
C_C3MPP_R0 c2(.a(i_en),.b(o_ack),.c(si_req_b),.z(o_en),.reset_b(
    i_reset_b));

endmodule

`endif
```

E.4 Verification

E.4.1 bfm_lego2master

```
//-----------------------------------------------------------------------
// Copyright: Oticon A/S
// Project   : Aphrodite
// Author    : jhp
// Created   : 28.02.05
//
// Functionality: Master BFM
//
//-----------------------------------------------------------------------

// ---------------------------------------
// Defines
// ---------------------------------------



`define READY_DATA_WL 1
`define ENABLE_DATA_WL 1

`timescale 1ns/1ps

// ---------------------------------------
// Module
// ---------------------------------------
module bfm_lego2master(
  data_out,
  rdy_out
);

  // ---------------------------------------
  // Parameters
  // ---------------------------------------
  parameter T_CLK=1000;
  parameter NUM_OF_INPUTS=4;
  parameter DATA_DATA_WL=18;

  // ---------------------------------------
  // Ports
  // ---------------------------------------
  output [DATA_DATA_WL*NUM_OF_INPUTS-1:0] data_out;
  output [`READY_DATA_WL*NUM_OF_INPUTS-1:0] rdy_out;

  // ---------------------------------------
  // Signals
  // ---------------------------------------
```

```
reg [DATA_DATA_WL*NUM_OF_INPUTS-1:0] data_i;
reg [`READY_DATA_WL*NUM_OF_INPUTS-1:0] rdy_i;
reg [`ENABLE_DATA_WL*NUM_OF_INPUTS-1:0] enable_chan_i;

reg [DATA_DATA_WL*NUM_OF_INPUTS-1:0] data_o;
reg [`READY_DATA_WL*NUM_OF_INPUTS-1:0] rdy_o;

// ---------------------------------
// Functionality
// ---------------------------------

task initialize;
  begin
    data_i = 0;
    rdy_i = 0;
    enable_chan_i = 0;
    data_o = 0;
    rdy_o = 0;
  end
endtask // initialize


//=============================================
// clear_txs - 1. clear the actual transaction(s)
//=============================================

// TX task - clear transmision
task clear_txs;
  begin
  data_i=0;
  rdy_i=0;
  enable_chan_i=0;
  end
endtask //clear_txs


//=============================================
// setup_txs - 2. setting up the actual transaction(s)
//=============================================

// TX task - setup transmision
task setup_txs;
  input [3:0] input_chan_int; //number between 0<=CH<=NUM_OF_INPUTS
      -1
  input [17:0] data_int;
begin
      data_i=set_data(data_i,data_int,input_chan_int);
      enable_chan_i=set_enable(enable_chan_i,1'b1,input_chan_int);
```

```
  end
  endtask //setup_txs


  //=============================================
  // txs - 3. shooting the actual transaction(s)
  //=============================================

  // TX task - multiple masters
  task txs;
    begin
      data_o = 'bx;
      #(T_CLK);
      data_o = data_i;
//    #('T_CLK);
      rdy_o = enable_chan_i;
      #(T_CLK);
      data_o = 'bx;
      rdy_o = 0;
    end
  endtask // txs



  //=============================================
  // set_data
  //=============================================

  function [DATA_DATA_WL*NUM_OF_INPUTS-1:0] set_data;
      input [DATA_DATA_WL*NUM_OF_INPUTS-1:0] data_data;
      input [DATA_DATA_WL-1:0] data_int;
      input [3:0] input_chan_int;
    begin : body
      integer i, j;

      reg [DATA_DATA_WL*NUM_OF_INPUTS-1:0] res;

      reg [DATA_DATA_WL-1:0]       vec [0:NUM_OF_INPUTS-1];
      reg [DATA_DATA_WL-1:0]       d1,d2;

      res = data_data;

      for (i = 0; i < NUM_OF_INPUTS; i = i + 1) begin
    if (input_chan_int==i) begin
      for (j = 0; j < DATA_DATA_WL; j = j + 1) begin // iterate bits
        res[i * DATA_DATA_WL + j] = data_int[j];
      end
    end
      end
```

```
      set_data = res;

    end
  endfunction // set_data


  //=============================================
  // set_enable
  //=============================================

  function [`ENABLE_DATA_WL*NUM_OF_INPUTS-1:0] set_enable;
      input [`ENABLE_DATA_WL*NUM_OF_INPUTS-1:0] enable_data;
      input [`ENABLE_DATA_WL-1:0] enable_int;
      input [3:0] input_chan_int;
    begin : body
      integer i, j;

      reg [`ENABLE_DATA_WL*NUM_OF_INPUTS-1:0] res;

      reg [`ENABLE_DATA_WL-1:0]       vec [0:NUM_OF_INPUTS-1];
      reg [`ENABLE_DATA_WL-1:0]       d1,d2;

      res = enable_data;

      for (i = 0; i < NUM_OF_INPUTS; i = i + 1) begin
    if (input_chan_int==i) begin
      for (j = 0; j < `ENABLE_DATA_WL; j = j + 1) begin // iterate
          bits
        res[i * `ENABLE_DATA_WL + j] = enable_int[j];
      end
    end
      end

      set_enable = res;

    end
  endfunction // set_enable


  //=============================================
  // misc
  //=============================================

  assign data_out = data_o;
  assign rdy_out = rdy_o;

endmodule //bfm_lego2master
```

E.4.2 bfm_lego2slave

```
//------------------------------------------
// Copyright: Oticon A/S
// Project  : Aphrodite
// Author   : jhp
// Created  : 28.02.05
//
// Functionality: Slave BFM
//
//------------------------------------------
`include "global.v"

// ---------------------------------------
// Module
// ---------------------------------------
module bfm_lego2slave(
  data_in,
  master_in,
  rdy_in,
  result,
  reset_b
);

  // ---------------------------------------
  // Parameters
  // ---------------------------------------
  parameter DATA_WL=18;
  parameter N_SOURCES=12;
  parameter SOURCE_WL=4;
  parameter SLAVE_ID = 0;

  // ---------------------------------------
  // Ports
  // ---------------------------------------
  input [DATA_WL-1:0] data_in;
  input master_in;
  input rdy_in;
  input reset_b;
  output result;

  // ---------------------------------------
  // Signals
  // ---------------------------------------
  reg [DATA_WL*N_SOURCES-1:0] expected_data;
  reg [N_SOURCES-1:0] expected_master;
  reg [N_SOURCES-1:0] expecting_data;
  reg [DATA_WL-1:0]           tmp_data;
  reg tmp_master, tmp_expecting;
```

```
reg [SOURCE_WL-1:0] source_in;

reg result;
// ------------------------------------
// Functionality
// ------------------------------------
integer n_packets; //Number of received packets

initial begin
  expected_data=0;
  expected_master=0;
  expecting_data=0;
  n_packets=0;
end

always @(rdy_in)
begin


  if(rdy_in==1 && reset_b==1'b1)
  begin

    //The 4 LSB is the sender of the packet
    source_in = data_in[3:0];

    tmp_data=expected_data>>(source_in*DATA_WL);
    tmp_master=expected_master>>source_in;
    tmp_expecting=expecting_data>>source_in;

      //$write(".");

    //Count number of received packets
    n_packets=n_packets+1;
    //Check if the correct data has been received
    if(tmp_expecting)
      result=(data_in==tmp_data && master_in==tmp_master);else
    result=0;

    `ifdef DEBUG_LEVEL3
      $display("Port␣%0d:␣data:␣%b,␣expected:␣%b,␣source␣%d",
          SLAVE_ID,data_in, tmp_data,source_in);
      $display("␣␣Result␣%b",result);
    `endif

    `ifdef DEBUG_LEVEL2
    if(result==0)
    begin
      if(tmp_expecting!=1'b1)
```

```
            $display("Port: %0d received data when it was not supose to
                .",SLAVE_ID); else
          begin
            $display("Port %0d: expected %d from source %0d, got %d.",
                SLAVE_ID,tmp_data,source_in,data_in);
            $display("          - expected master %b, got %b",tmp_master
                ,master_in);
          end
        end
        `endif
      end else
      if(rdy_in==0 && reset_b==1'b1)
      begin
        //Reset Data
        expecting_data = expecting_data | (1'b1<<source_in);
        expecting_data = expecting_data ^ (1'b1<<source_in);
        expected_data = expected_data | (({{DATA_WL}{1'b1}})<<(
            source_in*DATA_WL));
        expected_data = expected_data ^ (({{DATA_WL}{1'b1}})<<(
            source_in*DATA_WL));
//      expected_data = expected_data ^ (({{DATA_WL}{1'bx}})<<(
    source_in*DATA_WL));
//      expected_master = expected_master ^ (1'bx<<source_in);
        expected_master = expected_master | (1'b1<<source_in);
        expected_master = expected_master ^ (1'b1<<source_in);
      //Set result wire high, This means that bursts of errors will be
          detected inidivuduably.
        result=1;
      end
    end

    //
    // setExpectedData task
    //

    task setExpectedData;
      input [DATA_WL-1:0] data_int;
      input master_int;
      input [SOURCE_WL-1:0] source_int;



      begin
//      $display("port %0d: expect %d from %0d",SLAVE_ID, data_int,
    source_int);
        expected_data = expected_data | (data_int<<(source_int*DATA_WL)
            );
        expected_master = expected_master | (master_int<<source_int);
        expecting_data =  expecting_data | (1'b1<<source_int);
```

```
        tmp_data=expected_data>>(source_int*DATA_WL);
        if(tmp_data == data_int)
        begin
        end
        else begin
//   $display("port %0d: expect %d, from %0d but got %d",SLAVE_ID,
    data_int, source_int,tmp_data);
//            $stop;
          end
     end


  endtask // setExpectedData
endmodule // bfm_lego2slave
```

E.4.3 Configuration

```
/*
  Description:
    Address configuration for the NoC. Dertermines the destinations
        for
    each input port. Also handles multicast.

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/
`ifndef _Configuration_v
`define _Configuration_v

`include "global.v"

module Configuration(clk_sys, reset_b, o_enable, o_master);

// Outputs
input clk_sys;
input reset_b;
output [`N_OUTPUTS*`N_INPUTS-1:0] o_enable;
output [`N_OUTPUTS*`N_INPUTS-1:0] o_master;
reg [0:`N_OUTPUTS*`N_INPUTS-1] s_enable;
reg [0:`N_OUTPUTS*`N_INPUTS-1] s_master;



//Ugly way to invert the bits, but I can not figure out how to do it
//without getting compiler errors
genvar i,j;
generate
    for(i=0;i<`N_INPUTS;i=i+1)
    begin : INPUT_generation
        for(j=0;j<`N_OUTPUTS;j=j+1)
        begin : OUTPUT_generation
          assign o_enable[i*`N_OUTPUTS+j] = s_enable[i*`N_OUTPUTS+j];
          assign o_master[i*`N_OUTPUTS+j] = s_master[i*`N_OUTPUTS+j];
        end
    end
endgenerate

`ifndef SYNTHESIS_ON
initial begin
  clear;
end

/*
    clear task
```

```
*/
task clear;
begin
  s_enable=0;
  s_master=0;
end
endtask // clear

/*
    enableRoute task

    Enables the route from output port 'source' to input port'
       destination'
    0 is the first port
*/
task enableRoute;
    input [3:0] source;
    input [3:0] destination;
    input master;

    begin
      s_enable[source*`N_OUTPUTS+destination]=1;
      s_master[source*`N_OUTPUTS+destination]=master;
    end
endtask // enableRoute

  /*
   disableRoute task

   Disables the route from output port 'source' to input port'
      destination'
   0 is the first port
   */
task disableRoute;
    input source;
    input destination;

    begin
      s_enable[source*`N_OUTPUTS+destination]=0;
      s_master[source*`N_OUTPUTS+destination]=0;
    end
endtask // disableRoute

  /*
   print task

   Disables the route from output port 'source' to input port'
      destination'
   0 is the first port
```

```
   */
  task print;
    integer i;
    reg [`N_OUTPUTS-1:0] tmp;
    reg [`N_OUTPUTS-1:0] tmp2;
    begin
      $display("Printing configuration");
      for(i=`N_INPUTS-1;i>=0;i=i-1)
    begin: gen4
      tmp = s_enable>>(i*`N_OUTPUTS);
      tmp2 = s_master>>(i*`N_OUTPUTS);
      $display("  %b      -      %b", tmp,tmp2);
    end
    end
  endtask // print

`endif // `ifndef SYNTHESIS_ON
endmodule

`endif
```

E.4.4 mutex

```
‘include "global.v"

‘timescale 1ns/1ps

module mutex_testbench();
// Declare inputs as regs and outputs as wires
reg i1,i2;
wire o1,o2;
  reg o;

C_MUTEX2 m(i1,i2,o1,o2);

‘define CHECK_00 o=o1; ‘CHECK_0(o) o=o2; ‘CHECK_0(o)
‘define CHECK_01 o=o1; ‘CHECK_0(o) o=o2; ‘CHECK_1(o)
‘define CHECK_10 o=o1; ‘CHECK_1(o) o=o2; ‘CHECK_0(o)
‘define CHECK_11 o=o1; ‘CHECK_1(o) o=o2; ‘CHECK_1(o)

// Initialize all variables
  initial begin
  $display ( "i1\t i2\t o1\to2" );
  $monitor ( "%b\t %b\t %b\t %b" ,
    i1, i2, o1, o2);
  i1 = 0;
  i2 = 0;
  #50;
  $display("System is now resat");
  ‘CHECK_00
  #50 i1 = 1;
  #50 ‘CHECK_10
  #50 i2 = 1;
  #50 ‘CHECK_10
  #50 i1 = 0;
  #50 ‘CHECK_01
  #50 i1 = 1;
  #50 ‘CHECK_01
  #50 i2 = 0;
  #50 ‘CHECK_10
  #50 i1 = 0;
  #50 ‘CHECK_00
  #50 i2 = 1;
  i1 = 1;
  #50 i1 = 0;
  i2 = 0;
  #50 $finish;
end

endmodule
```

E.4.5 noc_top_testbench

```
//-----------------------------------------------------------------
// Copyright: Oticon A/S
// Project  : Aphrodite
// Author   : jhp
// Created  : 28.02.05
//
// Functionality: Testbench for module noc_top.
//
//-----------------------------------------------------------------

//`define MIKKEL_TB

`define TESTBENCH

`define ADDER_TEST
`define UNICAST_TEST
`define MULTICAST_TEST
`define MULTICAST2_TEST
`define CHAOS_TEST

//-----------------------------------------------------------------
// Defines
//-----------------------------------------------------------------
`include "global.v"
`timescale 1ns/1ps

`define INPUTS 16
`define OUTPUTS 12
`define DATA_WL 18

`define CLK_FREQ 1   //10 Mhz -> 100 ns
`define T_CLK (1000*1/`CLK_FREQ)

`define T_RESET 10*`T_CLK // Length of initial reset-pulse
`define T_FS (64*`T_CLK) // Sampling period

//-----------------------------------------------------------------
// Module
//-----------------------------------------------------------------

module noc_top_testbench;

  //-----------------------------------------------------------------
  // Signals
  //-----------------------------------------------------------------

  integer      error;  // error count
```

```
integer      error_total;
integer      handle;
integer      received_packets;
integer      debug;

reg          clk;
reg          clear_b;
reg          reset_b;

wire [`INPUTS-1:0] rin;
wire [`INPUTS*`DATA_WL-1:0] din;

wire [`OUTPUTS-1:0] rout, masterout;
wire [`OUTPUTS*`DATA_WL-1:0] dout;
wire [`OUTPUTS-1:0] res;

wire [`N_INPUTS*`N_OUTPUTS-1:0] s_conf_enable,s_conf_master;

wire [`DATA_WL-1:0] adder_data_out_0;
wire                adder_ready_0;

wire [`DATA_WL-1:0] adder_data_out_1;
wire                adder_ready_1;

//Genarate variables
genvar i;


//------------------------------------------------------------------------
// Functionality
//------------------------------------------------------------------------

// Device under test
noc_top noc_top_inst (
  .clk_sys(clk),
  .reset_b(reset_b),
  .clear_b(clear_b),
  .din(din),
  .rin(rin),
  .dout(dout),
  .rout(rout),
  .masterout(masterout),
  .i_conf_master(s_conf_master),
  .i_conf_enable(s_conf_enable)
  ); //noc_top_inst
//defparam  noc_top_inst.INPUTS = `INPUTS;
//defparam  noc_top_inst.OUTPUTS = `OUTPUTS;

Configuration configuration(
```

```
    .clk_sys(clk),
    .reset_b(reset_b),
    .o_enable(s_conf_enable),
    .o_master(s_conf_master)
  );

  // Adders
  noc_adder noc_adder_0 (
    .clk(clk),
    .reset_b(reset_b),
    .select(2'd1), //select),
    .req(rout[0]),
    .data_in(dout[17:0]),
    .rdy(adder_ready_0),
    .data_out(adder_data_out_0)
  );

  noc_adder noc_adder_1 (
    .clk(clk),
    .reset_b(reset_b),
    .select(2'd2), //select),
    .req(rout[1]),
    .data_in(dout[35:18]),
    .rdy(adder_ready_1),
    .data_out(adder_data_out_1)
  );

  //-----------------------------------------------
  // Masters
  //-----------------------------------------------
  bfm_lego2master mst (
    .data_out(din),
    .rdy_out(rin)
  );
  defparam mst.NUM_OF_INPUTS=`INPUTS;
  defparam mst.T_CLK=`T_CLK;


  //-----------------------------------------------
  // Slaves
  //-----------------------------------------------
  bfm_lego2slave slv[`OUTPUTS-1:0] (
    .data_in(dout),
    .master_in(masterout),
    .rdy_in(rout),
    .reset_b(reset_b),
    .result(res));
  //Define parameters to slave
  generate
```

```
  for(i=0;i<`OUTPUTS;i=i+1)
    begin
      defparam slv[i].SLAVE_ID = i;
      defparam slv[i].SOURCE_WL=4;
      defparam slv[i].DATA_WL=`DATA_WL;
      defparam slv[i].N_SOURCES=`INPUTS;
    end
endgenerate


task setupSlave;
  input [3:0] id;
  input [`DATA_WL-1:0] dat;
  input master;
  input [3:0] sender;
  begin
     if(id==0)
       slv[0].setExpectedData(dat,master,sender); else
     if(id==1)
      slv[1].setExpectedData(dat,master,sender); else
     if(id==2)
       slv[2].setExpectedData(dat,master,sender); else
     if(id==3)
       slv[3].setExpectedData(dat,master,sender); else
     if(id==4)
       slv[4].setExpectedData(dat,master,sender); else
     if(id==5)
       slv[5].setExpectedData(dat,master,sender); else
     if(id==6)
       slv[6].setExpectedData(dat,master,sender); else
     if(id==7)
       slv[7].setExpectedData(dat,master,sender); else
     if(id==8)
       slv[8].setExpectedData(dat,master,sender); else
     if(id==9)
       slv[9].setExpectedData(dat,master,sender); else
     if(id==10)
       slv[10].setExpectedData(dat,master,sender); else
     if(id==11)
       slv[11].setExpectedData(dat,master,sender);
    end
endtask


task countNumberOfReceivedPackets;
  begin
    received_packets=0;
    //      for(l1=0;l1<`OUTPUTS;l1=l1+1)
```

```
    //      received_packets = received_packets + slv[11].n_packets
        ;
    received_packets = received_packets + slv[0].n_packets;
    received_packets = received_packets + slv[1].n_packets;
    received_packets = received_packets + slv[2].n_packets;
    received_packets = received_packets + slv[3].n_packets;
    received_packets = received_packets + slv[4].n_packets;
    received_packets = received_packets + slv[5].n_packets;
    received_packets = received_packets + slv[6].n_packets;
    received_packets = received_packets + slv[7].n_packets;
    received_packets = received_packets + slv[8].n_packets;
    received_packets = received_packets + slv[9].n_packets;
    received_packets = received_packets + slv[10].n_packets;
    received_packets = received_packets + slv[11].n_packets;
    if (debug)
      $display("t=%8.2f : %0d packets was recieved succecfully",
          $realtime,received_packets);
  end
endtask


//---------------------------------------------
// Test sequence
//---------------------------------------------
initial
begin : test_sequence

  integer ok,l1,l2,l3;
  integer packets;
  reg [`DATA_WL-1:0] dat;

  //Setup timing
  $timeformat(-9, 10, " ns", 10);
  $printtimescale;
  $printtimescale(mst);
  $printtimescale(noc_top_inst);

  // Initialize signals
  $display("INFO: Initializing signals");
  reset_b = 0;
  clear_b = 0;
  error = 0;
  error_total=0;
  packets=0;

  debug=0;

  mst.initialize;
```

```
    #(`T_RESET);
    reset_b = 1;
    clear_b = 1;
    #(`T_RESET);

    $display("INFO: Starting tests");

//-----------------------------------------------------------------
// Adder Test
//-----------------------------------------------------------------

`ifdef ADDER_TEST
    //
    // Add
    //
    $display("\nINFO: Add test. Sending data from one input to an add
        output");

    l1=1; //input
    l2=0; //output
    //Setup route
    configuration.clear; //s_enable=s_master=0
    #0.01 //We must delay here.. At least in modelsim
    configuration.enableRoute(l1,l2,1); //src,dst,master

//PACKAGE1
    dat = 28<<4|l1;
    //Setup slave
    setupSlave(l2,dat,1,l1); //1:slave_number(output),2:data,3:master
        ??,4:master_number(input)
    //Send data
    mst.clear_txs;
    mst.setup_txs(l1,dat); //input_chan,target,data
    mst.txs;
    packets=packets+1;

    //countNumberOfReceivedPackets;
    //$display("t=%8.2f : %0d of %0d packets was recieved succecfully
        ",$realtime,received_packets,packets);
    //Wait till data has been recieved
    #(3*`T_CLK);

//PACKAGE2
    dat = 17<<4|l1;
    //Setup slave
    setupSlave(l2,dat,1,l1); //1:output,2:data,3:1??,4:input
    //Send data
    mst.clear_txs;
    mst.setup_txs(l1,dat); //input_chan,target,data
```

```
    mst.txs;
    packets=packets+1;

    //countNumberOfReceivedPackets;
    //$display("t=%8.2f : %0d of %0d packets was recieved succecfully
        ",$realtime,received_packets,packets);
    //Wait till data has been recieved
    #(3*`T_CLK);


    l1=1; //input
    l2=1; //output
    //Setup route
    configuration.clear; //s_enable=s_master=0
    #0.01 //We must delay here.. At least in modelsim
    configuration.enableRoute(l1,l2,1); //src,dst,master

//PACKAGE1
    dat = l2<<4|l1;
    //Setup slave
    setupSlave(l2,dat,1,l1); //1:slave_number(output),2:data,3:master
        ??,4:master_number(input)
    //Send data
    mst.clear_txs;
    mst.setup_txs(l1,dat); //input_chan,target,data
    mst.txs;
    packets=packets+1;

    //countNumberOfReceivedPackets;
    //$display("t=%8.2f : %0d of %0d packets was recieved succecfully
        ",$realtime,received_packets,packets);
    //Wait till data has been recieved
    #(3*`T_CLK);

//PACKAGE2
    dat = 7<<4|l1;
    //Setup slave
    setupSlave(l2,dat,1,l1); //1:output,2:data,3:1??,4:input
    //Send data
    mst.clear_txs;
    mst.setup_txs(l1,dat); //input_chan,target,data
    mst.txs;
    packets=packets+1;

    //countNumberOfReceivedPackets;
    //$display("t=%8.2f : %0d of %0d packets was recieved succecfully
        ",$realtime,received_packets,packets);
    //Wait till data has been recieved
    #(3*`T_CLK);
```

```
//PACKAGE3
    dat = 11<<4|11;
    //Setup slave
    setupSlave(12,dat,1,11); //1:output,2:data,3:1??,4:input
    //Send data
    mst.clear_txs;
    mst.setup_txs(11,dat); //input_chan,target,data
    mst.txs;
    packets=packets+1;

    //countNumberOfReceivedPackets;
    //$display("t=%8.2f : %0d of %0d packets was recieved succecfully
        ",$realtime,received_packets,packets);
    //Wait till data has been recieved
    #(3*`T_CLK);


    #(5*`T_CLK);

    $display("INFO: Add functionality was succesfully tested");
    countNumberOfReceivedPackets;
    $display("       %0d of %0d packets was recieved succecfully",
        received_packets,packets);
    $display("       %0d ERRORS detected", error);

    error_total = error_total+error;
    error=0;

`endif //ADDER_TEST


//-----------------------------------------------------------------
// Unicast Test
//-----------------------------------------------------------------

`ifdef UNICAST_TEST
    //
    // UniCast
    //
    $display("\nINFO: Unicast test. Sending data from all inputs to
        all outputs");
    for(11=0;11<`INPUTS;11=11+1)
    for(12=0;12<`OUTPUTS;12=12+1)
    begin
      //Setup route
      configuration.clear;
      #0.01 //We must delay here.. At least in modelsim
      configuration.enableRoute(11,12,1);
```

```
        packets=packets+1;

        dat = (l2<<4)|l1;
        //Setup slave
        setupSlave(l2,dat,1,l1);
        //Send data
        mst.clear_txs;
        mst.setup_txs(l1,dat); //input_chan,target,data
        mst.txs;

        //Wait till data has been recieved
        #(6*`T_CLK);
      end
      $display("INFO: Unicast was succesfully tested");
      countNumberOfReceivedPackets;
      $display("      %0d of %0d packets was recieved succecfully",
          received_packets,packets);
      $display("      %0d ERRORS detected", error);

      error_total = error_total+error;
      error=0;
`endif //UNICAST_TEST


`ifdef MULTICAST_TEST
      //
      // MultiCast
      //
      $display("\nINFO: Multicast");
      for(l1=0;l1<`INPUTS;l1=l1+1)
      begin
        //We are sending data from l1 to (l2 and l3)
        $display(" - Input %0d is multicasting",l1);
        for(l2=0;l2<`OUTPUTS;l2=l2+1)
        for(l3=l2+1;l3<`OUTPUTS;l3=l3+1)
        begin
        //Setup route
        configuration.clear;
        #0.01 //We must delay here.. At least in modelsim
        configuration.enableRoute(l1,l2,0);
        #0.01 //We must delay here.. At least in modelsim
        configuration.enableRoute(l1,l3,0);
        packets=packets+2;

        dat = (l3<<8)|(l2<<4)|l1;
        //Setup slaves
        setupSlave(l2,dat,0,l1);
        setupSlave(l3,dat,0,l1);
        //Send data
```

```
      mst.clear_txs;
      mst.setup_txs(l1,dat); //input_chan,target,data
      mst.txs;

      //Wait till data has been recieved
      #(10*`T_CLK);
    end
    end
    $display("INFO: Multicast succesfully tested");
    countNumberOfReceivedPackets;
    $display("      %0d of %0d packets was recieved succecfully",
       received_packets,packets);
    $display("      %0d ERRORS detected", error);
    error_total = error_total+error;
    error=0;

`endif //MULTICAST_TEST


`ifdef MULTICAST2_TEST
    //
    // MultiCast2
    //
    $display("\nINFO: Multicasting2. Two simultanously multicasts ");
     //Several multicasts at a time
    for(l1=0;l1<`INPUTS-1;l1=l1+1)
    begin
      //We are sending data from l1 and l1+1 to (l2,l2+1, l3 and l3
         +1)
      $display(" - Input %0d and %0d are multicasting",l1, l1+1);
      for(l2=0;l2<`OUTPUTS-1;l2=l2+1)
      for(l3=l2+2;l3<`OUTPUTS-1;l3=l3+1)
      begin
      //Setup route
      configuration.clear;
      #0.01 //We must delay here.. At least in modelsim
      configuration.enableRoute(l1,l2,0);
      #0.01 //We must delay here.. At least in modelsim
      configuration.enableRoute(l1+1,l3,1);
      #0.01 //We must delay here.. At least in modelsim
      configuration.enableRoute(l1,l2+1,1);
      #0.01 //We must delay here.. At least in modelsim
      configuration.enableRoute(l1+1,l3+1,0);
      packets=packets+4;

      dat = ((l2+1)<<8)|(l2<<4)|l1;
      mst.clear_txs;
      //Setup slaves
      setupSlave(l2,dat,0,l1);
```

```
        setupSlave(l2+1,dat,1,l1);
        mst.setup_txs(l1,dat); //input_chan,target,data

        dat = ((l3+1)<<8)|(l3<<4)|(l1+1);
        //Setup slaves
        setupSlave(l3,dat,1,l1+1);
        setupSlave(l3+1,dat,0,l1+1);
        mst.setup_txs(l1+1,dat); //input_chan,target,data
        //Send data
        mst.txs;

        //Wait till data has been recieved
        #(60*`T_CLK);
      end
      end

      $display("INFO: Multicast2 succesfully tested");
      countNumberOfReceivedPackets;
      $display("      %0d of %0d packets was recieved succecfully",
          received_packets,packets);
      $display("      %0d ERRORS detected", error);
      error_total = error_total+error;
      error=0;
`endif //MULTICAST2


`ifdef CHAOS_TEST
      $display("\nINFO: Simultanus sending test");
      configuration.clear;
      mst.clear_txs;
      #0.01 //We must delay here.. At least in modelsim
      for(l1=0;l1<`OUTPUTS;l1=l1+1)
      begin
        configuration.enableRoute(l1,l1,1);
        dat = (l1<<4)|l1;
        packets=packets+1;
        mst.setup_txs(l1,dat); //input_chan,target,data
        setupSlave(l1,dat,1,l1);
      end
      //Send the data and wait
      mst.txs;
      #(`INPUTS*10*`T_CLK);
      $display("INFO: Succesfully tested");
      countNumberOfReceivedPackets;
      $display("      %0d of %0d packets was recieved succecfully",
          received_packets,packets);
      $display("      %0d ERRORS detected", error);
      error_total = error_total+error;
      error=0;
```

```
    $display("\nINFO: Chaos test. Everybody is sending to the same
        receiver");
    for(l2=0;l2<`OUTPUTS;l2=l2+1)
    begin
      configuration.clear;
      #0.01 //We must delay here.. At least in modelsim
      mst.clear_txs;
      for(l1=0;l1<`INPUTS;l1=l1+1)
      begin
        configuration.enableRoute(l1,l2,0);
        dat = (l2<<4)|l1;
        packets=packets+1;
        mst.setup_txs(l1,dat); //input_chan, target, data
        setupSlave(l2,dat,0,l1);
      end
      //Send the data and wait
      mst.txs;
      #(`INPUTS*10*`T_CLK);
    end
    $display("INFO: Succesfully tested");
    countNumberOfReceivedPackets;
    $display("       %0d of %0d packets was recieved succecfully",
        received_packets,packets);
    $display("       %0d ERRORS detected", error);
    error_total = error_total+error;
    error=0;

`endif //CHAOS_TEST

    //Count number of received packets
    countNumberOfReceivedPackets;

    $display("\nINFO: All tests done");
    if(received_packets==packets)
    $display("INFO: all %0d packets was recieved succecfully",packets
        ); else
      $display("INFO: %0d of %0d packets was recieved succecfully",
          received_packets,packets);
    $display("INFO: %0d ERRORS were detected", error_total);

    if(received_packets==packets && error_total==0)
      $display("PASSED"); else
      $display("FAILED");

    // Store error in a file for use by make
    handle = $fopen("sim.exitcode");
    $fdisplay(handle, "%d", error_total);
    $fclose(handle);
```

```
`ifdef MIKKEL_TB
    $stop;
`endif
    $finish;

  end

  //------------------------------------------------------------------------
  // Error detection
  //------------------------------------------------------------------------
  integer cc;
  always @(res)
  begin
    for(cc=0;cc<`OUTPUTS;cc=cc+1)
    if(res[cc]==0)
    begin
        error = error+1;
    end
  end


  //------------------------------------------------------------------------
  // Clock generation
  //------------------------------------------------------------------------

  always #(`T_CLK/2)
    if (clk === 1) clk = 0;
    else clk = 1;

  reg [5:0] count;

  assign    cq_fs = count === 0;

  always @(negedge clk or negedge reset_b)
    if (! reset_b)
      begin
        count <= 0;
      end
    else
      begin
        count <= count + 1;
      end

endmodule
```

E.5 Bundled data blocks

E.5.1 P_merge

```
/*
  Description:
    Merger for 4 phase bundled data protocol

    2 busses are merged into a single bus ussing arbitration.
    The 2 incoming lines does not need to be mutual exclusive

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/


`ifndef _P_merge_v
`define _P_merge_v

`include "global.v"

module P_merge(
  i_data1,
  i_req1,
  o_ack1,
  i_data2,
  i_req2,
  o_ack2,
  o_data,
  o_req,
  i_ack,
  i_reset_b
  );

/*
 Parameters
*/
parameter BUS_WIDTH=`BUS_WIDTH; //Number of data bits

/*
 Inputs
*/
input [BUS_WIDTH-1:0] i_data1, i_data2;
input i_ack,i_req1, i_req2,i_reset_b;

/*
 Outputs
*/
output o_ack1, o_ack2, o_req;
```

```
output [BUS_WIDTH-1:0] o_data;

/*
 Internal signals
*/
wire s_grant1,s_grant2;
wire s_grant1_t, s_grant2_t;
wire s_req;

/*
 Netlist
*/

//Mutex
C_MUTEX2 mutex(i_req1,i_req2,s_grant1_t,s_grant2_t);

//The grant Signal must be buffered to support 'BUS_WIDTH' ports.
    This is done in the template cell
TC_AND2A #(BUS_WIDTH) grant1(s_ack2,s_grant1_t,s_grant1); //s_grant1
    <= !s_ack2 && s_grant_t1
TC_AND2A #(BUS_WIDTH) grant2(s_ack1,s_grant2_t,s_grant2); //s_grant2
    <= !s_ack1 && s_grant_t2

//output acks
C_C2_R0 ack1(i_ack,s_grant1,s_ack1,i_reset_b);
assign o_ack1 = s_ack1;
C_C2_R0 ack2(i_ack,s_grant2,s_ack2,i_reset_b);
assign o_ack2 = s_ack2;


//data generation
// This is done in an AND-OR contruct. If the cell library support it
    , this is done
// in a complex cell. Else a NAND-NAND construct
C_AOR22 aor22[BUS_WIDTH-1:0] (.a(i_data1), .b(s_grant1), .c(i_data2),
     .d(s_grant2),.z(o_data));

//Request generation
C_OR2 reqor (s_grant1, s_grant2, s_req);

//Put in some delay on the request
//This delay must be larger than the TC_AND2A gates + routing
TC_delay #400 delay1(s_req,o_req);


endmodule

`endif
```

E.5.2 P_merge_tree

```
/*
  Description:
    Binary merge tree for a 4 phase bundled data protocol

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/

`ifndef _P_merge_tree_v
`define _P_merge_tree_v

`include "global.v"

module P_merge_tree(
  i_data,
  i_req,
  o_ack,
  o_data,
  o_req,
  i_ack,
  i_reset_b
  );

/*
  Parameters
*/
parameter INPUTS=2;  //Number of inputs
parameter BUS_WIDTH=10;  //Width of the bus

/*
  Inputs
*/
input [INPUTS*BUS_WIDTH-1:0] i_data;
input [INPUTS-1:0] i_req;
input i_ack;
input i_reset_b;

/*
  Outputs
*/
output [INPUTS-1:0] o_ack;
output [BUS_WIDTH-1:0] o_data;
output o_req;

/*
  Internal signals
*/
```

```
wire s_req1, s_req2;
wire s_ack1, s_ack2;
wire [BUS_WIDTH-1:0] s_data1, s_data2;

/*
  Netlist
*/
genvar i;

`define lower_n (INPUTS/2)
`define upper_n (INPUTS-`lower_n)

// Generate an upper and lower merge tree and connect
// them by a merge element
generate
if(INPUTS==1)
begin
  assign o_data=i_data;
  assign o_req=i_req;
  assign o_ack=i_ack;
end
else
begin
  P_merge_tree #(`lower_n, BUS_WIDTH) merge_tree_lower(
    .i_data(i_data[`lower_n*BUS_WIDTH-1:0]),
    .i_req(i_req[`lower_n-1:0]),
    .o_ack(o_ack[`lower_n-1:0]),
    .o_data(s_data1),
    .o_req(s_req1),
    .i_ack(s_ack1),
    .i_reset_b(i_reset_b)
    );
//    defparam merge_tree_upper.INPUTS=`lower_n;
//    defparam merge_tree_upper.BUS_WIDTH=BUS_WIDTH;

  P_merge_tree #(`upper_n, BUS_WIDTH)merge_tree_upper(
    .i_data(i_data[INPUTS*BUS_WIDTH-1:`lower_n*BUS_WIDTH]),
    .i_req(i_req[INPUTS-1:`lower_n]),
    .o_ack(o_ack[INPUTS-1:`lower_n]),
    .o_data(s_data2),
    .o_req(s_req2),
    .i_ack(s_ack2),
    .i_reset_b(i_reset_b)
    );
//    defparam merge_tree_upper.INPUTS=`upper_n;
//    defparam merge_tree_upper.BUS_WIDTH=BUS_WIDTH;
  //The merger for this stage
  P_merge merger (
    .i_data1(s_data1),
```

```
    .i_req1(s_req1),
    .o_ack1(s_ack1),
    .i_data2(s_data2),
    .i_req2(s_req2),
    .o_ack2(s_ack2),
    .o_data(o_data),
    .o_req(o_req),
    .i_ack(i_ack),
    .i_reset_b(i_reset_b)
    );
  defparam merger.BUS_WIDTH=BUS_WIDTH;

end

endgenerate
endmodule

`endif
```

E.5.3 P_multicast

```
/*
  Description:
    P_multicast

    Multicast module for a 4 phase bundled data protocol

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/
`ifndef _P_multicast_v
`define _P_multicast_v

`include "global.v"

module P_multicast(
  i_routes,
  i_route_en,
  i_data,
  i_req,
  o_ack,
  o_data,
  o_req,
  i_ack,
  i_reset_b);

/*
  Parameters
*/
parameter N_MC=2;    //Number of multicasts
parameter DATA_WIDTH=`DATA_WIDTH;
parameter BUS_WIDTH=`BUS_WIDTH;
parameter ROUTE_WIDTH=`ROUTE_WIDTH;

/*
  Inputs
*/
input [DATA_WIDTH-1:0] i_data;
input [N_MC*ROUTE_WIDTH-1:0] i_routes;
input [N_MC-1:0] i_route_en;
input i_req, i_ack;
input i_reset_b;


/*
  Outputs
*/
output [BUS_WIDTH-1:0] o_data;
```

```
output o_req, o_ack;

/*
  Internal signals
*/
wire [ROUTE_WIDTH-1:0] s_route;


/*
  NetList
*/
Multicaster multicaster(
  .i_en(i_route_en),
  .i_data(i_routes),
  .i_req(i_req),
  .i_ack(i_ack),
  .o_data(s_route),
  .o_req(o_req),
  .o_ack(o_ack),
  .i_reset_b(i_reset_b));
defparam multicaster.N_MC=N_MC;
defparam multicaster.DATA_WIDTH = ROUTE_WIDTH;

//Assign output
assign o_data = {s_route,i_data};
endmodule

`endif
```

E.5.4 P_network

```
/*
  Description:
    Network using 4 phase bundled data

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/

`ifndef _P_network_v
`define _P_network_v

`include "global.v"

module P_network(
  i_data,
  i_req,
  o_ack,
  o_data,
  o_req,
  i_ack,
  i_reset_b
);

/*
  Parameters
*/
parameter INPUTS=`N_INPUTS;
parameter OUTPUTS=`N_OUTPUTS;
parameter BUS_WIDTH=`BUS_WIDTH;
parameter DATA_WIDTH=`DATA_WIDTH;

/*
  Inputs
*/
input [INPUTS*BUS_WIDTH-1:0] i_data;
input [INPUTS-1:0] i_req;
input [OUTPUTS-1:0] i_ack;
input i_reset_b;

/*
  Outputs
*/
output [INPUTS-1:0] o_ack;
output [OUTPUTS*BUS_WIDTH-1:0] o_data;
output [OUTPUTS-1:0] o_req;

/*
```

```
  Internal signals
*/
wire  s_ack, s_req;
wire [BUS_WIDTH-1:0] s_data;

/*
 Netlist
*/
P_merge_tree #(INPUTS, BUS_WIDTH) merge_tree (
  .i_data(i_data),
  .i_req(i_req),
  .o_ack(o_ack),
  .o_data(s_data),
  .o_req(s_req),
  .i_ack(s_ack),
  .i_reset_b(i_reset_b));

P_router_tree #(OUTPUTS, BUS_WIDTH, DATA_WIDTH) router_tree (
  .i_data(s_data),
  .i_req(s_req),
  .o_ack(s_ack),
  .o_data(o_data),
  .o_req(o_req),
  .i_ack(i_ack),
  .i_reset_b(i_reset_b));

endmodule

`endif
```

E.5.5 P_router

```
/*
  Description:
   Router for 4 phase bundled data protocol


   MSB          LSB
   ----------------
   ROUTE -   DATA

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/


`ifndef _P_router_v
`define _P_router_v

`include "global.v"

module P_router(
  i_data,
  i_req,
  o_ack,
  o_data1,
  o_req1,
  i_ack1,
  o_data2,
  o_req2,
  i_ack2,
  i_reset_b
  );

/*
 Parameters
*/
parameter BUS_WIDTH=`BUS_WIDTH;   //Width of the bus
parameter DATA_WIDTH=23;          //Data width

/*
 Inputs
*/
input [BUS_WIDTH-1:0] i_data;
input i_ack1, i_ack2,i_req,i_reset_b;

/*
 Outputs
*/
```

```
output o_ack, o_req1, o_req2;
output [BUS_WIDTH-1:0] o_data1, o_data2;

/*
 Internal signals
*/
wire s_route1, s_route2;
wire s_ack, s_req1, s_req2;
wire s_req1_buffered, s_req2_buffered;
wire s_req_delayed;
wire [BUS_WIDTH-1:0] s_data1, s_data2;

/*
 Netlist
*/
//Route. The most significant bit determines the current route
assign s_route2 = i_data[BUS_WIDTH-1];
TC_INV #1 inv_route(.a(s_route2),.z(s_route1));

TC_delay #200 delay1(.a(i_req),.z(s_req_delayed));


//Request out
TC_INV #2 inv_ack(s_ack,s_ack_b);
C_C3P_R0 req1c (.a(s_req_delayed), .b(s_ack_b), .c(s_route1), .z(
    s_req1), .reset_b(i_reset_b));
C_C3P_R0 req2c (.a(s_req_delayed), .b(s_ack_b), .c(s_route2), .z(
    s_req2), .reset_b(i_reset_b));

//Buffer up s_req driver
TC_BUF #(BUS_WIDTH+1) req_buf1(.a(s_req1), .z(s_req1_buffered));
TC_BUF #(BUS_WIDTH+1) req_buf2(.a(s_req2), .z(s_req2_buffered));

//Put in some delay on the delay line
TC_delay #400 req1buf(.a(s_req1_buffered), .z(o_req1));
TC_delay #400 req2buf(.a(s_req2_buffered), .z(o_req2));

//Ack out
C_OR2 ackor (.a(i_ack1), .b(i_ack2), .z(s_ack));
assign o_ack = s_ack;

//Data out. Shift the route one left as the MSB was used to determine
     this route
assign s_data1 = {i_data[BUS_WIDTH-2:DATA_WIDTH],1'b0,i_data[
    DATA_WIDTH-1:0]};
assign s_data2 = {i_data[BUS_WIDTH-2:DATA_WIDTH],1'b0,i_data[
    DATA_WIDTH-1:0]};
C_AND2 dataand1[BUS_WIDTH-1:0] (.a(s_data1), .b(s_req1_buffered), .z(
    o_data1));
```

```
C_AND2 dataand2[BUS_WIDTH-1:0] (.a(s_data2), .b(s_req2_buffered), .z(
    o_data2));

endmodule

`endif
```

E.5.6 P_router_tree

```
/*
  Description:
    Binary router tree for a 4 phase bundled data protocol


  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/

`ifndef _P_merge_tree_v
`define _P_merge_tree_v

`include "global.v"

module P_router_tree(
  i_data,
  i_req,
  o_ack,
  o_data,
  o_req,
  i_ack,
  i_reset_b
  );

/*
  Parameters
*/
parameter OUTPUTS=2;    //Number of outputs
parameter BUS_WIDTH=10;
parameter DATA_WIDTH=6;

/*
  Inputs
*/
input [BUS_WIDTH-1:0] i_data;
input i_req;
input [OUTPUTS-1:0] i_ack;
input i_reset_b;

/*
  Outputs
*/
output o_ack;
output [OUTPUTS*BUS_WIDTH-1:0] o_data;
output [OUTPUTS-1:0] o_req;

/*
```

```
  Internal signals
*/
wire s_req1, s_req2;
wire s_ack1, s_ack2;
wire [BUS_WIDTH-1:0] s_data1, s_data2;

/*
  Netlist
*/
genvar i;

`define lower_n (OUTPUTS/2)
`define upper_n (OUTPUTS-`lower_n)


// Generate an upper and lower router tree and connect
// them by a router
generate
if(OUTPUTS==1)
begin
  assign o_data=i_data;
  assign o_req=i_req;
  assign o_ack=i_ack;
end
else
begin
  P_router_tree #(`lower_n, BUS_WIDTH, DATA_WIDTH) route_tree_lower(
    .i_data(s_data1),
    .i_req(s_req1),
    .o_ack(s_ack1),
    .o_data(o_data[`lower_n*BUS_WIDTH-1:0]),
    .o_req(o_req[`lower_n-1:0]),
    .i_ack(i_ack[`lower_n-1:0]),
    .i_reset_b(i_reset_b)
    );

  P_router_tree #(`upper_n, BUS_WIDTH, DATA_WIDTH) route_tree_upper(
    .i_data(s_data2),
    .i_req(s_req2),
    .o_ack(s_ack2),
    .o_data(o_data[OUTPUTS*BUS_WIDTH-1:`lower_n*BUS_WIDTH]),
    .o_req(o_req[OUTPUTS-1:`lower_n]),
    .i_ack(i_ack[OUTPUTS-1:`lower_n]),
    .i_reset_b(i_reset_b)
    );

  //parameter SIZE=`BUS_WIDTH;
  //parameter DATA_WIDTH=23;
  P_router #(BUS_WIDTH, DATA_WIDTH) router(
```

```
    .i_data(i_data),
    .i_req(i_req),
    .o_ack(o_ack),
    .o_data1(s_data1),
    .o_req1(s_req1),
    .i_ack1(s_ack1),
    .o_data2(s_data2),
    .o_req2(s_req2),
    .i_ack2(s_ack2),
    .i_reset_b(i_reset_b)
    );
end

endgenerate
endmodule

`endif
```

E.5.7 P_sink

```
/*
  Description:
    Sink for parallel bundled data

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/

`ifndef _P_sink_V
`define _P_sink_V

`include "global.v"

module P_sink(i_data,i_req, o_ack);

parameter SIZE=`BUS_WIDTH;
parameter SINK_ID=-1;

//Inputs
input [SIZE-1:0] i_data;
input i_req;
//Outputs
output o_ack;

//This is just to have some delay
TC_delay #1000 delayblock (i_req, o_ack);

`ifdef DEBUG_LEVEL2
always @(posedge i_req)
begin
  if(SINK_ID==-1)
    $display("Sink: %x",i_data);
  else
    $display("Sink %2d: %x",SINK_ID,i_data);
end
`endif

endmodule
`endif
```

E.6 *1-of-5* blocks

E.6.1 PC_bundled_1of4

```
/*
  Description:

    Protocol converter:  Converts from 1of4 encoding into bundled
       datra

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/
`ifndef _PC_bundled_1of4_v
`define _PC_bundled_1of4_v

`include "global.v"

module PC_bundled_1of4(
  i_data,
  i_req,
  o_ack,
  o_data,
  i_ack,
  i_reset_b);

input [1:0] i_data;
input i_req, i_ack, i_reset_b;
output o_ack;
output [3:0] o_data;

//wire [1:0] s_data2;
wire [3:0] s_data4, s_data4_2;
wire s_req_delay, s_ack_b;

conv_2_1of4 converter(.i_data(i_data), .o_data(s_data4));
//Delay request for 2 gates.. something like 400 ps
TC_delay #400 delay(.a(i_req),.z(s_req_delayed));

C_C2P_R0 c_elements[3:0](
 .a(s_req_delayed),
 .b(s_data4_2),
 .z(o_data),
 .reset_b(i_reset_b));

C_AND2A and2a[3:0](
 .a(i_ack),
 .b(s_data4),
  .z(s_data4_2));
```

```
assign o_ack=i_ack;



/*C_C3PP_R0 c_elements[3:0](
 .a(s_req_delayed),
 .b(s_data4),
 .c(s_ack_b),
 .z(o_data),
 .reset_b(i_reset_b));

assign o_ack=i_ack;
TC_INV inv(.a(i_ack),.z(s_ack_b));
*/
endmodule

module conv_2_1of4(i_data, o_data);

// Inputs
input [1:0] i_data;
// Outputs
output [3:0] o_data;

//Netlist
C_NOR2   nor1(i_data[0],i_data[1],o_data[0]);
TC_AND2A #4 nd2(i_data[1],i_data[0],o_data[1]);
TC_AND2A #4 and3(i_data[0],i_data[1],o_data[2]);
C_AND2   and4(i_data[0],i_data[1],o_data[3]);
endmodule

`endif
```

E.6.2 PC_1of4_bundled

```
/*
  Description:

    Protocol converter:  Converts from parallel bundled data into 1
       of4 encoding

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/
`ifndef _PC_1of4_bundled_v
`define _PC_1of4_bundled_v

`include "global.v"

module PC_1of4_bundled(
  i_data,
  o_ack,
  o_data,
  i_ack,
  o_req,
  i_reset_b);

input [3:0] i_data;
input i_ack, i_reset_b;
output o_ack, o_req;
output [1:0] o_data;
wire s_req;

assign o_ack=i_ack;

conv_1of4_2 converter(.i_data(i_data), .o_data(o_data));
C_OR4 req(.a(i_data[0]),.b(i_data[1]),.c(i_data[2]),.d(i_data[3]),.z(
    o_req));


endmodule

module conv_1of4_2(i_data, o_data);

// Inputs
input [3:0] i_data;
// Outputs
output [1:0] o_data;

//Netlist
C_OR2 or_LSB(i_data[1],i_data[3],o_data[0]);
C_OR2 or_MSB(i_data[2],i_data[3],o_data[1]);
```

```
endmodule
`endif
```

E.6.3 S_latch

```
/*
  Description:
    1 of 4 latch

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/


`ifndef _S_latch_V
`define _S_latch_V

`include "global.v"


module S_latch(i_data, i_eop, o_ack, i_ack, o_data, o_eop, i_reset_b)
    ;

//Inputs
input [3:0] i_data;
input i_eop, i_ack, i_reset_b;
//Outputs
output o_ack,o_eop;
output [3:0] o_data;


//Internal signals
wire s_ack_b;
wire so_eop;
wire [3:0] so_data;
wire s_or_t1, s_or_t2;

//Memory is made from c-elements
TC_INV #5 inv(i_ack,s_ack_b);
C_C2_R0 c_data[3:0](s_ack_b,i_data,so_data,i_reset_b);
C_C2_R0 c_eop(s_ack_b,i_eop,so_eop,i_reset_b);
assign o_data = so_data;
assign o_eop = so_eop;

//Completion detection. which generates ack
C_OR5 or5(so_data[0],so_data[1],so_data[2],so_data[3],so_eop,o_ack);

//ambit synthesis off

/*
  Error checking
*/
```

```
`ifdef ERROR_CHECKING
integer count;
always @(posedge i_data or posedge i_eop)
begin
    count=0;
    if(i_eop) count = count+1;
    if(i_data[0]) count = count+1;
    if(i_data[1]) count = count+1;
    if(i_data[2]) count = count+1;
    if(i_data[3]) count = count+1;
    if(count>1)
    begin
      $display("S_latch: ERROR");
      $display("  More than 1 signal is high. This should not be
          happening in a 1of5 protocol!");
      $display("  i_data: %b",i_data);
      $display("  i_eop: %b",i_eop);
      $stop; //masked by synthesis off
    end
end
`endif

//ambit synthesis on

endmodule
`endif
```

E.6.4 S_merge

```
/*
  Description:
    S_merge

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/


`ifndef _S_merge_V
`define _S_merge_V

`include "global.v"

/*
  Internal sub-module
*/
module blok(i_data, i_eop, i_enable, i_ack, o_data, o_eop, o_ack,
    i_reset_b);

input [3:0] i_data;
input i_enable,i_ack,i_reset_b, i_eop;
output [3:0] o_data;
output o_ack,o_eop;

wire si_ack_b;
wire [3:0] so_data;
wire so_eop;

TC_INV #5 inv(i_ack,si_ack_b);

C_C3_R0 c_eop(i_eop, si_ack_b, i_enable, so_eop, i_reset_b);
C_C3P_R0 c_data[3:0](si_ack_b, i_data, i_enable, so_data, i_reset_b);
C_OR5 orelem (so_data[0],so_data[1],so_data[2],so_data[3],so_eop,
    o_ack);

assign o_data=so_data;
assign o_eop=so_eop;

endmodule

/*
  S_merge module
*/
module S_merge(i_data1, i_eop1, o_ack1,i_data2, i_eop2, o_ack2,
    o_data, o_eop, i_ack, i_reset_b);
```

```
input [3:0] i_data1, i_data2;
input i_ack, i_reset_b, i_eop1, i_eop2;
output [3:0] o_data;
output o_ack1, o_ack2, o_eop;

wire s_complete1, s_complete2;
wire s_req1, s_req2;
wire s_grant1_t, s_grant2_t;
wire s_grant1, s_grant2;
wire s_eop1, s_eop2;
wire s_eop1_b, s_eop2_b;
wire s_reset1_b, s_reset2_b;
wire [3:0] s_data1, s_data2, s_data;

//Completion detection
C_OR5 or1(i_data1[0], i_data1[1], i_data1[2], i_data1[3], i_eop1,
    s_complete1);
C_OR5 or2(i_data2[0], i_data2[1], i_data2[2], i_data2[3], i_eop2,
    s_complete2);

TC_AND2A #1 and2a_reset1(.a(s_eop1),.b(i_reset_b),.z(s_reset1_b));
TC_AND2A #1 and2a_reset2(.a(s_eop2),.b(i_reset_b),.z(s_reset2_b));

//Mutex
C_C2MP_R0 sr1(.a(s_reset1_b), .b(s_complete1), .z(s_req1), .reset_b(
    i_reset_b));
C_C2MP_R0 sr2(.a(s_reset2_b), .b(s_complete2), .z(s_req2), .reset_b(
    i_reset_b));
//C_SR sr1(.set_b(s_complete1_b), .reset_b(s_reset1_b), .q(s_req1), .
    q_b());
//C_SR sr2(.set_b(s_complete2_b), .reset_b(s_reset2_b), .q(s_req2), .
    q_b());
C_MUTEX2 mutex(s_req1, s_req2, s_grant1_t, s_grant2_t);

//Grants
TC_AND2A #5 and_grant1(.a(s_eop2),.b(s_grant1_t),.z(s_grant1));
TC_AND2A #5 and_grant2(.a(s_eop1),.b(s_grant2_t),.z(s_grant2));

//The 2 enable blocks
blok c_blok1(i_data1, i_eop1, s_grant1, si_ack, s_data1 , s_eop1,
    o_ack1, i_reset_b);
blok c_blok2(i_data2, i_eop2, s_grant2, si_ack, s_data2 , s_eop2,
    o_ack2, i_reset_b);

//Output blok
C_OR2 or_data[3:0](s_data1, s_data2, s_data);
C_OR2 or_eop(s_eop1, s_eop2, s_eop);
S_latch latch_output(s_data, s_eop, si_ack, i_ack, o_data, o_eop,
    i_reset_b);
```

```
/*
  Error checking
*/
`ifdef ERROR_CHECKING
integer count;
// Check that two acces are not allowed at the same time
always @(posedge s_grant1_t or posedge s_grant2_t)
begin
    if(s_grant1_t==1 && s_grant2_t==1)
    begin
      $display("S_merge: ERROR");
      $display("  Both inputs was granted access at the same time!");
      $stop;
    end
end

// Check that two acces are not allowed at the same time
/*always @(posedge s_grant1_t or posedge s_grant2_t)
begin
    if(s_grant1_t==1)
    begin

      $display("S_merge: ERROR");
      $display("  Both inputs was granted access at the same time!");
      $stop;
    end
end*/

`endif

endmodule

`endif
```

E.6.5 S_merge_tree

```
/*
  Description:

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/

`ifndef _S_merge_tree_v
`define _S_merge_tree_v

`include "global.v"

module S_merge_tree(
  i_data,
  i_eop,
  o_ack,
  o_data,
  o_eop,
  i_ack,
  i_reset_b
  );

/*
  Parameters
*/
parameter INPUTS=2;
parameter BUS_WIDTH=4;

/*
  Inputs
*/
input [INPUTS*BUS_WIDTH-1:0] i_data;
input [INPUTS-1:0] i_eop;
input i_ack;
input i_reset_b;

/*
  Outputs
*/
output [INPUTS-1:0] o_ack;
output o_eop;
output [BUS_WIDTH-1:0] o_data;

/*
  Wires
*/
genvar i;
```

```
`define lower_n (INPUTS/2)
`define upper_n (INPUTS-`lower_n)

  wire s_eop1, s_eop2;
  wire s_ack1, s_ack2;
  wire [BUS_WIDTH-1:0] s_data1, s_data2;

  generate
  if(INPUTS==1)
  begin
    assign o_data=i_data;
    assign o_eop=i_eop;
    assign o_ack=i_ack;
  end
  else
  begin
    S_merge_tree #(`lower_n, BUS_WIDTH) merge_tree_lower(
      .i_data(i_data[`lower_n*BUS_WIDTH-1:0]),
      .i_eop(i_eop[`lower_n-1:0]),
      .o_ack(o_ack[`lower_n-1:0]),
      .o_data(s_data1),
      .o_eop(s_eop1),
      .i_ack(s_ack1),
      .i_reset_b(i_reset_b)
      );

    S_merge_tree #(`upper_n, BUS_WIDTH) merge_tree_upper(
      .i_data(i_data[INPUTS*BUS_WIDTH-1:`lower_n*BUS_WIDTH]),
      .i_eop(i_eop[INPUTS-1:`lower_n]),
      .o_ack(o_ack[INPUTS-1:`lower_n]),
      .o_data(s_data2),
      .o_eop(s_eop2),
      .i_ack(s_ack2),
      .i_reset_b(i_reset_b)
      );

    S_merge merger (
      .i_data1(s_data1),
      .i_eop1(s_eop1),
      .o_ack1(s_ack1),
      .i_data2(s_data2),
      .i_eop2(s_eop2),
      .o_ack2(s_ack2),
      .o_data(o_data),
      .o_eop(o_eop),
      .i_ack(i_ack),
      .i_reset_b(i_reset_b)
      );
```

```
    end

endgenerate
endmodule

`endif
```

E.6.6 S_network

```
/*
  Description:
    NoC network using Serial 1of4 data

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/

`ifndef _S_network_v
`define _S_network_v

`include "global.v"

module S_network(
  i_data,
  i_eop,
  o_ack,
  o_data,
  o_eop,
  i_ack,
  i_reset_b);

parameter INPUTS=`N_INPUTS;
parameter OUTPUTS=`N_OUTPUTS;
parameter BUS_WIDTH = 4;

// Inputs
input [INPUTS*BUS_WIDTH-1:0] i_data;
input [INPUTS-1:0] i_eop;
input [OUTPUTS-1:0] i_ack;
input i_reset_b;

// Outputs
output [INPUTS-1:0] o_ack;
output [OUTPUTS*BUS_WIDTH-1:0] o_data;
output [OUTPUTS-1:0] o_eop;

/*
  Internal signals
*/
wire  s_eop, s_ack;
wire [BUS_WIDTH-1:0] s_data;

S_merge_tree #(INPUTS, BUS_WIDTH) merge_tree(
  .i_data(i_data),
  .i_eop(i_eop),
  .o_ack(o_ack),
```

```
  .o_data(s_data),
  .o_eop(s_eop),
  .i_ack(s_ack),
  .i_reset_b(i_reset_b)
  );

S_router_tree #(OUTPUTS, BUS_WIDTH) router_tree(
  .i_data(s_data),
  .i_eop(s_eop),
  .o_ack(s_ack),
  .o_data(o_data),
  .o_eop(o_eop),
  .i_ack(i_ack),
  .i_reset_b(i_reset_b)
  );

endmodule

`endif
```

E.6.7 S_router

```
/*
  Description:
    S_merge

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/


`ifndef _S_router_V
`define _S_router_V

`include "global.v"

module route_control(
  i_route1,
  i_route2,
  i_eop12_b,
  o_eop,
  o_route1,
  o_route2,
  i_reset_b);

/*
 Inputs
*/
input i_route1, i_route2, i_eop12_b, i_reset_b;

/*
 Outputs
*/
output o_eop, o_route1, o_route2;

/*
 Internal signals
*/
wire s_g1, s_g2;
wire s_q1, s_q2;
wire i_eop12_b_reset;
wire s_locked_b;


/*
 Netlist
*/
C_C2_R0 c1(i_route1, s_locked_b, s_g1, i_reset_b);
C_C2_R0 c2(i_route2, s_locked_b, s_g2, i_reset_b);
```

```
C_AND2 and_reset(.a(i_eop12_b), .b(i_reset_b),.z(i_eop12_b_reset));

C_C2MP_R0 sr1(.b(s_g1), .a(i_eop12_b_reset), .z(s_q1), .reset_b(
    i_reset_b));
C_C2MP_R0 sr2(.b(s_g2), .a(i_eop12_b_reset), .z(s_q2), .reset_b(
    i_reset_b));

C_NOR2 nor2(.a(s_q1), .b(s_q2),.z(s_locked_b));

C_AND2A and21(.a(s_g1), .b(s_q1),.z(o_route1));
C_AND2A and22(.a(s_g2), .b(s_q2),.z(o_route2));

C_NOR2 nor22(.a(s_g1),.b(s_g2),.z(s_eop));
C_NAND2 nand2(.a(i_eop12_b),.b(s_eop),.z(o_eop));

endmodule //Route_control

/*********************
  S_router module
*********************/
module S_router(
  i_data,
  i_eop,
  o_ack,
  o_data1,
  o_eop1,
  i_ack1,
  o_data2,
  o_eop2,
  i_ack2,
  i_reset_b);

/*
 Inputs
*/
input [3:0] i_data;
input i_ack1, i_ack2 ,i_reset_b, i_eop;

/*
 Outputs
*/
output [3:0] o_data1, o_data2;
output o_ack, o_eop1, o_eop2;

/*
 Internal signals
*/
wire si_ack1_b,si_ack2_b;
```

```
wire s_route1, s_route2;
wire so_eop1, so_eop2;
wire [3:0] so_data1, so_data2;
wire so_ack1,so_ack2;
wire s_eop12_b;
wire s_req1,s_req2;

/*
 Netlist
*/

TC_INV #5 inv1(i_ack1,si_ack1_b);
TC_INV #5 inv2(i_ack2,si_ack2_b);
//eop and data out
C_C3_R0 c_eop1(i_eop,si_ack1_b,s_route1,so_eop1,i_reset_b);
C_C3_R0 c_eop2(i_eop,si_ack2_b,s_route2,so_eop2,i_reset_b);
C_C3P_R0 c_data1[3:0](i_data,si_ack1_b,s_route1,so_data1,i_reset_b);
C_C3P_R0 c_data2[3:0](i_data,si_ack2_b,s_route2,so_data2,i_reset_b);
assign o_eop1=so_eop1;
assign o_eop2=so_eop2;
assign o_data1=so_data1;
assign o_data2=so_data2;

//BIG or completion
C_OR8 or8(so_data1[0],so_data1[1],so_data1[2],so_data1[3],
          so_data2[0],so_data2[1],so_data2[2],so_data2[3],so_ack1);

//router control
C_NOR2 nor_eop12b(so_eop1,so_eop2,s_eop12_b);
assign s_req2 = i_data[2];
assign s_req1 = i_data[0];
route_control routecontrol(
  .i_route1(s_req1),
  .i_route2(s_req2),
  .i_eop12_b(s_eop12_b),
  .o_eop(so_ack2),
  .o_route1(s_route1),
  .o_route2(s_route2),
  .i_reset_b(i_reset_b));

//ack
C_OR2 or_ack(so_ack1, so_ack2,o_ack);
endmodule //S_router

`endif
```

E.6.8 S_router_tree

```
/*
  Description:

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/

`ifndef _S_merge_tree_v
`define _S_merge_tree_v

`include "global.v"

module S_router_tree(
  i_data,
  i_eop,
  o_ack,
  o_data,
  o_eop,
  i_ack,
  i_reset_b
  );

/*
  Parameters
*/
parameter OUTPUTS=2;
parameter BUS_WIDTH=4;

/*
  Inputs
*/
input [BUS_WIDTH-1:0] i_data;
input i_eop;
input [OUTPUTS-1:0] i_ack;
input i_reset_b;

/*
  Outputs
*/
output o_ack;
output [OUTPUTS*BUS_WIDTH-1:0] o_data;
output [OUTPUTS-1:0] o_eop;

/*
  Wires
*/
genvar i;
```

```
`define lower_n (OUTPUTS/2)
`define upper_n (OUTPUTS-`lower_n)

  wire s_eop1, s_eop2;
  wire s_ack1, s_ack2;
  wire [BUS_WIDTH-1:0] s_data1, s_data2;

  generate
  if(OUTPUTS==1)
  begin
    assign o_data=i_data;
    assign o_eop=i_eop;
    assign o_ack=i_ack;
  end
  else
  begin
    S_router_tree #(`lower_n, BUS_WIDTH) route_tree_lower(
      .i_data(s_data1),
      .i_eop(s_eop1),
      .o_ack(s_ack1),
      .o_data(o_data[`lower_n*BUS_WIDTH-1:0]),
      .o_eop(o_eop[`lower_n-1:0]),
      .i_ack(i_ack[`lower_n-1:0]),
      .i_reset_b(i_reset_b)
      );

    S_router_tree #(`upper_n, BUS_WIDTH) route_tree_upper(
      .i_data(s_data2),
      .i_eop(s_eop2),
      .o_ack(s_ack2),
      .o_data(o_data[OUTPUTS*BUS_WIDTH-1:`lower_n*BUS_WIDTH]),
      .o_eop(o_eop[OUTPUTS-1:`lower_n]),
      .i_ack(i_ack[OUTPUTS-1:`lower_n]),
      .i_reset_b(i_reset_b)
      );

    S_router router(
      .i_data(i_data),
      .i_eop(i_eop),
      .o_ack(o_ack),
      .o_data1(s_data1),
      .o_eop1(s_eop1),
      .i_ack1(s_ack1),
      .o_data2(s_data2),
      .o_eop2(s_eop2),
      .i_ack2(s_ack2),
      .i_reset_b(i_reset_b)
      );
```

```
    end
endgenerate
endmodule

`endif
```

E.6.9 S_sink

```
/*
  Description:
    Sink for 1of5 data

  Created by:
    Mikkel Stensgaard -  mikkel@stensgaard.org
*/


`ifndef _S_sink_V
`define _S_sink_V

`include "global.v"

module S_sink(i_data,i_eop,o_ack);

parameter NUMBER=1; //Sink number. Used purely for debugging
parameter MAX_BITS_IN_ENTIRE_WORD=16;

input [3:0] i_data;
input i_eop;
output o_ack;

wire s_or_t1, s_or_t2;


//Completion detection. which generates ack
C_OR5 or5(i_data[0],i_data[1],i_data[2],i_data[3],i_eop,o_ack);

//ambit synthesis off

/*`ifdef DEBUG_LEVEL2
always @(posedge o_ack)
begin
  $display("S_sink%2.d: %b,%b",NUMBER,i_data,i_eop);
end
`endif
*/

/*
    The following is receiving an entire word and displaying it
*/
reg [MAX_BITS_IN_ENTIRE_WORD-1:0] debug_data;

initial begin //masked by synthesis off
    debug_data=0;
end
```

```
always @(posedge o_ack)
begin
    if(i_eop==0)
    begin
        debug_data = (debug_data<<2) | `CONV_1of4_to_2(i_data);
    end
    else
    begin
`ifdef DEBUG_LEVEL2
        $display("S_sink%2.d: %x - %b",NUMBER,debug_data,debug_data);
`endif
        debug_data=0;
    end
end

/*
  Error checking
*/
`ifdef ERROR_CHECKING
integer count;
always @(posedge i_data or posedge i_eop)
begin
    count=0;
    if(i_eop) count = count+1;
    if(i_data[0]) count = count+1;
    if(i_data[1]) count = count+1;
    if(i_data[2]) count = count+1;
    if(i_data[3]) count = count+1;
    if(count>1)
    begin
      $display("S_sink: ERROR");
      $display("  More than 1 signal is high. This should not be
          happening in a 1of5 protocol!");
      $display("  i_data: %b",i_data);
      $display("  i_eop: %b",i_eop);
      $stop; //masked by synthesis off
    end
end
`endif

//ambit synthesis on

endmodule
`endif
```

E.6.10 S_source

```
/*
  Description:
    S_source for 1of5 data

  Created by:
    Mikkel Stensgaard - mikkel@stensgaard.org
*/


`ifndef _S_source_V
`define _S_source_V

`include "global.v"

module S_source(o_data, o_eop, i_ack, i_reset_b);
//Parameters
parameter WORD_LENGTH=16;

output [3:0] o_data;
output o_eop;
input i_ack,i_reset_b;

`ifndef SYNHESIS_ON

//registers
reg [3:0] o_data;
reg o_eop;

always @(i_reset_b)
begin
  if(i_reset_b==0)
  begin
      o_data=0;
      o_eop=0;
  end
end

integer i;
reg [1:0] data_current;
// TX task
task sendWord;
    input [WORD_LENGTH-1:0] data;

    begin
      `ifdef DEBUG_LEVEL2
        $display("S_source: %x",data);
      `endif
```

```
        //Send data MSB first
        for(i=0;i<WORD_LENGTH;i=i+2)
        begin
          data_current = (data>>(WORD_LENGTH-i-2));
          o_data=`CONV_2_to_1of4(data_current);
          `ifdef DEBUG_LEVEL3
              $display("sending %b - %b",data_current,o_data);
          `endif
          wait(i_ack==1);
          o_data=0;
          wait(i_ack==0);
        end
        //Send eop first
        o_eop=1'b1;
        wait(i_ack==1);
        o_eop=1'b0;
        wait(i_ack==0);
    end
endtask // tx

// sendSingle
task sendSingle;
    input [3:0] data;
    input eop;

    begin
        o_data=data;
        o_eop=eop;
        wait(i_ack==1);
        o_data=0;
        o_eop=0;
        wait(i_ack==0);
    end
endtask // tx

`endif
endmodule
`endif
```