Asynchronous Implementation of Virtual Channels in On-Chip Networks

Mathias Nicolajsen Kjærgaard

LYNGBY 2004
EKSAMENSPROJEKT
IMM-THESIS-2004-25

IMM

Trykt af IMM, DTU

Abstract

On-chip network has been proposed as a method to overcome two major challenges in future SoC designs: The challenge of increasing design-effort needed to implement reliable inter-module communication in SoCs, and the projected bottleneck in non-scalable global wires.

Several proposals for NoC designs have already been proposed, but mostly using synchronous approaches. This thesis investigates design of on-chip network links using asynchronous circuits, and presents three link designs of which two are providing virtual channels. The link designs have been implemented using customizable macros, which are generating link instances as verilog standard cell netlists. Link instances have been simulated with back-annotated pre-layout timing estimations for a $0.18\mu m$ CMOS technology. The implementations are evaluated on performance and cost to identify the trade-offs present when choosing between the designs, and to determine the penalty for increasing the number of channels on the link.

Keywords: System-on-Chip, Network-on-Chip, Virtual-Channels, Asynchronous Design

Contents

Preface **vii**

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Objective . . . 3
- 1.3 Overview . . . 3

2 On-chip Networks and Virtual-channels **5**
- 2.1 Topology . . . 5
- 2.2 Routing Protocol . . . 6
- 2.3 Switching Strategy . . . 7
- 2.4 Flow Control Mechanism . . . 8
- 2.5 Network Interface . . . 10

3 Design Flow **11**
- 3.1 Standard Cell Design . . . 11
- 3.2 Synthesis of Control Circuits with Petrify . . . 12
- 3.3 Macro Expansion of Net-lists using GNU m4 . . . 12
- 3.4 Timing Estimations by Synopsys . . . 14
- 3.5 Net-list Simulation in ModelSim . . . 14
- 3.6 Design Evaluation Techniques . . . 16
  - 3.6.1 Throughput and Latency . . . 16
  - 3.6.2 Area Estimation . . . 16
  - 3.6.3 Energy Measurements . . . 16
- 3.7 Automation of Design Flow . . . 18

4 Link Implementations **21**
- 4.1 Asynchronous Design . . . 21
  - 4.1.1 Handshake Channels . . . 22
  - 4.1.2 Link Interface . . . 23
  - 4.1.3 Circuit Reset . . . 24

4.2 Basic Components . . . 24
4.2.1 Passivator . . . 24
4.2.2 Mutual Exclusion . . . 25
4.2.3 Arbitration . . . 25
4.2.4 Delay-Insensitive Encoding and Decoding . . . 28
4.3 Physical Channels . . . 30
4.4 Virtual-channels with Multiplexed Data-path . . . 31
4.5 Virtual-channels with Pipelined Data-path . . . 33

5 Results and Discussion **39**
5.1 A sample on-chip network . . . 39
5.2 Performance . . . 41
5.2.1 Unloaded Link . . . 41
5.2.2 Bandwidth Sharing . . . 45
5.3 Area . . . 47
5.3.1 Cell Area . . . 48
5.3.2 Interconnect Area . . . 48
5.4 Energy . . . 50
5.4.1 Dynamic Energy . . . 50
5.4.2 Leakage Power . . . 50
5.5 Quality of Measurements . . . 51
5.6 Comparison of Virtual Channel Implementations . . . 52
5.7 Future Work . . . 53

6 Conclusion **57**

A Design Flow Scripts **59**

B Net-list Macros **63**
B.1 Some common m4 constructs . . . 63
B.2 N-channel Mutex . . . 64
B.3 Impl. 1 . . . 65
B.4 Impl. 2 . . . 66
B.5 Impl. 3 . . . 67
B.6 Funnel . . . 69

C CD Content **71**

Bibliography **73**

Glossary and Abbreviations **77**

List of Figures

2.1 Four network topologies considered for on-chip networks. . . . 6
2.2 Two nodes in a network connected by a unidirectional link with 2 virtual channels. 9

3.1 STG describing a 2-input C-element 13
3.2 The AO5NHS complex gate used for C-element implementation 13
3.3 Structure of the link testbench. 15
3.4 One stage in a quad-rail FIFO. Only one group($W = 2$) shown, but a wider FIFO is indicated by the dotted wires 17

4.1 A unidirectional NoC link with N channels. 21
4.2 Handshake channel and component notation. 23
4.3 Data-valid schemes for a push handshake channel. 23
4.4 A non-latching passivator. 25
4.5 A two input mutex with CMOS metastability-filter 25
4.6 A two input mutex with standard cell metastability-filter . . . 25
4.7 8-channel mutex built from 7 2-channel mutexs. 26
4.8 4-channel handshake arbiter. 27
4.9 Pull arbiter. Reset function left out. 28
4.10 Pull branch. 28
4.11 1-of-4 encoding is done by the cell DE24HS. 29
4.12 1-of-4 decoding with completion detection. 29
4.13 Link channels implemented as actual physical channels. 30
4.14 A single physical channel. 31
4.15 Link implementation with multiplexed datapath. 32
4.16 Static data-flow structure of the link. 32
4.17 Link implementation with multiplexed and pipelined datapath. 34
4.18 Link implementation using a pipelined data-path to increase overall throughput. 35
4.19 Funnel and horn structure with four virtual channels. 36
4.20 A bundled-data latch as used in the *funnel* and the *horn*. . . . 36
4.21 A simple latch controler. 36

4.22 Latch controller for the decoupling latch at input to the funnel. 37
4.23 Funnel and horn in an unbalanced tree structure. 38

5.1 A NoC sample. 40
5.2 Cycle time on an idle link as a function of the number of channels. 42
5.3 Cycle time on an idle link as a function of the flit width(16 channels). 43
5.4 Cycle time on an idle link as a function of the number of repeaters on the link(16 channels, 16 bit data). 44
5.5 Total throughput of the link. 46
5.6 Sharing of bandwidth on a link with 8 virtual channels. All but channel 2 and 5 are eager. 46
5.7 Total throughput of imp. 3 with varying number of virtual channels. 47
5.8 Total cell area of one link. 48
5.9 Total interconnect area occupied by the sample NoC. Only one metal layer used. 49
5.10 Energy consumption for transporting one flit through the link. Flit payload data is either random or all-zero. 51
5.11 Leakage power in link instances with varying number of channels. 52
5.12 Optimized pipelined link with credit system. 54

Preface

This thesis is written as partial fulfillment of the requirements for the degree of Master of Science at Technical University of Denmark. The master thesis project has been carried out at IMM Department of Information Technology with Prof. Jens Sparsø as advisor and PhD student Tobias Bjerregaard as co-advisor. The projected was started September 22nd 2003 and this thesis was handed in April 22nd 2004.

Lyngby, April 2004

Mathias Nicolajsen Kjærgaard, s973371
mnk@mnk.dk

Acknowledgments

First, I'd like to thank my advisor, Prof. Jens Sparsø, and co-advisor, PhD student Tobias Bjerregaard, for great help and exciting discussions on asynchronous design and on-chip networks. Also thanks to my girlfriend Alisa, my parents, my brother and my two sisters for support and encouragement, during the thesis work, and the rest of my study at DTU. And at last, thanks to the Free Software Foundation and many others for providing me with free(as in freedom) and reliable software for my DTU project workstation.

Chapter 1

Introduction

1.1 Background

For the past two decades we have witnessed an exponential growth in the number transistors that can be placed inside a single chip. This is a result both of increasing density and increasing die-size, and nothing indicates that this evolution should not continue in the years to come[18].

To benefit from the advancements in production technology it is necessary that system design process is evolving at the same pace. Two main trends for enhancing system design productivity is an increasing level of abstraction and an increased level of automation[18]. In the past the level of abstraction has been raised from device level to gate and macro-cell level, and the latest step is the use of *intellectual property*(IP) blocks to compose a system design. With modern production technology it is possible to put entire computer systems on a single chip with CPU, DSPs, memory and IO-controllers where each of these modules are IP blocks. This design methodology is called System-on-Chip(SoC) and is already in wide use.

Today SoCs most often use either ad-hoc global wiring or time-division multiplexed buses for communication between modules on the chip. Ad-hoc wiring may have a substantial influence on the design costs, and buses are predicted to become a bottleneck for future SoC design because of the shared medium. Network-on-Chip(NoC) has been proposed as solution to these problems[34, 15, 5]. In the NoC design approach the global wires are replaced with segmented wires(*links*) connecting network *nodes*. Each module in the SoC is connected to a node in the network and in that way acts as clients on the network.

Ideally a general purpose NoC could be designed and verified once and for all, and then used in several SoC designs instantiated with appropriate parameters for the given application. If IP vendors use standard interface between the NoC nodes and SoC IP cores, a NoC design could be just another IP block that you buy along with other components needed for a SoC design. It would also allow you to replace one network implementation with another in a plug-and-play manner to fit new requirements. This decoupling of *communication* and *computation* is considered a very important aspect of NoC design.

Another challenge that arise from future technology advancements is that the length of global wire does not scale as opposed to transistors and local interconnect, and therefore global interconnect is projected to become a major bottleneck in future deep sub-micron(DSM) integrated circuits[33]. NoC design has the potential of increasing the wire utilization through sharing and might help avoiding the bottleneck in global on-chip communication.

Decreasing clock cycles and increasing die sizes will render it impossible to distribute a single global clock signal. Time-of-flight(TOF) delays alone will set a lower bound of approximately $220ps$ for corner-to-corner communication[33]. RC-delays will however stay the dominant delay factor in the near future, and the corner-to-corner delay will be considerably longer than the $220ps$ limit posed by TOF[33]. This conflicts with the $12GHz$ circuits projected for future $50nm$ technology[18] and calls for dividing the chip into smaller modules with separate clock domains. This scenario is supported by the Globally Asynchronous - Locally Synchronous(GALS) design methodology.

NoC offers a structured approach to the design of a GALS system with the network being *globally asynchronous* part and the SoC modules being the *locally synchronous* parts. Resent proposals for NoC architectures[27, 19] does however use synchronous techniques, but asynchronous solutions has also been proposed in [2] which presents the delay insensitive interconnect network *Chain*.

An asynchronous design have several advances over synchronous NoCs. Asynchronous circuits has low power consumption proportional to the activity in the network. Ideally an idle network would therefore have zero power consumption. Asynchronous circuits use either matched delays or delay insensitive techniques to obtain actual-case latency. This makes the circuits more robust and since global wires in a NoC may have significant delay variations due to cross-talk, temperature and process variations, it may also improve performance compared to synchronous circuits which always assume worst-case latency. Asynchronous circuits also have lower emission of electromagnetic noise since current spikes caused by the clock are avoided. The drawback of asynchronous design is that it so far only is a small niche in the area of chip design, and therefore it lacks of CAD and Electronic Design Automation(EDA) tools with fluent design flows as we know from synchronous design.

On-chip networks are very close related to multiprocessor networks and much of the research done in this area can be used in the NoC-arena as well. One example is the concept of *virtual channels* which was proposed by William J. Dally as a means of avoiding deadlocks and reducing network latency[14, 12]. Many studies have been made to investigate how the number of virtual channels is influencing the performance of the network, and how virtual channels can be used to support a variety routing protocols. The actual cost of implementing virtual channels in on-chip network link is however unexplored and hence the subject of this thesis.

1.2 Objective

The objective of this project is to construct and evaluate implementations of asynchronous on-chip network links with virtual channels. The implementations will be evaluated on power, area and performance to determine the cost of adding virtual channels and to identify trade-offs between these parameters when choosing link implementation. It will also be investigated how the implementations are affected by future technology advancements.

1.3 Overview

Chapter 2 will give a short introduction to on-chip network and general network concepts while pointing out distinctions between multiprocessor networks and on-chip networks. The link implementations in Chapter 4 has high focus on regularity and customize-ability and therefore Chapter 3 will go through the design flow and explain how these goals are achieved. This chapter can be skipped if you just want to "get to the point". Chapter 4 will present three implementations of on-chip network links, and go through the design decisions for each of them. Chapter 5 will analyze and discuss performance and cost parameters of the link implementations based on an extensive set simulation results. Chapter 5 will also propose some future improvements for the implementations, and at last Chapter 6 will conclude this thesis.

Appendix A and B includes a few design-flow scripts and sample source-code listings. Full source-code and all scripts are included on the CD enclosed in this report. Content of the CD is listed in Appendix C. If the CD is missing and you need the files, please contact the author(`mnk@mnk.dk`). A number of abbreviations will be used and they are listed at the last page of this thesis.

Chapter 2

On-chip Networks and Virtual-channels

On-chip networks share many concepts with interconnection networks for traditional multiprocessor systems which has been an area of active research for many years. Basic knowledge of the area of multiprocessor networks is assumed in the following discussion. A good introduction to the subject can be found in [11].

When classifying networks it is traditionally done by identifying four key properties which is *topology*, *routing algorithm*, *switching strategy* and *flow control mechanism*[11]. In this chapter we will go through these properties and relate them to on-chip networks.

2.1 Topology

According to [35, 15] the best choice for network topology in a NoC will be the *mesh* or the *torus*. These topologies are straight forward to layout on a chip due to the 2D square structure. The torus topology has twice the bisection bandwidth of a mesh network at the expense of a doubled wire demand[15]. The *fat-tree* topology which is the best choice from a connection point of view and which is widely used in multiprocessor systems[11], suffers from very complex wiring demands[34]. In [19] they argue that an irregular application-specific topology often is the best choice, since many SoCs uses modules with varying size and communication requirements unlike multiprocessor systems which are mostly homogeneous. Figure 2.1 shows the four topologies mentioned here.

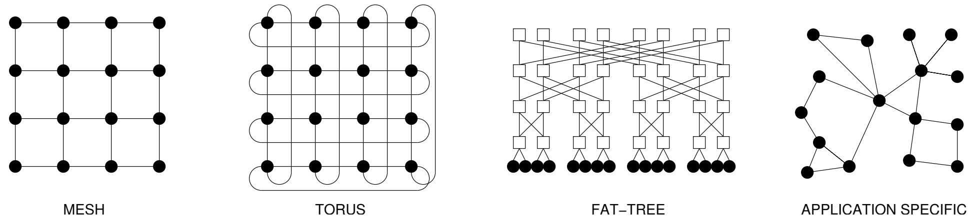

Figure 2.1: Four network topologies considered for on-chip networks.

2.2 Routing Protocol

The routing protocol determines which route in the network a packet takes when traveling from source to destination. The two main issues for a routing protocol is deadlock avoidance and traffic shaping. A deadlock is the situation where the network halts because all buffers are full and there is a circular dependency somewhere that prevents communication to proceed. With the *store-and-forward* routing scheme, deadlock can be avoided by structuring the use of buffers in the network nodes, but with the *wormhole* routing scheme a packet may span several network nodes at the same time and therefore a different approach must be taken. In [14] is presented a solution for deadlock-free routing in a wormhole routed network using the concept of *virtual-channels*. Section 2.4 has a detailed discussion on virtual-channels and the benefits of using virtual-channels in a NoC design. The idea of *wormhole* routing is to lower the demand for buffers in the network nodes by starting to forward the packet to the next hop as soon as the first flit has arrived. This approach also reduces the ideal latency of a packet transmission. These properties of wormhole routing fit very well with the requirements of a NoC design.

A routing protocol can either be *deterministic* or *adaptive*. In a deterministic routing protocol the route is solely determined by which source and destination the packet is traveling between. This routing scheme may lead to congested areas in the network and poor utilization of the network capacity. Adaptive routing has the purpose of routing packet around failing nodes and congested areas in the network to improve performance and fault tolerance[13]. As in [27, 34] it will here be assumed that on-chip network links newer fails or corrupts data, and hence it is only for performance reasons that adaptive routing is employed. The NoC design presented in [19] does however support data corruption on the links by applying CRC error-detection, and future NoCs may even support failing links as a trade-of for aggressive performance optimizations.

Design-time knowledge of the traffic pattern in the network can be used to place the modules in the network in a way that minimize congestion. In a SoC design it might often be obvious which modules that need heavy intercommunication and therefore should be placed close to each other. This is different from multiprocessor systems which have uniform nodes and communication patterns that depends on application software.

In the area of multiprocessor interconnection networks, several studies exists of performance relationship between deterministic and adaptive routing. In [13] it is shown that adaptive routing can provide increased performance while maintaining the network deadlock-free. Good performance result however requires that adaptive routing is accompanied by an increased number of virtual channels[26]. The performance improvements of adaptive routing and virtual channels comes with the cost of increased switch complexity which results in longer delays and larger nodes[1].

2.3 Switching Strategy

Two main switching strategies exist. *Circuit switching* provides a reserved point-to-point connection between the communicating nodes. Often this connection offers some *guaranteed services* which makes it particular suitable for streaming real-time data. To be able to make guarantees for data-transfers made on a shared medium it is necessary to make *reservations* on connection setup. In [27] is presented a synchronous NoC which uses time-slots to reserve bandwidth for a particular connection. In an asynchronous network there is no global notion of time and therefore time-division is not possible. Instead the reservation can be made on virtual-channels. As proposed in [6] virtual-channels can be used to establish logically independent streams with guaranteed service between nodes in a NoC. These logical streams/circuits can be seen as static wormholes in the network, which is either established at design time or dynamically using some kind of network configuration system. In [6] it is investigated how many channels is needed on each link to establish an all-to-all network of these logical circuits, and for instance a 25 node torus network would require 15 channels on each link. The implementation of *guaranteed service* on a shared medium is closely related to *flow control mechanisms* which is the subject of the next section.

The alternative to circuit switching is *packet switching*. In this switching strategy data is not transmitted on a predefined circuit, but instead routing informations is bundled with the data when it is transmitted. At the source, a *message* is put into (possibly several) *packets* which consists of a header and the payload data. The header contains the routing informations and possibly a sequence number. Each packet is then routed through the network, and at the destination node the packets are assembled into the original *message*. Often packet switched communication is used to provide *best effort service* as opposed to *guaranteed service*. *Best effort service* does not give any guaranties of latency or throughput, but instead it has better average utilization of the network resources[11, 27].

2.4 Flow Control Mechanism

Network flow control can be performed at different levels in the network stack. End-to-end flow control can be used to minimize congestion in the network by throttling the injection of new traffic. In [13] throttling is achieved by enforcing that incoming traffic may only use a subset of the channels on a link. The result is a more stable network but a slightly increased latency.

Flow control must also be performed on *link level*. When more than one data element are waiting to get transferred over the same network link, it is the job of the *flow control mechanism* to decide in which order the elements are transferred. In a network using store and forward routing this flow control is performed at packet level, which means that when a packet is scheduled for transmission, it *will* be transmitted as a whole and not just partial. The packets are buffered in the network nodes until they are elected for transfer by the flow control mechanism.

When using wormhole routing the packet will be divided into smaller pieces(*flits*) and the flow control will be performed on flit level. The size of a flit is often related to the physical implementation of the network link. For instance the NoC design presented in [15] has a 256 bit wide data-path on each link and therefore the flit is also 256 bit.

Since only the first flit(s) in a packet contains the routing information, it is import that the following flits does not get lost from the head-flit. One way to keep flits from the same packet associated with each other is to use *virtual channel* flow control as proposed in [12]. Virtual channels are logically independent channels with separate sets of input and output buffers but sharing the same physical channel.

In virtual channel flow control each packet is assigned to a virtual channel when the header flit arrives at the node. The selection of which virtual channel a packet should be assigned to, depends on routing information and flow control decisions. All subsequent flits from that packet will be switched to the same virtual channel, and no other flits are allowed to mix in on this virtual channel. In this way it ensured that all flits in a packet stay together. When the last flit is transmitted from the source node, the head-flit may

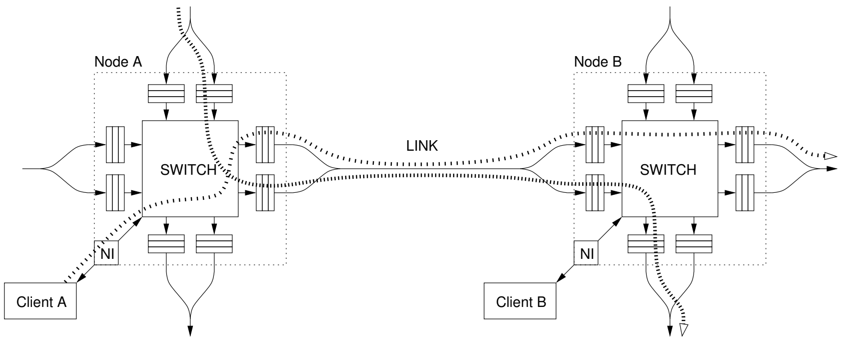

Figure 2.2: Two nodes in a network connected by a unidirectional link with 2 virtual channels.

already have arrived at the destination. In this situation the packet will occupy exactly one virtual channel on each link on the route from source to destination.

It is the job of virtual channel flow control to ensure that flits are not transmitted over the physical channel unless there is free output buffer available for the given virtual channel. Otherwise the flit would either block the link or it would have to be discarded, and neither of these choices are acceptable. To avoid this situation the sending end of the link must have information on the the buffer-status on the receiving end. This can be done by hardwiring status flags for the output buffers on the link or by sending *credits* in the opposite direction of the data, each time a flit-buffer is freed in the receiving end[11].

Figure 2.2 show two nodes in a network with unidirectional links. Each link has two virtual channels. The fat dotted lines on the figure indicates two wormholes through which a packet is in the process of being transmitted. Since they use separate virtual channels on the link, they will not block each other if one of the packets is stalled.

Allocation of bandwidth to the virtual channels can be done using *random*, *round robin*, *priority* or some other arbitration scheme. The best choice depends on the application of the network, but in a NoC design the cost in terms of area and latency may be the determining factors when choosing arbitration scheme. As mentioned earlier is it possible to use virtual channels as medium for guaranteed service traffic. It requires however that the flow control mechanism is aware that these channels require special care to ensures that all service guarantees are met.

2.5 Network Interface

The connection between network nodes and network clients are made through a network interface(NI) which may have several responsibilities. In a GALS design the NI must provide synchronization between the asynchronous network and the synchronous SoC module. Many IP modules using standard interfaces like Open Core Protocol[21] already exist, and therefore the NI must also provide wrappers for these standard interfaces. The NI may also provide high-level network abstractions like setup of circuit switched communication, multi-casting of data to several receivers or even a shared memory abstraction. In [25] is presented an NI design with the features just mentioned.

Chapter 3

Design Flow

This chapter will go through the design flow used for the link implementations presented in the next chapter. As mentioned earlier design automation is an important means for increasing productivity in chip design. The goal is therefore that the link implementations presented here can be instantiated as part of a complete NoC design in a fully automated process. For a NoC design, instantiation parameters could be information like number of modules, module sizes and guaranteed service requirements. From these parameters, a new set of instantiation parameters for switch and link modules can be derived. The generated NoC instance is then assembled with the system modules, and gate level simulations can be performed to verify functionality. Layout of the NoC system will require tight integration with floor-planning routines to ensure that modules and network components are placed appropriately.

The link design presented in the next chapter will take following instantiation parameters:

channel count is the number of channels supported by the link.

data width is the width of the data-path.

link length is the number of repeaters on each wire on the link.

3.1 Standard Cell Design

All circuits presented here are implemented using a generic standard cell library to increase portability of the designs. The library used is CORELIB8DHS HCMOS8D 1.8V which contains 777 combinatorial and sequential cells. This cell library is accompanied by several timing specifications for different operation conditions. The timing specification used here is the $1.95V/-40°$ best-case versions since this is the only version containing power measurements. The reason for choosing CORELIB8DHS HCMOS8D 1.8V is that the libraries was already installed and well-known in the department.

3.2 Synthesis of Control Circuits with Petrify

Some of the control circuits used in the link implementations are generated by the asynchronous synthesis tool Petrify[10, 9]. Petrify is given a signal transition graph(STG) describing the behavior of the control circuit and produces a speed-independent circuit implementing this behavior. An introduction to the Petrify design flow and listing of the STG requirements posed by Petrify can be found in [28]. Petrify can map the output circuit onto a specific standard cell library if it is given a corresponding gate library in the *genlib*-format. A script for translating the standard cell library files into *genlib*-format has been created by Tobias Bjerregaard and the output from this script was used in this project. For correct speed-independent operations of the circuits generated by Petrify, the gates in the genlib-library must be guaranteed hazard-free. Here it will be assumed that this is the case for the CORELIB8DHS HCMOS8D library, but this assumption must be verified before using the library in actual chip design.

The Petrify version used in this project is Petrify 4.2 which is public available from the Petrify homepage[8]. This version introduce a number of new features, which includes automatic generating reset signals in the output net-list with the command line options `-rst0` or `-rst1`. This functionality does however include some bugs. The added gates are not mapped to the standard cell library and `segmentation fault` has been experienced when synthesizing complex circuits with the `-rst` option.

An example of a circuit element synthesized by Petrify is the Muller C-element. Figure 3.1 shows the STG describing a C-element and the resulting standard cell implementation is shown in Figure 3.2. This STG is drawn in a program called Visual STG Lab(VSTGL) which can be downloaded from the website at SourceForge[16].

3.3 Macro Expansion of Net-lists using GNU m4

The GNU m4 macro processor is used to achieve a high level of customization of the link implementations. All designs are described in Verilog files contain-

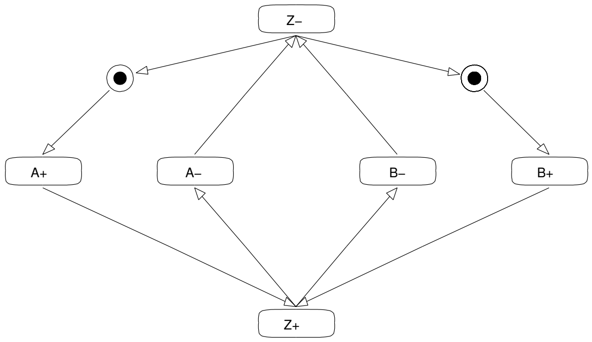

Figure 3.1: STG describing a 2-input C-element

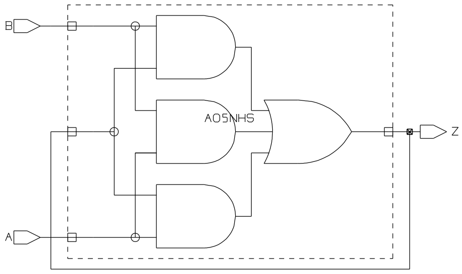

Figure 3.2: The AO5NHS complex gate used for C-element implementation

ing embedded m4 macro definitions and expansions. When the description files are processed through m4, the configuration parameters described earlier will be used to make an actual instance of the link implementation. The output from m4 is a number of Verilog net-lists which can be passed to Synopsys for timing analysis as described below. A m4 definition file with some standard constructs is listed in Appendix B.1.

3.4 Timing Estimations by Synopsys

Timing estimates are generated using Design Compiler®(DC) from Synopsys. The net-lists generated by the m4 macros are loaded into Design Compiler. To avoid that DC makes optimizations to the design, all parts are marked as don't-touch. Optimizations done by DC are unwanted since they are aimed for synchronous designs and may introduce logical hazards which will break the asynchronous control circuits.

This also mean that we must take over an import task that DC would normally perform on a synchronous circuit, namely to check for design constraint violations and to fix eventual problems. The most importing issue here is scaling of gates which drives a large fanout net. This issue has been solved by identifying all potential large-fanout net in each design, and then inserted a m4 macro which created appropriate scaled buffers. This macro uses the rule of thumb presented in [24] which says: *select an optimum fanout of 4*. The result is a chain of buffers where the input capacitance is multiplied in each step until the wanted driving-strength is reached.

Most of the logic in the link implementations will be part of the network nodes. Since a network node should be relatively small designs, the wire load model is set to `enclosed` which results in a flat slope on the dependency between fanout and wire-lengths.

The propagation delays estimated by DC are written to a Standard Delay Format(SDF) file which is used for gate-level simulations as described in the next section.

3.5 Net-list Simulation in ModelSim

ModelSim® from Model Technology is used for simulation of the link implementation. These simulations with back-annotated timing are used for functional verification and for performance analysis. A library of functional Verilog HDL descriptions is provided with the HCMOS8D standard cell library, and these has been compiled into a ModelSim simulation library.

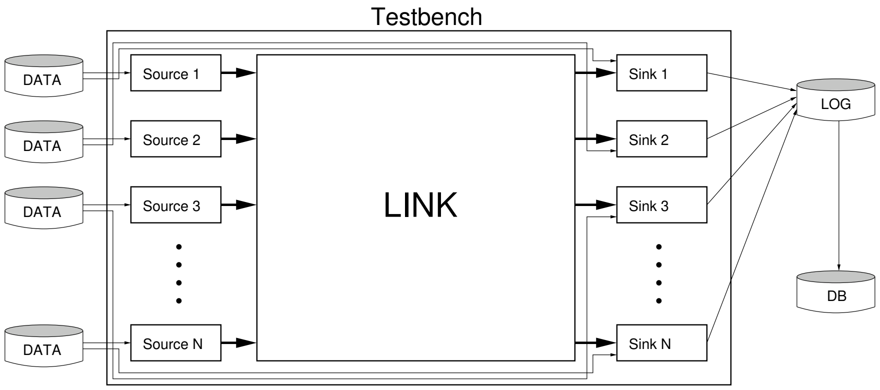

Figure 3.3: Structure of the link testbench.

Simulations of a link implementation is done by placing it in a behavioral VHDL test-bench. The same test-bench is used for all links since they use the same interface. As the link implementations themselves, the test-bench is also parameterized with regard to the number of virtual channel. The test-bench use text-files for stimuli input - one file for each virtual channel. The stimuli files are generated by the Perl script listed in Appendix A. The structure of the test-bench setup is illustrated in Figure 3.3. All *source* and *sink* modules operate concurrently to ensure that no internal dependencies in the test-bench will influence the performance measurements.

The test-bench ensures that the link behaves correctly with respect to handshake protocol at the interface. Correct transmission of data is ensured by the sink module which check all received data against the data file. If errors occur during simulation then exception will be thrown and the simulation will suspend.

During simulation the test-bench writes a transfer log-file with one entry for each sent and received flit. After simulation this log-file is parsed to a database which is used for statistical queries as described in Section 3.6.1.

At the same time all toggling information in the link cells are recorded into a Value Change Dump(VCD) file. These toggling informations is used for estimation of energy consumption in the link circuits which will be described in Section 3.6.3.

3.6 Design Evaluation Techniques

3.6.1 Throughput and Latency

The primary performance measures for networks are *bandwidth* and *latency* and both depends one many factors in the network. Only the link implementation is in focus in this project and therefore it is not possible to make a complete performance analysis for on-chip networks.

When describing link performance, the terms *throughput* and *latency* will be used. Latency is the time passing from a flit is injected in the sending end of the link, until it arrives at the receiving end of the link. Throughput will describe the number of flits per time-unit that can be transported through the link or through a single channel at maximum speed.

Latency and throughput on a channel will be affected by contention when several virtual channels are eager to transmit at the same time. Therefore simulations will be performed with varying link load scenarios.

Performance measures are obtained by by making queries to the simulation database described above. Sample queries are shown in Appendix A.

3.6.2 Area Estimation

There is two area concerns for on-chip network links. The first concern is the *logic* area used for implementation of flow control mechanism, signal coding, pipeline buffers and wire repeaters. Estimations for these area measures can be reported by DC using the `report_area` command.

The second area concern is the interconnect area consumed by the global wires connecting network nodes. Since the area estimates made by DC are pre-layout, they will not account for these long wires. Instead the global interconnect area will be calculated using technology design rules and knowledge of the number of wires in each link implementation.

3.6.3 Energy Measurements

Energy consumption is an important aspect for a NoC design. Only asynchronous designs are presented here which means that the power consumption is highly dependent on the activity in the network. In an idle network the power will be equal to the leakage power of the circuit. Design Compiler has a `report_power` function which calculates dynamic power on basis of circuit capacitances and estimated circuit activities. The estimation of circuit activity is however targeted at synchronous designs and does not take the actual

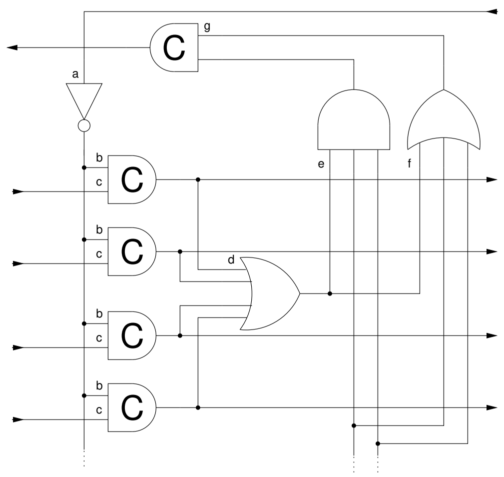

Figure 3.4: One stage in a quad-rail FIFO. Only one group($W = 2$) shown, but a wider FIFO is indicated by the dotted wires

activity into account. Therefore these estimates are not suitable for power estimations in a asynchronous circuit.

It is however possible to provide Design Compiler with circuit activity information using the Switching Activity Interchange Format(SAIF) format. These activity informations can be extracted during simulation of the circuits and should therefore be quite accurate. ModelSim does not provide direct support for the SAIF file format, but the VCD file created by ModelSim can be converted into a SAIF file using the Synopsys tool `vcd2saif`.

To verify that power calculations based on SAIF files are reliable, a simple FIFO is analyzed. The FIFO is using a delay insensitive 1-of-4 pipeline latch[28, 3] as shown in Figure 3.4. A rough estimate of the power consumption for the FIFO can be calculated by counting the number of standard-load transitions involved in a handshake. The std.load of the HCMOS8D process is given to $6fF$ in [29]. We will assume that all gates has an input capacitance equal to a std.load, which is not entirely correct since the input capacitance to the AND/OR gates is a little below the std.load, whereas the input capacitance to the C-element is a little higher. In theory the power consumption of a 1-of-N coded FIFO is data-independent. In a real layout some data dependence may arise due to cross-talk between code-words or varying capacitive coupling on the wires. This analysis will leave out these details and assume that power is data independent.

Below is summarized how the standard-load transitions of the FIFO-stage in Figure 3.4 is counted. W will represent the bit-width of the FIFO.

(a) The inverter driving one input on all latching C-gates has a fanout of $2 \times W$. When $W > 2$ the inverter must be scaled up as described in Section 3.4. The input marked a is therefore accounted as $1/2 \times W$ std.loads.

(b) The C-gate input marked b makes one cycle for each handshake. The contribution is thus $2 \times W$ std.loads.

(c-f) At each handshake only one of the four data inputs makes a cycle. The inputs marked c, d, e and f thus contributes $1/2 \times W$ std.loads each.

(g) Both inputs on the C-gate doing completion detection makes a cycle per handshake. This means a constant of 2 std.loads is contributed.

The total number of std.load cycles is:

$$(1/2 \times W) + (2 \times W) + 4 \times (1/2 \times W) + 2 = 4.5 \times W + 2$$

In [30] the following figure for power consumption in the HCMOS8D technology is given: $35nW/Gate/MHz/Stdload$. A 16 bit FIFO($W = 16$) thus has an estimated energy consumption of $(4.5 \times 16 + 2) Stdload \times 35fJ/Stdload = 2.6pJ/handshake$. Table 3.1 show energy consumption reported by DC for several FIFO configurations compared to calculated values as described above. Actually DC is reporting a power estimate, and the numbers in column four has been calculated by dividing the power estimate by ($activity \times W \times stages$). The energy reported by DC includes wire switching energy which is contributing with approximately 50% of the total. This explain why the energy consumption reported by DC is approximately 2 times the values calculated from std.loads. When W is increased from 16 to 32, the AND/OR gates are replaced with gate trees and therefore DC is reporting increased energy consumption. This is not covered by the std.load calculation above. The last two rows has a lower activity because the data producer was throttled. The results in Table 3.1 show that the use of SAIF files in DC makes it possible to obtain credible energy estimations for asynchronous circuits based on simulated activities in the circuits.

3.7 Automation of Design Flow

Some standard methodologies from software development have been applied to the implementation project, to achieve a smooth development cycle with easy test and simulation. This includes extensive use of *GNU make* for all steps in the cycle. For customization of the link implementation, a *configure*

W	Stages	Activity	DC(simulated SAIF)	Std.load calc.
16	16	322	313	162
16	32	319	305	162
32	16	208	357	159
32	32	208	350	159
32	16	87	362	159
32	32	87	354	159

Table 3.1: Power consumption($fJ/flit/stage/bit$) on a FIFO with varying $W(bits)$, activity($flits/\mu S$) and number of FIFO stages.

script is provided, which also is common in software development projects. When the configure script run without parameters, it will explain which parameters are need, as shown below:

```
[s973371@cstpro7 src]# ./configure
Usage: ./configure <LINK-IMPL> <CHANNEL-COUNT> <DATA-WIDTH> <STAGE-COUNT>
<LINK-IMPL> choose the link implementation(a number 1-3)
<CHANNEL-COUNT> is the number of channels on the link(must be a power of 2)
<DATA-WIDTH> is the width of the bundled-data interface for each channel
<STAGE-COUNT> is the number of buffers/latches on the link
```

When it has been decided which configuration to analyze, for instance for power consumption, the following command can be executed:

```
[s973371@cstpro7 src]# ./configure 2 16 32 5
[s973371@cstpro7 src]# make power-report
```

This will initiate all the procedures described earlier in this chapter:

- Control circuit described as signal transition graphs will be synthesized into gate level net-lists by Petrify.
- All macros in the link descriptions will be expanded by m4 using the parameters given to the configure script.
- The resulting net-lists is compiled with DC and timing-information is extracted from the design.
- A simulation is performed using the timing-information created by DC, and the simulation results are added to the database.
- The VCD file produced by the ModelSim simulation is converted into SAIF format.

- The link design is loaded back into DC to make a power report using the newly created SAIF file.
- Result from a throughput query is printed to the screen to facilitate comparison of power and activity.

Appendix A show the *Makefile* for the project.

Chapter 4

Link Implementations

This chapter will present three asynchronous designs of a unidirectional NoC link with support for multiple channels. All links will be presented in a simple scale with only a few channels. When describing extensions to these, N will denote the number of channels on the link. Figure 4.1 show a NoC link and the context. The link is surrounded with a dashed line. In the implementations presented here we will assume the flit-size to be the same as the width of the data-path. The data width will be denoted W.

4.1 Asynchronous Design

All circuits presented here are using asynchronous design methodologies, which are proved and well documented[28, 22] but not in wide commercial use yet. The fundamental difference between synchronous and asynchronous circuits is that the clock signal is replaced by implicit or explicit data-valid

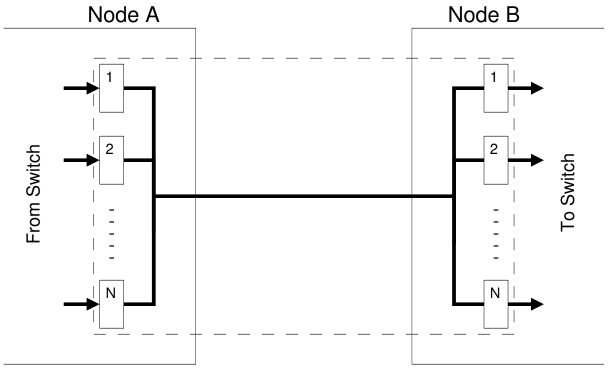

Figure 4.1: A unidirectional NoC link with N channels.

information associated with each data element.

All designs will use 4-phase "return to zero"(RTZ) handshakes to avoid the complications of 2-phase protocol as described in [28]. In the link ends all circuits use *bundled-data* protocol, also called *single rail* in [22]. This reduces the logic area of the link since bundled-data representation only needs half the wires of a delay insensitive encoding like dual-rail or 1-of-4. Also the link-ends after layout should have only a very limited extent on the chip, and therefore delay matching can be made using tight timing assumptions. The bundled-data protocol also avoids the synchronization over a possibly wide data-path, which might decrease performance.

The long wires in the physical channel can be heavily influenced by cross-talk which can cause the propagation delay to vary a lot. Therefore the physical channel use delay insensitive encoding as described in Section 4.2.4.

4.1.1 Handshake Channels

The link designs will be presented as circuits composed of *handshake components* which are communicating via *handshake channels*. Please note the distinction between *handshake channels* and link channels described earlier. The design diagrams will use the concept of *static data-flow structures* presented in [28], combined with the notions of handshake channel types presented in [22]. A short introduction will be given here.

In the following discussion we will assume that the bundled-data protocol is used on all handshake channels, even though a few handshake channels in the link designs are using different protocols. A handshake channel consists of a *request* and a *acknowledge* signal, and possibly some data. Three types of handshake channels will be used, and these are shown in Figure 4.2. The fat dot is marking the *active* party on the channel which is the component driving the request signal, and the open circle is marking the *passive* party which is the component driving the acknowledge signal. When data is included on a handshake channel, an arrow will mark the direction of the data-flow. On a *push* handshake channel, data is flowing from the *active* to the *passive* party which means that data-valid information is encoded on the *request* signal. On a *pull* handshake channel, data-valid information is encoded on the *acknowledge* signal. The *nonput* handshake channel has no data associated, and therefore it is only used for synchronization.

Figure 4.2 also show three basic handshake components, namely the *fork*, *join* and *latch* component. The rest of the handshake components will be presented as we go through the link implementations.

In [22] is presented the concept of *data-valid schemes* which defines how data-valid information is encoded on the request(*req*) and acknowledge(*ack*)

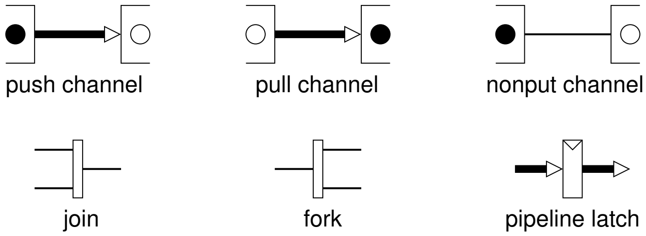

Figure 4.2: Handshake channel and component notation.

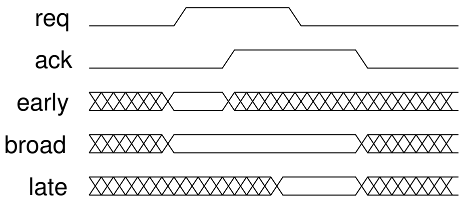

Figure 4.3: Data-valid schemes for a push handshake channel.

signals. Figure 4.3 shows the three main data-valid schemes on a push handshake channel. Similar schemes is defined for a pull channel. The *early* scheme defines that data *may* be released by the sender after $ack \uparrow$, but this does not necessarily mean that the data actually *is* released. If for instance the sender component guarantees that data remains valid until some time after $req \downarrow$, it might be possible to simplify the receiving component by taking advantage of this guarantee. This scheme is called *extended early* and will be used on some handshake channels in the implementations.

The data-valid schemes is used to reason about correct operation of the circuits, and to identify the timing assumptions which must be verified after a link has been instantiated. Generally, when operations are added to the data-path, these operations must be accompanied by delay elements in the control circuit. The data-path operations used in the link implementations are however only simple mux and demux circuits with relatively short delays. These delays may in most cases be matched by the internal delays in the control circuits, and thereby delay insertion can be avoided.

4.1.2 Link Interface

All link implementations will use the same external interface to make them directly comparable. This interface is an asynchronous 4-phase bundled-data interface similar to the handshake channels described above. It has been chosen not to include input or output buffers in the implementations since the links are tested directly in a test-bench. If the link is connected to a switch in a real system, some buffers must be added in both ends to improve performance and to decouple link activity from the switch[12].

When no buffers is included, link-level flow control must be performed on the basis of the information available at the link interface. To emphasis the fact that both sending and receiving end of a link channel must indicate that they are capable of completing a flit transfer, before the transfer is actually started, we will let both ends connect to an *active* handshake channel. In the sending end, a $req\uparrow$ from the environment will indicate that a flit is ready for transfer, and in the receiving end a $req\uparrow$ from the environment will indicate that a free buffer is ready to receive a flit. Therefore the link is *passive* in both ends, which means that the sending end will be connected to *push* channels and the receiving end will be connected to *pull* channels.

4.1.3 Circuit Reset

We will assume an active *HIGH* reset signal is present at all nodes to initialize the link. This reset signal can be a global reset signal or a signal generated at each node on power-up. Since Petrify fails to insert reset signals in the synthesized circuits, reset functionality must be inserted manually. This has been done in all link implementations, but the reset functionality is left out in all the circuit diagrams presented in the coming sections. For correct reset of the link, all inputs must be set low, and reset set high, long enough for the reset to propagate through the link-wires.

4.2 Basic Components

4.2.1 Passivator

The link is passive on both input and output channels and therefore somewhere in the link, the data must be transferred from a push channel to a pull channel. The component making this conversion is a *passivator*[22] and a non-latching version is depicted in Figure 4.4. As described in [22] this passivator implementation requires a broad data-valid scheme on the input `A` and guarantees an early data-valid scheme on the output `B`. If an isochronic fork is assumed on the output of the C-element the passivator actually guarantees an *extended early* data-valid scheme. Later in this chapter it will be discussed how we can benefit from this, and what implications it might have.

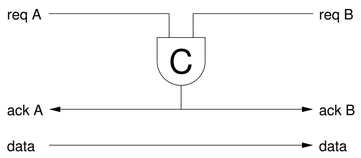

Figure 4.4: A non-latching passivator.

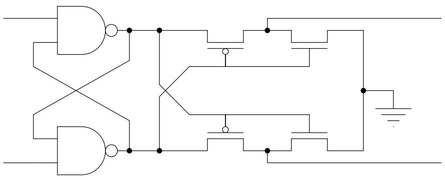

Figure 4.5: A two input mutex with CMOS metastability-filter

4.2.2 Mutual Exclusion

The CORELIB8DHS HCMOS8D cell library does not have any cells for synchronization and mutual exclusion(mutex) which is needed for the link implementations. A basic mutex circuit as presented in [20] is shown in Figure 4.5. This is a transistor level implementation which is incompatible with the choice of using pure standard cells. As mentioned in [28], it is possible to implement the metastability filter using wide gates as shown in 4.6, but this implementation oscillate in gate level simulation if both input are raised at the same time. Therefore a behavioral mutex implementation will be used for all simulations.

4.2.3 Arbitration

Two of the designs presented later in this chapter will implement *virtual channels* sharing a single physical channel. Several virtual channels may

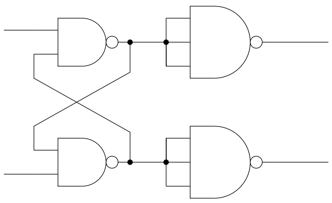

Figure 4.6: A two input mutex with standard cell metastability-filter

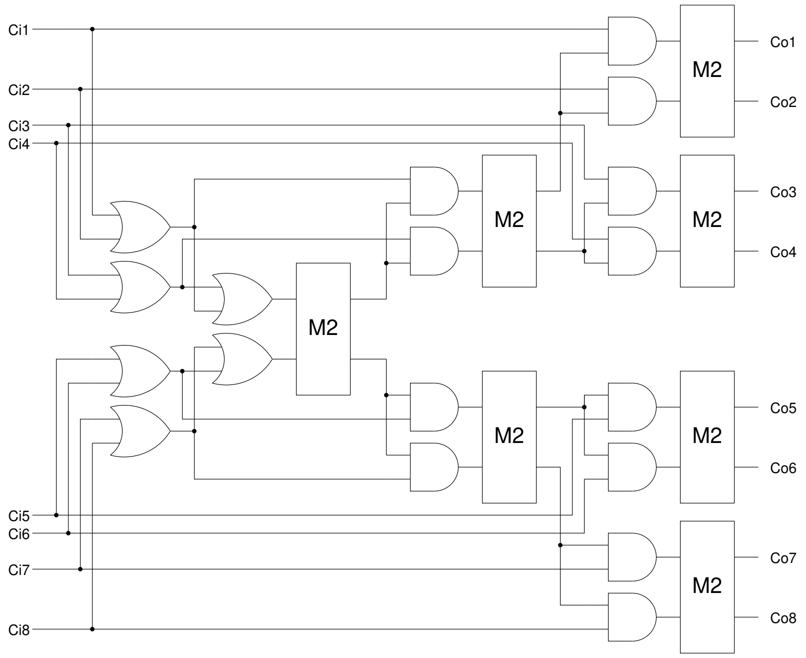

Figure 4.7: 8-channel mutex built from 7 2-channel mutexs.

try to access the physical channel simultaneously since they are operating concurrently and with no dependencies among them. This means that the arbitration for the channel must be a part of the link-implementation.

In [28] is presented a two input handshake arbiter which consists of three elements: A mutual exclusion element, some circuitry to ensure that the hole handshake is finished before the shared resource is released and at last a standard merge element. The mutex presented in previous section only has 2 channels, but we will need N virtual channels. A N-channel mutex can be constructed by using multiple 2 channel mutexs as described in [23]. The "genex 3×1" presented in [23] has been extended and a resulting 8 channel mutex is shown in Figure 4.7.

The N-channel mutex is implemented as a recursive net-list macro which is listed in Appendix B.2. This means that any number of channels is supported, but the delay in the mutex will increase when N is increased. The delay of a N-channel mutex can be calculated as:

$$(log_2(N) - 1) \times (t_{pd_{AND}} + t_{pd_{OR}}) + log_2(N) \times t_{pd_{MUTEX2}}$$

The N-channel mutex can be used to implement a N-channel handshake arbiter. A 4-channel handshake arbiter is shown in Figure 4.8. This circuit

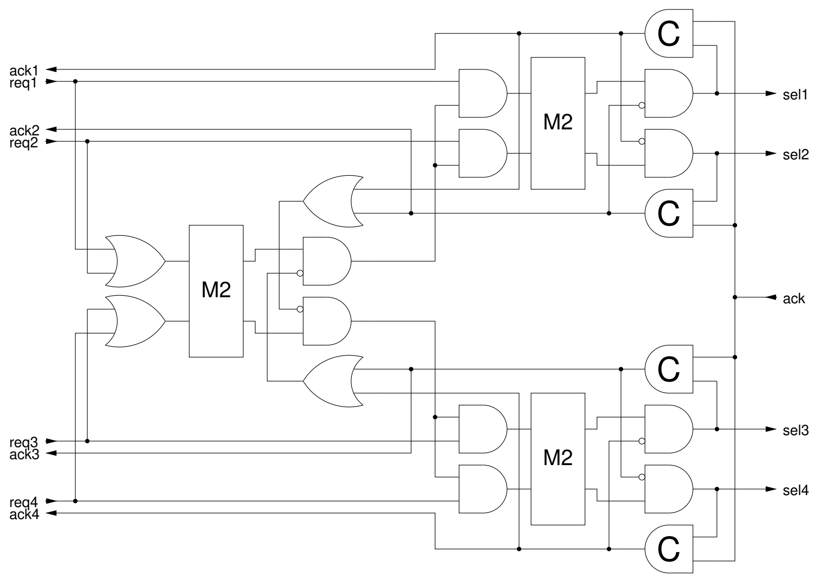

Figure 4.8: 4-channel handshake arbiter.

is an extension of the handshake arbiter presented in [28]. Since information on which channel is selected, is needed for multiplexing, the request signal is encoded along with the selected channel as 1-of-N signal. Like the N-channel mutex the handshake arbiter is defined as a recursive net-list macro.

The forward latency of the N-channel handshake arbiter can be calculated as follows:

$$(log_2(N) - 1) \times (t_{pd_{AND}} + t_{pd_{OR}}) + log_2(N) \times (t_{pd_{MUTEX2}} + t_{pd_{AND}})$$

The reverse latency is unaffected by N and only includes the latency in a single C-element(t_{pd_C}).

In the arbiter described above the *active* ports are on the left hand side and the passive port is on the right hand side. Given the notion of data flowing from left to right, this means that the arbiter is connected to push handshake channels on both sides. The last link implementation does however make heavy use of *pull* channels which means that a arbiter supporting *pull* input and output ports is needed. Figure 4.9 shows a pull arbiter with two active input ports and one passive output port. When a request is received on the output channel it is forwarded to both input channels. If both input channels acknowledge this request, the first arriving acknowledge will be selected, and the other will have to wait until the handshake has completed and a new request arrives on the output channel. The pull arbiter

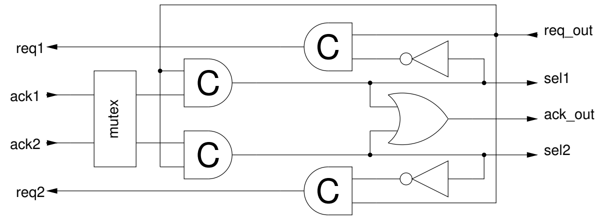

Figure 4.9: Pull arbiter. Reset function left out.

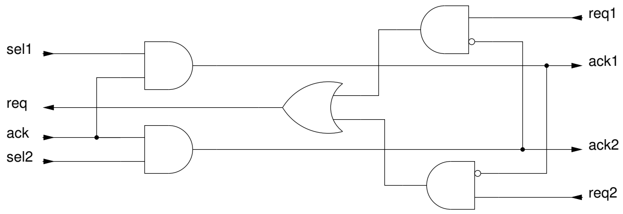

Figure 4.10: Pull branch.

includes an explicit *data valid* signal on the output port, since the `sel1` and `sel2` will be treated like bundled data.

The branch circuit accompanying the pull arbiter is shown in figure 4.10. The branch also expects `sel1` and `sel2` to be bundled data. Had they been encoded for delay insensitivity using dual-rail, the two AND-gates on the left should be replaced with C-gates.

4.2.4 Delay-Insensitive Encoding and Decoding

All link implementations presented here use a delay-insensitive encoding on the long wires. For the data part a 1-of-4 encoding as presented in [3] is used. 1-of-4 encoding is used partly because encoding and decoding is easy, and because repeater stages for this encoding are relatively cheap to implement[4]. 1-of-2 encoding which is also known as dual-rail encoding has an even simpler encoding and decoding, but suffers from a doubled energy consumption compared to 1-of-4 and has therefore been discarded.

Encoding from bundled data protocol to 1-of-4 is handled by a single cell from the cell library as shown in Figure 4.11. This encoder requires *extended early* data-valid scheme on its input to ensure that the value on `A1` and `A2` is not changed while `EN` is high. DE24HS outputs are inverted which means that an empty codeword is represented with Z1N to Z4N being logical high. It is assumed that the DE24HS outputs are hazard free on transitions on the

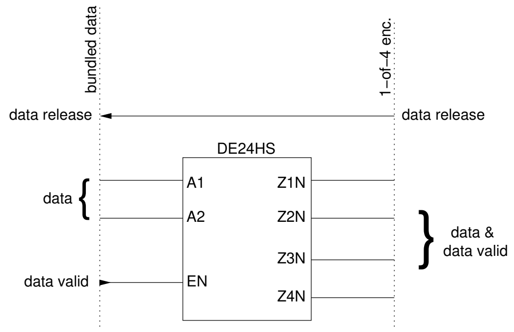

Figure 4.11: 1-of-4 encoding is done by the cell DE24HS.

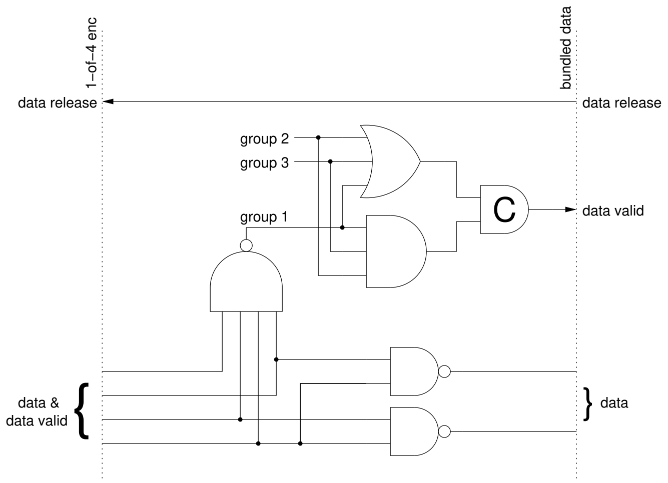

Figure 4.12: 1-of-4 decoding with completion detection.

`EN` signal.

Decoding the 1-of-4 signal back to bundled data in the receiving end also involves completion-detection. A decoding circuit is shown in Figure 4.12 which indicates a 6 bit data-path(3 groups of two bits). Fan-in of the OR and the AND gate will be $W/2$. When $W > 16$ this is implemented as two trees of AND and OR gates, since the largest AND/OR gate in the cell library has a fan-in of 8. The 1-of-4 decoder guarantees the *early* data-valid scheme on its output.

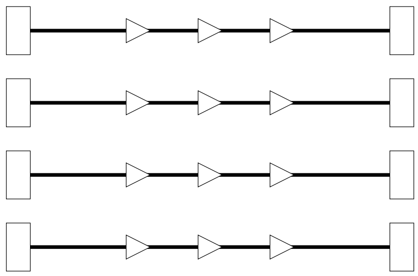

Figure 4.13: Link channels implemented as actual physical channels.

4.3 Physical Channels

The first design strategy presented here is to use actual physical channels to obtain multiple channels on a link. The motivation for investigating this strategy, is that the concept of virtual channels is inherited from multiprocessor interconnection network, where link wires are relatively expensive compared to router logic and buffers[12]. In on-chip network this relation may however have changed, since there is a quite large amount wiring resources on a chip when compared to inter-chip wiring. In [15] it is proposed to use a 300 wires in each link in a NoC to realize a 256 bit wide flit. This is quite different from most multiprocessor system which uses a channel width of 8 or 16 bits[11]. This implementation proposes to split the large wiring resources into several narrow channels to avoid the control logic implementing link level flow control. Thereby trading a lower bandwidth of the individual virtual channel for reduced energy consumption and increased aggregated bandwidth. The implementation will be used as a reference point for performance and cost comparisons between the implementations. When referring to this design we will call it *Implementation 1* or *imp. 1*.

The structure of the link is outlined in Figure 4.13. The wire count in this link is linear dependent of the N and therefore it is infeasible to use this implementation when large number channels is needed. For small numbers of channels it might however be a good solution because of its simplicity.

All channels in the link are identical and each has it own set of resources. The structure of a single channel is outlined in Figure 4.14. It consists of a passivator and the delay insensitive encoding and decoding. The box in each end of the channel indicates the environment in form of input and output buffers. These buffers are not included in the actual implementations. The dotted rectangle covers the long wires between the nodes, and everything on the left of rectangle is placed in the sending end of the link, and everything on the right of the rectangle is belonging to the receiving end. These conventions

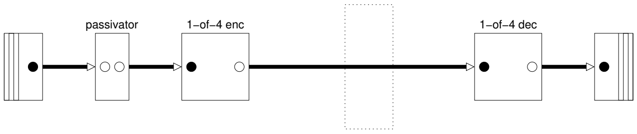

Figure 4.14: A single physical channel.

is used in all design drawings presented here.

The passivator input port is connected directly to the link input and therefore at least a broad data-valid scheme is required on the link input(see Section 4.2.1). The passivator output is however connected to the 1-of-4 encoder which requires at least *extended early* data-valid scheme as described in Section 4.2.4. This conflicts with the *early* scheme guaranteed by the passivator. The *extended early* requirement from the 1-of-4 encoder will only be met if an isochronic fork is assumed for the passivator acknowledge outputs. In the prototype implementation this assumption does not hold because of large fanout on input to the 1-of-4 decoder, but the conflict is masked by a data-valid period in the link input which is longer than the required *broad* scheme. This is actually a realistic situation if the input buffers are using edge-triggered registers, but this timing assumption must be verified when the link has been inserted in a NoC. Another way to solve the conflict is to increase the data-valid period from *early* to *broad* between the passivator and the 1-of-4 encoder. As described in [22] the conversion from *early* to *broad* on a pull channel can only be performed by latching the input data.

The 1-of-4 decoder is connected directly to the link output which means that this implementation guarantees *early* data-validity on its output.

This implementation suffer from high interconnect usage when many channels is required, and bad utilization of these wiring resource if the majority of bandwidth requirements is concentrated on small subset of these channels.

4.4 Virtual-channels with Multiplexed Data-path

To avoid the problems of high interconnect usage and bad utilization, several *virtual* channels can be multiplexed onto a single *physical* channels. This is the strategy for the link implementation presented in this section, and the concept is illustrated in Figure 4.15. When referring to this design we will

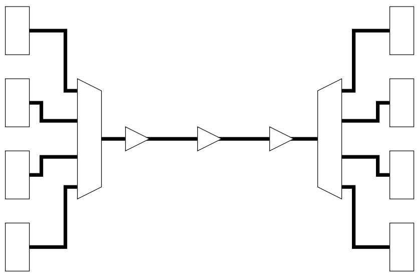

Figure 4.15: Link implementation with multiplexed datapath.

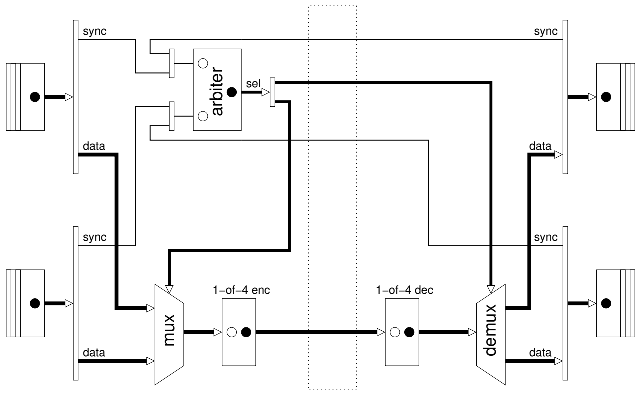

Figure 4.16: Static data-flow structure of the link.

call it *Implementation 2* or *imp. 2*.

Since the data-path is multiplexed, link level flow control must be employed. Different strategies for link level flow control exists, e.g. random, round-robin or priority. This implementation will support an encapsulated flow control module. This module can be replaced to support one of the strategies just mentioned, but the performance results presented in the next chapter will use the handshake arbiter illustrated in Figure 4.8 for flow control. The flow control strategy offered by this handshake arbiter is rather "unfair" and may result in poor network characteristics, but as assurance that only a single channel is selected at any time, the handshake arbiter behaves correctly.

Figure 4.16 presents a overview of the second link implementation in a version with two virtual channels. As in the previous section, the dotted rectangle is placeholder for the long wires between the sending and the receiving end.

We will give a brief introduction to the circuit by going through a single flit transfer on the link. When employing link level flow control it must always be ensured that the receiving end has buffer capacity to store the flit, before it is sent of from the sending end. This is assured by the join elements joining the `sync` handshake channel from the sending end with the `sync` handshake channel from the receiving end. When a request signal is present from both `sync` handshake channels, this virtual channel can engage in the arbitration for the physical channel. The arbiter ensures that only one channel is selected and outputs the selection as a 1-of-N encoded signal which is forked to the multiplexers in the sending end and the demultiplexer in the receiving end. The multiplexer select the correct data value and passes it on to the 1-of-4 encoder.

The receiving end has two delay insensitive signals as input from the sender; the 1-of-4 encoded data and the 1-of-N encoded virtual channel select signal. The data is decoded back to bundled data and the demultiplexer forwards the data to the correct virtual channel. When the output buffer has accepted the data it will take down the request signal, which will send back acknowledge on the data channel and initiate the *return to zero* cycle.

The acknowledge signal in each of the `sync` handshake channels from the receiving end is redundant, since acknowledgment of the synchronization is carried implicit in the `sel` handshake signal. Therefore the actual circuit implementation has been optimized to remove these redundancies and this reduces the number of link-wires by N. Each physical channel will have two wires per bit in the data-path, two wires for each virtual channel and a single wire for the acknowledge on the `sel` handshake channel.

Just as imp. 1, this implementation includes link wires in the handshake cycle. When the latency in these wires increase in future technologies, this will result in long cycle times, and a large part of the circuit being inactive most of the time. The last design proposal will solve these problems.

4.5 Virtual-channels with Pipelined Data-path

The last design strategy is to use pipelining to improve throughput and circuit utilization on a link with multiplexed data-path. The concept is illustrated in Figure 4.17. When referring to this design we will call it *Implementation 3* or *imp. 3*.

In multiprocessor networks it is not possible to pipeline the network links since they are just plain cables, but in a on-chip network, link wires are routed on top of silicon which easily can be used for pipeline buffers. A prerequisite for gaining performance through pipelining is that the delay in the pipeline latches them selves remains small compared to the delay in

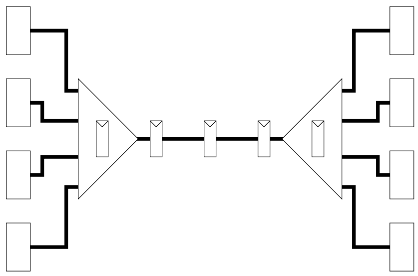

Figure 4.17: Link implementation with multiplexed and pipelined datapath.

the wires/combinatorial logic which is between the latches. Otherwise the penalty in latency and power consumption will overrule the advances of an increased throughput. Technology scaling is constantly increasing delay of global wires relative to gate delays and therefore wire pipelining will be more and more feasible in the future. A minimum sized corner-to-corner wire in 50*nm* technology is expected to require 138 repeaters for optimal delay, whereas the 180*nm* technology used here only requires 22[33].

Figure 4.17 hide the fact that input and output ports of a virtual channel must be synchronized before a transfer is started, to prevent the link from being blocked. A synchronization channel (without pipelining) for each virtual channel will make this assurance. Figure 4.18 show the structure of the circuit in a link with two virtual channels. As in the first implementation, a passivator is used to connect the push and the pull side of the circuit. Therefore *broad* data-validity is required at the link input. Each virtual channel also has a *fork* and a *join* component which splits data from synchronization in the sending end, and merges them back together in the receiving end. The fork transfers the data-valid scheme from its input to its output, and therefore the input to the funnel component is *early*. Even though the output from the horn component is *extended early* as we will see shortly, the join will degrade data-validity at the link output to *early*[28].

By pipelining the `data` handshake channel it is possible to merge several virtual channels onto the same physical channel and let them share the increased bandwidth. The *funnel* and the *horn* components are responsible merging and branching the `data` handshake channels, and Figure 4.19 shows the internals of these components for a link instance with 4 virtual channels. The construct is similar to the FLEETzero switching network presented in [7]. This Funnel-Horn network does however not make any switching, since flits are expected to arrive at the same handshake channel as they are inserted to.

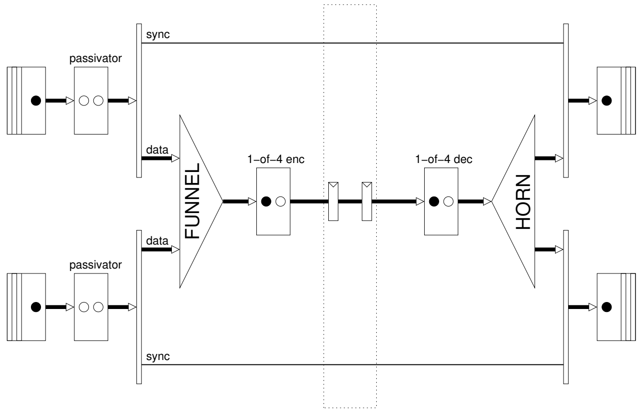

Figure 4.18: Link implementation using a pipelined data-path to increase overall throughput.

In the *funnel*, all input handshake channels are merged to a single handshake channel using a binary tree structure. At each level of the tree, two extra wires are added to the data-path, and these wires tell from which subtree(left or right) the flit originated from. This information is dual-rail encoded all the way from the merge to the branch module, but in the funnel and horn latches these select wires are treated as bundled data to avoid completion detection. The merge element must be arbitrating since the `data` handshake channels coming from different virtual channels are not mutual exclusive. Each merge element is constructed from the pull arbiter in 4.9 and a combinatorial multiplexer. In the *horn* which is placed in the receiving end, a similar tree of branch elements will guide the flit to the correct output port based on the extra data added in the funnel. The branch element is constructed from the pull branch in Figure 4.10 and a combinatorial demultiplexer.

In a pipelined circuit it is the "slowest" stage that will determine the performance of the whole circuit, and therefore the funnel and the horn components has pipeline latches inserted at each level in the tree. The physical link is pipelined using the latch described in Section 3.6.3, whereas the funnel and the horn are pipelined using a bundled-data latch which is shown in Figure 4.20.

The latch controller used inside the funnel and the horn is the *simple* latch controller shown in Figure 4.21. This latch controller has a tight coupling between input and output side which means that there is unnecessary dependencies between the handshakes on the input and the output. These dependencies will break the merging mechanisms described above, because a

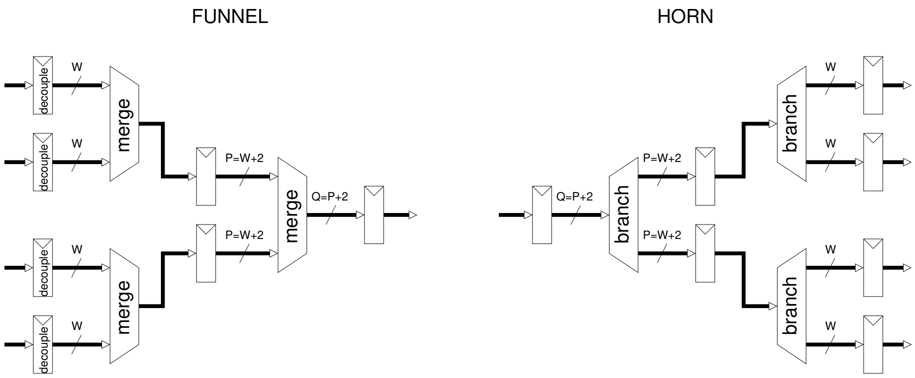

Figure 4.19: Funnel and horn structure with four virtual channels.

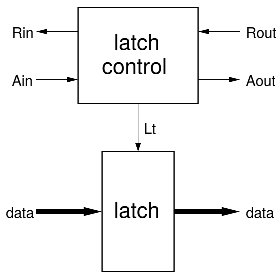

Figure 4.20: A bundled-data latch as used in the *funnel* and the *horn*.

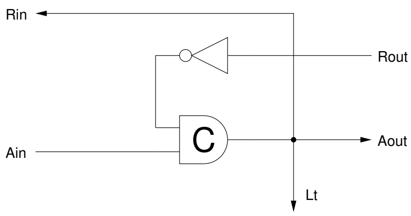

Figure 4.21: A simple latch controler.

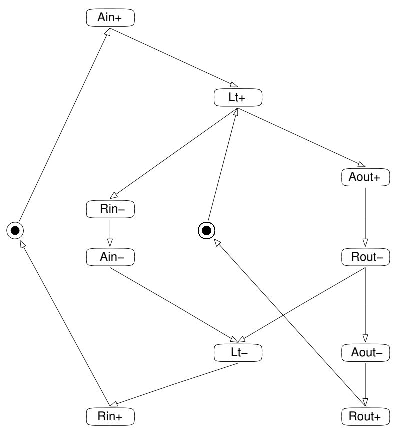

Figure 4.22: Latch controller for the decoupling latch at input to the funnel.

virtual channel will not release the physical channel before the RTZ part of the synchronization channel has started, which will not happen before the flit has been transfered to the other end of the link. Therefore a latch which is able to decouple the handshake on the output port is inserted at each input in the funnel component. A latch controller with these capabilities is presented in [17]. Small adjustments has been made to fit it into a pull channel, and the resulting STG is depicted in 4.22. The STG presented in [17] has explicit added a internal variable to ensure *complete state coding*(CSC)[9], whereas the STG in Figure 4.22 rely on Petrify to solve the CSC-conflict.

As seen in the STG in Figure 4.22, the decouple latch assumes *early* on its input and produces *early* on its output. The merge and branch component will produce *early* on the output when *early* is provided at the input. The "simple" latch assumes *early* on the input but produces *extended early* on its output[28]. This ensures that data-validity is correct at the input of the 1-of-4 encoder.

The flow control offered by the funnel-horn construct is a form of round robin. If all channels are eager to transmit, they will share the link bandwidth equally. This is caused by the tree structure of the arbiter circuits, and the fact the arbiter will alternate between the inputs if both are eager. These properties of the of the funnel-horn can be used to differentiate the service guarantees on the channels. By making the tree unbalanced, some channels connected closer to the root of the tree will obtain a larger share of the link bandwidth. The concept is illustrated in Figure 4.23 where one channel is guaranteed half the link bandwidth, one channel is guaranteed $\frac{1}{4}$ and the last two are guaranteed $\frac{1}{8}$. Any of the channels can however obtain a larger part of the bandwidth than it is guaranteed, if other channels are not using their

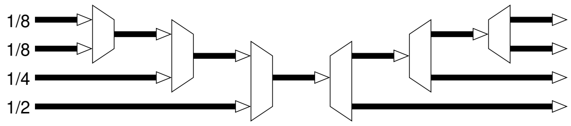

Figure 4.23: Funnel and horn in an unbalanced tree structure.

share. Therefore this form of guaranteed service will suffer from more jitter than guaranteed service based on time-slot reservations. An upper limit for throughput on a single channel will however be imposed by the `sync` channel.

The physical channel will have two wires for each bit in the data-path, two wires for each virtual channel, $2 \times log_2(N)$ wires for virtual channel identification and a single wire for acknowledge in the pipeline.

This implementation has the disadvantage that it only improves aggregated throughput of the link and not the throughput of a single channel. This is a limitation caused by the decision to leave output buffers out of the link implementations, but it will not prevent us from estimating performance of an implementation without this disadvantage. We will come back to this in Section 5.7.

Chapter 5

Results and Discussion

This chapter will present performance and cost measurements for the link implementations presented in previous chapter.

All measurements are based on simulation of the link implementations with back-annotated pre-layout timing information. Each simulation trial consists of 1000 flit transfers on each eager channel. There has been conducted experiments with larger simulation datasets, but the changes in results were insignificant, and larger datasets made it infeasible to perform simulations within reasonable time. Flit payload data is random in all simulations except dynamic power measurements which is also performed with all-zero payload.

The variable parameter(number of channels, number of repeaters, flit-width) in the simulation trials is using powers of 2 to be able to cover a larger interval without excessive amount of simulation. Also, implementation 2 and 3 only supports channel counts which are a power of 2, because of recursive definitions of binary tree-structures.

5.1 A sample on-chip network

To put the area and power measurements into perspective a sample NoC will be presented here. The sample NoC is purely imaginative, but the design decisions for the network will be based on the motivations presented in Chapter 2, or by using decisions for similar sample network presented by others. We will assume that basic structure is similar to the example presented in [15]. The NoC system have 16 modules connected by a folded torus network as shown in Figure 5.1. All links in the network are bidirectional but communication in each direction is unrelated and therefore one bidirectional link can be looked upon as two unidirectional links. The total number of

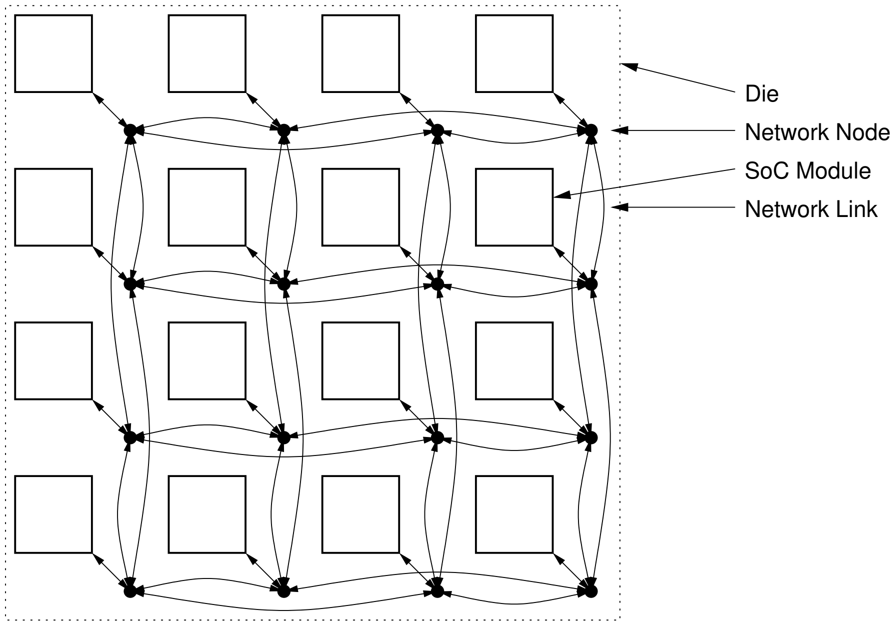

Figure 5.1: A NoC sample.

unidirectional links is 64. As seen in the figure, a folded torus topology will result in varying link-lengths when laid out in 2D. The length of the longest links in a folded torus network can be calculated as:

$$2 \times \frac{S}{\sqrt{N}}$$

where N is the number of nodes in the network and S is the length of the die side. The long links can use scaled-up wires if uniform delays in all links is important. The longest links in this sample NoC will bee half the length of a die side.

We assume that the sample network system is using the $0.18\mu m$ VLSI design platform HCMOS8[30], which is the platform used in the link implementations. This platform is nearly five years old and already has two successors, HCMOS9 which is a $0.13\mu m$ technology and COMS090 which is a $90nm$ technology, and therefore the sample NoC will only be used as a reference point for projecting performance and cost in contemporary or future technologies.

It is hard to predict module size for future SoC design, and the optimal size will probably be very dependent on the application system. For the sample NoC we will assume a module size of $500K$ gates. The HCMOS8D has an average gate density of $85K/mm^2$, so with the assumption of $500K$

gates in each module, a module will take up $5.8mm^2$ of die area. With 16 nodes laid out as uniform tiles on the die plus some overhead for the network, we end with a $100mm^2$ die. The long links in the sample network will be $5mm$ and a total link length is $64 \times (1/2 \times 5mm + 1/2 \times 2.5mm) = 240mm$. According to [33] the optimum distance between repeaters on global wires in a $0.18\mu m$ technology is approximately $2mm$. The long links in the sample network would therefore require one or two repeaters for minimum latency.

We will assume a flit width of 16 bit and we will assume that each link has 16 channels; 8 channels to support adaptive best effort routing, and 8 channels for guaranteed service reservations.

If nothing else is mentioned, the results presented in the following section will use a 16 bit wide data-path, 16 link-channels and use 2 repeater/pipeline stages on the physical wires between the link ends.

5.2 Performance

As mentioned earlier, *latency* and *throughput* are the two key performance measures for networks and network components. The performance measures presented here will mainly focus on throughput defined as "transfered flits per second", but latency measures will be presented in the end of Section 5.2.1.

When a link implements virtual channels, the performance seen by a single channel will be highly dependent on activity on the others. In the next section we will investigating the situation where only a single channel on the link is active, and Section 5.2.2 will investigate the situation where several channels are active.

5.2.1 Unloaded Link

In this section *cycle time*[28] will be used to describe the performance, and we will define it as

$$cycletime = \frac{1}{throughput_{(flit/second)}}$$

on a unloaded link. We will use *cycle time* instead of *throughput* because the phenomenons that we will spot in the performance graphs will be related to circuit *delays*, and therefore easier to see in a time-graph.

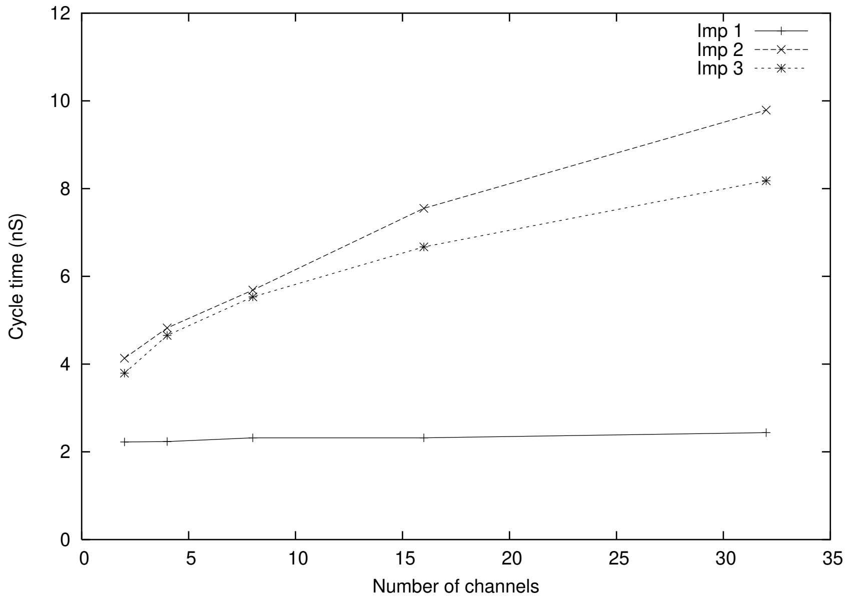

Figure 5.2: Cycle time on an idle link as a function of the number of channels.

Throughput with Varying Channel Count

The graphs in Figure 5.2 show how cycle time depends on the number of channels in the link. In implementation 1, one would expect that the cycle time is independent of the number of channels on the link, but Figure 5.2 show that the cycle time is slightly increasing. This is a result of the first order gate-delay estimations made by DC. When estimating wire lengths it is not only the fanout of the wire it selves, but also the total number of gates in the module that has influence on the estimate. In an optimal layout of this implementation, the cycle time/throughput would be independent of the number of virtual channels. The effect of slightly increased gate-delays as the the number of gates in the link increases will be present in all the following results.

The cycle time of implementation 2 and 3 is increasing logarithmic with the number of virtual channels. This is caused by the added delay in the arbitration circuits, and it increases logarithmic because the arbiters are implemented as trees. The graph shows that there is a significant throughput degradation when implementing *virtual* channels. The next section will investigate if the lost bandwidth in a virtual channel link implementation can be regained by using a wider flit. The performance of imp. 3 is a little better than imp. 2, and the difference is increasing with the number of channels.

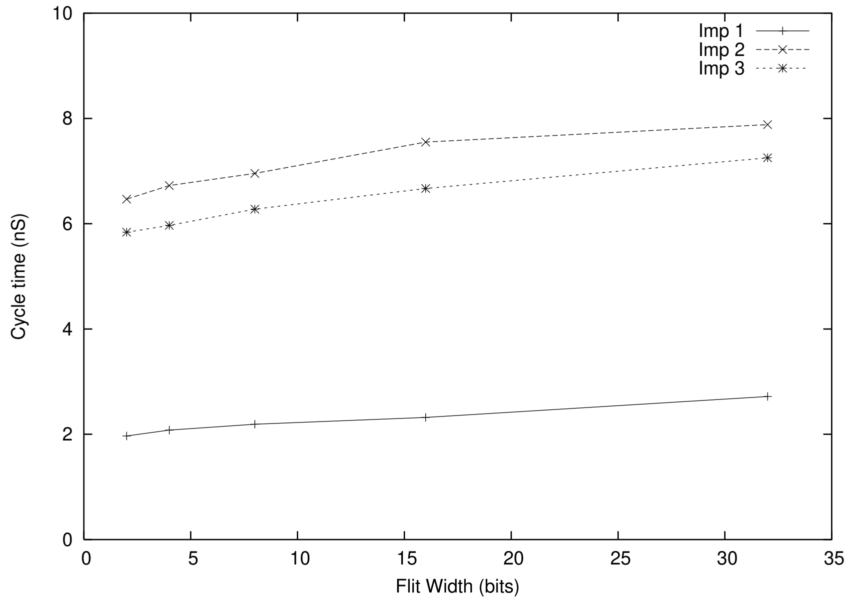

Figure 5.3: Cycle time on an idle link as a function of the flit width(16 channels).

Throughput with Varying Flit Width

The flit width will also affect link performance. In the bundled-data part of the links, a wider data-path increases the load on the control circuits driving latches and multiplexers. The latency caused by the increased load can be minimized by scaling the driver gates[32]. In the delay insensitive part of the circuits, a wider data-path will also cause increased latency because the hole data-path is synchronized at each pipeline latch. It may be possible to avoid this by dividing the data-path into smaller gangs at the cost of extra acknowledge wires. The performance measurements from simulation with varying flit width is shown in Figure 5.3. As expected we see a logarithmic increasing cycle time caused by the completion-detection tree getting deeper as the data-path becomes wider. Above 16 bit the slope is however rather flat, which means that it *is* feasible to increase link bandwidth by widening the data-path. For imp. 3 the cycle time is increased by approx. 10% when doubling the flit width from 16 to 32 bit.

Throughput with Varying Link Length

Future chip technology advances will increase delays in global wires compared to gate delays, and here we will investigate how these changes will affect

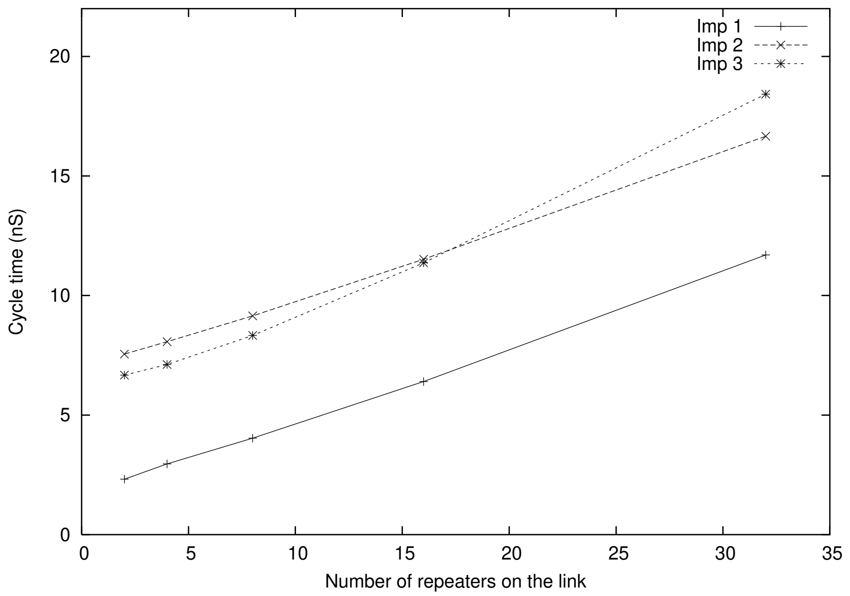

Figure 5.4: Cycle time on an idle link as a function of the number of repeaters on the link(16 channels, 16 bit data).

link performance. Since simulations are performed with pre-layout timing informations, the delays in link wires are not realistic. Therefore longer link wires will be emulated by increasing the number repeaters between the sending end and the receiving end. As seen in Figure 5.4 the cycle time on the link is linear dependent on the the link wire delay. It is obvious that the graph for imp. 3 is not what we desired from a pipelined version. As described earlier, this is caused by the fact that a single channel can not exploit the pipeline because it is limited by the synchronization handshake channel. We will come back to this problem in Section 5.7. The graph has a steeper slope for imp. 3 than imp. 2 because the delay through a pipeline latch is longer than through a simple buffer.

Latency

At network level, *latency* describe the time passing from a packet is sent until the packet is received. Often a packet is divided into several flits and therefore packet latency at link level will depend on both link latency and cycle time. We will defined link *link latency* as the time passing from valid data and request signal is asserted at the input of a channel, to the data is available at the output and the acknowledge signal goes high. Table 5.1 lists

Implementation	N	Cycle time(nS)	Latency(nS)
imp1	2	2.22	0.88
imp2	2	4.13	1.13
imp2	32	9.79	3.35
imp3	2	3.79	2.38
imp3	32	8.18	6.67

Table 5.1: Cycle time and latency for different link instantiations.

cycle time and latency for some link instances. In implementation 1 and 2 the difference between cycle time and latency is rather high. This is because latency only includes the *forward latency* of the circuit, whereas cycle time includes the full handshake. In implementation 3, the cycle time is closer to the latency because decoupling in the pipeline lets the RTZ part of the handshake take place concurrently with the data-transfer. The latency in imp. 3 is approximately 2 times the latency of imp. 2. This is due to the forward latency added by the pipeline latches.

5.2.2 Bandwidth Sharing

Now we will investigate how the performance is affected when several channels on the same link is transmitting. In implementation 1, the aggregated throughput of the link is simply the throughput shown in Figure 5.2 multiplied with the number of channels. The total link throughput of implementation 2 and 3 when all channels are eager to transmit is shown in Figure 5.5 together with the throughput of a single channel. In implementation 2 the aggregated throughput is higher than what a single channel can achieve alone, because the waiting channels are able to do some initial steps in the handshake before they reach the arbitration.

Implementation 3 has a considerably higher aggregated throughput when the number of active channels is raised. This is due to a better utilization of the data pipeline. The agregated throughput should have been constant when the number of virtual channels is increased, but it shows a slight decrease. This is again caused by the first order delay estimations made by DC. In imp. 3 link instances where $log_2(N) > W/2$, completion detection of the select signal will be in the critical path and total throughput will degrade with higher channel count.

How the bandwidth shown in Figure 5.5 is divided between the virtual channels can be seen in Figure 5.6. This figure show throughput on each channel for a link instance with 8 virtual channels where all inputs are eager

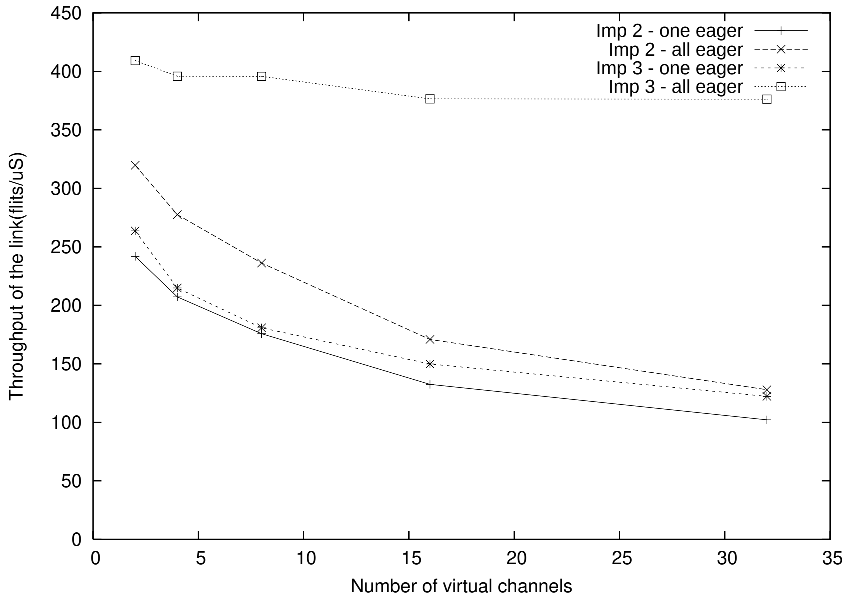

Figure 5.5: Total throughput of the link.

to transmit, but output 2 and 5 will not accept flits. As described earlier, imp. 2 is unfair because only two channels are allowed to communicate. It is random which couple of adjacent channels are chosen.

In imp. 3 the bandwidth is shared among all the channels willing to transmit, but channels placed in a subtree together with blocked channels will take over the unexploited bandwidth. Therefore channel 1 and 6 have doubled throughput.

The results in Figure 5.5 are from a link with only two repeater stages on the physical channel. Therefore it is possible for two active channels to

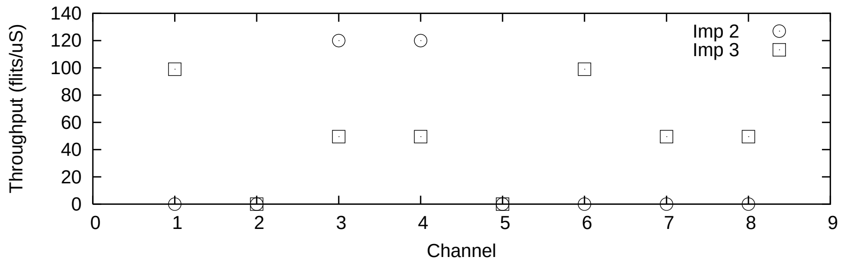

Figure 5.6: Sharing of bandwidth on a link with 8 virtual channels. All but channel 2 and 5 are eager.

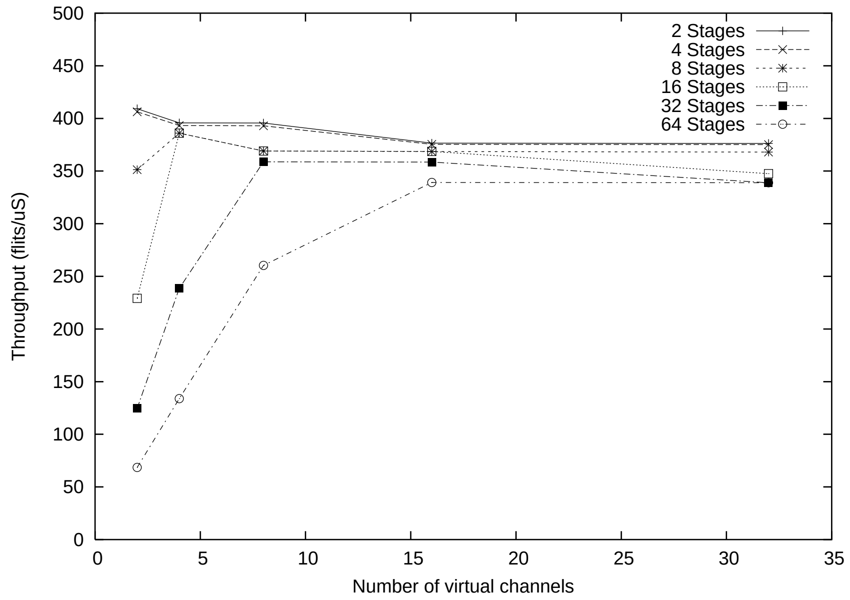

Figure 5.7: Total throughput of imp. 3 with varying number of virtual channels.

fill up the pipeline. Figure 5.7 shows how the total throughput of imp. 3 is affected when the number of pipeline stages in the link is increased. The graphs show that a long pipeline requires more channels to be fully utilized. If the link has too few channels the pipeline becomes data-limited[28]. Link instances reach full utilization of the pipeline when the number of channels is higher than 1/4 of the number of pipeline stages. This means that only about every fourth pipeline latch holds a data element, meaning that the pipeline has a dynamic wavelength of approximately 4[28]. The variations in throughput of link instances with 32 channels is again caused by the first order delay estimations made by DC.

5.3 Area

Now we will compare the area cost-parameter for the link implementations and investigate how it is affected by the number of channels on the link. The area consumed by a NoC link is divided in cell area and interconnect area.

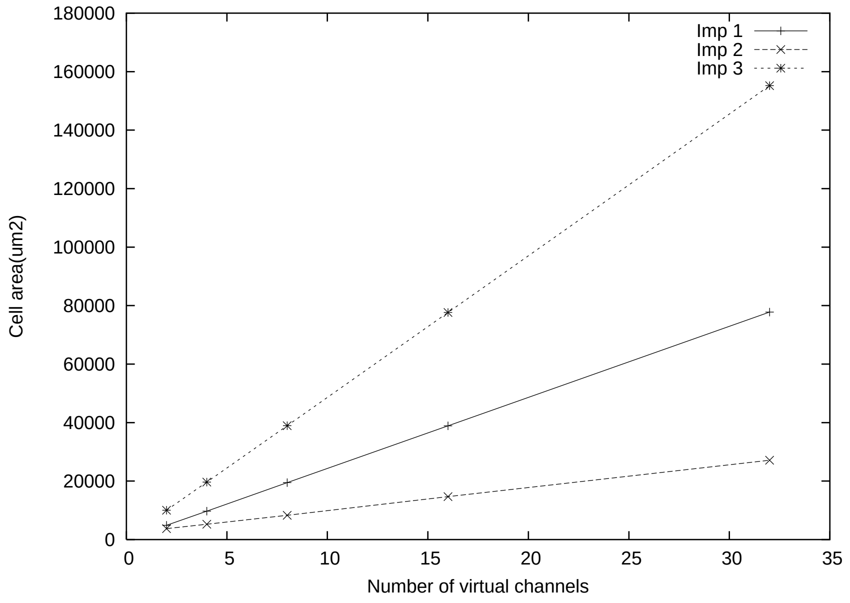

Figure 5.8: Total cell area of one link.

5.3.1 Cell Area

The cell area reported by DC for link instances with varying number of channels is shown in Figure 5.8. These area measurements include both link logic placed in the network nodes, and the logic implementing the repeater stages. Implementation 3 is the most expensive in terms of cell area because of the pipeline latches. If we assume that the sample network presented in Section 5.1 requires 16 bit flits and 16 channels on each link, then implementation 1 will occupy $(64 \times 38000\mu m^2)/100mm^2 = 2.4\%$ of the total die area, implementation 2 will occupy 0.9%, and implementation 3 will occupy 4.9%.

When in future the feature size decreases two scenarios can be imagined. Either both network and modules scale down or module complexity increases resulting in increased communication demands. The latter will require wider links. Therefore, in both cases, the link occupation of cell area will remain at the level described above.

5.3.2 Interconnect Area

The area estimations just presented do not take interconnect area for the physical channels into account. The number of wires on the link was given for each implementation in the previous chapter. We will assume that all link

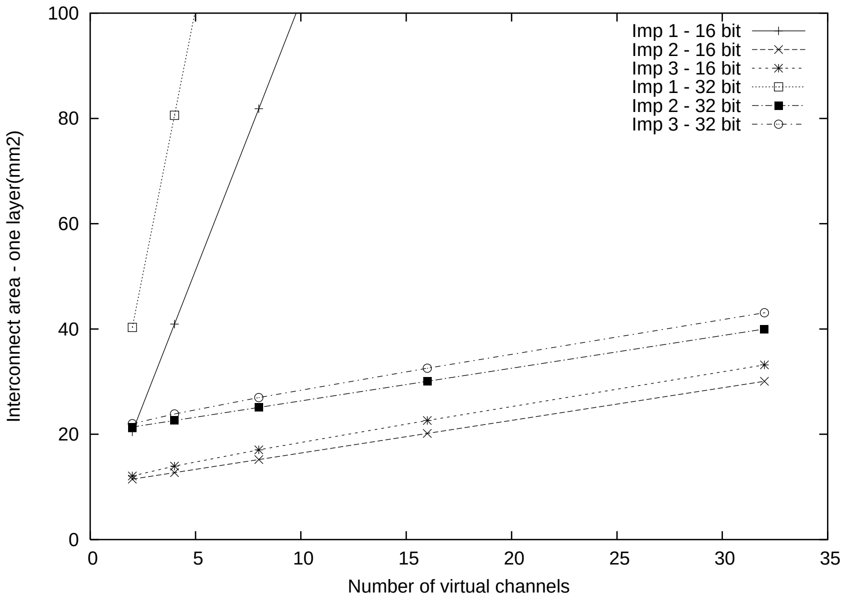

Figure 5.9: Total interconnect area occupied by the sample NoC. Only one metal layer used.

wires are routed in one of the upper metal layers(*metal5* or *metal6*) which has a minimum width and spacing of $0.64\mu m$. Given that the total length of all links in the sample NoC presented earlier is approximately $240mm$, the interconnect area occupied by each wire in a link will be $240mm \times (64\mu m + 64\mu m) \approx 0.31mm^2$(the total die area is $100mm^2$). Figure 5.9 shows the interconnect-area occupied by link wires in the sample NoC. The graph shows that implementation 1 is infeasible if more than just a few channels are desired.

Implementation 2 and 3 use almost the same amount of interconnect area. Imp. 3 uses a little more because of the channel identification which is added to the data-path. With two virtual channels and 16 bit flits, approximately 10% of an upper metal layer is used for NoC interconnect.

Since the global interconnect wires do not scale down and future NoC will have higher communication demands, the link wire occupation of interconnect area will increase.

5.4 Energy

We will now investigate another cost-parameter for the links, namely energy consumption. First we will look into dynamic energy caused by network activity and then we will consider idle power consumption in form of cell leakage.

5.4.1 Dynamic Energy

Figure 5.10 shows how much energy it takes to transfer one flit from one end to the other in each link implementation, and how data-dependent the energy consumption is. The measurements include only switching power in cells and in local interconnect, and not power dissipation in the link-wires. Energy consumption in the data-path link-wires is data independent since 1-of-4 coding is used. In implementation 1 and 2 the link-wires will add a constant contribution, since all control informations on the link are one-hot coded. For implementation 3 the link-wire contribution will be slightly increasing because the select signal is encoded as a $log_2(N)$ bit dual-rail gang.

As expected we see that the energy consumption on imp. 1 is unaffected by the number of channels on the link. The energy consumption on imp. 2 is increasing slightly because of added complexity in arbiter and multiplexer circuits. Increased multiplexer complexity is also the reason of larger data-dependence. For imp 3 we see that it is quite expensive in terms of power, to pipeline data on the link. There is a logarithmic increase of energy consumption, since each time the number of channels is doubled, another level is added to the funnel and horn structure, and therefore two extra pipeline stages are added to the path a flit must travel. Imp. 3 also has larger and increasing data-dependence because the pipeline registers in the funnel and horn is using bundled data protocol.

The dynamic energy consumption in cells and local interconnect will scale down with feature sizes, but since global interconnect does not scale, the energy consumption in link wires will relatively increase in the future.

5.4.2 Leakage Power

Cell leakage power for the link instances with varying number of channel is shown in Figure 5.11. As expected the leakage power is proportional to the cell area shown in 5.8. If the sample network is using imp. 3 links and 16 virtual channels, the total leakage power for the NoC links will be $64 \times 5\mu W = 0.32W$. Compared to the switching power of the link transmitting

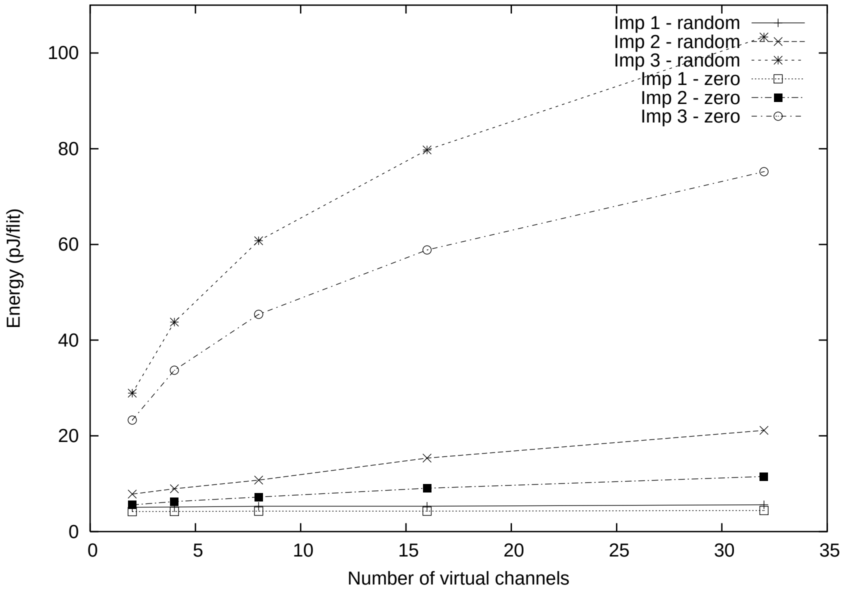

Figure 5.10: Energy consumption for transporting one flit through the link. Flit payload data is either random or all-zero.

at maximum throughput, the leakage power is rather insignificant(approx. 1/1000).

Future technology generations will, however, suffer from increased cell-leakage. The leakage power will therefore become an increasingly larger part of the total link power consumption.

5.5 Quality of Measurements

The measurements presented in previous sections are based on pre-layout wire estimates which may lead to some uncertainty in the results. Layout of the designs was not performed due to the limited time available for the project. It is however an interesting subject for further investigation, and especially simulation with realistic wire delays on the link wires would be valuable.

The behavioral mutex used in the simulations is also cause for some uncertainty, but since it used in both virtual channel implementations, it should not invalidate comparison of these implementations.

It has been assumed that the complex gates used in the implementations are guaranteed hazard-free. If this assumption does not hold, these gates

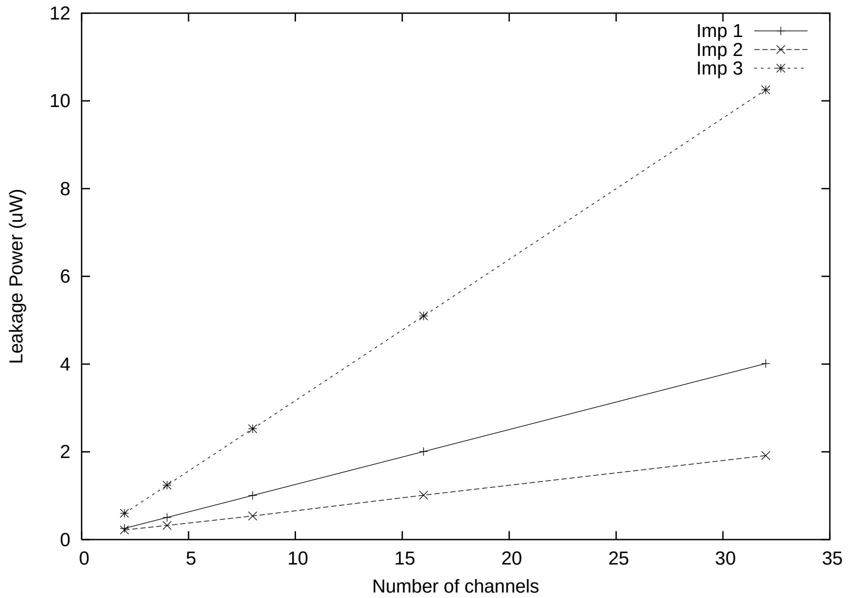

Figure 5.11: Leakage power in link instances with varying number of channels.

must be replaced by constructs of simple gates, which may increase latency, area and power consumption.

5.6 Comparison of Virtual Channel Implementations

The results in previous chapter shows that generally imp. 3 has better performance than imp. 2. The performance benefits of implementation 3 are particular interesting on long links when pipelining is fully utilized. For instance in a link with 32 repeater stages and 16 virtual channels, imp 3 has over 5 times higher aggregated throughput than imp. 2.

The increased throughput offered by imp. 3 comes with the cost of larger cell area and energy consumption because of the pipeline latches. These latches may however reduce the requirements for buffers in the link ends, and thereby make the actual area/energy penalty of imp. 3 less significant.

Imp. 2 has the advantage of using an encapsulated module for arbitration which can be replaced if a different flow control mechanism is wanted. In imp. 3 the arbitration is distributed in the funnel component, and the offered flow control depends on the structure of the funnel tree. The arbitration module used for imp. 2 in previous chapter is “unfair” since two eager channels can starve all others forever, which is probably unacceptable in most networks.

5.7 Future Work

Due to the limited time available, some interesting implementation alternatives and optimizations have been left out of the project. I will mention a few possibilities here as an inspiration for further research in the area of asynchronous NoC link implementations.

Purely Delay Insensitive

It was chosen in this project to use bundled data protocols in the link-ends to reduce circuits area and to avoid the burden of completion detection. The use of bundled data protocols does however suffer from the same timing validation problems as synchronous circuits. A comparison of performance and cost of a similar but purely delay insensitive link implementation with the links presented here, would be an interesting subject for further research. Such an implementation would reduce the need for timing analysis, which would improve support for automatized link and NoC generation.

Credit based link-level flow control

Implementation 3 has significant improved aggregated throughput compared to imp. 2, but this throughput can not be utilized by a single channel. As described earlier, this is a limitation caused by the decision not to include buffers in the link-implementations. The link has no knowledge of the *number* of empty output buffers on a channel — only *whether or not* at least a *single* buffer is free. This means that *one* flit at most may be on its way across the link at any time. Not until this flit has been injected to the output buffer, and that buffer has announced its willingness to accept another flit, can a new flit from that particular channel be sent off from the sending end of the link. If output buffers were included in the link implementation, information on free buffers(credits) could be pipelined in the opposite direction of the flits. This would allow a single channel to use the full throughput of the data pipeline, if the number of buffers and pipeline stages is balanced correctly. The concept is illustrated in Figure 5.12. If a funnel-horn structure is used in the credit handshake channels, the number of link wires will be reduced from $2 \times W + 2 \times N + 2 \times log_2(N) + 1$ to $2 \times W + 2 \times (2 \times log_2(N) + 1)$, and

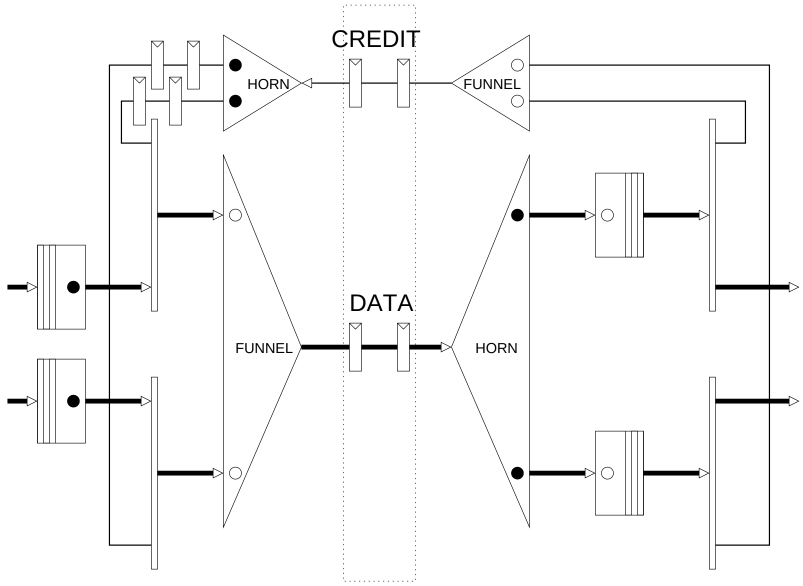

Figure 5.12: Optimized pipelined link with credit system.

thereby removing the linear relation between number of virtual channels and number of link wires.

Low Level Optimizations

None of the circuits presented here have been subject to low-level optimizations since the goal was to *compare* different implementation strategies, and not to come up with a single optimized solution. All circuits could however benefit from different forms of optimization, and I will mention a few possibilities here.

One way of improving the circuits is to implement critical parts of a design using custom design at device level. The C-element which is widely used in all link designs and currently implemented using a complex gate, would be an obvious choice for such an optimization.

In [32] is presented a design process for creating complete asynchronous circuits using custom layout for maximum performance. A family of FIFO control circuits called GasP, which is using this design process, is presented in [31]. These GasP circuits are used in the FLEETzero[7] chip mentioned earlier. Some impressive performance is shown for the FLEETzero chip. With up to $1.55 GDI/s$ on a $0.35 \mu m$ technology, it indicates that there is plenty of room for performance improvements of the circuits presented here.

All control circuits in the link designs use the 4-phase handshake protocol. This protocol has redundant signaling which increase the latency and energy dissipation[28]. The 2-phase handshake protocol, which has no redundancy, does increase circuit complexity, but it may be viable in the link designs because they contain no computation.

Chapter 6

Conclusion

Three asynchronous link designs for on-chip network have been presented. For each design a customizable standard cell implementation has been created. Customizability has been achived by embedding GNU m4 macro definitions and calls in the HDL files. With refinements this approach might be usefull for defining complete NoC implementations.

Via an extensive set of simulation trials, these implementations has been used to evaluate the link designs on cost and performance parameters. Which of the implementations to choose for a given on-chip network depends on the requirements for the system and properties of the technology in which the system is implemented. If only a few channels are needed on each link, and global interconnect is not the limiting factor in the system, then implementation 1 is the best choice. However, global interconnect *is* projected to be the limiting factor in future technology, and therefore imp. 1 will become infeasable.

If latency on link wires is short, imp. 2 will provide comparable performance with imp. 3, but has a significantly lower cell area and energy consumption than imp. 3. When wire delay increases in the future, performance of imp. 2 will degrade, and imp. 3 will become the best choice. Implementation 3 can be used to provide differentiated service guarantees to the virtual channel on the link, as proposed. The drawback of implementation 3 consists in that a single channel can not utilize the increased throughput. This issue must be addressed.

In the two virtual channel implementations are cycle time and energy consumption increasing logarithmicly with the number of virtual channels on a link, whereas logic area and interconnect area are increasing linearly with the number of virtual channels. Since future technology will be wire limited, the linear increase of interconnect area represents a problem for implementation of large numbers of virtual channels. Given that this problem will be addressed, and that logic area will be a relatively cheap resource in future technology generations, these results show that it will be possible to implement a large number of virtual channels at a relatively low cost.

Appendix A

Design Flow Scripts

This appendix lists a few design-flow scripts. The rest is found on the enclosed CD.

Project Makefile

```
CONFIG_FILE = config
include $(CONFIG_FILE)
M4 = m4 $(shell awk '{print "-D" $$0}' $(CONFIG_FILE))
SYNOPSYS_WORK = synopsys-work.tmp
SYNOPSYS_OUT = synopsys-out.tmp
MODELSIM_OUT = modelsim-out.tmp
DATA_DIR = data.tmp
SIMULATION_OUT = $(MODELSIM_OUT)/simulation-stdout.txt
STATIC_FILES = c2.v c3.v c.v SRLATCH.v passivator.v \
fork_pull.v join_pull.v arbiter_pull.v branch_pull.v
COMMON_FILES = $(patsubst %.v.in,%.v,$(wildcard *.v.in)) od_pull_lctl.v
CHANNEL_FILES = $(patsubst %.v.in,%.v,$(wildcard channel/*.v.in))
NOSYN_FILES = $(patsubst %.v.in,%.v,$(wildcard nosyn/*.v.in))
DYNAMIC_FILES = $(COMMON_FILES) $(CHANNEL_FILES) $(NOSYN_FILES)
TESTBENCH_FILES = tb.vhd sink.vhd source.vhd testbench.vhd
TESTBENCH_OUT = $(patsubst %.vhd,work/%/_primary.dat, $(TESTBENCH_FILES))
LINK_FILES = $(patsubst %.v.in,%.v,$(wildcard link${LINK_IMPL}/*.v.in))
SYN_FILES =  $(LINK_FILES) $(COMMON_FILES) $(CHANNEL_FILES) $(STATIC_FILES)
CLASS_FILES = $(patsubst %.java,%.class,$(wildcard java/noc/analysis/*.java))
VSIM = vsim -L CORELIB8DHS -sdftyp /testbench/link1=$(SYNOPSYS_OUT)/link.sdf \
-quiet +nowarnTSCALE -t ps "work.testbench(structure)"
#PETRIFY_TM=-icsc2 -rst1 -tm2 -tm_ratio1 -nolatch
PETRIFY_TM=-icsc3 -tm2 -nolatch

all: link testbench

verilog: $(SYN_FILES)

testbench: $(TESTBENCH_OUT)

work/%/_primary.dat: %.vhd tb.vhd
vcom -93 $<

$(SYNOPSYS_OUT)/link.v: $(SYNOPSYS_WORK) $(SYNOPSYS_OUT) $(SYN_FILES)
```

```
export DESIGN_FILES="{$(SYN_FILES)}" ; dc_shell -f compile.dcsh
mv $@ $@.tmp; sed 's/\\in\[\([0-9]\+\)\]/int_\1/g' $@.tmp > $@

work/link/_primary.dat: $(SYNOPSYS_OUT)/link.v $(NOSYN_FILES)
vlog $(SYNOPSYS_OUT)/link.v $(NOSYN_FILES)

link: work work/link/_primary.dat

debug: work link testbench data $(MODELSIM_OUT)
$(VSIM) -do "debug.do"

$(SIMULATION_OUT): work link testbench data $(MODELSIM_OUT) $(CLASS_FILES)
$(VSIM) -c -do "simulate.do" -std_output $(SIMULATION_OUT)
java -cp java:lib/mysql-connector-java-3.0.11-stable-bin.jar \
 noc.analysis.SimulationAnalysis $(SIMULATION_OUT) $(CONFIG_FILE)

analysis: $(SIMULATION_OUT)

work:
vlib work

data: data.tmp data.tmp/in1.bin


power-report: $(SYNOPSYS_OUT)/link.v $(MODELSIM_OUT)/simulation.saif
dc_shell -f report-power.dcsh

area-report: $(SYNOPSYS_WORK) $(SYNOPSYS_OUT) $(SYN_FILES)
export DESIGN_FILES="{$(SYN_FILES)}" ; dc_shell -f report-area.dcsh

$(MODELSIM_OUT)/simulation.vcd: $(SIMULATION_OUT)
touch $@

$(MODELSIM_OUT)/simulation.saif: $(MODELSIM_OUT)/simulation.vcd
vcd2saif -i $< -o $@ -keep_leading_backslash

%.class: %.java
javac -d java $<

%.bin: $(CONFIG_FILE) $(DATA_DIR) generate-data.pl
./generate-data.pl $(CHANNEL_COUNT) $(DATA_WIDTH) $(DATA_DIR)/in%d.bin

%.tmp:
mkdir $@

%.g: %.stg
sed -e "/###/,$$ d" $< > $@

%.v: %.g
PETRIFY_LIB_PATH=../lib ; petrify -no $(PETRIFY_TM) -vl $@.tmp $<
sed -f fix-petrify-bugs.sed $@.tmp > $@

%.v: %.v.in $(CONFIG_FILE) macros.m4
${M4} $< > $@

%.vhd: %.vhd.in $(CONFIG_FILE) macros.m4
${M4} $< > $@

start: log db netlist

log:
mkdir log
```

```
db:
mkdir db

netlist:
mkdir netlist

clean:
rm -rf log db netlist work *.tmp $(DYNAMIC_FILES) $(LINK_FILES)
./run-sql.sh sql/clear.sql DUMMY
```

Stimuli data generator

```
#!/usr/bin/perl

$all_eager = 1;

if($#ARGV != 2) {
    print "Usage: ./generate-data.pl <CHANNEL-COUNT> <DATA-WIDTH> <FILENAME>\n";
    print "<FILENAME> should use %d to place the channel number in the name.\n";
    exit 1;
}

$channel_count=$ARGV[0];
$data_width=$ARGV[1];
$file_name=$ARGV[2];

for ($i=1; $i<=$channel_count; $i++) {
    $file = ">".sprintf($file_name, $i);

    open(OUTFILE, $file) or die "Can't open file:".$file;

    if ($i == 1 | $all_eager) {
$delay = 1;
    } else {
$delay = 1000000+$i;
    }
    $format = sprintf("%010d %%0%dX\n", $delay, $data_width/4);
    $offset = (16**3)*$i;
    for ($j=1; $j<=1000; $j++) {
printf OUTFILE ($format, abs(rand(2**$data_width)));
# printf OUTFILE ($format, $offset+$j);
# printf OUTFILE ($format, 0);
    }

    close OUTFILE;

}
```

Simulation Database Queries

Throughput Query

```
-- Find average througput on the link
SELECT @PARAMETER@, count(*)/(max(recv) - min(sent)) as throughput,
CHANNEL_COUNT, LINK_IMPL, STAGE_COUNT, DATA_WIDTH
```

```
FROM flit
GROUP BY DATA_WIDTH, CHANNEL_COUNT, LINK_IMPL, STAGE_COUNT;
```

Latency Query

```
-- Find average latency on channel 1
SELECT @PARAMETER@, avg(recv-sent) as latency,
CHANNEL_COUNT, LINK_IMPL, STAGE_COUNT, DATA_WIDTH
FROM flit
WHERE channel=1
GROUP BY DATA_WIDTH, CHANNEL_COUNT, LINK_IMPL, STAGE_COUNT;
```

Appendix B

Net-list Macros

This appendix includes some sample net-list macros. Full source-code for the link implementations is included on the CD enclosed with this report. Appendix C presents an overview of the CD-content.

B.1 Some common m4 constructs

```
define(comment, `ifelse(HDL_LANG, vhdl, --, //) $1')dnl
comment(`macros.m4 included')
changecom(`/*', `*/')dnl
define(`DATA_SIZE', `[1:DATA_WIDTH]')dnl
define(`SEL_SIZE', `[1:CHANNEL_COUNT]')dnl
define(BUFFER, BFHS)dnl
define(`for_each_channel', `forloop(`CHANNEL_NUMBER', 1, CHANNEL_COUNT, `$1')')dnl
define(`forloop',
       `pushdef(`$1', `$2')_forloop(`$1', `$2', `$3', `$4')popdef(`$1')')dnl
define(`_forloop',
       `$4`'ifelse($1, `$3', ,
              `define(`$1', incr($1))_forloop(`$1', `$2', `$3', `$4')')')dnl
define(`id', `ifelse($#, 2, `$1`'CHANNEL_NUMBER`'_`'$2', `$1`'CHANNEL_NUMBER')')dnl
define(`CHAR', `translit($1, `1-8', `A-H')')dnl
dnl
dnl N_INPUT_GATE generates a N-input and/or gate from sdt. cells
dnl $1=CELL_NAME, $2=INPUT_COUNT, $3=INPUT_NAME, $4=OUTPUT_NAME,
dnl $5=COMPONENT_PREFIX, $6=MAX_GATE_INPUTS
define(`N_INPUT_GATE', `dnl
ifelse(eval($2>$6), 1, `dnl
forloop(`J', 1, eval($2/$6), `dnl
define(`NNN', $6)dnl
wire w_`'$5`'_`'J;
$1 $5`'_`'J`'(forloop(`K', 1, $6, `.CHAR(K)($3`'eval((J-1)*$6+K)), ').Z(w_`'$5`'_`'J));
')dnl end forloop
N_INPUT_GATE(`$1', eval($2/$6), w_`'$5`'_, $4, $5`'_, $6)dnl
', `dnl else $2>$6
define(`NNN', $2)dnl
ifelse($2, 1, `
assign $4 = $3`'1;
', `dnl else $2==1
$1 $5`'_1(forloop(`K', 1, $2, `.CHAR(K)($3`'eval(K)), ').Z($4));
')dnl end else $2==1
```

```
')dnl end else $2>$6
')dnl end define
dnl
define(`log2', `ifelse($1, 2, 1, `eval(1+log2(eval($1/2)))')')dnl
define(`log4', `ifelse(eval($1<=4), 1, 1, `eval(1+log4(eval($1/4)))')')dnl
dnl
dnl $1=FANOUT, $2=INPUT, $3=OUTPUT, $4=NAME
define(`BUFFER_CHAIN', `ifelse(dnl
eval($1<=4), 1, `dnl
assign $3 = $2;
', eval($1<=16), 1, `dnl
BFHSX4 $4 (.A($2), .Z($3));
', eval($1<=64), 1, `dnl
wire $4`'_W;
BFHSX4 $4`'_1 (.A($2), .Z($4`'_W));
BFHSX16 $4`'_2 (.A($4`'_W), .Z($3));
', `dnl
wire $4`'_W;
BFHSX8 $4`'_1 (.A($2), .Z($4`'_W));
BFHSX32 $4`'_2 (.A($4`'_W), .Z($3));
')')dnl
```

B.2 N-channel Mutex

```
dnl This file should be expanded with m4!!!
define(HDL_LANG, verilog)dnl
include(macros.m4)dnl
ifdef(`OR_GATE', , `define(OR_GATE, OR2HS)')dnl
ifdef(`AND_GATE', , `define(`AND_GATE', AN2HS)')dnl
ifdef(`MUTEX2', , `define(`MUTEX2', `mutex2')')dnl

module mutex (
forloop(`I', 1, CHANNEL_COUNT, `dnl
in`'I,
out`'I,
')dnl
reset
);

forloop(`I', 1, CHANNEL_COUNT, `dnl
input in`'I;
output out`'I;
')dnl
input reset;

define(`mutex_macro', `dnl
ifelse(eval($1==2), 1, `dnl
MUTEX2 mutex_2_1 (.in1($2`'1), .in2($2`'2), .out1($3`'1), .out2($3`'2));
', `dnl else
forloop(`I', 1, eval($1/2), `dnl
wire in_`'eval($1/2)`'_`'I, out_`'eval($1/2)`'_`'I, dnl
int_`'$1`'_`'eval(I*2-1), int_`'$1`'_`'eval(I*2);
OR_GATE or_`'$1`'_`'I (.A($2`'eval(I*2-1)), dnl
.B($2`'eval(I*2)), .Z(in_`'eval($1/2)`'_`'I));
MUTEX2 mutex_`'$1`'_`'I (.in1(int_`'$1`'_`'eval(I*2-1)), dnl
.in2(int_`'$1`'_`'eval(I*2)), .out1($3`'eval(I*2-1)), .out2($3`'eval(I*2)));
')dnl end forloop
forloop(`I', 1, $1, `dnl
AND_GATE and_`'$1`'_`'I (.A($2`'I), dnl
```

```
.B(out_‘’eval($1/2)_‘’eval((I+1)/2)), .Z(int_‘’$1‘’_‘’I));
’)dnl end forloop
mutex_macro(eval($1/2), in_‘’eval($1/2)_, out_‘’eval($1/2)_)dnl
’)dnl end ifelse
’)dnl end define
dnl
mutex_macro(CHANNEL_COUNT, in, out)dnl

endmodule
```

B.3 Impl. 1

```
dnl This file should be expanded with m4!!!
define(HDL_LANG, verilog)dnl
include(macros.m4)dnl

module link (
for_each_channel(‘dnl
id(in, data),
id(in, req),
id(in, ack),
id(out, data),
id(out, req),
id(out, ack),
’)dnl
reset
);

input reset;
wire reset_buf;
BUFFER_CHAIN(CHANNEL_COUNT, reset, reset_buf, _RESET_BUF)


for_each_channel(‘dnl
// Channel id()
input DATA_SIZE id(in, data);
input id(in, req), id(out, req);
output DATA_SIZE id(out, data);
output id(in, ack), id(out, ack);

wire id(req), id(req_reset), id(ack);

AN2AHS id(_AND_RESET) (.A(reset_buf), .B(id(req)), .Z(id(req_reset)));

passivator id(_PAS) (
    .in_req(id(in, req)),
    .in_ack(id(in, ack)),
    .out_req(id(req_reset)),
    .out_ack(id(ack))
);

channel id(_CHANNEL) (
    .in_req(id(req)),
    .in_ack(id(ack)),
    .in_data(id(in, data)),
    .out_req(id(out, req)),
    .out_ack(id(out, ack)),
    .out_data(id(out, data)),
    .reset(reset)
```

```
);

')dnl

endmodule
```

B.4 Impl. 2

```
dnl This file should be expanded with m4!!!
define(HDL_LANG, verilog)dnl
include(macros.m4)dnl

module link (
for_each_channel(`dnl
id(in, data),
id(in, req),
id(in, ack),
id(out, data),
id(out, req),
id(out, ack),
')dnl
reset
);

for_each_channel(`dnl
input DATA_SIZE id(in, data);
input id(in, req), id(out, req);
output DATA_SIZE id(out, data);
output id(in, ack), id(out, ack);
')dnl
input reset;

wire Sreq, Rreq, Sack, Rack;
wire DATA_SIZE Sdata, Rdata;

for_each_channel(`dnl
wire id(Srdy), id(rdy_req), id(Ssel), id(Rsel);
//enable id(enable)`'(.ReqIn(id(in, req)), .Enable(id(Srdy)), .ReqOut(id(rdy_req)));
c2 id(enable)`'(.A(id(in, req)), .B(id(Srdy)), .Z(id(rdy_req)));
')dnl

handshake_arbiter _ARBITER1(
for_each_channel(`dnl
    id(.req_in)`'(id(rdy_req)),
    id(.ack_in)`'(id(in, ack)),
    id(.req_out)`'(id(Ssel)),
')dnl
    .ack_out(Sack),
    .reset(reset)
);

N_INPUT_GATE(`OR`'NNN`'HS', CHANNEL_COUNT, Ssel, Sreq, _O1, 8)dnl

mux _MUX1(
for_each_channel(`dnl
    id(.din)`'(id(in, data)),
    id(.sel)`'(id(Ssel)),
')dnl
    .dout(Sdata)
```

```
);

channel id(_CHANNEL) (
    .in_req(Sack),
    .in_ack(Sreq),
    .in_data(Sdata),
    .out_req(Rack),
    .out_ack(Rreq),
    .out_data(Rdata),
    .reset(reset)
);

for_each_channel(`dnl
wire DATA_SIZE id(out, data);
wire id(Rack);
c2 id(_C) (.A(id(Rsel)), .B(Rreq), .Z(id(Rack)));
\1bitBuffer id(_BUF_RDY) (.in(id(out, req)), .out(id(Srdy)));
\1bitBuffer id(_BUF_SEL) (.in(id(Ssel)), .out(id(Rsel)));
assign id(out, ack) = id(Rack);
assign id(out, data) = Rdata;
')dnl

N_INPUT_GATE(`OR`'NNN`'HS', CHANNEL_COUNT, Rack, Rack, _O2, 8)dnl

endmodule
```

B.5 Impl. 3

```
dnl This file should be expanded with m4!!!
define(HDL_LANG, verilog)dnl
include(macros.m4)dnl

module link (
for_each_channel(`dnl
id(in, data),
id(in, req),
id(in, ack),
id(out, data),
id(out, req),
id(out, ack),
')dnl
reset
);

for_each_channel(`dnl
input DATA_SIZE id(in, data);
input id(in, req), id(out, req);
output DATA_SIZE id(out, data);
output id(in, ack), id(out, ack);
')dnl
input reset;

for_each_channel(`dnl
wire id(Ssync_req), id(Ssync_ack), id(Sdata_req), id(Sdata_ack), dnl
 id(SDdata_req), id(SDdata_ack), id(joined_req), id(joined_ack), dnl
 id(Sdata_req_reset);
wire DATA_SIZE id(SDdata_data);

passivator id(_PASS) (
```

```
    .in_req(id(in, req)),
    .in_ack(id(in, ack)),
    .out_req(id(joined_req)),
    .out_ack(id(joined_ack))
);

fork_pull id(_FORK) (
.in_req(id(joined_req)),
.in_ack(id(joined_ack)),
.out1_req(id(Ssync_req)),
.out1_ack(id(Ssync_ack)),
.out2_req(id(Sdata_req_reset)),
.out2_ack(id(Sdata_ack))
);
AN2AHS id(_AND) (.A(reset), .B(id(Sdata_req)), .Z(id(Sdata_req_reset)));
decouple_latch id(_DECOUPLE_L) (.in_req(id(Sdata_req)), .in_ack(id(Sdata_ack)), dnl
.in_data(id(in, data)), .out_req(id(SDdata_req)), .out_ack(id(SDdata_ack)), dnl
.out_data(id(SDdata_data)), .reset(reset));

wire id(Rsync_req), id(Rsync_ack), id(Rdata_req), id(Rdata_ack);

\1bitBuffer id(_BUF_SYNC_REQ) (.in(id(Rsync_req)), .out(id(Ssync_req)));
\1bitBuffer id(_BUF_SYNC_ACK) (.in(id(Ssync_ack)), .out(id(Rsync_ack)));

join_pull id(_JOIN) (
.in1_req(id(Rsync_req)),
.in1_ack(id(Rsync_ack)),
.in2_req(id(Rdata_req)),
.in2_ack(id(Rdata_ack)),
.out_req(id(out, req)),
.out_ack(id(out, ack))
);

')dnl end for_each_channel

define(`DEPTH', log2(CHANNEL_COUNT))dnl
wire [1:eval(DATA_WIDTH+DEPTH*2)] link_in_data, link_out_data;
wire link_in_req, link_in_req_reset, link_in_ack, link_out_req, link_out_ack;

funnel _FUNNEL (
for_each_channel(`dnl
.id(in_req)`'(id(SDdata_req)),
.id(in_ack)`'(id(SDdata_ack)),
.id(in_data)`'(id(SDdata_data)),
')dnl
.out_req(link_in_req_reset),
.out_ack(link_in_ack),
.out_data(link_in_data),
.reset(reset)
);
AN2AHS _AND_FUNNEL_RESET (.A(reset), .B(link_in_req), .Z(link_in_req_reset));

channel_select _CHANNEL (
.in_req(link_in_req),
.in_ack(link_in_ack),
.in_data(link_in_data),
.out_req(link_out_req),
.out_ack(link_out_ack),
.out_data(link_out_data),
.reset(reset)
);
```

```
horn _HORN (
.in_req(link_out_req),
.in_ack(link_out_ack),
.in_data(link_out_data),
for_each_channel(`dnl
.id(out_req)`'(id(Rdata_req)),
.id(out_ack)`'(id(Rdata_ack)),
.id(out_data)`'(id(out, data)),
')dnl
.reset(reset)
);

endmodule
```

B.6 Funnel

```
dnl This file should be expanded with m4!!!
define(HDL_LANG, verilog)dnl
include(macros.m4)dnl
define(`MAX_DEPTH', log2(CHANNEL_COUNT))dnl

module funnel (
for_each_channel(`dnl
id(in_req),
id(in_ack),
id(in_data),
')dnl
out_req,
out_ack,
out_data,
reset
);

for_each_channel(`dnl
output id(in_req);
input id(in_ack);
input DATA_SIZE id(in_data);
')dnl
input reset, out_req;
output out_ack;
output [1:eval(DATA_WIDTH+2*MAX_DEPTH)] out_data;


dnl $1=INPUT_COUNT, $2=INPUT_PREFIX, $3=OUTPUT_PREFIX, $4=UNIQE_PREFIX, <$5=DEPTH>
define(`FUNNEL_MACRO', `dnl
define(`DEPTH', `ifelse($#, 4, 2, $5)')dnl
define(`OUTPUT_PREFIX', `ifelse(eval($1>2), 1, $4, $3)')dnl

forloop(`J', 1, eval($1/2), `dnl
define(`OUTPUT_INDEX', ifelse($1, 2, `', `J'))dnl
wire [1:eval(DATA_WIDTH+DEPTH)] OUTPUT_PREFIX`'Idata`'OUTPUT_INDEX, dnl
OUTPUT_PREFIX`'data`'OUTPUT_INDEX;
wire OUTPUT_PREFIX`'Ireq`'OUTPUT_INDEX, OUTPUT_PREFIX`'Iack`'OUTPUT_INDEX, dnl
OUTPUT_PREFIX`'req`'OUTPUT_INDEX, OUTPUT_PREFIX`'ack`'OUTPUT_INDEX;
arbiter_pull $4`'ARB`'J (
.in1_req($2`'req`'eval(J*2-1)),
.in1_ack($2`'ack`'eval(J*2-1)),
.in2_req($2`'req`'eval(J*2-0)),
.in2_ack($2`'ack`'eval(J*2-0)),
```

```
.out_req(OUTPUT_PREFIX`'Ireq`'OUTPUT_INDEX),
.out_ack(OUTPUT_PREFIX`'Iack`'OUTPUT_INDEX),
.sel1(OUTPUT_PREFIX`'Idata`'OUTPUT_INDEX[eval(DATA_WIDTH+DEPTH-1)]),
.sel2(OUTPUT_PREFIX`'Idata`'OUTPUT_INDEX[eval(DATA_WIDTH+DEPTH-0)]),
.reset(reset)
);

mux`'eval(DATA_WIDTH+DEPTH-2) $4`'MUX`'J (
.din1($2`'data`'eval(J*2-1)),
.din2($2`'data`'eval(J*2-0)),
.sel1(OUTPUT_PREFIX`'Idata`'OUTPUT_INDEX[eval(DATA_WIDTH+DEPTH-1)]),
.sel2(OUTPUT_PREFIX`'Idata`'OUTPUT_INDEX[eval(DATA_WIDTH+DEPTH-0)]),
.dout(OUTPUT_PREFIX`'Idata`'OUTPUT_INDEX[1:eval(DATA_WIDTH+DEPTH-2)])
);

latch`'eval(DATA_WIDTH+DEPTH) $4`'LATCH`'J (
.in_req(OUTPUT_PREFIX`'Ireq`'OUTPUT_INDEX),
.in_ack(OUTPUT_PREFIX`'Iack`'OUTPUT_INDEX),
.in_data(OUTPUT_PREFIX`'Idata`'OUTPUT_INDEX),
.out_req(OUTPUT_PREFIX`'req`'OUTPUT_INDEX),
.out_ack(OUTPUT_PREFIX`'ack`'OUTPUT_INDEX),
.out_data(OUTPUT_PREFIX`'data`'OUTPUT_INDEX),
.reset(reset)
);

')dnl end forloop
ifelse(eval($1>2), 1, `FUNNEL_MACRO(eval($1/2), $4, $3, $4`'_, eval(DEPTH+2))')dnl

')dnl end define

FUNNEL_MACRO(CHANNEL_COUNT, in_, out_, _U)dnl

endmodule
```

Appendix C

CD Content

The enclosed CD contains all source files and design-flow scripts needed to instantiate the link implementations presented in this thesis. Here is an overview of the CD content:

/ Design flow scripts and source files common for the link implementations.

/nosyn/ Behavioral link modules.

/channel/ Modules implementing the physical channel(shared by all implementations).

/link1/ Modules specific for imp. 1.

/link2/ Modules specific for imp. 2.

/link3/ Modules specific for imp. 3.

/java/ Java program for parsing log-files.

/lib/ Library files for the Java program.

/sql/ SQL scripts for statistical queries

Full file-list:

```
./channel/1bitBuffer.v.in
./channel/Nbit1of4_select_latch.v.in
./channel/Nbit1of4latch.v.in
./channel/NbitBuffer.v.in
./channel/channel.v.in
./channel/channel_select.v.in
./channel/wires.v.in
./.synopsys_dc.setup
./.synopsys_vss.setup
./1of4dec.v.in
./1of4enc.v.in
```

```
./1of4latch.v
./Makefile
./SRLATCH.v
./arbiter_pull.v
./branch_pull.v
./c.v
./c2.stg
./c2.v
./compile.dcsh
./configure
./d_latch.v.in
./debug.do
./fix-petrify-bugs.sed
./fork_pull.v
./generate-data.pl
./handshake_arbiter.v.in
./join_pull.v
./macros.m4
./modelsim.tcl
./mutex.v.in
./mutex2.v
./od_pull_lctl.stg
./passivator.v
./report-area.dcsh
./report-power.dcsh
./run-batch.sh
./run-sql.sh
./simulate.do
./sink.vhd
./source.vhd
./tb.vhd.in
./testbench.vhd.in
./java/noc/analysis/SimulationAnalysis.java
./lib/mysql-connector-java-3.0.11-stable-bin.jar
./link1/link.v.in
./link2/demux.v.in
./link2/link.v.in
./link2/mux.v.in
./link3/decouple_latch.v.in
./link3/demux.v.in
./link3/dr_latch.v.in
./link3/funnel.v.in
./link3/horn.v.in
./link3/latch.v.in
./link3/link.v.in
./link3/mux.v.in
./nosyn/mutex2.v.in
./nosyn/mutex_check.v.in
./sql/average-latency-channel-1.sql
./sql/clear.sql
./sql/create.sql
./sql/throughput-link.sql
```

Bibliography

[1] Kazuhiro Aoyama and Andrew A. Chien. The cost of adaptivity and virtual lanes in a wormhole router. *VLSI Design*, 2(4):315–333, 1995.

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

[3] W.J. Bainbridge and S.B. Furber. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 118–126, March 2001.

[4] W.J. Bainbridge, W.B. Toms, D.A. Edwards, and S.B. Furber. Delay-insensitive, point-to-point interconnect using m-of-n codes. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 132–140, May 2003.

[5] Luca Benini and Giovanni De Micheli. Network on chips: A new soc paradigm. *IEEE Comput. Soc: Computer*, 35(1):70–78, 2002.

[6] Tobias Bjerregaard. How many is enough. October 2003. Internal IMM white-paper(DRAFT).

[7] W.S. Coates, J.K. Lexau, I.W. Jones, S.M. Fairbanks, and I.E. Sutherland. Fleetzero: an asynchronous switching experiment. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 173–182, March 2001.

[8] Jordi Cortadella. Homepage for the public domain tool petrify. `http://www.lsi.upc.es/~jordic/petrify/`.

[9] Jordi Cortadella. Petrify: a tutorial for the designer of asynchronous circuits. Included in the Petrify distribution. `http://www.lsi.upc.es/~jordic/petrify/distrib/`.

[10] Jordi 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, 1997.

[11] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture*, chapter 10. Morgan Kaufmann, 1999.

[12] William J. Dally. Virtual-channel flow control. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):194–205, 1992.

[13] William J. Dally and Hiromichi Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *Parallel and Distributed Systems, IEEE Transactions on*, 4(4):466–475, 1993.

[14] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, 1987.

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

[16] Sune Frankild and Hans Palbøl. Visual stg lab website, 2000. `http://vstgl.sourceforge.net/`.

[17] Steve B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2):247–253, 1996.

[18] ITRS. International technology roadmap for semiconductors - 2003 edition, 2003. `http://public.itrs.net/`.

[19] A. Jalabert, S. Murali, L. Benini, and G. De Micheli. xpipescompiler: a tool for instantiating application specific networks on chip. *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, 2:884–889, 2004.

[20] Alain J. Martin. Programming in vlsi: From communicating processes to delay-insensitive circuits. Technical report, California Institute of Technology, 1989. `http://resolver.caltech.edu/CaltechCSTR:1989.cs-tr-89-01`.

[21] Open Core Protocol International Partnership. Ocp website and specification, 2004. `http://www.ocpip.org/`.

[22] Ad M.G. Peeters. *Single-rail Handshake Circuits*. PhD thesis, Technische Universiteit Eindhoven, 1996. `http://alexandria.tue.nl/extra3/proefschrift/PRF12B/9602110.pdf`.

[23] A. Pereira, A. Borges, and A. Ferrari. Exclusion relation of k out of n and the synthesis of speed-independent circuits. In *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, pages 155–159, September 2003.

[24] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits - A Design Perspective*. Prentice Hall, 2003.

[25] A. Radulescu, J. Dielissen, K. Goossens, E. Rijpkema, and P. Wielage. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network con .guration. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, 2004.

[26] Swaminathan Ramany and Derek Eager. The interaction between virtual channel flow control and adaptive routing in wormhole networks. In *Conference Proceedings. 1994 International Conference on Supercomputing*, pages 136–145, 1994.

[27] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. *Computers and Digital Techniques, IEE Proceedings-*, 150(5):294–302, 2003.

[28] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design*. Number 3 in European Low-Power Initiative for Electronic System Design. Kluwer Academic Publishers, 2001.

[29] STMicroelectronics. Corelib8dhs hcmos8d 1.8 tec 3.1.a release notes, 2001.

[30] STMicroelectronics. Hcmos8d 0.18um standard cells family, 2002. `http://www.st.com/stonline/books/pdf/docs/8642.pdf`.

[31] Ivan Sutherland and Scott Fairbanks. Gasp: a minimal fifo control. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 46–53, March 2001.

[32] Ivan E. Sutherland and Jon K. Lexau. Designing fast asynchronous circuits. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 184–193, March 2001.

[33] Dennis Sylvester and Kurt Keutzer. A global wiring paradigm for deep submicron design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pages 242–252, 2000.

[34] Daniel Wiklund and Dake Liu. Switched interconnect for system-on-a-chip designs. In *Proc of the IP2000 Europe conference*, October 2000.

[35] Daniel Wiklund and Dake Liu. Design of a system-on-chip switched network and its design support. In *Communications, Circuits and Systems and West Sino Expositions, IEEE 2002 International Conference on*, pages 1279–1283, June 2002.

Glossary and Abbreviations

ASIC Application Specific Integrated Circuit

CPU Central Processing Unit

CSC Complete State Coding

DC Design Compiler®

demux De-multiplexer

DI Delay Insensitive

DSM Deep Sub-micrometer

DSP Digital Signal Processor

FIFO First In - First Out buffer

HDL Hardware Description Language

IP Intellectual Property

mutex Mutual exclusion element

mux Multiplexer

NI Network Interface

NoC Network-on-Chip

RTZ Return To Zero

SAIF Switching Activity Interchange Format

SoC System-on-Chip

STG Signal Transition Graph

VCD Value Change Dump