Designing Efficient Network Interfaces For System Area Networks

Inauguraldissertation

zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

der Universität Mannheim

vorgelegt von

Dipl.-Inf. Lars Rzymianowicz

aus Rendsburg

Mannheim, 2002

Dekan:	Professor Dr. Herbert Popp, Universität Mannheim
Referent:	Professor Dr. Ulrich Brüning, Universität Mannheim
Korreferent:	Professor Dr. Volker Lindenstruth, Universität Heidelberg

Tag der mündlichen Prüfung: 28. August 2002

Designing Efficient Network Interfaces For System Area Networks

Inauguraldissertation

zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

der Universität Mannheim

vorgelegt von

Dipl.-Inf. Lars Rzymianowicz

aus Rendsburg

Mannheim, 2002

Abstract

Designing Efficient Network Interfaces For System Area Networks

by
Lars Rzymianowicz
Universität Mannheim

The network is the key component of a Cluster of Workstations/PCs. Its performance, measured in terms of bandwidth and latency, has a great impact on the overall system performance. It quickly became clear that traditional WAN/LAN technology is not too well suited for interconnecting powerful nodes into a cluster. Their poor performance too often slows down communication-intensive applications. This observation led to the birth of a new class of networks called System Area Networks (SAN).

But still SANs like Myrinet, SCI or ServerNet do not deliver an appropriate level of performance. Some are hampered by the fact, that they were originally targeted at another field of application. E.g. SCI was intended to serve as a cache-coherent interconnect for fine-grain communication between tightly coupled nodes. Its architecture is optimized for this area and behaves less optimal for bandwidth-hungry applications. Earlier versions of Myrinet suffered from slow versions of their proprietary LANai network processor and slow on-board SRAM. And even though a standard I/O bus with a physical bandwidth of more then 500 Mbyte/s (PCI 64 bit/66 MHz) has been available for years, typical SANs only offer between 100-200 Mbyte/s.

All the disadvantages of current implementations lead to the idea to develop a new SAN capable of delivering the performance needed by todays clusters and to keep up with the fast progress in CPU and memory performance. It should completely remove the network as communication bottleneck and support efficient methods for host-network interaction. Furthermore, it should be ideally suited for use in small-scale (2-8 CPUs) SMP nodes, which are used more and more as cluster nodes. And last but not least, it should be a cost-efficient implementation.

All these requirements guided the specification of the ATOLL network. On a single chip, not one but four network interfaces (NI) have been implemented, together with an on-chip 4x4 full-duplex switch and four link interfaces. This unique “Network on a Chip”

architecture is best suited for interconnecting SMP nodes, where multiple CPUs are given an exclusive NI and do not have to share a single interface. It also removes the need for any additional switching hardware, since the four byte-wide full-duplex links can be connected by cables with neighbor nodes in an arbitrary network topology.

Despite its complexity and size, the whole network interface card (NIC) only consists of a single chip and 4 cable connectors, a very cost-efficient architecture. Each link provides 250 Mbyte/s in one direction, offering a total bisection bandwidth of 2 Gbyte/s at the network side. The next generation of I/O bus technology, a 64 bit/133 MHz PCI-X bus interface, has been integrated to make use of this high bandwidth. A novel combination of different data transfer methods has been implemented. Each of the four NIs offers transfer via Direct Memory Access (DMA) or Programmed I/O (PIO). The PIO mode eliminates any need for an intermediate copy of message data and is ideally suited for fine-grain communication, whereas the DMA mode is best suited for larger message sizes. In addition, new techniques for event notification and error correction have been included in the ATOLL NI. Intensive simulations show that the ATOLL architecture can deliver the performance expected. For the first time in Cluster Computing, the network is no more the communication bottleneck.

Specifying such a complex design is one task, implementing it in an Application Specific Integrated Circuit (ASIC) is an even greater challenge. From implementing the specification in a Register/Transfer-Level (RTL) module to the final VLSI layout generation it took almost three years. Implemented in a state-of-the-art IC technology with the CMOS-Digital 0.18 um process of UMC, Taiwan, the ATOLL chip is one of the fastest and most complex ASICs ever designed outside the commercial IC industry. With a die size of 5.8x5.8 sqmm, 43 on-chip SRAM blocks with 100 kbit total, 6 asynchronous clock domains (133-250 MHz), one large PCI-X IP cell and full-custom LVDS and PCI-X I/O cells, a carefully planned design flow had to be followed. Only the design of the full-custom I/Os and the Place & Route of the layout were done by external partners. All the rest of the design flow, from RTL coding to simulation, from synthesis to design for test, was done by ourselves. Finally, the completed layout was given to sample production in February 2002, first engineering samples are expected to be delivered 10 weeks later.

The ATOLL ASIC is one of the most complex and fastest chips ever implemented by a European university. Recently, the design has won the third place in the design contest organized at the Design, Automation & Test in Europe (DATE) conference, the premier European event for electronic design.

Table of Contents

List of Figures

List of Tables

1 Introduction

While in Desktop Computing the latest improvements in performance of computer hardware seem to have outrun the demand by typical software, High Performance Computing (HPC) continues to be one of the main reasons for accelerating hardware like microprocessors or networks. The need to solve large problems like weather forecast or earthquake simulation drives the development of faster CPUs, while vice versa faster hardware enables scientists to attack even larger problems. This chapter introduces Cluster Computing as a new alternative to accelerate High Performance Computing. It also discusses the emergence of a new class of networks called System Area Networks. Finally, a short introduction is given into the field of ASIC design and its most important problems.

1.1 Cluster Computing

Cluster Computing[1][1], [2] has established itself as a serious alternative to Massive Parallel Processing (MPP) and Vector Computing in the field of High Performance Computing. The initial idea was developed back in the 1960's when IBM linked several of their mainframes together to provide a platform capable of dealing with large commercial workloads. However, MPP and vector machines from companies like Cray, SGI, IBM, Intel, NEC, Hitachi, etc. dominated the HPC world throughout the 70's and 80's. With the emergence of the personal computer (PC) and its fast progress in terms of performance, mainly driven by Intel's x86 microprocessors, it became a viable option to use interconnected standard PCs as platform for running HPC applications. Several factors accelerated this trend:

- increasing performance of desktop CPUs from Intel/AMD, closing the gap to high end RISC microprocessors (Alpha, MIPS, PowerPC, SPARC)
- high performance networks interfacing to standard PC I/O technology like PCI
- low costs of mass-fabricated PC components, compared to classic MPP or Vector machines, which are build in quantities of a few hundreds or thousands

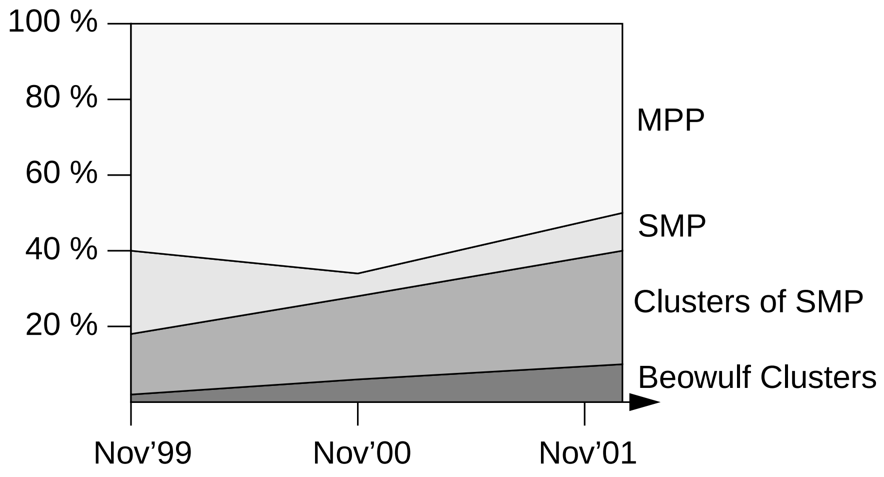
1. a Cluster is a collection of interconnected computers working together as a single system

- standard HPC libraries like MPI [3] or PVM [4] are freely available in different implementations across a wide variety of different platforms
- a stable, high performance Unix-style operating system is freely available with Linux

1.1.1 Trends

The first so-called Beowulf cluster [5] was assembled by the team around Donald Becker and Thomas Sterling at NASA's Goddard Space Flight Center in 1994. It consisted of 16 PCs equipped with Intel 486-DX4 100 MHz CPUs and 16 Mbyte RAM, connected via 10 Mbit Ethernet. This way of building a low-cost, yet powerful supercomputer was adopted by many research groups throughout the world. Today several thousands of clusters are in operation, the largest installations with more than 1.000 nodes. One of the largest clusters ever built, the ASCI Red system from Intel with more than 9.000 Pentium Pro machines at the Sandia National Labs, USA, was No.1 on the Top500 supercomputer list [6][1] from 1997 to 2001.

Figure 1-1. Number of machine types in the Top500 Supercomputer list [6]



Besides PC clusters, several companies build clusters out of small- to medium-scale SMP machines. E.g., IBM uses its RS/6000 SP nodes with up to 16 CPUs per SMP, whereas Compaq builds its SC Series supercomputers by clustering AlphaServer GS machines with up to 32 CPUs. These machines are also often referred to as cluster of SMPs or constellations. Figure 1-1 shows the increasing usage of clusters, according to the Top500 lists of the last three years.

The current trend is to move away from traditional supercomputers to more cost-efficient cluster systems consisting of Commodity-Off-The-Shelf (COTS) components. Another

1. "TOP500 Supercomputer Sites", www.top500.org

main advantage is the better scalability of clusters. Users can start with a small system and add nodes from time to time to match an increasing need for performance. A big indicator for this trend is the fact, that all recent Teraflop systems of the national Accelerated Strategic Computing Initiative (ASCI) program in the USA are clusters of SMPs. These systems are normally assembled in multiple steps, starting with a small installation followed by several upgrades. Only one of the top ten systems of the latest Top500 list is a traditional supercomputer, a Hitachi Vector machine, all other entries are cluster of SMPs.

1.1.2 Managing large installations

But since the number of nodes inside a typical cluster grows fast, it becomes more complicated to make efficient use of the system. A lot of effort has been put into the implementation of resource management software. These tools help to install and configure the operating system and parallel libraries across hundreds of nodes with perhaps different components and equipment. Another major task is the scheduling of parallel jobs and the allocation of processes to idling nodes. And with an increasing chance of failure of single nodes inside a cluster with 1.000 nodes or more, terms like availability, checkpointing, fault detection and isolation become more important. So the focus in Cluster Computing is shifting from developing fast hardware more towards implementing software to manage and easily use installations with 100 or more nodes. One of the main goals of software development for clusters is the idea to present the cluster as a so-called Single System Image (SSI) to the user. The underlying architecture is hidden from the user, who sees the cluster as a single, large parallel computer.

1.1.3 Driving factors and future directions

Figure 1-2. Fields of development for Cluster Computing

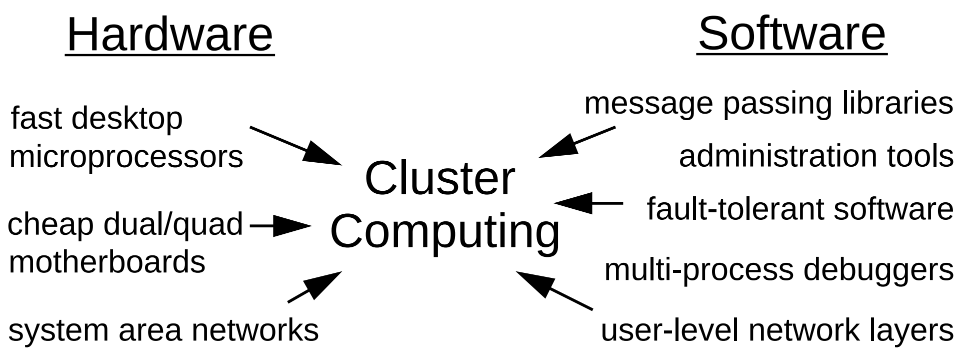


Figure 1-2 depicts all fields of development that contribute to the increasing use of clusters for High Performance Computing. Recent research activities extend the idea of Cluster Computing to an even further decoupled architecture called Grid. Grid Computing [7] connects several computing resources (clusters, single SMP/MPP/Vector/PC machines) in different locations to one single computing system. To overcome the heterogeneity of all components (different platforms, operating systems, networks, etc.) one defines a common protocol to exchange data between all participating nodes inside the Grid. First implementations are available, but a wide adoption of Grid Computing is yet to come.

1.2 System Area Networks

A fast network is the key component of a high performance cluster. First installations used traditional Local Area Network (LAN) technology like 10/100 Mbit Ethernet as interconnect between nodes inside the cluster. But it became quickly clear that these networks are a substantial performance bottleneck. Traditional MPP supercomputers like the Cray T3E or the SGI Origin rely on dedicated and proprietary high performance networks with node-to-node bandwidths of 300 Mbyte/s and more. With typical system bus bandwidths of more than 1 Gbyte/s inside a node, these interconnects can handle the communication demand of even highly fine-grain parallel applications.

1.2.1 The need for a new class of networks

To be competitive in the field of High Performance Computing clusters need to be equipped with networks matching the performance of these proprietary solutions. First experiences were made with existing solutions, either Wide Area Networks (WAN) or LAN. Networks like ATM, HiPPI or SCI offer more physical bandwidth than 100 Mbit Fast Ethernet, but were designed with different applications in mind.

Table 1-1. Bandwidth gap of clusters vs. MPPs[a]

machine	system bus	internode network	system/network ratio
Cray T3E-1350 with Alpha 21164 675 MHz	1.2 Gbyte/s	650 Mbyte/s	1.85
SGI Origin 3800 with MIPS 14k 500 MHz	3.2 Gbyte/s	1.6 Gbyte/s	2
PC cluster with Pentium III 1 GHz and Fast Ethernet	1 Gbyte/s	12 Mbyte/s	**85**

a. typical system configurations in the year 2000

E.g., ATM is tuned for wide area connections with its relatively small packet size of 53 bytes and its support for Quality of Service (QoS). And with 155/622 Mbit/s physical bandwidth it offers more than Fast Ethernet, but is still way behind multi-gigabit networks. On the other hand, a network like HiPPI supports up to 1.6 Gbit/s, but is so expensive that it clashes with the low-cost idea of Beowulf Computing. Table 1-1 gives an impression of the gap between system and network bandwidth inside different parallel architectures.

With almost two orders of magnitude between system bus and network bandwidth clusters with standard LAN technology are no match for traditional supercomputers. This led quickly to several projects, both at universities and commercial companies. The goal was to develop a low latency, high bandwidth network with a range of a few meters (up to 10). Two components had to be constructed:

- a Network Interface Card (NIC), which provides a link interface via cable into the network and uses a standard interface to connect to the host system (for PCs that is the PCI bus)
- a multi-port switch, which is used to connect single nodes into a cluster. The number of ports typically lies in the range of 6 to 32.

1.2.2 Emerging from existing technology

This new class of networks was named System Area Networks (SAN) to point out their different application in contrast to existing LAN/WANs. Most developments adopted existing technologies from the world of classical parallel computers. E.g., the first version of Myrinet, one of the most successful SANs today, was originally developed for a fine-grain supercomputer called Mosaic [8] by research groups at the California Institute of Technology (Caltech) and the University of Southern California (USC). Or the company Quadrics, offering now the QsNet SAN, emerged from the well-known supercomputer manufacturer Meiko Ltd., which built cache-only supercomputers like the CS-2 [9]. The main component of QsNet, the ELAN III ASIC, is the third generation of the ELAN communication processor introduced in the CS-2.

At the end of the 90's, several SANs were introduced and widely used in clusters. Networks like Myrinet, ServerNet, QsNet and SCI will be discussed in Chapter 2 in detail. With their bandwidth in the range of 100-400 Mbyte/s and a one-way latency of around 10 us they facilitated clusters to compete with traditional supercomputers.

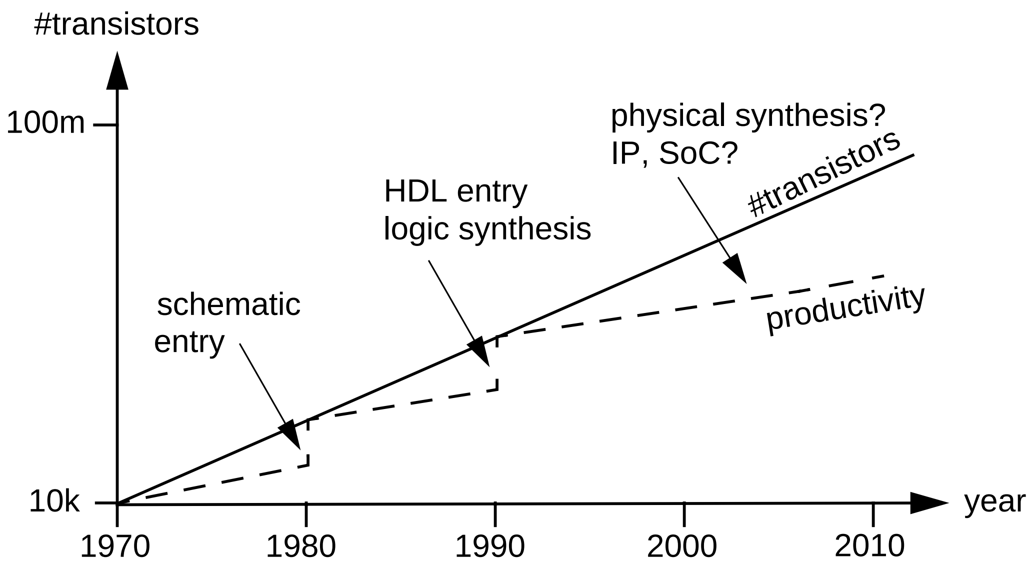
1.3 ASIC Design

The development of logic circuits as Application Specific Integrated Circuits (ASIC) continues at a rate predicted by Gordon E. Moore back in 1965. This famous Moore's Law [10] predicts that the number of transistors per IC doubles every 18 months. It has been valid throughout the last 30 years and seems to continue to be true for the near future. It is made possible by constant advancements in semiconductor technology and silicon fabrication.

1.3.1 Using 10+ million transistors

ASIC designers face more and more the problem to be able to make use of all the potential transistors on a silicon die. This is known as the productivity gap. Every few years, the Electronic Design Automation (EDA) industry needs a big step forward in methodology to keep pace with the steep technology curve. Figure 1-3 depicts this situation and shows some of the improvements of past years and decades.

Figure 1-3. The productivity gap in IC development



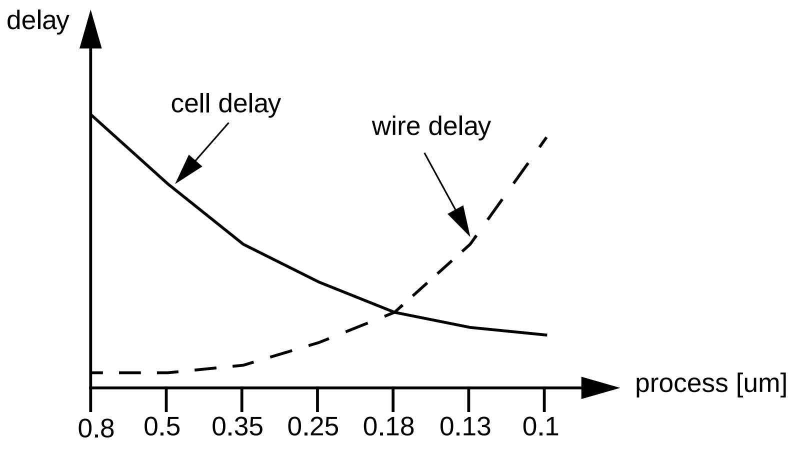
Designers steadily increase the level of abstraction for modeling logic circuits to enhance their productivity. They moved from full-custom VLSI layout to schematic entry and on to Hardware Description Languages (HDL) like Verilog [11] and VHDL [12], which are tightly coupled with logic synthesis. The next big step in modeling abstraction would be moving to behavioral or architectural descriptions of logic circuits. First steps have been made into this direction, but it is not yet clear, if the languages used are based on C/C++ like SystemC [13], or if it is an extension to an existing HDL like Superlog [14].

While ASICs approach the 100 million transistor count and clock frequencies of multiple GHz, designers face a handful of severe problems.

1.3.2 Timing closure

The IC design flow used over the last years is split into two separate steps, called frontend and backend. The frontend flow uses logic synthesis to turn a design specified in an HDL into a so-called netlist of logic cells. Optimization goals like area or timing guide the synthesis process into the right direction. To calculate the timing delay of logic paths tools rely on quite precise cell delays and estimated wire delays. Estimation is necessary, since the synthesis process only defines the interconnection of logic cells, not their location on the chip. The estimations were not a problem when cell delays dominated wire delays, like in older process technologies (0.35-1 um). But with shrinking structures this proportion gets inversed. This trend is shown in Figure 1-4. Wire delays are going to dominate cell delays in process technologies beyond 0.18 um. As consequence, the estimations of synthesis tools get more and more imprecise. This leads to huge differences in timing before and after layout.

Figure 1-4. Cell vs. wire delay[1]



These timing mispredictions force the designer to iterate several times between frontend and backend design to reach his timing goal. These iterations could last several weeks or even months, which is unacceptable, since time-to-market is a major factor for success. EDA companies address this problem by incorporating physical design into the synthesis process. This is known as physical synthesis, or when frontend and backend design are fully integrated, called a RTL-to-GDSII flow.

1.3.3 Power dissipation

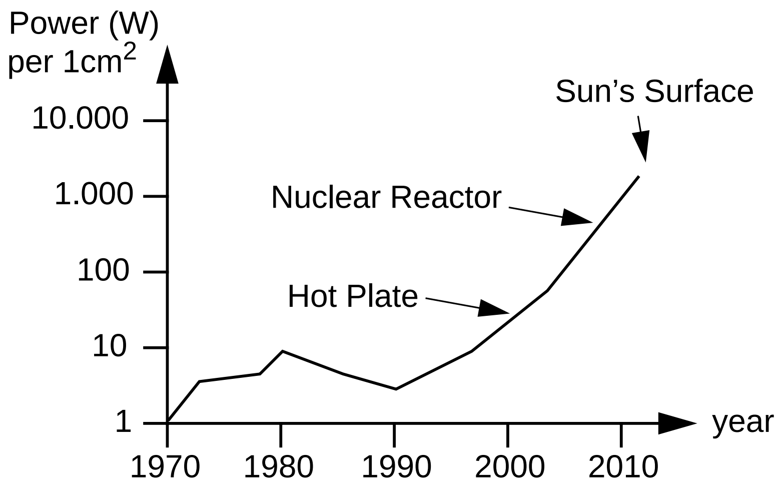
With frequencies beyond 1 GHz and more than 10 million transistors on chip, current microprocessors dissipate between 40-70 W. Extensive cooling is needed to prevent the CPU from overheating and being damaged by effects like electromigration [15]. The speed of an ASIC is also slowed down by rising temperatures. Recent projections [16]

1. the figure visualizes only the trend, actual numbers may vary from vendor to vendor

show that power is becoming quickly the main hurdle for future generations of chips, as depicted in Figure 1-5.

Several techniques are used to reduce the power consumption of ICs. The problem is being attacked in both domains, design methodology as well as process technology. Semiconductor manufacturers develop new technologies with reduced supply voltages to keep power consumption at an acceptable level. New fabrication techniques like Silicon-On-Insulator (SOI) reduce the amount of leakage current.

Figure 1-5. Power dissipation of ICs in the next decade[1] [16]



IC designers attack the problem at several abstraction levels. For very high frequencies of 1 GHz and more it has been found out that about 50-70 % of total power is consumed by the clock tree of a chip. This identifies the clock tree as an ideal point of power optimization. The trend of building System-on-a-Chip (SoC) designs with lots of components on a single die also lowers the utilization factor of on-chip components. Hardly all components are active at the same time, some may idle, waiting for input data, etc. So one can disable certain functional units for the time they are not necessary. This is done by suppressing the clock signals for the whole unit, a technique called clock gating. E.g., a microprocessor could disable its floating point unit as long as no floating point instructions enter the instruction buffer. This could save a significant amount of power while running integer-dominated applications. Another method is to adjust the main frequency to the current demand for processing power. This technique is used heavily for mobile computers like laptops or PDAs.

1. if current trends continue without major improvements in power reduction

1.3.4 Verification bottleneck

Another major problem is to validate the design before shipping the layout to the chip manufacturer. With increasing design complexity the verification space, the number of different input and state combinations, becomes almost unmanageable. More and more effort has to be put into the functional test of a design, both in terms of testbench complexity and processing power to run them. E.g., the team that developed the newest UltraSPARC III microprocessor from Sun [17] used a server farm with 3.000 CPUs in total to run the huge amount of testbenches in an acceptable time frame. A single verification run can easily consume several Gbyte of memory while running for hours or days.

Verification is needed at all levels of the design flow. From high-level simulations of abstract functional implementations down to transistor-level simulations of the final layout, after each stage one has to verify that the design still meets all goals defined in the specification. Catching bugs as early as possible has become a significant factor in meeting the time-to-market goals of an IC project.

1.4 Contributions

This dissertation introduces a major redesign of the ATOLL architecture for a high performance System Area Network. It combines several unique features not found in current solutions, like the support for multiple network interfaces and the inclusion of an on-chip switch component. Data transfer between the host system and the network is optimized by a combination of PIO- and DMA-based mechanisms. A novel event notification technique greatly enhances the capabilities of the NI.

Besides discussing the architecture, it also describes the implementation of the design in a state-of-the-art semiconductor technology. Putting all the described functionality into a single chip is an extremely difficult task and has never been done before. A carefully planned design flow has been established to manage this large project with limited resources and manpower.

Extensive simulations were done to prove the functional correctness of the design and to make sure all performance goals are met. At the end, ideas for the next generation of ATOLL are discussed.

Though the author is responsible for the largest part of design and implementation work regarding the ATOLL chip, several colleagues of the Chair of Computer Architecture have helped by designing some significant parts of the chip. Leaving out those parts in this thesis would prevent the reader from getting a deep understanding of the whole architecture. So instead, those parts contributed by others are therefore discussed here and marked by footnotes. References to additional literature have been added, if possible.

1.5 Organization

The dissertation is organized in six chapters. This first chapter introduced Cluster Computing and System Area Networks in general. It presented current trends and gave also an insight into the problems in modern IC design. The following chapter then discusses current SANs more in detail. After listing the main design concepts, a broad overview is given about the architectural features of current networks. Chapter 3 presents the motivation for a novel SAN architecture call ATOLL. The rest of the chapter then introduces the ATOLL architecture. The main ideas behind ATOLL are presented, as well as their implementation. Chapter 4 follows with a broad overview about the development of the ATOLL ASIC. The main design steps are presented, together with the most important results. This is followed by a performance evaluation in Chapter 5. Finally, Chapter 6 summarizes the results, draws conclusions and discusses areas of future work.

2 System Area Networks

The network is the most critical component of a cluster. Its capabilities and performance directly influence the applicability of the whole system for HPC applications. After describing some traditional network technology, the most important general design issues for high performance networks are discussed. This is followed by a survey of existing SAN solutions. Their architecture and main properties are described and evaluated in the order in which the networks have evolved over the years.

2.1 Wide/Local Area Networks

According to recent cluster rankings[1], about half of all clusters are still equipped with standard 100 Mbit/s Fast Ethernet network technology. This fact has mainly two reasons: costs and application behavior. While several SANs are available today, they cannot really compete with the mass-market prices of Fast Ethernet, even regarding their price vs. performance ratio. On the other hand, lots of applications have been finetuned to the limited performance of LANs in the early days of Cluster Computing. When only Ethernet was available, programmers had no choice than to avoid communication where possible and to use more coarse grained communication patterns in their applications. So a large set of programs are tailored towards the high latency and low bandwidth of Ethernet. Running these applications then on a cluster equipped with a high performance SAN does not use the full potential of those interconnects. Significant modifications to the program code would be needed, but are rarely done.

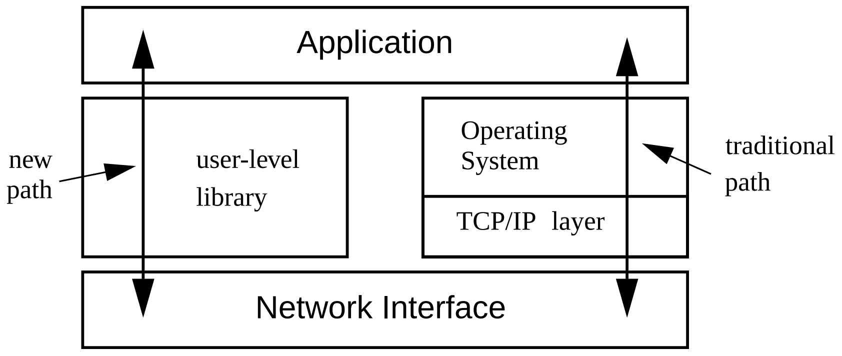
2.1.1 User-level message layers

First clusters running MPI/PVM applications used a normal TCP/IP layer to communicate over Fast Ethernet. But the TCP/IP protocol stack inside an operating system (OS) kernel is quite large, resulting in excessive latencies in the order of 50-70 us for sending a short message between two nodes. This high latency clashes with the goal of competing with supercomputers, which normally offer one-way latencies below 10 us. Since about 90 % of this latency can be attributed to the software, researchers started to implement so-called user-level message layers [18], which bypass the OS for inter-node communication. By

1. “Clusters @ Top500”, clusters.top500.org

removing the OS from the communication path, the sending/receiving of a message can be speed up significantly, as shown in Figure 2-1.

Figure 2-1. User-level vs. OS-based TCP/IP communication



Implementations like U-Net [19], GAMMA [20] and Fast Messages [21] all provide a low-level Application Programming Interface (API) to the network. Only some initialization routines interact with the OS. All the functions to send/receive messages between different nodes of a cluster directly access the network interface. Implementations mainly differ in their levels of security and reliability. The fastest implementations simply ignore any security issues (memory protection, multitasking an NI) because of the fact that most production clusters run a single parallel job with a one-to-one mapping of processes to CPUs for highest application performance.

Table 2-1. Comparison of user-level libraries for Fast Ethernet[a]

User-level library	System configuration	latency (us)	bandwidth (Mbyte/s)
U-Net	DEC 21140 chipset, Intel Pentium 133 MHz	30.0	12.1
GAMMA	DEC 21143 chipset, AMD K7 500 MHz	14.3	12.1
TCP/IP	DEC 21143 chipset, Intel Pentium II 350 MHz	58	10.5

a. taken from the GAMMA website: www.disi.unige.it/project/gamma

Table 2-1 shows that user-level libraries can reduce latency by 50-75 %, compared to TCP/IP performance. But the low physical bandwidth of Ethernet remains a critical bottleneck.

A few other LAN/WANs have been tested as cluster interconnect, but proved to be as inefficient as Fast Ethernet. As mentioned earlier, ATM provides more physical bandwidth, but its protocol is more oriented towards Quality-of-Service (QoS) applications like streaming audio/video media. So overall, most LAN/WAN technology is inappropriate as cluster interconnect. Only Fast Ethernet, and recently also its upgrade Gigabit Ethernet, can be used in combination with user-level message layers, if the applications are mostly sensitive to latency and not to bandwidth.

2.2 Design goals

Before several cluster interconnects are presented in detail, this section gives an overview of the main design trade-offs for interconnect hardware. For each design topic, several possibilities are presented and evaluated. With this basic knowledge in mind, the reader should be able to rate concrete implementations according to their usability and performance for specific applications.

Several decisions must be made when designing a cluster interconnect. The most important is undeniably the price/performance trade-off.

2.2.1 Price versus performance

In the last few years clusters of PCs have gained huge popularity due to the extreme low prices of standard PCs. Traditional supercomputer technology is replaced more and more by tightly interconnected PCs. In the interconnect market, though, a huge gap exists between interconnects of moderate bandwidth like Fast Ethernet at a low price ($ 50-100 for a network adapter) and high performance networks like Myrinet or ServerNet ($ 1000 and more). Of course, this is also a consequence of low production volumes. But other factors, such as onboard RAM or expensive physical layers such as Fiber Channel, can raise costs significantly.

2.2.2 Scalability

Table 2-2. Supercomputers and their different network topology

Machine	Topology
Cray T3E	3D torus
IBM SP2	omega (multistage)
SGI Origin 2000	hierarchical fat hypercube
nCube	hypercube
Thinking Machines CM-5	fat tree

Scalability is another crucial issue. It refers to the networks ability to scale almost linear with the number of nodes. A good topology is the key factor for good scalability. Interconnects in traditional supercomputers normally have a fixed network topology (mesh, hypercube, etc.) and hardware/software relies on the fixed topology. Table 2-2 gives an overview about the variety of network topologies used in recent supercomputers.

But clusters are more dynamic. Often a small system is set up to test, if the cluster fits to the application needs. With increasing demand for computing power, more and more nodes are added to the system. The network should tolerate the increased load and deliver nearly the same bandwidth and latency to small clusters (8-32 nodes) and to large ones (hundreds of nodes). A large mesh will show increased latency compared to a small one, since the average distance between nodes also increases. Large switches (16x16, 24x24) forming a cluster-of-clusters topology can help to compensate this effect [22]. Similarly, a hypercube network cannot be upgraded from 64 to 96 nodes because it needs a power of two as node count. Therefore, modern cluster interconnects should allow to use an arbitrary network topology. Hardware/software determines the topology at system start-up and initializes routing tables, etc.

2.2.3 Reliability

Applications for parallel computing can be roughly divided into two main classes, scientific and business computing. Especially in the business field, corrupted or lost message data cannot be tolerated. To guarantee data delivery, protocol software of traditional WAN/LAN networks compute CRCs, buffer data, acknowledge messages, and retransmit corrupted data. This protocol layer has been identified as one main reason for poor latency in current networks. For clusters with their needs for low latency and thin protocol layers, this overhead must be minimized.

First, cluster interconnects with their short range physical layers have proven to be almost error-free. The computation of CRCs can be easily done on-the-fly by the NI itself. Possible errors can be signaled to software through interrupts or status registers. To relieve software from buffering message data, the NI could also temporary buffer the message data and initiate retransmissions in case of errors. Overall, the cluster interconnect should present itself to the user as a reliable network without additional software overhead for safe data transmission.

2.3 General architecture

A general design decision must be made between a dumb NI, which is controlled and managed by the CPU, and an intelligent and autonomous NI performing most of the work by itself. The first solution has the advantage of low design effort resulting in short time-to-market and redesign costs. On the other hand, enabling the NI to do jobs, such as data transfer or matching receiver ID with its network address/path, can free the microprocessor from this work and reduce start-up latency for message transfers.

Advantages of both methods can be glued together by adding a dedicated communication processor to the system [23]. This node design has been chosen for some parallel architectures (Intel Paragon, MANNA [24]) and resulted in good performance values, especially for communication intensive applications. In the following, the two main trade-offs are presented.

2.3.1 Shared memory vs. distributed memory

The first decision of a designer of cluster interconnects is the memory (programming) model to be supported. The shared memory model makes the cluster network transparent to processes through a common global address space. Virtual memory management hardware and software (MMU, page tables) is used to map virtual addresses to local or remote physical addresses. Since the overhead of applying this model to the whole address space is quite large, interconnects supporting shared memory offer the ability to map remote memory pages into local applications address spaces, like DEC's (later Compaq) Memory Channel [25].

Figure 2-2. Write operation to remote memory

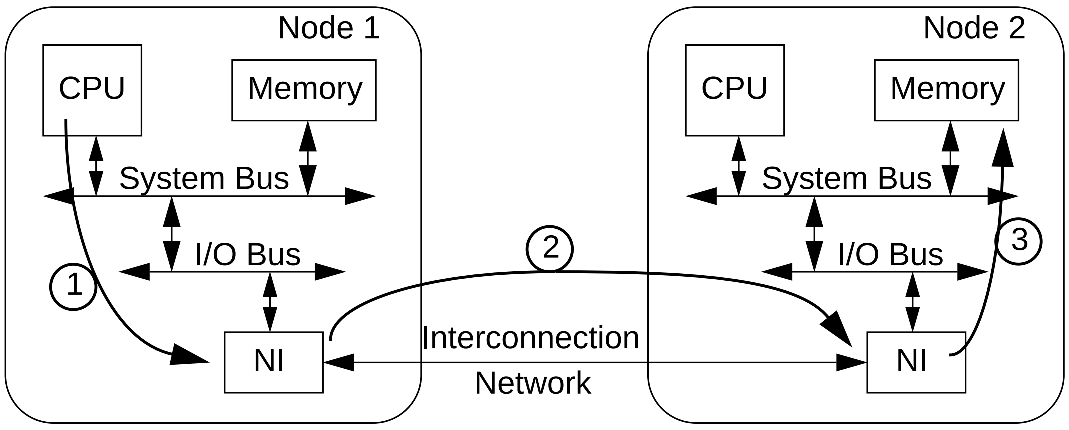


Figure 2-2 shows an example of a write operation to remote memory, where the NI resides on the I/O bus. The operation can be split up into 3 main steps, which are labeled with their according number:

1. the CPU writes the message data to a shared memory region, which virtual memory address is mapped to the NI on the I/O bus

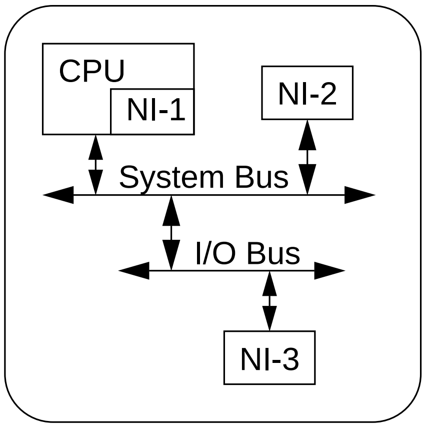
2. the NI indexes an address translation table with the write address to determine the destination node of the transaction. It then transfers data to the remote node for further processing, together with a remote write address
3. the destination node receives the data, and uses the address to write data to local memory. If the address is virtual, it has to do another translation step. But this could also already have been done by the sending NI. This depends on whether the shared address space uses virtual or physical addresses

A lot of work has to be done by the NI, if the virtual shared memory is intended to be cache-coherent across all cluster nodes, as known from SMP systems. A cache coherence protocol must observe the memory space on a cache line or page base. Writes must be propagated to all nodes owning a copy of the memory cell, or these copies must be invalidated. For short, the overhead of cluster-wide cache coherence can be manageable for small systems, but gets inefficient for large node numbers. The only remaining large-scale shared memory supercomputer today is the SGI Origin [26] with its so-called ccNUMA architecture.

In the distributed memory model, message passing software makes the network visible to applications. Data can be sent to other nodes through send/receive API calls. Compared to the shared memory model, the user has to explicitly call communication routines to transfer data to or from the network. Besides Memory Channel and SCI, which support the shared memory model, all remaining interconnects presented here rely on the message passing model.

2.3.2 NI location

Figure 2-3. Possible NI locations



The location of the NI inside a system has a great impact on its performance and usability. In general, the nearer it is to the microprocessor, the more bandwidth is typically available.

As depicted in Figure 2-3, there are three possible locations for the NI:

NI-1

An interesting solution is support for communication at the instruction set level inside a microprocessor. By moving data into special communication registers, it is transferred into the network at a rate equal to the processor speed. This technique has been realized in the past in some architectures; its most famous representative is the Transputer [27] from INMOS. Through four on-chip links at full processor clock speed, the Transputer was an ideal candidate as a building block for grid-interconnected massive parallel computers. Similar implementations are the iWarp [28] or related systolic architectures.

Although these architectures are very interesting from the designers view, the market for this kind of microprocessors proved to be too small. Most implementations reached the prototype phase, but had no commercial success. Some research projects also tried to include a network interface at the cache level, but this saw the same fate. Another try in this direction is the Alpha 21364 (EV8) microprocessor [29], which has 4 on-chip inter-processor links, each providing a data rate of 6.4 Gbyte/s. But Compaq has recently announced the discontinuation of the family of Alpha CPUs, so that the EV8 microprocessor will not be fabricated.

NI-2

Assuming a high performance system bus design, this location is an ideal place for a network interface. Todays system buses offer very high bandwidths in the range of several Gbytes/s. Common cache coherence mechanisms can be used to efficiently observe the NI status. The processor could poll on cache-coherent NI registers without consuming bus bandwidth. If the register changes its state (e.g., a status flag is set to indicate message arrival), the NI could invalidate the observed cache line. On the next load instruction, the new value is fetched from the NI. DMA controllers can read/write data from/to main memory using burst cycles at a very high bandwidth. Although there are several advantages to design the NI with a system bus interface, only a few NIs are implemented in this way. The reason for this is that each processor has its own bus architecture and thus ties an NI implementation to a specific processor. The market for cluster interconnects is not yet large enough to justify such a specialization. Furthermore, commercial interests are likely to prevent the upcoming of standard processor bus architectures, even though more than just SANs would benefit from them. Only proprietary interconnects can be designed for the system bus, an example is the SAN adapter of the IBM SP2.

NI-3

Most current interconnects have I/O bus interfaces, mainly PCI. The reason is the great acceptance of PCI as a standard I/O bus. PCI-based NIs can be plugged into any PC or workstation, even forming heterogeneous clusters. A 32 bit/33 MHz PCI device can deliver a peak data rate of 132 Mbyte/s, which can be nearly reached with long DMA bursts. To avoid that the PCI bus becomes the main bottleneck between system buses and physical layers with gigabytes per second bandwidth, most SANs have already moved on to 64 bit/66 MHz PCI bus interfaces. Since the I/O bus even then remains a major bottleneck, SAN developers await the implementation of upcoming I/O bus standards like PCI-X [30] and 3GIO [31]. But a transition only makes sense when they are widely used in the mainstream PC industry. A disadvantage of the I/O bus location is the loss of properties such as cache coherence.

Most interconnects presented in this chapter use the I/O bus as their interface to the host.

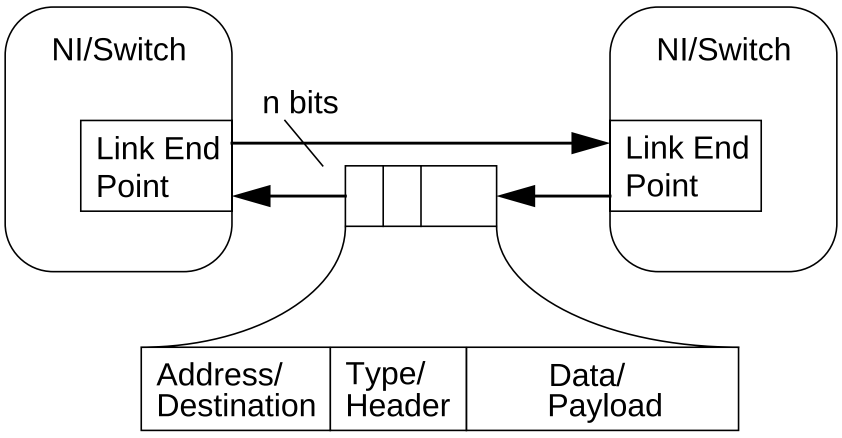
2.4 Design details

In the following, a closer look is taken at some specific implementation details. Small modifications of the hardware can have a great impact on the NIs overall performance. This section focuses on various main mechanisms for interconnection networks. Additional literature [32], [33] is recommended for an even more detailed analysis.

A general rule of thumb could be: Keep the frequent case simple and fast. For example, mechanisms for error detection and correction should be implemented in a way that they do not add overhead to error-free transmissions. In the very rare case of a transmission error, some overhead can be accepted, since error rates of current physical layers are very low. The NI should also be able to pipeline data transfers. So the head of a message packet can be fed into the network, even if the tail is still fetched from memory. This enables low start-up latencies and good overall throughput.

The term link protocol is used for the layout of messages, which are transmitted over the physical layer and the interaction between communicating link endpoints. Figure 2-4 shows two link endpoints (which could reside in a NI or a switch), connected by two unidirectional channels for sending and receiving data. Also, it depicts the general layout of a message. Typically, message data is enclosed by special control datawords, which can be used to detect start/end of message data and to signal link protocol events (receiver cannot accept more data, request for retransmission, etc.).

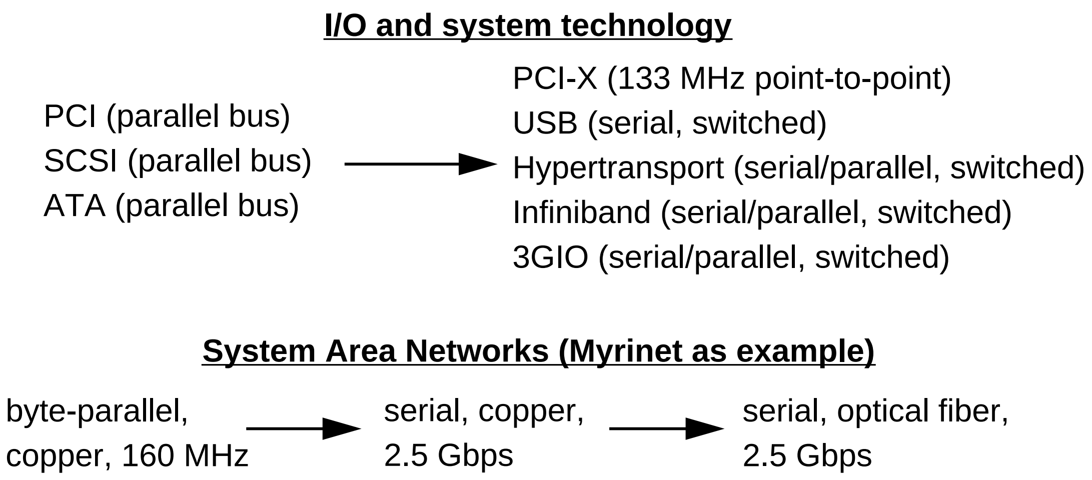
Figure 2-4. A bidirectional link and the general message format



2.4.1 Physical layer

Choosing the right physical medium of a channel is a trade-off between raw data rate, availability and cable costs. Copper is still the most used physical medium for link cables, but optical links, which have been broadly used to enhance the capacity of Wide Area Networks, are on the verge of penetrating the LAN and SAN markets. Myricom, one of the leading SAN suppliers, announced in July 2001 fiber links and the willingness to replace all copper-based cables with fiber within a few months.

Figure 2-5. Ultra-fast serial, switched connections replace parallel bus architectures


I/O and system technology
PCI (parallel bus)
SCSI (parallel bus)
ATA (parallel bus)
PCI-X (133 MHz point-to-point)
USB (serial, switched)
Hypertransport (serial/parallel, switched)
Infiniband (serial/parallel, switched)
3GIO (serial/parallel, switched)
System Area Networks (Myrinet as example)
byte-parallel, copper, 160 MHz
serial, copper, 2.5 Gbps
serial, optical fiber, 2.5 Gbps

Another trend to observe is the replacement of medium-fast parallel links with high-speed serial connections. This is not only true for the SAN market, but also for the whole I/O infrastructure in PCs and workstations. Figure 2-5 shows this trend by listing current and future I/O and network technologies. Newest I/O and system technologies start with serial implementations, but provide an upgrade path for using multiple connections in parallel. They all move from a bus-based topology to a full-switched network for better bandwidth utilization and easier implementation. Myrinet serves here as an example for the SAN market. It starts byte-parallel, goes serial, then switches from copper to fiber. The next announced step is the use of multiple serial connections per NIC.

One of the main reasons for using serial connections is the reduced pin count on switches. Latest switch technology with parallel links is limited by the pin count of IC devices. To transmit signals at a high clock rate (200-500 MHz) and a reasonable power consumption, the Low Voltage Data Signaling (LVDS) technique (two wires transmit complementary current levels of signals) is used. So an 8x8 unidirectional switch with 32 bit differential signal lines would result in 1024 (= (8+8)*32*2) pins only for the links, which is at the upper limit of todays IC packaging. Bytewide links, as used by most SANs, have been a good compromise for the last years. Network switches of moderate sizes can be built, while raw data rate still exceeds the one of serial mediums.

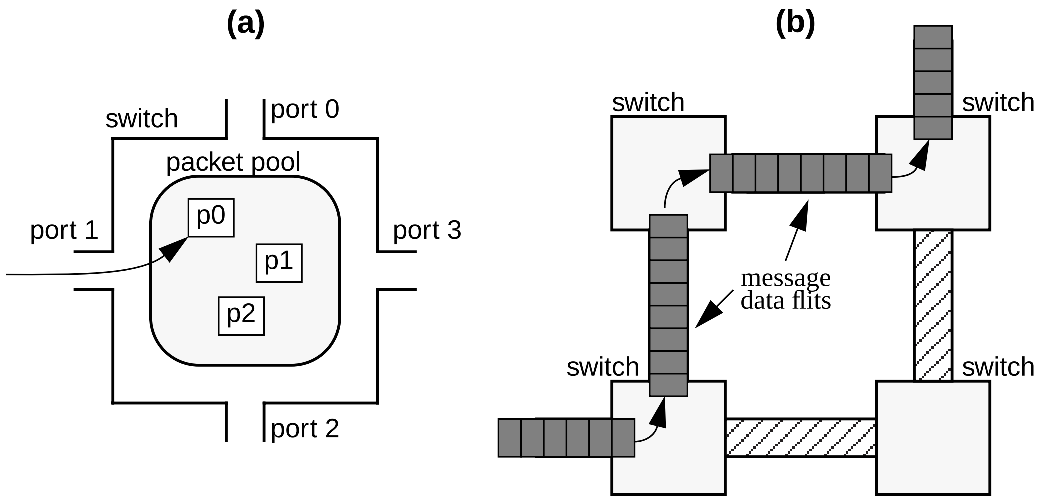
But recent developments have made it possible to transmit signals via copper cables at rates of 1-3 Gbps, with 10 Gbps technology on the horizon. Another limitation of electrical transmission at these fast rates is the limited range. Normally, only a few meters can be spanned, until the signals need to be received or refreshed. And signals become more sensitive to noise effects from parallel bit lines or even other electrical equipment in the surrounding. Optical layers have a clear advantage here, since an optical fiber does not emit any electromagnetic radiation at all. And techniques like Dense Wavelength Division Multiplexing (DWDM) or Time Division Multiplexing (TDM) promise to lift the data rate on optical fibers to 10 Gbps and more.

2.4.2 Switching

The term switching refers to the transfer method of how data is forwarded from the source to the destination in a network. Two main packet switching techniques, as depicted in Figure 2-6, are used in todays networks, store & forward and cut-through switching. The first stores a complete message packet in a network stage before the data is sent to the next one. This mechanism needs an upper bound for the packet size (MTU, Maximum Transfer Unit) and some buffer space to store one or several packets temporary.

In Figure 2-6 (a), packet p0 just arrived at the switch through port 1 and is placed into the packet buffer pool. Packets p1 and p2 have been received in total and are now forwarded towards their destination through different ports. This is the common switching technique found in LAN/WANs, because it is easier to implement and the recovery of transmission errors involves only the two participating network stages.

Figure 2-6. Packet switching techniques



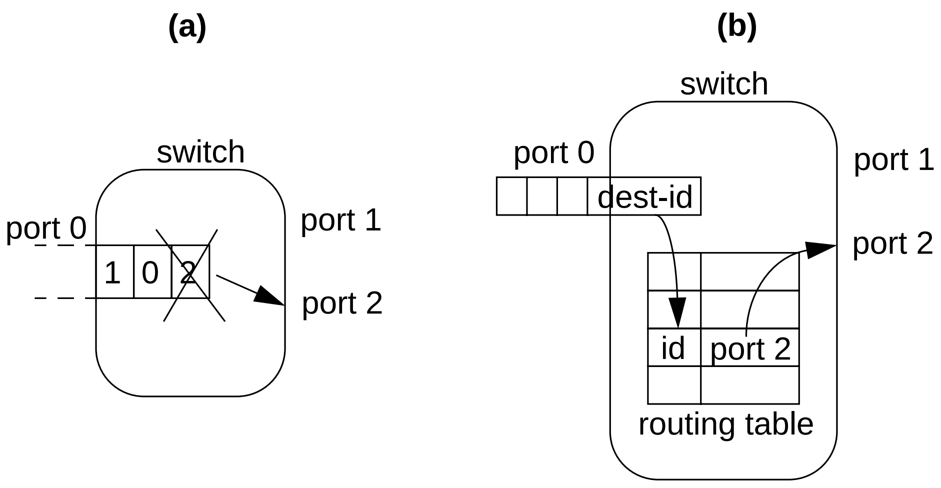
Newer SANs like ServerNet, Myrinet and QsNet use cut-through switching (also referred to as wormhole switching), where the data is immediately forwarded to the next stage as soon as the address header is decoded. In Figure 2-6 (b), one sees how a message finds its way through the network like a 'worm'. Low latency and the need for only a small amount of buffer space are the advantages of this technique. But error handling is more complicated, since more network stages are involved. Corrupted data might be forwarded towards the destination before it is recognized as erroneous.

2.4.3 Routing

The address header of a message carries the information needed by routing hardware inside a switch to determine the right outgoing channel, which brings the data nearer to its destination. Although a lot of deterministic and adaptive routing algorithms have been proposed, the latter will not be studied here. Adaptive routing schemes try to find dynamically alternative paths through the network in case of overloaded network paths or even broken links. But adaptive routing has not found its way into real hardware yet. Two mechanisms are used in todays interconnects: source-path/wormhole and table-based routing [34].

In Figure 2-7 (a), an example of wormhole routing [35] is given. A message enters a switch on port 0 and carries the routing information at the head of the message packet. As soon as the first dataword is received, routing hardware can determine the outgoing channel. Used routing data is stripped off, so the routing information for the next switch now leads the message. The entire path to the destination is attached to a message at its source location.

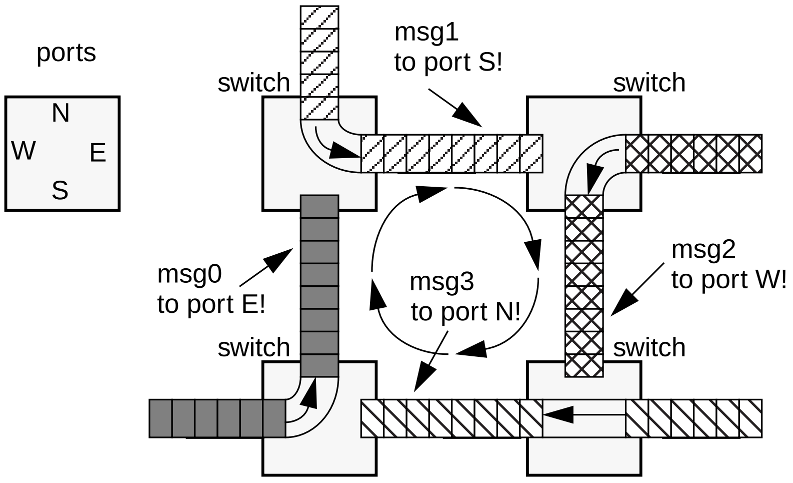
Figure 2-7. Routing mechanisms



In Figure 2-7 (b), a switch containing a complete routing table is shown. For each destination node its corresponding port is stored. If messages enter the switch a table lookup determines the right outgoing channel. Routing table size is proportional to the number of nodes, which can be a limiting factor for large cluster configurations with hundreds or even thousands of nodes. The former method is easier to implement and faster, whereas the latter one provides more flexibility. The routing table could provide alternative paths, if the current addressed path is overloaded. Or based on link utilization information the routing engine could try to find the fastest path towards the destination. But with such a non-deterministic routing one needs to be careful. Link protocols could rely on the fact that messages are delivered in order, which is not assured with such a form of adaptive routing. Another problem is the prevention of a so-called livelock, where a message is always routed over alternative paths, but never reaches the final destination.

A problem for both routing mechanisms is the avoidance of deadlocks. A deadlock appears when several messages block each other in such a way, that no message can reach its destination and the network is blocked in total. This situation is depicted in Figure 2-8. All messages form a circle of chained, blocked port requests. No message can progress, thus the network is jammed up. One can solve this problem by restricting the routing of messages in such a way that these circles of requests cannot appear. One possible solution is a strict x-y dimension routing in 2D meshes, where all messages are first routed in the horizontal direction, and then vertical. Many other solutions exist [36], more or less complex and efficient.

Figure 2-8. Messages forming a deadlock



2.4.4 Flow control

Flow control [37] is used to avoid buffer overruns inside link end points, which can result in the loss of data. Before the sender can start a transmission, the receiver must signal the ability to accept the data. One possible solution is a credit-based scheme, where each sender gets a number of credits from the receiver. On each packet transmission, the sender consumes a credit point and stops when all credits are consumed. After freeing some buffer space, the receiver can restart the transmission through handing additional credits to the sender. Or, the receiver can simply signal the sender if he can accept data or not. In both cases, the flow control information travels in the opposite direction relative to the data (reverse flow control). For example, Myrinet inserts STOP and GO control bytes into the opposite channel of a full-duplex link to stop or restart data transmission on the sender side.

2.4.5 Error detection and correction

Though todays physical layers have very low error rates, the network must offer some mechanisms for error detection and possibly correction in hardware. In the era of user-level NI protocols, it is no longer acceptable that software has to compute a CRC. This task can easily be done in hardware. For example, the NI adapter can compute a CRC on the fly while data is transferred to it. This CRC is appended to the message data and can be checked at each network stage. If an error is detected, the message can be marked as corrupted. The receiver can then send a request for retransmission back to the sender. But, of course, this assumes that the complete data is buffered on the sender side. Especially in fields like business computing with its need for fault-tolerant hardware it is also common to replicate hardware. E.g., some vendors add another full network for redundancy and always transmit data via both connections. The additional costs can be accepted for applications like transaction servers, where even a few minutes of system failure can produce a significant drop in sales volume. But in more cost-sensitive areas like Cluster Computing users tend to handle program failure in software via checkpointing or the like.

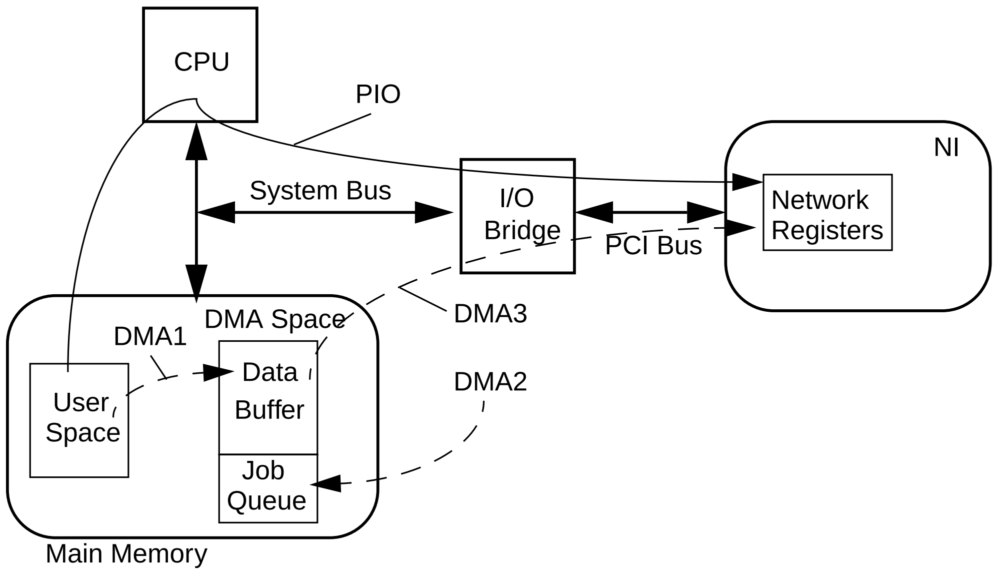
2.5 Data transfer

Efficient transfer of message data between the nodes main memory and the NI is a critical factor in achieving nearly the physical bandwidth in real user applications. To reach this goal, modern NI protocol software involves the OS only when the network device is opened or closed by user applications. Normal data transfer is completely done in user-level mode by library routines to avoid the costs of OS calls. The goal is a zero copy mechanism, where data is directly transferred between the user space in main memory and the network. Examples are shown for a NI located on the PCI bus, since this is the current (but not preferred, as earlier mentioned) location of todays network adapters. Also, the focus is more on interconnects for message passing because of the broader design space.

2.5.1 Programmed I/O versus Direct Memory Access

Message data can be transferred in two ways: Programmed I/O (PIO), where the processor copies data between memory and the NI, and Direct Memory Access (DMA), where the network device itself initiates the transfer. Figure 2-9 depicts both mechanisms on the sender side. PIO only requires that some NI registers are mapped into the user space. The CPU is then able to copy user data from any virtual address directly into the NI and vice versa. PIO offers very low start-up times, but gets inefficient with increasing message size, since processor time is consumed by simple data copy routines. DMA needs a bit more setup time, since the DMA controller inside the NI normally needs physical addresses to transfer the correct data. Most interconnects offering DMA transfer require that pages are pinned down in memory, so the OS cannot swap them out to disk. This makes it feasible to hand over physical addresses to the NI, but adds an additional copying step to transfer the user data into the DMA region (loss of zero copy property). After data is copied (step DMA1 in Figure 2-9), the processor starts the transfer by creating an entry in a job queue (DMA2), which can reside either in main memory or the NI. The NI sets up a DMA transfer to read the message data from memory (DMA3), which is then fed into the network. DMA is not suitable for small messages, but it relieves the processor so it can do useful work in case of large messages.

Figure 2-9. PIO vs. DMA data transfer



Several factors influence the performance of both mechanisms. The simplest PIO implementation writes message data sequentially into a single NI register, which resides in I/O space. This normally results in single bus cycles and poor bandwidth. To achieve an acceptable bandwidth, the processor must be able to issue burst cycles. This can be done by choosing a small address area as target, which is treated as memory. Writing on these consecutive addresses enables the CPU or the I/O bridge to apply techniques like write combining, where several consecutive write operations are assembled in a special write buffer and issued as burst transaction. This mechanism can be found in most modern microprocessor architectures. Another solution would be an instruction set support for cache control (cache line flush, etc.), as implemented in the PowerPC architecture. Since the PCI bus implements variable-length burst transactions, a DMA controller inside the NI could try to read/write a large block of data in one burst cycle. Experiments have shown that it is possible to reach about 90 % of the peak bandwidth with long bursts (110-120 Mbyte/s on a 32 bit/33 MHz PCI bus with 132 Mbyte/s peak bandwidth).

To sum it up, PIO is superior to DMA for small messages up to a certain size where the copy overhead stalls the processor too long from useful work. If one recalls that the majority of the typical network traffic is caused by small messages, it becomes clear that an NI designer should implement support for both mechanisms.

2.5.2 Control transfer

If DMA is used for transferring message data, another critical design choice is the mechanism on how to signal the microprocessor the complete reception of a whole message. This is often referred to as control transfer. In polling mode, the CPU continuously reads an NI status register. The NI sets a flag bit in case of a completed transaction. If the NI resides on the I/O bus, this could waste a lot of valuable bandwidth. As an improvement, the NI could mirror its status into main memory. This would enable the processor to poll on cache-coherent memory, thus saving bandwidth.

Another solution is to interrupt the CPU. But this results in a context switch to kernel mode, which is an expensive operation. A hybrid solution could enable the NI to issue an interrupt when message data is present for a specific time value without data transfer. A programmable watchdog timer could be located inside the NI to do this job.

2.5.3 Collective operations

So far, we have only presented mechanisms to send or receive messages in a point-to-point manner. Software for Parallel Computing often uses collective communication techniques such as barrier synchronization or multicasts. This is especially true with networks for virtual shared memory, where updated data must be distributed to all other nodes. For example, supercomputers like Cray-X/MP or AlliantFX [38] offered a dedicated synchronization network. Todays cluster networks leave this task to software, where tree-based algorithms map a broadcast to a hierarchical send/forward scheme. Only few interconnects have direct hardware support for collective operations. The barrier register of the Synfinity interconnect [39] is one example.

Networks with a shared bus like Fast Ethernet can easily broadcast data, whereas the integration into point-to-point networks like Myrinet or ServerNet is more complicated. Often the hardware realization of collective operations implicates a restriction of the network topology. This issue is an area for further improvements of todays cluster networks.

2.6 SCI

SCI (Scalable Coherent Interface) [40] is an IEEE standard (ANSI/IEEE Std 1596-1992) finally approved in 1992. The goal was to develop a network capable of interconnecting multiple systems into one unified distributed memory machine. It should overcome the limitations of bus-based approaches, which have a very limited scalability (up to 32/64 CPUs). The standard defines a set of hardware protocols and memory transactions, with the option of a hardware cache coherence protocol.

2.6.1 Targeting DSM systems

Several goals influenced the standardization of SCI as a Distributed Shared Memory (DSM) network. Of course, the premier goal is high performance in all relevant aspects. To compete as an interconnect for DSM machines with bus-based SMP systems, SCI needs to provide the same level of data bandwidth, message latency and low CPU overhead as its competing architectures. The continuous progress in CPU and memory speed should not outrun the network in terms of performance. The second important goal is scalability in many ways. SCI should be able to scale well beyond hundreds of nodes. That relates to the cache coherence mechanism, to the interconnect technology in terms of used media and its length, as well as to the addressing scheme to share a terascale memory space.

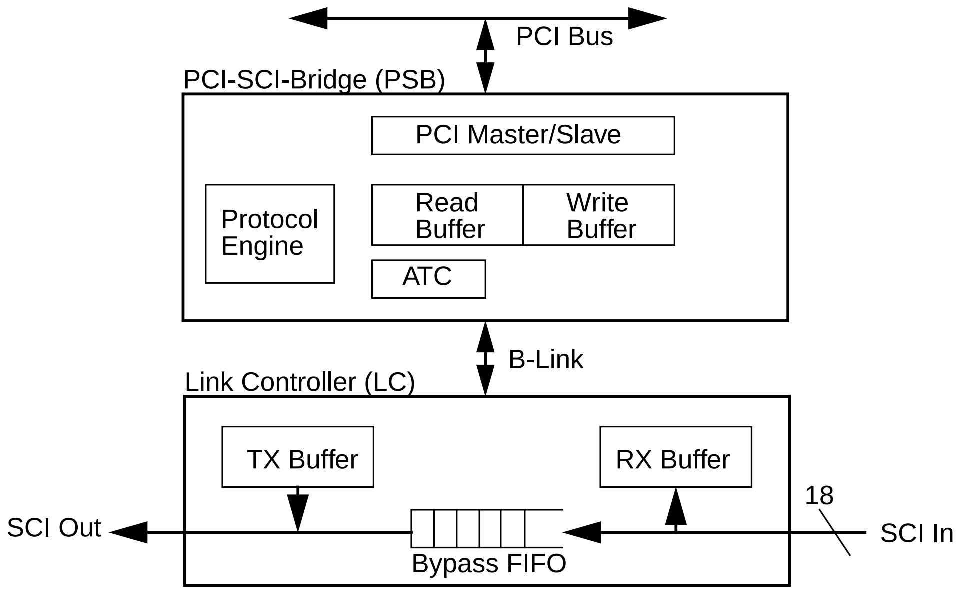
In the following, the main concepts behind SCI are presented:

- all SCI networks should be build out of unidirectional, point-to-point links. Most implementations use bit-parallel, copper-based links with a range of a few meters. This is normally sufficient to interconnect a few machines into a tightly-coupled DSM system.
- links should rely on latest signaling technology to provide best performance. Link speeds vary today between 500/667 Mbyte/s for SAN cables (LVDS, CMOS) and 1 Gbyte/s for intra-system connections in more advanced, but costly technologies (GaAs, BiCMOS).
- though it is most common today to interconnect PCs or workstations with SCI, its intention was also to connect different system components (memory modules, disk arrays, etc.) to an SCI network. But this would require a broad adoption of SCI interface technology. All relevant systems use either SMP or single-CPU machines as nodes. SCI can support up to 64k nodes, but real systems use normally a two-digit number of nodes.
- the SCI standard does not specify a specific network topology, but since most adapters have two unidirectional links (1 in-, 1 out-link), common topologies are single rings, or for a larger number of nodes 2D tori or rings of rings. With faster nodes and their improved single-node bandwidth (PCI 64 bit/66 MHz), the ring structures have become a significant bottleneck, especially for larger installations. Manufacturers have responded with small-scale switches (8/16 ports) or adapters with 2, or even 3 link connectors to form 2D/3D rings.

- the transaction layer uses split transactions (request/response) to access remote data. The standard defines a limit of 64 outstanding requests, but most implementations use a lower number due to buffer restrictions. SCI provides a global, physically distributed 64 bit address space. The upper 16 bits of an address are used as node identifier.
- though coherency is one of the two defining features of SCI (besides scalability), it is only included as option in the standard. Due to its complexity and need for additional resources (cache coherence engine, distributed directory) only few DSM producers have implemented it. Since most SCI adapters connect to I/O bus technology like PCI, cache coherence is not supported.

2.6.2 The Dolphin SCI adapter

Figure 2-10. Architecture of Dolphin's SCI card [40]



The SCI standard was intended to specify an open interface to provide interoperability across multiple vendors, connectors and devices. But the complexity of the specification has led to implementations, which concentrated on necessary features and left out others. This resulted in a market with a few proprietary incompatible solutions. Despite a few vendors of DSM systems, like Data General's AViiON [41] (now part of EMC), Convex Exemplar [42] (now HP) and Sequent's NUMA-Q [43] (now IBM), the most used SCI SAN implementation is developed by Dolphin Interconnect LCC[1].

1. www.dolphinics.com

Figure 2-10 depicts the block diagram of the two main chips on the NIC, the PCI-SCI-Bridge (PSB) and the Link Controller (LC). Both chips communicate via a proprietary B-Link bus. This decoupled architecture allows attaching multiple LC chips to the B-Link, a separate development of future generations of both chips, and the use of LC chips for a non-PCI environment. The main blocks on these chips are:

- the PCI Master/Slave Interface handles all PCI bus transactions. It also contains a DMA controller for direct memory-to-memory transfers.
- Read/Write Buffers contain slots for 128 byte data packets associated with every SCI transaction.
- the Protocol Engine manages all SCI transfers. Up to 16 read/write streams can be handles simultaneously.
- the Address Translation Cache (ATC) is used to map PCI to SCI addresses. It also contains some page attributes. The whole Address Translation Table (ATT) is located in separate SRAM on the adapter, an ATC miss triggers are reload of the referenced entry into the ATC.
- TX/RX Buffers on the LC inject/extract SCI packets into/out of the SCI link. Packets addressing other nodes are simply forwarded via a Bypass FIFO.

The latest version of the PSB (PSB66) offers a 64 bit/66 MHz PCI interface, and the most recent LC (LC3) chip offers a link bandwidth of 667 Mbyte/s. Besides the normal CPU-initiated read/write transactions some additional features were implemented. A programmable DMA engine processes a linked list of control blocks specifying data to be sent to remote nodes. This frees the CPU in case of large message transfers for better throughput. A feature called mailbox triggers on specially tagged SCI packets, which are placed into a separate message pool. An interrupt is raised on reception of such packets to make the CPU aware of the received data. Since in a multi-CPU node several concurrent write accesses can occur, a special write gathering is performed to forward data via the SCI link in larger data blocks. A store barrier mechanism offers the possibility to flush gathered write data for separate pages.

2.6.3 Remarks

SCI has been used successfully as interconnect in DSM systems, but is less efficient in a cluster environment due to certain architectural properties. A lot of components have been optimized for fine-grain communication patterns to support system-wide coherence. A relative small packet size, unidirectional links for ring structures, slow remote read operations, etc. hinder the hardware to unfold its full potential when used as cluster interconnect. Studies [44] have shown that the sustained node-to-node bandwidth on SCI is more limited than in other networks.

Another bottleneck is the limited ability to form scalable networks. A popular topology for SCI is a 2D ring structure, e.g. the largest SCI Cluster in Europe at the Paderborn Center for Parallel Computing (PC^2) uses an unidirectional 8x12 2D torus. But this also means that on a ring containing 12 nodes, all these 12 nodes have to share the link bandwidth of 500 Mbyte/s. This is a drawback of SCI systems compared to other full-duplex, fully switched solutions like Myrinet or QsNet.

All this led to a relatively low acceptance of SCI as cluster interconnect. While other technologies are used now to build Clusters of thousand or more nodes, SCI Clusters are rarely larger than 32 nodes. Vendors have responded with small-scale (6-8 ports) switches, but they can only soften the bandwidth bottleneck. Compared to the SAN market, the manufacturers of SCI-based DSM systems have developed some efficient and fast systems. But all three mentioned before (Data General, Convex and Sequent) have gone out of business or have been taken over by other companies. Most product lines have been phased out.

2.7 ServerNet

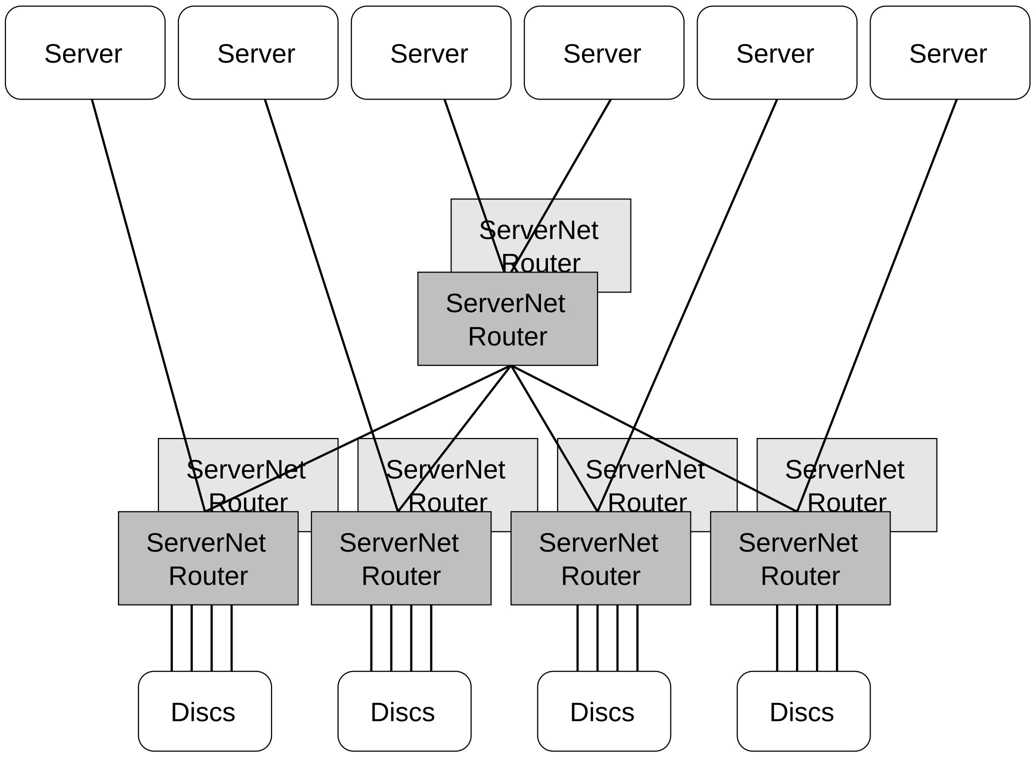
In 1995, Tandem introduced one of the first commercially available implementation of a SAN called ServerNet. Since its introduction, ServerNet equipment has been sold by Tandem (now owned by Compaq) for more than one billion dollars. In 1998, Tandem announced the availability of ServerNet II [45], the follow-up to the first version, which raises the bandwidth and adds new features while preserving full compatibility with ServerNet I.

With ServerNet, Tandem, a major computer manufacturer in the business area, addressed one of the main server problems: limited I/O bandwidth. Tandem’s customers, mainly business companies running large database applications, needed more I/O bandwidth to keep up with the growing data volumes their servers should be able to handle. So ServerNet was intended as a high bandwidth interconnect between processors and I/O devices, but turned quickly into a general purpose SAN.

2.7.1 Scalability and reliability

With scalable I/O bandwidth as the primary goal, ServerNet consists of two main components: endnodes with interfaces to the system bus or various I/O interfaces and routers to connect all endnodes to one clustered system. One main design goal was the ability to transfer data directly between two I/O devices, thus relieving processors of plain data copy jobs. By being able to serve multiple simultaneous I/O transfers, ServerNet removes the I/O bottleneck and offers the construction of scalable clustered servers. Figure 2-11 shows a sample system configuration. Most ServerNet configurations, besides the Himalaya series of Tandem itself, use an I/O (PCI) adapter instead of directly attaching to the system bus.

Figure 2-11. A sample ServerNet network [46]



2.7.2 Link technology

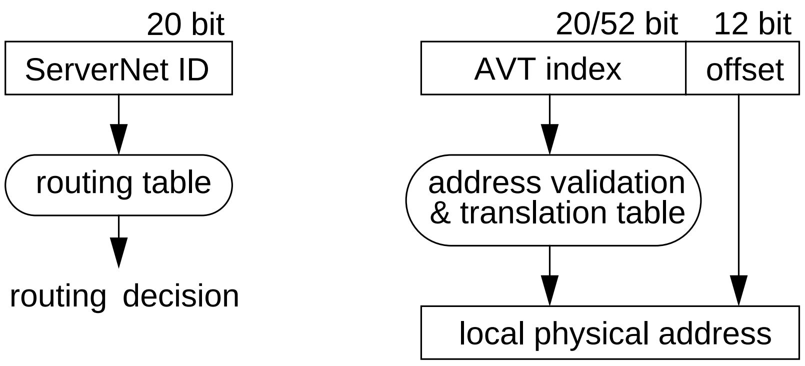
ServerNet is a full duplex, wormhole switched network. The first implementation uses 9 bit parallel physical interfaces with LVDS/ECL signaling, running at 50 MHz. ServerNet II raises the physical bandwidth to 125 Mbyte/s, driving standard 8b/10b serializers/ deserializers to connect to 1000BaseX (Gigabit Ethernet) standard cables. With the support of serial copper cables, ServerNet is able to span across significantly longer distances. For compatibility reasons, ServerNet II components also implement the interface of the first version. Together with additional converter logic, ServerNet I and II components can be mixed within one system, enabling the customer to easily upgrade an existent cluster with components of the new generation without the need to replace ServerNet I components. Links operate asynchronously and avoid buffer overrun through periodic insertion of SKIP control symbols, which are dropped by the receiver. Special flow control symbols are exchanged between two link endnodes to ensure, that data does not have to be dropped due to lack of buffer space.

2.7.3 Data transfer

The basic data transfer mechanism supported is a DMA-based remote memory read/write. An endnode can be instructed to read/write a data packet of up to 64 byte (512 byte in ServerNet II) from/to a remote memory location. The address of a packet consists of a 20 bit ID and a 32/64 bit address field. The ServerNet ID uniquely identifies an endnode and the route towards the destination.

The address can be viewed as a virtual ServerNet address. The lower 12 bits are the page offset, whereas the upper bits are an index into the Address Validation Translation Table (AVT). Via this indirection, the receiver is able to check read/write permissions of the sender, as depicted in Figure 2-12. To support communication models, for which the destination of the message is not known in advance, the address can also specify one of several packet queues, to which data is then appended.

Figure 2-12. ServerNet address space [45]



A main feature of ServerNet is its support for guaranteed and error free in-order delivery of data on various levels. On the link layer, a CRC check is done in each network stage to validate the correct reception of the message. Each link is checked through the periodical exchange of heartbeat control symbols. Each endpoint assures correct transmission by sending acknowledges back to the sender. In case of errors, the hardware invokes driver routines for error handling.

2.7.4 Switches

ServerNet I offers 6 port switches, which can be connected in an arbitrary topology. Router II, the next generation of ServerNet switches, raises the number of ports to 12. In-

and outports contain FIFOs to buffer a certain amount of data and are connected through a 13x13 crossbar. The additional port is used to inject or extract control packets. Each Router offers a JTAG and processor interface for debug or management services. One special feature of ServerNet switches is the ability to form so called Fat Pipes. Several physical links can be used to form one logical link, connecting two identical link end-points. The switches can now be configured to dynamically choose one of the links, which leads to a better link utilization under heavy load.

2.7.5 Software

The good reliability of the ServerNet hardware makes it possible to implement low over-head protocol layers and driver software. Tandem clusters run the UNIX and WindowsNT operating systems. With its packet queues, the second generation of this SAN introduces a mechanism to efficiently support the message passing model of the Virtual Interface Architecture (VIA)[1] [47], a message layer specification for cluster networks.

To provide an easy way of managing the network, a special sort of packets is defined called In Band Control (IBC) packets. These packets use the same links as normal data packets, but are interpreted by an 8 bit microcontroller. The IBC protocol is responsible for initialization, faulty node isolation and several other management issues. IBC packets are used to gather status or scatter control data to all ServerNet components.

2.7.6 Remarks

Though it is hard to find detailed performance numbers, ServerNet technology seems to be a very reliable and, with its second generation, also high performance SAN. ServerNet focuses on the business/server market and has only poorly been accepted by researchers in the area of technical computing so far, though it would be interesting to see the performance of message passing libraries such as MPI and PVM.

ServerNet implements a lot of properties, which are extremely useful for cluster computing: error handling on various levels, a kind of protection scheme (AVT), standard physical layers (1000BaseX cables) and support for network management (IBC). But despite its commercial success, Compaq has lately announced the discontuinuation of ServerNet in favor of the upcoming InfiniBand interconnect. Whether the ServerNet technology is further developed to fit the Infiniband requirements, or it is abandoned in favor of external network technology is not clear yet.

1. www.viarch.org

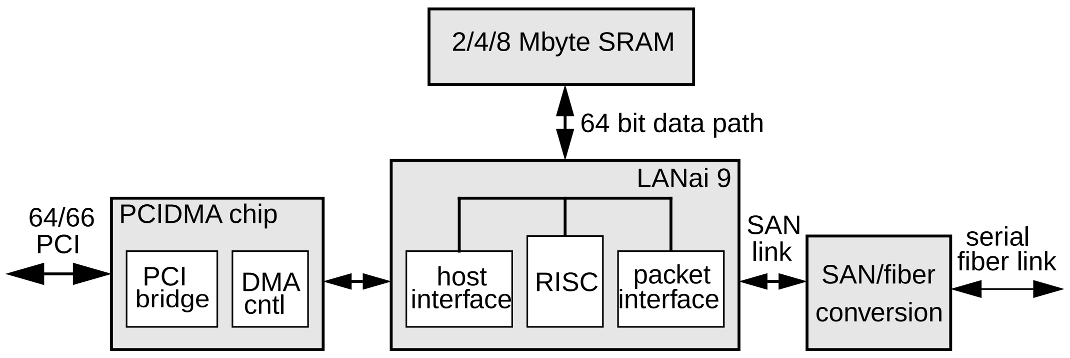
2.8 Myrinet

Myrinet [48] is a SAN evolved from supercomputer technology and the main product of Myricom[1], a company founded in 1994. It has become quite popular in the research community, resulting in 150 installations of various sizes through June 1997. Today thousands of clusters are equipped with the Myrinet network, with some really large installations of 256+ nodes. A major key to its success is the fact that all hardware and software specifications are open and public.

The Myrinet technology is based on two earlier research projects, namely Mosaic and Atomic LAN by Caltech and USC research groups. Mosaic was a fine grain supercomputer, which needed a truly scalable interconnection network with lots of bandwidth. The Atomic LAN project was based on Mosaic technology and can be regarded as a research prototype of Myrinet, implementing the major features such as network mapping and address-to-route translation; however, with some limitations (short distances (1 m) and a topology (1D chains) not very suitable for larger systems). Eventually, members of both groups founded Myricom to bring their SAN technology into commercial business.

2.8.1 NIC architecture

Figure 2-13. Architecture of the latest Myrinet-2000 fiber NIC [49]



Regarding the link and packet layer, Myrinet is very similar to ServerNet (or vice versa). They differ considerably in the design of the host interface. A Myrinet host interface consists of two major components: the LANai chip and its associated SRAM memory. The LANai is a custom VLSI chip and controls the data transfer between the host and the network. Its main component is a programmable microcontroller, which controls DMA engines responsible for the data transfer directions host to/from onboard memory and

1. www.myri.com

memory to/from network. So message data must first be written to the NI SRAM, before it can be injected into the network. This intermediate buffering adds some latency, the more the larger the message is. The SRAM also stores the Myrinet Control Program (MCP) and several job queues. A recent improvement is the upgrade of the link from a byte-parallel copper-based implementation to a serial optical fiber. The basic architecture is depicted in Figure 2-13.

More than other SAN developers Myricom has continuously improved the architecture and the hardware components of the Myrinet network:

- the first version of the NIC was based on Sun's SBus. But with the broad adoption of PCI and PCs becoming the main node systems instead of RISC workstations, a PCI-based NIC was developed. The first version implemented a 32 bit/33 MHz PCI interface. A later version upgraded to 64 bit/66 MHz.
- the LANai chip started at 33 MHz, latest versions (v9) are running at 133/200 MHz.
- on-board SRAM was steadily enlarged, from 512 Kbyte up to latest NIC versions with 8 Mbyte. This was necessary to satisfy the need for more buffer space on the NIC to store larger MCPs, provide more space for message data, etc.
- first links were full-duplex byte-parallel links running at 1.28 Gbit/s in each direction over copper cables up to 10 m. With Myrinet-2000 a serial version of the copper-based links was introduced, running at 2 Gbit/s.

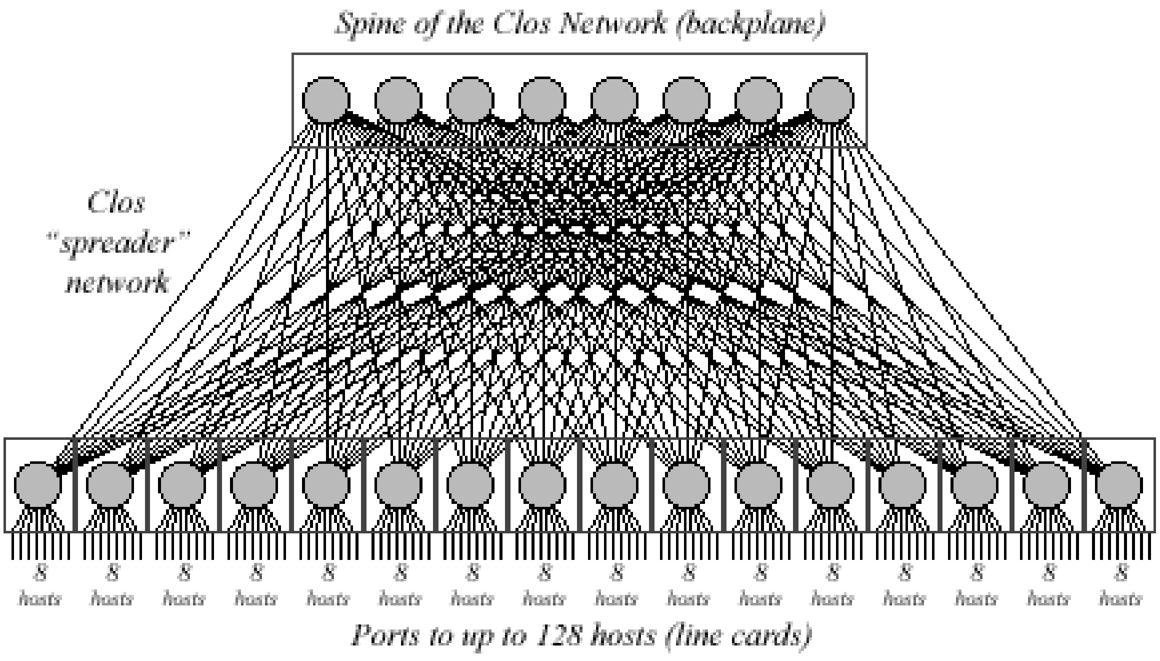
2.8.2 Transport layer and switches

Data packets can be of any length and are forwarded using cut-through switching. They consist of a routing header, a type field, the payload and a trailing CRC. Myrinet uses wormhole routing. While entering a switch, the first header byte encodes the outgoing port. The switch strips off the leading byte and forwards the remaining part of the packet to the appropriate output port. When the packet enters its destination host interface, the routing header is completely eaten up and the type field leads the message. Special control symbols (STOP, GO) are used to implement reverse flow control.

On the link level, the trailing CRC is computed in each network stage and substituted for the previous one. A packet with a nonzero CRC entering a host interface then indicates transmission errors. MTBF (Mean Time Between Failure) times of several million hours are reported for switches and interfaces. On detection of cable faults or node failure, alternative routes are computed by the LANai. To prevent deadlocks from long-term blocked messages, time-outs generate a forward reset (FRES) signal, which causes the blocking stage to reset itself.

Latest Myrinet switch technology [50] is build around a single crossbar chip with 16 ports, called XBar16. A rack-mountable line card equipped with one XBar16 offers 8 front panel ports for connecting nodes. 8 ports connect to a backplane interface. Multiple line cards can now be inserted into racks of different size. A backplane is mounted with 8 XBar16 chips, forming the spine of a Myrinet network. The preferred topology for a Myrinet network is the Clos network. Compared to other popular network topologies like 2D/3D tori/grids, hypercubes, fat trees, etc., a Clos network offers full-bisection bandwidth, full rearrangability, good scaling and multi-path redundancy. It is the preferred topology for the latest large-scale Myrinet installations with 256 and more nodes. Figure 2-14 shows a sample configuration for a 128 node Clos network.

Figure 2-14. A Clos network with 128 nodes [50]



2.8.3 Software and performance

As mentioned before, all Myrinet specifications are open and public. The device driver code and the MCP are distributed as source code to serve as documentation and base for porting new protocol layers onto Myrinet. This has motivated many research groups to implement their own message layers and is one of the main reasons for the popularity of Myrinet. Device drivers are available for Linux, Solaris, WindowsNT, DEC Unix, Irix and VxWorks on Pentium (Pro), Sparc, Alpha, MIPS and PowerPC processors. A patched GNU C-compiler is available to develop MCP programs.

The performance of the Myrinet network is highly depended on the software layer used to access the Myrinet hardware. Quite a number of software layers have been implemented, e.g. Active Messages [51], Fast Messages [52], BIP [53], Parastation [54] and many others. Poor quality of early Myrinet software was one issue leading to the development of many external implementations.

Over the last two years, Myricom has developed with the GM message layer [55] a quite stable and fast software layer, which is broadly used now for new installations. The second message layer with a significant user base is SCore [56] from the Japanese Real World Computing Partnership[1] (RWCP). Table 2-3 summarizes basic performance numbers for this two layers.

Table 2-3. Performance of GM and SCore over Myrinet [55], [56]

	GM[a]	SCore[b]
sustained bandwidth	245 Mbyte/s	146 Mbyte/s
message size with 50% BW_{max}	900 Byte	600 Byte
one-way latency	7 us	13.3 us

a. 66/64 PCI, Myrinet-2000, fiber, LANai 9 200 MHz

b. 33/64 PCI, Myrinet-SAN, copper, LANai 7 66 MHz

2.8.4 Remarks

The great flexibility of the hardware due to the programmable LANai microcontroller is one of the major advantages of Myrinet. It has attracted a lot of attention from the research community and fueled the implementation of lots of message layers on top of the Myrinet network. Another reason for the success is Myricom's policy of continuous small improvements to hardware and software.

Bottlenecks like slow onboard SRAM or LANai chips have been removed, early versions of low-performance software have been replaced. Regarding market share, Myrinet seems to dominate the SAN market right now. With shipping more than 5.000 NICs and almost 10.000 switch ports in 1Q/2001, Myrinet is the network to beat in this area. And with switch technology scaling to 1.000 nodes and more it is well prepared for future teraflop Cluster installations.

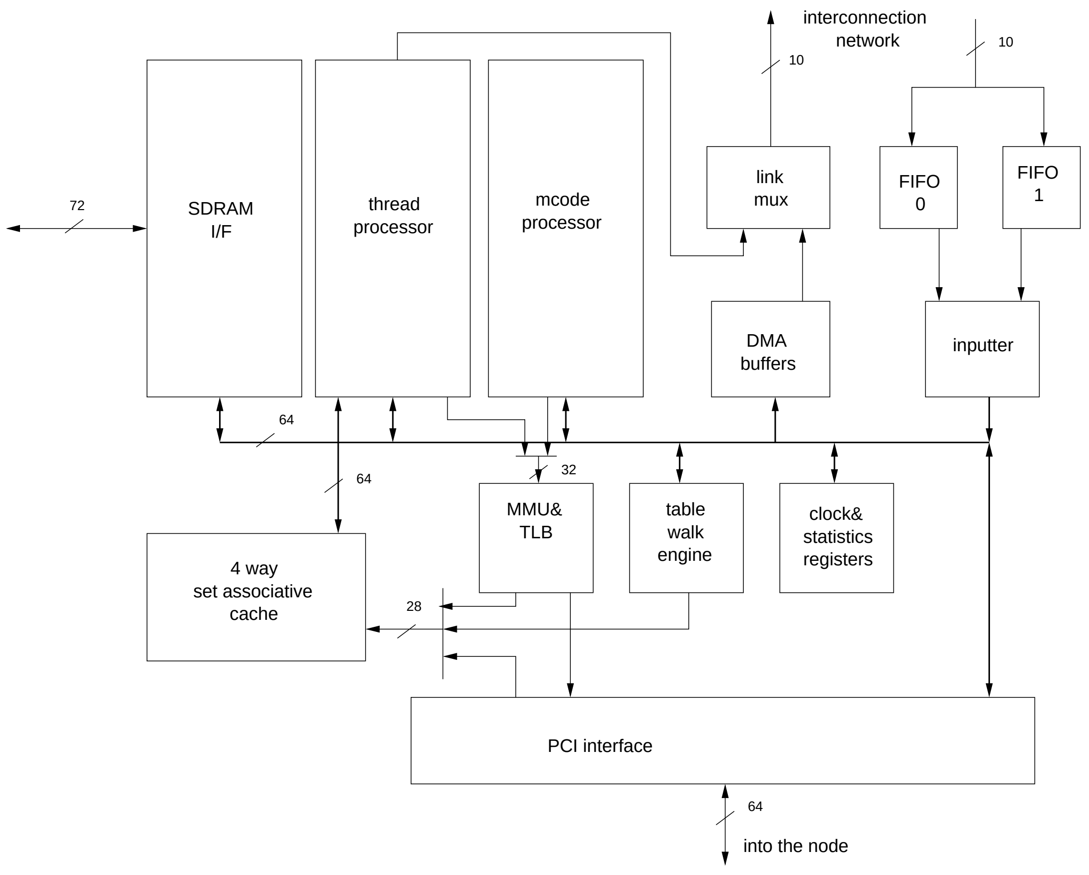
1. pdswww.rwcp.or.jp

2.9 QsNet

QsNet [57] is a SAN developed by Quadrics Supercomputers World Ltd[1]. Similar to other SANs, QsNet has its root in traditional supercomputer technology, since Quadrics emerged from the well-known supercomputer manufacturer Meiko Ldt. Influences of their cache-only machines are still visible in the QsNet architecture. QsNet is the only SAN with a seamless integration of the network interface into the nodes memory system. This unique feature of a globally shared, virtual memory space is made possible by address translation and mapping hardware directly integrated into the NIC.

2.9.1 NIC architecture

Figure 2-15. Block diagram of the Elan-3 ASIC [57]



The third generation of the Elan ASIC is the key component of the QsNet NIC. Its architecture is depicted in Figure 2-15. In the following, a brief description of the functional units is given:

- a 64 bit/66 MHz PCI interface is used for communication with the host.

1. www.quadrics.com

- full-duplex 10 bit LVDS links connect the NIC via copper cables to the network at a rate of 400 Mbyte/s per direction.
- a 32 bit microcode processor supporting up to four concurrent threads. Threads have different tasks: control of the inputter, setting up the DMA engine, scheduling of threads, and communicating with the host.
- a 32 bit thread processor, which offloads processing of higher level library tasks from the host CPU.
- a Memory Management Unit (MMU) with a Translation Look-Aside Buffer (TLB) to do table walks and translate virtual into physical addresses
- a 64 bit SDRAM interface to connect to external 64 Mbyte SRAM, together with a 8 Kbyte on-chip four-way set-associative memory cache.

2.9.2 Switches and topology

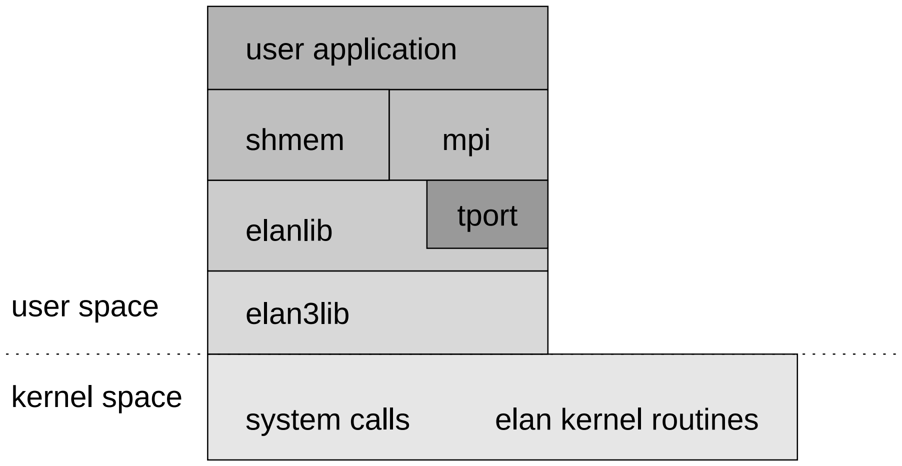
Besides the NIC, two different switches (16 and 128 port) are used to connect QsNet nodes into a fat-tree network. The basic building block is a line card with 8 Elite-3 switch chips. Each Elite-3 chip is a 8-port full-duplex switch, with two virtual channels per input link. Multiple line cards are then used to construct a full-bisection, multi-route fat-tree network.

Source-path routing is used to deliver network packets to their destination. The sender attaches a sequence of routing tags to the head of a message. Each network stage interprets the first routing tag, removes it and forwards the message towards its destination. Special tags are used to support a broadcast function, which can be utilized to send a message simultaneously to all remote nodes, or even a group of distinct nodes. At the link level, all network traffic is pipelined in a wormhole manner, with an end-to-end acknowledgment of packets. In case of transmission errors, the sending NIC retries the transmission automatically without intervention from the host side.

2.9.3 Programming interface and performance

Figure 2-16 shows the overall structure of the programming interface for a QsNet network. A layer called Elan3lib directly interacts with the hardware. Kernel routines are mainly used for initialization tasks, like mapping parts of a process local address space into a globally shared virtual address space. The Elan3lib supports a programming model with cooperating processes. The main functions are used to map/allocate memory and to set up remote DMA transfers. Processes communicate mostly via events, e.g. to synchronize the host process with a thread running on the Elan chip. The Elanlib is a higher layer, hiding all the hardware- and revision-dependent details. It offers a point-to-point message passing model with the use of tags to filter messages at the receiving side. It supports both synchronous and asynchronous message delivery.

Figure 2-16. Elan programming libraries [57]



The unique feature of QsNet is its ability to directly send data from a process virtual address space without any intermediate copying. The MMU inside the Elan-3 chip is synchronized to the MMU of the host CPU, or more exact, MMU tables are kept consistent between the QsNet NIC and the OS kernel running on the host node. That way, a user process can call a send routine of the Elanlib with a virtual address, the Elan-3 MMU translates the virtual into a physical address, either in main memory or in the SRAM on the QsNet adapter. This is made possible by extending the OS kernel with functions to ensure consistency of MMU tables. Special memory allocation functions of the Elanlib offer the possibility to map portions of the on-board SRAM into user processes, e.g. to give the NIC fastest access to DMA descriptor tables.

Table 2-4. Performance of QsNet[a] [57]

	Elan3lib	MPI
sustained bandwidth	335 Mbyte/s	307 Mbyte/s
message size with 50 % BW_{max}	900 Byte	3 Kbyte
one-way latency	2.4 us	5.0 us

a. Dual 733 MHz Pentium III, Serverworks HE chipset, Linux 2.4

QsNet currently leads all SANs in performance, due to its advanced hardware support of message passing primitives. Its unique ability to communicate directly between virtual address spaces without intermediate copies removes a lot of processing overhead. Advanced features like a hardware broadcast boost the performance of collective operations, like data multicast or synchronization barriers. This is especially true for large-scale clusters with a performance of several teraflops. To make effective use of these features, the OS kernel has to be patched, and libraries like MPI have to be highly optimized. Table 2-4 displays the main performance numbers, both for the Elan3lib and MPI.

2.9.4 Remarks

Though it offers the best performance of todays SANs, QsNet has not attracted the level of attention like e.g. Myrinet. The reason is a relative high price, in the order of three times the price of competitive solutions. A significant part of the costs is due to the large amount of SRAM on the NIC (64 Mbyte). Quadrics has a strong relationship with the High Performance Computing division of Compaq. QsNet is the preferred interconnect for their Alpha SC series[58], based on SMP nodes with multiple Alpha CPUs. Though only a few cluster installations exist, these are quite impressive. The latest one is the Terascale Computing System at the Pittsburgh Supercomputing Center (PSC). A cluster of 750 quad processor Compaq AlphaServer ES45s, each node equipped with two QsNet adapters, delivers a peak performance of 6 teraflops. At the date of installation (October 2001), this system, named ‘Le Mieux’, is the most powerful supercomputer dedicated to unclassified research.

2.10 IBM SP Switch2

Though a proprietary network not intended for the PC cluster market, the Switch2 interconnect [59] from IBM is very similar in its architecture and use to more general SAN solutions like Myrinet or QsNet. An intelligent host adapter, driven by an embedded microcontroller, send/receives data to/from the network. The first generation of SP Switch technology has been used to interconnect IBM RS/6000 machines since the mid 90’s. But at the end of the decade it was outdated with its 150 Mbyte/s link bandwidth and slow on-board logic.

To remain one of the top HPC manufacturers, IBM developed with the second generation of SP Switch technology an interconnect able to keep up the performance of SP clusters. Switch2 is a key component for IBM’s RS/6000 SP parallel machines. These machines are clusters of high-end SMP workstations. Several terascale systems are IBM SP machines, among them the most powerful supercomputer today, the ASCI White[1] machine. With its 8.192 CPUs in total, the machine is capable of delivering 12.3 teraflops, twice as much computing power than the second fastest machine (Le Mieux).

1. www.llnl.gov/asci

Though due to its proprietary interface not usable for general cluster computing, its technology is quite interesting and therefore, shortly presented here.

2.10.1 NIC architecture

Figure 2-17. Block diagram of the Switch2 node adapter [59]

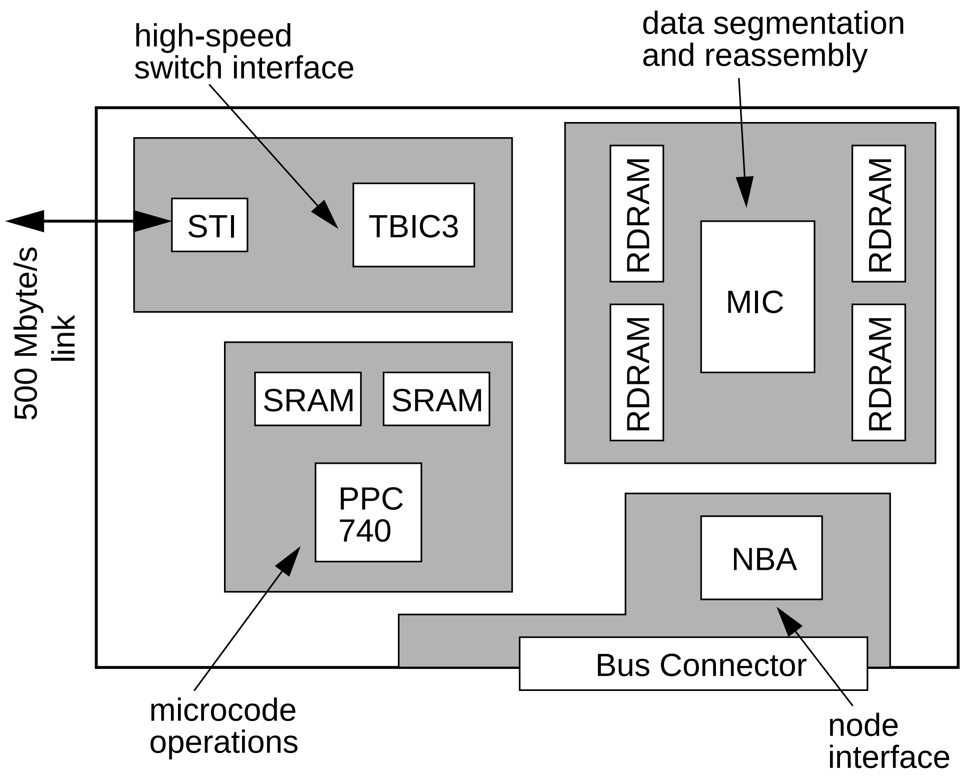


Figure 2-17 depicts the top-level architecture of the SP Switch2 host adapter. As shown in the figure, one can partition the network adapter into four regions: a high-speed switch interface, a module for data segmentation and reassembly, one for running microcode, and one region to interface to the node system. The main components of these regions are:

- a node bus adapter (NBA) controls the communication with the host via a 16 byte, 125 MHz 6XX bus connector. The 6XX bus is the main system bus of the node, directly connecting the Switch2 adapter to the CPUs and memory modules. CPUs can issue load/store instructions to access the adapter.
- a Self Timed Interface (STI) chip connects the adapter via a byte-parallel link to the network. A link is a full-duplex connection driving differential signals at 500 MHz. The link can either be an on-board connection of few inches, or a copper cable of up to 10 meters. The TBIC3 chip is an interface controller, connecting the STI to the on-board RAM and the PowerPC 740 microprocessor. It contains hardware to offload packet reassembly and segmentation from the main CPU.

- 16 Mbyte of fast Rambus RDRAM is located on the adapter to provide sufficient buffer space for message data. A Memory Interface Chip (MIC) controls the RDRAM and parallelizes multiple accesses to the RDRAM from both the NBA and the TBIC3.
- a PowerPC 740 microprocessor is used to control all on-board components via microcode. It is responsible for packet header generation, making routing decisions, handle error conditions and communicating with the host. Its microcode program is stored in 4 Mbyte SRAM, along with some other status/control information.

The node architecture is highly decoupled to allow several data transmissions to occur at the same time. While the host CPU is transferring new message data to the RDRAM, the PPC 740 may read header information from the RDRAM, and the TBIC3 is forwarding a message from RDRAM to the STI. All those datapaths at least provide a bandwidth of 1 Gbyte/s, with an aggregate bandwidth of 2.4 Gbyte/s to the RDRAM.

The general purpose PowerPC 740 microprocessor offloads a lot of tasks from the host CPU. It basically provides a low-level message passing interface to the host CPU. Higher software layers like MPI or IP then build up on these routines. To send a message, the host simply writes a work ticket into a job queue residing in the SRAM. The PPC 740 monitors this queue, and then sets up the data transfer via DMA from the NBA into RDRAM. After data is completely written to RDRAM, the microprocessor sets up header and data information for the TBIC3, which then forwards data autonomously to the STI. Message data is segmented into 1 Kbyte units. On the receiving side, the TBIC3 forwards incoming header information to the PPC 740 for further investigation. The CPU then decides, where to place message data and communicates this information to the TBIC3. The TBIC3 sets up DMA transfers to write message data into the RDRAM. Upon completion, it notifies the microprocessor, which then instructs the NBA to write the message out to the hosts main memory.

Additional to the message transfers, the Switch2 offers some advanced features. On is the generation of a Time Of Day (TOD) signal. This signal is used to synchronize all components of the network to a master TOD, even with a compensation of cable delays. This way, a synchronization of the whole network can be maintained at about 1 us.

2.10.2 Network switches

SP machines are mostly connected in a bidirectional multistage interconnection network (BMIN) topology. Similar to the fat tree network of QsNet, which is a BMIN, they offer high scalability, multi-route paths and reward communication locality. SP switches are 32-port switches, where 16 ports are normally used to connect to nodes, while the other

16 ports are used to interconnect with other switches. Such a switch contains 8 interconnected 8-port Switch3 [60] chips.

SP switches use source-path wormhole routing with a credit-based flow control scheme to prevent buffer overflow. Each Switch3 chip contains input/output ports, together with a large central buffer queue. This 8 Kbyte central buffer is used to store message data in large chunks in case its targeted output port is blocked by another message in transit. This mechanism reduces the head-of-the-line blocking, where a blocked message occupies several network stages and prevents all other messages on this path from advancing.

Each output port has two output queues assigned, one low- and one high-priority queue. High-priority packets may bypass other messages for faster delivery. Two recent additions in routing enable better performance for a SP network. First, a restricted form of adaptive routing can be used to let switches determine the fastest network path. This feature is especially helpful under heavy workloads to utilize the network in the most efficient manner. Multicast routing is provided via enabling multiple output ports to read the same packet from the central buffer queue. The specification of output ports can be either via a table lookup or encoding of routing bytes.

2.10.3 Remarks

The new generation of IBM’s SP Switch technology offers enough performance to let clusters of RS/6000 SP SMP nodes compete in the HPC market. With a link bandwidth of 500 Mbyte/s and a scalable network offering advanced features like multicast or adaptive routing, SP machines are especially well suited for large terascale installations. Three out of currently six large teraflop machines of the Accelerated Strategic Computing Initiative (ASCI) are IBM SP systems.

Up to 350 Mbyte/s sustained bandwidth has been measured for MPI applications. A relative high latency of 17 us could be declared with the highly decoupled architecture of the node adapter. The 6XX bus interface is both an advantage and a drawback. The adapter can directly access main memory, without an I/O bridge in between. But the use of the Switch NICs is limited to two versions of IBM’s POWER3 SMP Nodes. For better throughput in SMP nodes with up to 8 CPUs, each node board offers two slots, so one can attach two Switch2 adapters to each node.

Features like multicast and adaptive routing have not been implemented with such a level of hardware support in a SAN yet. It will be interesting to see, if and how other SAN manufacturers will adopt them.

2.11 Infiniband

I/O bandwidth is more and more becoming a limited resource in todays server systems. To overcome this bottleneck, a lot of different I/O technologies have been developed for various application areas. Technologies like PCI, USB, AGP, SCSI, IDE, Firewire, Ethernet, Fibre Channel, etc. are used to connect several device classes to a system, among them networking, storage, input/output and graphics. Most of them are outmoded shared-bus architectures, with poor scalability and serious bandwidth limitations. Furthermore, most of them need a significant amount of CPU intervention, eating up a lot of the performance benefit from faster microprocessors and memory modules.

To overcome the architectural limitations of bus-based approaches, several major computer vendors have worked towards a new standard for I/O connectivity. Future I/O and Next Generation I/O (NGIO) were two competing solutions from different vendors. The need for a unified I/O technology finally led to the formation of the InfiniBand Trade Association[1] (IBTA) in 1999 through a merger of those two forums. Its goal is to provide a unified platform for server-to-server and I/O connectivity, based on a message-based fabric network. In October 2000, the organization released the first version of the InfiniBand specification [61], a three volume set of documents describing the architecture, configuration and use of an InfiniBand fabric.

Companies like IBM [62] and Intel heavily push the development of IB hardware. First products are available now, but mainly used for software development and prototype demonstrations. With its target to replace different technologies it is expected, that IB products first enter the market in one or two main areas and will spread to other application areas over time. First installations might appear in the storage area, where the demand for bandwidth is extremely high. IB will be first deployed in high-end servers for enterprise-class business applications, like databases, transaction systems or webservers. It is then expected that the technology moves down into the PC/workstation mass market and turns into the unified I/O technology it is said to be.

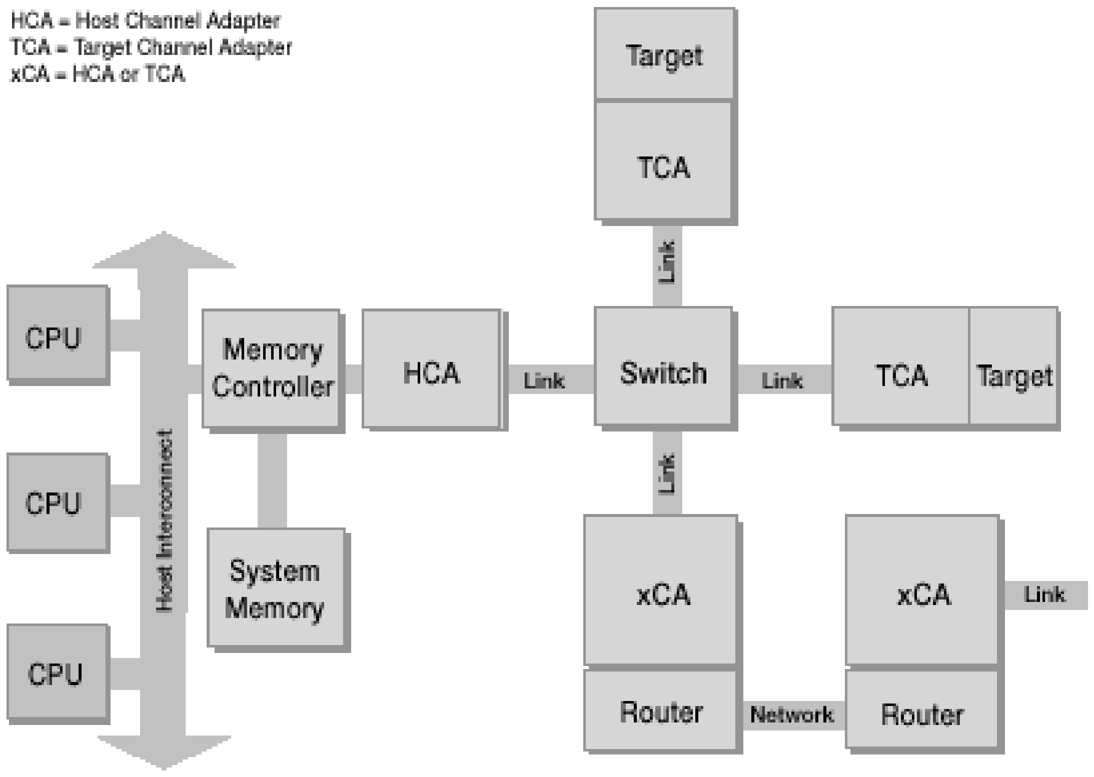
2.11.1 Architecture

Figure 2-18 depicts the general architecture of an IB network. It contains the four building blocks: Host Channel Adapter (HCA), Target Channel Adapter (TCA), Switches and Routers. A HCA is an active network interface, similar to a SAN NIC. It interacts with the host to generate/consume network packets. The HCA provides support for DMA-based data transfer, memory protection and address translation, and multiple concurrent

1. www.infinibandta.org

accesses to the network from several processes. A typical HCA location is inside a server, where it interfaces to the system bus via a memory controller to provide fast access to the IB network. The TCA is a reduced passive version of the HCA, mainly intended to serve as IB interface for I/O devices like disks, graphics, etc. Switches connect local components into a IB subnet, whereas routers connect the subnet to a larger global InfiniBand network.

Figure 2-18. The InfiniBand architecture [61]



The foundation of communication in an InfiniBand fabric is the ability to queue up a set of jobs that hardware executes. This is done via Queue Pairs (QP), one for send and one for receive operations. User applications place work requests in appropriate queues, which are then processed by the hardware. On completion, the hardware acknowledges the finished job via a completion queue.

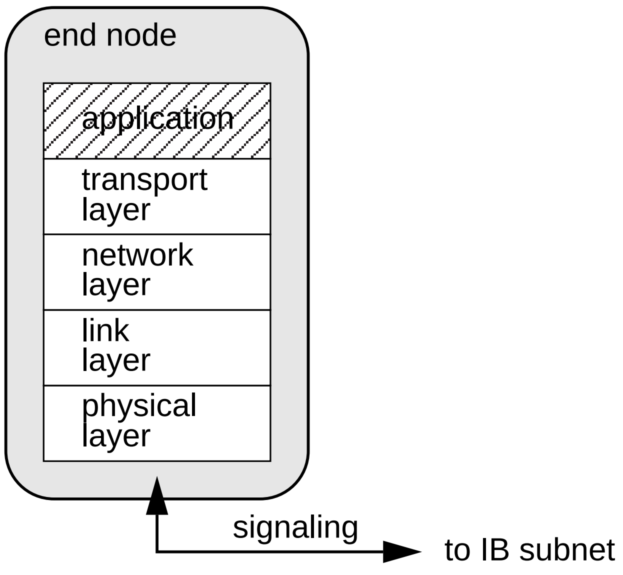
Applications can set up multiple QPs, each one independent from the others. A send job specifies the local data to be sent, and can include the remote address where to place the data. On the receiving side, a job specifies where to place incoming data. Most of the communication mechanisms for IB have been adopted from the Virtual Interface Architecture (VIA), a previous standardization effort for communication within clusters.

InfiniBand supports connection oriented and datagram communication. A connected service establishes a one-to-one relationship between a local and a remote QP. A datagram QP is not tied to a single remote consumer.

2.11.2 Protocol stack

The specification separates IB into several layers, as shown in Figure 2-19: transport, network, link and physical layer. This layered approach helps to hide implementation details between layers, which use a fixed service interface to build on each other.

Figure 2-19. InfiniBand layered architecture [61]



Physical layer

The physical layer specifies how single bits are put on the wire to form symbols. It defines control symbols used for framing (start/end of packet), data symbols and fillers (idles). A protocol defines correct packet formats, e.g. alignment of framing symbols or length of packet sections. The physical layer is responsible for establishing a physical link when possible, informing the link layer whether the link is up or down, monitoring the status of the link, and passing data and control bytes between the link layer and the remote link endpoint.

Link layer

The link layer describes the packet format and protocols for packet operation. This includes flow control and routing of packets within a subnet. It basically defines two types of packets: link management and data packets. Link management packets are exchanged between the two link layers on a connection, and are used to train and

maintain link operation. They negotiate operational parameters such as bit rate, link width, etc. They are also used to convey flow control credits.
Data packets carry a payload of up to 4 Kbyte. A concept called Virtual Lanes (VL) is used to multiplex a single physical link between several logical links. Up to 16 different VLs may be implemented, but only VL0 (data) and VL15 (management) are required.

Figure 2-20. InfiniBand data packet format [61]

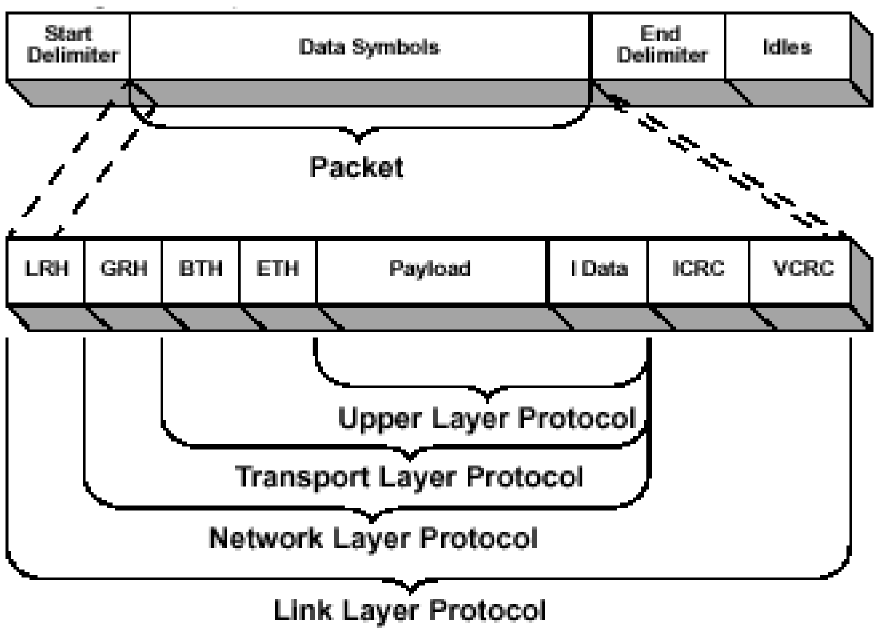


Figure 2-20 depicts the IB data packet format and which layer utilizes which part of the packet to encapsulate needed information. The Local Route Header (LRH) specifies source and destination within a subnet, and the VL to use. The payload of a packet can be 0-4 Kbyte large. Each packet is completed by two CRC words: an Invariant CRC (ICRC) and a Variant CRC (VCRC). The ICRC covers all fields which should not change during a transmission. The VCRC covers all fields. This combination allows switches and routers to modify header fields and still maintain end-to-end data integrity.

Network layer

The network layer implements the protocol to route packets between subnets. It uses the Global Route Header (GRH) to identify source and destination ports across multiple subnets in the format of an IPv6 address. The GRH is interpreted by routers, which may modify the LRH and GRH to forward a packet towards its destination.

Transport layer

The transport layer is responsible to deliver a packet to the proper QP and to instruct the QP on how to process the payload. Segmentation of messages larger than a Maximum Transfer Unit (MTU) is also a task of this layer. It utilizes two fields of the packet header to accomplish its job. The Base Transport Header (BTH) specifies the destination QP, the packet sequence number and a packet type (send, remote DMA, read, atomic). The sequence number is used by reliable connections to detect lost packets. The Extended Transport Header (ETH) gives type-specific information, like remote addresses, total message length, etc.
An Immediate Data (IData) word of 4 byte can be attached to the packet. This word is placed on the receiving side into the completion queue entry, allowing to broadcast single data words without transferring payload into DMA areas.

A Software Transport Interface is defined on how to configure, access and operate IB communication structures. Additional management services are defined to provide an interface for network-wide configuration and administration of IB components.

2.11.3 Remarks

The aggressive goal of InfiniBand to completely take over the whole I/O and server connectivity market is a challenging task. It adopts a lot of mechanisms and technologies from current SANs and tries to apply them to all forms of I/O. Whether it will succeed or fail is also highly dependent on the level of cooperation between the leading industry companies backing IB. The global approach of InfiniBand stands in contrast to possible optimizations for a specific application area, as it is cluster computing. E.g., the header overhead of an IB packet is quite large: up to 106 byte. This means that in applications with a fine-grain communication pattern the protocol overhead consumes a significant portion of the physical bandwidth.

On the other hand, competing solutions can try to present an InfiniBand software interface to applications, while breaking down IB communication structures onto more efficient hardware. At this point, one cannot say if it really becomes the general purpose interconnect, or ends up as a replacement for SCSI and Fiber Channel in the high-end storage market.

3 The ATOLL System Area Network

The idea to develop a new System Area Network was driven by the need for a high performance cluster interconnect that would reduce costs to a minimum and therefore is able to replace Ethernet as the most commonly used cluster network. Cost reduction goes hand in hand with the limited opportunities of a small group of researchers to design and implement such a complex network.

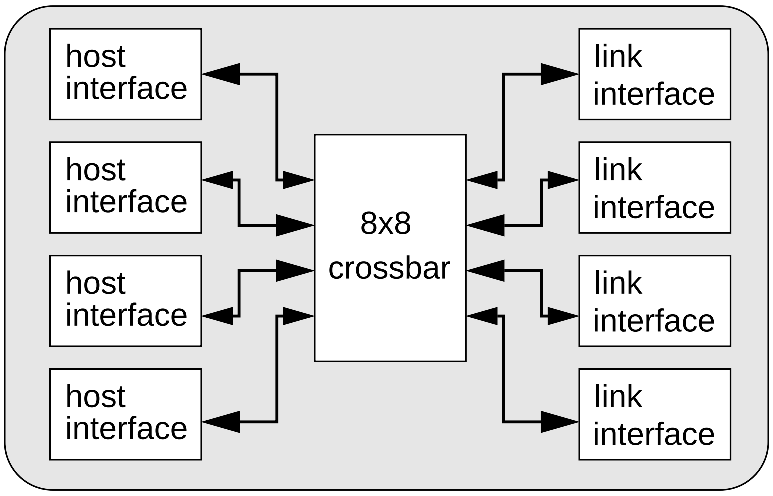
So a large scale integration of all components was a major factor guiding the process of specification and design of a new SAN. This led to the idea to break with the traditional partitioning of a network into a node interface and switches connecting them together. The result is a combined interface/switch device, which serves as a single basic building block for a new generation of SANs.

3.1 A new SAN architecture: ATOLL

The main idea behind this approach was formulated earlier within another context. The first version of ATOLL [63] was designed as a system component for a massive parallel architecture called PowerMANNA [64]. The node design consisted of a quad-CPU board, equipped with PowerPC 620 microprocessors. Besides several memory banks, also an ATOLL chip connects to the system bus to provide a low latency, high bandwidth network connecting all nodes within a system chassis in a 2D grid. The ATOLL chip includes four Bus Master Engines (BME) to give each node CPU exclusive access to the network. Special instructions provide the ability to start communication jobs in an atomic way. This, together with the extreme low latency, gave the design its name: ATOmic Low Latency (ATOLL).

Most internal structures have been redesigned, due to the different environment of the MPP and the SAN version of ATOLL. But the overall 4x4 structure, as shown in Figure 3-1, remained as the main characteristic. Several techniques have been adopted, like wormhole routing and a mechanism for link-level error correction and retransmission of corrupted data.

Figure 3-1. The ATOLL structure



The number of host and link interfaces is a trade-off between the goal to provide as much performance as possible and what is technically feasible. The bottleneck of todays SANs is still the network, considering that advanced 64 bit/66 MHz PCI bus implementations offer a bandwidth of up to 528 Mbyte/s. With PCI-X and its 1 Gbyte/s entering the market, one lets a lot of bandwidth unused. And since dual-CPU, and in the near future quad-CPU nodes become an attractive option as cluster node architecture, one can overcome the overhead associated with multiplexing a single NI.

The upper limit for the number of host interfaces is given by the amount of resources needed to implement them, e.g. control logic, data buffers, etc. The upper limit for the number of link interfaces is given by the amount of pins needed in the IC package for implementing the parallel differential signal lines. The 4x4 structure turned out to be a well balanced system architecture to completely remove the network as communication bottleneck for the first time in Cluster Computing.

3.1.1 Design details of ATOLL

Some of the major design decisions are a natural consequence of the experiences with other solutions. ATOLL is a message passing network interfacing to the dominant I/O technology PCI, or more specific, to its latest upgrade PCI-X. It utilizes source path and wormhole routing on the link level to enable very fast data forwarding. Byte-parallel copper links were the best choice for the interconnect at the start of the project. SANs are moving now towards high speed serial links, but this technology was still premature at that time.

A unique technique to detect and immediately correct transmission errors on links has been implemented to remove costly error checking and correction in software. Since newest cabling technology provides an almost errorfree environment, even for high signal speeds, this mechanism offers the possibility to treat the network as a reliable media.

The mechanisms for data transfer between the host and the NI were driven by the fact that the performance of DMA- or PIO-based approaches is highly dependent on the granularity of the communication. Therefore, the decision was made to support both techniques. This will make it possible to pick the fastest transfer, based on message size and other impacts of the node system. A novel event notification mechanism avoids costly interrupts and enables the CPU to poll on cache-coherent memory.

Some advanced features had to be omitted to keep the complexity at a manageable level. It would have been interesting to implement features like adaptive and multicast routing. As discussed earlier, they can give a huge performance boost, especially for large installations with hundreds of nodes. But the first version of ATOLL targets the market of small to medium clusters, with the number of nodes somewhere between 8 and 256. The current design will be sufficient for these dimensions.

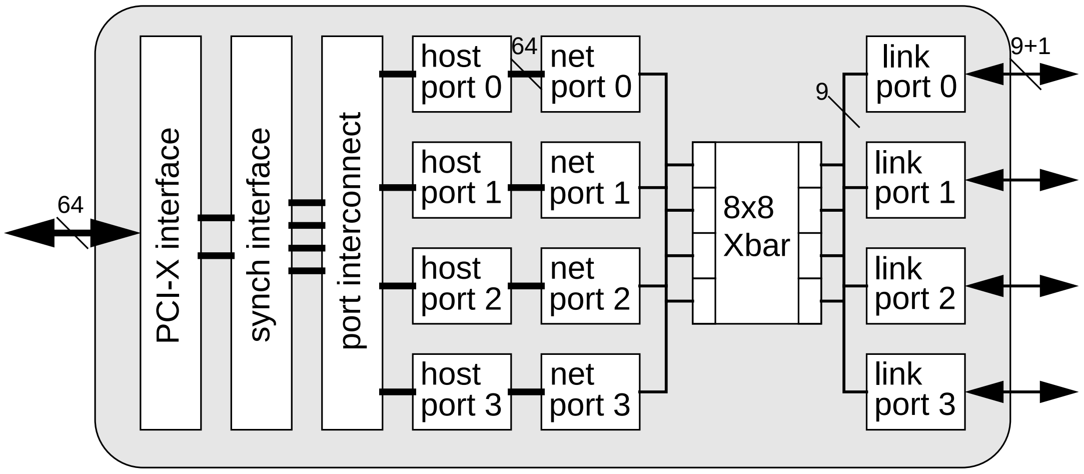
In the following, the major features and mechanisms for the ATOLL System Area Network are summarized:

- best cost-efficient solution by integrating all necessary SAN components into a single IC
- support for SMP nodes by multiple independent host interfaces
- removing the need for external switch hardware by integration of a switch component
- high sustained bandwidth of multiple concurrent data streams by implementing a highly decoupled architecture
- PIO- and DMA-based data transfer to/from host
- efficient control transfer via coherent NIC status information in host memory
- error detection and correction on the link level

The rest of this chapter will introduce the architecture of the ATOLL network chip. The main focus is on the implemented functionality, and how to make use of it via accessing the control/status registers of the device. Each top-level unit will be described separately, its design and its typical use and operation. Regarding the huge complexity of the implementation (over 400 unique modules with about 30.000 lines of code), the level of detail had to be restricted. So not every single state machine or block of control logic is covered in all aspects. But the description of the most important mechanisms and units should enable the reader to gain an in-depth insight into the ATOLL architecture.

3.2 Top-level architecture

Figure 3-2. Top-level ATOLL architecture



ATOLL is a true 64 bit architecture. All addresses used by the device to access data structures in main memory have 64 bit base addresses. However, the actual pointers used to reference the start/end of individual data units are 32 bit offsets. This provides a sufficient amount of continuous memory space for data areas (4 Gbyte), while limiting the expense of internal arithmetic units. When a read/write address is forwarded to the PCI-X interface, the base address is added to the current offset. Figure 3-2 depicts the top-level architecture of ATOLL.

In the following, a brief overview of each functional unit is given:

PCI-X interface

the PCI-X interface is used to communicate with the host system. It can act as bus master or slave, and provides sufficient support for latest improvements to the PCI-X bus protocol, e.g. split transactions.

Synchronization interface

since the core of ATOLL runs with a higher clock frequency than the PCI-X interface, all control and data signals crossing this clock domain border must be synchronized to prevent signal corruption. This is done in a safe manner by the synchronization interface. It also converts the application interface of the PCI-X interface into a highly independent and concurrent interface for all four possible data transfer directions (read/write, master/slave).

Port interconnect

the port interconnect multiplexes the access to the PCI-X interface between all four host ports. Sufficient buffer space is provided to assemble multiple transfer requests. It also contains the global status and configuration register sets for ATOLL.

Host port

a host port contains all logic to enable PIO- and DMA-based message transfer. A small interchangeable context keeps all addresses and offsets needed to access data structures for messages residing in main memory. Large SRAM blocks supply enough buffer space to take up message data and store it for further processing. Multiple concurrent data transfers can be active at a time, e.g. sending data into the network from a FIFO in the DMA unit, while reading data from the receive FIFO of the PIO unit.

Network port

the network port converts a stream of tagged 64 bit datawords from the hostport into a 9 bit-wide data stream conforming with the link packet protocol and vice versa.

Crossbar

the crossbar is a full-duplex 8x8 port switch. It interprets routing bytes of incoming messages to decode the outgoing port, which can be any of the 8 ports.

Link port

the link port provides a full-duplex interface to the network. It prevents buffer overrun by ensuring a reverse flow control scheme. Special retransmission hardware automatically detects corrupted link packets and retransmits them. Enough buffer space is included to support cable lengths of up to 20 m.

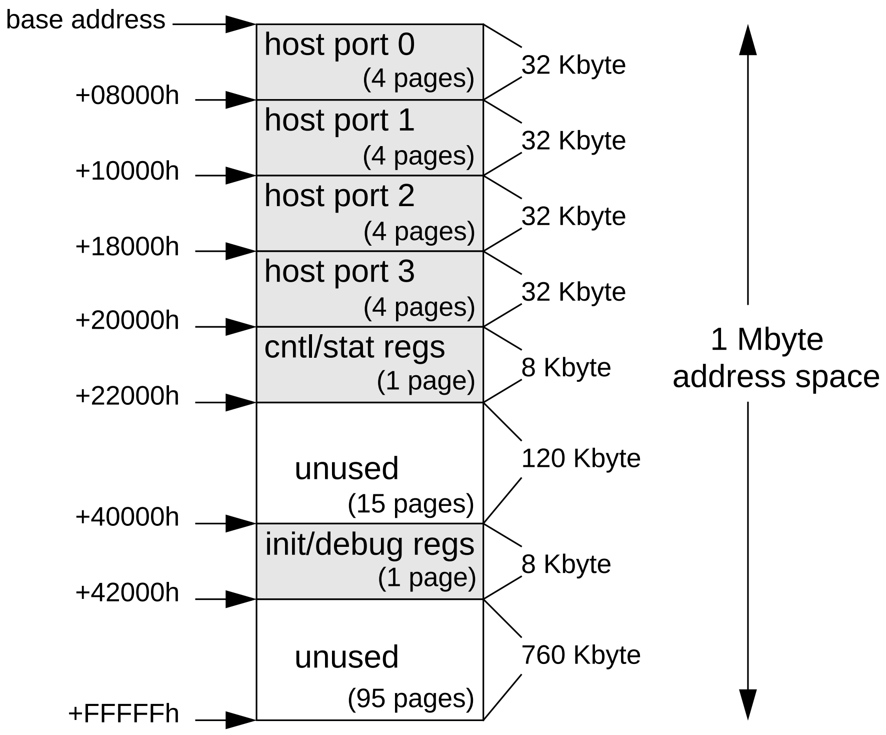
The whole architecture is optimized to provide the highest level of sustained bandwidth and an extreme low latency. All host/network/link port units consist of independent modules for both transfer directions. In contrast to some other SAN interfaces, message data is not temporary stored in large external RAM modules. Rather multiple smaller data

FIFOs are spread all along the data paths from the network to the host. These can be viewed as distributed on-chip data RAM.

3.2.1 Address space layout

The whole ATOLL device requests an PCI-X address space of 1 Mbyte at system start-up. Only the first 260 Kbyte if this address space are currently used. Different parts of the address space are assigned separate memory pages to provide the possibility of using varying memory page control schemes. E.g., some pages could be defined as cachable, but pages containing status registers should not be cached to make sure, each access to them returns valid and up to date data. Since the intention is to support all possible cluster node platforms (x86, Alpha, SPARC, etc.), the decision was made to select 8 Kbyte pages, since the Alpha architecture uses this memory page size, whereas x86 microprocessors use 4 Kbyte. So parted address regions with different memory page attributes can be implemented on both platforms.

Figure 3-3. Address layout of the ATOLL PCI-X device



Another reason for separate pages is the level of protection for different address areas. The user needs access to the registers controlling the PIO- and DMA-based message transfer, but the access to critical control registers should be protected. E.g. a normal user should not be able to alter the frequency of the core clock, or reset parts of the chip.

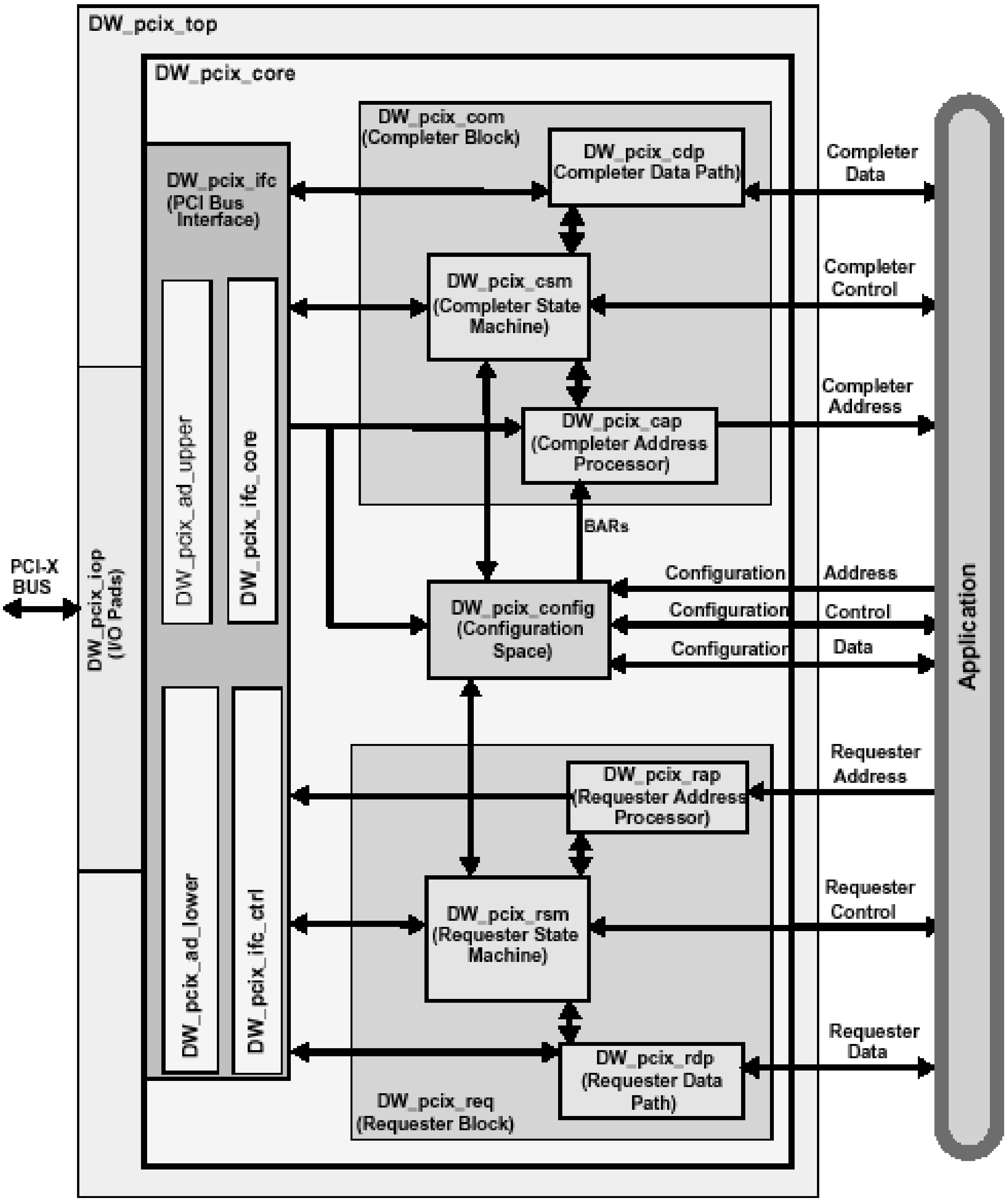
In all 8 Kbyte pages only the lower 4 Kbyte part is used. The upper 4 Kbyte are left unused. And read/write accesses to any unused addresses inside the whole ATOLL address space return the data value 0, respectively consume the written data without any further action. This prevents system failure by erroneous, misaligned accesses to the device. Figure 3-3 shows the address layout of the ATOLL PCI-X device. All addresses given are relative offsets to the base address in hex format.

The page for the initialization and debug registers at address 40000h was appended in a late stage of the design cycle. To ease the insertion of the registers and the decoding of addresses, it was simply placed at an address with a unique address bit (a set addr[18] bit references the init/debug registers). In further versions of the architecture it could be located just after the control/status registers to implement a more compact address layout.

The layout of the different pages is discussed in detail later in this chapter, at the appropriate sections. All four host port address regions have exact the same layout, and are further illustrated in the section about the host port. The control/status registers are located in the port interconnect and are discussed there. Finally, the initialization/debug registers are located in the synchronization interface and are specified in its section.

3.3 PCI-X interface

Figure 3-4. PCI-X interface architecture [65]


DW_pcix_top
DW_pcix_core
DW_pcix_com (Completer Block)
DW_pcix_cdp (Completer Data Path)
DW_pcix_ifc (PCI Bus Interface)
DW_pcix_csm (Completer State Machine)
DW_pcix_cap (Completer Address Processor)
DW_pcix_ad_upper
DW_pcix_ifc_core
DW_pcix_iop (I/O Pads)
PCI-X BUS
BARs
DW_pcix_config (Configuration Space)
Configuration
Configuration
Configuration
Completer Data
Completer Control
Completer Address
Address
Control
Data
Application
DW_pcix_rap (Requester Address Processor)
DW_pcix_rsm (Requester State Machine)
DW_pcix_rdp (Requester Data Path)
DW_pcix_req (Requester Block)
DW_pcix_ad_lower
DW_pcix_ifc_ctrl
Requester Address
Requester Control
Requester Data

The PCI-X bus interface module used in the ATOLL chip is an external IP cell from Synopsys, Inc [65]. Its top-level architecture is depicted in Figure 3-4. It implements a bus interface fully compliant to the PCI-X bus specification[66]. The IP cell is split into four main blocks:

DW_pcix_ifc

the DW_pcix_ifc module contains the PCI bus interface. It performs multiplexing of outgoing addresses and data onto the PCI-X AD bus, and registers all incoming signals. Parity generation and checking is part of this module, as well as detection of the PCI-X bus mode (32/64 bit, 33/66/100/133 MHz).

DW_pcix_com

the DW_pcix_com module implements the Completer logic. It is responsible for all actions necessary when the device acts as a bus slave. Data written onto the device needs to be forwarded to the application. When addressed by a read cycle, the address has to be delivered to the application, which then returns the data.

DW_pcix_req

the DW_pcix_req module contains the Requester logic. It controls all bus master transactions triggered by the application. It automatically retries transfers, if a slave device (like the PCI bus bridge) disconnects in the middle of a data transfer.

DW_pcix_config

the DW_pcix_config module implements the PCI configuration space, as defined by the PCI bus specification. It serves read/write requests to the configuration space.

The complexity of the PCI-X bus interface is quite high, since it implements all the special protocol cases defined in the specification. This complexity is also visible at the interface on the application side. But since the functionality needed by the ATOLL core logic is quite restricted, a lot of features of the PCI-X interface are ignored and disabled. Some examples are:

- the configuration module provides an interface to the application to read/write configuration registers, and to modify/control its behavior. This interface is completely disabled, all output signals of the DW_pcix_config unit are left unconnected, all input signals are set to their inactive value.

- the completer interface provides additional signals to force a disconnect of the current bus transaction, e.g. when the requested data must be fetched from external RAM. But since the ATOLL core can deliver all data within a few cycles, this feature is not needed, the corresponding signals are disabled.

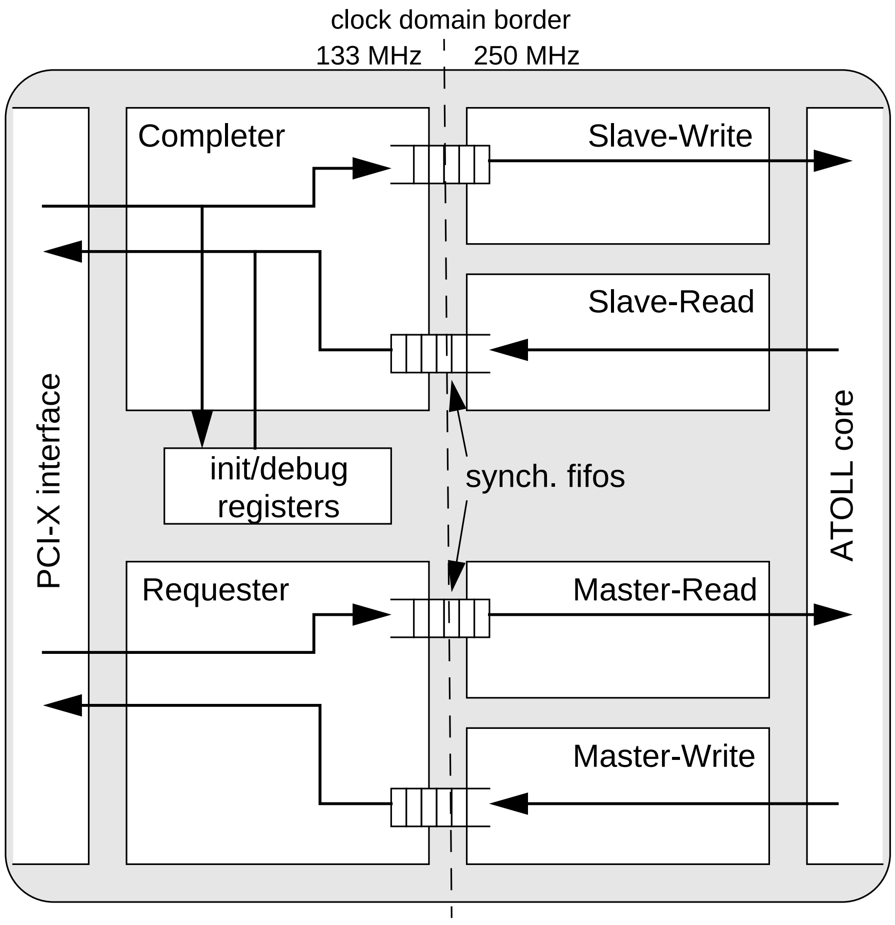
In the following, the main parameters are given, which were defined during generation of an IP cell adapted to the needs of the ATOLL core logic:

- the device can function as PCI-X or PCI device, as defined during system start-up
- the device is capable of acting as 64 bit bus device. It can request 64 bit transactions and react on them. But it can also fall back into a 32 bit-only mode
- the device is capable of running at the highest bus speed defined by the specification, 133 MHz. It also supports lower bus frequencies of 100, 66 and 33 MHz
- a single 64 bit Base Address Register (BAR) is defined, requesting an address space of 1 Mbyte at system start-up. This address space is defined as prefetchable
- the cache line size register is configured to support up to 4 bits, resulting in a maximum cache line size of 16 DWORDS
- no power management functions are implemented
- the registers for minimum grant (MIN_GNT) and maximum latency (MAX_LAT) are both set to 255, their maximum value. This signals the request of the device for long bus burst transfers
- the maximum memory read byte count is set to its highest possible value to support burst transfers of up to 4 Kbyte
- the signal INTA is used by the device to generate interrupts
- split/delayed bus transaction are supported

3.4 Synchronization interface

The synchronization interface connects the PCI-X bus module to the ATOLL core logic. On the PCI-X side, it implements the completer and requester interface defined by the application side of the unit. The completer interface is used for read/write accesses in slave mode, when the ATOLL device is the target of a bus transaction. The requester interface is used for reading data from main memory in master mode, or writing data to memory. On the ATOLL side, these combined read/write interfaces are split into unique, and mostly independent paths. This results in four dedicated interfaces: Slave-Read, Slave-Write, Master-Read and Master-Write. Between those interfaces, in the middle of the synchronization interface, all control and data signals pass a clock boundary via special synchronization elements, which are described in detail later on. Figure 3-5 depicts this structure, visualizing the main data flow direction.

Figure 3-5. Structure of the synchronization interface

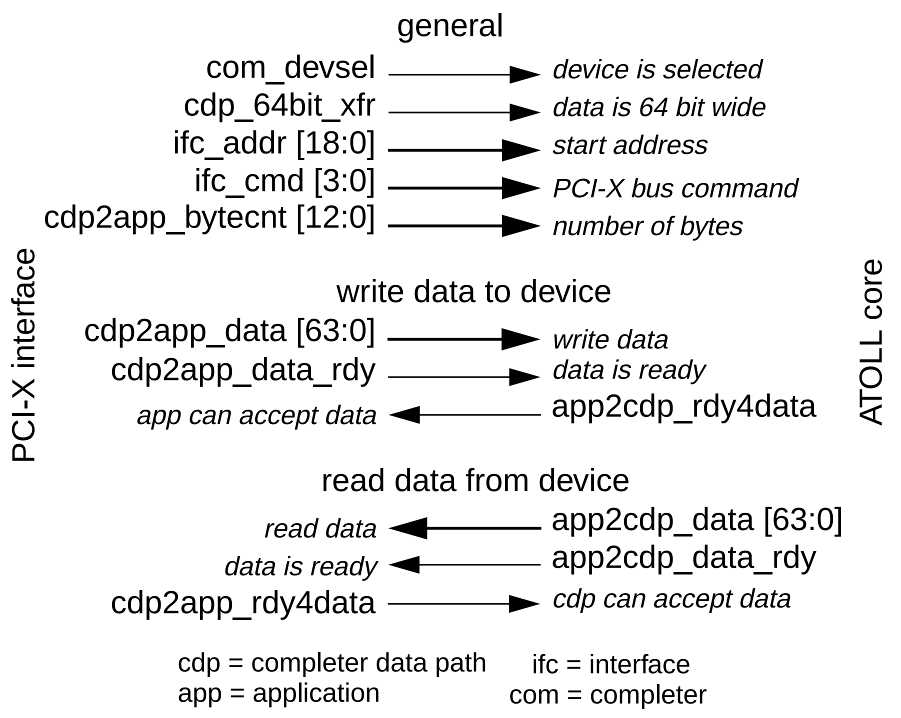


3.4.1 Completer interface

The completer interface needs to separate read and write accesses, and interacts with the Slave-Write and Slave-Read data paths. All signals of the interface used on the PCI-X side are shown in Figure 3-6. The general signals are utilized for both transfer directions. The PCI-X bus specification defines several types of read/write bus commands, but ATOLL only distinguishes between read and write operations. Specifying the number of bytes of a bus transfer is a new feature introduced with PCI-X, so it has no meaning when operating in plain PCI mode. Data is then transferred via a two-way handshaking. The producer simply signals that data is ready, and the consumer signals that it is ready to accept data.

Transfers occur on each clock edge with both signals set. Detailed timing diagrams of the completer interface can be looked up in the databook [65] of the IP cell.

Figure 3-6. Completer interface signals



For write transactions, the completer stores the start address and pushes all incoming data into a synchronization FIFO towards the Slave-Write data path. At the end, the address is also handed over, together with the number of 64 bit words transmitted. This method limits write bursts to a length of 64 words, the depth of the data FIFO. But it saves resources compared to a solution, where each single data word is tagged with its address.

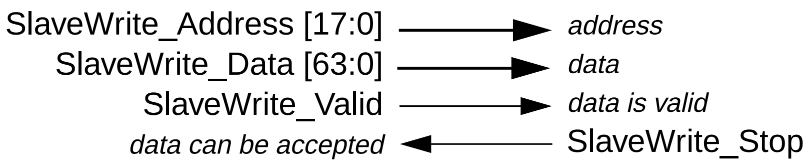
In case of a read request, the address is immediately forwarded to the ATOLL core to request the data. After a few cycles, the ATOLL core delivers the data passing it through another 64 word-depth FIFO stage. There is one side condition regarding the relation of both clocks, when the device is in PCI-X mode. PCI-X forbids wait cycles on the target side, once the target has started to deliver data. So after the first data word is transferred, the target must deliver data on each successive clock cycle.

Since the Slave-Read path on the ATOLL side delivers a data word on each second cycle, the ratio of PCI-X to ATOLL clock frequency must be at least 1:2. So if the PCI-X interface runs at its highest frequency of 133 MHz, the ATOLL clock must be at least 266 MHz. This restriction is a remnant of an earlier version of the design. A future version could simply implement the ATOLL side in a way, so it delivers data on each clock cycle. In case of running plain PCI mode, the end of the transfer is signaled via a deasserted device select signal. The completer then deasserts the valid signal for the address, so the ATOLL side knows it can stop delivering data. All prefetched data still in the data FIFO is then discarded by flushing the FIFO.

For both transfer directions the completer interface is also responsible to convert a 64 bit data stream to a 32 bit one, if the PCI-X bus is only capable of running 32 bit transactions.

3.4.2 Slave-Write data path

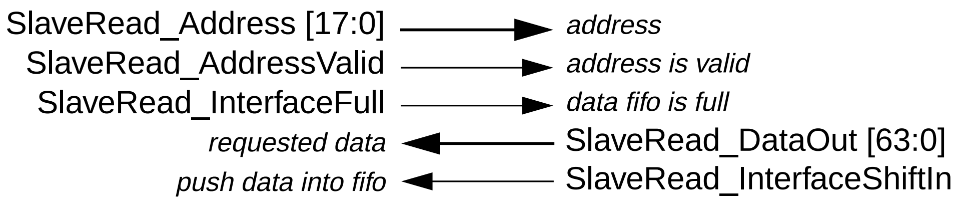
Figure 3-7. Slave-Write path signals



On the ATOLL side, the Slave-Write path is plain simple. As soon as an address-length pair is handed over from the completer, this unit forwards each data word from the FIFO to the ATOLL core, together with its address. Data is transferred again via a two-way handshaking, the signals are shown in Figure 3-7. On every clock edge, data is latched if it is valid and the consumer side can accept the data. This valid/stop scheme is used throughout the whole architecture to transfer data between adjacent units.

3.4.3 Slave-Read data path

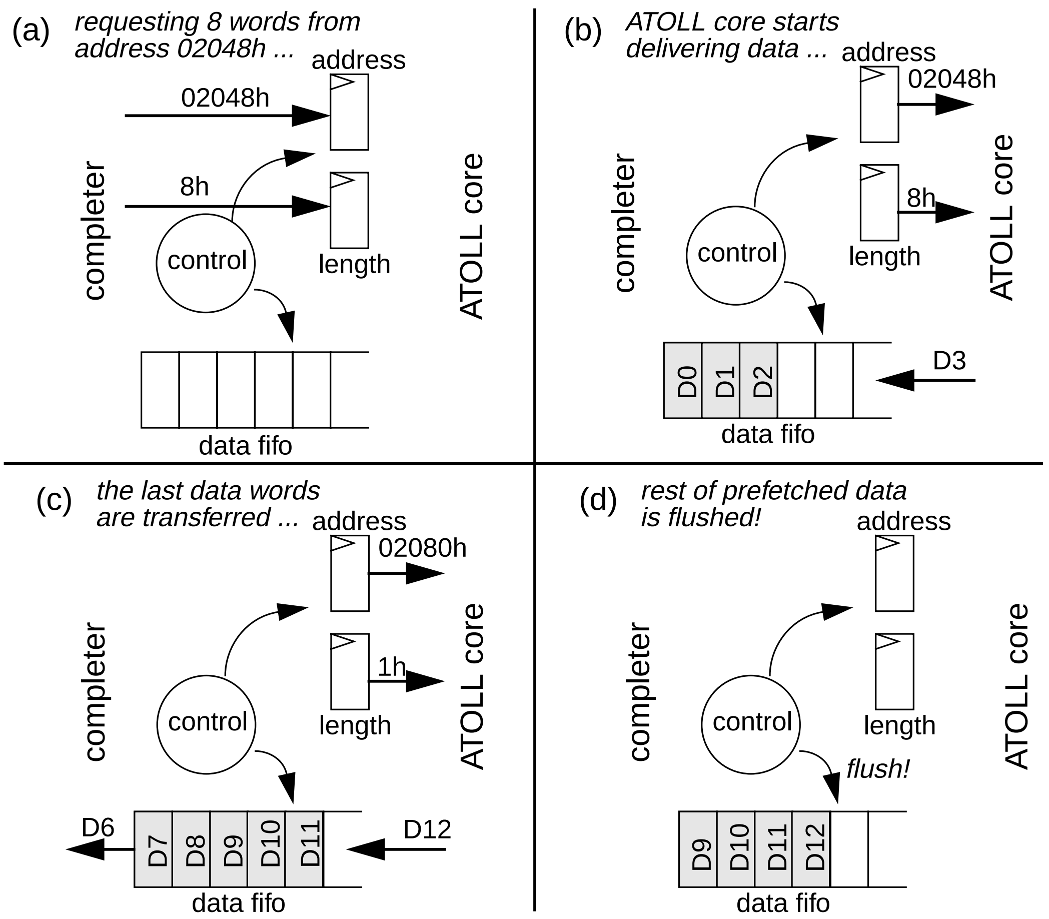
Figure 3-8. Slave-Read path signals



The Completer delivers the address, a valid signal and the byte count for a read access. In case of running in plain PCI mode, the PCI bus does not use a byte count, but instead signals the end of a transfer via deasserting a signal prior to the last cycle. Since the only burst read access to ATOLL could be the reading of message data from the data FIFOs of the PIO-mode, a prefetching mechanism is used to not let a request fail due to missing data. So in plain PCI mode, the completer sets the byte count to the maximum possible value. The Slave-Read path now starts to fetch data from the ATOLL core as long as the address is signaled as invalid, or the byte count is satisfied.

Figure 3-8 depicts the interface signals to the ATOLL core. In case the core delivers data too fast, e.g. when the PCI-X interface runs only at 33/66 MHz, then a full data FIFO is signaled to prevent buffer overflow. When running with 100/133 MHz, data is transferred every second clock cycle. Below that, the interface is slowed down to transfer data only every fourth cycle. This mechanism was introduced to prevent that lots of prefetched data assembles in the data FIFO, which needed to be shifted out one by one in an early version of the implementation after the transaction ended (the FIFO simply lacked a flush signal). Later on, a custom version of the data FIFO was developed with such a flush signal to render this mechanism unnecessary. It could be simply removed in a later version of the implementation. Figure 3-9 visualizes a typical Slave-Read transfer.

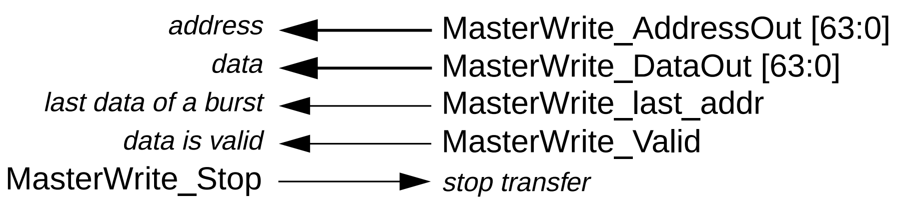
Figure 3-9. Typical Slave-Read transfer



3.4.4 Master-Write data path

When acting as bus master, the ATOLL device tries to transfer data in bursts as large as possible to use the bus as efficiently as possible. When the ATOLL core writes out data into main memory, each data word is transferred together with its destination address. An additional signal marks the last data word of a burst. Data is handed over via the normal two-way handshake. Figure 3-10 displays all interface signals.

Figure 3-10. Master-Write path signals



The control logic of the Master-Write data path pushes incoming data into a FIFO. It stores the first address and only counts incoming words as long as they form a continuous stream of addresses. At the end of a continuous data block, it hands over address and word count to the requester unit, which then passes the data burst on to the PCI-X interface. There are four conditions marking the end of a burst:

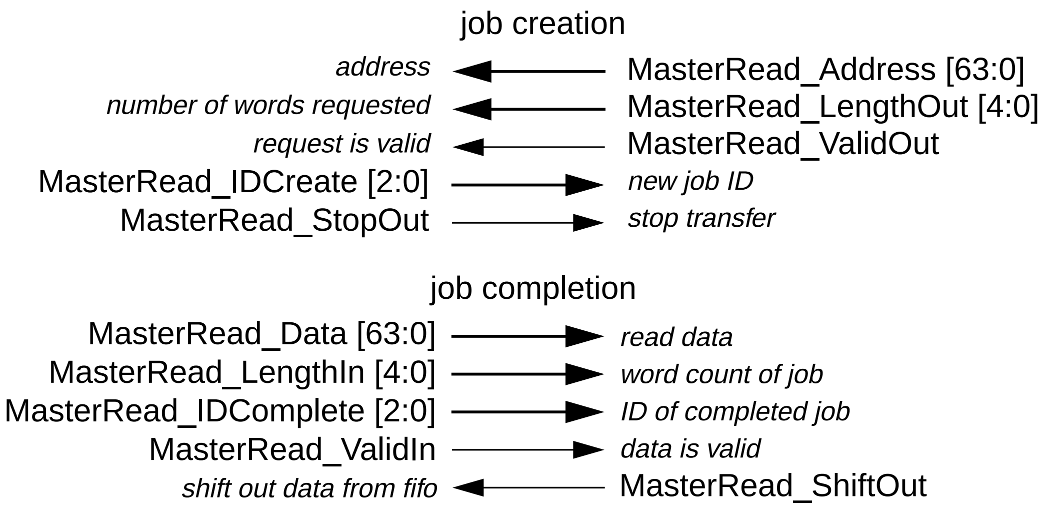
- the address of a data word does not match the previous ones. This might happen if multiple data streams from different host ports are mixed
- the ATOLL signals the last data word of a burst
- the data FIFO is full, so the maximum size for a single burst is reached (currently 64 words)
- or no data has been transferred for a certain amount of time (currently 128 cycles)

3.4.5 Master-Read data path

Reading data from main memory requires more complex logic. To implement an efficient utilization of the PCI-X interface, requests to read data are handled in a split-phase manner. The ATOLL core issues a request to load a certain number of data words from a specific address. Up to 32 words can be requested with a single job. The Master-Read path tries to combine successive jobs into one large burst. It then instructs the requester unit to fetch data from the PCI-X bus and forwards the data to the ATOLL core.

Since multiple host ports might request data, each job is assigned one of 8 job IDs to be able to associate returned data with the job requesting it. So up to 8 jobs might be outstanding, each with up to 32 words. That way, the PCI-X interface is kept busy and only idles, if no requests for data are active at all. Figure 3-11 shows the interface signals, grouped into signals used for job creation or completion.

Figure 3-11. Master-Read path signals



The control logic of the Master-Read path keeps outstanding jobs in an internal queue. Incoming data is then tagged with the correct ID and word count before handing it over to the ATOLL core. Job requests are served strictly in order to ensure a fair service.

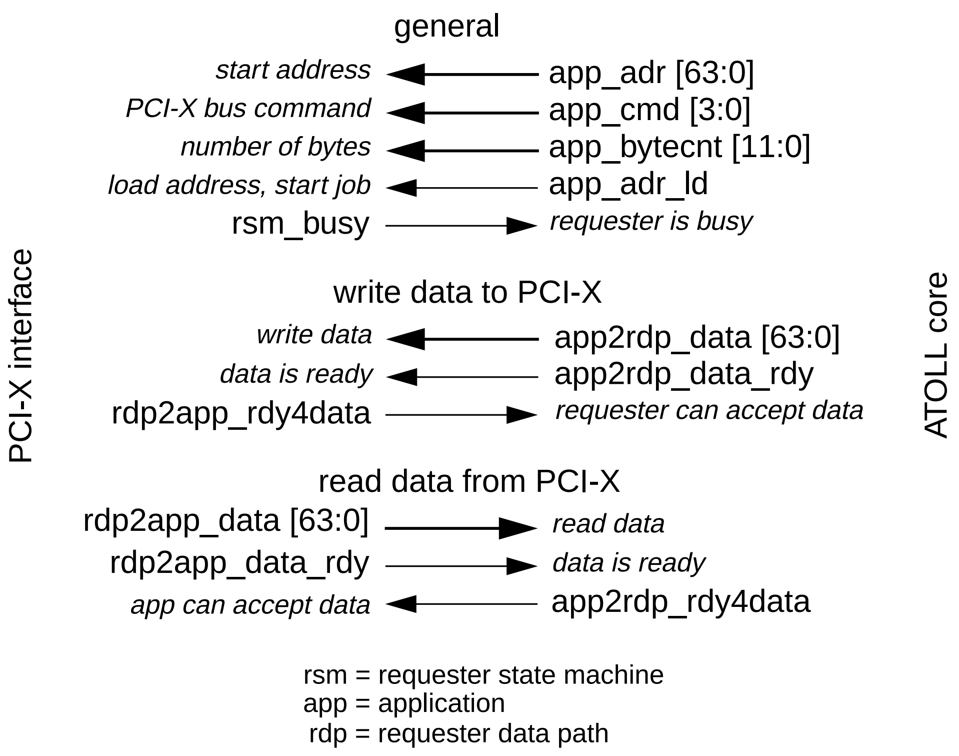
3.4.6 Requester interface

The requester unit needs to multiplex the access to the PCI-X interface between the Master-Write and the Master-Read data paths. Requests for work is scheduled in a fair round-robin fashion between them to prevent starvation of one of the paths. Both job scenarios follow the same procedure, which is briefly described below:

- if a request for a job is handed over by one of the two paths, the requester loads the start address and the byte count into the PCI-X interface
- it signals the type of bus command to the interface. These are `Memory Read/ Write Block` in case of PCI-X, and `Memory Read/Write` in plain PCI mode
- data is then transferred via the usual two-way handshaking
- after delivering the last data word, the PCI-X interface is released and the requester looks for new job requests

In contrast to the completer interface, the conversion of 64 bit data into 32 bit data is performed automatically by the PCI-X interface. An interruption of the bus transaction is also managed by the interface, e.g. when a bus target disconnects in the middle of a transfer. All these special cases are served and managed by the interface, greatly simplifying the requester logic. Figure 3-12 lists all interface signals of the Requester part.

Figure 3-12. Requester interface signals

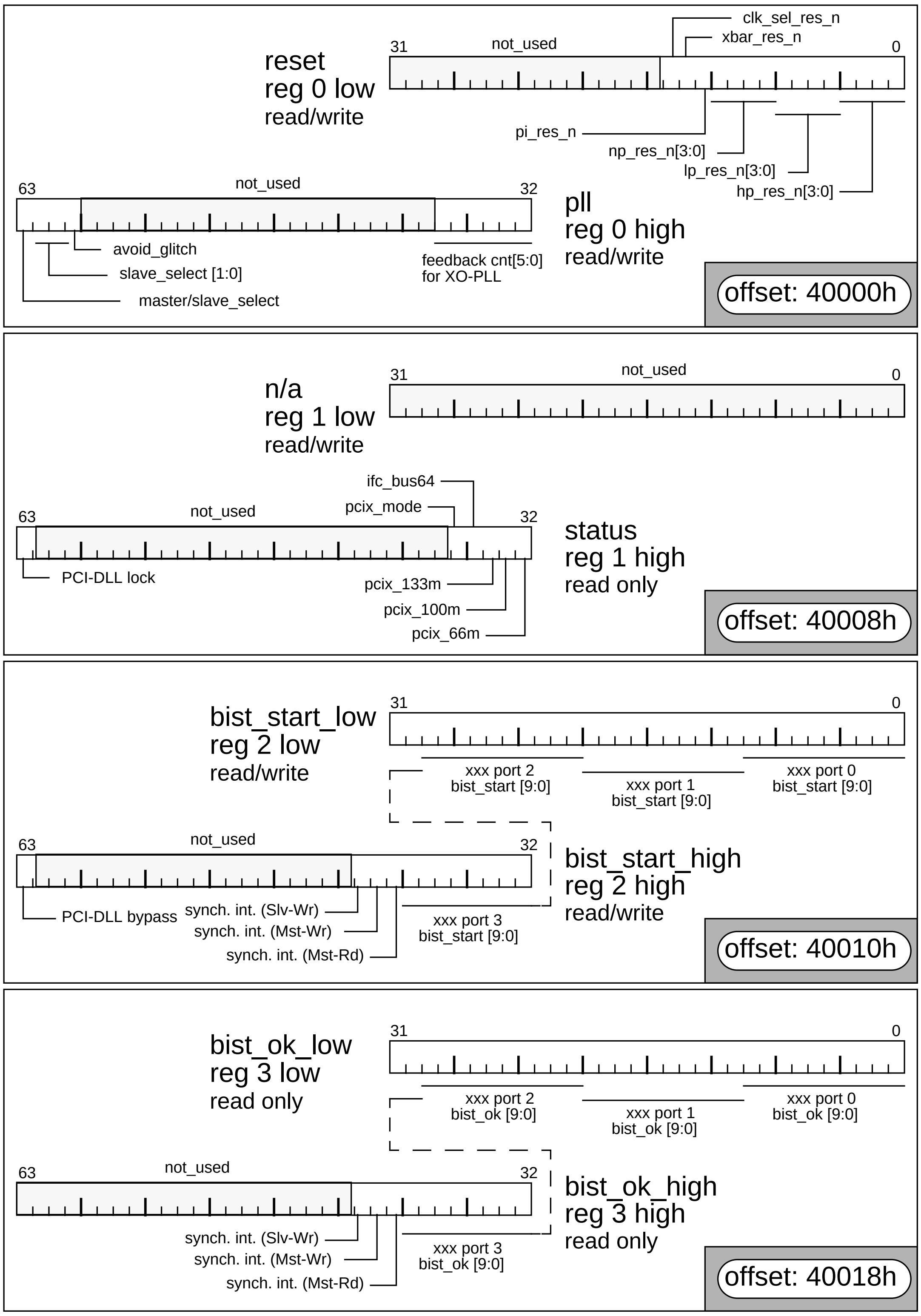


3.4.7 Device initialization

A few initialization and debug registers have been placed into the synchronization interface. The registers reside on a separate page, because some critical functions are provided, which should only be accessible from a privileged user/administrator. They have been moved out of the ATOLL core, since they cannot assume that a stable clock is running in the core. Also the configuration of the ATOLL core clock is located here. Only four 64 bit registers are allocated in total, split up into 32 bit low/high parts.

Figure 3-13 gives an overview about each register and the meaning of its individual bits. Not all bits are used, these are grayed out. In addition, some registers are read only, ignoring write accesses to them. The given address offset is relative to the base address of the whole ATOLL device.

Figure 3-13. Device initialization and debug registers



Reset register

besides the global power-on reset signal, each major block of ATOLL has been assigned a unique reset signal. This offers the possibility to reset a specific part of the device without reinitializing the whole chip. The abbreviations stand for: host port (`hp`), network port (`np`), link port (`lp`), port interconnect (`pi`), crossbar (`xbar`) and clock logic (`clk_sel`).

PLL register

the ATOLL clock logic includes a configurable Phase Locked Loop (PLL) to set the internal clock to anywhere within 175-350 MHz, in steps of 14 MHz. The 6 bit feedback counter configures the clock frequency. Besides running a chip with its own clock, there is the possibility to take one of the four incoming clocks from the links as main clock signal. Which one is selected via `slave_select [1:0]`. The signal `master/slave_select` is then used to switch between the master and the selected slave (link) clock. Another signal can be set to make sure this transition is made without glitches on the clock signal.

Status register

The lower 5 bits sample some status bits set by the PCI-X interface. It detects at system start-up, if it runs in PCI or PCI-X mode, whether it sits on a 64 bit bus, and which PCI bus frequency is configured. The signal `PCI-DLL lock` is set, when the on-chip Digital Locked Loop (DLL) has adjusted the PCI clock to a fixed clock tree delay, as specified in the PCI-X specification.

BIST start register

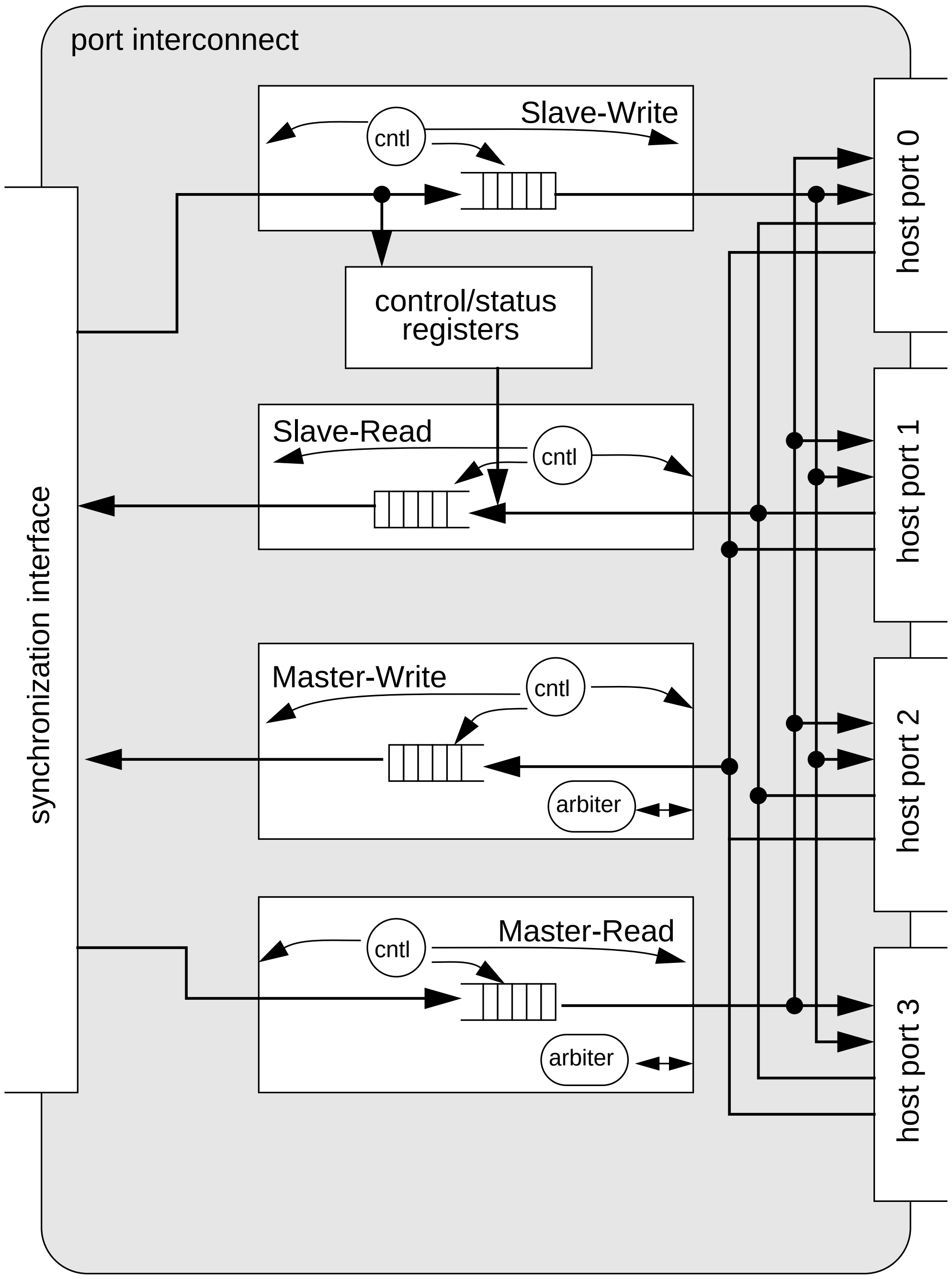
All SRAM memory cells have a Built-In Self Test (BIST) logic attached. It checks the memory for erroneous bits by successive reads/writes of certain bit patterns. This is done normally via the JTAG test logic, which is used to check chips for manufacturing faults. But these registers provide an additional way to run the BIST tests via software. Setting the start bits triggers the internal control engine of the BIST logic to run through all memory cells. Each bit controls one of the 43 SRAM cells in the ATOLL chip.
All cells within a 10 bit word of `xxx port 0/1/2/3` are in the same order ([0:9]): hp_pio_snd, hp_pio_rcv, hp_dma_snd, hp_dma_rcv, np_snd, np_rcv_buf0, np_rcv_buf1, lp_in, lp_out_buf0, lp_out_buf1.
Three bits are used to control the three SRAMs in the synchronization interface. An additional bit is used to bypass the PCI-DLL.

BIST ok register

These bits are the corresponding signals to the BIST start bits. Due to different sizes of the SRAMs some checks are faster than others. But after 10 us all SRAMs are tested, and all 43 bits should be set. Any bits remaining unset flag a flawed SRAM.

3.5 Port interconnect

Figure 3-14. Structure of the port interconnect



The port interconnect multiplexes the four ATOLL core data paths described in “Synchronization interface” on page 60 between all four host ports: Slave-Write, Slave-Read, Master-Write and Master-Read. It temporary buffers data in FIFOs to ease the implementation as an IC. The data nets would otherwise travel a long distance across the chip from the host ports to the synchronization interface, making the task of a physical implementation extremely difficult. And some additional pipeline stages on each path give the possibility to assemble a few data words in this module, even if the path is blocked further ahead.

The implementation of both Slave paths is straight forward. An incoming address from the synchronization interface is decoded to find out the target host port. This can be done by analyzing only 2 bits of the address, according to “Address space layout” on page 56. In case of the Slave-Write path, the addressed host port then pops the data from the FIFO. If data should be read from one of the host ports, the addressed host port starts delivering data, until the Slave-Read path deasserts the valid signal for the address, thus ending the transfer. Both data paths also handle requests for the control/status registers.

In case of both Master paths, the access to them must be multiplexed between all host ports. Each path contains an arbiter, which observes request signals from the host ports. It grants the data path to only one host port, and releases the grant, if the request signal is deasserted. This is forced by a host port itself after a specific amount of cycles to prevent one host port blocking the access to the data path. Arbitration is done based on a fair round-robin schedule policy. The Master-Write path is multiplexed between 8 requesters, since two individual units inside each host port need to write data out to memory. The main unit is the one receiving messages in DMA mode, spooling data out to buffer regions residing in main memory. The other unit mirrors relevant status information into memory, removing the need to poll the device for it. These status packets consist of only four 64 bit data words, thus the impact on the overall data throughput is relative small.

3.5.1 ATOLL control and status registers

A separate module[1] hosts the control/status registers needed for managing the use of ATOLL. These are registers configuring the global state of the device, as well as host port specific registers. Since they control important settings of the device, these registers should not be visible nor accessible to the normal user. They are normally controlled by a supervisor, mostly via an administration interface. These registers provide the following functionality:

1. designed and implemented by Prof. Dr. Ulrich Brüning

Port interconnect

- link driver control
- start/stop of DMA engines
- interrupt generation (mask & clearance)
- global counter
- debug control of the crossbar
- additional status information

All registers can be split into four categories: control, status, debug and extension. The registers are aligned to cache line boundaries of eight 64 bit words. This is to prevent the CPU from fetching more than one register with each access, possibly loading several registers into prefetch buffers in the system's chipset. Though all registers are free of side effects, this decision was made to ensure that a user always gets the up to date status of a register.

Table 3-1. Control registers of ATOLL

register name	offset	mode, width	comment
hw_cntl	20000h	r/w, 32	hardware control, link enable, DMA engines
cnt_load	20040h	r/w, 64	global counter load
hp_cntl	20080h	r/w, 64	host port specific control
hp_dma_thres	200C0h	r/w, 32	threshold value to determine receive mode (DMA/PIO)
irq_timeout	20100h	r/w, 32	IRQ time-out value
irq_mask	20140h	r/w, 32	IRQ mask setting
irq_clear	20180h	r/w, 32	IRQ clearance
hp_timeout	201C0h	r/w, 32	host port time-out value for PIO mode

In Table 3-1, all control registers are listed with their address offset relative to the device base address, their mode (read/write, read-only) and whether they use 32 or 64 bit. The status registers make various status information visible to the user of the device. They are listed in Table 3-2 in the same manner as in the previous table.

The internal crossbar offers the possibility to observe its status and to insert/pull out link data in case of severe problems. Table 3-3 lists the eight registers, their use is described in detail in the ATOLL Hardware Reference Manual [67].

Table 3-2. Status registers of ATOLL

register name	offset	mode, width	comment
hw_state	20200h	ro, 32	hardware state, clock configuration, DMA engines
cnt	20240h	ro, 64	global counter value
hp_state	20280h	ro, 32	host port specific status, last mode of access
lp_retry	202C0h	ro, 32	link port retry counter
hp_irq_case	20340h	ro, 32	host port IRQ cases, out of buffer space
irq_case	20380h	ro, 32	global IRQ cases, link error, clock failure

Table 3-3. Debug registers of ATOLL

register name	offset	mode, width	comment
debug_xbar0	20400h	r/w, 64	debug info from crossbar port 0
debug_xbar1	20440h	r/w, 64	debug info from crossbar port 1
debug_xbar2	20480h	r/w, 64	debug info from crossbar port 2
debug_xbar3	204C0h	r/w, 64	debug info from crossbar port 3
debug_xbar4	20500h	r/w, 64	debug info from crossbar port 4
debug_xbar5	20540h	r/w, 64	debug info from crossbar port 5
debug_xbar6	20580h	r/w, 64	debug info from crossbar port 6
debug_xbar7	205C0h	r/w, 64	debug info from crossbar port 7

Finally, two more registers are used as extension to the above set of registers to control the debugging of the crossbar and to read/write data on 8 general purpose pins of the chip, as listed in Table 3-4.

Table 3-4. Extension registers of ATOLL

register name	offset	mode, width	comment
debug_cntl	20600h	r/w, 32	control of crossbar debug
gp_io	20640h	r/w, 32	8 general purpose I/O pins

In the following, all registers are described in detail. They are listed in groups of functional classes, rather than in the strict sequence given in the tables.

Figure 3-15. Hardware control/status and global counter

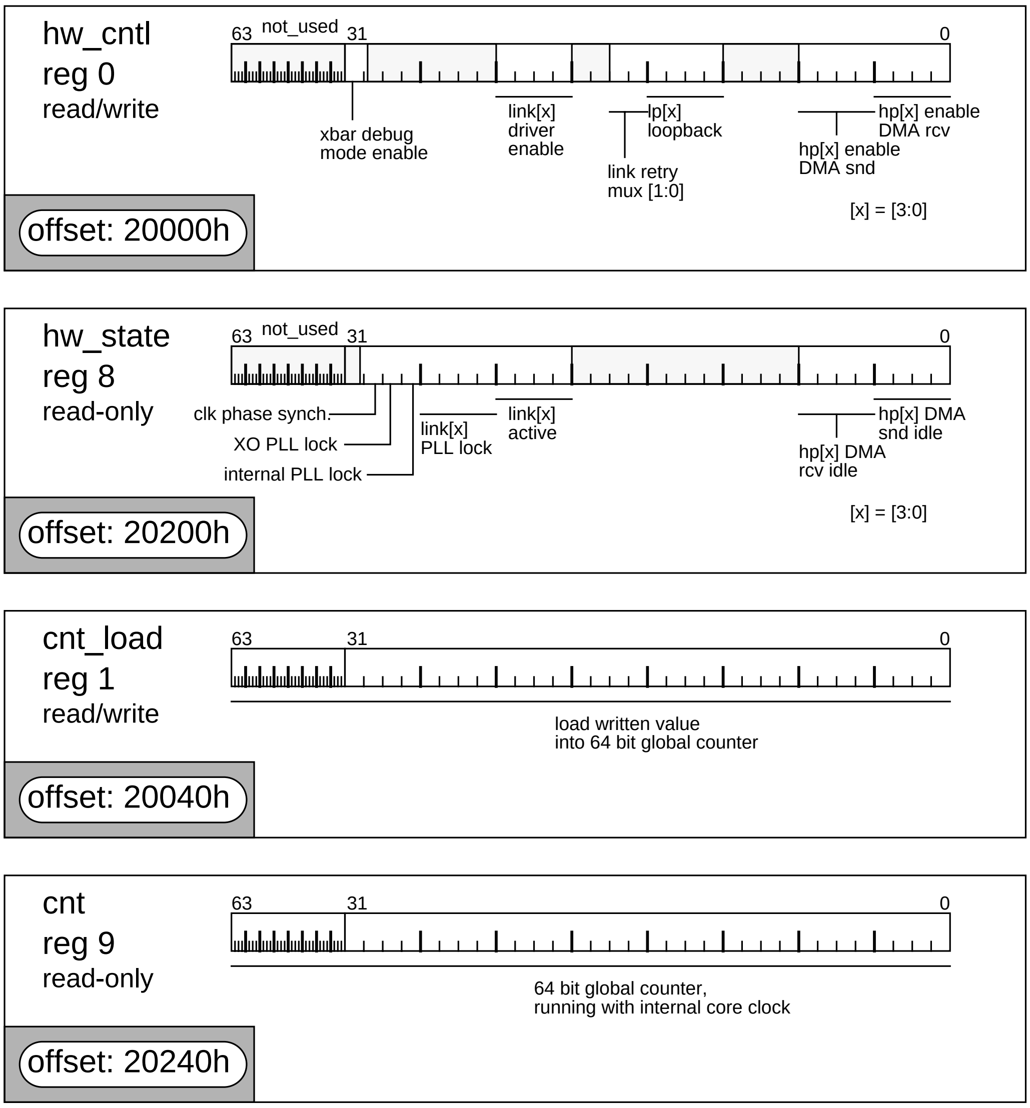


Figure 3-15 depicts the hardware control/status register, as well as the global counter registers. The latter is simply an internal counter, which is running with the internal ATOLL core clock. A write access to the cnt_load register loads the written value into the counter. The hardware control/status registers assemble various configuration bits, which are now described in more detail:

- hw_cntl[7:0] enables the DMA engines inside the four host ports. The lower four bits control the DMA engines in the receive unit, whereas the upper four bits control the send DMA units. These bits are helpful, if the context of the host ports should be switched, e.g. in case the device is multiplexed between several user processes

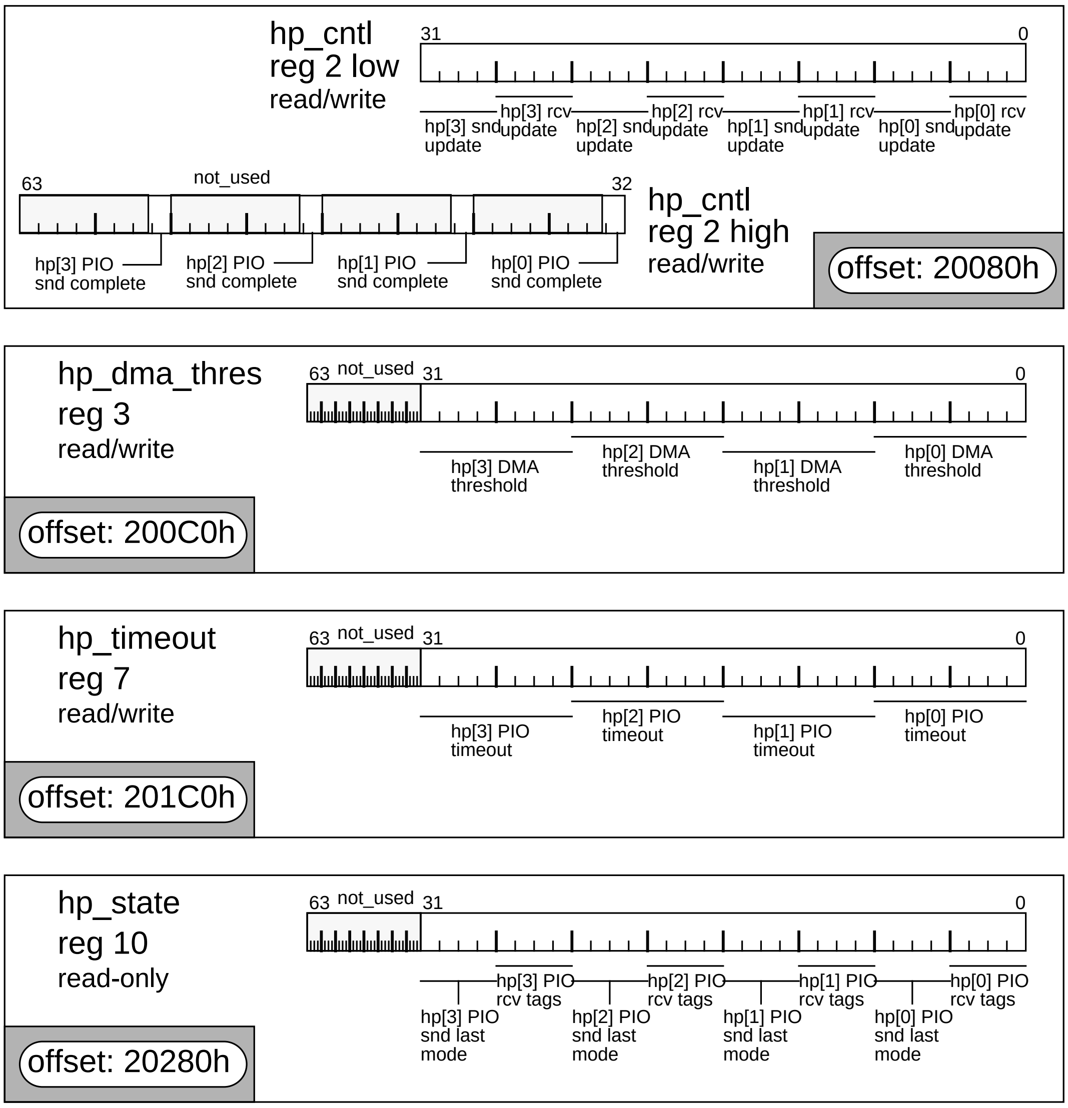
- hw_cntl[15:12] controls a loopback mode of the link ports. Message data normally sent over the link is then fed back into the device via the input path of the link port. This is helpful to isolate cable failure from internal chip problems
- hw_cntl[17:16] controls the generation of an interrupt based on link bit errors. Bit errors force a link packet retransmission by the link port. This event is signaled by the link port, and accumulated in internal 4 bit counters. link_retry mux[1:0] determines the significant bit of the accumulator, that triggers an interrupt. So one can configure it in a way that an interrupt is generated after 1, 2, 4 or 8 link packet retransmissions
- hw_cntl[23:20] enables the LVDS driver I/O cells, so link data is driven over the cable
- hw_cntl[31] is used to activate a special mode for the crossbar debug registers
- hw_state[7:0] signals idling DMA engines in the host ports. This is useful to wait for completion of DMA jobs, after the DMA engines have been disabled. By accident, the bits for send/receive have been switched in relation to the hw_cntl bits
- hw_state[23:20] are set, if the corresponding link is active
- hw_state[27:24] are set, if the corresponding link PLL is locked
- hw_state[30:28] are set, if the two PLLs for the main ATOLL core clock are locked. The highest bit is set, if the phases of two clock signals are synchronizes, when the main clock should be switched from the on-chip clock signal to a link clock

Figure 3-16 shows the registers to control the host ports. Their meaning is as follows:

- hp_cntl[3:0] determines the update frequency of status information written out by the PIO receive unit inside a host port. After n words have been received from the network and pushed into the 64 word data FIFO, the unit triggers an update of the mirrored FIFO fill level in main memory. To span the whole range of the FIFO, the 4 bit value is multiplied by 4, so updates can occur after 4, 8, 12, 16, ..., 60 words. If set to 0, this mechanism is disabled. Setting this value is a trade-off between the need for up to date status information and limiting the bandwidth used for updates. The last word of a message frame (header or data) always triggers the update, so one is informed, if enough or all remaining data is available
- hp_cntl[7:4] does the same job, but for the PIO send module. If the fill level of the FIFO varies by the threshold value, an update is triggered. Also writing the last word of the data frame triggers an update

- hp_cntl[31:8] have the same meaning like hp_cntl[7:0], but for the other host ports

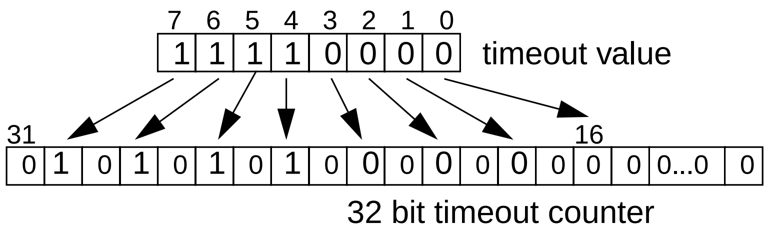
Figure 3-16. Host port specific control/status registers



- hp_cntl[56, 48, 40, 32] can be used to disable the immediate forwarding of data written into the PIO send data FIFO towards the network. If set, the PIO send unit waits, until all data of the message is written to the FIFO. This prevents the insertion of incomplete messages into the network in case the user process is interrupted (normal process scheduling, interrupt service) or simply crashes due to an error in the program code

- the 8 bit values specified in the hp_dma_thres register are used to determine the mode used to receive a message. This threshold value is compared to the length of an incoming message in the receive part of a host port, or more specific, to the length of the data frame. It is forwarded to the DMA-receive unit, if the length is greater or equal to the threshold value. So setting it to 0 pipes all messages to the DMA-receive unit, disabling the PIO mode for receiving messages. Since length and threshold are specified in terms of 64 bit data words, the largest message to be received via the PIO mode is 254*8=2032 byte large, if the threshold is set to FFh

Figure 3-17. Loading the PIO-receive time-out counter



- the hp_timeout register specifies an upper limit for the time data has been pushed into the data FIFO in the PIO-receive unit without reading it. This mechanism shall prevent data to clog in the host port and to block a path backwards into the network. An internal 32 bit counter is loaded with the 8 bit time-out value each time a data word is shifted in or out of the FIFO. To span a large time segment, each second bit of the upper 16 bit is loaded with a bit from the time-out value, as shown in Figure 3-17. This makes it possible to configure a time-out of up to 5.7 s, with steps of 262 us, assuming a 4 ns clock cycle. Each cycle nothing happens, the counter is decremented. If it runs down to 0, an interrupt is generated to force the host to pull the data out of the host port
- the hp_state register provides some information about the status of the PIO unit. In case of an interrupted or incomplete PIO-based message transfer, one must be able to recover the situation, avoiding the need to reset the complete device. So driver software needs to be able to detect, in which state the PIO unit was left by a user process. In case of the PIO-Send module, this register shows the mode of the last access to the unit. More specific, it shows the tags of the last data word written to the FIFO. According to the tags, driver software is able to complete the message by writing the missing data to the FIFO. In case of the PIO-Receive unit, one simply sees the tags of the data word at the head of the FIFO. So software is able to pull out the rest of a message one by one. The 4 bit tags are {Last, Data, Header, Route}, according to the three frames a message is composed of, and an additional bit to mark the last word of a frame

Figure 3-18. Interrupt registers

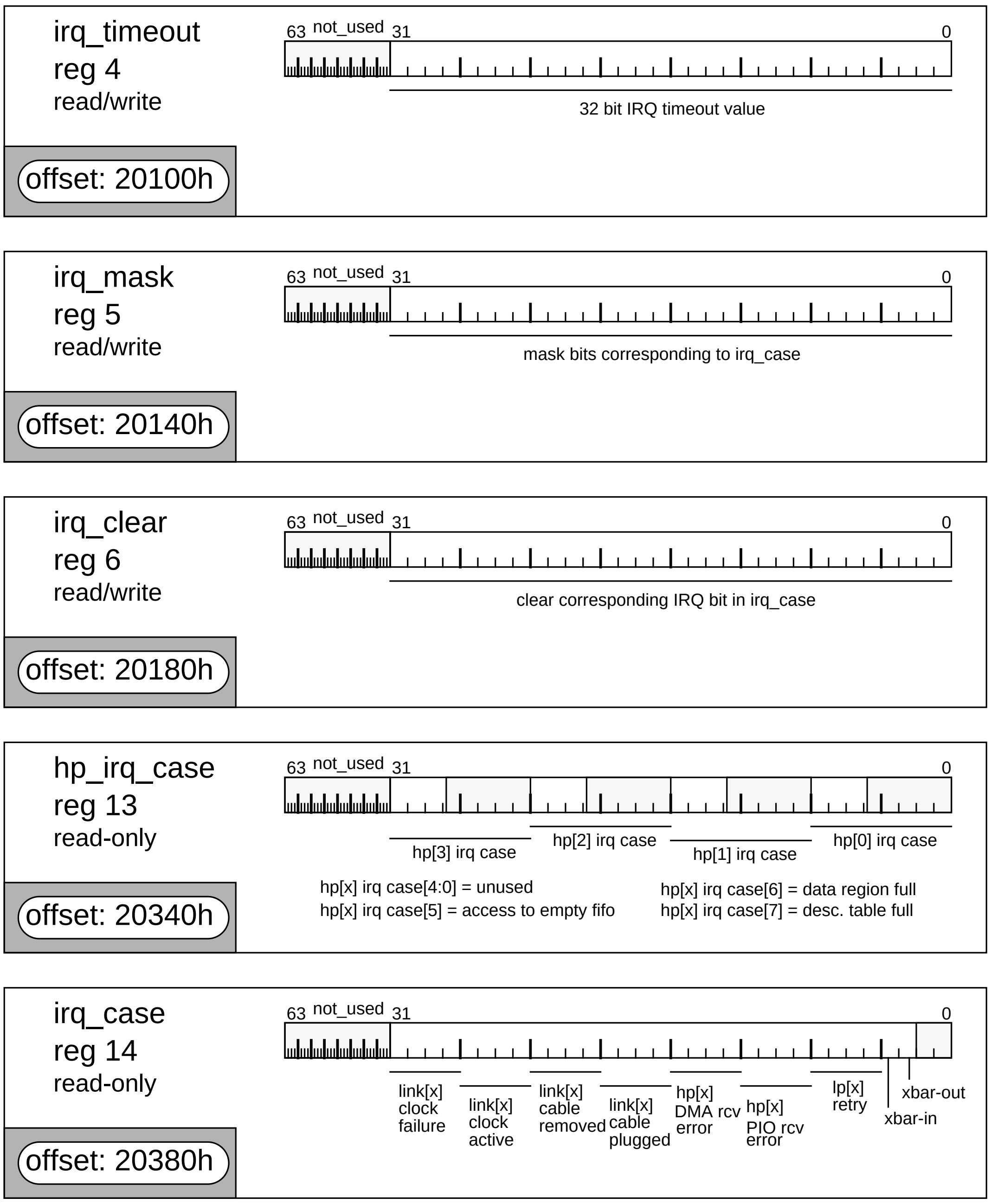


Figure 3-18 gives an overview about all registers related to the generation of interrupts. There are various sources for interrupt generation, and a user is able to mask those conditions, which should not result in a system interrupt. Each register is described in detail in the following:

- the irq_timeout register is used in case the host system has crashed and is not able to serve an interrupt. In such a case, the node should not block communication in the rest of the cluster by letting pending incoming messages block paths backwards into the network. In case of such a severe system failure, the device should simply consume all incoming data traffic. So every time an interrupt request is signaled by the device, an internal counter is loaded with this time-out value. If it runs down to 0 without any reaction from the host side, it is assumed that the host is down. This information is then flagged to all units

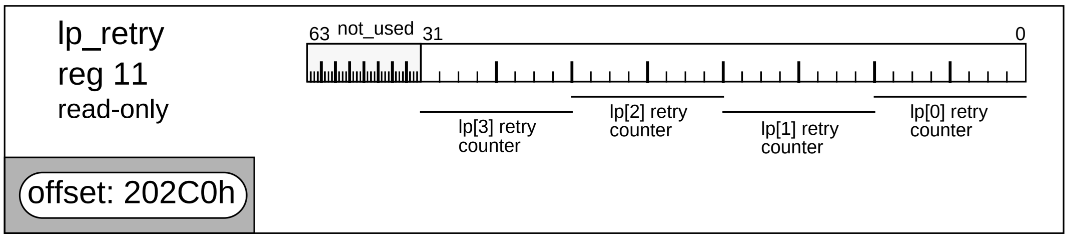
- the irq_mask register is used to mask specific interrupt sources. The bits correspond to the IRQ bits in the irq_case register. An interrupt is masked, when its bit is set to 0
- the irq_clear register is utilized to clear a corresponding interrupt in the irq_case register. Writing a 1 to a bit clears the interrupt
- the hp_irq_case register offers some more information in case an interrupt is generated by a host port. It flags a read access to an empty data FIFO in the PIO-receive unit. In case of the DMA-receive module, two data buffers in main memory can be filled to a level, where no more messages can be written out to memory. These are the descriptor table and the data region. These events are flagged by this register. The remaining 5 bits of a byte used for each host port were used in previous versions, but late modifications rendered them unnecessary

All possible interrupts are flagged by the irq_case register. The specific bits are described in the following:

- irq_case[3:2] are set if error conditions occur in one of the input or the output paths of the crossbar
- irq_case[7:4] flag an amount of bit errors and link packet retransmissions on a link. The actual limit for the generation of this interrupt is controlled via the two lp_retry_mux bits in the hw_cntl register
- irq_case[11:8] are used to flag an error in the PIO-receive units of a host port. They were intended to be the ‘logical OR’ of all possible PIO error conditions, which are then specified in the hp_irq_case in detail. But due to late modifications all but one error condition were dropped, so these bits are basically the same as the hp[x] irq case[5] bits, showing a read access to an empty PIO-receive data FIFO
- irq_case[15:12] show a general error in the DMA-receive units of a host port. They are the ‘logic OR’ of the two error conditions specified in the hp_irq_case register
- irq_case[23:20] and irq_case[19:16] observe the status of link active signals and are set, if a link cable is plugged or unplugged

- irq_case[31:28] and irq_case[27:24] observe the status of the associated link clocks and flag their activity or failure

Figure 3-19. Link retry register



The lp_retry status register, as shown in Figure 3-19, provides for each link the value of an 8 bit accumulator counting bit errors on links and the corresponding link packet retransmissions. They can be used to measure bit error rates over a greater time range.

Besides the registers presented over the last pages, there are other, less important ones, which are not described in detail here. There is an additional register gp_io to control 8 general purpose I/O pins. And some registers are used to debug the internal status of the crossbar. More specific information about them can be looked up in the ATOLL Hardware Reference Manual [67].

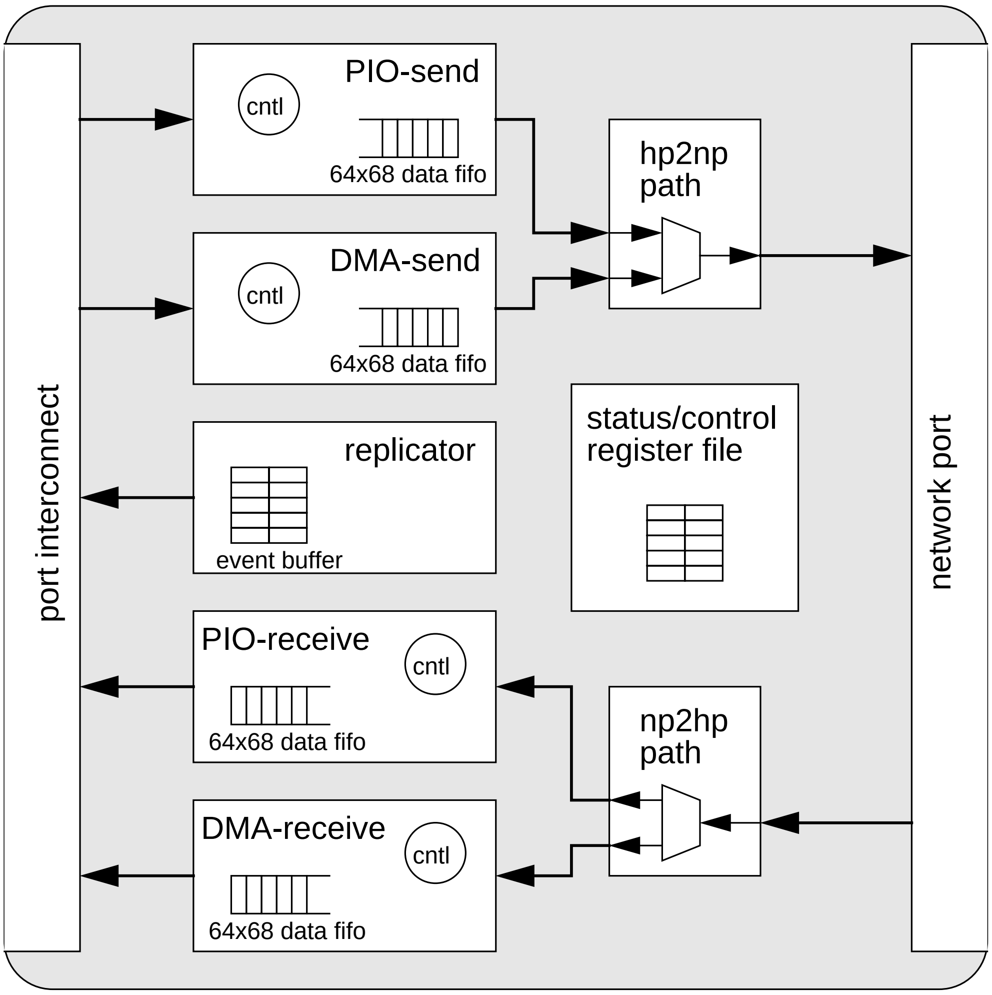
In the following, a typical interrupt service sequence is given:

- assuming the descriptor table of host port 3 is full and the receiving part is not able to spool out a new message to memory, the host port would flag this event to the interrupt control unit
- this unit compares the event with the corresponding bit in the irq_mask register to make sure the event is not masked
- it then raises the interrupt line, which is driven out to the PCI-X bus by the PCI-X bus interface
- a system trap calls the ATOLL driver software to deal with the interrupt
- the driver software checks the irq_case register to figure out the reason for the interrupt
- it allocates memory space for the descriptor table, either by enlarging it or moving messages in a temporary buffer

- the interrupt is then cleared by the software by writing to the corresponding bit of the irq_clear register. The host port resumes processing messages as soon as there are again free slots in the descriptor table

3.6 Host port

Figure 3-20. Structure of the host port

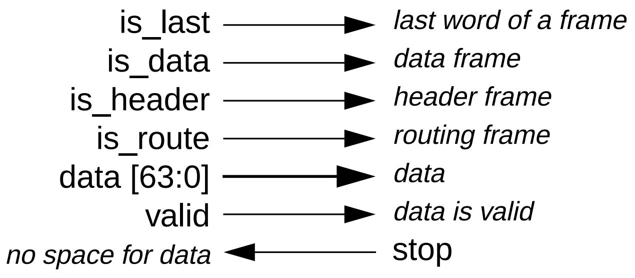


The host port [68][1] is the main building block of the ATOLL chip. It is responsible for data transfer from or to the network, either by PIO or DMA. Each host port has a small working set of some status and control registers needed to access data structures residing in main memory. Data paths for sending and receiving messages are strictly separated, offering the possibility of multiple concurrent data transfers into and out of the network. Each mode is also handled separately, so four unique modules are used to handle data

1. Berthold Lehmann implemented an early simulation model of the host port

transfer: PIO-send, PIO-receive, DMA-send and DMA-receive. Another unit keeps the working set and interfaces with all transfer modules. A sixth module called replicator is responsible to update relevant status information residing in main memory. This gives a CPU fast and cachable access to information without the need to frequently poll the device. Finally, two units are used to control the access of the host port to the interfacing network port. In case of sending messages, the access must be multiplexed between the PIO- and DMA-send units. For incoming messages, one must forward them either to the PIO- or the DMA-receive unit. This decision is configurable and relies on the size of the incoming message. Figure 3-20 depicts the overall structure of a host port.

Figure 3-21. Interface between host and network port.

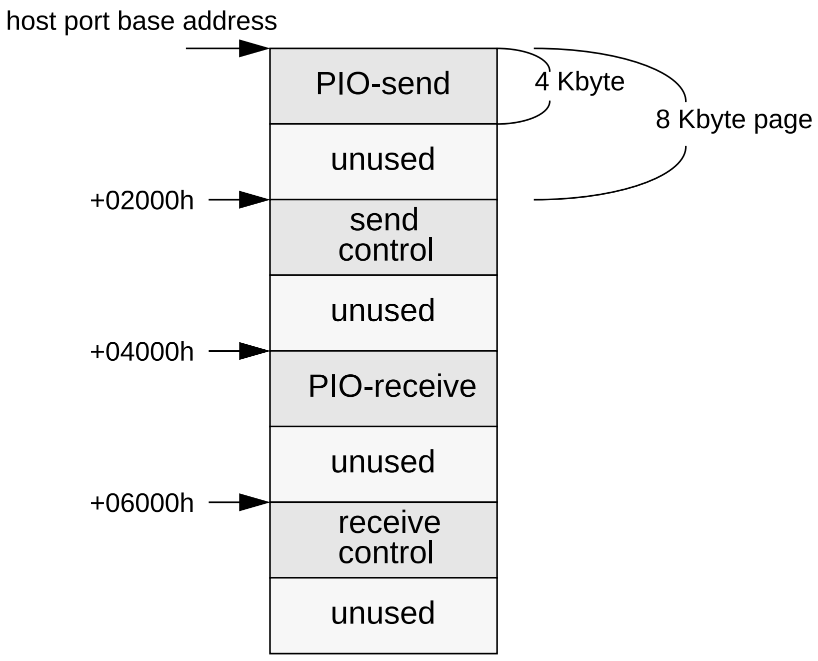


The interfaces towards the port interconnect correspond to the ones between the port interconnect and the synchronization interface described earlier. So DMA-receive and replicator use a Master-Write interface, DMA-send uses the Master-Read interface, PIO-send the Slave-Write interface, and the PIO-receive unit utilizes a Slave-Read interface. On the network port side, the data protocol is even more simple. A stream of tagged 64 bit words is transferred via the previously mentioned two-way handshake signaling. Two independent interfaces handle incoming and outgoing data traffic in parallel. Figure 3-21 depicts the interface signals. The tags associate each data word with one of the frames of an ATOLL message, and additionally mark the last word of a frame.

3.6.1 Address layout

Each host port occupies 4 consecutive 8 Kbyte pages in the address space of the ATOLL device, as described earlier in “Address space layout” on page 56. These are used to transfer data in PIO mode by read/write accesses to specific addresses. For the DMA-based data transfer, only a small working set is controlled via this address area. Figure 3-22 shows the general use of each page, whereas the detailed address layout of each separate page is described later in the corresponding sections. As stated earlier, ATOLL uses 8 Kbyte pages, but leaves the upper 4 Kbyte unused. Different pages might need different memory control/caching options, and this layout offers the possibility to support a wide variety of architectures with different page sizes.

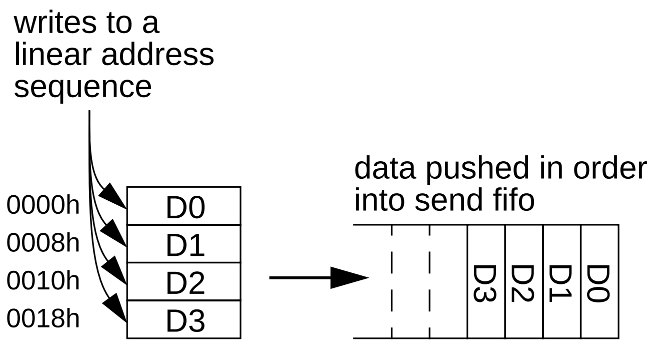
Figure 3-22. Host port address layout



3.6.2 PIO-send unit

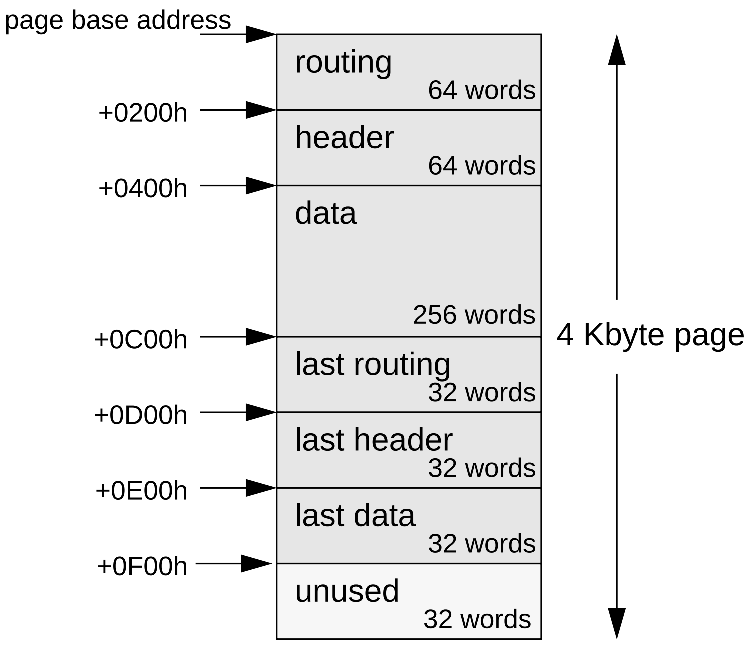
Programmed I/O is intended as a fast and efficient way to communicate small amounts of data. A lot of parallel applications tend to transfer only a few bytes per message, spreading global parameters or exchanging border values of a grid-based computation. A DMA-based data transfer involves copying data into buffers, setting up a job descriptor and waiting for the device to complete the job. This overhead can be ignored for larger messages, but adds up for small ones, significantly decreasing performance. So a method is needed to transfer a few words with nearly no overhead at all. These ideas led to the implementation of the PIO-send mechanism found in the ATOLL chip.

Figure 3-23. Mapping a linear address sequence to a FIFO


writes to a linear address sequence
0000h
0008h
0010h
0018h
D0
D1
D2
D3
data pushed in order into send fifo
D3
D2
D1
D0

The FIFO for keeping the data to be sent is directly made accessible to the user. The PCI-X bus is most efficiently used with burst transfers, combining as much data as possible into a single transaction. So pushing data into the FIFO is done by writing the data to a linear sequence of addresses. This gives the data the possibility to assemble itself in CPU and chipset write buffers to form a large burst transfer. Since the PCI-X bus delivers data strictly in order, there is no chance of data getting mixed up by out of order delivery. This mechanism is shown in Figure 3-23.

Figure 3-24. Layout of PIO-send page

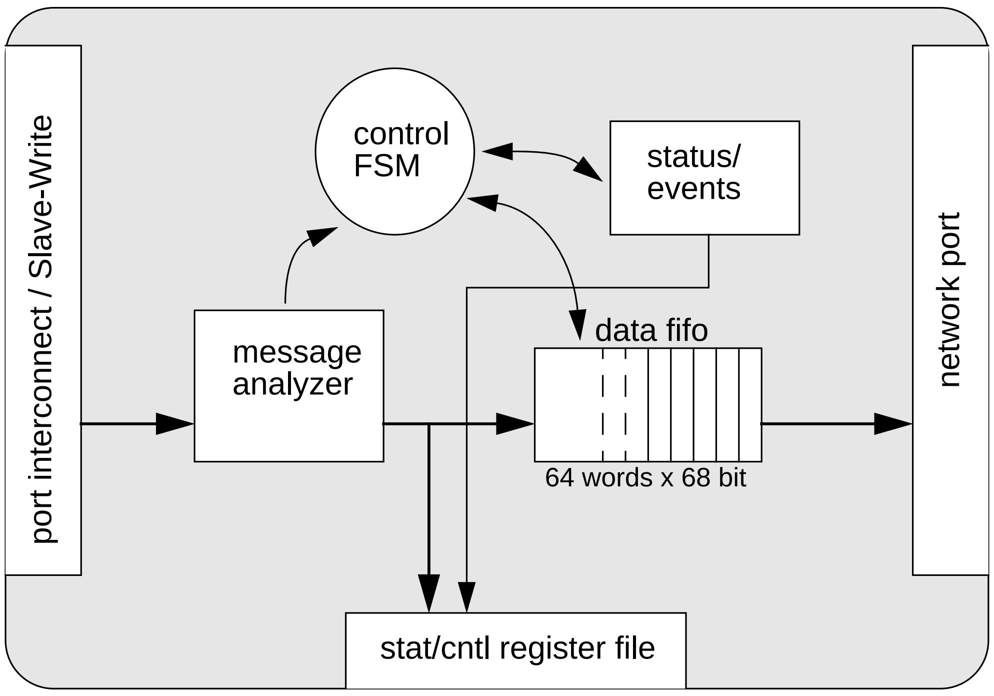


But in addition to the raw data, the PIO-send unit also needs to know the specific tags of the data word, according to the framing of ATOLL messages. This is done by providing different regions inside the page for each frame. And since also each last word of a frame must be marked, there are some addresses for only the last words of a frame. Most of a message is normally payload data, so this area has been assigned the largest address region. All address areas are depicted in Figure 3-24, together with their start offset relative to the page base address and their size in terms of 64 bit words. Since the FIFO has a depth of 64 words, all areas are sufficient to serve the largest bursts possible.

Figure 3-25 gives an overview about the structure of the PIO-send module. Apart from normal message data, this module also processes data which should be written into the status/control register file of the host port. So incoming data is first analyzed, and then handed over to the register file or pushed into the send FIFO. As soon as data is pushed into the FIFO, control logic requests the access to the network port. If the access is granted, data is forwarded to the network port. This immediate forwarding of the message can be delayed until a full message has been written to the FIFO by the configuration bit in the hp_cntl register described earlier. A separate unit is monitoring the fill level of the FIFO and passes this information on to the replicator and the register file. Assuming a message with one routing word, 4 header words and 8 data words, a typical sequence of sending a message via the PIO mode is as follows:

- first, the user checks the FIFO fill level to make sure, that there is enough space left to keep all message data
- the single routing word is written to the 'last routing' area
- 3 header words are written to the 'header' area, the 4.word to 'last header'
- 7 data words are written to the 'data' area, the 8.word to 'last data'

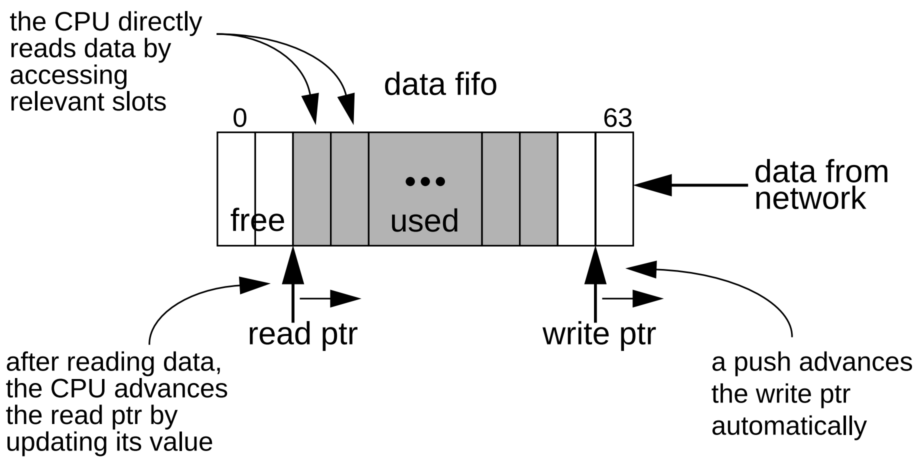
Figure 3-25. Structure of the PIO-send unit



3.6.3 PIO-receive unit

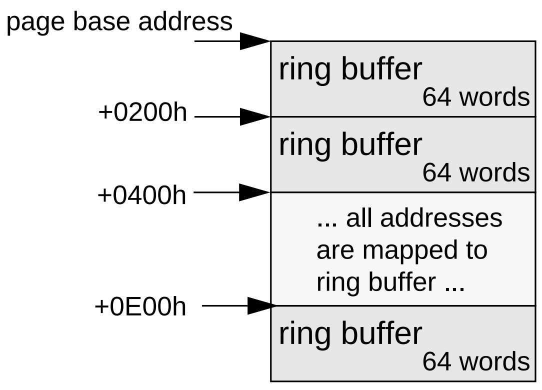
First versions of the PIO-receive unit implemented the same method as used by the PIO-send module, mapping the data FIFO to a set of contiguous address areas. But problems come up with this approach, when one considers prefetching data from the FIFO at various levels (chipset, CPU cache). Declaring the PIO-receive page as prefetchable is necessary to enable efficient data transfer between the device and the CPU. But one can not assure that data prefetched is also consumed by the CPU. E.g., in case of an interrupt some prefetched data might get lost, since a read access to the FIFO always pops the data out of the FIFO as side effect. So a subsequent reload of previously prefetched data results in delivery of wrong data.

Figure 3-26. Utilizing a ring puffer for PIO-receive


the CPU directly reads data by accessing relevant slots
data fifo
0
63
•••
free
used
data from network
read ptr
write ptr
after reading data, the CPU advances the read ptr by updating its value
a push advances the write ptr automatically

To deal with this problem one needs to remove the side effect of the read access. Therefore, the decision was made to give the user direct control over the deletion of data from the FIFO. This is done by treating the FIFO as a pointer-controlled ring buffer based on RAM (in fact, that is exactly the implementation method for large FIFOs in ATOLL). So data is pushed into the FIFO by the interfacing network port, advancing the write pointer of the ring buffer. A user is able to read data from the ring buffer, and frees up the accessed slots of the ring buffer by writing a new value of the read pointer to the unit. This mechanism is shown in Figure 3-26.

Figure 3-27. Layout of PIO-receive page



The page assigned to the PIO-receive module now references the ring buffer, whereby the buffer is continuously mapped all over the page, as depicted in Figure 3-27. This is simply done by taking the lower 6 bits of the start address as offset into the ring buffer. This also enables a user to read across the upper border from word 63 to word 0 without interruption. The first address is taken as start offset, then the unit delivers data as long as the transaction is valid.

Figure 3-28. Structure of the PIO-receive unit

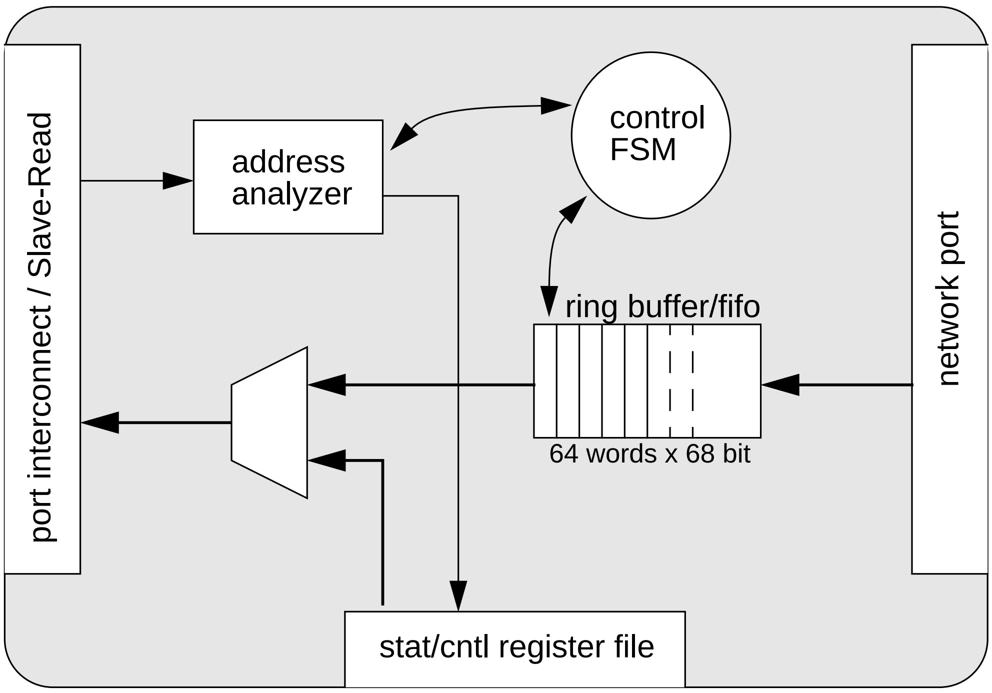


Figure 3-28 shows the structure of the PIO-receive unit. An incoming address is first analyzed to detect, if the read request targets the register file or the data FIFO. In case of the register file, the relevant offset is simply forwarded and data is returned on the next cycle. When addressing the ring buffer, data is popped from it until the transaction is ended. But internally, the pop operation is done using a temporary FIFO pointer. The read pointer, which is also used to determine the fill level of the FIFO, is left untouched, until it is explicitly updated by a write access to its entry in the host port register file.

3.6.4 Data structures for DMA-based communication

DMA-based communication is a good way to offload work from the main CPU. The CPU communicates with the device via a set of data structures and job descriptors, which exactly describe the work the device should do. There are two main data areas residing in main memory for DMA-based communication in ATOLL, called data region (DR) and descriptor table (DT). The data region is simply a place to assemble message data, which is to be send into the network by the device. This data region is needed, since ATOLL can only deal with physical addresses, and not with the virtual addresses of the source data locations in the user’s virtual address space. Though this is not impossible to implement, as demonstrated by Quadrics QsNet, its complex implementation was abandoned for the first version of ATOLL. So ATOLL needs a pinned-down memory area to temporary buffer the message payload.

Figure 3-29. Data region

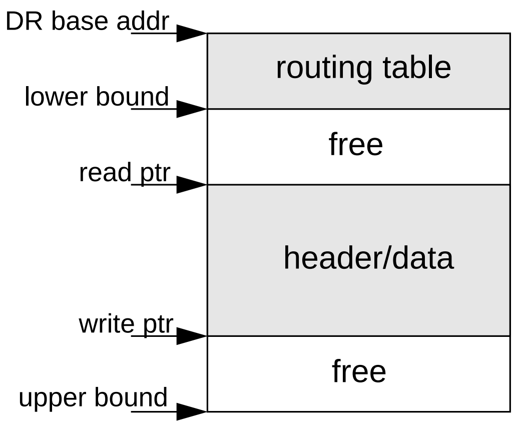
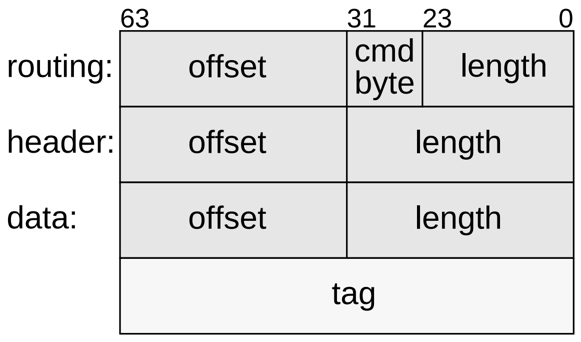


Figure 3-29 depicts the layout of a data region. A base address marks the beginning, whereas an upper bound marks the end of the area. Read and write pointers are used to index into this region, which is used as ring buffer. In contrast to header and data payload, the routing information of a message is normally static. At system start, all routing paths to all other nodes are computed and remain fixed during the application run. So to prevent the copying of the same data again and again into the data region, the decision was made to provide a small static area to keep the routing table. This is done by defining a lower bound, which is used as next offset in case data hits the upper bound. Separate data regions exist for sending and receiving data, with the receiving data region lacking a lower bound. It is simply not needed, since incoming messages have no routing information.

The second data area is the descriptor table, which keeps fix-sized job descriptors for each DMA message. The job descriptors consist of four 64 bit words. Three words are used to describe the three message frames, their offset relative to the base address of the data region and their length. The 4.word is an additional tag, which can be utilized by the software to distinguish between different protocols or message types.

Figure 3-30 shows the layout of a job descriptor. The offsets point to the associated frame data, relative to the base address of the data region. Lengths are given in terms of 64 bit words. The upper byte of the routing length is used as an additional command byte. Currently, only one special command is encoded. By setting the lowest bit of this byte to 1 one can mark a job as non-consuming. This means that data is read from the data region as normal, but the read pointer is not advanced afterwards. This offers the possibility to reference the data again. This method can be used to implement broadcasts on the sending side by copying the payload just once into the data region and referencing it with multiple descriptors.

Figure 3-30. Job descriptor



The command byte is used on the sending side only. When receiving a message, this byte is used as a status byte to flag special events associated with the message. In the current version of ATOLL, only the lowest bit is used to mark incomplete messages. This event occurs, when the number of words received from the network differs from the message length information, which is always at the start of the header frame of a message. E.g., this can occur, when a user process core dumps while sending a message in PIO mode, and the driver software completes the message, but not with the right number of words.

Similar to the data region, there are separate descriptor tables for the DMA-send and the DMA-receive units. As described earlier, a host port generates an interrupt when these resources are exhausted on the receiving side. Driver software should then always free up a large amount of memory space to make sure that messages waiting to be received do not block the network.

3.6.5 Status/control registers

The status/control registers needed for a host port are stored in a separate register file. It interfaces with all other components of a host port to gather the status data needed and spreads out the control information to the units. A separate page is used for the send and the receive part, but only nearly a dozen of registers are implemented for each page. Most of them store the base addresses and offsets needed for DMA-based message transfer, whereas only a few are needed to control the PIO mode.

Table 3-5 lists all registers of the send page. The offsets are relative to the page base address. All registers are 64 bit words, but not all bits might be relevant. Only the base addresses occupy all 64 bits, the other offsets are 32 bit wide. And the fill level of the PIO-send FIFO fits into the lower 6 bits of the associated register.

Table 3-5. Send status/control registers of a host port

register name	offset	mode, width	comment
snd_DT_base	0000h	r/w, 64	base address of the DT
snd_DT_read_ptr	0008h	r/w, 32	read pointer of the DT
snd_DT_write_ptr	0010h	r/w, 32	write pointer of the DT
snd_DT_upper_bound	0018h	r/w, 32	upper bound of the DT
snd_DR_base	0020h	r/w, 64	base address of the DR
snd_DR_lower_bound	0028h	r/w, 32	lower bound of the DR
snd_DR_read_ptr	0030h	r/w, 32	read pointer of the DR
snd_DR_write_ptr	0038h	r/w, 32	write pointer of the DR
snd_DR_upper_bound	0040h	r/w, 32	upper bound of the DR
snd_repl_base	0048h	r/w, 64	base address of the replicator
snd_fifo_fill_level	0800h	ro, 6	fill level of the PIO-send data FIFO

Table 3-6 lists all registers of the receive page. The DMA registers are almost the same as for the send page, except the unnecessary lower bound register for the data region.

Table 3-6. Receive status/control registers of a host port

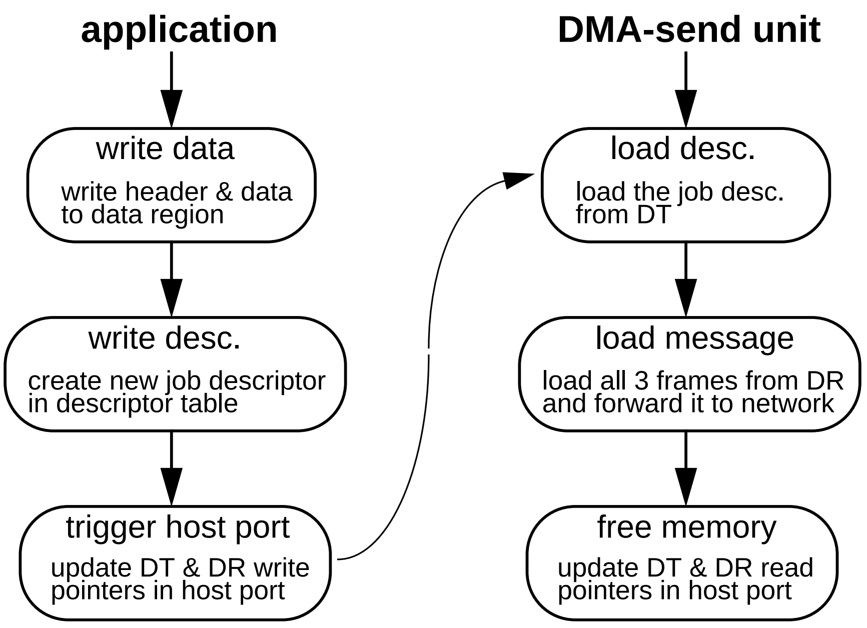
register name	offset	mode, width	comment
rcv_DT_base	0000h	r/w, 64	base address of the DT
rcv_DT_read_ptr	0008h	r/w, 32	read pointer of the DT
rcv_DT_write_ptr	0010h	r/w, 32	write pointer of the DT
rcv_DT_upper_bound	0018h	r/w, 32	upper bound of the DT
rcv_DR_base	0020h	r/w, 64	base address of the DR
rcv_DR_read_ptr	0028h	r/w, 32	read pointer of the DR
rcv_DR_write_ptr	0030h	r/w, 32	write pointer of the DR
rcv_DR_upper_bound	0038h	r/w, 32	upper bound of the DR
rcv_fifo_fill_level	0800h	ro, 6	fill level of the PIO-receive data FIFO
rcv_fifo_read_ptr	0808h	r/w, 6	read pointer of the PIO-receive data FIFO
semaphore	FC00h	r/w, 64	semaphore, set on read as side effect
cnt	FE00h	ro, 64	global counter

Besides the fill level of the PIO-receive data FIFO, the receive page also includes a register for the FIFO read pointer. Writing a new value to it is equivalent to shifting out the previously read data words. Each host port has also read access to the internal global counter, which is located in the supervisor initialization registers described earlier. It is simply spread out to all host ports, so user applications can e.g. precisely time certain ATOLL library calls.

Another special feature is the semaphore register. It is intended to be utilized in case the host port needs to be multiplexed between several applications. This is normally not the case for production-type clusters, since performance is degraded significantly. But it might make sense for application development or debugging purposes. A write access simply writes the specified data to the register. But a read access returns the stored value and, as side effect, sets all bits to 1. So one could initialize the semaphore by writing a 0 to it. When several user processes try to read the register, only one gets the 0 as value, all others see all bits set to 1. So the one with the 0 has acquired the semaphore, all others must wait, until the locking application frees it by writing a 0 back to the register.

3.6.6 DMA-send unit

Figure 3-31. Control flow of sending a DMA message


application
write data
write header & data to data region
write desc.
create new job descriptor in descriptor table
trigger host port
update DT & DR write pointers in host port
DMA-send unit
load desc.
load the job desc. from DT
load message
load all 3 frames from DR and forward it to network
free memory
update DT & DR read pointers in host port

The DMA-send unit is responsible to observe the data areas for sending messages, and to process all valid entries in the send descriptor table. If a job is available, it loads the descriptor, and subsequently loads all message frame data. As soon as requested data is returning from the PCI-X interface it is forwarded to the network port. Figure 3-31 shows the general control flow of sending a message via the DMA mode, for both the user application and the DMA-send unit. The application triggers the DMA-send control logic by writing new values of the write pointers for DT & DR into the host port. These are continuously monitored by the host port. As soon as read and write pointers of the descriptor table are unequal, a valid job entry is present and processed.

Figure 3-32. Structure of the DMA-send unit

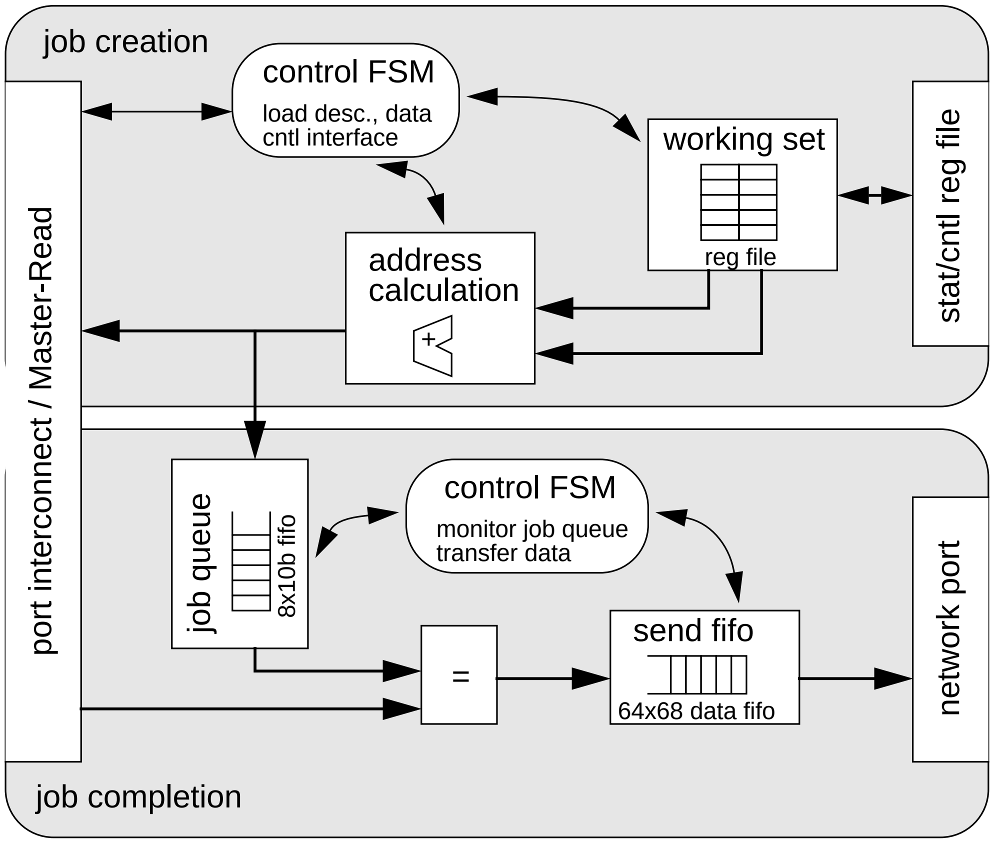


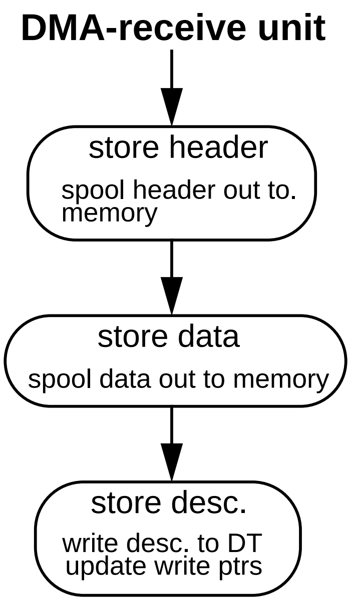
Figure 3-32 depicts the structure of the DMA-send unit. It is split up into two subdesigns. One part is responsible for requesting data from memory, labeled with ‘job creation’ in the figure. As described earlier, the Master-Read interface implements a split phase transaction scheme, where blocks of up to 32 words can be requested. Each job is assigned a unique ID, with up to 8 jobs in progress. The ‘job creation’ part now keeps copies of the relevant status/control registers in a working set module. This working set is continuously updated from the status/control register file of the host port.

As soon as an entry is detected in the descriptor table, it is requested from memory and stored in the working set. Step by step the ‘job creation’ part now computes the load addresses from fixed base addresses and the offsets given in the descriptor. A set of jobs is then generated to fetch all frame data from memory. The associated job IDs are pushed into a job queue. This is necessary since multiple host ports might request data in an arbitrary sequence. So each host port must keep the jobs it has requested to match them with the ID of returning jobs.

The second part is called ‘job completion’ and is responsible to accept the data the other part has requested. It therefore monitors the IDs of incoming jobs to detect the ones it should process. Incoming data is then placed in a FIFO, marked with its corresponding tags. As soon as data is in the FIFO, this part requests access to the network port. If granted, data is transferred towards the network. Since the FIFO can only store up to 64 words, the unit cannot request more data. Doing so could cause all data paths to be blocked in case the path towards the network is not available. This can happen when also the PIO-send unit is transferring a message, or the path through the crossbar is occupied by a message just passing through the ATOLL chip.

3.6.7 DMA-receive unit

Figure 3-33. Control flow of the DMA-receive unit



The DMA-receive unit is the counterpart to the DMA-send unit. It gets message data from the network port and spools this data out to the DMA data structures in main memory. Figure 3-33 shows the main control sequence of the unit. An incoming message consists of the header and the data frame, since the routing frame is no more needed.

Similar to the DMA-send unit, the DMA-receive module keeps the relevant registers in a small working set. It then writes header and data out to memory. After all message data has been processed, it assembles a descriptor pointing to the data spooled out to memory. This job descriptor is then stored in the receive DT, which is monitored by the user application. The write pointers of both DT & DR are updated to reflect the newly arrived message.

Figure 3-34. Structure of the DMA-receive unit

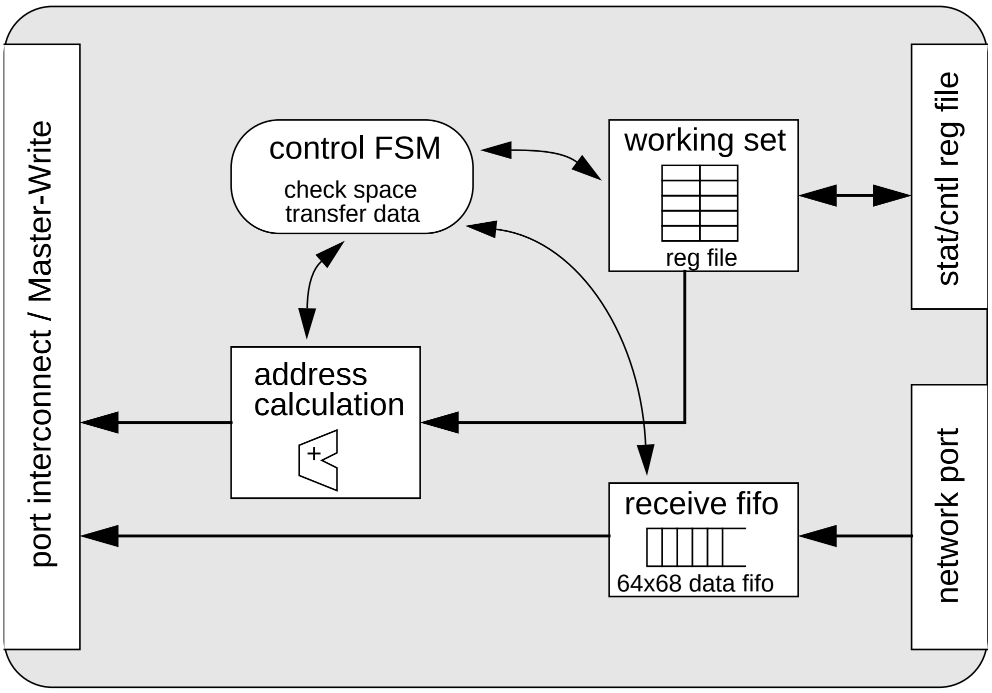


Figure 3-34 gives an overview about the structure of the DMA-receive module. Incoming data is first stored in a FIFO. Addresses are computed for each data word from the register values of the working set. Prior to handing data over to the Master-Write interface of the port interconnect, the control logic makes sure, that enough memory space is available to store the message in the DMA-receive data structures. If this is not the case, the unit generates an interrupt and waits, until software frees up memory and updates the relevant read pointers of DT & DR.

Though each data word is transferred with its address, it makes no sense to mix multiple data streams from several active DMA-receive units on a word-by-word basis. So as soon as data enters the FIFO from the network side, the unit requests exclusive access to the port interconnect. If granted, it keeps the request up until no more data is available. Only then the request is deasserted, and other host ports might get access to the port interconnect. This way a strong fragmentation of the data stream is avoided, with the probability to assemble larger burst transfers.

3.6.8 Replicator

The replicator is used to avoid polling status registers of the ATOLL device, which could degrade overall system performance by interfering with data transfer bursts. Those registers needed by application software are automatically written out to a separate portion of main memory. There are 8 registers in total written out, separated in two groups of 4 registers, one for send and one for receive operations. An update consists of all 4 registers of each block. New register values are handed over to the replicator, and additional trigger signals force it to update the values in memory.

The replicator utilizes the same Master-Write interface of the port interconnect as the DMA-receive unit. Table 3-7 shows the layout of the mirrored registers in main memory. Each host port can be assigned a unique base address for its replicator through the associated register in its control/status register file. The offsets given are relative to this base address. Though located in normal main memory, those locations are declared as read-only, since they should not be altered by software (except initialization at system start-up).

Table 3-7. Layout of the replicator area

register name	offset	mode, width	comment
snd_DT_read_ptr	0000h	ro, 32	read pointer of the send DT
snd_DR_read_ptr	0008h	ro, 32	read pointer of the send DR
snd_fifo_fill_level	0010h	ro, 6	fill level of the PIO-send data FIFO
semaphore	0018h	ro, 64	current semaphore value
rcv_DT_write_ptr	0020h	ro, 32	write pointer of the receive DT
rcv_DR_write_ptr	0028h	ro, 32	write pointer of the receive DR
rcv_fifo_fill_level	0030h	ro, 6	fill level of the PIO-receive data FIFO
rcv_fifo_read_ptr	0038h	ro, 6	read pointer of the PIO-receive data FIFO

The read and write pointers of the DMA modes are updated on each change. This is also true for the semaphore and the read pointer of the PIO-receive FIFO. But as described in previous sections, the update frequency of the FIFO fill level can be configured. So while utilizing these registers in the replication area one must keep in mind, that the real value might differ from the mirrored one by up to the threshold value. E.g., if an update is made every 8 words and the mirrored FIFO fill level is 32, then the real FIFO fill level inside the host port can be in the range 25-39. And even worse, there can be pending updates waiting to be transferred to main memory in the device. This might happen if the data paths are congested due to multiple active host ports.

So before interpreting the value in the replication area, an application should read the value directly from the device once to make sure, that it is the same as the mirrored one. And while doing calculations with these values, e.g. to check if the PIO-send FIFO has enough space left for a message to send, the software should take the threshold into account.

3.7 Network port

The network port is converting the tagged data stream of 64 bit words into a byte-wide stream according to the link protocol defined for ATOLL and vice versa. It is split into two separate units for sending and receiving messages. There is no interaction between those two units, so they both can process messages concurrently. The sending part also generates CRC values for error detection later in each network stage.

3.7.1 Message frames and link protocol

As stated earlier, an ATOLL message consists of three frames: routing, header and data. The routing frame is consumed while the message is routed towards its destination, so only header and data frame enter the receiving path of a network port. Each frame consists of several link packets, which can consist of up to 64 data bytes. Only full 64 words are used to form a message, so the number of data bytes is always a multiple of 8. At least one data word must be in every frame to ease the implementation.

Figure 3-35. Message frames

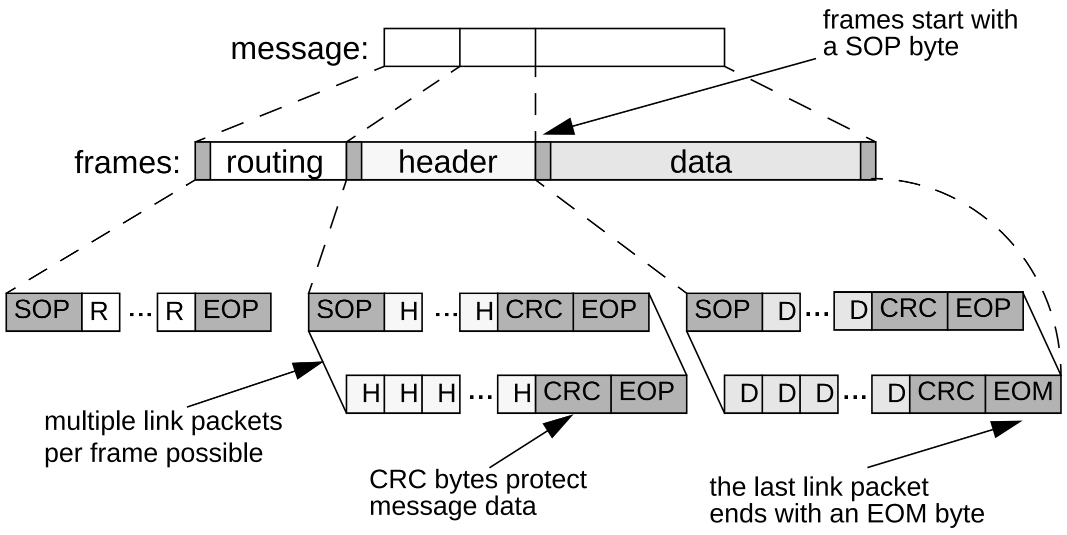


Figure 3-35 gives an overview about the framing of ATOLL messages and the link protocol used for the byte-wide data stream in the network. Separate control bytes are used to enclose the normal message data bytes and to ensure a correct framing. An additional ninth bit is used to distinguish between control and data bytes (0 = data, 1 = cntl).

There are 4 control bytes used at the network port to build up the framing of message data:

- SOF (Start Of Frame) is the 1.byte of a new frame

- a CRC value is computed for each link packet in the header and data frames. It is appended to the end of the link packet
- EOP (End Of Packet) marks the end of a link packet
- EOM (End Of Message) marks the end of the whole message, and replaces the EOP byte of the last link packet

Link packets in the routing frame do not have a CRC value attached, since one must react on bit errors in routing bytes immediately. This is handled by using an error detecting code for routing bytes, which is discussed in detail later on.

Each frame can be composed of several link packets, but the normal case should be that a single link packet is enough for the routing and the header frame. E.g., 64 routing bytes are sufficient to define routing paths in a 2D grid of 32 x 32 = 1024 nodes. A special constraint exists for the header frame. The 1.word of it must be the length of the message, given as separate 32 bit values for header and data frame. The 2.word must be the tag. This is necessary to give the host port at the receiving side the possibility to check autonomously, if the message should be received in PIO or DMA mode. As stated earlier, this decision is made based on the length of the message and a configurable threshold. The DMA-send unit automatically ensures this constraint, but e.g. software utilizing the PIO-send mode has to take care of this constraint, too.

3.7.2 Send path

Figure 3-36. Structure of the send network port unit

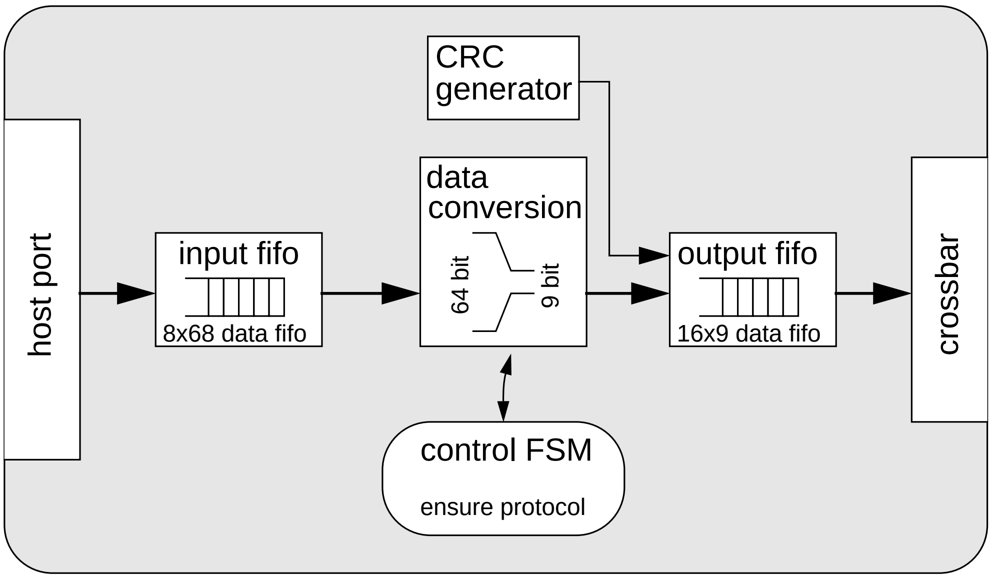
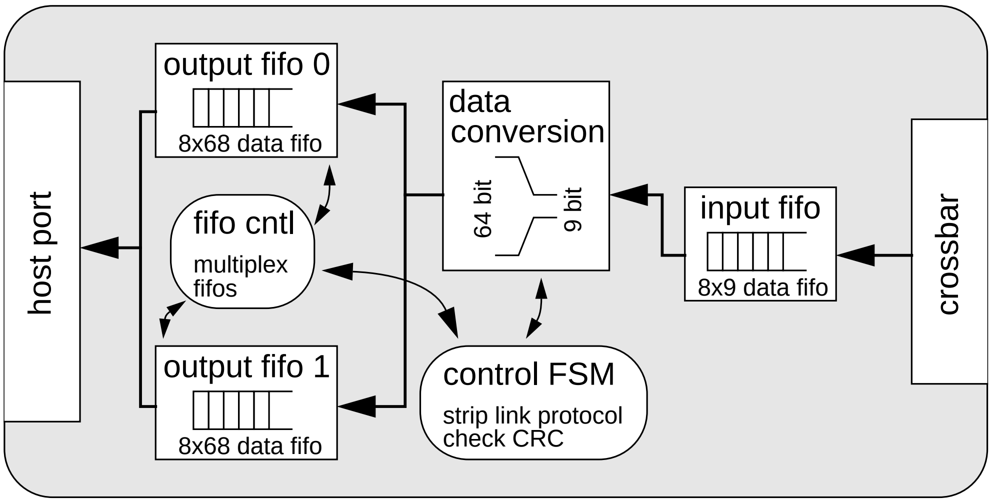


Figure 3-36 depicts the overall structure of the send unit of the network port. It can be viewed as a 3-stage pipeline consisting of an input, a processing and an output stage. The host port delivers a stream of tagged 64 bit words. This data is temporary stored in a small FIFO. The following processing stage now shifts out the data word by word and forms a stream of bytes, tagged with the cntl/data flag bit. It inserts control bytes according to the link protocol where necessary. A CRC generator computes the CRC value of the processed data. After a full link packet has been processed, this CRC value is appended and the link packet is completed by an EOP or EOM control byte. This stream of link data is pushed into an output FIFO, which keeps the data until it is transferred towards the crossbar.

3.7.3 Receive path

Figure 3-37 shows the structure of the receive unit inside the network port. It basically consists of the same 3-stage processing pipeline as the send part, but only in the opposite direction. Data assembles in a small input FIFO. The following protocol stage strips off the control bytes from the link data stream. It rebuilds a stream of tagged 64 bit words. There might be some unused routing bytes left at the head of the message. This happens e.g. when only 3 network hops are needed. Since frames are made of multiple 64 bit words, 5 bytes are unused. These are simply dropped until the header frame begins.

Figure 3-37. Structure of the receive network port unit



The output stage is a bit more complex than in the send unit. Incoming link packets might be marked as corrupted, since transmission errors were discovered somewhere along the path the message took through the network. These packets end with a special error control byte EOP_ERR (End Of Packet ERRor), instead of the normal EOP byte. Since these packets are automatically retransmitted to the network stage which detected the error, the same link packet follows the corrupted one in the data stream. So the corrupted link packet needs to be filtered out at the receive unit of the network port. This means data pushed into the output stage cannot be forwarded to the host port prior to the complete reception of a correct link packet.

Providing only one output FIFO would result in a significant performance penalty, since the data stream would be repeatedly stopped and restarted. This would happen, since pushing data into the FIFO and popping data from it cannot be overlapped. In case of erroneous link packets one would otherwise need to keep track of which words to transfer and which ones to delete.

To keep data flowing a second FIFO was implemented, and the data path switches on each link packet between them. This way, the normal operation of this unit is as follows:

- a link packet is processed and pushed into FIFO 0
- after making sure it is valid, the next incoming packet is pushed into FIFO 1 and the control logic of the output stage is ordered to forward the data in FIFO 0
- while the second link packet is processed, the first one is transferred to the host port
- after finishing the second packet, FIFO 0 is empty again and roles are switched again

Simulations showed a significant gain in sustained throughput of the unit using two output FIFOs instead of just one. In case of only one FIFO, a few wait cycles were introduced in the network port. But even more serious was that these short periods of blockages in the data stream triggered the flow control of previous links, causing larger idle times on the links along the data path.

3.8 Crossbar

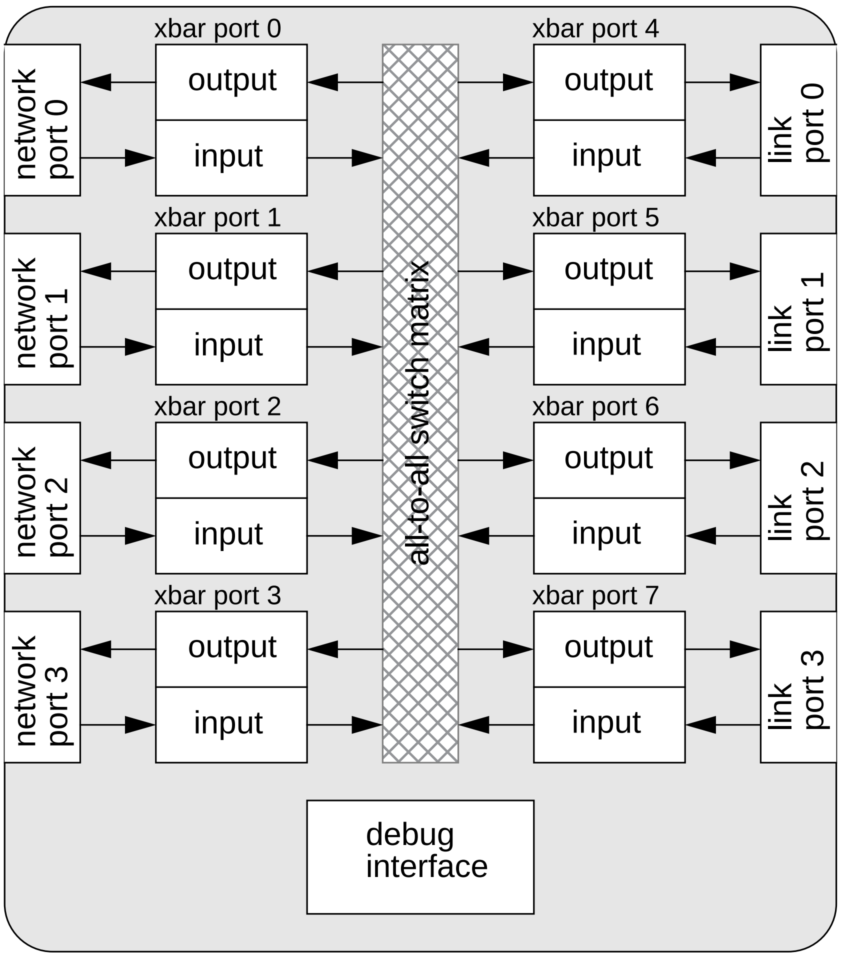
The crossbar [69][1] is a full-duplex 4x4 switch, providing an all-to-all connection between the 4 network ports on one side and the 4 link ports on the other side. It is made of 8 identical crossbar ports, which again are split up into an input and an output unit. An additional debug interface observes the status of the crossbar and can be utilized to resolve major failures, like deadlocks or misrouted messages. The overall structure of the crossbar is shown in Figure 3-38.

The input unit of a crossbar port strips off the first routing byte of an incoming message and sets the request for the addressed crossbar port. This can be any of the 8 ports, includ-

1. designed and implemented by Prof. Dr. Ulrich Brüning and Jörg Kluge

ing itself. It then waits for the other side to grant the access and forwards data, until the end of the message is reached. To not monopolize a specific crossbar port, each input unit deasserts a request for at least 3 cycles between back-to-back messages. This gives other ports a chance to request the port.

Figure 3-38. Structure of the crossbar



The output unit of a crossbar port arbitrates its data path in a fair round-robin fashion. Once a request is served, data flows into the unit and is buffered in a small FIFO. If the following unit signals its ability to accept data, the message is transferred out of the cross-bar. All interfaces use the two-way handshaking introduced earlier.

The additional debug interface observes the status of the crossbar and can be accessed via the debug registers listed in “ATOLL control and status registers” on page 71. The detailed implementation of the crossbar and the use of the debug interface is beyond the scope of this document. Further information can be looked up in the ATOLL Hardware Reference Manual [67].

3.9 Link port

The link port is the gateway to the network. It directly drives and receives signals over the link cables. As all other top-level building blocks of ATOLL, it is also split up into independent units for sending and receiving data. Its main task is to ensure proper and error-free transmission of data. This includes a reverse flow control protocol to prevent buffer overflow on the receiving side in case of blocked data paths.

A unique feature of an ATOLL link is its per-link error detection and correction. In contrast to the end-to-end error detection and software-driven retransmission found in other networks, ATOLL link packets are checked in each network stage. Retransmission is completely handled by hardware and occurs immediately on the link the error was introduced. This provides an extremely fast way to solve the issue of rare bit errors due to environmental influences on the link cables.

3.9.1 Link protocol

Two types of bytes make up the link data stream: data and control bytes. They are differentiated by an additional ninth bit. Data bytes itself can be separated into two classes: routing bytes and normal payload bytes for the header and data frames. A routing byte is special, since it carries the information of the output port its message should take at the next crossbar. A bit error in a routing byte would result in catastrophic failure of the network. The message would take a different path then specified, causing it to arrive at a false destination, or even worse, let it end somewhere in a network stage due to missing routing information. So one must react immediately in case of errors in routing bytes, the normal CRC link packet protection scheme of the other two frames is useless here.

Therefore, the upper bit of a routing byte is a parity bit calculated from the remaining 7 bits. This offers the opportunity to detect a single-bit error immediately. Such a faulty bit is replaced with a special CANCEL control byte, signalizing all other downstream units that this routing link packet is to be ignored. It is then retransmitted just like any other erroneous link packet.

Table 3-8 gives an overview about the format and encoding of all data and control bytes. The first two rows show the encoding of normal data and routing bytes, whereby bit d[8] is the bit distinguishing between data and control bytes. The rest of the table depicts the encoding of all 11 control bytes used in the ATOLL link protocol.

Table 3-8. Encoding of data and control bytes

hex	d[8]	d[7]	d[6]	d[5]	d[4]	d[3]	d[2]	d[1]	d[0]	comment
0xx	0	-	-	-	-	-	-	-	-	normal data byte
0xx	0	par	-	-	-	-	-	-	-	routing byte with parity bit
	d[5]	d[4]	d[3]	d[2]	d[1]	p[2]	d[0]	p[1]	p[0]	ECC bit positions, hamming code
1FF	1	1	1	1	1	1	1	1	1	IDLE (filler byte)
100	1	0	0	0	0	0	0	0	0	SOF (Start Of Frame)
107	1	0	0	0	0	0	1	1	1	EOM (End Of Message)
119	1	0	0	0	1	1	0	0	1	EOP (End Of Packet)
11E	1	0	0	0	1	1	1	1	0	EOP_ERR (End Of Packet ERRor)
12A	1	0	0	1	0	1	0	1	0	STOP (STOP sending of data)
12D	1	0	0	1	0	1	1	0	1	CONT (CONTinue sending data)
133	1	0	0	1	1	0	0	1	1	POSACK (POSitive ACKnowledge)
134	1	0	0	1	1	0	1	0	0	NEGACK (NEGative ACKnowledge)
14B	1	0	1	0	0	1	0	1	1	RETRANS (RETRANSmit link packet)
14C	1	0	1	0	0	1	1	0	0	CANCEL (CANCEL routing)

Since only data bytes are protected by link packet CRCs or parity bits, another method had to be found for the control bytes. They are protected by a hamming code, which is an error correcting code (ECC). So an error is not only detected, but also corrected on the fly. This is accomplished by using 3 bits of the byte as parity bits, marked as p[2:0] in the table. The parity bits are encoded as follows:

```
p[0] = XOR(d[6], d[4], d[2])
p[1] = XOR(d[6], d[5], d[2])
p[2] = XOR(d[6], d[5], d[4])
```

Using this hamming code, a correct control byte can be identified by p[2:0] = 000. Any nonzero value of the parity bits points to the erroneous bit position, which is then simply inverted to correct a single-bit error. E.g. a value of `011` for the parity bits identifies bit d[2] as incorrect.

The control bytes SOF, EOM, EOP, EOP_ERR and CANCEL were described earlier. The IDLE byte is simply used as filler, since a byte is transmitted on each clock edge over a link. STOP and CONT are the bytes used for the reverse flow control. POSACK and NEGACK are used by a receiving link port to signal the sender a good or a bad link packet. The RETRANS byte is used to lead any retransmitted link packet.

3.9.2 Output port

Figure 3-39. Structure of the output link port

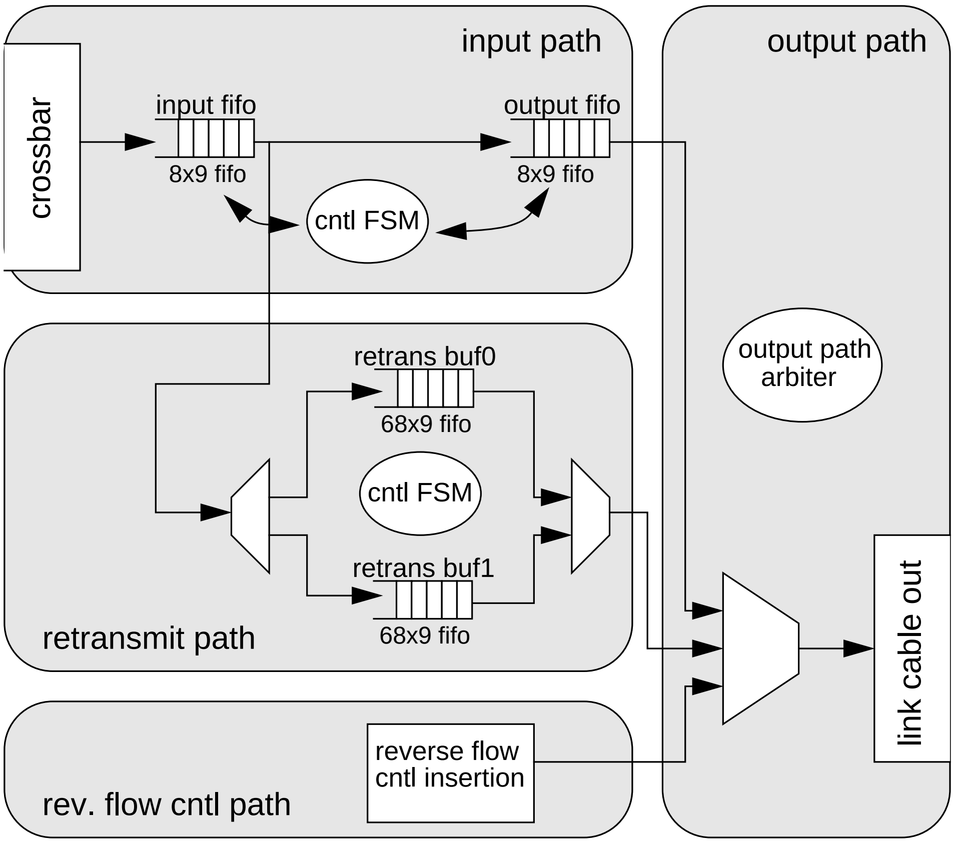


Figure 3-39 gives an overview about the structure of the link port output unit. It is split up into 4 areas according to the different tasks it should manage. The input path gets message data from the crossbar and stores it temporary in a small input FIFO. Control logic now forwards each link packet towards the output path, storing data in another FIFO.

The retransmit path is responsible to keep a copy of each link packet sent over the link. If the receiving side signals a corrupted packet, this unit can retransmit the erroneous packet. Since the acknowledge from the other side has a certain delay (error detecting logic, cable propagation time), it would be a waste of bandwidth to wait for it after each single link packet. So the retransmit path has two identical retransmit buffers to be able to store 2 packets in parallel.

During the unit waits for the acknowledgment of the first packet, a second one can be transmitted. Thus transmission and acknowledgment of link packets is overlapped, offering the possibility to operate the link at full capacity. Sometimes it happens that small packets are transferred, e.g. routing or header packets with only 1-2 words. If both buffers are filled and wait for their corresponding acknowledge, the input path is stopped as long as one of the buffers is free again.

Figure 3-40. Reverse flow control mechanism

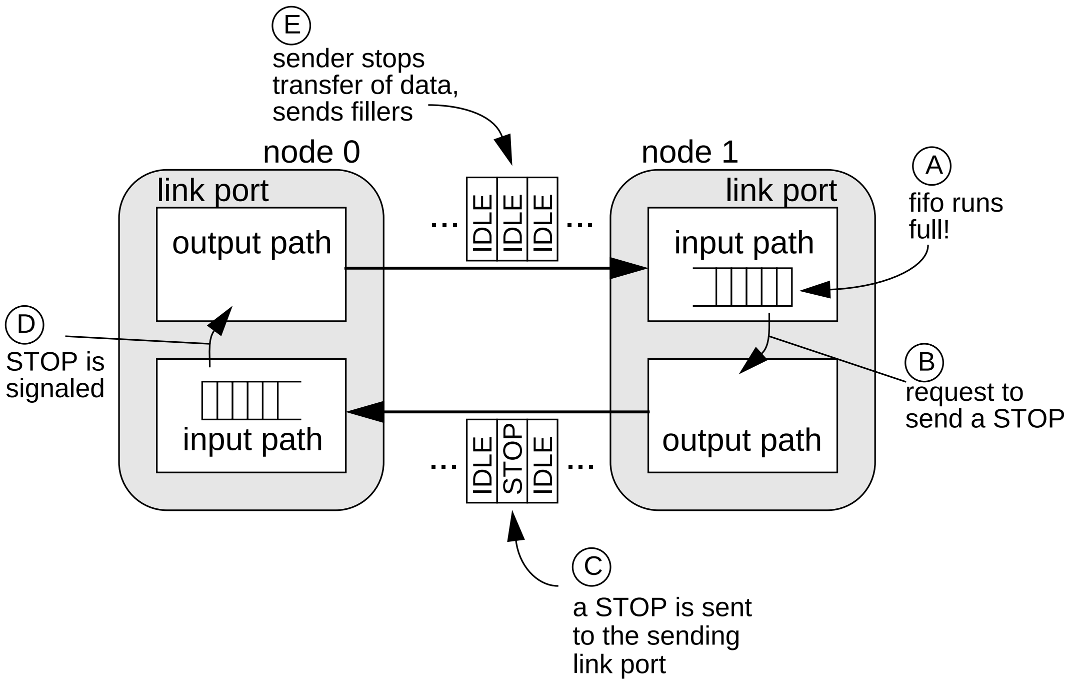


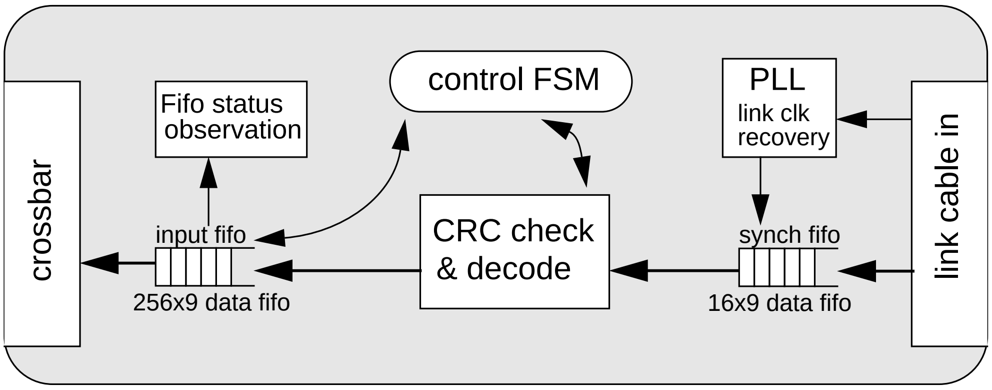
Figure 3-40 shows how the reverse flow control path is utilized to insert control bytes into the opposite data stream of a full-duplex link:

- A: the FIFO on the receiving side runs full, because the path is blocked further ahead
- B: the input path requests its associated output path to send a STOP signal
- C: the STOP byte is inserted into the data stream on the opposite path of a link
- D: the input path of the sender filters out the STOP byte and signals its reception to the output path
- E: the sending output path recognizes the STOP request and stops transmission of message data. Instead, it sends only IDLE fillers

The same procedure happens in case the blockage of the path is removed and the FIFO can again store data. This is signaled by sending a CONT control byte. This mechanism is also used for the acknowledgment of link packets. The checking input path requests to send a POSACK or NEGACK byte, which is filtered out at the other side. So the reverse flow control path of the output unit is responsible for the insertion of these four control bytes into the link data stream. Additional arbiter and datapath logic multiplex the access to the link cable between the three subunits of the output path.

3.9.3 Input port

Figure 3-41. Structure of the input link port



The structure of the input path of the link port is shown in Figure 3-41. It can be viewed as a pipeline of 3 stages: data synchronization, error checking & decode and storage. Since all ATOLL chips have their own internal clock signal, one must first synchronize the incoming data to the clock of the receiving node. An additional tenth signal line is used in the cable to transfer the clock. This is done at half the rate to prevent the introduction of noise onto the data wires. This clock signal is now recovered by a PLL and used to push data into a dual-clock data FIFO. Only non-ILDE bytes are pushed into the FIFO to prevent an overflow by a slightly faster running sender. The output port makes sure, that at least 1 IDLE byte is sent per 64 byte link packet.

The second stage decodes all control bytes and filters out flow control signals, which need to be forwarded to the corresponding output path (STOP, CONT, etc.). It checks routing bytes for parity errors, possibly corrects bit errors of control bytes by using the hamming code, and checks the CRC of link packets. For debugging purposes, a multiplexer can be used to choose between the data from the cable and the data sent out by the link port. This offers the possibility to run the link port in a kind of loopback mode.

Finally, data is pushed into a large FIFO. This FIFO is observed by some control logic, which triggers the sending of STOP and CONT control bytes to stop and restart the link data stream. The FIFO can store up to 256 bytes, and the data stream is halted, when it is filled to 50 %. For a short period of time data is still coming into the link port, since the request needs some time to arrive at the opposite side. This configuration is enough to support links of up to 25 m. Falling again under the 50 % mark triggers the request to send a CONT byte.

4 Implementation

The implementation of the ATOLL architecture as an Application Specific Integrated Circuit (ASIC) [70] has been a huge challenge. With the complexity and the aggressive level of integration, the task of implementing the ATOLL chip is comparable to high-end commercial chip developments at major semiconductor companies. The goal could only be attained by following a carefully planned project schedule, maximizing the productivity of the manpower at our disposal. Other big hurdles have been the restricted financial budget and the limited access to chip development tools. Normally, ASIC projects of this size are heavily supported by the technology and tool vendors to help the design team solving critical issues. The available support from this side for the ATOLL project was very limited. But despite all these obstacles, the design team was able to ship the final transistor layout to production in February 2002, almost 3 years after implementing the first simulation modules.

The Europractice IC Service[1], which is funded by the European Union (EU), is normally the only way for European research labs and universities to fabricate a few prototype chips to validate the practicability of their ideas. The only available state-of-the-art CMOS process at the time the project was planned was the 0.18 um process from UMC, Taiwan. So this technology has been chosen for the implementation of ATOLL. Most Electronic Design Automation (EDA) tools used for the chip development were also acquired via Europractice. Almost all the tools are developed by Synopsys, Inc. and Cadence Design Systems, Inc., two of the leading EDA companies.

The ATOLL ASIC is a standard cell based design. Using such pre-configured cell libraries, containing logic gates like AND, OR, NAND, NOR, NOT, DFF, etc. in different sizes with several driving strengths, speeds up the design time, since one has not to deal with traditional VLSI design techniques. But the disadvantage is the loss of performance in terms of speed and greater power consumption, since these cell libraries are often very conservative to ensure proper function in silicon. Europractice offers such a library for the UMC process from Virtual Silicon Technologies, Inc. (VST).

1. www.europractice.imec.be

To concentrate on the development of the logic unique to ATOLL, several commonly used building blocks were integrated by using external Intellectual Property (IP) cells. These cells are:

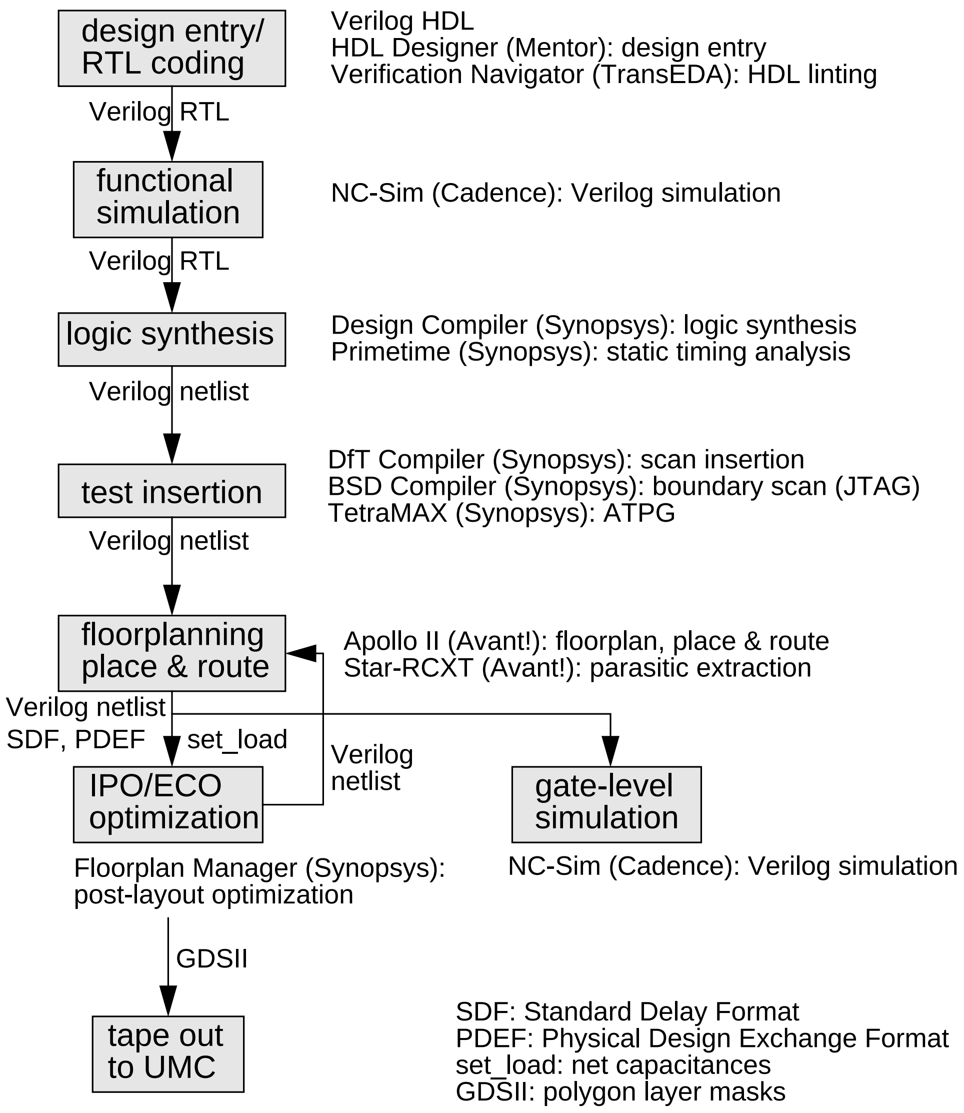
- a PCI-X interface, donated by Synopsys
- DFF-based fifo structures, included in the DesignWare IP library from Synopsys
- RAMs of different sizes, generated by a RAM compiler from VST
- two kinds of PLLs, generated by a PLL compiler from VST
- three special full-custom I/O cells (PCI-X, LVDS-IN, LVDS-OUT), developed by an analog expert team from the University of Kaiserslautern

4.1 Design Flow

The design flow [71] pretty much follows the standard ASIC design flow used over the last decade to design digital ICs. The size of the design and its aggressive target frequency would have been better manageable with some of the new EDA tools targeting high end designs. These new tools merge the logical and physical design steps, providing a faster implementation and more predictable results. Examples are Physical Compiler from Synopsys and Physically Knowledgeable Synthesis (PKS) from Cadence. But these tools were not part of the Europractice tool packages at the start of the implementation phase. Cadence PKS tools have been made available lately, though.

So a design flow had to be established from the accessible tools. Where necessary, additional tools, e.g. for design entry and HDL linting, were acquired to further enhance productivity, if affordable. Figure 4-1 depicts the overall design flow, the tools used for each step and the design formats exchanged between them. Since the design team had almost no knowledge of backend design (placement & routing of standard cells) and to further reduce the workload, it was chosen to draw on the backend service offered by IMEC, Belgium for Europractice customers. Each of the illustrated steps is described in detail in the following sections.

Figure 4-1. ATOLL design flow


design entry/ RTL coding
Verilog HDL
HDL Designer (Mentor): design entry
Verification Navigator (TransEDA): HDL linting
Verilog RTL
functional simulation
NC-Sim (Cadence): Verilog simulation
Verilog RTL
logic synthesis
Design Compiler (Synopsys): logic synthesis
Primetime (Synopsys): static timing analysis
Verilog netlist
test insertion
DfT Compiler (Synopsys): scan insertion
BSD Compiler (Synopsys): boundary scan (JTAG)
TetraMAX (Synopsys): ATPG
Verilog netlist
floorplanning place & route
Apollo II (Avant!): floorplan, place & route
Star-RCXT (Avant!): parasitic extraction
Verilog netlist
SDF, PDEF
set_load
Verilog netlist
IPO/ECO optimization
gate-level simulation
Floorplan Manager (Synopsys): post-layout optimization
NC-Sim (Cadence): Verilog simulation
GDSII
tape out to UMC
SDF: Standard Delay Format
PDEF: Physical Design Exchange Format
set_load: net capacitances
GDSII: polygon layer masks

4.2 Design entry

The whole design has been implemented using the Verilog HDL on Register-Transfer Level (RTL). This is achieved by specifying the behavior of the logic in a cycle-accurate manner. The design process followed a mixed bottom-up, top-down approach. From the top-level of the chip, a hierarchy of modules has been implemented. A step-by-step refinement of module interfaces and the logic inside functional units has given early feedback on the consequences of design decisions and sometimes led to the rearrangement of logic for better performance. Other modules have been designed the same time by merging basic functional units into larger and more complex blocks.

To deal with the large design hierarchy and to enhance its visualization, the decision was made to use a schematic-based design entry tool. After evaluating several alternatives, the HDL Designer Series from Mentor Graphics was chosen. It offers good visualization opportunities, multiple entry formats (schematics, state machines, truth tables, HDL code) and team-based design management. Its RCS-based version management has been utilized to prevent conflicting modifications to the design, as well as to provide the ability to fall back to previous versions of modules.

For an efficient implementation of complex control logic as Finite State Machines (FSM), a custom developed tool called FSMDesigner [72] has been used. Its optimized HDL code generation proved to be a valuable help to deal with complex, hard to implement control logic. The most obvious advantage was to be able to debug control logic at the more abstract level of FSMs, compared to plain HDL code. This provided a fast turnaround time while debugging the design. Regarding the fact that most of the functional bugs were discovered in the control part of a unit, this helped to save weeks of development time. In a late stage of the design, the state machine editor of the HDL Designer Series was used instead, after having figured out how to apply our special implementation style for FSMs to it.

4.2.1 RTL coding

Since the level of expertise in writing Verilog RTL code varied a lot inside the design team, a way had to be found to make sure that all designers produce quality code, which could be easily merged into the whole design. A set of rules and guidelines [73] for writing Verilog code was established, similar to the Reuse Methodology Manual [74], which is widely used in industry. To automate the compliance checking of the code, a special HDL linting tool was acquired, called VN-Check. This tool is part of the larger Verification Navigator (VN) tool suite from TransEDA. A rule database was implemented and each time a designer wanted to check in new code, it was first passed through the linter. This proved to be a fast and efficient way to catch a lot of coding errors, which normally cause problems later in the flow. E.g., it ensured a consistent clocking and reset method, avoided unwanted latches, and forced designers to follow a consistent naming style.

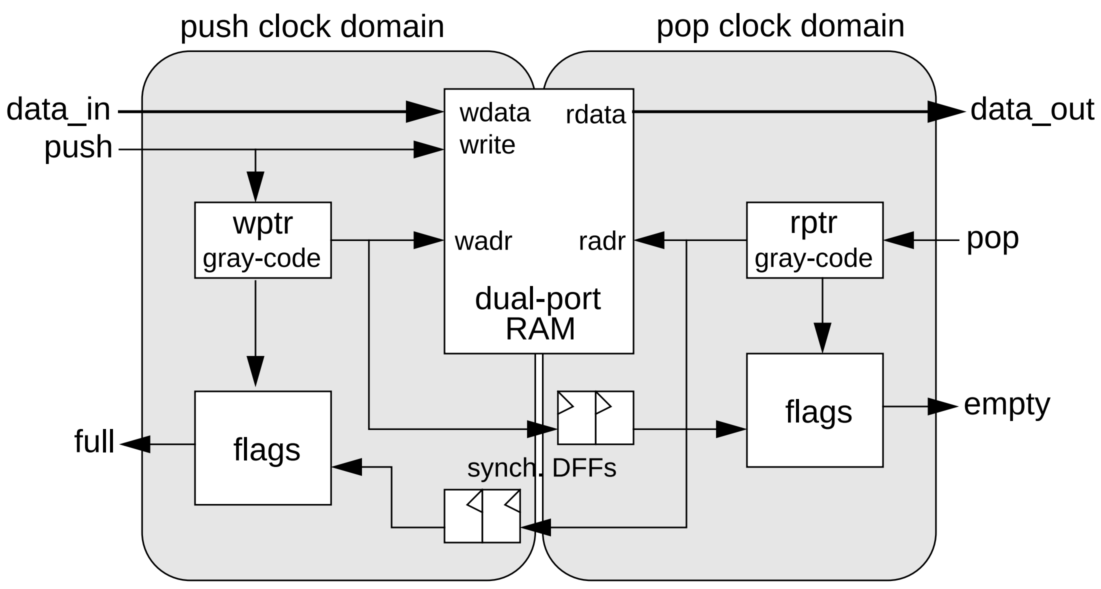
4.2.2 Clock and reset logic

Besides the implementation of the architecture, the clock and reset logic[1] of an ASIC is always a critical factor. The whole chip integrates 6 different clock domains: PCI-X, ATOLL and 4 link domains. The PCI-X clock is generated by logic on the host main

1. designed and implemented by Patrick Schulz for ATOLL

board. Based on the system configuration, it can be set to 33/66/100/133 MHz. An on-chip Delay Locked Loop (DLL)[1] has been implemented to provide a fixed clock tree delay, as requested by the PCI-X specification. The main ATOLL clock is generated by an on-chip oscillator, which uses an external crystal residing on the network card next to the ASIC. It is then internally multiplied by a PLL. The multiply factor is configurable, so one can run the chip from 175 MHz up to 350 MHz. The stepping width for the PLL is 14 MHz. This offers the possibility to tweak the clock frequency of the ASIC up to its physical limit, since the assumptions made during the design process, e.g. bad supply voltage and high temperature, are often far too pessimistic. Finally, to sample data coming over a link from another node, the clock is sent over the link as additional signal line. This is done at half the rate of the original clock to prevent unnecessary noise on the cable. At the receiving side, this clock signal is again doubled, phase-aligned and inverted to sample incoming data into a synchronization fifo. This fifo is then read with the internal ATOLL clock.

Figure 4-2. A dual-clock synchronization fifo



All signals crossing a clock domain border must be synchronized into the receiving clock domain. Sampling an asynchronous signal can result in metastability of flipflops, letting them oscillate for a certain amount of time. The probability of metastability can be reduced by sampling an asynchronous signal multiple times. For modern technologies, a double-sampling is sufficient to ensure proper function of logic.

1. designed and implemented by Prof. Dr. Ulrich Brüning

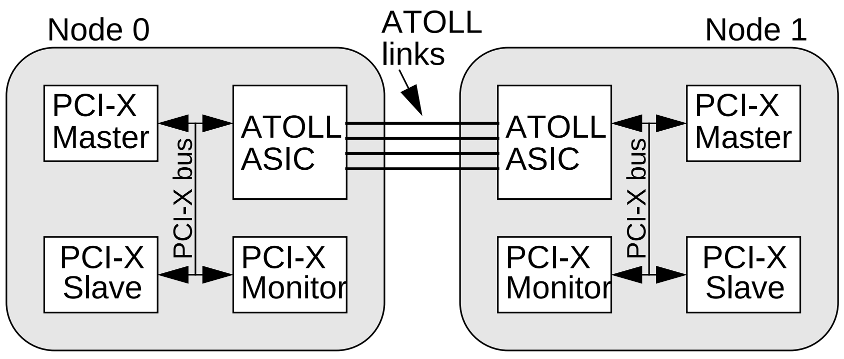
So passing data or control signals across a clock domain border is simply done by using two flipflops in a pipelined fashion. Where it is necessary to pass signals in opposite directions, e.g. the two-way handshake signals, a dual-clock fifo structure is utilized to safely transfer data. Some things [75] have to be observed while designing such a dual-clock fifo. Basically, the push and pop interfaces are driven by different clocks. Data is stored in a pointer-controlled RAM, with the write pointer residing in the push interface, and the read pointer controlled by the pop interface. Internally, these pointers are then synchronized to the opposite interface to calculate the fifo fill level and to generate the appropriate control flags (full, empty, etc.). Using normal binary-coded pointers could result in a failure, since sampling a value which is just incremented from 0111 to 1000 could result in any possible bit combination. This is avoided by using a gray-code for the pointers, allowing only one bit to change on each transition. This could result in sampling an old data, but not a completely wrong one. Figure 4-2 depicts the structure of such a dual-clock fifo.

4.3 Functional simulation

Regarding the size of the design, simulation runtime was a major issue. Therefore, prior to running any simulations, a benchmark was set up to compare the performance of all simulators accessible: Verilog XL, NC-Sim (both Cadence), VCS (Synopsys), and Modelsim (Mentor). The compiling simulators NC-Sim and VCS clearly dominated the interpreting XL and Modelsim. Since more licenses were available for NC-Sim, it was chosen as main simulator for both RTL and gate-level simulations.

4.3.1 Simulation testbed

Figure 4-3. Testbed for the ATOLL ASIC



To test the ATOLL ASIC in an environment as close to its real use as possible, a testbed[1] has been set up to simulate a network of 2 PCs connected by the ATOLL network, as

1. implemented with the help of Patrick Schulz

shown in Figure 4-3. The two ATOLL ASICs are connected back-to-back by their 4 links. On the host side, Synopsys PCI-X FlexModels were used. These are a set of bus functional models (BFM), configured to act as the central components of a PC node:

- a Master BFM is used to model the CPU
- a Slave BFM is used to model main memory
- a Monitor BFM is used to check all ongoing bus transactions for PCI-X compliance

Several Verilog tasks have been developed to control the simulation via the Master BFM. They also encapsulate all calls to the FlexModel BFMs, so higher-level testbenches do not have to deal with the control of single PCI-X bus cycles. Some examples for such tasks are:

- atoll_init initializes the ATOLL ASIC, sets up data structures in main memory
- send_dma: assembles message data in main memory, enqueues a new send descriptor into the descriptor table, and triggers the DMA engine inside the ATOLL ASIC to process the currently generated DMA job
- receive_pio: receives a message by Programmed I/O

Summarizing, 22 Verilog tasks with more than 2.000 lines of code have been implemented. Using these basic tasks as a kind of "testbench API", several top-level testbenches have been developed. They are used to test the implementation for correctness and to sort out all functional bugs. The implemented tasks could serve as a starting point for the development of a low-level ATOLL message layer. They issue a sequence of load/store operations to control the various features for message transfer, very similar to the algorithms software would need to implement.

4.3.2 Verification strategy

A specific verification strategy was chosen to be followed throughout the verification process to coordinate all simulation efforts. This method [76] is called 'shotgun & sniper rifle', according to its dual-way approach. A set of specific corner-case testbenches is used to test all the critical issues the designers can think of ('sniper shots'). These testbenches are small and use fixed parameters to call the basic tasks. On the other side, some testbenches use a large sequence of tasks with random parameters (e.g., message size, link to send data on, DMA or PIO mode, etc.) to cover large portions of the verification space ('shotgun'). In total, 11 'sniper' testbenches and one large parameterized 'shotgun' testbench were implemented, together about 15.000 lines of code.

4.3.3 Runtime issues

Memory usage of the simulator is surprisingly low, the compiled design uses only 63 Mbyte in total. But it has been a problem after starting to write dumps for analysis purposes. When dumping the whole design, the code inflates to more than 1 Gbyte, causing the machine to swap memory pages. This slows down the simulation significantly. Dumping has been limited to the first levels of hierarchy to overcome this performance drop. While debugging errors buried deeply in the design hierarchy, a second run of the erroneous testbench has recorded only the events from some specific modules.

The small corner-case testbenches run only for some minutes. But the large regression testbenches simulate sending/receiving thousands of messages of random size via a random combination of network interfaces and links. They run for days, and even when limiting the dump to 3-5 levels of hierarchy, the dump files still grow to several Gbyte after a day. So the decision has been made to turn off dumping at all for these testbenches to run them as fast as possible.

The ability to write checkpoints of the simulation out to disk to be able to restart it from a point just before an error occurred would have been very helpful. But unfortunately, the FlexModels from Synopsys can not be restarted from a checkpoint written by NC-Sim. So one has to rerun the whole testbench to write a dump containing the occurred error, even if it happens after several days. Since this methodology would have disrupted the time schedule of the whole project, a workaround has been found in stopping the regression test after two days. The testbench is then started over and over again with modified random numbers (Verilog random numbers are semi-random, the same simulation will always generate the same sequence of random numbers).

Though one run of such a regression testbench runs for 48 hours, it simulates only about 110 ms of real time. About 50.000 messages are sent in such a run, with a data payload from a few bytes up to 4 Kbyte. The small ‘sniper shot’ testbenches caught about 80 % of all functional bugs found. Once they ran without errors, the regression tests still found some bugs, but the error rate dropped rapidly in the first weeks of simulation.

After several (about 15) runs of the large ‘shotgun’ testbenches completed without errors, the decision was made to declare the design to be stable enough to be implemented in silicon. Table 4-1 shows some simulation statistics.

Table 4-1. Testbench statistics

	'sniper shot'	'shotgun'
runtime	10-30 min	48 h
no. of simulated messages	250	50.000
started after ...	every design modification	all the time
no. of total runs	60-80	50-60

4.4 Logic synthesis

Setting up a synthesis flow for such a large design is a non-trivial task. Since the whole design is too large to be synthesized in one top-down compile, the design had to be broken down in smaller parts using a "Divide & Conquer" approach. Several methods are well known for using Design Compiler from Synopsys for such large designs, sometimes referred to as bottom-up or compile-characterize-write script-recompile flow.

4.4.1 Automated synthesis flow

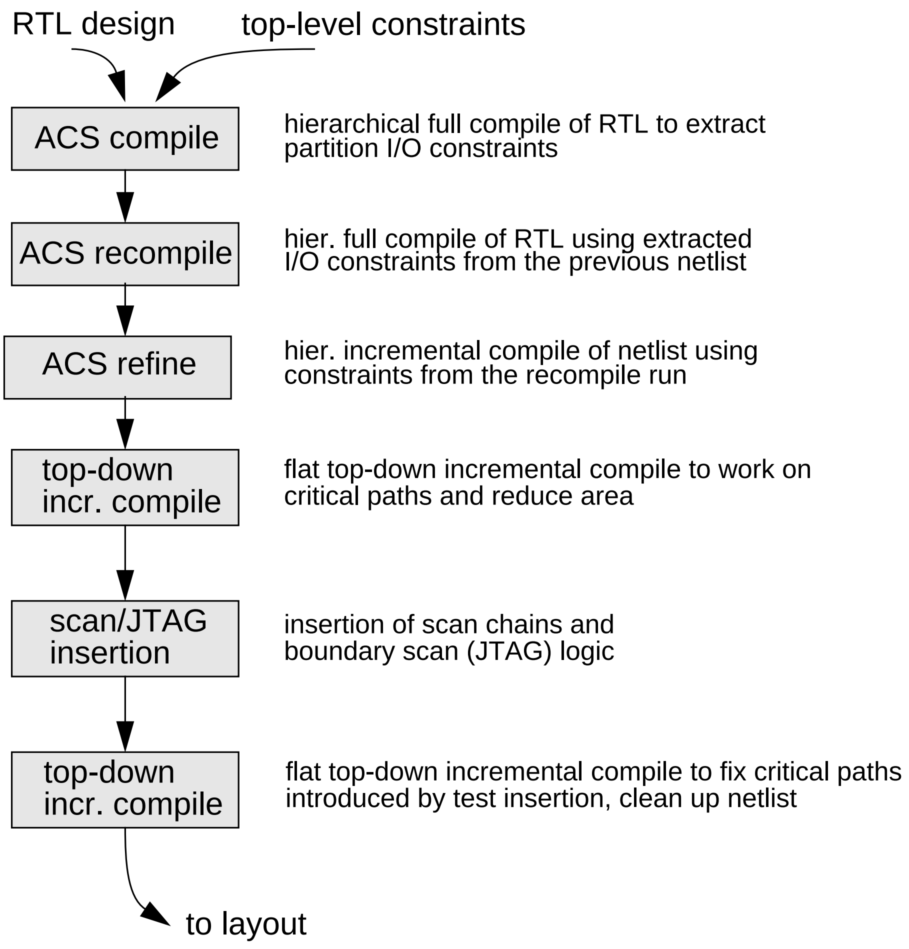
All these mixed top-down/bottom-up flows require a lot of scripting and data management to efficiently constrain and compile subdesigns, which are then glued together into higher-level modules. Every major ASIC company has set up its own flow based on preconfigured scripts and custom compile strategies. In 1998, Synopsys introduced a TCL command line mode for their tools, significantly enhancing the ability to implement custom procedures and functions. This enhancement and the need for a stable and easy-to-use synthesis flow resulted in the release of a feature called Automated Chip Synthesis (ACS) [77]. ACS is a set of TCL procedures intended to automate the bottom-up compilation of large designs. It automatically partitions the design, propagates constraints down the hierarchy, writes out compile scripts for each partition, and finally generates a Makefile to control the whole synthesis flow of the design. It reduces the setup of the flow to specifying only top-level constraints and a few scripts to drive the ACS flow. Recent benchmarks [78] show that ACS produces good results in terms of timing and area compared to other methods. The fact that it is based on TCL procedures makes it highly configurable. That proved to be a major advantage, as it was necessary to patch one ACS procedure to work around a serious bug.

The 3 main procedures of ACS are:

- acs_compile_design: does a hierarchical compile of the design using user-specified top-level constraints

- acs_recompile_design: extracts timing constraints from a previous ACS run, and uses them to run a full compile on the RTL design
- acs_refine_design: extracts timing constraints from a previous ACS run, and uses them to run an incremental compile on the current netlist

Figure 4-4. Synthesis flow


RTL design
top-level constraints
ACS compile
hierarchical full compile of RTL to extract partition I/O constraints
ACS recompile
hier. full compile of RTL using extracted I/O constraints from the previous netlist
ACS refine
hier. incremental compile of netlist using constraints from the recompile run
top-down incr. compile
flat top-down incremental compile to work on critical paths and reduce area
scan/JTAG insertion
insertion of scan chains and boundary scan (JTAG) logic
top-down incr. compile
flat top-down incremental compile to fix critical paths introduced by test insertion, clean up netlist
to layout

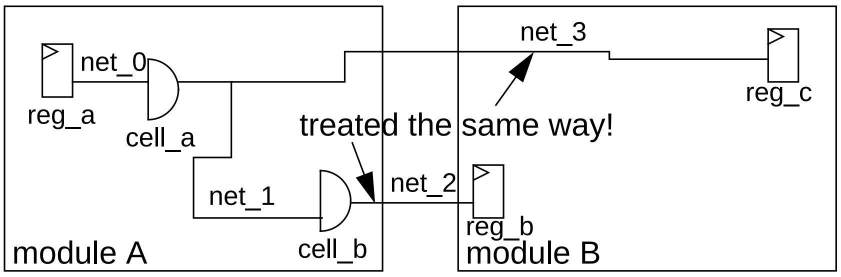
Those three commands were used in the above order to establish a base netlist for further refinements. Top-level compiles were then used to clean up the netlist, globally optimize remaining critical paths and prepare the netlist for layout. Figure 4-4 depicts the overall synthesis flow used for the ATOLL ASIC.

4.4.2 Timing closure

At the beginning, the design was pad-limited (around 380 functional I/O pads), which means that the area of the chip is determined by the area needed to place all I/O cells, not by the size of the core logic. So area was not a major constraint during synthesis of the core logic. Timing was more critical, especially in the PCI-X clock domain. Very early in the design flow some basic compile runs on parts of the RTL design were used to check, if any parts would be a major problem regarding timing closure. Based on these runs, a lot of Verilog modules have been rewritten. E.g. pipeline stages were inserted, and large functional blocks were broken down into several smaller ones. This proved to be a big advantage for the further work, since once started with the whole synthesis flow, it was never necessary to go back to RTL coding.

Traditional logic synthesis tools know the logical structure of a design, and have fairly detailed information about the timing of standard cells. They lack any information about the physical implementation of a design, since this is done at a later stage of the whole flow. To calculate the delay of logic paths and find an optimal netlist composed of interconnected standard cells, tools need to estimate the effect of nets in terms of length and delay introduced by them. Library vendors provide with each cell library a set of wire load models. A wire load model is a way to predict the load, and with it the delay, of nets based on a statistical evaluation of previously implemented designs using the same technology. But since designs might differ a lot, these estimations can be very imprecise. This was no major concern with older technologies when cell delays dominated the wire delays. But with technologies of 0.18 um or even smaller geometries, this proportion has changed dramatically. The result is that the physical implementation has a huge impact on the overall performance of a design.

Figure 4-5. Logic synthesis lacks physical information



Wire load models are often a set of tables containing net lengths/delays, indexed by the fanout (the number of cells driven by the net) of the net. Different tables are provided for modules of different size. So a synthesis tool treats nets of the same level of hierarchy the same way. Assuming a placed design composed of two modules, as shown in Figure 4-5, this can lead to mispredicted net lengths:

- though net_0 and net_1 are part of the same module, their lengths differ a lot
- also net_2 and net_3 both run between both modules, but the cells connected to them are placed very differently

These inaccuracies get even worse the larger the design is. To enhance the accuracy, one can generate so-called custom wire load models, which are specific to the design implemented. They were generated for ATOLL from a trial layout. But it turned out that even these custom wire load models still were inexact. Dealing with very high-fanout nets [79] was another crucial issue.

Figure 4-6. Improvement of timing slack and cell area

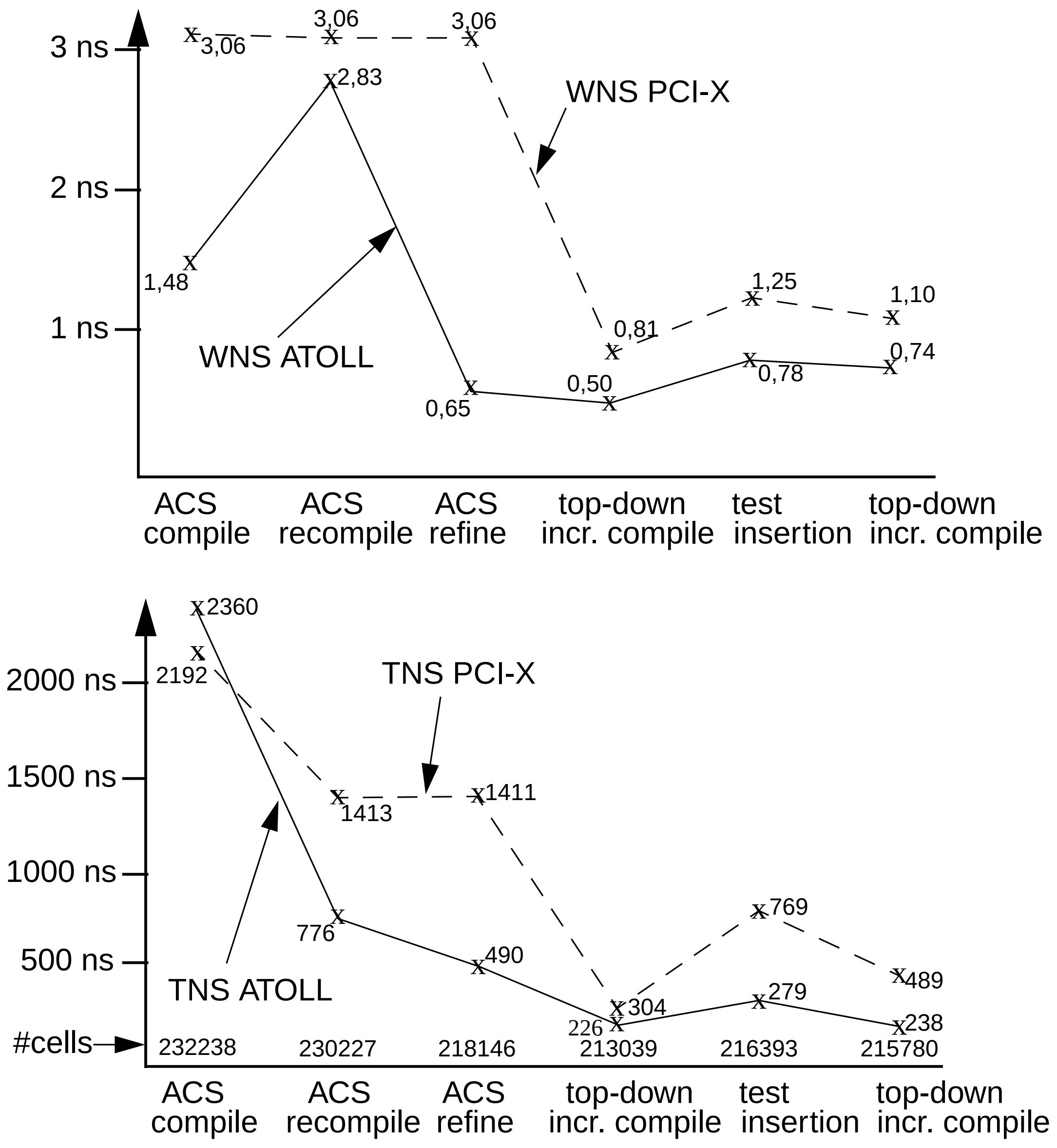


Figure 4-6 shows the timing and area results after each step of the synthesis flow. The numbers are given separately for each of the two major clock domains, PCI-X and ATOLL. The upper diagram depicts the Worst Negative Slack (WNS), which is the logic path with the largest timing violation. The lower one shows the Total Negative Slack

(TNS), which is the sum of all violated paths. Additionally, it lists the number of standard cells used to implement the design as a gate-level netlist.

The WNS in the PCI-X domain remains stable over the 3 ACS runs, since the PCI-X core is precompiled via encrypted synthesis scripts. These scripts are delivered by Synopsys together with the encrypted Verilog source code. The resulting netlist is read in at start and protected against modification until the first top-down incremental compile. However, the TNS in the PCI-X domain shrinks a bit, because the synchronization logic interfacing the PCI-X core to the rest of the chip lies outside the protected PCI-X core.

WNS and TNS of the ATOLL domain are significantly reduced during the ACS runs, with a temporary larger WNS for the second run. This is a consequence of too pessimistic constraints extracted from the first trial, misleading the recompile step. The first top-down incremental compile then improves timing of the PCI-X part a lot, whereas most optimizations possible for the ATOLL domain seem to have been done by the ACS runs. Test insertion than adds some slack, mostly because of inserting the JTAG boundary scan logic into paths to/from I/O pads. These paths are critical, especially paths through the PCI-X pads. The number of cells shrinks significantly with the ACS refine step and the first incremental compile on the gate-level netlist. The last steps only make local optimizations with little impact on overall cell count.

Though timing and area improve a lot over the whole flow, there are still some violated paths in the netlist at the end. This is caused by very strict and pessimistic constraints on some parts of the design. An early version of the flow finished with no slack at all, but was too optimistic regarding net lengths and load capacitances at some places. So the post-layout timings differed a lot from the pre-layout numbers, which caused post-layout optimization to fail. The constraints and also the wire load models were then tightened for some critical modules. Most of the violated paths run through I/O pads. Since they are on top-level, the tool wrongly estimated a long net between the pad and the core logic. But since these nets are quite short after layout, the pre-layout slack was accepted.

Another interesting fact is that the TNS of the PCI-X domain at the end is still twice the TNS of the ATOLL domain, though the PCI-X part is only about 10 % of the whole design. The reason for this unbalanced ratio is the fact that the PCI-X part includes a lot of tightly constraint I/O pads, which are still violated after synthesis. On the other hand, the ATOLL core logic is connected to LVDS pads with less stringent constraints, so fewer violations occur in it.

4.4.3 Design for testability

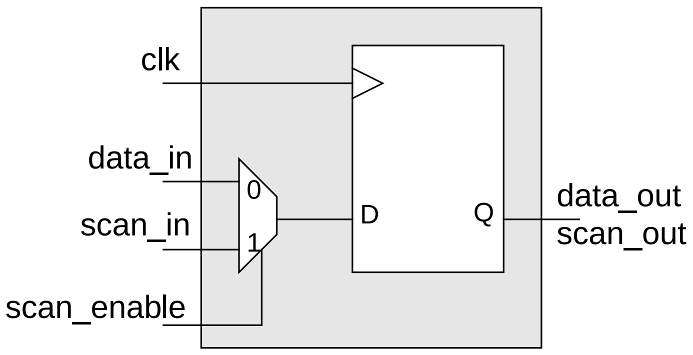
Several factors can cause errors in ASIC production. The amount of fully functional chips, compared to the total number of dies, is called yield and typically varies between 40-80 % of the whole production volume. To efficiently sort out the good from the bad chips one adds on-chip test logic, which is utilized to test silicon dies before they are packaged and mounted on Printed Circuit Boards (PCB). Separate tests are used to isolate errors in chip packages and PCBs. The whole area is often referred to as Design for Testability (DfT), and its whole variety of methods and applications is beyond the scope of this document. An in-depth discussion of the DfT methods[1] used for ATOLL can be looked up in additional literature [80], this section will give only a broad overview.

ATOLL contains three types of test logic:

- full scan using multiplexed flipflop scan style
- JTAG-compliant boundary scan
- Built-In Self Test (BIST) for all RAMs

Full scan means that all registers in the design are replaced with scannable DFFs. These can be switched via a scan_enable signal into scan mode, forming a large chain of scan flipflops. Figure 4-7 depicts the internal structure of such a scan DFF.

Figure 4-7. Multiplexed flipflop scan style



ATOLL contains 4 full scan chains, which are partitioned according to the different clock domains. Only the flipflops driven by the 4 link clocks are assembled in one chain, since each link clock domain only drives about 160 DFFs. One chain is used for the PCI-X clock domain, and the other 2 scan chains link up all registers of the main ATOLL clock.

1. implemented by Patrick Schulz

Start- and endpoint of these chains are some general purpose I/O cells, which are not timing critical and can tolerate some additional load. Table 4-2 lists all full scan chains.

Table 4-2. Internal scan chains

scan chain	clock domain	number of DFFs
0: GP_IO[7] -> GP_IO[3]	ATOLL	20.965
1: GP_IO[6] -> GP_IO[2]	ATOLL	20.965
2: GP_IO[5] -> GP_IO[1]	PCI-X	4.617
3: GP_IO[4] -> GP_IO[0]	4 link clocks	660

Not all DFFs can be linked up in the scan chains, e.g. if they are driven by a noncontrollable clock. But these are only a few, e.g. the DFFs in the test and clock logic. In total, about 98.5 % of all DFFs are scanned. The unbalanced lengths of the scan chains affects the time needed for testing. But this can be accepted due to the low production volume targeted for the ATOLL chip.

Boundary scan logic [81] is used to drive specific values out of the chip via the I/O cells. This is utilized for board-level tests, using a standardized JTAG interface. Internal logic, called Test Access Point (TAP) controller, provides the signals used to control the boundary scan logic. The boundary scan cells are located between the I/O cell ring and the core logic. JTAG is quite popular for this task, since it only needs 5 additional I/O pads to shift data in and out of the chip. Besides the described task, it is also used in ATOLL to control the BIST logic.

The internal BIST logic [82][1] is used to test all 43 instantiated RAM macros in the ATOLL chip. It uses a 12-N-March algorithm to repeatedly write 0’s and 1’s to each RAM cell to detect any faulty bit cells. As stated earlier, it is fully controllable and observable through either the JTAG TAP controller or supervisor registers. Normally, the BIST is run together with the board-level test, just after board production. The software-controlled BIST is intended for checking boards, which show unstable behavior while in use.

4.5 Layout generation

As mentioned earlier, the IMEC IC Backend Service group was hired to do the layout job. Place & Route of a design is a difficult and crucial step in the design flow and needs a lot of experience and knowledge about the tools and the technology. Since this know-how was not present in the local design group, it was not feasible to learn this task in an acceptable amount of time, without further extending the time frame of the project.

1. implemented by Erich Krause

Layout generation

In addition to handing over the gate-level netlist to the backend team for layout preparation, some more data is necessary to ensure an efficient and optimal implementation. Since the ATOLL design has some aggressive timing goals, the layout flow was carefully planned. A smooth and close collaboration of the layout step with synthesis is an important factor to keep the amount of post-layout optimization steps at a minimum. From discussions with other ASIC designers who did timing-critical chips and various literature about high-end ASIC design it was figured out that three things are crucial for reaching timing closure:

- a well considered floorplan[1] for placing macros and top-level blocks
- using custom wire load models for precise prediction of wire delays during synthesis
- a timing-driven layout flow to optimize critical paths during cell placement and wire routing

Figure 4-8. Floorplan used for the ATOLL ASIC

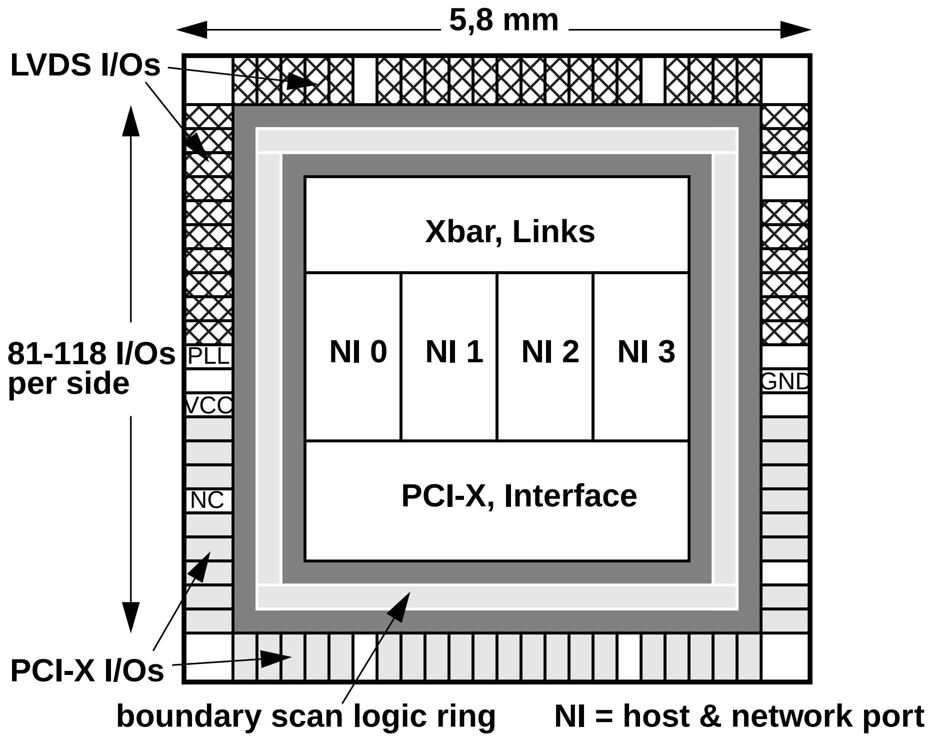


Figure 4-8 depicts the floorplan used for the ATOLL ASIC. But it needed some time to convince the layout team at IMEC to use it. The default method used for most of their customers is to preplace only the RAM macros, and let the tool place all cells freely. After

1. designed with the help of Prof. Dr. Ulrich Brüning

analyzing first layout trials it became clear that this default methodology may work for the typical small- to -medium-sized designs done at IMEC, which have modest timing requirements. But it is not suited for the kind of timing-critical and large designs like the ATOLL chip. Since the top level of the chip is well structured, it really paid off to bring in the knowledge about the logical structure of the design. The core area was separated into the 6 regions shown in the floorplan. During placement, cells contained in the logical structure associated with these regions were allowed to be placed only within their region, with allowing only some small exceptions to avoid highly congested areas. All 43 RAM macros were preplaced according to the floorplan, along with some other critical cells. E.g., the DFFs driving the LVDS outputs were preplaced, making sure that the skew between single bits of the same link is as low as possible.

The usage of custom wire load models proved to be an advantage, as described earlier. But still after several runs to finetune them, there were parts of the design, where pre- and post-layout net delays varied significantly. This inaccuracy is an inherent problem of the use of wire load models to estimate net lengths, and it gets worse the larger a design is. However, a lot of violated paths could be optimized by buffer insertion or cell sizing during post-layout optimization.

To overcome the gap between constraint-driven logic synthesis and physical layout, newer versions of Place & Route tools offer the possibility to bring in some knowledge about the timing of a design. A timing-aware placement of cells then greatly enhances the performance of the layout, since the tool focuses on the optimal layout of critical paths. Unfortunately, the backend team could not use this type of flow for the ATOLL chip. Trying to use a timing-driven placement, the tool constantly crashed and never finished. So one had to fall back to the conventional wire length-driven flow, which optimizes net lengths between all cells, whether they are part of a critical path or not. This was the main cause for the large amount of post-layout optimization steps.

4.5.1 Post-layout optimization

After the design is placed and routed, one needs to analyze the timing of the layout, since it might vary a lot from the netlist version produced by synthesis. Several standardized data formats exist to hand over data between different design tools. The following data was delivered by the backend team for timing analysis and post-layout optimization:

- the gate-level netlist, now containing buffer trees for all clock and reset nets
- extracted point-to-point timing of cells in the Standard Delay Format (SDF)

- extracted capacitive load information of nets (set_load commands for the synthesis tool)
- physical cell locations in the Physical Design Exchange Format (PDEF)

This data is fed back into the synthesis tool, which is used to run a so-called In-Place Optimization (IPO) or Engineering Change Order (ECO). Since the synthesis tool is now aware of the real cell and net delays, it can calculate the exact timing of each path and try to optimize the ones that do not meet the timing goal. The following methods are used to do this:

- cell upsizing is used to enhance the driving capability of a gate by replacing a cell with the same logic function, but a higher driving strength. This speeds up the path, but also increases area
- cell downsizing is done by replacing a cell with a lower drive version of it. This can be useful to reduce the load of nets driving this cell, or to save power
- buffer insertion is used to break up long nets, resulting in reduced load and faster transition times for driving cells
- buffer removal is done when synthesis has added too many buffers to a path, from which some are unnecessary

Figure 4-9. Improving a timing-violated path

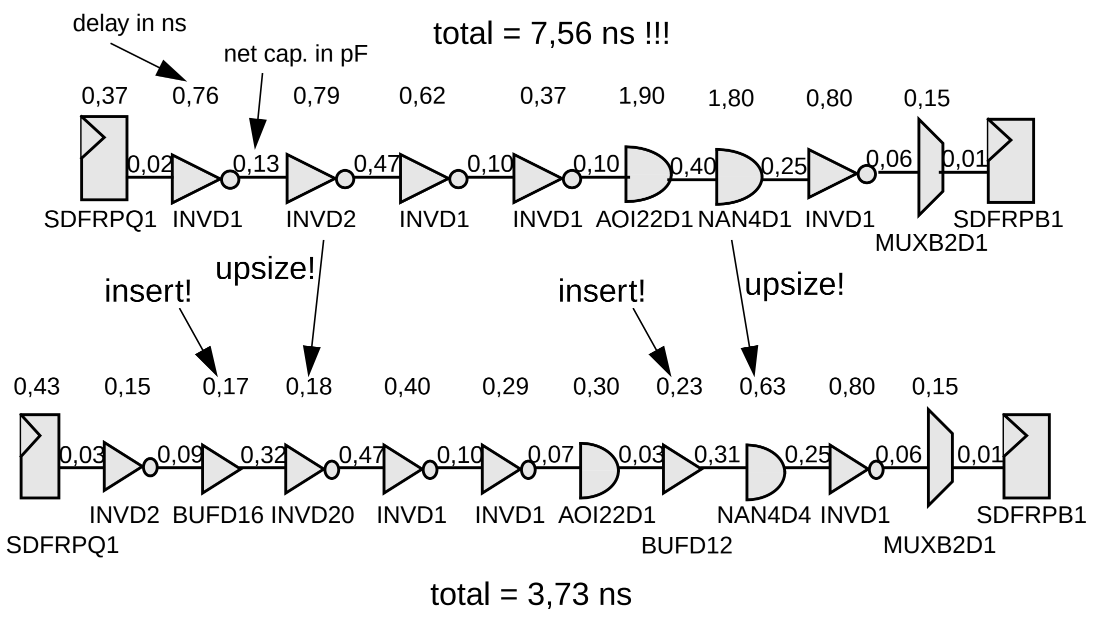
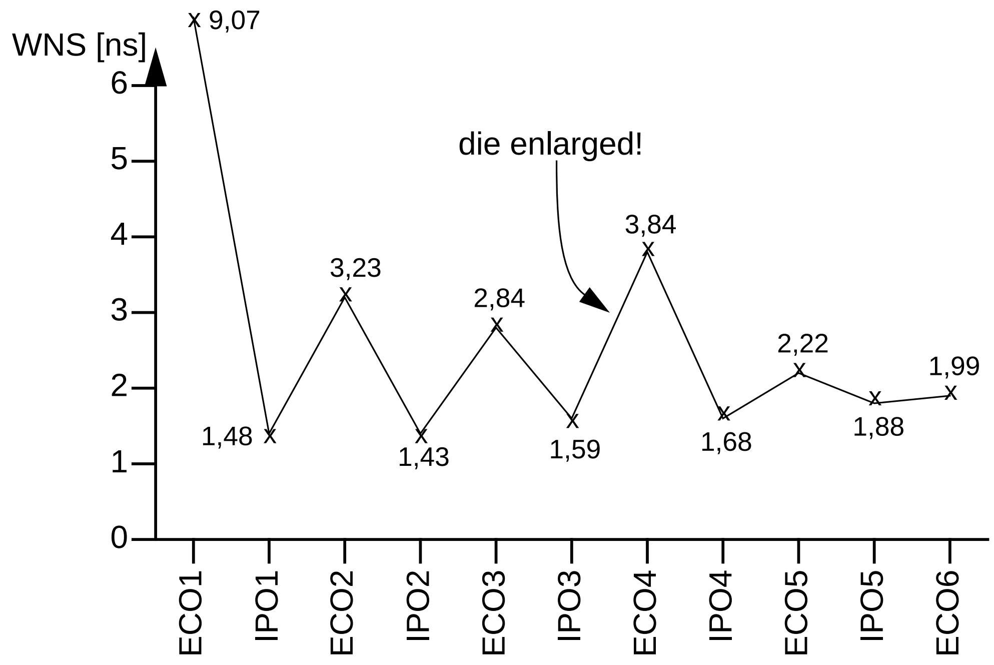
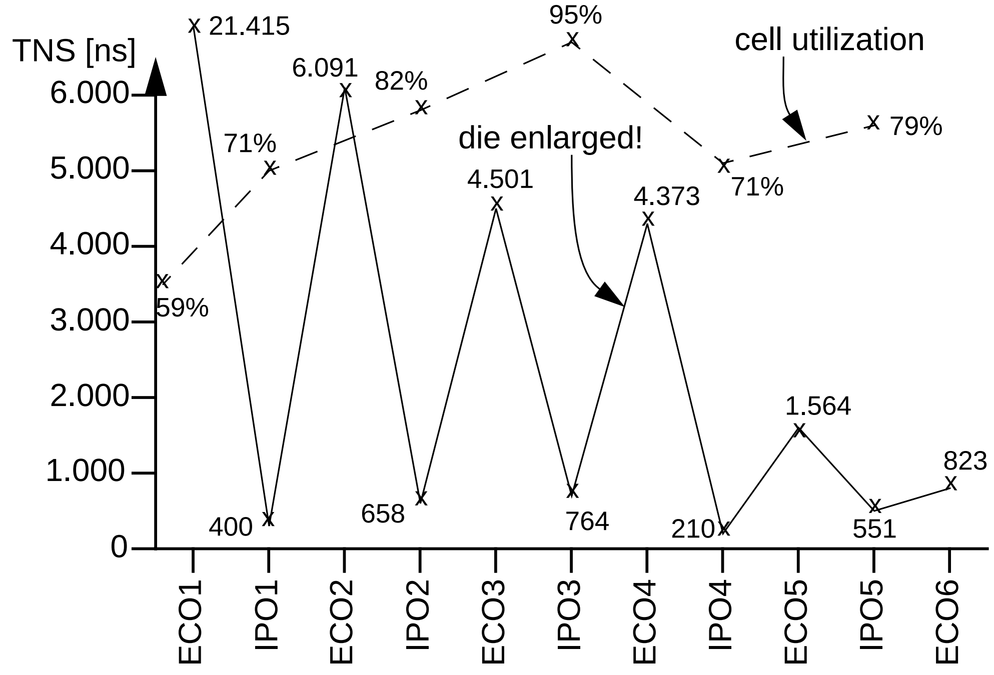


Figure 4-9 shows an example of how the timing delay of a path can be halved by these modifications. E.g. some mispredictions lead to excessive cell delays of nearly 2 ns. Upsizing those cells or splitting huge net loads can significantly speed up the logic. Especially during the first IPO iterations such drastically improvements were made to parts of the design. In case of buffer insertion, the synthesis tool does not rely on wire load models to predict the length of newly created nets. Since cell locations are known from the PDEF information, a basic routing algorithm is used to calculate the net length based on the distance between both cells.

Figure 4-10. Timing optimization during IPO/ECO





After all possible optimizations are done, the new gate-level netlist is again transferred to the layout tool. An ECO step then compares the old and the new version of the netlist, making the necessary changes. During this process it might be necessary to move upsized cells, since their area increased. This has again side effects on all surrounding logic, leading to different net lengths and loads. So timing data can again vary between the results of the IPO during synthesis and after the ECO done bye the layout tool. But normally the difference shrinks and the timing converges towards the goal.

All in all, 6 post-layout iterations were done for the ATOLL ASIC. Figure 4-10 shows, how the global WNS and TNS improve from one optimization step to the next. An ECO result refers to the timing after updating the layout, whereas IPO refers to the numbers achieved after running an optimization in the synthesis tool. According to Figure 4-6, the first layout is done on a slightly violated netlist with 1,10 ns WNS and 700 ns TNS. Numbers after the first layout called ECO1 are much worse. The WNS increases by a factor of 9, and the TNS even by 30. During all following iterations, the timing bounces up and down between ECO and IPO runs.

The first 3 iterations reduce the timing slack a lot, but at the expense of area. In the lower part of the diagram, the cell utilization is given. The core area of an ASIC is subdivided into rows of cell slots, and the ratio of occupied vs. total slots is referred to as cell utilization. The higher this value, the more difficult is the task of a layout tool to run an ECO, since it is very limited in its decisions where to place and move cells and where to route nets. This effect is visualized by the relative small improvements made from ECO2 to ECO3. After running IPO3 in the synthesis tool, the cell utilization had grown to 95 %. The layout tool then refused to run an ECO on the netlist delivered by IPO3, since a few areas of the chip were so heavily congested, that no more nets could be routed through them.

The decision was made to enlarge the die by adding 12 ‘not connected’ (NC) dummy I/O cells to each side of the I/O ring. Since an ECO can only be run on a fixed core area, a full Place & Route had to be done, referred to as ECO4 in the diagram. Cell utilization dropped below 70 %, but on the other side all cells were newly placed, resulting in some more timing slack. But the remaining iterations shrinked the TNS below 1.000 ns. Some paths remained violated and could not be optimized to meet their timing goal.

An in-depth analysis showed, that most of these violated timing paths run through the PCI-X I/O cells. Some of the boundary scan logic was replaced during the die enlargement in a bad way, resulting in some very long paths. These paths were optimized a lot by buffer insertion and excessive cell upsizing, but about 20 paths still have a slack of 1-

2 ns. The worst slack in the ATOLL clock domain was about 1.4 ns, with only a handful of paths above 0.5 ns slack.

Since the deadline for the design submission to fabrication was reached after running ECO6, the decision was made to accept the remaining slack and go into production. This decision is backed by two facts:

- the ATOLL core clock is configurable, so it can be tailored to the real capabilities of the chips
- all design steps were done based on worst-case technology data. In terms of the cell library used for ATOLL this means 125 C temperature, 1.62 V core voltage (instead of 1.8 V nominal) and a bad process factor. In reality, things are not that bad. Assuming real world conditions (70-80 C, 1.8 V), timing should be about 20 % better than estimated by the Static Timing Analysis (STA) tools

4.5.2 Post-layout simulation

A simulation of the gate-level netlist with annotated SDF cell delays was executed to ensure that no logic got lost somewhere in the design flow, as happened at one point during the ACS synthesis runs. Another issue was the validation of the critical clock and reset logic, e.g. the PCI-X DLL, which includes a chain of delay elements. The same testbenches as for RTL simulation were used, just replacing the RTL design with the gate-level netlist. However, a few modifications were needed to get the simulation up and running. E.g., all DFFs are annotated with setup/hold timing checks. This is a problem for DFFs on clock domain borders, since the incoming asynchronous data will trigger setup/hold violations during simulation. Resulting metastable data outputs are suppressed by double-sampling these signals, as described earlier. But the simulation models do not reflect this behavior. They propagate an ‘x’ (unknown) value, causing the whole simulation to fail. This problem was solved by extracting a list of the affected DFFs during synthesis. A Perl script then was used to find the related entries in the SDF data and to disable the relevant timing checks.

Early versions of the netlist, which still had some timing violations, were used at a reduced clock speed to run one of the large regression testbenches. Of course, it was significantly slower than the RTL simulation. But the testbench ran for nearly two days without failure.

5 Performance Evaluation

Besides the aggressive scale of integration, implementing a high performance network was the primary goal of the ATOLL development. Two types of metrics are important in the field of Cluster Computing. Message latency measures the time needed to send data from one node of the cluster to another one. It lies in the range of a few microseconds for modern networks and is the dominant performance metric for applications with a fine-grain communication behavior. If a user application tends to exchange only a few bytes, but this at a fast rate, the network should not slow down the program.

The second important factor is the sustained bandwidth a network can provide. It measures the actual data rate on a network link, compared to the physical bandwidth limit. A well developed network should come very close to the physical limit, proving to make efficient use of the network resources. Bandwidth is usually measured by setting up a continuous data stream with a fixed message size, sending the same message hundreds or even thousands of times. Summing up the total amount of data sent and dividing it by the time needed erases start-up effects and is the preferred method to measure the so-called sustained bandwidth of a network.

Those two metrics are usually measured on the application level. So no distinction is made between the performance of software and hardware. And usually the network hardware accounts for only a small part of the number, especially regarding latency. E.g. a typical parallel cluster application linked to the MPI message passing library may call a MPI_send function. This function implements only a high-level MPI layer and calls a mid-level point-to-point abstraction layer. This again can call a low-level ATOLL hardware layer. The system architecture of a cluster is also an important factor. The CPU bus interface, the memory controller and the I/O bridge can have a great impact on the performance of the whole system.

Since the performance of an ATOLL-equipped cluster can only be measured on a real system, this chapter focuses on measuring the performance of only the network part. No comparisons are made to other network solutions, since it would be unfair to compare ATOLL's network-only numbers to full application-level numbers of other networks. Hardware-only performance values have not been published for other implementations.

Some performance measurements regarding the ATOLL software can be looked up in the dissertation of Mathias Waack [83].

All presented performance measurements were derived from simulations of the environment specified in “Simulation testbed” on page 112. Numbers are given first for a single host port in use. But since the multiple-interface architecture is a defining feature of ATOLL, both metrics are also given for two and all four host ports in parallel use. This provides an insight into the applicability of the ATOLL network for clustering dual- or quad-CPU machines.

5.1 Latency

The latency is measured both for PIO- and DMA-based message transfer. Since it usually scales linearly with an increasing message size, it is only quantified for relative small messages. Latency is measured from the first access to the ATOLL device on the sending node until the last data transfer on the receiving side. For the PIO mode this means to start timing the transfer, when the first routing word is written to the PIO send fifo. The transfer is complete, when the last data word is read from the PIO receive fifo. Regarding the DMA mode, the measurement begins with triggering the DMA engine inside the host port by updating the relevant descriptor table pointer. And it ends with the receiving host port updating the relevant pointers in the replication area. This means that the assembling of messages in the data structures residing in main memory is not included in the timed process. Since the copying of message data into and out of the buffers in memory can be fully overlapped with the send/receive operations controlled by the ATOLL device, this is a proper reduction of the latency measurement onto the crucial part of the whole operation.

Figure 5-1. Latency for a single host port in use

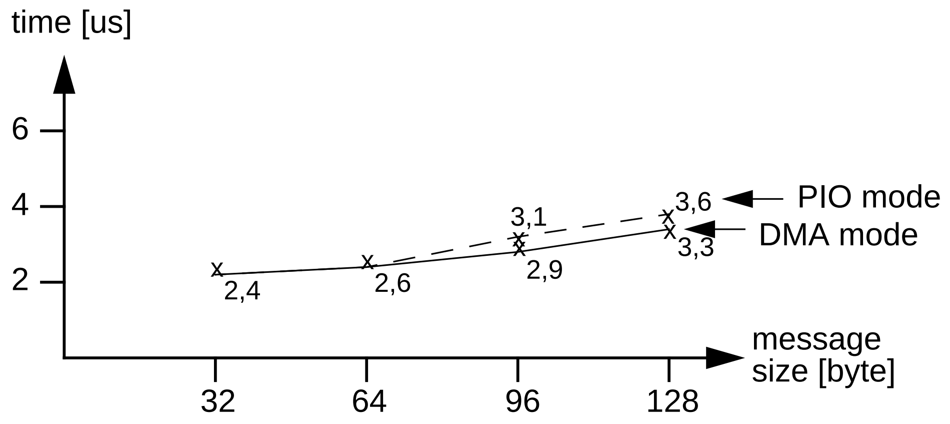


Figure 5-1 depicts the one-way latency for a single host port in use. Latency starts at about 2,4 us for a message size of 32 byte. It then slightly increases with the size. Surprisingly, the difference between PIO and DMA mode is almost negligible. At first glance, one could assume that the PIO mode would be slightly faster, since it directly writes data into the network on the sending side. Reading data from memory in DMA mode has some more latency. But this is outweighed by the need to break up the PIO mode send operation into 6 different PCI-X cycles, according to the PIO send address layout (3 frames, 3 additional accesses for each last word of a frame). Some additional latency is added also on the receiving side, since the complete reception of a message is not immediately notified to the host CPU. The CPU polls the fifo fill level with a certain frequency, about one access each 0,3 us.

On the other hand, DMA mode needs only 3 PCI-X read transactions on the sending side (descriptor, routing, header & data combined). Once data enters the ATOLL chip, it is forwarded quite fast towards the network. E.g. it needs about 0,24 us for the first byte of a message from the PCI-X bus to show up on the network link.

Figure 5-2. Latency for multiple host ports in use

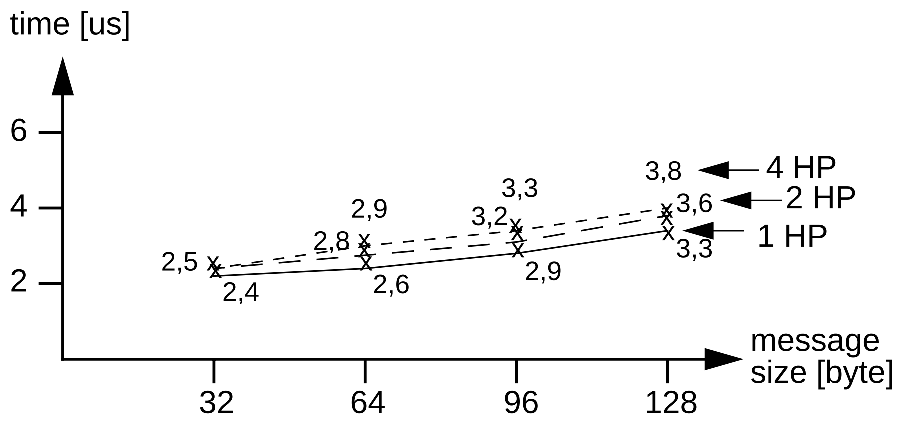


Figure 5-2 shows that the number of active host ports in parallel has only a minor effect on message latency. Only DMA mode transfers were used. E.g. a15 % increase in latency for all four host ports in use is an acceptable value, compared to a single host port. Further investigation revealed that the multiple accesses to the PCI-X bus from different host ports were smoothly interleaved by the logic in the port interconnect and the synchronization interface. During an active PCI-X transfer, several read requests can queue up in the related data paths and can be started as soon as the current transfer is finished. The idle time between back-to-back PCI-X transfers also depends on the performance of the bus arbiter. The simulation used 6 PCI-X cycles to switch control of the bus, a similar value should be found in real implementations.

All in all, the latency numbers taken from simulation are very promising. Even when operating in a quad-CPU node system with multiple message transfers in parallel, latency is remarkable low. This should distinguish ATOLL from other networks, which are typically multiplexed in software when installed inside a SMP node.

5.2 Bandwidth

Measuring the bandwidth of a network is normally done for very large messages. But simulating the sending of a 1 Mbyte message, and this even 10-100 times, was not practical in the described simulation environment. It would have required weeks of simulation runtime. So instead, messages between 1-4 Kbyte are used, repeated 10 times in a row. This should deliver a good insight into the performance of the ATOLL network regarding bandwidth.

Figure 5-3. Bandwidth for a single host port in DMA mode

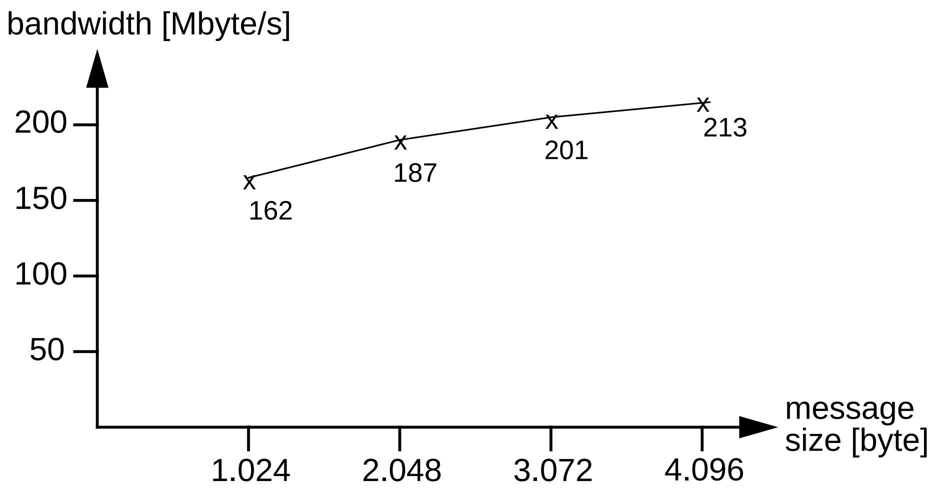
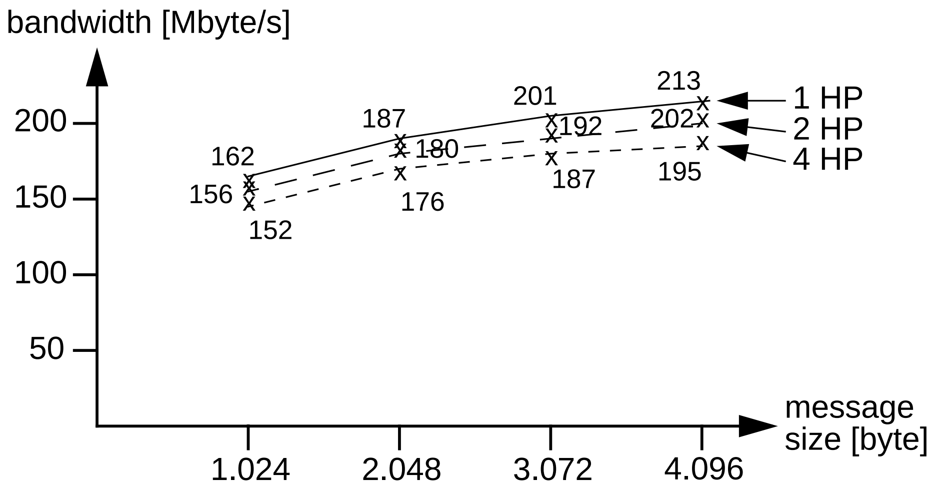


Figure 5-3 depicts the bandwidth of DMA-based message transfer for a single active host port. Reaching 213 Mbyte/s for a 4 Kbyte message is a very good number. These values will decrease by 5-15 % when adding software overhead, but they should still be very competitive. The bandwidth asymptotically approaches a maximum sustained bandwidth of 225-230 Mbyte/s, or about 90 % of the theoretical maximum bandwidth. This shows a good utilization of internal data paths.

Figure 5-4 gives an insight into the bandwidth for multiple host ports in use. Similar to latency, the impact of multiple parallel message transfers is relatively low. Bandwidth drops by only 9 % with all four host ports sending 4 Kbyte messages. The reason for the good scalability of the ATOLL device is the same as for latency. Multiple message transfers are handled very efficiently by the internal logic. The requests from all host ports to read data from main memory are queued and served with minimum overhead. The bottleneck is the PCI-X bus, but with its physical bandwidth of 1 Gbyte/s it is still able to keep all requesters busy. Regarding the internal conversion of a 64 bit data stream into a byte-wide link protocol in the network port, it is sufficient for a host port to deliver one data word on each eighth cycle. This rate can be almost hold up, even for all host ports busy.

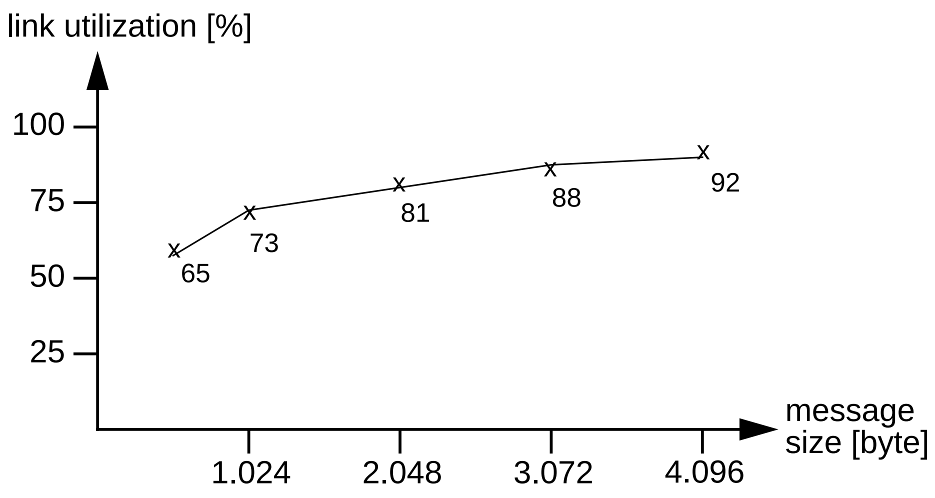
Figure 5-4. Bandwidth for a multiple host ports in use



5.3 Resource Utilization

During the functional simulation of the ATOLL chip implementation, an efficient utilization of internal resources was a major goal, besides the validation of the logic. Early simulations brought up some bottlenecks resulting from too few on-chip resources, which then were resolved by enlarging fifo structures or performing similar enhancements.

Figure 5-5. Network link utilization

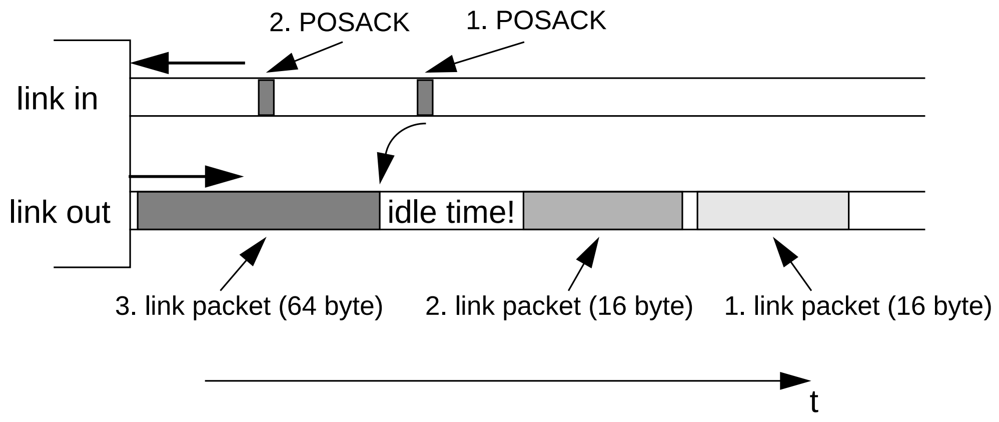


One major resource is the network link. Its efficient use is a crucial factor of overall performance. So one driving force behind the definition of the link protocol was to keep the control overhead for sending packets as low as possible. The amount of control bytes, as defined in the link protocol for message framing, reverse flow control and packet acknowledgment, is kept to the minimum possible.

Figure 5-5 shows the link utilization, given as rate between raw message data vs. total bytes sent over the link for a message. Control overhead is quite high below a message size of 1 Kbyte and makes up for about a third of link traffic. But utilization quickly exceeds 90 % and approaches a maximum value of nearly 95 % for large messages.

Control bytes account for only a small portion of link overhead. Most of the wasted bandwidth is caused by small link packets, which normally appear at the head of a message. The maximum size of 64 bytes is almost never used by link packets belonging to the routing and header frames. Most of the messages sent via the ATOLL network should manage to get along with 1-3 data words (up to 24 data bytes). These consecutive link packets introduce some idle time in the link ports, since only 2 packets can be in transit at one time. After both retransmission buffers have been filled, the link port waits for the positive acknowledgment of the first packet. Only when the POSACK control byte for this packet has been received, the buffer is freed and the next packet can be transmitted.

Figure 5-6. Idle time introduced by small link packets



Though only about 20 idle cycles are introduced when sending two back-to-back link packets with 16 byte payload, this short period of a blocked data path propagates in both directions along the link. E.g. it causes the delayed issue of a read request to main memory in the host port due to the lack of space in the main send data fifo. Figure 5-6 depicts the situation mentioned, showing the idle time introduced by a late acknowledgment byte. But this situation is a rare event in the ATOLL architecture and the additional resources needed for more retransmission buffers far outweighed the gain. So the decision was made to get along with two buffers.

While message data travels through several top-level units in the ATOLL device, it is temporary stored in multiple data fifos spread all over the architecture. So a data path from the PCI-X bus towards the network can be seen as a very deep pipeline, with about 40 register stages in total. The size of the fifos is tailored to provide a steady and continuous data stream. It should be prevented that any of the main units runs out of data, once a message transfer is started.

Figure 5-7. Fifo fill level variation

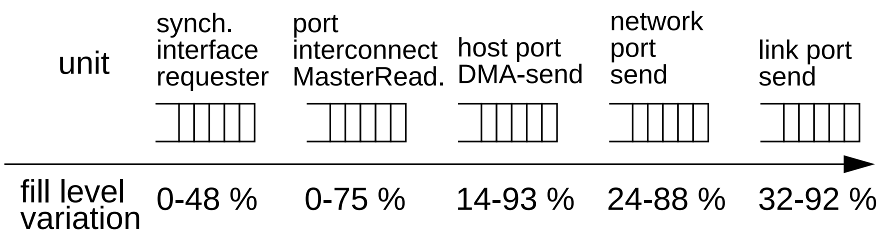


Figure 5-7 lists the variation of the fifo fill level along the data path for the DMA-send mode. It is measured while sending a relatively large message. The numbers display minimum and maximum fullness, once then message transfer has started. One can see that the first two fifos still run empty during transmission. Only a fraction of the total PCI-X bandwidth is used by a single path, so it is not necessary to keep these fifos filled. The more one gets towards the network, the higher is the average fill level of fifos. Once started, the last 3 fifos never run out of data during a message transfer. The fill level still varies a lot, mainly just after the operation started. But once a few data words have been processed, a contiguous data stream keeps all units busy.

All performance measurements were done assuming no network contention. In reality, this assumption is of course too optimistic. So performance numbers will also decrease due to network congestion. But the performance values presented in this chapter demonstrate that the ATOLL architecture is capable of maximizing most internal resources. The network is no more the communication bottleneck in a cluster equipped with the ATOLL network. Instead, the interface to the host system, in this case the PCI-X bus, is the limiting factor. One can overcome this limitation only by locating the network interface closer to the CPU, e.g. on the system bus. But as mentioned earlier, this restricts the use of the NI to a single microprocessor architecture.

6 Conclusions

Clusters are emerging as a competitive alternative to Vector or MPP supercomputers in the field of High Performance Computing. The excellent price/performance ratio of mass-market PC technology makes it very attractive to assemble lots of desktop computers and tie them together with a high performance network. These Beowulf computers start to show up in rankings like the Top500 supercomputer list, and are even more popular for assembling small- to medium-sized clusters of 32-256 nodes.

The key component is a fast network. A new class of so-called System Area Networks emerged, since the traditional LAN/WAN technology was quickly identified as a performance bottleneck. SANs like Myrinet, SCI and QsNet offer message latencies in the order of 5-15 us and sustained bandwidths in the range of 100-300 Mbyte/s. But the price/performance ratio of most networks is still too high to gain a broader acceptance of these new SANs. So the majority of clusters is still equipped with standard Ethernet technology.

Recent trends in PC technology pose new problems to SANs. E.g. SMP desktop computers are becoming available at a price advantage, compared to multiple single-CPU machines. New clusters are often assembled with dual-CPU nodes. This offers a cost advantage, together with less requirements for area, administration and power usage. And the next generation of microprocessor technology will even make 4-8 CPU SMP nodes an attractive node option, mostly based on better SMP support by CPUs.

To sum up, current SAN technology has helped Cluster Computing to make a big step forward, but there is still plenty of room for improvements in performance and cost.

6.1 The ATOLL SAN

This dissertation introduces a novel SAN version of the ATOLL architecture, derived from a first MPP version of the architecture. It integrates all necessary components of a network into one single chip, including the switch. ATOLL provides support for not only one, but four network interfaces by an aggressive replication of resources. Four byte-wide link interfaces running at 250 MHz offer the possibility to directly connect nodes in different topologies, without the need for any external switching hardware.

ATOLL offers a sophisticated mechanism to dynamically choose between PIO- and DMA-based message transfer. This supports extremely low start-up latency for small messages through a Programmed I/O interface, as well as high sustained bandwidth for large messages by autonomous DMA engines inside each host port. An efficient notification mechanism avoids the use of costly interrupts by a cache-coherent polling of status registers in main memory. The consecutive sending/receiving of messages can be done in an overlapping fashion, keeping the utilization of internal resources at a very high level.

An error detection and correction protocol avoids end-to-end control of data transmissions. Bit errors introduced by environmental effects are discovered and solved by a packet retransmission mechanism on each link. On the host side, a state-of-the-art PCI-X interface offers up to 1 Gbyte/s of I/O bandwidth. This is needed to serve the bandwidth requirements of the ATOLL core, which has a bisection bandwidth of 2 Gbyte/s on the network side.

Early performance evaluations promise extremely low latency and a very competitive sustained bandwidth of ATOLL. Multiple data transfers via all four host ports can be supported with a negligible performance impact. This will be an outstanding feature of a cluster composed out of SMP nodes, which are equipped with the ATOLL network card.

Besides redesigning the architecture of ATOLL towards a SAN, this dissertation describes also the implementation of the ATOLL ASIC. Its sheer size and complexity posed a lot of problems, which were solved by a well-planned design flow and a sophisticated design methodology. Despite not quite reaching the timing goal, a transistor layout of the chip was shipped to prototype production in February 2002. It is expected that the real chips can be operated at about 90 % of the targeted clock frequency.

The ATOLL ASIC is one of the most complex and fastest chips ever implemented by a European university. Recently, the design has won the third place in the design contest organized at the Design, Automation & Test in Europe (DATE) conference[1], the premier European event for electronic design.

6.2 Future work

The future development of a second generation of ATOLL will be greatly influenced by technology trends in the whole computer industry. Upcoming interconnect standards like InfiniBand target the server-to-server connection. But it is still questionable, if it will really take over the whole range of I/O as predicted. Other emerging standards like 3GIO

1. www.date-conference.com

from Intel are still in the specification phase, but might become a serious contender for InfiniBand in the area of low-level I/O connectivity, as needed for graphics, I/O devices (keyboard, mouse, modem), etc. InfiniBand might be restricted to the fields now known as Storage Area Networks, replacing such technologies as Fiber Channel and all variations of SCSI. But how both technologies coexist is still part of an ongoing discussion [84]. On the other hand, recent optimizations have been specified for existing technologies like PCI-X. The next generation of PCI-X, as specified in the upcoming v2.0 specification, will support dual- or quad-pumped data busses, raising the maximal physical bandwidth to 4 Gbyte/s.

Of course, the definition of the next generation architecture will be also greatly influenced by the experiences users gain with the first generation of ATOLL. E.g. the PIO mode takes up significant resources inside the chip. If the performance gain, compared to the DMA mode, is not large enough to justify the additional resources, it might be an alternative to drop it and use the free resources for a better implementation of the DMA mode.

Mainly there are two options for the development of the next version of ATOLL. A small and easier to manage option is a so-called technology shrink. It is normally done by making only minor modifications to the overall architecture, but implementing it in the newest technology. So within 1-2 years, one could again implement almost the same architecture by the following steps:

- using a 0.10 um CMOS technology, targeting a core frequency of 500-700 MHz
- going from parallel copper cables to serial fiber links. The reduced pin count needed would also offer the possibility to increase the number of link interfaces to 6-8
- moving from a PCI-X v1.0 bus interface to a PCI-X v2.0 interface offering up to 4 Gbyte/s
- enlarging internal RAM/fifo structures to provide more buffer space

These enhancements would help to keep up with the progress in desktop technology, resulting in a network with three or four times the performance of the current version. This is a possible alternative if no major design flaws are detected during the widespread usage of the first version.

A more challenging approach would be a major redesign of the whole architecture. The current one is efficient but not very flexible. Adding new features is almost impossible, due to the fixed implementation of all control logic in hardware. The current trend in IC design is moving towards so-called Systems-on-a-Chip (SoC) designs. Rather than designing all necessary logic from scratch, one assembles pre-build IP blocks into a larger system. These blocks can be bus interfaces, like the PCI-X interface already used in ATOLL. But a lot of microprocessor cores are also available, e.g. MIPS, ARM, PowerPC, etc. So one could take advantage of the millions of transistors modern IC technology offers by a mixture of soft- and hardmacros, which are glued together by some amount of self-implemented logic. Recent market surveys have shown that the current percentage of chip area used by IP cells is about 20-30 %. Some reports [85] predict that this number increases to 80-90 % within the next 5 years. This way, the implementation effort could be kept manageable, since most of the logic comes as completely verified transistor layout. More concentration could be devoted to the definition of the overall architecture and its fine tuning for highest performance.

The architecture would head towards the ones of Myrinet, QsNet or the IBM SP network, which all have a programmable microprocessor as the key component of the NIC. But ATOLL would offer the whole system combined on a single chip. And with todays advanced technology, even multiple controllers could be implemented. This could be used to split up the work into a host and a network side, avoiding any bottlenecks introduced by off-loading too much work onto a single controller.

Both options have their pros and cons. But it shows that the ATOLL architecture has the potential to compete with the most advanced commercial solutions in the SAN market in the future.

Acknowledgments

This thesis would not have been possible without the help and contributions by many colleagues. My advisor Professor Dr. Ulrich Brüning was always available to discuss technical issues and provided a working environment, which made it possible to finally complete one of the largest non-commercial chip projects.

The whole team of the Chair of Computer Architecture contributed to the development of the ATOLL network. My colleague Patrick Schulz took over crucial parts of the implementation like the insertion of test structures and provided significant support while setting up the complex functional testbed. And my colleague Mathias Waack implemented all the software necessary to make use of the ATOLL network from a user's point of view.

I would like to thank all those individuals for their contributions, which gave me the possibility to complete my own work.

Bibliography

[1] Rajkumar Buyya (editor). "High Performance Cluster Computing: Architectures and Systems, Volume 1." Prentice Hall PTR, 1999.

[2] Gregory F. Pfister. "In Search of Clusters, Second Edition." Prentice Hall PTR, January 1998.

[3] William Gropp, Ewing Lusk, Anthony Skjellum. "Using MPI: Portable Parallel Programming with the Message-Passing Interface." MIT Press, 1994.

[4] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam. "PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing." MIT Press, 1994.

[5] Thomas Sterling, Daniel Savarese, Donald J. Becker, John E. Dorband, Udaya A. Ranawake, Charles V. Packer. "BEOWULF: A Parallel Workstation For Scientific Computation." CRC Press, Proceedings of the 24th International Conference on Parallel Processing, Volume I, pages 11-14, Boca Raton, FL, August 1995.

[6] The Top500 Supercomputer List, www.top500.org

[7] Ian Foster, Carl Kesselman (editors). "The Grid: Blueprint for a New Computing Infrastructure." Morgan Kaufmann Publishers, July 1998.

[8] Jakov N. Seizovic. "The Architecture and Programming of a Fine-Grain Multicomputer." California Institute of Technology (Caltech), Technical Report CS-TR-93-18, 1993.

[9] Duncan Roweth. "Industrial Presentation: The Meiko CS-2 System Architecture." ACM Press, Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, page 213, June 1993.

[10] Gordon E. Moore. "Cramming more Components onto Integrated Circuits." Electronics, Volume 38, Number 8, April 1965.

[11] Donald E. Thomas, Philip R. Moorby. "The Verilog Hardware Description Language, Third Edition." Kluwer Academic Publishers, 1996.

[12] Ben Cohen. "VHDL Coding Styles and Methodologies, 2nd Edition." Kluwer Academic Publishers, 1999.

[13] J. Bhasker, Sanjiv Narayan. "RTL Modeling Using SystemC." Proceedings of the International HDL Conference, San Jose, CA, March 2002.

[14] Peter L. Flake, Simon J. Davidmann, David J. Kelf. "SUPERLOG: Evolving Verilog and C for System-on-Chip Design." Proceedings of the International HDL Conference, San Jose, CA, March 2000.

[15] Nagaraj, Frank Cano, Haldun Haznedar, Duane Young. "A Practical Approach to Static Signal Electromigration Analysis." ACM/IEEE Press, Proceedings of the 1998 Conference on Design Automation (DAC), pages 572-577, Los Alamitos, CA, June 1998.

[16] Patrick P. Gelsinger. "Microprocessors for the New Millennium -Challenges, Opportunities and New Frontiers." Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, February 2001.

[17] Sun Microsystems, Inc. "Sun Compute Farms for Electronic Design." Technical White Paper, 2000.

[18] Raoul A. Bhoedjang, Tim Rühl, Henri E. Bal. "User-Level Network Interface Protocols." IEEE Computer, Vol. 31, No. 11, pages 53-60, November 1998.

[19] Matt Welsh, Anindya Basu, Thorsten von Eicken. "Low-Latency Communication over Fast Ethernet." Proceedings EUROPAR-96, August 1996.

[20] Giuseppe Ciaccio. "Optimal Communication Performance on Fast Ethernet with GAMMA." Springer, Proceedings of the International Workshop on Personal Computers based Networks Of Workstations (PC-NOW), Lecture Notes in Computer Science (LNCS) 1388, Orlando, Florida, March 1998.

[21] Scott Pakin, Vijay Karamcheti, Andrew A. Chien. "Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs." IEEE Concurrency: Parallel Distributed & Mobile Computing, Vol. 5, No. 2, pages 60-73, April 1997.

[22] Yueming Hu. "A Simulation Research on Multiprocessor Interconnection Networks with Wormhole Routing." IEEE-CS Press, Proceedings of the International

Conference on Advances in Parallel and Distributed Computing, Shanghai, China, March 1997.

[23] B. H. Lim, P. Heidelberger, P. Pattnaik, M. Snir. "Message Proxies for Efficient, Protected Communication on SMP Clusters." IEEE-CS Press, Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), pages 116-127, February 1997.

[24] Wolfgang K. Giloi, Ulrich Brüning, Wolfgang Schröder-Preikschat. "MANNA: Prototype of a Distributed Memory Architecture with Maximized Sustained Performance." Proceedings Euromicro PDP Workshop, 1996.

[25] Marco Fillo, Richard B. Gillett. "Architecture and Implementation of Memory Channel 2." Digital Technical Journal, Vol. 9/1, 1997.

[26] James Laudon, Daniel Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server." ACM Press, Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA), Computer Architecture News, Vol. 25/2, pages 241-251, June 1997.

[27] Colin Whitby-Strevens. "The Transputer." IEEE Press, Proceedings of the 12th International Symposium on Computer Architecture (ISCA), pages 292-300, Boston, MA, June 1985.

[28] Thomas Gross, David R. O’Hallaron. "iWarp: Anatomy of a Parallel Computing System." MIT Press, 1998.

[29] Robert O. Mueller, A. Jain, W. Anderson, T. Benninghoff, D. Bertucci, et. al. "A 1.2GHz Alpha Microprocessor with 44.8GB/s Chip Pin Bandwidth." Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, February 2001

[30] PCI Special Interest Group. "PCI-X Addendum to the PCI Local Bus Specification." PCI-SIG, Revision 1.0, September 1999.

[31] Ajay V. Bhatt. "Creating a Third Generation I/O Interconnect." Technology and Research Labs, Intel Corp., Whitepaper, 2001.

[32] Isaac D. Scherson, Abdou S. Youssef (editors). "Interconnection Networks for High-Performance Parallel Computers." IEEE-CS Press, 1994.

[33] Andrew A. Chien, Mark D. Hill, Shubhendu S. Mukherjee. "Design Challenges for High-Performance Network Interfaces." IEEE Computer, Vol. 31, No. 11, pages 42-45, November 1998.

[34] Jose Duato, Sudhakar Yalamanchili, Lionel Ni. "Interconnection Networks: An Engineering Approach." IEEE-CS Press, 1997.

[35] Prasant Mohapatra. "Wormhole routing techniques for directly connected multi-computer systems." ACM Computing Surveys, Vol. 30, No. 3, pages 374-410, September 1998.

[36] William J. Dally and Charles L. Seitz. "Deadlock-Free Message Routing in Multi-processor Interconnection Networks." IEEE Transactions on Computers, Vol. C-36, No. 5, pages 547-553, 1987.

[37] Allan R. Osborn, Douglas W. Browning. "A Comparative Study of Flow Control Methods in High-Speed Networks." IEEE-CS Press, Proceedings of the 12th Annual International Phoenix Conference on Computers and Communications, pages 353-359, Tempe, AR, March 1993.

[38] David C. DiNucci. "Programming Parallel Processors: Alliant FX/8." Addison-Wesley, pages 27-42, 1988.

[39] Jeff Larson. "The HAL Interconnect PCI Card." Springer, 2nd International Workshop CANPC, Lecture Notes in Computer Science, Vol. 1362, Las Vegas, NV, February 1998.

[40] Hermann Hellwagner, Alexander Reinefeld (editors). "SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Compute Clusters." Springer, Lecture Notes in Computer Science, Vol. 1734, 1999.

[41] Data General Corp. "AViiON AV 20000 Server Technical Overview." White Paper, 1997.

[42] G. Abandah, E. Davidson. "Effects of Architectural and Technological Advances on the HP/Convex Exemplar’s Memory and Conmmunication Performance." ACM Press, Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA), ACM Computer Architecture News, Vol. 26, No. 3, pages 318-329, June 1998.

[43] David X. Wang. "New Scalable Parallel Computer Architecture - Non-Uniform Memory Access (NUMA-Q)." IEEE Press, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, NV, June 1997.

[44] Ch. Kurmann, Thomas Stricker. "A Comparison of two Gigabit SAN/LAN technologies: Scalable Coherent Interface versus Myrinet." Proceedings of the SCI Europe Conference (EMMSEC), Bordeaux, France, September 1998.

[45] David Garcia, William Watson. "ServerNet II." Springer, Proceedings CANPC Workshop, Lecture Notes in Computer Science, Vol. 1417, 1998.

[46] Alan Heirich, David Garcia, Michael Knowles, Robert Horst. "ServerNet-II: a Reliable Interconnect for Scalable High Performance Cluster Computing." Technical Report, Compaq Computer Corporation, September 1998.

[47] Compaq Computer Corp., Intel Corp., Microsoft Corp. "Virtual Interface Architecture Specification." Version 1.0, December 1997.

[48] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, Wen-King K. Su. "Myrinet - A Gigabit-per-Second Local-Area-Network." IEEE Micro, Vol. 15, No. 1, pages 29-36, February 1995.

[49] Myricom, Inc. www.myri.com.

[50] Myricom, Inc. "Guide to Myrinet-2000 Switches and Switch Networks." User Guide, August 2001.

[51] Steven S. Lumetta, Alan Mainwaring, David E. Culler. "Multi-protocol Active Messages on a Cluster of SMPs." ACM/IEEE-CS Press, Proceedings of the ACM/IEEE International Conference on Supercomputing, San Jose, CA, November 1997.

[52] Scott Pakin, Mario Lauria, Andrew Chien. "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet." ACM/IEEE-CS Press, Proceedings of the ACM/IEEE International Conference on Supercomputing, San Diego, CA, November 1995.

[53] Loic Prylli, Bernard Tourancheau. "BIP: A New Protocol Designed for High Performance Networking on Myrinet." Springer, Lecture Notes in Computer Science, Vol. 1388, 1998.

[54] Joachim M. Blum, Thomas M. Warschko, Walter F. Tichy. "PULC: ParaStation User-Level Communication. Design and Overview." Springer, Lecture Notes in Computer Science, Vol. 1388, 1998.

[55] Myricom, Inc. "The GM Message Passing System." User Guide, 2000.

[56] Yutaka Ishikawa, Hiroshi Tezuka, Atsuhi Hori, Shinji Sumimoto, Toshiyuki Takahashi, Francis O’Carroll, Hiroshi Harada. "RWC PC Cluster II and SCore Cluster System Software - High Performance Linux Cluster." Proceedings of the 5th Annual Linux Expo, pages 55-62, 1999.

[57] Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, Eitan Frachtenberg. "The Quadrics Network: High-Performance Clustering Technology." IEEE Micro, No. 1, pages 46-57, January 2002.

[58] Compaq Computer Corp. "Compaq Alpha SC Announcement." Whitepaper, November 1999.

[59] IBM. "RS/6000 SP: SP Switch2 Technology and Architecture." Whitepaper, March 2001.

[60] Craig B. Stunkel, Jay Herring, Bulent Abali, Rajeev Sivaram. "A New Switch Chip for IBM RS/6000 SP Systems." ACM/IEEE-CS Press, Proceedings of the ACM/IEEE International Conference on Supercomputing, Portland, OR, November 1999.

[61] InfiniBand Trade Association. "InfiniBand Architecture Specification." Vol.1, 2 & 3, Rev 1.0, October 2000.

[62] IBM. "InfiniBand: Satisfying the Hunger for Network Bandwidth." Whitepaper, June 2001.

[63] Ulrich Brüning, Lambert Schaelicke. "Atoll: A High-Performance Communication Device for Parallel Systems." IEEE-CS Press, Proceedings of the Conference on Advances in Parallel and Distributed Computing, pages 228-234, 1997.

[64] Peter M. Behr, Samuel Pletner, A. C. Sodan. "PowerMANNA: A Parallel Architecture Based on the PowerPC MPC620." IEEE-CS Press, Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, pages 277-286, Toulouse, France, January 2000.

[65] Synopsys, Inc. "DesignWare DW_pcix MacroCell Databook." October 2001.

[66] PCI Special Interest Group (PCISIG). "PCI-X Addendum to the PCI Local Bus Specification." Revision 1.0, September 1999.

[67] Ulrich Brüning, Jörg Kluge, Lars Rzymianowicz, Patrick Schulz. "ATOLL Hardware Reference Manual." Internal Databook, 2002.

[68] Berthold Lehmann. "Implementation of an efficient hostport for the ATOLL SAN-Adapter." Diploma Thesis, Institute of Computer Engineering, University of Mannheim, March 1999.

[69] Jörg Kluge, Ulrich Brüning, Markus Fischer, Lars Rzymianowicz, Patrick Schulz, Mathias Waack. "The ATOLL approach for a fast and reliable System Area Network." Third Intl. Workshop on Advanced Parallel Processing Technologies (APPT'99) conference, Changsha, P.R. China, October 1999.

[70] Michael J. S. Smith. "Application-Specific Integrated Circuits." Addison-Wesley, VLSI Systems Series, 1997.

[71] Pran Kurup, Taher Abbasi, Ricky Bedi. "It's the Methodology, Stupid!" ByteK Designs, Inc., 1998.

[72] Lars Rzymianowicz. "FSMDesigner: Combining a Powerful Graphical FSM Editor and Efficient HDL Code Generation with Synthesis in Mind." Proceedings of the 8th International HDL Conference and Exhibition (HDLCON), pages 63-68, Santa Clara, CA, April 1999.

[73] Tim Leuchter, Lars Rzymianowicz. "Guidelines for writing efficient RTL-level Verilog HDL code." University of Mannheim, http://mufasa.informatik.uni-mannheim.de/lsra/persons/lars/verilog_guide

[74] Michael Keating, Pierre Bricaud. "Reuse Methodology Manual for System-On-A-Chip Designs, 2nd Edition." Kluwer Academic Publishers, June 1999.

[75] Clifford E. Cummings. "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs." Synopsys User Group Meeting (SNUG), San Jose, CA, March 2001.

[76] Peet James. "Shotgun Verification: The Homer Simpson Guide to Verification." Synopsys User Group Meeting (SNUG), Boston, MA, September 2001.

[77] Synopsys, Inc. "Automated Chip Synthesis User Guide, v2001-08." Synopsys Online Documentation (SOLD), August 2001.

[78] Steve Golson. "A Comparison of Hierarchical Compile Strategies." Synopsys User Group Meeting (SNUG), San Jose, CA, March 2001.

[79] Rick Furtner. "High Fanout without High Stress: Synthesis and Optimization of High-Fanout Nets using Design Compiler 2000.11." Synopsys User Group Meeting (SNUG), Boston, MA, September 2001.

[80] Patrick Schulz. "Design for Test (DfT) and Testability of a Multi-Million Gate ASIC." Diploma Thesis, Institute of Computer Engineering, University of Mannheim, November 2000.

[81] Harry Bleeker, Peter van den Eijnden, Frans de Jong. "Boundary-Scan Test: A Practical Approach." Kluwer Academic Publishers, 1993.

[82] Erich Krause. "Integration, Test and Validation of a PCI IP Macrocell into the Multi-Million Gate ATOLL ASIC." Diploma Thesis, Institute of Computer Engineering, University of Mannheim, April 2001.

[83] Mathias Waack. "Concepts, Design and Implementation of efficient Communication Software for System Area Networks." Institute of Computer Engineering, University of Mannheim, June 2002.

[84] IBM. "InfiniBand and Arapahoe/3GIO: Complementary Technologies for the Data Center." Whitepaper, February 2002.

[85] International Business Strategies, Inc. "Analysis of SoC Design Costs: A Custom Study for Synopsys Professional Services." Technical Report, February 2002.