ICT/ECS-2008-41

Generalization of Slot Table Size for Virtual Circuits on Nostrum Networks on Chip

Master of Science Thesis
in Electronics and Computer Systems

Stockholm, Sweden, 2008

Supervisor: Dr. Zhonghai Lu

Examiner: Dr. Zhonghai Lu and Prof. Axel Jantsch

Alexander Wei Yin

Abstract

Since the late 1990's, the trend of integrated circuit design has been integrating several system components or IP blocks, such as processors and memories, on one chip. This is referred to the concept of System on Chip (SoC). However, with the increase of system scales in SoC systems, some problems such as inter-connection communication, scalability, system throughput and response time of the system have become the bottleneck of the system performance. To address these problems, researchers have proposed the concept of Network on Chip (NoC).

To achieve guaranteed bandwidth (GB) QoS on NoC system, Nostrum NoC which is proposed by KTH uses a Time-Division-Multiplexing (TDM) Virtual Circuit (VC) mechanism. However, in the current simulator, the number of LNs for the switches in the network is fixed to 4 and can not be configured. This limits the bandwidth allocation granularity because a VC can only reserve the communication bandwidth in a multiple of 1/4 packets/cycle.

The aim of this project is to generalize the slot table size from a fixed number to an arbitrary configurable number. Adopting the newly proposed concept of Logical Networks (LNs) instead of LNs, we expect the number of LNs can be generalized for each switch in the network. The number should be configured in the XML file which handles the configuration of the network and resources. After the generalization, the VCs in the network must still be free of contention.

After the generalization extension is implemented on the simulator, five different test cases are used to fully guarantee that the extended simulator works correctly. Each test case focuses on a unique aspect of the VC configuration. The results of the test cases prove that the extension on the simulator have been successful.

Acknowledgement

I would like to thank my examiner Professor Axel Jantsch for giving me this opportunity to finish the master thesis in NoC area which I would like to devote to. I want to thank my supervisor Dr. Zhonghai Lu. During the months of the project, I have always been receiving his help and support. He not only supervised me in the master thesis, but also in my personal life and future career even if I am in Finland. I have learned much more than programming skill from him. His way of thinking, his diligence and high efficiency deeply impressed me.

Mostly, I would like to thank my PhD supervisor Professor Hannu Tenhunen. He is a great gentleman and professor in all aspects of research and life. In the last 6 months, he has been financially supporting me to finish my master thesis. Except for my master thesis and other research work, he is also very caring about my personal life and even my mood. He rented an apartment for me before I found a room in Finland. When the earthquake in China happened, he paid me to go to Sweden for a meeting with Chinese ambassador to comfort my feeling. He took me to several expensive meetings to give me a chance to know and learn from the big persons in the field. I would like to thank all my colleagues in Finland. I thank Dr. Pasi Liljeberg for his help, support, supervising and friendship. I thank Liang Guang for helping me to setup the simulation environment.

Finally, I give my deepest gratitude and love for my parents and my fiancée Wenjing. Since studying abroad, I have not met Wenjing personally for almost two years. But Christ's love sticks us together during this time. Our marriage will be held in August, 2008. I thank my parents for their endless love and support.

Alexander Wei Yin

June 16, 2008, Turku, Finland

Table of Contents

Chapter 1
Introduction

1.1 Project Background

Over the last forty years, the integrated circuit (IC) technology has provided the ability of integrating increasing number of elements in planar form [1]. Driven by the need of implementing increasing number of elements on a single chip, the IC technology has gone through the process of general-purpose logic (GPL) devices, memories, microprocessors, field programmable gate arrays (FPGA), application specific integrated circuits (ASIC), system-on-chip (SoC) and eventually to network-on-chip (NoC).

In the early 1960's, the GPL devices were the first implementation format used in the semiconductor industry. Soon after that, scientists developed the first memory devices and then the first microprocessor on chip in 1971. The successful combination of GPL devices, memories and microprocessor has formed the heart of digital systems and a revolutionary milestone known as embedded systems. Next phase of the IC technology was the micro controller unit (MCU) in 1970's. The idea of a MCU is to integrate a whole microprocessor system on a single chip. Since then, the hardware implementation of digital systems has followed the development from PLDs and FPGAs to ASICs. In 1990s, the introduction of intellectual property (IP) blocks and virtual components brought in a new implementation format called system-on-chip where a single chip contained mostly reusable IP based logic blocks. As the scale increased, normal SoC has encountered several big challenges such as deep submicron effect, global synchronous, power and thermal management, verification and productivity gap [2]. At the beginning of 21st century, the network-on-chip was proposed in the SoC community to solve those challenges.

The NoC architecture, as outlined in figure 1.1, provides the communication infrastructure for the resources. In this two dimensional mesh topology, each node in the network includes a switch and a resource which can be a processor core, a DSP core, a memory bank, a specialized I/O block such as Ethernet or Bluetooth protocol stack implementation, a graphics processor, a FPGA block, etc.

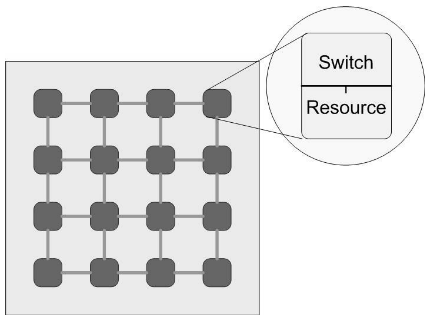


Figure 1.1 NoC Architecture

The data communication in a NoC can be categorized into two types: connection oriented and connectionless oriented. A communication is called connection oriented if a route is established between the source and the destination for the information transmission. In the connectionless systems, information is encapsulated into entities, which travels through the network from the source to the destination [3]. A communication scheme is called virtual channel or virtual circuit (VC) if it is possible to set up the network in such a way that the packets belonging to the same message reach the destination in a proper sequence. In the NoC platforms such as Nostrum [4] and Æthereal [5], the Time-Division Multiplexing (TDM) technology is adopted for VC design so that the VC shared buffers and link bandwidth in a time division manner.

In this project, Nostrum NoC platform is studied. Nostrum is the NoC concept proposed by the research group in Royal Institute of Technology (KTH), Sweden. It is a two dimensional mesh network. Each node represents a switch in the network which is also connected to a resource.

The interface on the current Nostrum NoC platform is compliant with AMBA Advanced eXtensible Interface (AXI) protocol. The AMBA 3 AXI protocol is the successor of AMBA 2.0, which is one of the protocols offered by ARM. Its acceptance and interest in the IP market has been well established. The AXI protocol is targeted at high-performance, high-frequency system designs and includes a number of features that make it suitable for high-speed submicrons interconnect [6]. The key features of the AXI protocol are:

- Separate address/control and data phases
- Support for unaligned data transfers using byte strobes
- Burst-based transactions with only start address issued
- Separate read and write data channels to enable low-cost Direct Memory Access (DMA)
- Ability to issue multiple outstanding addresses
- Out-of-order transaction completion
- Easy addition of register stages to provide timing closure

1.2 Problem Statement

In Nostrum NoC, the packets transfer is based on the concept of deflective routing, which implies no explicit use of queues where packets can get reordered, i.e. packets will leave a switch in the same order that they entered it [7].

Due to the deflective routing policy's non-reordering of packets, an implicit time division multiplexing in the network is created. This is called Temporally Disjoint Network (TDN) in Nostrum NoC [7].

In the current Nostrum NoC simulator, the number of TDNs is determined by the topology of the network and the number of buffer stages in the switches. The contribution to the number of TDNs that stems from the topology is called the Topology Factor. As stated before, the Nostrum NoC architecture is N X N Mesh network. In Mesh topology, the Topology Factor is 2. The Buffer Stages in Nostrum NoC is 2. According to [7],

TDN = Topology Factor x Buffer Stages

From this equation, it is obviously that the number of TDNs in Nostrum NoC should be set to 4 for each switch.

In [8], the concept of Logical Network is proposed which expands the range of the original TDN concept. By adopting the Logical Network into the Nostrum NoC, the number of LNs does not have to be restricted by the multiplication of Topology Factor and Buffer Stages. The free of the number of LNs allows more flexible reservation of communication link bandwidth in the network, which apparently can improve the performance of the network.

However, in the Nostrum NoC simulator, the number of LNs is still fixed to 4 and can not be modified. This limits the bandwidth allocation granularity because a VC can only reserve the communication bandwidth in a multiple of ¼ packets/cycle.

In this project, we aim at generalizing the slot table size from a fixed number to an arbitrary configurable number. We expect the number of LNs can be generalized for each switch in the network. The number of LNs should be able to be assigned in the xml file which handles the configuration of the network and resources. To simplify the configuration, the default number of LNs will be 4 if it is not re-assigned by the users. The user configurable number of LNs makes the communication bandwidth allocation finer, more flexible and more efficient.

1.3 Outline of Thesis Report

The rest of the master thesis report is organized as follows:

Chapter 2:
Chapter 2 focuses on the concept of NoC and its Quality of Services (QoS). It is the theoretical foundation of the whole project. The first section of chapter 2 gives a description of NoC and QoS. While in the second section, TDM virtual circuit technology is discussed in more details.

Chapter 3:
Chapter 3 illustrates the Nostrum NoC simulator. All the work in this project is based on this simulator. We firstly give an overview of the simulator and then present its structure. At the end of chapter 3, how the TDM VC is implemented in the simulator is discussed.

Chapter 4:
In chapter 4, the implementation work in this project is presented. It can be seen as an extension of the original Nostrum NoC simulator. Section 4.1 tells how the generalization is achieved while section 4.2 focuses mainly on the programming implementation.

Chapter 5:

Chapter 5 presents the evaluation of the implementation. It provides five different test cases to test the correctness of the implementation. The test cases proves the implementation in different aspects such as scale of the network, number of transactions, number of TDNs and communication link overlapping schemes.

Chapter 6:
In chapter 6, the project is summarized. The conclusion and future work are also presented in this chapter.

Chapter 2
NoC and QoS

In this chapter, we mainly discuss the concept of Network-on-Chip (NoC), its Quality of Service (QoS) and the TDM Virtual Circuit mechanism. This chapter illustrates the theoretical foundation of the simulator.

2.1 QoS in NoC

2.1.1 Origin of NoC

The fast developing IC manufacturing technology will provide the industry with billions of transistors on a single chip in a few years according to the Moore's Law. At the same time, the IP blocks integrated on chip will be increasing which will lead to an exponential rise in the complexity of their interaction. If this prediction holds, the traditional digital system design methods, especially System-on-Chip (SoC) and bus based system design, will soon face system bottleneck and design challenges [1] [2].

Communication verses computation. With the technology develops, the scale of electronic systems grows exponentially. In company with the use of reusable IP blocks, the design of computing component/circuit is not longer of the most importance in system design since the reusable IP blocks can provide satisfying circuit features and performances. Researches have pointed out that the data transmission and communication have profound effects on system performance. Communication often becomes the bottleneck in large scale systems. Therefore, the focus of system level design has turned to the communication.

Deep submicron effects. Cross-coupling, noise and transient errors are some of the unpleasant side-effects of technology scaling [1]. It requires significant experience, knowledge and skill to solve them. However, as mentioned above, the use of carefully designed IP cores greatly reduce these effects. Therefore, it is important for the system designers to make sure that the physical and electrical properties of IP cores will not change when they are combined in a system.

Global synchronization. As the scale of on-chip systems increases, it becomes almost impossible to mange the clock skew among different part of the chip. This is because that the clock signal usually has to take several clock cycles to transport from one component to another. Thus, it is unlikely that large chips will be synchronous designs with only one clock domain [1].

Design productivity gap. The development of compiling and synthesizing technology does not keep pace with that of the IC manufacturing technology [12]. The use of reusable IP block also helps to reduce the gap since it largely reduces the design time of computing components in systems. It is also expected that the on-chip communication scheme can be used.

Heterogeneity of functions. It is obvious that the functionality of on-chip system is becoming more and more complex. The IP blocks are designed by different designers with different physical and electrical properties and various functionalities. However, they have to be integrated on a single system. How this integration is done has become one of the biggest challenges in system level design.

In order to solve the aforementioned challenges, the concept of Network on Chip (NoC) is introduced by leveraging existing computer network principles. By adopting this concept, the performance of inter-component intra-chip communications can be improved. Since there is no arbitration requirement in a network, more transaction can occur simultaneously and thus let a resource (usually an IP block as a component in the network) transmit a message whenever it is ready, which may decreases the response time (delay) of messages and enhance the usage as well as the throughput of a system. Moreover, as each link in a NoC system is based on point-to-point (P2P) mechanism, the network wires and consequently the communications among resources can be pipelined. In the view of scalability, network architectures can be increased exponentially by inserting repeaters without performance degrading.

2.1.2 Different NoC topologies

The interconnection networks are composed of a set of shared routing nodes and channels, and the topology of the network refers to the arrangement of these nodes and channels [13]. The topology of an interconnection network is analogous to a roadmap. The channels (like roads) carry packets (like cars) from one router node (intersection) to another. A good topology exploits the characteristics of the available packaging technology to meet the bandwidth and latency requirements of the application at minimum cost.

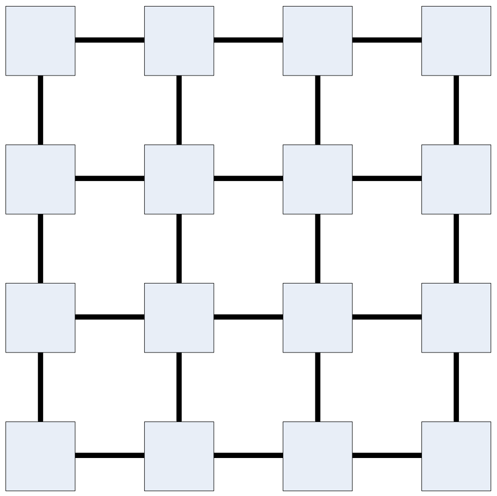


Figure2.1 MeshNetwork

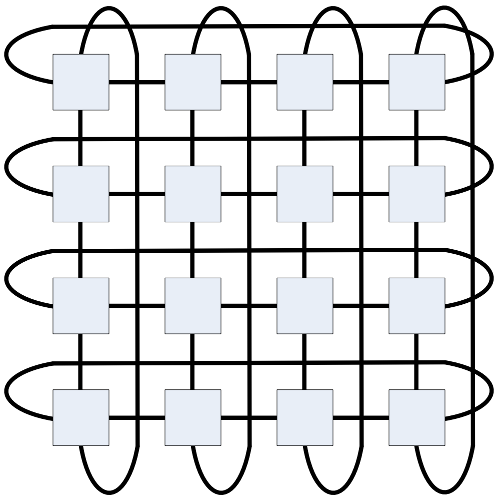


Figure 2.2 Torus Network

The two mostly used topologies in today's NoC research are the two dimensional (2D) Mesh network (figure 2.1) and Torus network (figure 2.2). The 2D Mesh and Torus network topologies provide a way to route data among nodes. They allow continuous connections and reconfiguration around broken or blocked paths by "hopping" from node to node until the destination is reached. A Mesh/Tours network whose nodes are all connected to each other is a fully connected network. The Mesh/Torus network topologies differ from other networks in that the component parts can all connect to each other via multiple hops, and they generally are not mobile. These topologies can be seen as two types of ad hoc network [14].

Although sharing some common characteristics, the Mesh and Torus have differences not only in topology but also in their performances. As can be seen from figure 2.1 and figure 2.2, the difference in topology between Mesh and Torus is that there is a link directly connects the first node and the last node in each row and column in Torus network. The Torus topology provides more communication bandwidth since the average hops of the Torus is smaller than that of the Mesh. However, the Mesh topology is simpler and provides better scalability.

In the last two years, the three dimensional (3D) NoC topology has emerged. The benefits of 3D ICs include: 1) higher packing density due to the addition of a third dimension to the conventional two-dimensional layout, 2) higher performance due to reduced average interconnect length, and 3) lower interconnect power consumption due to the reduction in total wiring length [15]. The topology of 3D NoC is shown in figure 2.3.

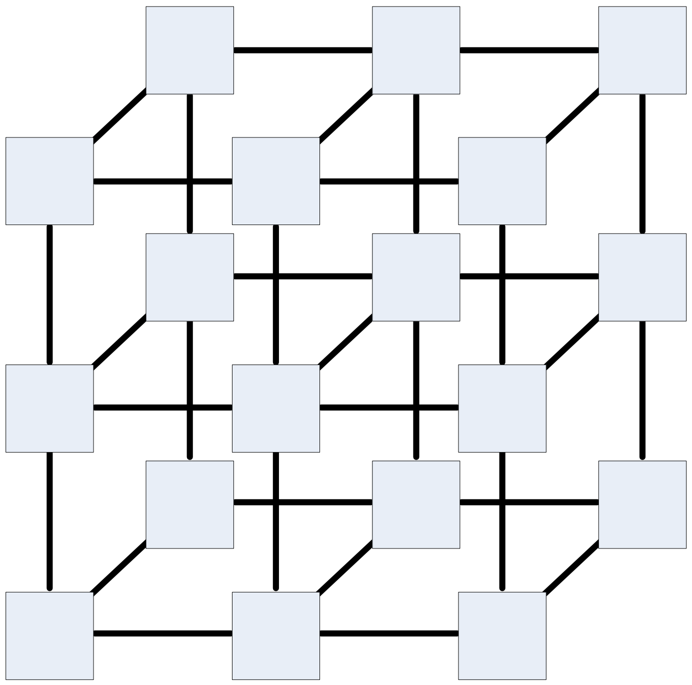


Figure 2.3 Topology of 3D NoC

Except for the aforementioned ones, other topologies such as fully connected network, fat tree network, star network, ring network, honeycomb network are also used in NoC. To overcome the shortcomings of each topology, sometimes, hybrid topologies are needed.

2.1.3 Protocol Stack of NoC

A set of rules and methods are required for transferring information from one resource to another in any system. These rules are generally referred as protocols [3]. In a packet switched network, communication protocols determine how a resource is connected to the network as well as how the information flows from source to destination.

In a NoC based system, there are little restrictions on the functionalities and properties of resources. More methods and rules are needed to guarantee the communication in NoC. Thus, researchers turned to the layered protocols in which various functions required for communications are divided hierarchically among different layers. The layered communication helps to mange the complexity due to various requirements of heterogeneous resource, and provides a mechanism to isolate various problems in different layers. The architecture which defines the protocol layers is referred as the protocol stack.

Inspired by the OSI seven-layer model [16], most NoC researchers have proposed similar layered stacks. One of the most popular protocol stacks is defined as follows [3] [4]:

- **Physical Layer:** This layer is concerned with physical properties of the physical medium used for connecting switches and resources with each other. In a NoC system, it specifies voltage levels, length and width of wires, signal timings, number of wires connecting two units etc.

- **Data Link Layer:** The purpose of this layer is to provide reliable information transfer across the physical link. The layer has the functionality of transferring one word of information from one node to a connected node without error. Since the two connected units may work asynchronously, the data link layer also has to take care of the hardware synchronization. The last responsibility of the data link layer is the data encoding or data rate management for controlling power consumption etc.

- **Network Layer:** The network layer provides the service of communicating a packet from one resource to another using the switches in the network. It also handles the buffering of packets and making routing decisions in the switches.

- **Transport Layer:** The transport layer handles the establishment of communication and delivery of messages using lower level layer. It also provides the traffic control, i.e. the load of the network is detected and overload of the network is avoided.

- **Application Layer:** For on chip communication, the functionality of the three highest layers in OSI model can be merged into this layer. Important services in this layer include message synchronization and management, conversion of data format at receiver and application dependent functionalities.

2.1.4 QoS in NoC

In the field of computer networking and other packet-switched telecommunication networks, the traffic engineering term quality of service (QoS) refers to resource reservation control mechanisms rather than the achieved service quality. Quality of service is the ability to provide different priorities to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow [17]. For example, a required bit rate, delay, jitter, packet dropping probability and/or bit error rate may be guaranteed.

A network or protocol that supports QoS may agree on a traffic contract with the application software and reserve capacity in the network nodes, for example during a session establishment phase. During this session it may monitor the achieved level of performance, for example the data rate and delay, and dynamically control scheduling priorities in the network nodes. It releases the reserved capacity during the tear down phase [17].

Many problems are prone to occur to packets as they travel from origin to destination, some of them are as follows:

- **Dropped Packets:** The routers might fail to deliver (drop) some packets if they arrive when their buffers are already full. Some, none, or all of the packets might be dropped, depending on the state of the network, and it is impossible to determine what will happen in advance. The receiving application may ask for this information to be retransmitted, possibly causing severe delays in the overall transmission.

- **Delay:** It might take a long time for a packet to reach its destination, because it gets held up in long queues, or takes a less direct route to avoid congestion. In some cases, excessive delay can render an application, such as VoIP or online gaming unusable.

- **Jitter:** Packets from the source will reach the destination with different delays. A packet's delay varies with its position in the queues of the routers along the path between source and destination and this position can vary unpredictably. This variation in delay is known as jitter and can seriously affect the quality of streaming audio and/or video.

- **Out-of-Order Delivery:** When a collection of related packets is routed through the Internet, different packets may take different routes, each resulting in a different delay. The result is that the packets arrive in a different order than they were sent. This problem requires special additional protocols responsible for rearranging out-of-order packets to an isochronous state once they reach their destination.

- **Error:** Sometimes packets are misdirected, or combined together, or corrupted, while being delivered. The receiver has to detect these errors and if any error is found, it should ask the sender to send again.

In some applications of the interconnection network, it is useful to divide network traffic into a number of classes to more efficiently manage the allocation of resources to packets. Different classes of packets may have different requirements; some classes are latency-sensitive, while others are not. Some classes can tolerate latency but not jitter [18].

The traffic classes in NoC system fall into two broad categories: guaranteed service classes and best effort classes.

Guaranteed service classes guarantee a certain level of performance as long as the traffic they inject complies with a set of restrictions. There is a service contract between the network and the client. As long as the client satisfies the restrictions in the service contract, the network will deliver the performance. The client side of the agreement usually restricts the volume of traffic that the client can inject, that is, the maximum offered throughput.

In contract, the network makes weak promises to the best efforts packets. Depending on the network, these packets may have arbitrary delay or even be dropped. As depicted by its name, the network will make its best effort to deliver the packets to their destination.

2.2 TDM Virtual Circuit

2.2.1 TDM VC

Before introducing the concept of TDM VC, it is necessary to describe the circuit switching and packet switching techniques.

In telecommunications, a circuit switching network is one that establishes a fixed bandwidth circuit (or channel) between nodes and terminals before the users may communicate, as if the nodes were physically connected with an electrical circuit [19]. The bit delay is constant during the connection. Each circuit cannot be used by other callers until the circuit is released and a new connection is set up. Even if no actual communication is taking place in a dedicated circuit, the channel remains unavailable to other users.

Packet switching is a communications method in which packets (discrete blocks of data) are routed between nodes over data links shared with other traffic [20]. In each network node, packets are queued or buffered, resulting in various delays. This contrasts with the circuit switching, which sets up a limited number of constant bit rate and constant delay connections between nodes for their exclusive use for the duration of the communication. Packet switching is used to optimize the use of the channel capacity available in digital telecommunication networks such as computer networks, to minimize the transmission latency, and to increase the robustness of communication.

The difference between circuit and packet switching is the manner the switch inside the router is controlled [21]. When the controlling information such as the size of packets, the configuration of switches is embedded with a burst of data, it is called packet switching. Circuit switching refers to the situation where the controlling information is sent beforehand and the connection is maintained so that any consecutive bursts of data follow the same path.

Time-Division Multiplexing (TDM) is a type of multiplexing in which two or more signals or bit streams are transferred apparently simultaneously as sub-channels in one communication channel, but are physically taking turns on the channel [23]. The time domain is divided into several recurrent timeslots of fixed length, one for each sub-channel. A sample byte or data block of sub-channel 1 is transmitted during timeslot 1, and that on sub-channel 2 is transmitted during timeslot 2, etc. The TDM frame consists of one time slot per sub-channel. After the last sub-channel, the cycle starts all over again with a new frame, starting with the second sample, byte or data block from sub-channel 1, etc.

The TDM techniques and virtual channel (VC) have been successfully combined in NoC systems. Except for addressing the blocking problem, the TDM VC provides the possibility guaranteed services.

2.2.2 Nostrum TDN

In Nostrum NoC, two classes of QoS have been implemented: Guaranteed Bandwidth (GB) and Best Effort (BE). The GB QoS in Nostrum NoC is based on TDM VC. The VCs are implemented using a combination of two concepts called “looped container” and “Temporally Disjoint Networks (TDN)”. The looped container are used to guarantee access to the network – independent of the current network load without dropping any packet; and the TDNs are used in order to achieve several VCs, plus ordinary BE traffic, in the network [7].

Due to the fact that the Nostrum NoC is based on deflective and bufferless routing, the Nostrum VC is close looped, i.e. each VC has a cyclic path. There is at least one container for each loop. The container is a special packet which can carry the actual information packets. The number of containers in a loop determines the utilization of the VC. In a NoC system, it is possible that different VCs share some common communication links. These VCs are called “overlapped” VCs. In Nostrum NoC, the number of overlapped VCs is limited to the number of TDNs on the shared links. Therefore, we may have many TDNs in a NoC system, but only few of them can overlap with others. The reason for this limitation is that only one VC can subscribe to the same TDN on a switch out port [7].

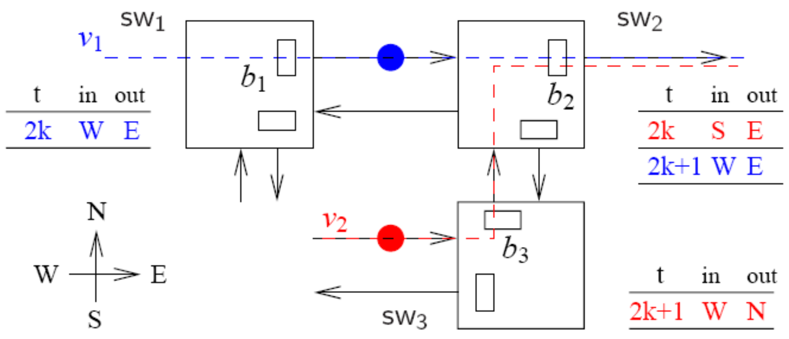


Figure 2.6 TDM VC in Nostrum NoC [8]

Figure 2.6 shows an example of TDN configuration in Nostrum NoC. In a mesh network with one buffer per output port in each switch, only two TDNs exist. To allow more TDNs, more buffers in the switch must be introduced. For example, having two buffers in each switch, one at the input port and the other at the output port, results in four TDNs. In figure 2.6, two VCs, v1 and v2, are configured. The routing table in the switches is as defined in the figure. The example shows that the overlapped VCs in the Nostrum NoC are free of contention.

As discussed in chapter 1, the number of TDNs in Nostrum NoC is fixed to 4 because of the topology factor and buffer stages. However, in [8], the concept of Logical Network (LN) is proposed. The logical network refers to an infinite set of associated (time slot, buffer) pairs with respect to a buffer on a given VC [8]. The LN generalizes the concepts of admission classes and TDN which are proposed to address the packet contention problem in deflection-routed networks and therefore removes the limitation on the number of TDNs as long as the overlapped VCs are free of contention.

Chapter 3
Nostrum Simulator

3.1 Simulator Overview

The Nostrum NoC simulator is a tool for fully simulating a network-on-chip system. It has been developed and maintained by the Nostrum group in Royal Institute of Technology (KTH), Sweden. It provides the cycle-accurate communications on a NoC which is of 2D Mesh topology and uses deflective routing mechanism.

The first version of the Nostrum NoC simulator is developed by Rikard Thid in 2002 as his master thesis [24]. It is named as Semla which stands for Simulation EnvironMent for Layered Architectures. Semla is based on SystemC. It is based on the 2D Mesh topology with hot potato deflective routing mechanism.

Later, Zhonghai Lu has developed a more advanced version of Nostrum NoC simulator called NNSE which stands for Nostrum Network-on-Chip Simulation Environment on the basis of Semla [25]. It has a graphical user interface (GUI) for the Semla. Moreover, a MATLAB based interface is added to the simulator, enabling the simulator to show the simulation result in figures. The wormhole switching mechanism and automatic traffic configuration and generation are added in NNSE. Also, the NNSE has extended the topology of Nostrum NoC simulator so that 2D Torus and Manhattan Street Network (MSN) etc. are supported.

At the beginning of 2008, some big changes have been done on the Nostrum NoC simulator. Mikeal Millberg has re-written the simulator based on all the previous work. The guaranteed bandwidth (GB) QoS and TDN mechanism are implemented. The simulator is still based on 2D Mesh topology. The GUI and MATLAB related features are not supported in the current simulator. However, the simulator is now compliant with the AXI CASI Model. The user interface which are two XML files have been developed in this version.

3.2 Simulator structure

3.2.1 General structure

The Nostrum NoC simulator is a layered network simulator. In order to communicate over the network, resources in the simulator are equipped with a Network Interface (NI) and Resource Network Interface (RNI). The NI provides a standard set of services that can be utilized by the RNI. The role of the RNI is to act as an adaptor between the resource's internal communication infrastructure and the standard set of services of the NI. The protocol stack for Nostrum can be divided into two general classes, namely

Nostrum stack and custom stack. Figure 3.1 illustrates the general structure of Nostrum simulator.

The Nostrum Simulator can be used for both best effort traffic using single-message passing between resources where switching decisions are made locally in the switches for every data routed through the network, as well as for guaranteed bandwidth traffic using virtual circuit.

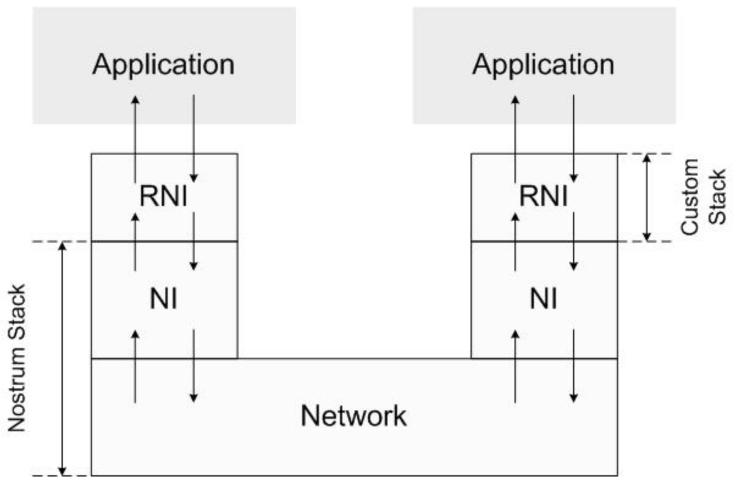


Figure 3.1 General structure of Nostrum simulator

The network architecture of Nostrum simulator is Mesh based. It consists of switches, links, resources and the interfaces. Figure 3.2 is shows how these components are connected in Nostrum NoC.

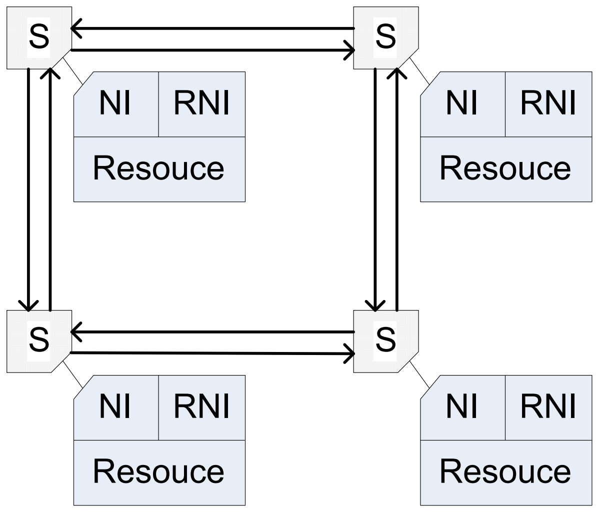


Figure 3.2 Network architecture in Nostrum NoC

As mentioned before, the current Nostrum simulator is compliant with AMBA AXI protocol. The AXI protocol is an interface between the applications and RNI. Thus, it is implemented above the RNI layer. In this report, we use the term "kernel" to refer to all the layers below and include RNI, such as RNI, NI, switch, etc. The applications accordant to the AXI protocol are divided into two groups: masters and slaves.

Figure 3.3 shows the whole structure of the current version of the Nostrum NoC simulator including kernel and AXI applications.

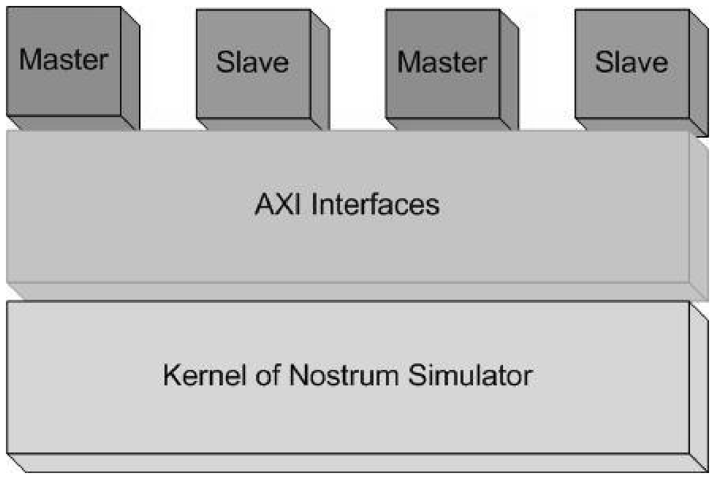


Figure 3.3 Structure of Simulator including kernel and applications

3.2.2 AXI Protocol

The AMBA AXI protocol is proposed for high-performance, high-frequency system and thus includes a number of features that support a high-speed submicron interconnection network [26]. The AXI specification was created with the following objectives to ensure its suitability for the next generation of designs.

- Suitability for high-bandwidth and low-latency designs
- To enable high-frequency operation without using complex bridges
- Meet the interface requirements of a wide range of components
- Suitability for memory controllers with high initial access latency
- Provide flexibility in the implementation of interconnect architectures
- Backward-compatibility with existing AMBA technologies

The key features of AXI protocol include:

- Separate address/control and data phases
- Support for unaligned data transfers using byte strobes
- Burst-based transactions with only start address issued
- Separate read and write data channels to enable low-cost direct memory access (DMA)
- Ability to issue multiple outstanding addresses
- Out-of-order transaction completion

- Easy addition of register stages to provide timing closure
- Protocol includes optional extensions that cover signaling for low-power operation

The AXI protocol defines the interface between masters and slaves. However, it does not require that masters and slaves to be connected directly. In general, the AXI protocol provides a single interface definition for describing interfaces:

- Between a master and the interconnect
- Between a slave and the interconnect
- Between a master and a slave

As is shown in figure 3.4, the interface definition supports a number of different interconnect implementations, such as bus, network, and etc. The interconnect between master/slave components is equivalent to another component with symmetrical master and slave ports to which real master and slave components can be connect.

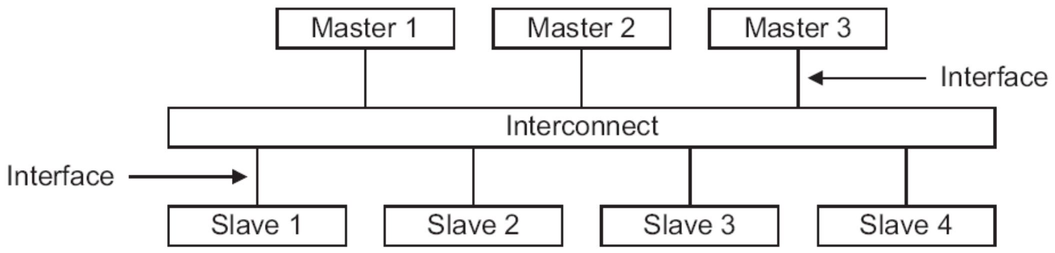
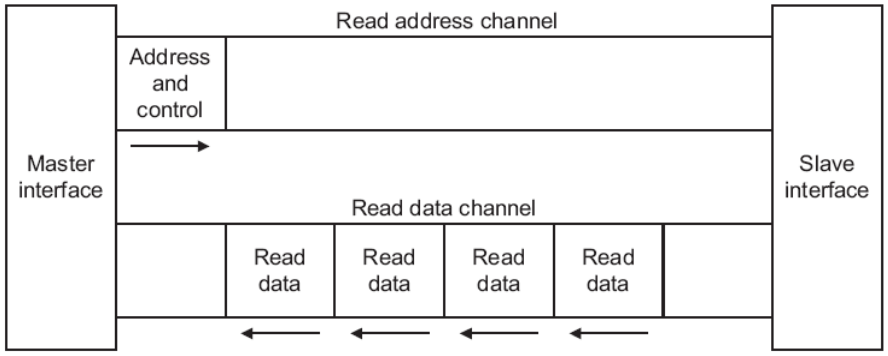


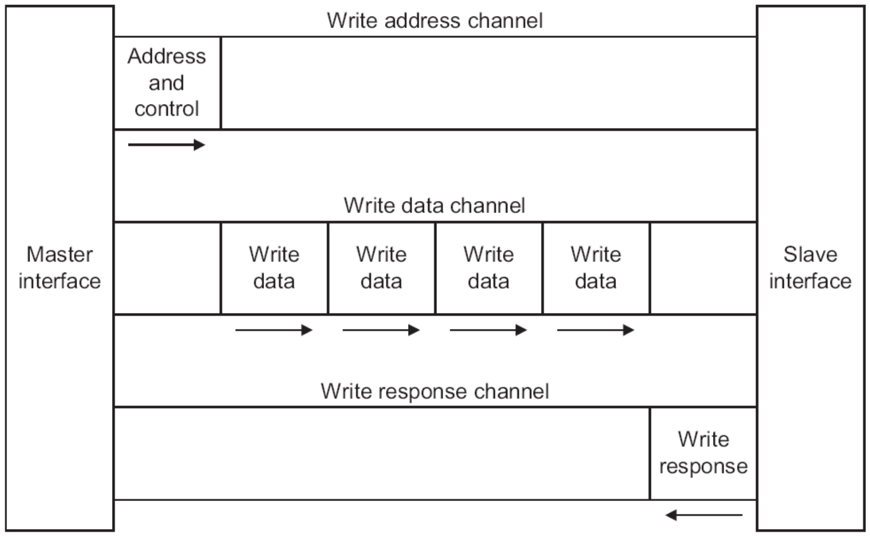
Figure 3.4 AXI Interface and Interconnection

One of the most important features of the AXI protocol is its separation of channels. There are five independent channels in AXI, namely, read address channel, write address channel, read data channel, write data channel and write response channel. In each transaction, address and control information that describes the nature of the data is transferred via address channel. The data is transferred from master to slave via write data channel or from slave to master via read data channel. In write transactions, the AXI protocol has an additional write response channel to allow the slave to notify the master the completion of transaction. Figure 3.5 shows the read and write transactions progress.

Each of the channels consists of a set of information signals and uses a two-way VALID and READY handshake mechanism. At the information source end, the VALID signal is used to show that valid data or control information is available on the channel. At the destination end, the READY signal is used to claim that it can accept the data. Both the read data channel and write data channel include a LAST signal to indicate that it is the final data item being transmitted.



(a) Read transaction



(b) Write transaction

Figure 3.5 AXI Transactions

3.2.3 Kernel of Nostrum Simulator

In this section, we will discuss the functionalities of all the important files in the Nostrum simulator, especially the files related to the kernel.

semla_datatypes.h and semla_datatypes.cpp

These two files define the basic data types that are used in the kernel of the simulator, such as message id, channel id, tdn, etc.

mp_if.h

The mp_if.h defines the TDN type (ChannelType) for the user configuration in the XML file. Moreover, the virtual functions claimed in the Mp_if class are the functions to connect the kernel of Nostrum with the applications. The mp_if object is used in the "resource" in figure 3.2.

mpdu.h and mpdu.cpp

In Nostrum NoC simulator, all the entities communicate by exchanging Protocol Data Units (PDUs) which are defined in mpdu.h. The PDU is essentially a bunch of data elements with various types and different identifiers. The purpose of the PDU is that they shall be passed up and down the protocol stack, avoiding unnecessary copying of data and therefore increase the speed of the simulator. Furthermore, the PDU is flexible; data elements can be added or removed easily. The mpdu object is created from the mp_if object in the "RNI" and used in "NI" and "switch" in figure 3.2.

simulation_params.h and simulation_params.cpp

These files are responsible for parsing the XML file which configures the topology of the network and defines the parameters of the simulator. The parsing methods in these files are inherited from the third party software Xerces-C++.

switch.h and switch.cpp

Switches are the cornerstone of the network. In these files, different ways of calculating priorities and gains are firstly defined. Then, it implements the methods of communication between switches and the NI. At last, the routing mechanism is implemented by comparing the gains of all the possible permutations. The switch object is depicted by the "switch" in figure 3.2.

ni.h and ni.cpp

The network interface is important due to the fact that it is the intermediate layer between RNI and switches. Its responsibilities include handling and controlling both the upstream and the downstream communication. The network interface also updates and maintains the TDN number for its neighboring RNI and switch. The NI object is illustrated by the "NI" in figure 3.2.

rni.h and rni.cpp

RNI is the interface between the applications and the kernel of the simulator. It extracts all the information from the application layer(s) and sends them to the network interface. It also receives the upstream packets from the network. The virtual functions defined in mp_if.h are implemented in RNI. The "RNI" in figure 3.2 is where the rni object located in Nostrum.

wire.h and wire.cpp

The wire object simulates the communication link between switches in the network. The only use of wires is to transmit the packets from one end of the wire to the other end. The double arrows in figure 3.2 are the objects of wires.

edge.h and edge.cpp

In Nostrum NoC simulator, the user is able to decide if the buffer in the edge elements should be enabled. If the edge is enabled, it is possible to route a packet in the edge direction. The purpose is to bounce the packet back to the direction from which it came after two cycles delay. Otherwise, all the routing to the edge direction should be denied.

network.h and network.cpp

In the network object, the network of the NoC system is set up. It creates the edges, wires, switches and NIs. After creating these elements, the network then connects them according to the configuration.

nostrum.h and nostrum.cpp

In the nostrum object, network is created and configured. Also, RNIs which is not a part of the network object is created and connect to the corresponding NIs according to the index of RNI and NI. By now, the kernel of the Nostrum NoC is created and configured.

3.3 TDN implementation in Nostrum

The TDN is implemented in two phases in Nostrum simulator, namely, switch and channel.

3.3.1 Switch phase

Preferred TDN and Virtual Channel are selected in the switch by assigning different priorities to all the possible routing permutations. The assignment is handled by setPreferedDirection (int tdn, int input, int output) function which is a member function of the switch. The function is called by the AXI_Resources. After setting the priorities, the routing_process() of the switch calculates the gain for every possible permutation combination, _gain[input][output]. Then the permutation with the highest gain is executed which simulates the process of routing in a switch.

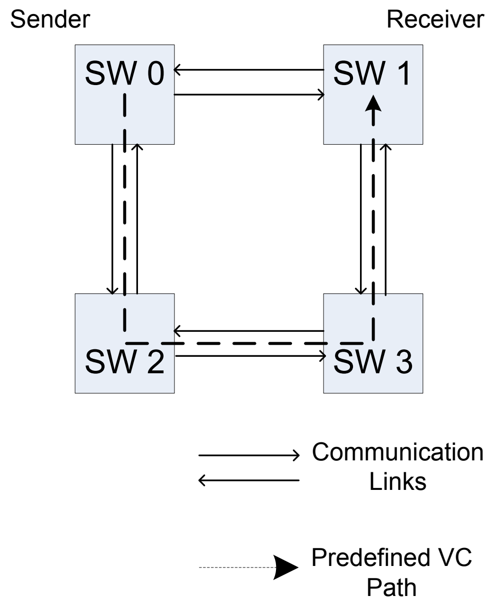


Figure 3.6 Example of TDN reservations in switches

Take Fig. 3.6 for an example. Suppose there is a predefined virtual circuit path of VC_TDN1 which is shown as the dotted line from the sender (SW0) to the receiver (SW1) and there are four TDNs in each switch. The TDN related data propagation will work in the following way. First, each switch updates its own current TDN each cycle by doing a modulo-4 addition. Second, in SW2, for example, (TDN 1, North, East) is assigned with the highest priority, which means when current TDN is VC_TDN 1, data from the North port will be delivered to the East port. All the other switches set the predefined VC as the preferred routing directions as well. This path is then reserved by VC_TDN 1.

3.3.2 Channel phase

In the Nostrum NoC simulator, data from different protocol layers is transmitted either down stream or up stream via certain inter media. The inter media that carries the data from a layer to its adjacent layers is called channel in this context. A channel is logically implemented by binding the output port of the source layer and the input port of the destination layer or vice versa.

The openChannel() functions is called by the AXI_Interconnect_Slave and AXI_Interconnect_Master which locate respectively on the master side and slave side of the interconnect network. When opening a channel, the function assigns a specific TDN to it. This is because in the Nostrum simulator, a channel simply maps all its traffic onto one TDN. The openChannel() function is one of the member functions of mp_if which consists of a number of virtual functions. The implementation of mp_if used by AXI_Interconnect_Slave and AXI_Interconnect_Master is defined in RNI.

In the Network Interface (NI), the TDN related channel is registered by registerChannel() function. In this function, parameters of niChInfo such as niPort (determines which TDN the data belongs to) and prio are registered. The function is called in ds_inco_ctrl_process() after identifying the channel type (TDN). The tdnCounter_process() of NI is responsible for updating the current TDN. In this version of Nostrum simulator, the total number of TDNs is fixed to 4. The channelGranter_process() of NI is one of the cores of TDN mechanism in Nostrum simulator. It sets the ready signal to the ds_inco_portReady which determines whether the data on a certain channel of RNI can be transmitted to NI. For example, suppose the current TDN is 1, the channelGranter_process() sets the ds_inco_portReady signal to (0,1,0,0) which implies only the data from TDN channel 1 can be transmitted to NI.

In the write(int cid, int msg) function of RNI, downstream data is pushed on ds_data_fifo[] according to its TDN. When using CHTYPE_DEFAULT_TDN or CHTYPE_ANY_TDN, all the data is pushed on ds_data_fifo[0], otherwise the data is pushed on the corresponding ds_data_fifo[TDN].

In the ds_outg_data_process() process of RNI, the RNI checks the current TDN (available port of NI) and sends the data from the corresponding ds_data_fifo[TDN].

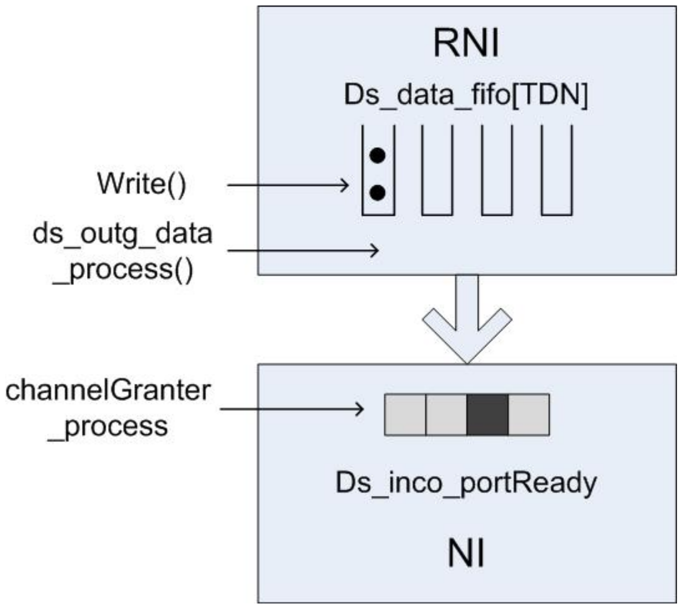


Figure 3.7 Example of TDN between NI and RNI

Fig. 3.7 shows an example of how the TDN works in RNI and NI. In NI, since there are 4 TDNs in the current version, a 4-bit vector is used to show the current TDN. There is only one valid bit at one time and this bit rotates in the vector. The process is controlled by tdnCounter_process() and channelGranter_process() of NI. In RNI, there are 4 ds_data_fifo to store data on different TDN. Suppose there are 2 packets on TDN0 and none on other TDNs, the process to send these two data is shown in table 3.1.

Cycle	Current TDN	
1	TDN0	Send the 1st packet in ds_data_fifo[0]
2	TDN1	Check ds_data_fifo[1]
3	TDN2	Check ds_data_fifo[2]
4	TDN3	Check ds_data_fifo[3]
5	TDN0	Send the 2nd packet in ds_data_fifo[0]

Table 3.1 Example of Sending two data on TDN0

3.3.3 Contention free

One of the most important requirements for the TDN in Nostrum simulator is that the overlapped communication links must be free of contention. The contention occurs when packets belong to different TDNs arrive at the same switch, acquiring the same communication link. However, only one of the packets can possess the link and thus the collided packets will not be routed in the predefined directions. Figure 3.8 is an example of the contention. The predefined VC path on TDN 0 is from switch 1 to switch 14 and the predefined VC path on TDN 1 is from switch 5 to switch 14. Suppose a packet takes only one clock cycle to transmit from a switch to any of its neighboring switch and the switches have no buffer to store the packet, it is obvious that the packets on the two predefined VC paths will collide at switch 6 if all the switches have the same initial TDN number.

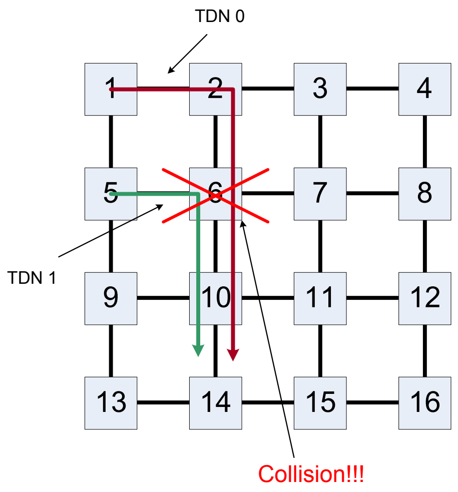


Figure 3.8 Example of contention

To make the Nostrum NoC simulator free of contention, a tricky solution is proposed which assigns the switches in the network with different initial TDN number. A minutia which is worth being mentioned is the setPosition() function in the NI. It categorizes the switches into two groups, namely, even switches and odd switches. If the sum of the xPos and yPos of a switch is an even integer, the switch is called an even switch. Otherwise, it is an odd switch. The initial TDN number of the odd switches is two cycles ahead of that of the even switches. As shown in figure 3.9, the switches with deeper color are even switches while the lighter ones are odd switches. The two predefined VC paths are the same as those in figure 3.8. In this example, packets on these VCs are free of contention since the switches have different initial TND number.

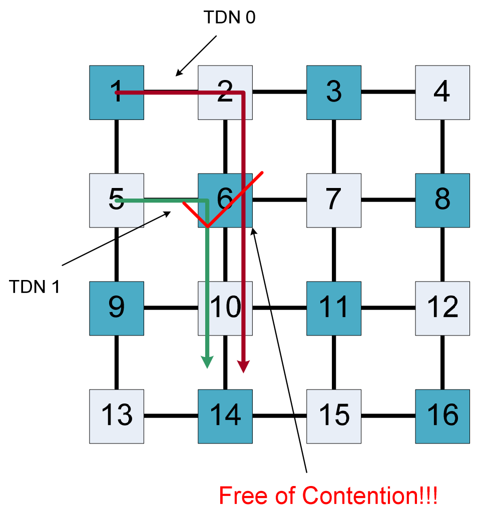


Figure 3.9 Free of contention mechanism

Chapter 4
Extension of Nostrum Simulator

In this chapter, the detailed implementation on Nostrum simulator is presented. The generalization is based on the concept of logical networks (LNs) instead of TDNs, therefore, from this chapter on, the generalized TDN is referred to LN. In the beginning of this chapter, we discuss how the LN is generalized and how this generalization will affect the simulator. Then the core part of the implementation in this project will be illustrated.

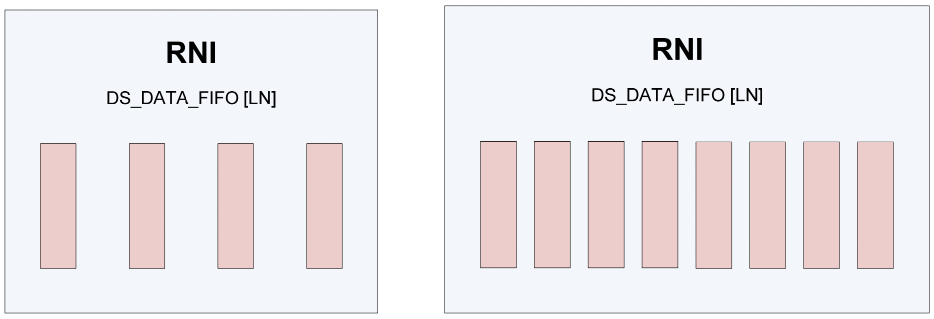
4.1 Generalization of LN

The generalization of LNs in Nostrum simulator can be divided into two levels. One is the kernel level while the other is the application level.

4.1.1 Kernel level generalization

As mentioned in chapter 3, there are three different layers in the kernel of Nostrum simulator, namely, switch, network interface (NI) and resource network interface (RNI).

The RNI controls when a packet from the resource can be forwarded to the network The packet from the resource is firstly stored in one of the FIFOs in RNI according to which LN the packet belongs to. Therefore, the most intuitive modification is to generalize the number of FIFOs in compliant with the number of LNs of the RNI. Figure 4.1 (a) shows the old structure of RNI and figure 4.1 (b) depicts a RNI with 8 LNs.



(a) RNI with 4 LNs (b) RNI with 8 LNs

Figure 4.1 RNI with generalized LN

After the number of FIFOs is generalized for the RNIs, each RNI is able to have a unique number of FIFOs. This is because the number of LNs in a RNI is assigned when the RNI is created.

When a packet reaches the proper FIFO in the RNI, it has to be delivered to the NI which is the immediate lower layer of RNI. The delivery is done when the LN of the packet matches the current LN of the RNI and NI which is kept by the NI.

From the perspective of LN, the NI is responsible for keeping and updating the current LN and informing it of its immediate neighboring RNI and switch. The NI uses a bit vector to inform the RNI of the current LN. Only one bit of the vector is set to '1' at one time which indicates the current LN. The bit '1' rotates in the vector so that the LN number rotates according to its maximum number of LN. Figure 4.2 shows how this mechanism works in a NI whose number of LN is set to 8.

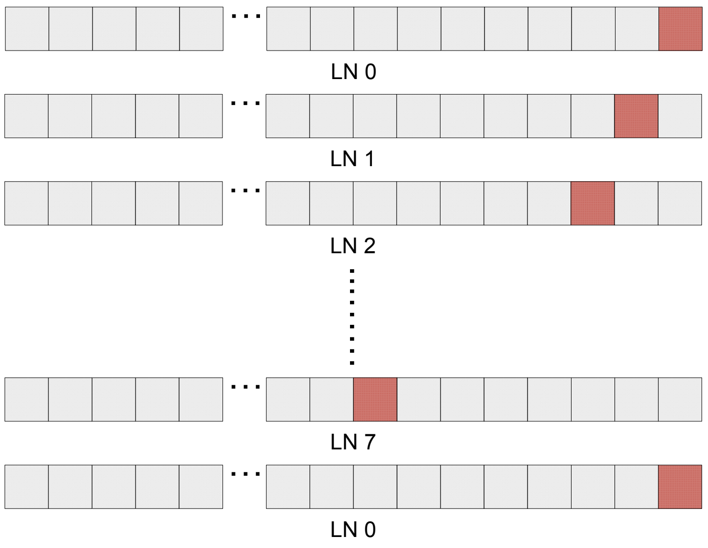


Figure 4.2 bit vector to indicate current LN

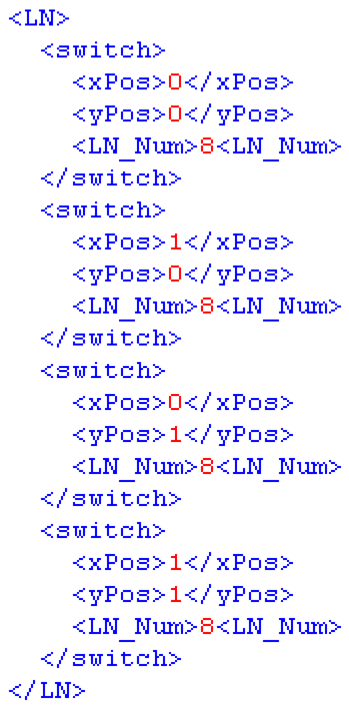
In this project, the size of the bit vector is fixed to 32, which implies that the simulator maximally support 32 LNs for each switch.

In the switches, the routing mechanism is kept in the new simulator. The only difference lies in that the preferred LN is no longer limited to 4 but can be any given number in the range [1, 32].

In this project, there are four kinds of LNs for the packets. The DEFAULT_LN means that the packet belongs to LN 0 but without guaranteed services. The ANY_LN now performs in the same way as the DEFAULT_LN. The BE_LN uses best effort QoS but it does not defines to which LN the packet belongs to. While the VC_LN uses virtual circuit QoS and does not decide the LN number.

As mentioned in chapter 1, for the sake of user configuration simplicity, the default value of the number of LNs in each switch is set to 4. To change the number of LNs for each switch, the user has to modify the XML file which configures the kernel of the simulator.

Figure 4.3 is an example of LN configuration in which the top left four switches have 8 LNs.



```
<LN>
  <switch>
    <xPos>0</xPos>
    <yPos>0</yPos>
    <LN_Num>8<LN_Num>
  </switch>
  <switch>
    <xPos>1</xPos>
    <yPos>0</yPos>
    <LN_Num>8<LN_Num>
  </switch>
  <switch>
    <xPos>0</xPos>
    <yPos>1</yPos>
    <LN_Num>8<LN_Num>
  </switch>
  <switch>
    <xPos>1</xPos>
    <yPos>1</yPos>
    <LN_Num>8<LN_Num>
  </switch>
</LN>
```

Figure 4.3 Example of LN configurations on kernel level

4.1.2 Application level generalization

Although the following generalization is called “application level”, almost all the implementation work is done in the kernel of the simulator. However, these methods are mainly used and functions are mainly called above the RNI level. They are more related to the applications which are compliant with AXI protocol.

Most of the work in this level is to solve the problem of how to separate the type and number of LN. For instance, in the old simulator, a virtual circuit LN which belongs to LN 0 is expressed as “VC_LN0”. Since there are fixed number of LNs, the type and number of LN can be enumerated. However, when the number of LNs is generalized, it is no longer possible to use the enumeration.

To solve the problem, two separate variables are used. The first variable defines the type of the LNs, which are, DEFAULT_LN, ANY_LN, BE_LN, or VC_LN. The second variable decides the LN number.

In the old application level input XML file, the terms such as “VC_LN0” are used. In this project, we use two parameters in the XML file, namely, connectionType and channelLN. Figure 4.4 illustrates how these parameters are used.

```
<Connection>
   <connectionId>0</connectionId>
   <connectionType>VC_LN</connectionType>
   <channelTDN>0</channelTDN>
   <VCConfigCollection>
     <VCConfig>
       <ln>0</ln>
       <SwitchElementCollection>
         <SwitchElement>
           <xPos>0</xPos>
           <yPos>0</yPos>
         </SwitchElement>
         <SwitchElement>
           <xPos>0</xPos>
           <yPos>1</yPos>
         </SwitchElement>
         <SwitchElement>
           <xPos>1</xPos>
           <yPos>1</yPos>
         </SwitchElement>
         <SwitchElement>
           <xPos>1</xPos>
           <yPos>0</yPos>
         </SwitchElement>
       </SwitchElementCollection>
     </VCConfig>
   </VCConfigCollection>
</Connection>
```

Figure 4.4 Example of LN configurations on application level

4.2 Implementation of Generalized LN

To illustrate how the generalization of LN is implemented, we follow the progress of this project.

Step 1: Generalize data types

There are two classes in which LN is defined. One is mpdu.h and the other is mp_if.h.

The mpdu is the data transmitted in the network. LN assigned to the mpdu is the LN to which the data belongs. In an object of the mpdu, chType variable defines whether the LN is of CHTYPE_DEFAULT_LN, CHTYPE_ANY_LN, CHTYPE_BE_LN or CHTYPE_VC_LN type. Port variable determines the LN number. These two variables are the ones used in the core of the network.

The mp_if is the interface between different layers. It controls to which channel the data will be transmitted. In this project, to generalize the number of LNs, the openChannel()

function has to be modified so that there are two separate parameters to express LN type and LN number.

The openChannel() function depicts the relationship between the LN related parameters in mpdu and those in mp_if. The mp_if parameters are mainly used in the layers above the RNI and the mpdu parameters are responsible for RNI, NI and switch layers. The relationship can be seen in figure 4.5.

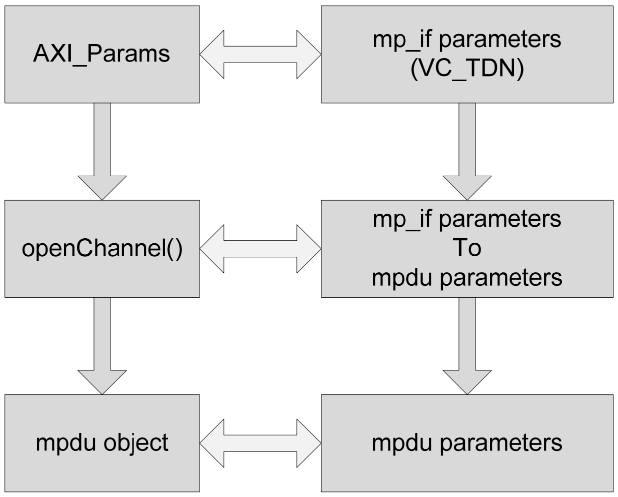

AXI_Params
mp_if parameters (VC_TDN)
openChannel()
mp_if parameters To mpdu parameters
mpdu object
mpdu parameters

Figure 4.5 Relation between mp_if and pdu objects

The implementation of all the virtual functions defined in mp_if including openChannel () function are implemented in the RNI. The openChannel () function is called by AXI_Interconnect_Master and AXI_Interconnect_Slave. Hence, these files have to be changed as well to fit the new openChannel () function.

To compliant with the separation of LN types and LN number, a member variable channelLN of integer type is added in the connectionInfo class in AXI_Setup.h.

When all the modifications in the two classes have been completed, we simply test the simulator with a simplified 2 X 2 mesh network and a write transaction. All the switches have 4 LNs.

Step 2: Create the vector for configurable number of LN

After the data types have been changed, the next phase of implementation is to assign the number of LN to each switch. The numbers are stored in an integer vector in simulation_params.h. The size of the vector is set to numberOfRNI which equals to the number of switches in the network. As mentioned before, the default number of LNs is 4, thus each member variable in the vector is initialized to 4.

Step 3: Generalize the number of LN and replace the constant value

It is obvious that in the Nostrum simulator, each switch and the NI and RNI attached to it should have the same number of LNs. Therefore, a member variable LN_Num which is the number of LNs is added in the definitions of switch, NI and RNI classes. Since the value is used in the constructors of these objects, the value assignment has to be completed as soon as the objects are created. Hence, value assignments are implemented in nostrum.cpp where RNI is created and network.cpp where switch and NI are created.

In the base simulator, the number of LNs in the switch, NI and RNI is determined by a constant value NUMBER_OF_LN which is 4. To generalize the number of LNs, we replace all the NUMBER_OF_LNs with the LN_Num in the objects.

Step 4: Generalize the switch, NI and RNI

As mentioned before, the size of the down streaming FIFOs has to be generalized according to the LN_Num of the RNI. In Nostrum simulator, there are two down streaming FIFOs: ds_data_fifo and ds_data_priority_fifo. They are resized to LN_Num in the constructor of the RNI.

In the NI class where the current LN is updated, the tdnCounter_process () has to be changed so that it maintains the correctness of the current LN. The modification is done by adjusting the rotation rate in the tdnCounter_process (). To correctly update the current LN, the rotation rate should be set according to the LN_Num instead of a constant value.

Informing the RNI of the current LN requires a tricky solution. There are two alternatives which may lead to different implementation complexities and work loads. The first alternative is to use an integer for the current LN. Obviously an integer is able to represent any LN number without upper bound. However, it may bring in much workload since the old simulator uses a SystemC bit vector to show the current LN. The second alternative aims at maintaining the consistency of the programming style and meanwhile, guaranteeing the system performance. In a system, there always has to be an upper bound of the number of LNs. Consequently, it is possible to increase the original bit vector to the upper bound instead of using an integer variable. In this project, we adopt the second way of implementation, and make the upper bound to 32.

Step 5: Testing

Until now, the implementation in the kernel has been completed. Different testing schemes are used to prove the correctness of the extension.

Step 6: Update the user interface

```
LN_Number.resize (Nostrum.numberOfRNI);

for (int i=0; i<Nostrum.numberOfRNI; i++){
  LN_Number[i] = 4;
}

xercesc::DOMNode* node;

node = xmlHandler->getNodeFromTagPath("LN",*root_node);

int tmp_xPos;
int tmp_yPos;
int tmp_LN;

if (node) {

  if (node->hasChildNodes()) {

    xercesc::DOMNodeList& children = *(node->getChildNodes());      // Create list of children

    for (int i = 0; i < (int) children.getLength(); i++){ // Examine children

xercesc::DOMNode* child = children.item(i);

if(XMLHandler::get_ELEMENT_Node(child)) {

  XMLHandler::setValue(*child, tmp_xPos, "xPos");
  XMLHandler::setValue(*child, tmp_yPos, "yPos");
  int tmpIndex = Nostrum.Network.sizeX * tmp_yPos + tmp_xPos;

  if ( XMLHandler::setValue(*child,tmp_LN,"LN_Num")) {
    LN_Number[tmpIndex] = tmp_LN;
    }
  }
} // for
}
}
```

Figure 4.6 An example of XML data parsing

After all the previous implementations and tests are done, the final step is to build the user interface which is the two XML input files in Nostrum simulator. The Xerces-C++ is used in the simulator to parse the input variable from XML file. As described in 4.1, both of the files are changed and new parsing functionalities are added in the simulation_params.cpp and AXI_Params.cpp. The code in figure 4.6 is an example of how the input data in XML file is parsed. To set a number of LNs for a certain switch, three parameters are required, namely, xPos, yPos and LN_Num. The xPos defines the position of the switch on the x-axis and yPos defines its position on the y-axis. Then, the user specified LN_Num is assigned to the corresponding switch according to its position.

Chapter 5
Evaluation

In this chapter the evaluation of this project is illustrated. It provides five different test cases on Nostrum NoC simulator to test the correctness of the implementation. The test cases prove the implementation in different aspects such as scale of the network, number of transactions, number of LNs and communication link overlapping schemes.

5.1 Overview

The five different testing schemes in this chapter prove the correctness of the extension. The first test case is an example of how the base simulator works with 4 LNs in all the switches. The second test case increases the number of LNs in all the switches to 8 and proves that the simulator can support a number of LNs other than 4. In order to show that each switch in the network is able to have a unique number of LNs, the test case 3 is performed. In this test case, three switches have 8 LNs and the other switch has 4 LNs. Test cases 4 and 5 expand the testing scheme in two other aspects. Test case 4 has two write transactions and two overlapped VC while test case 5 increases the scale of the network thus allows longer overlapped VC.

The simulation results are directly displayed on Linux terminal in text format. Except for the routing and the driving/receiving of AXI packets which are concerned in this project, it also displays much other information which is redundant for these test cases. Therefore, the concerned information is extracted and listed in table formats. From the tables, how a packet is routed throughout the network can be easily tracked. Since we have only predefined the VC for address write (AW) and write (W) packets, other packets are not listed in the tables due to the unpredictability.

5.2 Test Case 1

In this test case, a 2 X 2 mesh network in used with switch 1 connected to the master and switch 2 to the slave. All the switches are assigned with a LN number of 4. The AXI write transaction is set to LN0. The predefined VC in this test case is shown in figure 5.1.

The switches on the predefined VC path are configured as:

```
<SwitchConfigCollection>
  <SwitchConfig>
    <xPos>0</xPos>
    <yPos>0</yPos>
    <ln>0</ln>
    <inputDir>NI</inputDir>
    <outputDir>S</outputDir>
```

```
        </SwitchConfig>
        <SwitchConfig>
           <xPos>0</xPos>
           <yPos>1</yPos>
           <ln>0</ln>
           <inputDir>N</inputDir>
           <outputDir>E</outputDir>
        </SwitchConfig>
        <SwitchConfig>
           <xPos>1</xPos>
           <yPos>1</yPos>
           <ln>0</ln>
           <inputDir>W</inputDir>
           <outputDir>N</outputDir>
        </SwitchConfig>
        <SwitchConfig>
           <xPos>1</xPos>
           <yPos>0</yPos>
           <ln>0</ln>
           <inputDir>S</inputDir>
           <outputDir>NI</outputDir>
        </SwitchConfig>
  </SwitchConfigCollection>
```

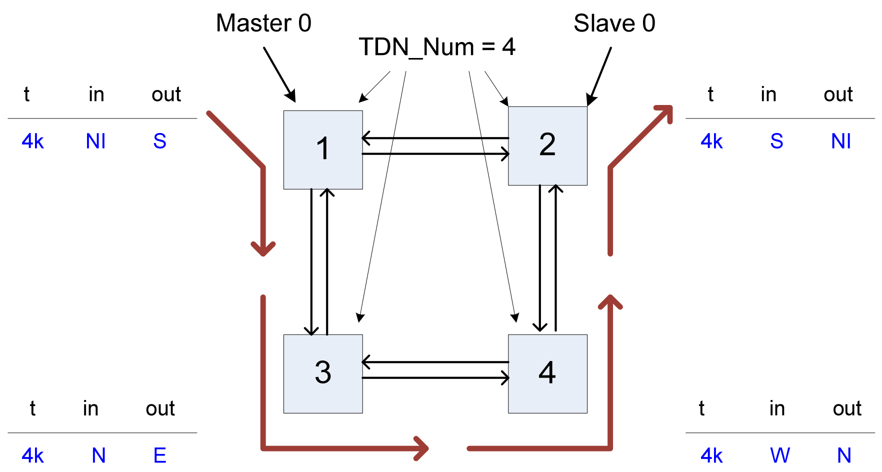


Figure 5.1 Predefined VC in test case 1

The aim of this test case is to show how the system works before the extension in this project.

The simulation result of test case 1 is listed in table 5.1.

Cycle	Switch	LN	AXI_behavior	Switch_behavior
50	Sw_0_0	2	AW Driven	
50	Sw_1_0	0		
50	Sw_0_1	0		
50	Sw_1_1	2		
51	Sw_0_0	3	W0 Driven	
51	Sw_1_0	1		
51	Sw_0_1	1		
51	Sw_1_1	3		
52	Sw_0_0	0	W1 Driven	AW is sent in diction 2 (south)
52	Sw_1_0	2		
52	Sw_0_1	2		
52	Sw_1_1	2		
53	Sw_0_0	1	W2 Driven	
53	Sw_1_0	3		
53	Sw_0_1	3		AW arrives Sw_0_1
53	Sw_1_1	1		
54	Sw_0_0	2	W3 Driven	
54	Sw_1_0	0		
54	Sw_0_1	0		AW is sent in direction 3 (east)
54	Sw_1_1	2		
55	Sw_0_0	3	W4 Driven	
55	Sw_1_0	1		
55	Sw_0_1	1		
55	Sw_1_1	3		AW arrives Sw_1_1
56	Sw_0_0	0		W0 is sent in direction 2 (south)
56	Sw_1_0	2		
56	Sw_0_1	2		
56	Sw_1_1	0		AW is sent in direction 0 (north)
57	Sw_0_0	1		
57	Sw_1_0	3		AW arrives Sw_1_0
57	Sw_0_1	3		W0 arrives Sw_0_1
57	Sw_1_1	1		

58	Sw_0_0	2		
58	Sw_1_0	0	AW Received	AW is sent in direction 4 (Ni)
58	Sw_0_1	0		W0 is sent in direction 3 (east)
58	Sw_1_1	2		
……	……			
62	Sw_1_0	0	W0 Received	……
……	……			
66	Sw_1_0	0	W1 Received	
……	……			

Table 5.1 Simulation Result for Test Case 1

The timing relationship among different AXI behaviors for the first write transaction is illustrated in figure 5.2.

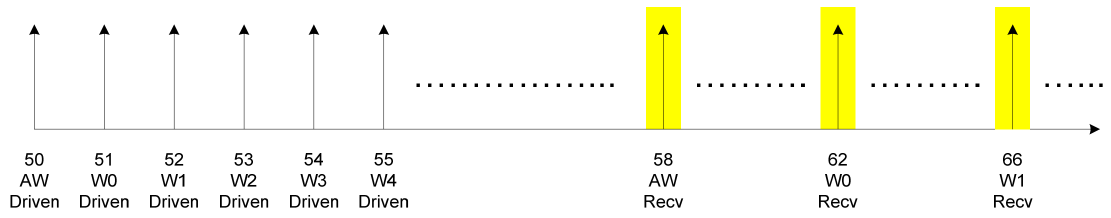


Figure 5.2 Timing relationship for test case 1

This is the original Nostrum NoC simulator in which the number of LNs for all the switches is fixed to four. From the table, it can be seen that:

- The packet occupies 1/4 of the communication link bandwidth
- Only at the LN which the packet belongs to, does the RNI route the packet into the network

5.3 Test Case 2

From this test case on, the numbers of LNs for all the switches have been generalized. In this case, a 2 X 2 mesh network in used with switch 1 connected to the master and switch 2 to the slave. All of the switches have 8 LNs. The AXI write transaction is set to LN0. The predefined VC in this test case is shown in figure 5.3.

The switches on the predefined VC path are configured as:

```
<SwitchConfigCollection>
```

```
      <SwitchConfig>
         <xPos>0</xPos>
         <yPos>0</yPos>
         <ln>0</ln>
         <inputDir>NI</inputDir>
         <outputDir>S</outputDir>
      </SwitchConfig>
      <SwitchConfig>
         <xPos>0</xPos>
         <yPos>1</yPos>
         <ln>0</ln>
         <inputDir>N</inputDir>
         <outputDir>E</outputDir>
      </SwitchConfig>
      <SwitchConfig>
         <xPos>0</xPos>
         <yPos>1</yPos>
         <ln>4</ln>
         <inputDir>N</inputDir>
         <outputDir>E</outputDir>
      </SwitchConfig>
      <SwitchConfig>
         <xPos>1</xPos>
         <yPos>1</yPos>
         <ln>4</ln>
         <inputDir>W</inputDir>
         <outputDir>N</outputDir>
      </SwitchConfig>
      <SwitchConfig>
         <xPos>1</xPos>
         <yPos>1</yPos>
         <ln>0</ln>
         <inputDir>W</inputDir>
         <outputDir>N</outputDir>
      </SwitchConfig>
      <SwitchConfig>
         <xPos>1</xPos>
         <yPos>0</yPos>
         <ln>0</ln>
         <inputDir>S</inputDir>
         <outputDir>NI</outputDir>
      </SwitchConfig>
</SwitchConfigCollection>
```

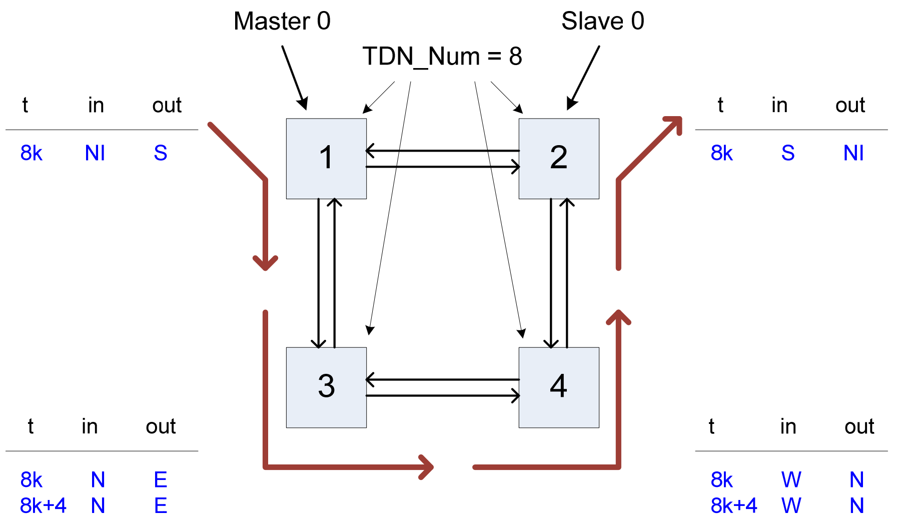


Figure 5.3 Predefined VC in test case 2

The aim of this test case is to test whether the new simulator can work correctly under a generalized number of LN.

The simulation result of test case 2 is listed in table 5.2.

Cycle	Switch	LN	AXI_behavior	Switch_behavior
50	Sw_0_0	2	AW Driven	
50	Sw_1_0	4		
50	Sw_0_1	4		
50	Sw_1_1	2		
51	Sw_0_0	3	W0 Driven	
51	Sw_1_0	5		
51	Sw_0_1	5		
51	Sw_1_1	3		
52	Sw_0_0	4	W1 Driven	
52	Sw_1_0	6		
52	Sw_0_1	6		
52	Sw_1_1	4		
53	Sw_0_0	5	W2 Driven	
53	Sw_1_0	7		
53	Sw_0_1	7		
53	Sw_1_1	5		

54	Sw_0_0	6	W3 Driven	
54	Sw_1_0	0		
54	Sw_0_1	0		
54	Sw_1_1	6		
55	Sw_0_0	7	W4 Driven	
55	Sw_1_0	1		
55	Sw_0_1	1		
55	Sw_1_1	7		
56	Sw_0_0	0		AW is sent in diction 2 (south)
56	Sw_1_0	2		
56	Sw_0_1	2		
56	Sw_1_1	0		
57	Sw_0_0	1		
57	Sw_1_0	3		
57	Sw_0_1	3		AW arrives Sw_0_1
57	Sw_1_1	1		
58	Sw_0_0	2		
58	Sw_1_0	4		
58	Sw_0_1	4		AW is sent in direction 3 (east)
58	Sw_1_1	2		
59	Sw_0_0	3		
59	Sw_1_0	5		
59	Sw_0_1	5		
59	Sw_1_1	3		AW arrives Sw_1_1
60	Sw_0_0	4		
60	Sw_1_0	6		
60	Sw_0_1	6		
60	Sw_1_1	4		AW is sent in direction 1 (north)
61	Sw_0_0	5		
61	Sw_1_0	7		AW arrives Sw_1_0
61	Sw_0_1	7		
61	Sw_1_1	5		
62	Sw_0_0	6		

62	Sw_1_0	0	AW received	
62	Sw_0_1	0		
62	Sw_1_1	6		
……	……			
70 …… 78 ……	Sw_1_0 …… Sw_1_0 ……	0 0	W0 received W1 received	

Table 5.2 Simulation Result for Test Case 2

The timing relationship among different AXI behaviors for the first write transaction is illustrated in figure 5.4.

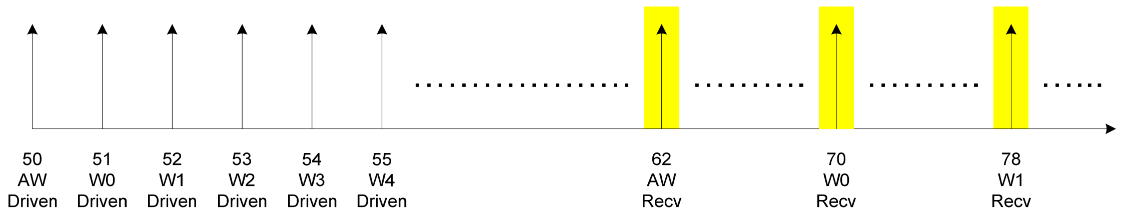


Figure 5.4 Timing relationship for test case 2

In this test case, all the switches have the LN number of 8, which means multiple of 1/8 link bandwidth is reserved by the predefined VC.

From the table, we can draw the conclusion that:

- The packet occupies 1/8 of the communication link bandwidth
- Only at the LN which the packet belongs to, does the RNI route the packet into the network

5.4 Test Case 3

In this test case, the master, slave, predefined VC path and the configuration of the switches alone the VC path are the same as they are in test case 2. The only difference is that in this test case, the number of LN for switch 4 is set to 4, which is different from the rest of the switches. Figure 5.5 depicts the configuration of this test case.

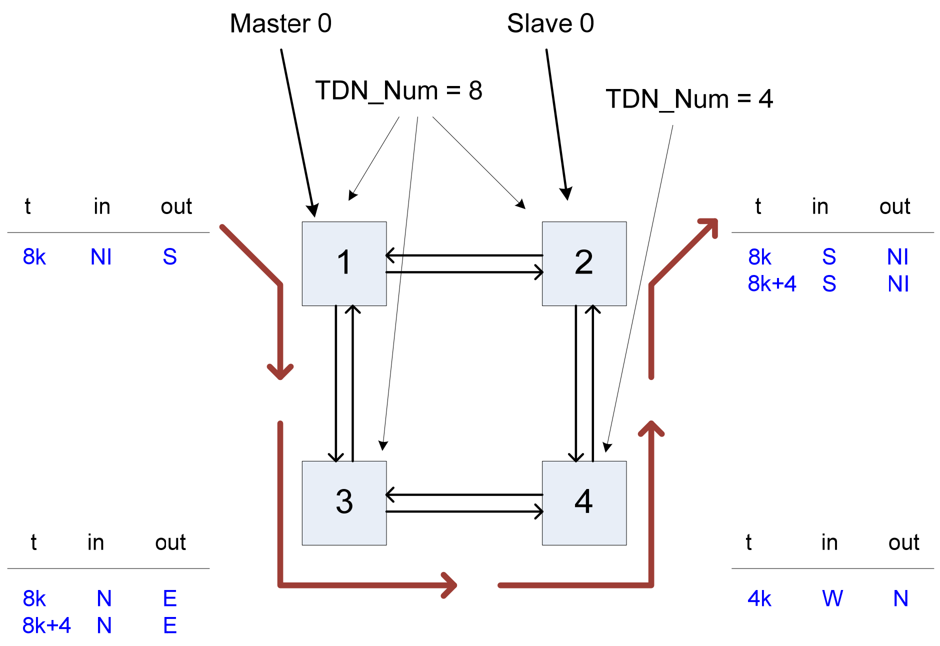


Figure 5.5 Predefined VC in test case 3

The purpose of this test case is to test that whether each switch in the network can have its unique number of LN. Here the switch 4 has a unique number of LN while the other three switches have another number of LN. Since all the switches on the predefined path except for switch 4 have 8 LNs which is a multiple of that of switch 4, theoretically it has no effect on the propagation time of the address write (AW) and write (W) packets in the first write transaction. However, when the current LN of switch 4 is 0, it is possible that no data is routed and consequently a time slot is "wasted". Therefore, if the result of this test case is the same as that of test case 2 for the AW and W in the first write transaction, it is proved that each switch in the network can have a unique number of LN and the simulator works correctly.

The simulation result of test case 3 is shown in table 5.3.

Cycle	Switch	LN	AXI_behavior	Switch_behavior
50	Sw_0_0	2	AW Driven	
50	Sw_1_0	4		
50	Sw_0_1	4		
50	Sw_1_1	2		
51	Sw_0_0	3	W0 Driven	
51	Sw_1_0	5		
51	Sw_0_1	5		

51	Sw_1_1	3		
52	Sw_0_0	4	W1 Driven	
52	Sw_1_0	6		
52	Sw_0_1	6		
52	Sw_1_1	0		
53	Sw_0_0	5	W2 Driven	
53	Sw_1_0	7		
53	Sw_0_1	7		
53	Sw_1_1	1		
54	Sw_0_0	6	W3 Driven	
54	Sw_1_0	0		
54	Sw_0_1	0		
54	Sw_1_1	2		
55	Sw_0_0	7	W4 Driven	
55	Sw_1_0	1		
55	Sw_0_1	1		
55	Sw_1_1	3		
56	Sw_0_0	0		AW is sent in diction 2 (south)
56	Sw_1_0	2		
56	Sw_0_1	2		
56	Sw_1_1	0		
57	Sw_0_0	1		
57	Sw_1_0	3		
57	Sw_0_1	3		AW arrives Sw_0_1
57	Sw_1_1	1		
58	Sw_0_0	2		
58	Sw_1_0	4		
58	Sw_0_1	4		AW is sent in direction 3 (east)
58	Sw_1_1	2		
59	Sw_0_0	3		
59	Sw_1_0	5		
59	Sw_0_1	5		
59	Sw_1_1	3		AW arrives Sw_1_1

60	Sw_0_0	4		
60	Sw_1_0	6		
60	Sw_0_1	6		
60	Sw_1_1	0		AW is sent in direction 1 (north)
61	Sw_0_0	5		
61	Sw_1_0	7		AW arrives Sw_1_0
61	Sw_0_1	7		
61	Sw_1_1	1		
62	Sw_0_0	6		
62	Sw_1_0	0	AW received	
62	Sw_0_1	0		
62	Sw_1_1	2		
……	……			
70 …… 78 ……	Sw_1_0 …… Sw_1_0 ……	0 0	W0 received W1 received	

Table 5.3 Simulation Result for Test Case 3

The only difference between test case 3 and test case 2 is that the number of LN for switch 4 is 4 instead of 8 in test case 3. Since the number of LNs for switch 4 is half of that in other switches, it is expected that the change in LN number has no effect on the arrival time of AW and W packets in the first write transaction.

The timing relationship among different AXI behaviors for the first write transaction is illustrated in figure 5.6.

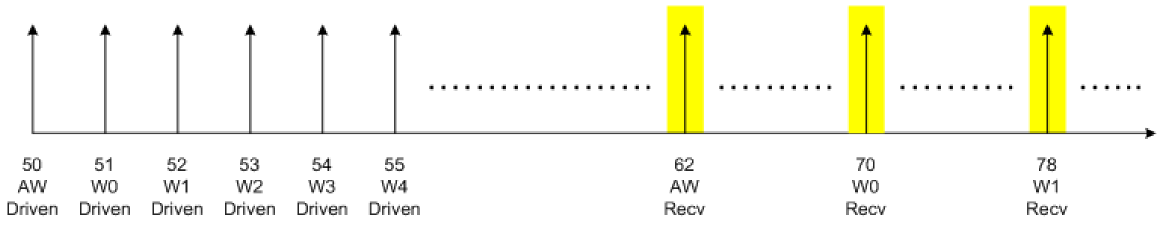


Figure 5.6 Timing relationship for test case 3

From table 5.3, it is obvious that:

- The packet occupies 1/8 of the communication link bandwidth

- Only at the LN which the packet belongs to, does the RNI route the packet into the network
- The result is identical with that of test case 2.

In this test case, switch 4 has only 4 LNs while it has 8 LNs in test case 2. Other switches in both test cases have 8 LNs. Consequently, as shown in figure 5.7, half of the reserved time slots of switch 4 are "wasted" in test case 3. The solid box means that a packet is being routed from the switch on the reserved time slot and the hollow box represents the reserved time slot on which no packet is routed.

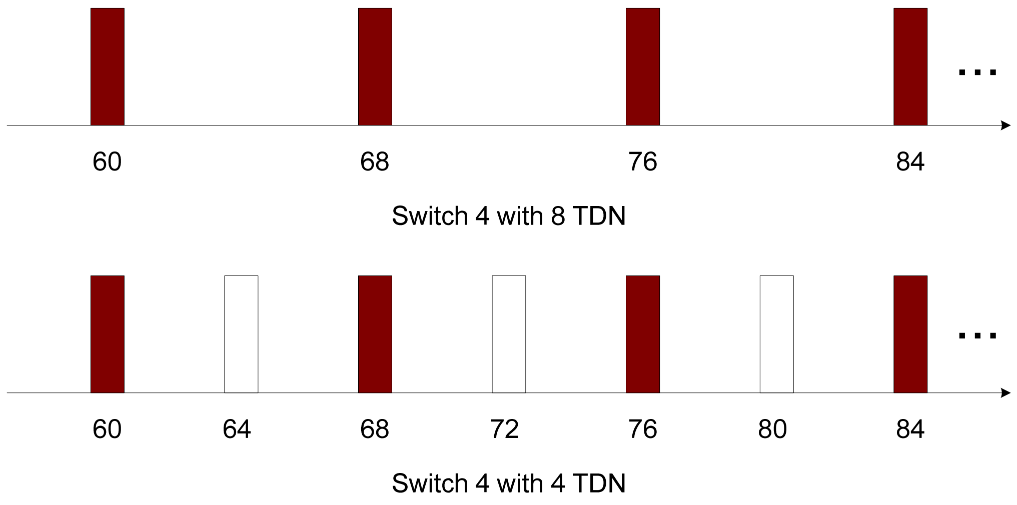


Figure 5.7 Comparison of reserved time slot for switch 4

5.5 Test Case 4

In this test case, a 2 X 2 Mesh network is adopted. Different from the previous cases, there are two write transactions in the network which means that there are two masters and two slaves connected to the four switches. In other words, each switch is assigned with a master or slave. The write transaction 1 is assigned to LN 0 and write transaction 2 is to LN 1. In this configuration, Master 0 locates at switch 1 and Master 1 locates at 3. For the slaves, Slave 0 is set to switch 2 and Slave 1 to switch 4. Each switch has 8 LNs in this test case. The configuration is shown in figure 5.8.

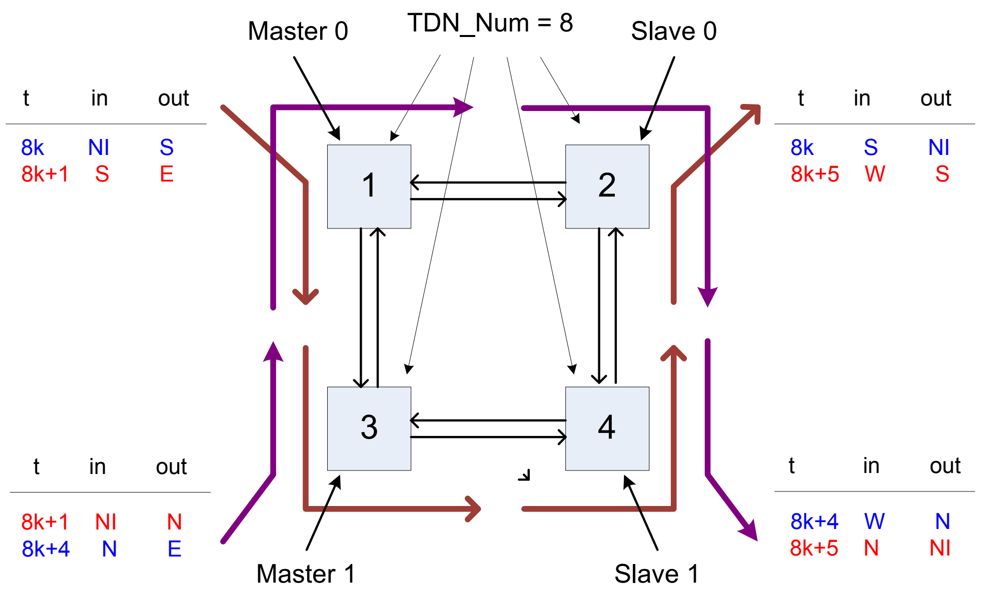


Figure 5.8 Predefined VC in test case 4

In this test case, the switches alone the two predefined VC paths need to be carefully configured. The configuration is shown as below:

```
<MasterCollection>
   <MasterInfo>
      <masterId>0</masterId>
      <pid>100</pid>
      <nid>1</nid>
   </MasterInfo>
   <MasterInfo>
      <masterId>1</masterId>
      <pid>300</pid>
      <nid>3</nid>
   </MasterInfo>
</MasterCollection>
<SlaveCollection>
   <SlaveInfo>
      <slaveId>0</slaveId>
      <pid>200</pid>
      <nid>2</nid>
   </SlaveInfo>
   <SlaveInfo>
      <slaveId>1</slaveId>
      <pid>400</pid>
      <nid>4</nid>
   </SlaveInfo>
```

```
</SlaveCollection>
<SwitchConfigCollection>
   <SwitchConfig>
      <xPos>0</xPos>
      <yPos>0</yPos>
      <ln>0</ln>
      <inputDir>NI</inputDir>
      <outputDir>S</outputDir>
   </SwitchConfig>
   <SwitchConfig>
      <xPos>0</xPos>
      <yPos>1</yPos>
      <ln>4</ln>
      <inputDir>N</inputDir>
      <outputDir>E</outputDir>
   </SwitchConfig>
   <SwitchConfig>
      <xPos>1</xPos>
      <yPos>1</yPos>
      <ln>4</ln>
      <inputDir>W</inputDir>
      <outputDir>N</outputDir>
   </SwitchConfig>
   <SwitchConfig>
      <xPos>1</xPos>
      <yPos>0</yPos>
      <ln>0</ln>
      <inputDir>S</inputDir>
      <outputDir>NI</outputDir>
   </SwitchConfig>
   <SwitchConfig>
      <xPos>0</xPos>
      <yPos>1</yPos>
      <ln>1</ln>
      <inputDir>NI</inputDir>
      <outputDir>N</outputDir>
   </SwitchConfig>
   <SwitchConfig>
      <xPos>0</xPos>
      <yPos>0</yPos>
      <ln>1</ln>
      <inputDir>S</inputDir>
      <outputDir>E</outputDir>
   </SwitchConfig>
   <SwitchConfig>
      <xPos>1</xPos>
      <yPos>0</yPos>
      <ln>5</ln>
      <inputDir>W</inputDir>
      <outputDir>S</outputDir>
    </SwitchConfig>
    <SwitchConfig>
      <xPos>1</xPos>
      <yPos>1</yPos>
      <ln>5</ln>
      <inputDir>N</inputDir>
      <outputDir>NI</outputDir>
    </SwitchConfig>
  </SwitchConfigCollection>
```

This test case is designed to test that whether the new simulator is able to work correctly with two different transactions from different masters and slaves and still free of contention.

The result of test case 4 is depicted in table 5.4.

Cycle	Switch	LN	Flow ID	AXI_behavior	Switch_behavior
50	Sw_0_0	2	1	AW Driven	
50	Sw_1_0	4			
50	Sw_0_1	4			
50	Sw_1_1	2			
51	Sw_0_0	3	1	W0 Driven	
51	Sw_1_0	5			
51	Sw_0_1	5	2	AW Driven	
51	Sw_1_1	3			
52	Sw_0_0	4	1	W1 Driven	
52	Sw_1_0	6			
52	Sw_0_1	6	2	W0 Driven	
52	Sw_1_1	4			
53	Sw_0_0	5	1	W2 Driven	
53	Sw_1_0	7			
53	Sw_0_1	7	2	W1 Driven	
53	Sw_1_1	5			
54	Sw_0_0	6	1	W3 Driven	
54	Sw_1_0	0			

54	Sw_0_1	0	2	W2 Driven	
54	Sw_1_1	6			
55	Sw_0_0	7	1	W4 Driven	
55	Sw_1_0	1			
55	Sw_0_1	1	2	W3 Driven	AW is sent in direction 0 (north)
55	Sw_1_1	7			
56	Sw_0_0	0	1 2		AW is sent in direction 2 (south) AW arrives Sw_0_0
56	Sw_1_0	2			
56	Sw_0_1	2	2	W4 Driven	
56	Sw_1_1	0			
57	Sw_0_0	1	2		AW is sent in direction 3 (east)
57	Sw_1_0	3			
57	Sw_0_1	3	1		AW arrives Sw_0_1
57	Sw_1_1	1			
58	Sw_0_0	2			
58	Sw_1_0	4	2		AW arrives Sw_1_0
58	Sw_0_1	4	1		AW is sent in direction 3 (east)
58	Sw_1_1	2			
59	Sw_0_0	3			
59	Sw_1_0	5	2		AW is sent in direction 2 (south)
59	Sw_0_1	5			
59	Sw_1_1	3	1		AW arrives Sw_1_1
60	Sw_0_0	4			
60	Sw_1_0	6			
60	Sw_0_1	6			
60	Sw_1_1	4	1 2		AW is sent in direction 1 (north) AW arrives Sw_1_1
61	Sw_0_0	5			
61	Sw_1_0	7	1		AW arrives Sw_1_0
61	Sw_0_1	7			
61	Sw_1_1	5	2	AW received	
62	Sw_0_0	6			
62	Sw_1_0	0	1	AW received	

62	Sw_0_1	0			
62	Sw_1_1	6			
……	……				
69	Sw_1_1	5	2	W0 received	
……	……				
70 …… 77 …… 78 ……	Sw_1_0 …… Sw_1_1 …… Sw_1_0 ……	0 5 0	1 2 1	W0 received W1 received W1 received	

Table 5.4 Simulation Result for Test Case 4

In this test case, two write transaction flows are used. Flow 1 is assigned to LN 0 and Flow 2 is to LN 1. The predefined paths have two overlapped communication links.

The timing relationship among different AXI behaviors for the first write transaction is illustrated in figure 5.9.

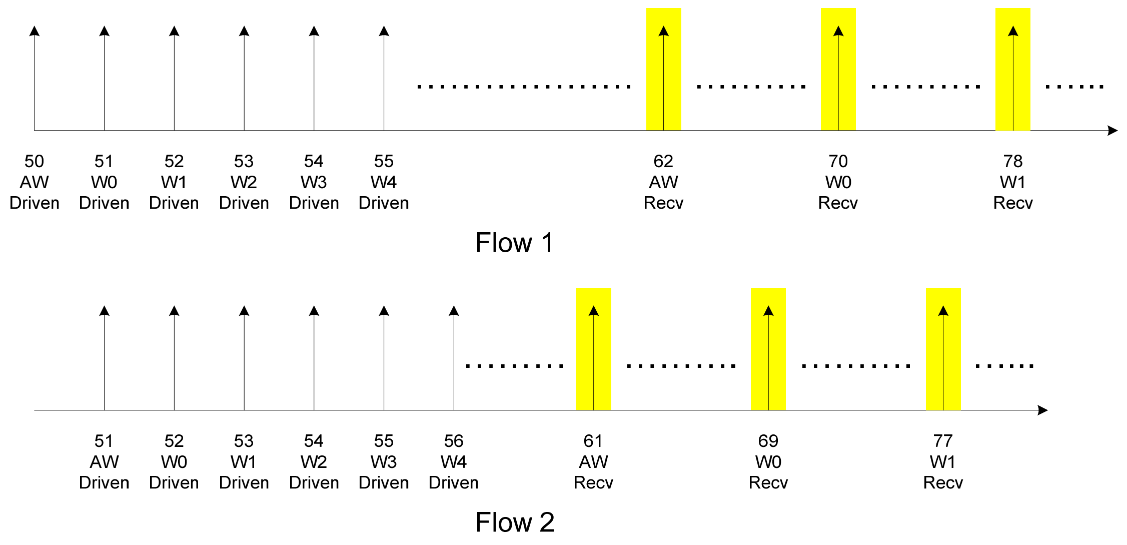


Figure 5.9. Timing relationship for test case 4

From the table, the following conclusions can be drawn:

- The packet occupies 1/8 of the communication link bandwidth

- Only at the LN which the packet belongs to, does the RNI route the packet into the network
- After the generalization of the number of LNs of the simulator, more than one transaction from different masters and slaves are still supported and free of contention.

5.6 Test Case 5

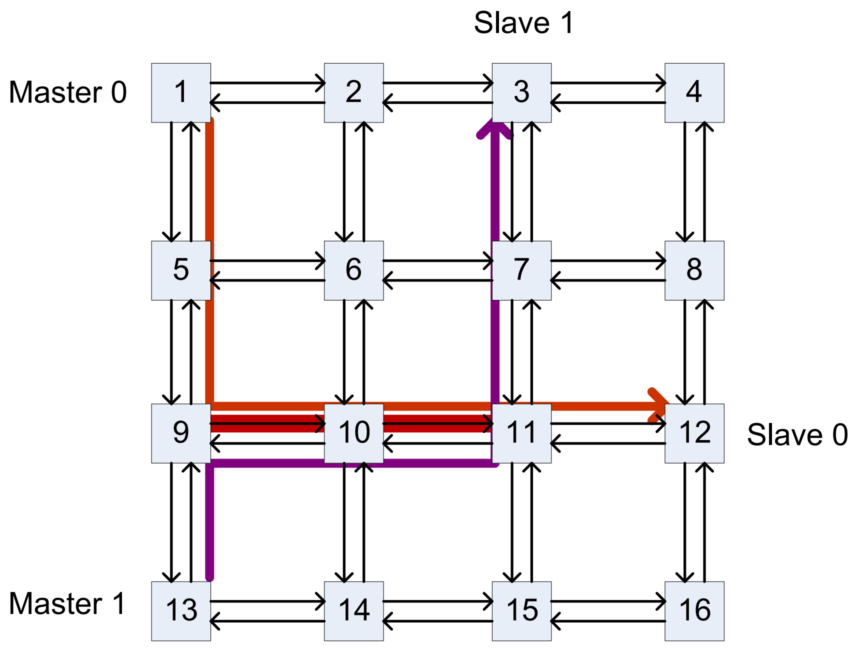
In this test case, the scale of the network is expanded to 4 X 4 which has sixteen switches and maximally can support sixteen masters or slaves. However, we also use two write transactions as we did in test case 4. Firstly, the two transaction scheme is able to provide overlapped communication links between different predefined VC paths. Moreover, it is a relatively simple scheme so that we can predict the theoretical result and compare it with the actual simulation result. In this case, the two masters respectively locate at switch 1 and switch 13 while two slaves at switch 12 and switch 3. Each switch in the network has 8 LNs. The configuration is shown in figure 5.10.

The longer the predefined VC path is, the more careful configuration for the switches alone the path is needed. In this test case, the following configuration is used:

```
<MasterCollection>
   <MasterInfo>
      <masterId>0</masterId>
      <pid>100</pid>
      <nid>1</nid>
   </MasterInfo>
   <MasterInfo>
      <masterId>1</masterId>
      <pid>300</pid>
      <nid>13</nid>
   </MasterInfo>
</MasterCollection>
<SlaveCollection>
   <SlaveInfo>
      <slaveId>0</slaveId>
      <pid>200</pid>
      <nid>12</nid>
   </SlaveInfo>
   <SlaveInfo>
      <slaveId>1</slaveId>
      <pid>400</pid>
      <nid>3</nid>
   </SlaveInfo>
</SlaveCollection>
<SwitchConfigCollection>
   <SwitchConfig>
```

```
    <xPos>0</xPos>
    <yPos>0</yPos>
    <ln>0</ln>
    <inputDir>NI</inputDir>
    <outputDir>S</outputDir>
</SwitchConfig>
<SwitchConfig>
    <xPos>0</xPos>
    <yPos>1</yPos>
    <ln>4</ln>
    <inputDir>N</inputDir>
    <outputDir>S</outputDir>
</SwitchConfig>
<SwitchConfig>
    <xPos>0</xPos>
    <yPos>2</yPos>
    <ln>4</ln>
    <inputDir>N</inputDir>
    <outputDir>E</outputDir>
</SwitchConfig>
<SwitchConfig>
    <xPos>1</xPos>
    <yPos>2</yPos>
    <ln>0</ln>
    <inputDir>W</inputDir>
    <outputDir>E</outputDir>
</SwitchConfig>
<SwitchConfig>
    <xPos>2</xPos>
    <yPos>2</yPos>
    <ln>0</ln>
    <inputDir>W</inputDir>
    <outputDir>E</outputDir>
</SwitchConfig>
<SwitchConfig>
    <xPos>3</xPos>
    <yPos>2</yPos>
    <ln>4</ln>
    <inputDir>W</inputDir>
    <outputDir>NI</outputDir>
</SwitchConfig>
<SwitchConfig>
    <xPos>0</xPos>
    <yPos>3</yPos>
    <ln>1</ln>
    <inputDir>NI</inputDir>
```

```
        <outputDir>N</outputDir>
    </SwitchConfig>
    <SwitchConfig>
        <xPos>0</xPos>
        <yPos>2</yPos>
        <ln>1</ln>
        <inputDir>S</inputDir>
        <outputDir>E</outputDir>
    </SwitchConfig>
    <SwitchConfig>
        <xPos>1</xPos>
        <yPos>2</yPos>
        <ln>5</ln>
        <inputDir>W</inputDir>
        <outputDir>E</outputDir>
    </SwitchConfig>
    <SwitchConfig>
        <xPos>2</xPos>
        <yPos>2</yPos>
        <ln>5</ln>
        <inputDir>W</inputDir>
        <outputDir>N</outputDir>
    </SwitchConfig>
    <SwitchConfig>
        <xPos>2</xPos>
        <yPos>1</yPos>
        <ln>1</ln>
        <inputDir>S</inputDir>
        <outputDir>N</outputDir>
    </SwitchConfig>
    <SwitchConfig>
        <xPos>2</xPos>
        <yPos>0</yPos>
        <ln>1</ln>
        <inputDir>S</inputDir>
        <outputDir>NI</outputDir>
    </SwitchConfig>
</SwitchConfigCollection>
```



Sw_1:

t	in	out
8k	NI	S

Sw_5:

t	in	out
8k+4	N	S

Sw_9:

t	in	out
8k+4	N	E
8k+1	S	E

Sw_10:

t	in	out
8k	W	E
8k+5	W	E

Sw_11:

t	in	out
8k	W	E
8k+5	W	N

Sw_12:

t	in	out
8k+4	W	NI

Sw_13:

t	in	out
8k+1	NI	N

Sw_7:

t	in	out
8k+1	S	N

Sw_3:

t	in	out
8k+1	S	NI

Figure 5.10 Predefined VC in test case 5

The intention of this test case is to test the two write transaction with different LNs. This scheme is more convincing than the previous ones due to the fact that it has larger network scale and longer overlapped predefined VC path.

The result of test case 5 is depicted in table 5.5.

Cycle	Switch	LN	Flow ID	AXI_behavior	Switch_behavior
50	Sw_0_0	2	1	AW Driven	
51	Sw_0_3	5	2	AW Driven	
51	Sw_0_0	3	1	W0 Driven	
52	Sw_0_3	6	2	W0 Driven	
52	Sw_0_0	4	1	W1 Driven	
53	Sw_0_3	7	2	W1 Driven	
53	Sw_0_0	5	1	W2 Driven	
54	Sw_0_3	0	2	W2 Driven	
54	Sw_0_0	6	1	W3 Driven	
55	Sw_0_3	1	2	W3 Driven	AW is sent in direction 0 (north)
55	Sw_0_0	7	1	W4 Driven	
56	Sw_0_3	2	2	W4 Driven	
56	Sw_0_0	0	1		AW is sent in direction 2 (south)
56	Sw_0_2	0	2		AW arrives Sw_0_2
57	Sw_0_2	1	2		AW is sent in direction 3 (east)
57	Sw_0_1	3	1		AW arrives Sw_0_1
58	Sw_1_2	4	2		AW arrives Sw_1_2
58	Sw_0_1	4	1		AW is sent in direction 2 (south)
59	Sw_1_2	5	2		AW is sent in direction 3 (east)
59	Sw_0_2	3	1		AW arrives Sw_0_2
60	Sw_2_2	4	2		AW arrives Sw_2_2
60	Sw_0_2	4	1		AW is sent in direction 3 (east)
61	Sw_2_2	5	2		AW is sent in direction 0 (north)
61	Sw_1_2	7	1		AW arrives Sw_1_2
62	Sw_2_1	0	2		AW arrives Sw_2_1
62	Sw_1_2	0	1		AW is sent in direction 3 (east)
63	Sw_0_3	1	2		W0 is sent in direction 0 (north)

63	Sw_2_1	1	2		AW is sent in direction 0 (north)
63	Sw_2_2	7	1		AW arrives Sw_2_2
64	Sw_0_0	0	1		W0 is sent in direction 2 (south)
64	Sw_0_2	0	2		W0 arrives Sw_0_2
64	Sw_2_0	0	2		AW arrives Sw_2_0
64	Sw_2_2	0	1		AW is sent in direction 3 (east)
65	Sw_0_2	1	2		W0 is sent in direction 3 (east)
65	Sw_0_1	3	1		W0 arrives Sw_0_1
65	Sw_2_0	1	2	AW received	
65	Sw_3_2	3	1		AW arrives Sw_3_2
66	Sw_3_2	4	1	AW received	
......	......				
73	Sw_2_0	1	2	W0 received	
......	......				
74 ...... 81 82	Sw_3_2 ...... Sw_2_0 ...... Sw_3_2	4 1 4	1 2 1	W0 received W1 received W1 received	

Table 5.5 Simulation Result for Test Case 5

In this test case, the network scale is enlarged to 4 X 4. Thus, it is easier to have a longer overlapped VC path. By enlarging the network scale, we can also draw the conclusion that the simulator is able to be expanded to any scale.

The timing relationship among different AXI behaviors for the first write transaction is illustrated in figure 5.11.

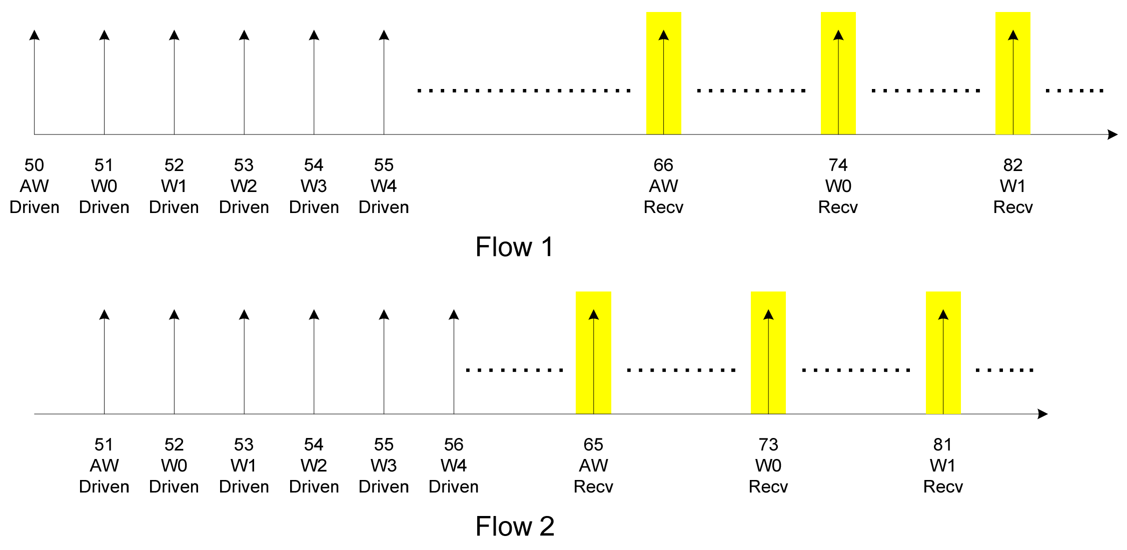


Figure 5.11. Timing relationship for test case 5

To conclude this test case, the proved characteristics are listed below.

- The packet occupies 1/8 of the communication link bandwidth
- Only at the LN which the packet belongs to, does the RNI route the packet into the network
- By the careful configuration of the network and switches, the overlapped communication links are free of contention
- The simulator can be expanded to any scale

Since the Nostrum NoC simulator uses deflective routing mechanism, packets can not be buffered in the switch. When more than one packet with the same output direction enters a switch, they are routed according to their priorities. The priority is decided by both the TDM VC and the age of the packet. Therefore, configuration has to be made for the switches to guarantee that the packets will follow the predefined path.

5.7 Summary

The five test cases are designed in a progressive manner. The first case illustrates how the TDM VC works in the base simulator. The second case proves that the new simulator is able to support a generalized number of LNs. However, it sets all the switches with the same number of LNs. Case 3 removes this limitation by setting one of the switches to a different number of LNs. Case 4 and 5 expand the testing scheme in two other dimensions. Case 4 expands the number of AXI transaction flows from one to two and forms two overlapped communication links. Case 5 enlarges the network scale and proves that the simulator is free of contention for any network scale.

In a write transaction, the propagation time for the data from the master to the slave via the predefined virtual circuit can be calculate as following:

TotalTime = [0, LN_Num] + 2 x PathLength

The first parameter shows the time that the data has to wait in the master side for its LN. For example, if the switch has 8 LNs and the current LN is 2, then the data has to wait for (8 - 2) = 6 clock cycles. The second parameter is the time for routing the data in the network. It depends on how long the predefined virtual circuit is. The process is shown in figure 5.12.

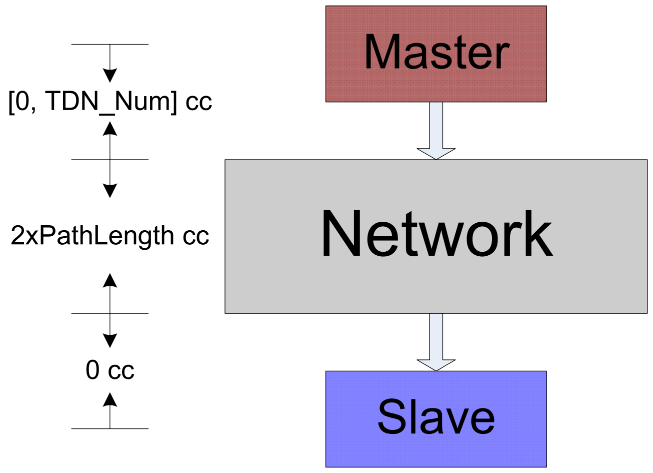


Figure 5.12 Propagation time of data transmission

Chapter 6
Summary and Conclusion

6.1 Summary

The goal of this thesis project is to generalize the number of LNs for the switches in Nostrum NoC simulator from a fixed constant to arbitrary configurable numbers. The number of LNs for each switch should be able to be different from others in the network. After the number of LNs is generalized, the whole NoC system should still be free of contention when the predefined VC paths overlapped with each other. To make the extension functionalities on the simulator user friendly, all the LN related parameters can be assigned in the two XML input files.

During the implementation phase, three programming languages/tools are used, namely C++, SystemC and Xerces-C++. C++ and SystemC are used for implementing the core of the simulator while Xerces-C++ is a tool to parse the XML input files and transfer configuration parameters.

In the testing phase, five different test cases are used to fully guarantee that the extended simulator works correctly. Each test case focuses on a unique aspect of the system. Test case 1 describes how the simulator works before the extension. In test case 2, all the switches in the network have 8 LNs. In test case 3, one of the switches has a different number of LNs from other switches. Test case 4 extends the number of information flows and thus creates two overlapped communication links. At last, test case 5 expands the scale of the network and has a longer overlapped predefined VC path.

The results of all the five test cases prove that the extension on the simulator is successful and therefore, the aim of the thesis project has been reached.

The difficulties that I have met during the thesis project are as follows.

Firstly, at the beginning of the project, I have been focused on the implementation of the AXI protocol on the simulator which is mainly above the RNI layers in the protocol stack of Nostrum. While the most of work in generalizing the number of LNs lies in the layers below the RNI. However, this problem turns out to be highly beneficial to me since it gives me deeper understandings in all the layers of the simulator.

Secondly, the simulator has never been set up anywhere except for the server in KTH. There is no instruction on how to set up the simulator. Moreover, several third party programming tools are required in the simulator. It took almost two weeks just to set up the simulator and configure the programming environment.

Thirdly, since the C++ compiler is improving all the time, some parts of the program in the old compiler may not be supported in the new version. In this project, the simulator which runs correctly on KTH server is not supported by the more up-to-date compiler on my workstation unless some parts of the code are changed. To find the differences in different versions of compiler is a time consuming process.

Fourthly, I did not have any experience in Linux system or Linux programming. However, the simulator only works under Linux environment. It took some time for me to get familiar with programming in Linux.

Lastly, implementing new features on an existed program is different from writing a new program. The first phase of the project is to understand the simulator. Due the lack of documentations on the simulator structure, it is hard to have a deep understanding of a program which has tens of files located in different folders.

Via the progress of the thesis project, I have conquered all these difficulties with the help from my supervisor, colleagues and friends. It gives me more experience in doing research and solving problems.

6.2 Future Work

Until now, the number of LNs for the switches can only be the multiples of 4. This is because that if the number of LNs is not a multiple of 4, e.g. if the number of LNs is 3 for the switches, the simulator can not guarantee the free of contention for packets on different LNs. The categorization and initialization of LNs for switches, combining with the theories proposed in [8], will help to solve the problem.

Another interesting topic is the hardware implementation of the LN mechanism. Via the hardware implementation at RTL level, the cost due to the generalization can be clearly shown. The extra cost stems from the difference between functional model and solid implementation. After generalization, the size of the FIFO, pipeline schemes and sorting algorithms will inevitably require more area cost. Comparing with the fixed number of LN, the generalization will have more hardware cost; however, it provides the finer bandwidth of the network and thus enhance the utilization.

Reference

[1] Axel Jantsch and Hannu Tenhunen. Networks on Chip, Chapter 1. Kluwer Academic Publishers. 2004

[2] Zhonghai Lu. Design and Analysis of On-Chip Communication for Network-on-Chip Platforms. Ph.D. thesis. Royal Institute of Technology, March 2007

[3] Axel Jantsch and Hannu Tenhunen. Networks on Chip, Chapter 5. Kluwer Academic Publishers. 2004

[4] Mikael Millberg, Erland Nilsson, Rikard Thid, Shashi Kumar, and Axel Jantsch. The Nostrum backbone - a communication protocol stack for networks on chip. In *Proceedings of the VLSI Design Conference*, Mumbai, India, January 2004

[5] K.Goossens, J. van Meerbergen, A. Peeters, and P. Wielage. Networks on silicon: Combining best-effort and guaranteed services. In Proceedings of Design Automation and Test Conference in Europe, March 2002

[6] ARM Limited. AMBA AXI Protocol v1.0 Specification. 2004

[7] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the Design Automation and Test Europe Conference (DATE)*, February 2004

[8] Zhonghai Lu and Axel Jantsch. Slot allocation using logical networks for TDM virtual-circuit configuration for network-on-chip. In *International Conference on Computer Aided Design (ICCAD)*, November 2007

[9] James A. Rowson and Alberto Sangiovanni-Vincentelli. Interfacebased design. In Proc. of the 34th Design Automation Conference, 1997

[10] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 19(12):1523–1543, Decmber 2000

[11] Marco Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In Proceedings of the 38th Design Automation Conference, June 2001

[12] Semiconductor Industry Association. Interbational technology roadmap for semiconductors. Technical report, World Semiconductor Council, 1999. Edition 1999

[13] William James Dally and Brian Towles. Principles and Practices of Interconnection Networks, Chapter 1. Morgan Kaufmann Publishers. 2004

[14] http://en.wikipedia.org/wiki/Mesh_network

[15] Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, Mahmut Kandemir. Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. 33rd International Symposium on Computer Architecture, 2006. ISCA '06.

[16] http://en.wikipedia.org/wiki/OSI_model

[17] http://en.wikipedia.org/wiki/QoS

[18] William James Dally and Brian Towles. Principles and Practices of Interconnection Networks, Chapter 15. Morgan Kaufmann Publishers. 2004

[19] http://en.wikipedia.org/wiki/Circuit_switching

[20] http://en.wikipedia.org/wiki/Packet_switching
[21] T. Marescaux, B. Bricke, P. Debacker, V. Nollet, H. Corporaal. Dynamic time-slot allocation for QoS enabled networks on chip. 3rd Workshop on Embedded Systems for Real-Time Multimedia, 2005
[22] William James Dally and Brian Towles. Principles and Practices of Interconnection Networks, Chapter 13. Morgan Kaufmann Publishers. 2004
[23] http://en.wikipedia.org/wiki/Time-division_multiplexing
[24] Rikard Thid. A network on chip simulator. Master's thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, IMIT/LECS 2002-17, August 2002
[25] Zhonghai Lu. A User Introduction to NNSE: Nostrum Network-on-Chip Simulation Environment. Royal Institute of Technology, Stockholm, November 2005
[26] AMBA AXI Protocol v1.0 Specification. 2003-2004 ARM Limited