Network On Chip implementation for Tinuso multicore configurations

Rune Ploug

Kongens Lyngby, 2011
IMM-B.Eng-2010-67

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

The DTU IMM ESE section is developing the many-core FPGA based Tinuso processor for research into future architectures of multi- and many-core processors. The focus of this thesis is to develop the infrastructure needed for Tinuso cores to access main memory over a Network On Chip (NOC).

The reason a NOC is interesting is that the Tinuso core which is designed for multi- and many-core implementations can be implemented with more than one core in a Tinuso processor architecture. For implementing this multi-core architecture different technologies is needed - one of the main features being the NOC.

The Tinuso core is already developed. It can directly access memory via a interface. To access memory with multiple cores in one processor a NOC is needed. This requires then a Network Interface Controller to interface between core and NOC is needed. Also a routing & switch component and a memory controller with network interface are needed.

In this project a NOC solution is designed for the Tinuso processor cores so that these cores can access main memory over the NOC. This was designed by analyzing current theory and design concepts based on specific prioritization of requirements. This prioritization was achieved via agile analysis of the base requirements, ideas for extensions and division of the base requirements into sub components and technologies.

The design chosen was a Torus 2D mesh with a YX routing algorithm with a twist of torus routing.

The design was implemented in VHDL for FPGA's and tested towards a simulated Tinuso core interface. This interface was design in cooperation with the Tinuso core developer[1] which developed his own test-bench. As a result slight differences was found between the actual core's expectations and what was used in the simulations here. The differences are documented in results.

Clock frequency results obtained from synthesis indicates that the NOC has to run at about half the clock frequency of the Tinuso cores[1]. This was expected as the implementation has not been optimized for fast clock frequency. This is the raw results from first total system test of an experimental prototype: the system is synthesized on a FPGA that is too small to even map it in synthesis.

The implemented solution demonstrates the feasibility of the design and network protocol when tested. It also demonstrates how many challenges there are in designing and implementing deadlock free solutions in concurrent systems. Concurrent access to the main memory from multiple cores failed in many cases in the test-bench as a result of a specific deadlock situation.

Most of the deadlocks were even expected as the testing went outside the requirements for this version of the NOC system. This was done to test support for interesting extensions such as core to core communication or and cache coherency.

Several solutions for the few deadlocks situations experienced in testing has been suggested demonstrating that there is many ways, with different tradeoffs, to handle deadlocks.

Resume

DTU's IMM ESE sektion udvikler den mange-kernede FPGA baserede Tinuso processor til forskning i fremtidige arkitekturer inden for multi- og mange-kernede processorer. Fokus i dette Diplom speciale er at udvikle infrastrukturen, som er nødvendig for at Tinuso kerner kan tilgå den primære hukommelse over et Network On Chip (NOC) design. Dette netværk er optimeret til intern kommunikation på chippen.

Grunden til at en NOC er interessant er at Tinuso kernen, som er designet til multi- og mange-kernede implementeringer, kan bliver implementeret med mere end én kerne i en Tinuso processor arkitektur. For at implementere denne multi-kerne arkitektur kræves forskellige teknologier, hvor NOC er en af de primære.

Tinuso kernen er allerede udviklet. Den kan direkte tilgå hukommelsen via et interface. For at tilgå hukommelsen med flere kerner i en processor skal der bruges et NOC. Til det skal bruges en Network Interface Controller til at kommunikere mellem kernen og NOC. Derudover skal en switch og router komponent samt en hukommelses kontroller bruges.

I dette projekt er der designet et NOC løsning til Tinuso processorens kerner, sådan at disse kan tilgå den primære hukommelse over NOC'en. Designet blev udført ud fra en analyse af nuværende teori og design koncepter baseret på en prioriteret kravspecifikation. Denne prioritering var udledt via agile analyse af de basale krav beskrevet øverst, ideer til udvidelser og uddybelse af de basale krav i mindre del komponenter og teknologier.

Det valgte design er et torus 2D mesh med YX trafikstyrings algoritme med en blanding af torus elementer.

Designet var implementeret i VHDL på en FPGA og testet mod et simuleret Tinuso kerne interface. Dette interface var designet i samarbejde med Tinuso kernens udvikler[1], som også udviklede sit eget testkørsels system. Som resultat heraf blev der fundet mindre forskelle mellem den faktiske kernes implementering af interfacet og det der var simuleret i dette speciales test opstilling. Forskellene er dokumenteret i resultat sektionen (Results).

Clock frekvens resultatet der blev dannet via syntesen indikerer at dette NOC system kan køre omkring halv frekvens af Tinuso kernen[1]. Dette var forventet, da denne implementering ikke er blevet optimeret til høj clock frekvens. Testen var den først runde af system tests af dette prototype system, der ikke engang kunne være på den FPGA syntesen var lavet efter.

Den implementerede løsning demonstrerer gennemførligheden af dette design og netværks protokollen der blev designet i testene. Der blev også demonstreret de mange udfordringer der i at designe og implementere løsninger der ikke går i baglås i parallelle, samtidige, systemer. Samtidig tilgang til hukommelsen fra flere kerner fejlede og gik i baglås i mange tilfælde, som konsekvens af en specifik ”deadlock” - ”baglås situation”.

De fleste af ”baglås situationerne” var faktisk forventede, da test opsætningen gik et godt stykke ud over de krav, der var sat. Dette blev gjort for at teste supportering af interessante forbedringer og udvidelser til den nuværende version af systemet, så som kerne til kerne kommunikation og ”cache coherency”.

Flere løsninger er blevet foreslået i dette speciale, for de få situationer hvor det vides, at det implementerede system kan gå i baglås via teori og tests. Løsningerne demonstrerer at der mange måder, med forskellige konsekvenser, at håndtere sådanne situationer.

Acknowledgments

First I would like to thank Pascal Schleuniger the original builder of the Tinuso core, the Tinuso project lead, my guide in this project, a team mate on Tinuso and for all his invaluable help, review and guidance throughout.

Secondly I'd like to thank Sven Karlsson that by his teachings in classes and guest lectures educated me about some of the topics I've come to love the most of computer science, in a enjoyable fashion. He also convinced me to work with hardware design and FPGA for my thesis despite this is a rare if not nontraditional direction for this degree. My fellow students and I all to often tend to focus on software aspects when it comes to parallel and concurrent topics. And this is despite our unique hardware background with multiple hardware design, architecture and FPGA courses. This project was a really interesting alternative way of “playing” with concurrent technology.

I'd also like to thank my parents for believing in me, nurturing my interests and giving me the support and tools all these years one way or another to get me to where I am today.
I started with parallel software development and computer fiddling long before starting at DTU thanks to your nurturing. I was always one of the best math and science students all the way to end of my high-school thanks to your help at times and nurturing where I had interests.

One of my fondest memory’s as a result of this is when I even manage to aced some of the introductory classes at DTU so much that I nearly got yelled at, in an oral examination:
It started with I got kicked out little more than a minute into a 15 minutes questioning which most of my fellows students was sweating over just passing and the guy before me had spent 25 minutes on. This came to everyone’s surprise including my own and my friends worried at first but I was pretty sure I had answered almost everything right. Later at the grading for my group the teacher came out and carefully explaining the others if they had passed or failed. Then he just pointing directly at me, looking a bit aggravated and started arguing I was wasting his time: “you! - you know how it went! Now get out of here!” and such banter. We where so surprised I had to ask him a second time just to be sure I had even passed!
To be fair the little time I had spent at this teachers lectures I had used to politely point out his errors and help other understand the content. Not that he wasn't nice - he later asked for me to TA this class and I would have gladly done it had I not already been working another job. Sadly as this was not a graded class, I never got a registered 12 anywhere to prove it. However everyone who was there that day still talk about this as one of the most funny and insane moments we had at DTU so far.

And yet despite all these advantages this degree has still been hard for me and many of my class mates: In the class I started with at DTU we have sadly seen quite a few dropped out simply for not having the right background in IT etc. despite good academics on paper.

This is why I strongly believe I would not be writing this thesis today had you not brought me all the way to the DTU bookstore to get books I did not even know existed anywhere else. I remember getting books in advanced C++, cryptography and concurrent programming long before amazon was a thing. This was almost 10 years before I started having classes in these topics and long before I really could understand half the content. Only in the last 3 years have I solved some of these riddles that have plagued me since I was a kid. I was ahead in classes because I had fun doing it when I knew what to use it for. That’s why I dared jump into this sort of thesis project.
Also a special thank you goes to you mom for proof reading this thesis – I had many horrible typos and errors at first that needed your corrections.

Finally last by not least I'd like to thank my beloved Kristina. For making so many beauty-full diagrams of my crude hand-drawn versions. For putting up with me during stressed periods of my study. And for so many other wonderful things you bring to my life – most days I don't know what I would do without your support.

Thank you.

Table of Contents

1 Introduction

The DTU IMM ESE section is developing the many-core FPGA based Tinuso processor for research into future architectures of multi- and many-core processors. The focus of this thesis is to develop the infrastructure needed for Tinuso cores to access main memory over a Network On Chip.

The reason a NOC is interesting is that the Tinuso core which is designed for multi- and many-core implementations can be implemented with more than one core in a Tinuso processor architecture. For implementing this multi-core architecture different technologies is needed - one of the main features being the NOC.

The Tinuso core is already developed. It can directly access memory via a interface. To access memory with multiple cores in one processor a NOC is needed. For that a Network Interface Controller to interface between core and NOC is needed. Also a routing & switch component and a memory controller with network interface are needed.

Several of the extensions and expansions can be developed on top of this basic setup. Of particular interest is core to core communications and deadlock prevention when multiple cores try to communicate on NOC at the same time. These and other steps towards supporting cache coherency protocols on top of the basic implementation will be introduced and analyzed.

Any steps towards cache coherency support are important for multi- and many-core systems as cache coherency is important for the overall speed of the processor when multiple cores are employed.

Other research related extensions that is also interesting will be studied in this thesis is: torus NOC configuration and routing design, good scalability features of NOC's and implementation of variable length of the main lanes in NOC systems.

1.1 Tinuso

The pipeline or processor core of the Tinuso architecture has already been developed. This core and architecture is designed and implemented specifically for FPGA implementation to research in multi- and many-core processor [1]. Its instruction set makes use of predicated instructions and supports C/C++ and assembly language programming. The Tinuso core is deeply pipelines and achieves clock frequencies close to 400 MHz in FPGA's comparable to the FPGA that will be used as the basis of this thesis.

The Tinuso architecture are designed to be easy extendable to be flexible for the research its used for. Tinuso already contains a co-processor interface to connect to a network interface which will be used in this thesis.

The name Tinuso comes from the Esperanto word for “tuna”. Since this species live in swarms there is a level of analogy to multi-core systems. Also Clupea, the Latin word for “herring”, is developed at DTU IMM which is an intelligent network interface. In the food chain the tuna is one level above the herring which is in good correlation with system level architecture.

1.2 Objectives

The requirements of Network On Chip and main memory interfacing can be broken into the following sub requirements and systems:

First we needed to design and implement a Network On Chip. For this we need to design a interface between the current Tinuso core and the Network Interface Controller which is the interface towards the NOC. The NIC also needs to be designed and implemented to interface with the NOC. Then a routing and switching component in the NOC is required.

To serve main memory request on the NOC a memory interface is required towards the memory. On-FPGA memory (SRAM) and controller interface will do for testing and smaller programs etc. This memory interface is meant to be combined with the NIC so this is reused for all interfaces towards the NOC.

To make sure the NOC interfaces and routing interacts correctly a network protocol has to be designed as the guideline for behavior and communication in the NOC.

Further all these parts should be as configurable as possible for an example in terms of lane width of the NOC system so differences can be studied – this is for a research processor after all.

2 Theoretical background

To understand the decisions made in this thesis some background theory is needed into the topics of general network theory and Networks On Chip specifically.

2.1 Networks in communication

The Internet is just one of many, many systems that interconnect devices for communications across any distance from nanometers to galaxies. Typically such systems are designed in a way that allows several devices to use the same communication channel for communicating with other devices on the channel. This is possible via a set of rules defined in what's called a network protocol to communicate. A communication channel would in case of human speaking be the air. It is the medium the signals, in this example sound, is traveling in. These systems of communicating devices are called a network as they often in design or effective real world wiring and connections looks like a spider networks. Hence, the term coined for an Internet protocol which is called "the World Wide Web" and is why the prefix "www." is used when requesting websites.

2.1.1 Network Communication basics

There are two main ways to look at and design networks: Package switching and circuit switching.

Package switching is the same way humans have designed the mail and package delivery systems. Your items needs to have a sender and receiver address and a distribution system then takes your items, sorts them and tries to deliver them to the receiver or another distribution central closer to the receiver. This is how the Internet and many other networks work.

In communication network terms your items are called packages with the distribution central called a router or switch as it routes your package via its header information of receiver and sender information and possible further data. The switch part is analogous to a train network where a big station might have many trains that need to go back and forth between very few lines of track so the station managers need to switch back and forth on the holding or loading tracks that in communication network are called buffers. Notice it's the holding & station tracks that are analogous to buffers in networks - the train stopping security features called buffers is something completely different. Buffers are used in many kind of routers and switches in many sorts of network types to temporally contain some information before transmitting it further.

In a network a human would be analogous to a node or a client E.I. a member of the network that can send and receive. In some cases nodes also route or switch the data for others just like humans can. For an example just think about much routing a receptionist or coordinator in a company does on a daily basis by selectively passing information to others.

To design circuit switching in communication network is analogous to directing flow of water or old school long distance phone calls. Say I want to talk to a person in a city in a nearby. What I then do is I send a signal to all the city's or other connection points in between and ask for them to open a most often physical line between me and the destination. They then open to a connection point they think is closer to the destination than where my line ends now. When the line is open I now have a line where I can send and receive my data in this case my voice.

Time Division Multiplexing is a circuit switching example where each device communicates with each other in specific allocated time slots. This is how a small number of today's digital phone-line connections can handle all the cross-Atlantic conversations. They use TDM to fit many conversations sliced up in small bites inside a single digital cable.

2.1.2 Network Protocols

Network protocols are needed just like the human language which are arguably a special case of network protocols on its own. They are required so that senders and receivers of messages have the same rules and guidelines to interpret and form messages.

Many of the behavioral norms and ideas of the Internet and other networks can be traced back to when humans used light, flag, smoke sources and other visual signals to communicate over distances. In fact Morse code is perhaps the best known network protocol. This is a good example as this protocol is not just the Morse code alphabet as most would believe but includes human behavior and rules. Such rules as who are allowed to use a ships Morse code sending device, emergency procedures, waiting for others to respond that they acknowledge and understand the message you have sent. Or to resent your message if the receiver couldn't recover enough of the message for it to be understandable with their human wit and so forth.

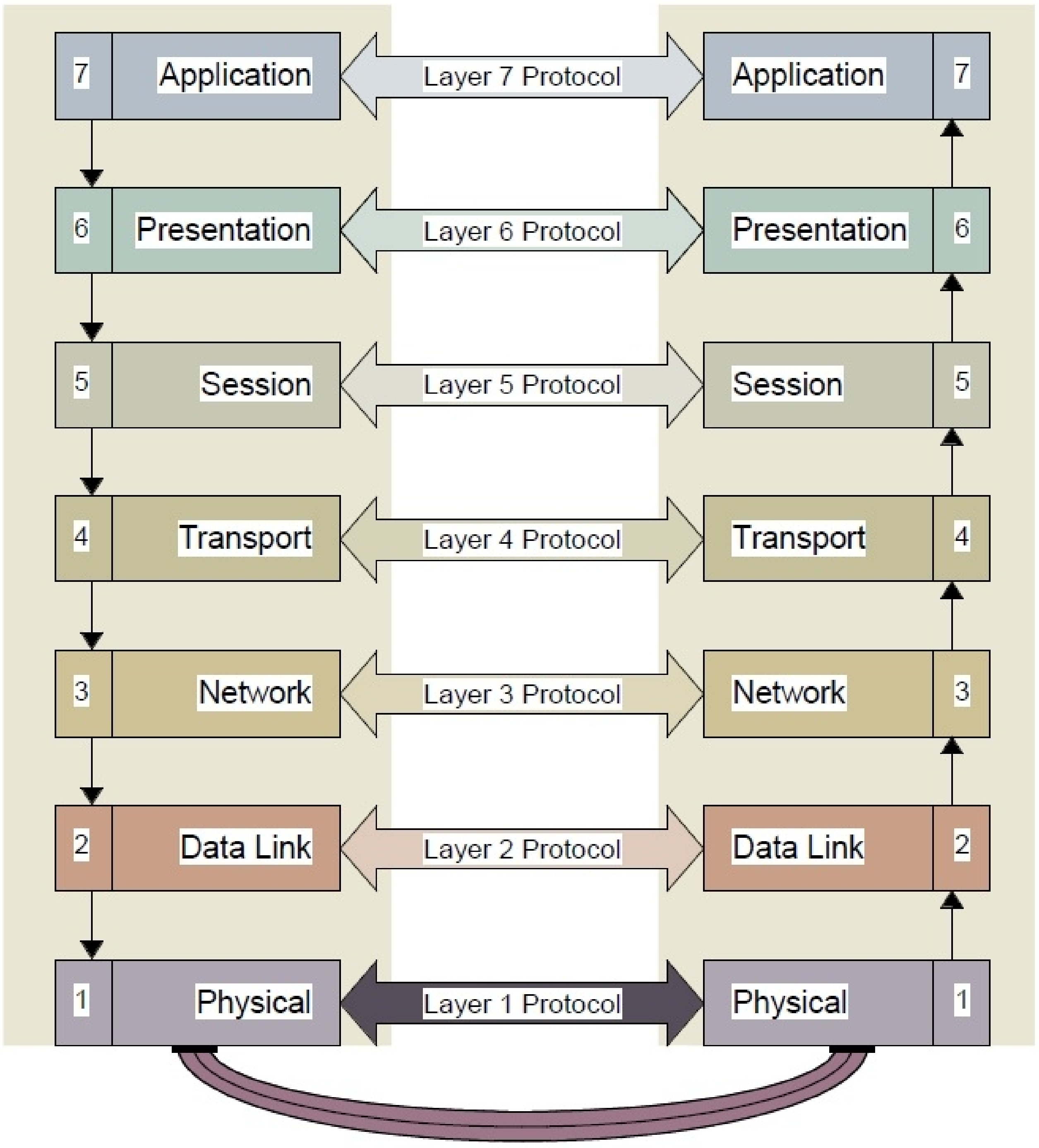
Network protocols usually have many if not all of the above described features and more depending on number of parameters. These deciding parameters could be the practicality in the intended networks maximum distance, speed and number of devices connected. For an example some networks like the Internet Protocol (IP) and most of its associates we now all use in our life on a daily basis for most of our communications has all some level of error correction. They have this via algorithm design in the protocol due to the messages have to travel relative long or hazardous distances where errors are likely. While others like the Tinuso system NOC protocol we have designed has no use for such features as their complexity and speed loss is vastly to great compared to the frequency of error rate internally on a chip in our current systems.

However this could again change in a not too distance future so protocols are often updated if practical or other “service” protocols are layered on top. For an example if some of the data transported in that network critically have to be error free or encrypted – think bank transfers or real-time emergency systems like in power plants or your car.

2.1.3 ISO Open Systems Interconnection

The International Standards Organization (ISO) has defined a reference model for protocols and networks called Open Systems Interconnection (OSI). The core concept of this model and all you really need to know if you have read the rest of this theory section is that OSI defines a 7 layer model for designing and implementing networks where each layer does not know or need to know of the one below or above if everyone adheres to the model.

The below is a 7 layer model – notice how the layers have color similarities if they are “merged” in the fewer layer models:



There is the lower hardware layers like package routing that makes actual physical and the basic sending of data from one end to the other possible in shorter distances the lower you come. The middle layers that can offer various services on top of this or in some cases total ignored for going straight to the top layers where the individual devices specific needs are met. In most cases via software programs that most often can use many or none of the middle layers as a service to reduce their work load and development time.

Top layer could be your internet banking's security layer and on top of it their functional system you interact with. The middle layers in this case would be internet protocols that help with basic security and correct transmission etc. and in the lowest layers we have the first have the Internet Protocol (IP), unless you count that as a middle layer. IP makes it possible for transmission to go all the way to the destination even though sender and receiver is on different internal networks. An example would be like the local one a company can have internally and the one is between space agencies and the International Space Station which in fact is one of the few places connected to the normal Internet where the lower protocols are actually different than de facto protocols due to the delay to and from ISS.

The lowest protocols just makes sure your network interface like your wireless chip can work with your wireless connection point at physical parameters such as frequencies, connection plugs, connection protocol to and from locally between the 2 or more devices on this communication line – in case of wireless often some form of the earlier mention Time Division Multiplexing is use to let more devices talk to the same wireless connection point or router at the same time at the cost of splitting the speed up in the number of clients connected.

Some think a 5 layer model of the same type is better but that's not a ISO standard and both models can have their usefulness depending on which type of network your building a device for. For reference the above theory used between 3 and 5 layers depending on how you count.

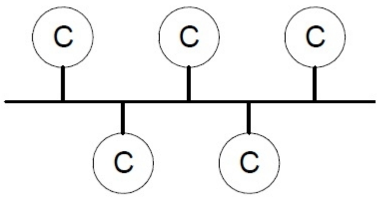
What really is essence here is that the concept here is to reuse as much as possible if you can. So every time you want to develop say a chat client for a smartphone you don't have to reinvent the entire Internet and the mobile communications network but can reuse the majority of components and it will still be comparable with what everyone else uses on the network.

As a developer you might even be so lucky that you get to choose whether you want to use top layer protocols that give you no fuss speed but gives you no guaranties at all or stability protocols both with guaranties of delivery of information and correct order of deliveries but as tradeoff for less speed. Or even security protocols that works on top and with existing networks but gives you perfect security in terms of encryption and prof of identity for sender and receiver like use in proper internet shopping payment systems. The no fuss speed option is actually the most basic option that all others work on top of or combine with as all it really does is trying to get data back and forth with sender and receiver addresses provided and using the lower hardware near protocols like package switching to try to actually move the data. In its bare for its often used in live media transmissions or other time critical transmissions where retransmissions and other time delaying features such as demanding error correction means data is coming in way to late to be practical except for special circumstances the end user program or device usually want to decide on its own and always can by building a layer on top.

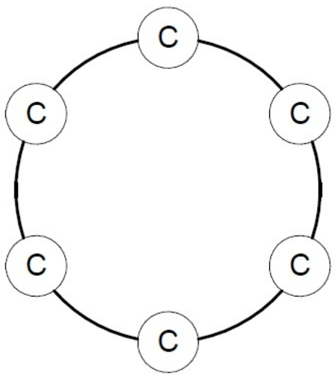
This reuse yet highly customizable end product is the beauty of the concept ISO has standardized in the OSI reference model.

2.1.4 Topologies

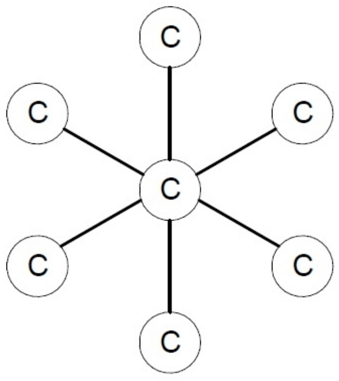
One important topic of generic networks is how the physical hardware devices are connected as this have huge ramifications for the network protocol design and difference performance parameters. Here are the major types:



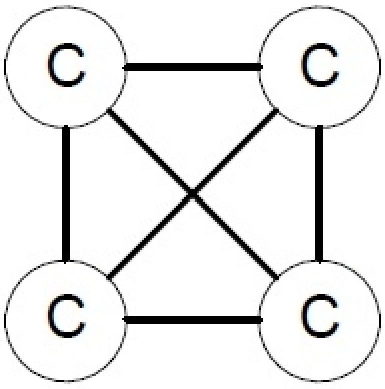
Bus: This is basically the way some wireless networks, some special application wired and older networks work. Everyone shares one channel and has to take turns communicating on it. This clearly have scalability issues if everyone on it wants to talk relative often and for extended time. Also if the channel malfunctions there is no backup. On the other hand due to how such networks is design if nodes malfunctions there are a good chance that only that devices malfunctioning and devices that communicates with these will suffer and the rest of the system can continue. It is also relatively simple, predictable and cheap implement compared to other topologies which makes it suitable for small systems.



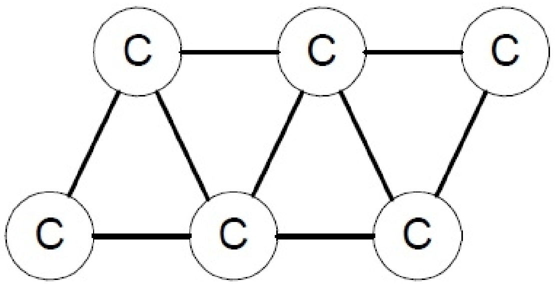
Ring: Everyone is connected in a ring and if you want to connect to someone that's not your neighbor you have to ask the neighbor to send it to him or to the next neighbor over and so forth. This is one step up from buses but with some advantages and limitations. Everyone has potential more speed as there is 2 ways around the network and if nobody else talks to your neighbors you have highest possible speed. If a link to one neighbor malfunctions it's possible to design it so that messages can go the long way around and still reach them. If both a devices neighbors malfunctions however total connection is lost would be lost. That is why most ring topologies are designed so that in case of malfunction the signal path between the neighbors of the malfunctioned is still open.



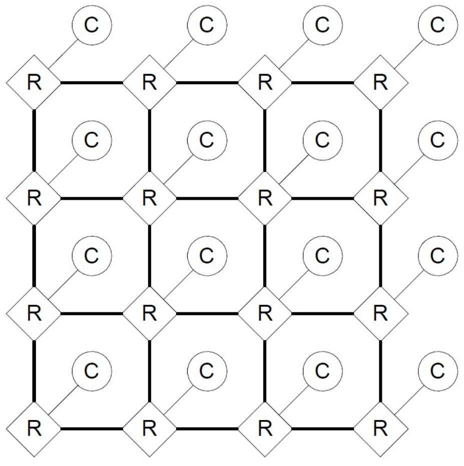
Star: Here everyone has to talk to the star in the center and that stars routing or switching system will decide where your messages will go. Everyone has full speed to the star device but is depending on the star to have enough internal speed to route fast enough to other devices than the star or for the star to respond fast enough. If two The Star device is a single point of failure like your wireless or wired network at home so if it goes down there is no local or Internet for you. However mobile networks, larger local wireless networks and such star networks work with more than one star within range a large part of the time depending on your location and network. This means the device can switch over in case of failure or overcrowded network but usually with some loss in signal speed or stability for wireless networks due to physical rules. On wired networks it's like having 2 internet providers or 2 routers for the same local network – if one fails there is a short downtime while you manually or automatically switch to the other if you are not connect to both at once which technically is a border line mesh case.



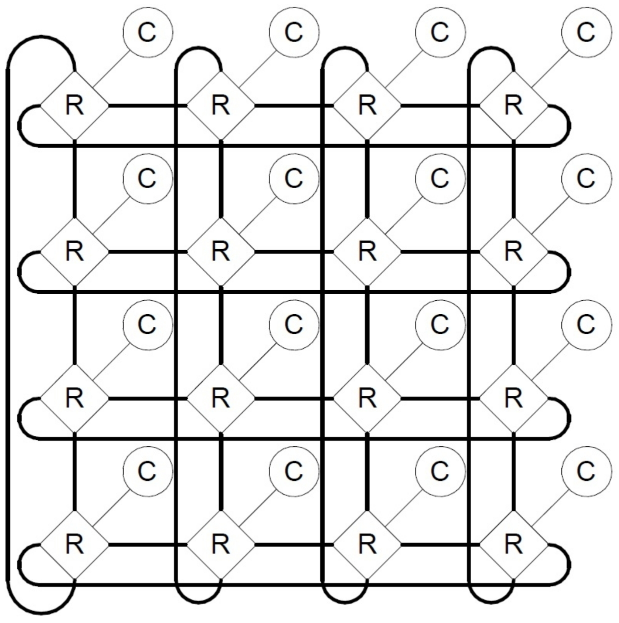
Mesh: This topology comes in two versions: Full mesh and partial mesh. In full mesh every node is connected to every other one directly with a channel. This means everyone has the possibility of full speed to everyone else if everyone can receiver and send to everyone else independently of how many they are communicating to at the same time. This requires very fast nodes or limits the speed advantage. Also the number of channels makes this both more expensive to setup and maintain compared to any other solution. It's also possibly unpractical for high speed connections with many nodes or node numbers such as the Internet where full mesh status would never be achievable with current technology as just the probability for a few channel to fail even before the full mesh is connect for the first time is practically a certainty. The stability for practical node size can however be very good as there is minimal points of failure E.I. a node has to fail on its own or loose connection to all its lines to loose connection to anyone or the receiver have to experience the same. Stability can also be very bad if a protocol doesn't take into account that everyone can retransmit to everyone else and overcrowd the network with.



Partial mesh version are where every node is connected to more than one other so that Ring topology is a special lowest version of partial mesh with connection to 3 other nodes and their being at least 4 nodes in the network the first normal partial mesh type in terms of size. It's basically all the advantages of full mesh topology with none or very limited version of the disadvantages as long as the partial mesh network includes more than 5 nodes and at least 4 connections or so depending on number of nodes. Dangers of overcrowding via retransmitted data in endless loops is still existent however. All normal partial mesh requires relative complex and there by expensive node and router solutions 3 sub types are interesting in relation to this thesis:



2D partial mesh – This is a type where a node is only connected to itself and its neighbors as seen in a 2D grid array. Typically only the adjacent nodes left and right and up and down are connected but connections where all 8 adjacent nodes in 2D grid is also an option. This means that every node have to ask its neighbor like in ring topologies if it was to communicate with logically distant nodes. This is worse than most other partial mesh types in terms of speed and delay but make for very simple high-speed switches and routers due to the limit number of routing options in a known grid. Edge nodes however only have 2 or 3 neighbor connections.



Torus partial mesh – This is a special case of 2D partial mesh where edge nodes have been connected so that left edge goes to right edge and top to bottom and vice versa. Meaning all nodes have 4 neighbors with the potential speed boosts this added routing can bring. This also means additional complexities to routing protocols which is a current research topic. In the Network On Chip section this topic and routing concepts related is introduced.

Dynamic partial mesh mash-up – This is not an official or de facto name for a topology if it can even be called that and is entirely the authors own interpretation of what exists in the wild. This is in the authors mind the best way to describe a subcategory where connections are changing often or rapidly and where many current networks can be placed as they exhibit the clear markings of a mash-up between several network topologies and routing concepts.

The core of this type is typical partial mesh with some nodes connected to many others in what is comparable to local star configurations in several layers. Many of the of the stars nodes like main core routing networks of the internet are also interconnect with each other directly for speed or stability reasons. It's a mash-up of network topologies and routing concepts.

It is how the Internet is setup at the world wide core infrastructure between internet providers and depending on your local or regional network this may also be the case there.

This mash-up is done mainly on key locations around the world like in Amsterdam, London and to a lesser extends the older original regional or national locations like the Danish Internet eXtange point at DTU when talking about the internet. These older regional locations is connected to a number of other locations over long distances across borders and most connections out from the region would go through these using them as both outbound star connection point and regional partial mesh interconnect center. Connections change all the time due to new faster/better technologies, repairs or expansion needs that are very rapidly changing.

This is one of the main reasons time and speed from your computer anywhere in the world to any other can be very unpredictable or the connection even unstable. Even for connections to the neighbor town if they are not directly connected.

On the other side the Internet Protocols makes it possible to at least have connections from practically most industrialized places in the world. With proper network infrastructure many European countries can even enjoy internet based real-time games or video links from other well establish internet nations on other side of the Earth such as South-Korea and East-coast USA with only between 1 and 4 hundred milliseconds delay. Comparably current mobile wireless technologies such as 3G mobile networks can have the same delay between 2 mobiles in the same room while the new LTE protocols has delays more comparable to home WiFi besides the public marketed speed boosts.

All these technologies can be seen as part of the internet or their own Dynamic partial mesh mash-up in their own right. In LTE, 3G and other mobile protocols there is a star configuration to the local mobile reception mast while the masts now are all running on some level of backbone network being interconnected at some level depending on what’s physically practical. In many homes WiFi networks are also connected with wired networks for internet access and stationary objects such TV's, gaming consoles etc.

In conclusion on this sub type: it can be very stable for over all routing with few or no single points of failures depending on how you use these networks which is constantly evolving. This was a primary goal the US military research division intended when they developed what is now the internet. This type can be very ineffective and slow at relative short distances due to the lack of any overall plan for local area interconnections which means there is no guaranty of geographical relations to performance in short distances.

2.2 Network On Chip

Networks on a chip comes from the ideas of using classical computer networking for on chip communications between a larger number of components where development time and communication performance is not well supported with direct links or buses. Originally naive implementations with large router components or the like was used which had good scalability in development time but was slow and used too much chip space. Currently designs in very fine tuned versions are emerging with tune in on the different tradeoffs faced. This is emerging partially with the need for interconnecting more and more components to satisfy performance goals. This has made NOC a very promising technology in everything from internals of heavy optical routers to the internals of a smartphone. Most embedded devices like smartphones are merging towards single chip designs that literally are the "System on a Chip" and for that the components which are often made by different manufactures and are insanely complex needs to communicate.

The main goal of NOC systems is to be fairly easily scalable with a range of resources such as general processor cores or special signal processors attached. The scalability is twofold;

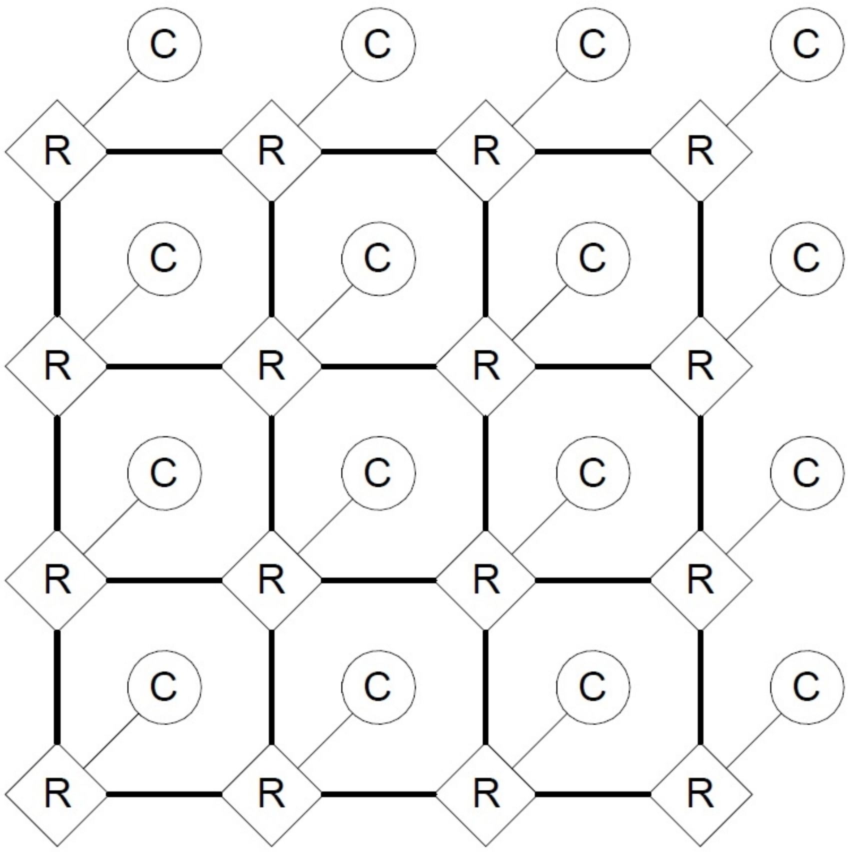
1.Scalability in design and implementation face where modularity and network infrastructure has to function so that it requires at worst close to linear cost with increasing number of resources connected and attaching new types of resources.

2.Scalability in terms of the network performance. It has to be both practical in terms of chip space usage, power and scale in speed at worst near linearly which means that full mesh and bus topologies are impractically expensive or to slow as the number of connected resources rise.

2.2.1 Network topologies

The resulting choices for network layout based on the scalability requirements is typically a mix of partial mesh, star topology and such network topologies custom made for the specific needs of communication between many resource inside one chip. Any NOC designs are application specific to enhance performance parameters such as energy or thought put for a particular combination of resources while some NOC designs are trying to be general in nature for compatibility with many resources. Below is two very popular choices explained that can fulfill the requirements for NOC:

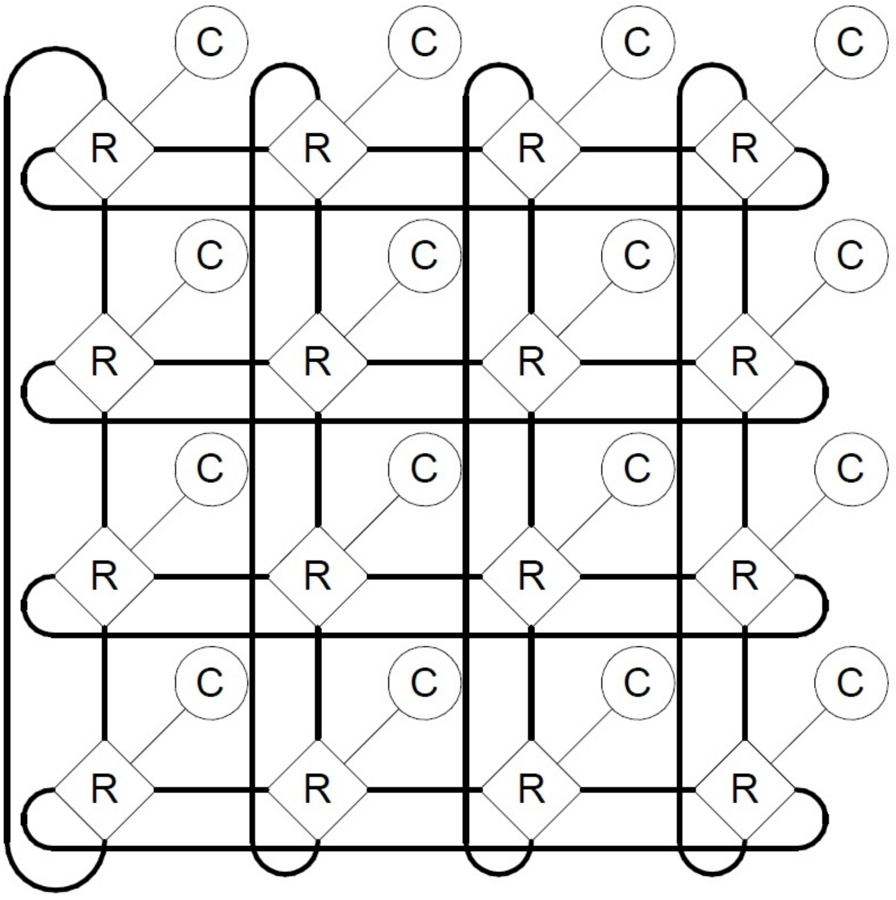
2D grid



4x4 2D grid – C is cores; R is the routing & switching components.

This graphs shows better than words the perhaps most well-known layout used successfully in NOC systems. The 2D grid is often but not always based on a partial mesh topology where each node is connected to its direct neighbors and a local resource. This means that in order for a package or signal to propagate to a faraway resource it has to travel many clock cycles. On the other hand the length of the connection wires is short so clock frequency can be high. It also means as many as all resources potentially can communicate simultaneously in best case. Often due to the relative slow rate of communicating over any network resources try to do as much internally as possible giving the same effect as in any other network where only a portion of the resources need to communicate to another resource at any time. This means the use of connections, bandwidth and power is well balanced and is better than many alternatives as the bandwidth scales well when adding additional resources. It's also relatively easy and predictable to develop fast routing algorithms for which is an important speed factor in NOC systems.

Torus



4x4 2D grid with torus edge connection scheme.

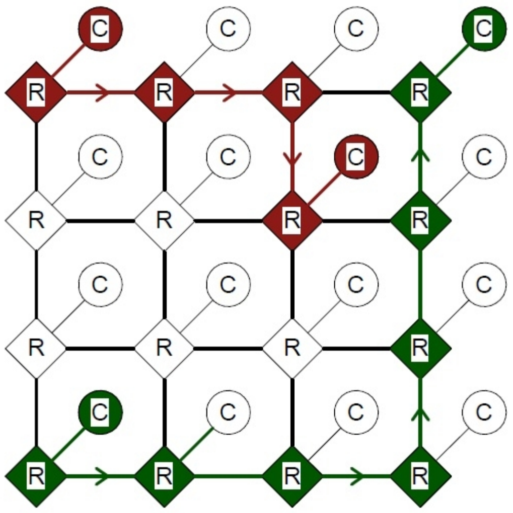
C is cores; R is the routing & switching components.

A torus design is 2D grid with added features. The edge nodes of this grid are connected to its opposite edge so that if all nodes where distributed with equal distance in a 3D map it would look like a torus shape. This makes routes between 2 nodes potentially a lot faster in some situations and adds more routes leaving less chance of intersection between communications lines. The advantages of a torus design largely depend on the resulting long wire length across the NOC from edge to edge in current 2D FPGA designs is limiting to the clock frequency of the NOC. Also a major factor is the routing algorithms used as there is many ways to support torus routing where the tradeoff is between complexity of routing algorithm and efficient use of torus advantages.

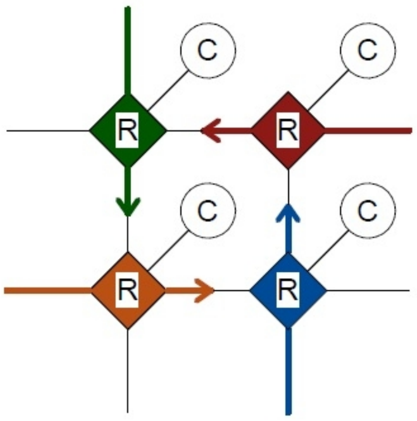
2.2.2 Routing algorithms

There are many well documented routing algorithms for NOC networks. In this section some of the most relevant and efficient routing algorithms for 2D grid and torus topologies are described.

XY

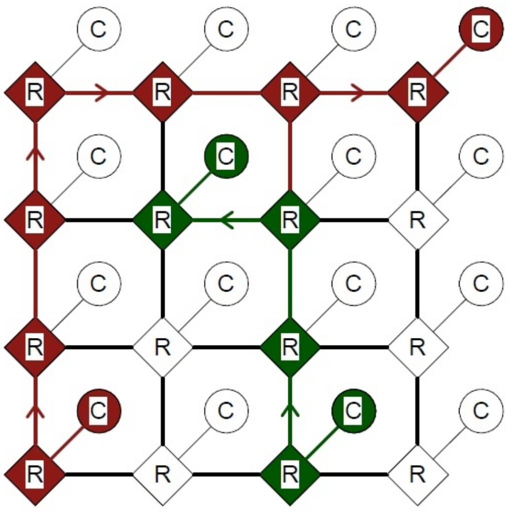


This routing algorithm is very popular for its efficiency in 2D grids. In test's it beats the current competitors for medium to large NOC designs by being deterministic [3], having no gridlocks and simple implementation leading to fast clock frequencies. The design is as simple as first moving in the horizontal direction or x-axis in a 2D coordinate system and the vertically or on the y-axis. This makes gridlocks like the below example impossible, when a signal being blocked by another wait for it to remove itself. This is due to the limit degrees of freedom in how many times the route is allowed to turn – i.e. there is no freedom; only movement in horizontal direction and then vertical is allowed and only once each. Below is the electronic version of a New York 1970'is gridlock:

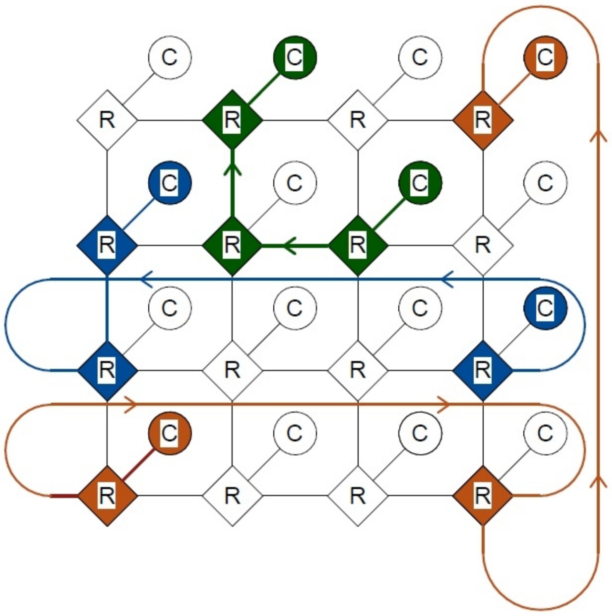


YX

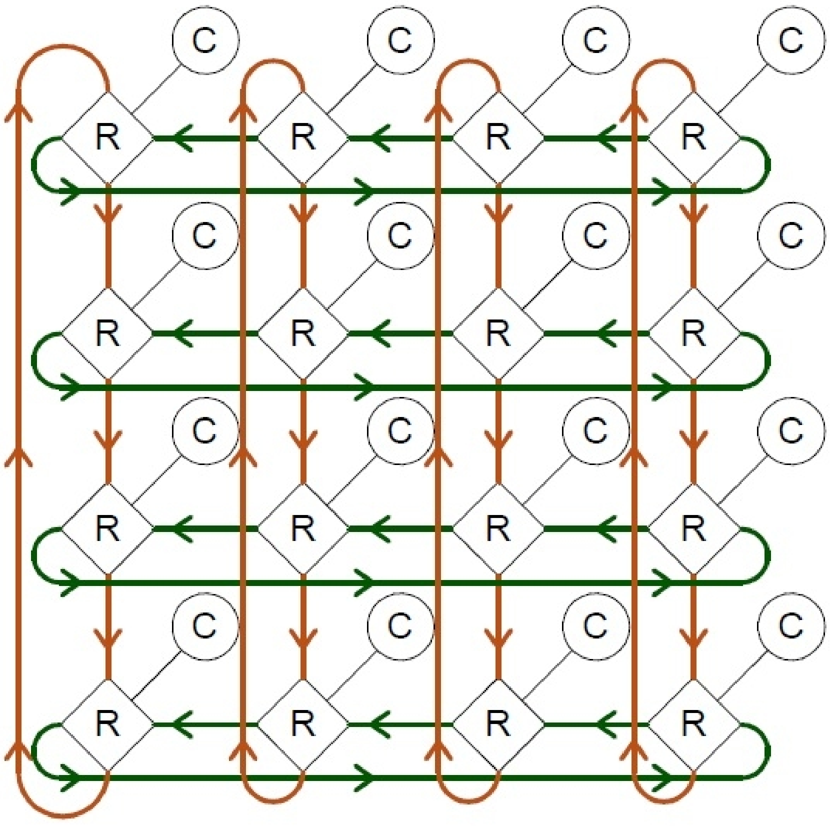
The same as XY routing but where the routing is first done in the vertical direction and then in the horizontal direction. YX is often just included as minor note in XY descriptions as there is no difference in speed, implementation time etc. between them in most systems.



Torus designs

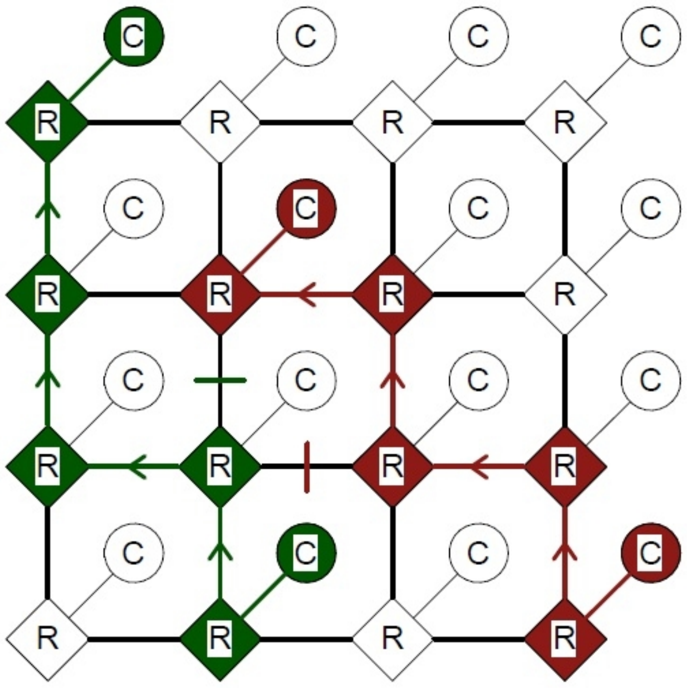


In torus designs the algorithms have to be somewhat different from normal 2D grid versions to effectively use the additional routing possibilities. Deadlocks are harder to avoid in a torus design but there are solutions. Generally they revolve around virtual channels where horizontal and vertical lines are not directly connected on a logical level. Instead package switching inside the router transport from one "virtual channel" to the other when a package has reached its destination in the axis it is traversing as illustrated below:



Partially adaptive

Partially adaptive algorithms is trading efficient use of connections for additional routing implementation complexity and there by possibly raw thought put loss due to big routing logic, blocking so as not to run into gridlocks etc. The algorithms do this by trying to steer signal connections or packages around blockages where other connections are using the fastest or first path selected to the destination with the algorithm used. The main difference from XY routing is that they can travel in x then y then x again or y the x the y again and such path with further degree of freedom. But they are not allowed full freedom as not to make the logic to complex when avoiding deadlocks hence the name partially adaptive. An example:



3 Analysis

This section will detail the planning, analyzing questions and results used to scope the tasks of this project.

3.1 Agile development

Agile analysis is an ongoing process throughout the project span. To understand it here is some general information about agile development.

Agile in this context is a way of doing, mainly, software development that is a kind of grand overall strategy that is subdivided into smaller components depending on what you look at. It is almost all inclusive with many features not usually incorporated into development thinking even though these are important to the end user and developer of the project. It's important here to state that many critics of agile argue that this is mostly not the fact as many of these features are now used by most developers that do not state they use agile development techniques. They argue that some of these changes were simply a result of smaller modernization or additions to older flawed techniques that became broadly accepted by most as flaws was found. On the other hand many agile and related developers argue that in fact it's the other developers that are slowly moving towards agile development by using most of its components while still using fundamentally flawed or not having a development strategy.

Generally agile has been accepted when used in relevant cases as leading to more efficient developing with better end user value and/or lower cost of development as feature creep and other often stated problems of untimely, bad or expensive products is battled. Feature creep is a well-known phenomenon. It describes the phenomenon for adding more feature than needed for the core product to work efficiently for waste majority of end users. It happens most often as the developers or end users think in some cases would be smart to have the feature. This usually gets worse as development progresses and end user requirements and values change or expand.

Agile been popularized with internet and mobile software applications as it works great for fast changing end user requirements, fast development cycles that can be used to put new features into a live running product often. Also research projects with many unknowns is good target as priorities here change can change very often as dead ends of research and too time consuming paths are found. In fact risk migration by uncovering problems with central features first is a key component of agile.

That is why based on previous experience the author decided with the Tinuso team (Sven Karlsson and Pascal Schleuniger) to use this as our development strategy on this project.

3.2 Agile analysis

Basically the concept of agile analysis is to in the team to update a list of the core features needed to finish the core product to a fully working state. This should be done as often as weekly is based on current status of development at the meeting. These features are then ranked on properties such as:

•The end user value for that feature.

•How critical the feature is for the product to function.

•Have the feature any unproven technology etc.

This range generally from "need to have" features that the product would fail to meet the most core functionality without too "nice to have" features that could be superficial changes, features that have very small effects or effect very few end user little. In between is “want to have” features that are perfect targets for additions to current release date of project if there is time when “need to have” features are done or for next versions. These are typically the targets for the rapid updates after launch that is part of modern internet and mobile application development. Bugs that leave non-core functionality not working as in tended would be a “want to have” fix while any bugs to the core functionality would almost exclusively be “need to have”.

From this table it’s easy to coordinate who works on what that short period inside a team if needed or just plan your own time effectively to be as productive as possible. Then more or less daily a number of sub tasks can be selected based on their critical importance to the features major features.

Generally a project has failed developing the product if it’s stopped early. This means before all features of the need to have list is not done making the product practically unusable for its intended use. On the other hand many projects are finishing the development early if all the “need to have” features is complete. This is because its often not worth focusing the resources on the project any more if the developer can work on other projects that is “need to have” within the organization according to case studies[2]. What is the central point of this is that it easy for everyone to see when it’s worth continuing development on a product or sub feature. There is always a clear picture at any time of what state the product is in - even after “need to have” features are implemented.

3.3 Scoping the task

As agile scoping as already explain in the previous sections is a on-going process thought out the length of a project this section is a mix initial thoughts and ideas with the end result schematic which can be seen below. The requirements for completing this project are divided into the 3 primary groups: “Need to have” features that has to be designed, implemented and tested. “Want to have” features which cover what the Tinuso team wished for this thesis to get as much of as possible. Then there is “nice to have” features which might be actually be implemented before “want to have” features as most of our want to have features have rather large implications on the total system while many nice to have features fortunately neatly can be developed largely as independent components with much less development time needed.

Need to have	Want to have	Nice to have
Network protocol	Communication between cores	Main memory interface
Switch/router	Cache coherency	
Network Interface Controller	Torus routing algorithm	
NIC interface towards Tinuso core	Multi-core access to the main memory	
Network interface for memory		
Test memory & interface		
Test-bench with simulated Tinuso core interface towards NIC		
Scalable Network On Chip		

The features seen here in the final table version is based on the requirements of cache coherency introduced in the introduction section. The mainly focus on the Network On Chip architecture which is the core interlinking feature between all these components and sub-components.

4 Design

There are several major design decisions critical to this project and in this chapter. This section covers the main sections of network protocol design, Tinuso core interface design and NOC design decisions. Test designs and related is appropriately in its own sections. The following are the final revision of these design decisions and where relevant changes from initial drafts are noted.

4.1 Network protocol design

One of the main features of a network protocol is the package design I.e. how the parts coordinate and understand what's being said over the communication line. In this system the header and data parts has to be able to be divided into smaller chunks depending on the lane width implemented. Minimum lane width supported was set at 16 bits but actually smaller could be supported like 8 bits width with this design but it would be harder to make an easy switch in implementation between lower than 16 bits without major changes.

Due to this system is to be implemented on a FPGA there is limits to how many points or connected resources that can be connected. In the current design 8 bits for addressing nodes giving near 256 unique address possibilities is then enough even if a few addresses should be reserved. To route the package there is a need for including both the sender and receiver information – the sender information being especially invaluable if return responses are needed once the package reaches the destination. The entire package structure is defined below:

Package design

Size in bits	Type:
8	Receivers address
8	Sender address
16	Type + special data (ack bit etc.)
0-288	Data

Type data:
To support cache coherency types related types and general types such a core to memory read or write operations effectively, 16 bits is set aside as type + special data.
There is no types included specifically for cache coherency such as acknowledgments, responses, write backs etc. but the structure supports extension with these by having enough types and special data space to support huge number of package types which doesn't need extra data appended for up to 8 bits of type related data.

Bit 3 down to 0 defines the length of data included in a package coded in the special steps. No data meaning all information is contained in the type and special data such as acknowledge (ACK) or not acknowledged (NACK) uses. Or 288 bits of additional data for sending more than entire cache lines in one package. A cache line being 8 X 32bits in this design as standard but this is configurable:

- "0000" = 0 bit
- "0001" = 16bit
- "0010" = 32bit
- "0011" = 64bit
- "0100" = 128bit

- •"0101" = 256bit
- •"0110" = 288bit

Bit 8 down to 4 defines the type of the package with these already defined types and plenty of space for additions:

- •0001 - read memory line from location defined in package data and return memory line
- •0010 – write package data to memory line location defined in first 32 bit of package data.
- •0011 – read-return type for the memory line returned to sender in type "0001"

Routing design

The address bits is defined as locations in a coordinate system with 2 axes so it's split in 2 x 4 bit numbers. 0000 0001 being lower left corner and 1111 1110 being the top right corner. 0000 0000 and is reserved for signaling and 10100000 as explained later is also reserved. The first 4 bits for location of the switch/resource pairs in the vertical direction and last 4 bit is location vertical so 0001 0100 would be location (1,3) in a normal coordinate system starting in (0,0), vertical bits starting at 0001 to work around 0000 0000 being reserved.

Further routing design decision is explained under the NOC design section as they are based on NOC design decisions.

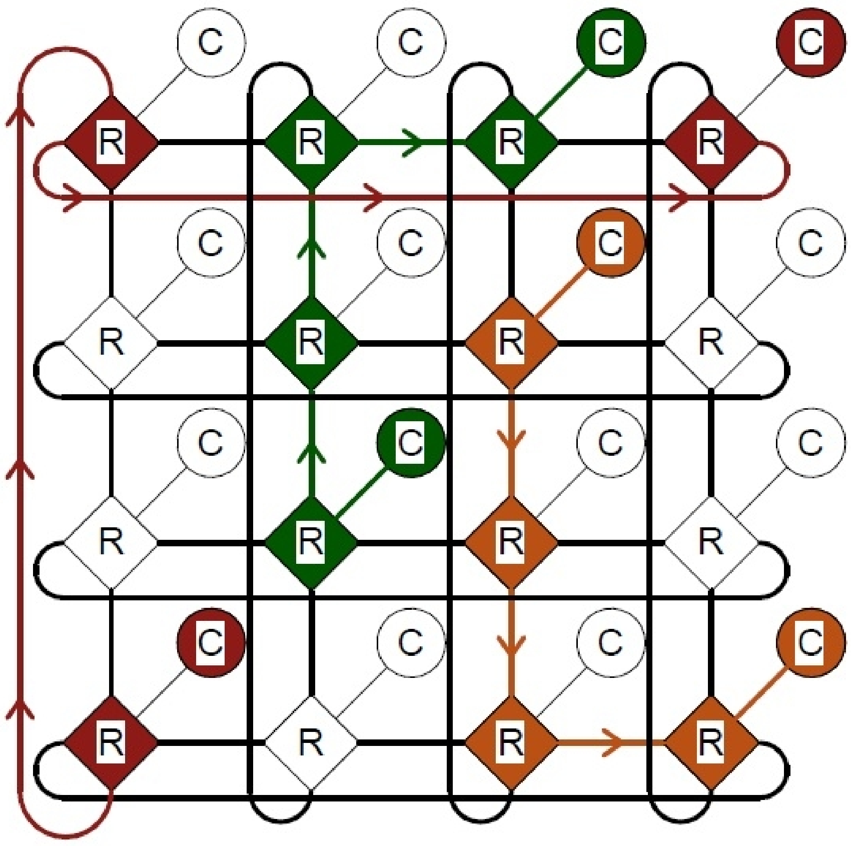
Line Ready Signal

The receiver of a package needs to send a signal when successfully stabilizing a communication line with the sender to make sure data are not lost down the line. This can happen in case of having to wait for other lines blocking the path or if the receiver is not ready. At least in this basic implementation this is an issue. For this the following line is send from receiver to sender: "0000001010100000". That word then need to be reserved as signaling lengths in NOC system easily comes to the length that a receiver getting this signal could have gone back to idle state after responding to this before sender got the command that this signal and would now think it was the start of an incoming package header. However since the length of this word is the same as both sender and receiver addresses put in a package header only one of the half's need to be reserved and the line 10100000 is then as a consequence reserved.

4.2 Network On Chip design decisions

The NOC system was chosen to be a 2D grid as a partial mesh. It uses YX routing design with a few novel design features. One of these novelties is an experimental torus design where edge nodes of this grid are connected so that if all nodes where distributed with equal distance in a 3D map it would look like a torus shape. This requires some additions to the routing protocol compared to normal 2D grid routing. The Torus design is not a novel NOC system in itself. Only the routing decisions related that used in the routing components are novel - at least to the authors knowledge the exact design is not mentioned anywhere else.

To simplify the implementation the YX routing is used with exception of routing at edge nodes where all routing choices are considered. Specifically if it can save connection length torus edge connections are used even if the end target node is not an edge node. But this is only considered if a package reaches a edge node so with larger NOC designs the effects on the central part of the grid is very little while the edge areas can have much shorter connections. In fact the corner area nodes keeps having a the same very short connection length to each other regardless of the NOC dimensions. See diagram below:



On a practical level it is decided that each router component should have 4 connections to its neighbors and one line to the NIC. The lines should be going to the neighbors up, down, left and right of the router. The standard lane width of these lines should be 16 for 16 bit per clock cycle as a good compromise between space, speed and power use and the width should be scalable, as easily as possible, to 32, 64 etc. when implementing.

The NIC should be implemented so that it is independent of the interface to the connected resource by only communicating with the resource indirectly via buffers and a glue logic process in VHDL. This is needed anyway to guaranty only successful transmitted packages from the NOC is transformed and send to the resource. Also to guaranty the resource that any packages it wants to send over the NIC is actually sent. Another important reason to do this is so that the NIC can independently send or receive a package from the NOC while the resource is also communicating with the NIC. The Tinuso interface and many other resources, especially if they support multiple send operations before receiving packages expects to always have access to the communicating with the NIC. In fact as is detailed in the next section it’s not even possible for the NIC to tell that its busy.

4.3 Tinuso core interface

The team working on the Tinuso hardware platform includes the original Tinuso pipeline and core developer Pascal Schleuniger. He is also the team lead on the Tinuso architecture, a PhD student and the guide for this project. Together with Pascal a interface between the core's cache controller and the planned Network Interface Controller was designed.

The interface consists of several flags the core can use to communicate with the NIC. There is a flag for requesting a read and one for requesting write operations with data provided in an address line and a data line. The data line sends 32 bit at a time for writing cache lines of 8x32bit. Also there is a special address line for which receiver the NIC should send this request to. This was for core to core communication support, cache coherency and other future additions.

The NIC has a flag for when it’s ready to send responds data it has been receiving to the core together with a 32 line for the data.

5 Implementation on FPGA

The implementation was done in Xilinx ISE 12.4 in VHDL as the natural choice: the author as most DTU students and staff have learned, teaches and uses this in course and projects. In the author previous history there has been a general indifference between the main hardware description languages: VHDL, System-C and Verilog. Also the Tinuso project is based on VHDL and everyone involved have previous experience with the ISE tool kit and VHDL.
The development cycle as previously stated is based on agile development practices so this chapter will both describe development process related events leading to the decisions made and the final resulting implementation.

5.1 Network On Chip

At the beginning the first goal was to make a NIC, a routing component with a 16 lane connection and a on-FPGA memory controller and memory device using then NIC interface as base.

The layout of the components is implemented in component that just was named *path* which in essence could have been called NOC or SOAC – System On A Chip. This component merely contains gating and the VHDL definitions of the lanes and other Input and Output (IO) from components. The name path was just selected as a development name for the file and VHDL component name as it had to have some name and "path" was referring to the content of this file and component being the paths between components.

The network protocol was already developed at close to the final specifications except for the line ready signal part and some minor definitions. A key part of the physical document the author developed for reference of the protocol also have notes and concept details on much of what is also documented in the design section. This was the basis for the implementation.

Development testing
A setup of a test-bench with 2 routing components connected to a NIC and the NIC & memory interface controller was made with some several debug signals outputted from the path component including all lanes which is feasible in smaller model NOC simulations.
During development in VHDL the test-bench was expanded. As described in detail in in section 5.6 it became clear at the first tests that the memory and memory interface had to physically be moved to the test-bench if not the entire system was to be optimized away.
To debug routing during development a third switch and a second NIC was put into the test-bench. This made it easy to debug any unintentionally behavior of components who should be idling as no signals should be routing to or thought them etc.

5.2 Scalability

Scalability is implemented in a number of ways the three most important being modularity of design to speed up addition scaling to very large NOC grid sizes, performance related scalability.

5.2.1 Development scalability and re-usability

A core concept in NOC designs is modularity and other re-usability techniques to improve development speeds. VHDL has the component concept which makes it even quicker to reuse previously made components. Also the VHDL feature packages and in that records is used to define global finite state machines state variables and constants. The records are used to define IO of components in a simple manner like the routing component below:

```
type switch_record_in is record
      clock : std_logic;
      IN1 : unsigned(LANE_WIDTHM1 downto 0);
      IN2 : unsigned(LANE_WIDTHM1 downto 0);
      IN3 : unsigned(LANE_WIDTHM1 downto 0);
      IN4 : unsigned(LANE_WIDTHM1 downto 0);
      IN5 : unsigned(LANE_WIDTHM1 downto 0);
      SNUM : unsigned(7 downto 0);
end record;
```

Note LANE_WIDTHM1 is a constant which is defined of the constant LANE_WIDTH but Minus 1. The other inputs shown are the clock and the SNUM which is the variable for the switch's unique position in the 2D grid. When sorting component IO in an input and output records the implementation and debugging is easier by having good organization overview and faster as a result.

As LANE_WIDTH is a global constant for all VHDL files in this implementation the scalability of lane width to 32, 64 etc. is much easier to implement. In all the components decision are as much as possible based on LANE_WIDTH. The routing component waits for package transmissions to finish and go to idle after exactly the needed clock cycles by using the package data size minus lane width time’s clock cycles from connection established.

In some situations such as interfacing towards resources in the NIC and the mem-NIC which have fixed 32 bit interfaces it was not feasible to implement total independence of lane width. This means some manual editing dependent on target lane width in the VHDL code is needed to fully support a change lane width. Alternatively additional logic would have to be used which would have further increased development time and chip space usage. For an example the buffers in a NIC scales with lane width to support one lane width of data per array element but do not lower the total number of elements in the buffer. This means if lane width is changed to 32 bit many interfaces to resources would only need to fill half the number of buffer elements.

Scaling of lane width was a parameter defined as important for the researching with this NOC system for the Tinuso system so this was quite an important addition and took some development time. While a package and records was not introduced at day one in implementation when it was introduced and implemented it made it very fast to develop most of the VHDL code lane width independent.

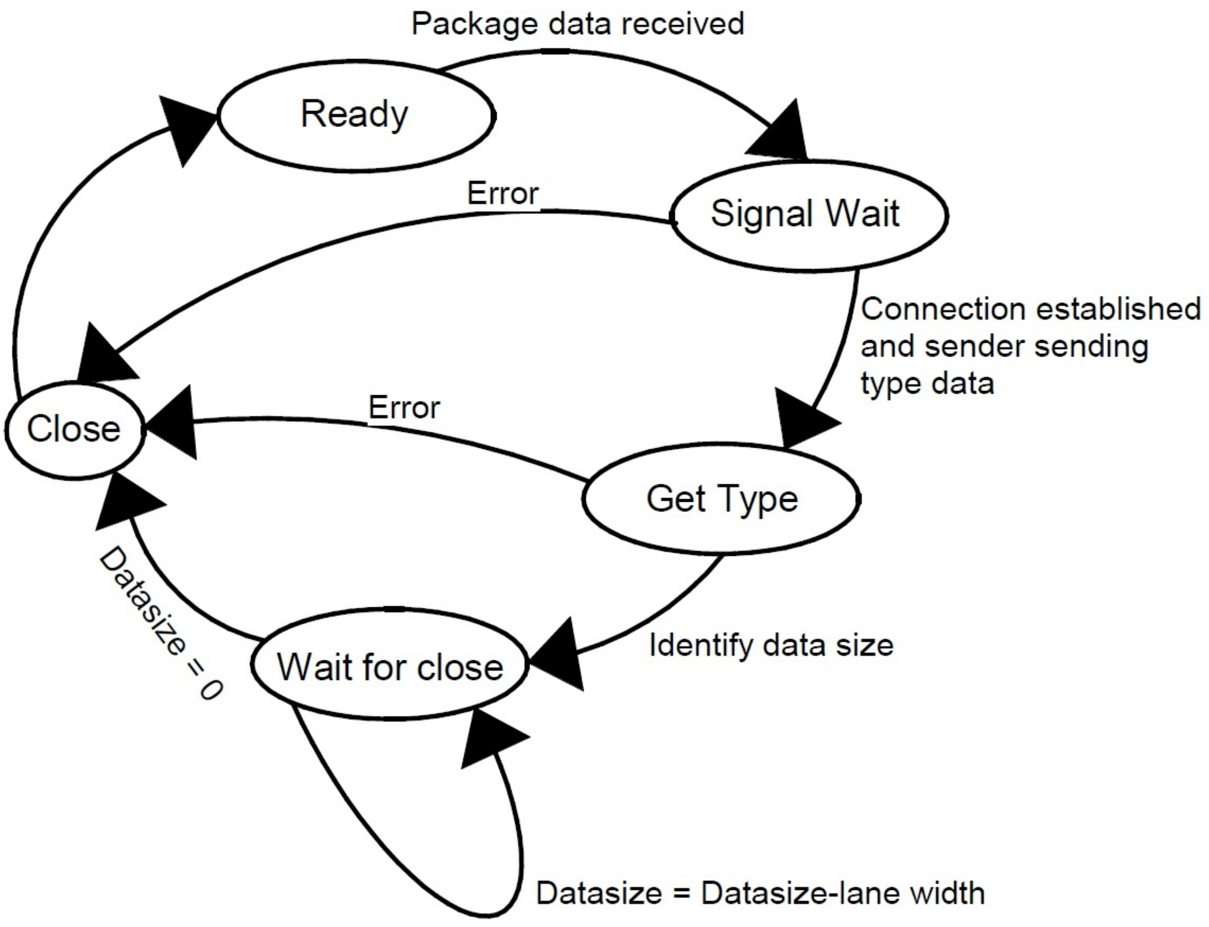
5.2.2 Performance scalability

As an integral feature of the NOC design used the general performance parameters scale with the number of resources attached. Other ways to scale performance could be to scale the lane width of the NOC. The implementation was made as independent or adjustable as feasible in regards to lane width which is already detailed in the previous section. For power and space performance parameters the lane width can even be reduced. The author estimates it's would take as much effort scaling the lane width from 16 to 8 as from 16 to 64.

5.3 Switch / router

In the following section the routing component implemented is referred as the switch as the in source file, component name and other code references it is referenced as such.

The switch is based around a process with a finite state machine:



Starting in Ready state a switch is idling. When the switch receives enough of a package header to get the sender and receiver address these are sent put into some variables that are checked at before the end of each clock cycle. In this area the input signal is routed to the relevant output signal via XY routing parameters and the torus parameters described in the design and network protocol. Then the switch state is set to SignalWait.

In SignalWait state the switch waits for the receiver of the package to acknowledge the connection is established with the line ready signal. Then the switch waits for the sender to send the type data which is also the acknowledgment from the sender that it sees the connection established signal from the receiver. This data is kept in a local register. The switch state is then set to GetType.

In the GetType state the switch converts the package data size to number of bit still needed to be transmitted and this number is stored in variable. This state was defined before SignalWait and could well be merged into that state and much it actually was but some was left as this would add additional maximum delay time to an already complex step. The switch state is then set to WaitForClose.

In WaitForClose state the switch uses the knowledge of how much data is still to be sent over the connection line to calculate how many additional cycles it should stay open with this connection. This is done by taking the initial package data type size in bits and subtracting the lane width for each clock cycle. When this counter reaches 0 the switch state is set to close.

In close state all the global signals and some variables is reset and the switch state is set to Ready. Note there is no transitional conditions for setting the state to Ready.

When a sender and receiver address is received in a switch and it is in Ready state the package header part is either ignored or used to start a routing. The way its deciding this is by first checking it’s not a line busy signal when a neighbor is indicating its busy which is indicated by all high or '1' on all the outgoing lines not used on the busy component. Also that it’s not just the connection idle signal which is all low or '0'. Then if more than one connection is coming in at the same time it’s prioritized with above first, below next, then left and then right. Incoming data from the local connected resource is prioritized last so that if that resource constantly tries to send packages it’s not always blocking others going by.

Note when a switch needs to check for the ready line signal in the SignalWait state it means at this point the switch state has to know where the data is being routed to. It has to know this so that I can look for the ready line signal coming in from the direction where the sender incoming data out. This means routing has to be determined before this can happen and happily routing decisions in NOC's have to be simple to be fast so this done in the same cycle where the sender and receiver address is received.

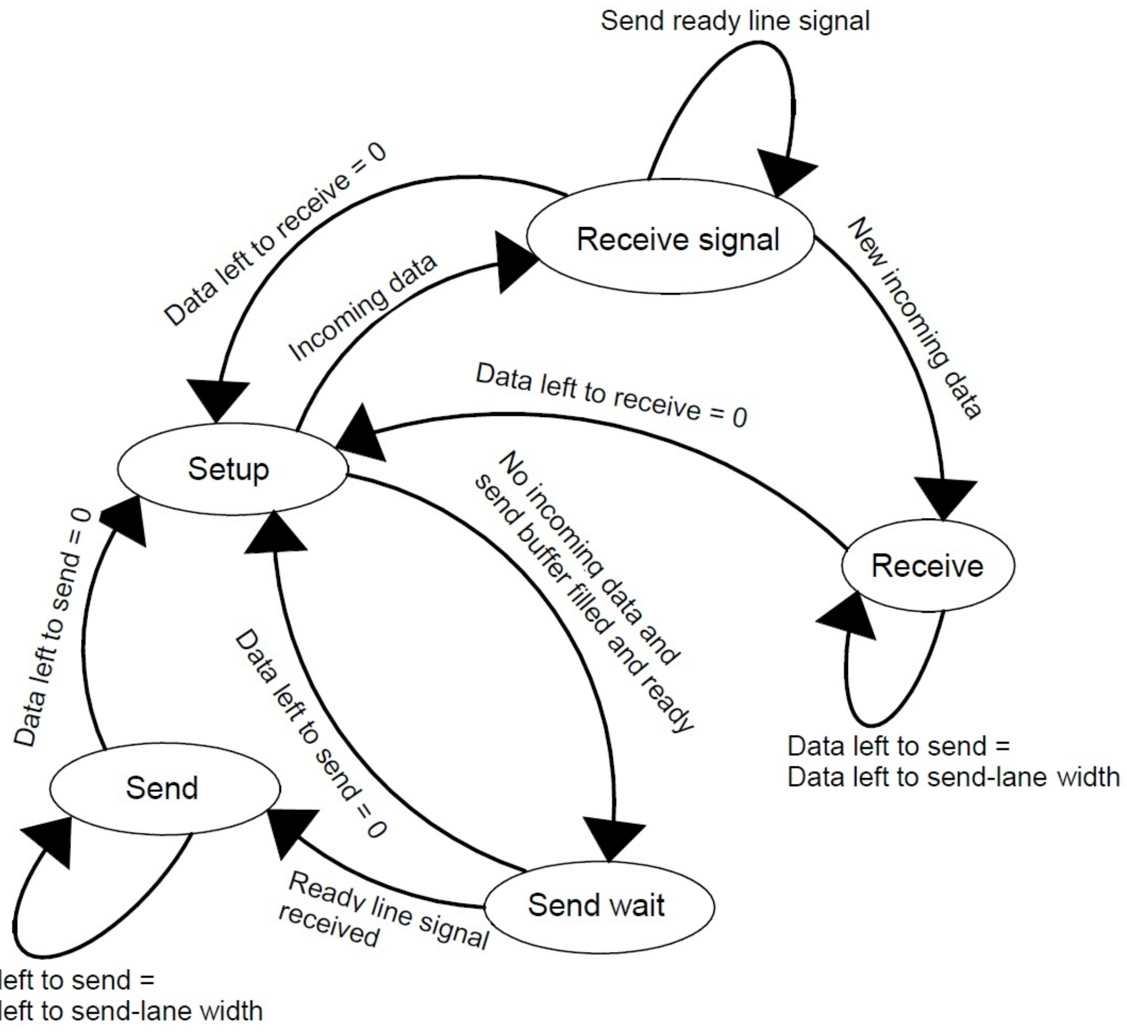
In case of an undefined even in SignalWait or GetType state is set to close.

5.4 Network Interface Controller (NIC)

The NIC is in this chapter split in two parts as the interface towards the Tinuso core is specifically implemented here as independently as possible within its own process with its own FSM. This is done so that the main NIC part towards the NOC can be reused easily for other resources.

Besides a central FSM structure the NIC also contains a third glue process which transfer the contents of sender and receiver buffers between the NIC and then Tinuso core interface processes when conditions have been met. These conditions are such that a buffer is only moved when filled and not in use - E.I. not being emptied or still in filling process. For this global signals for the component for each of the three processes and the buffers are defined with special definitions. One type is only able to be ether full or empty while the other type of signal only has ready or not ready states. I.e. this is sort of Boolean types which is just easier to debug.

NIC FSM:



Starting in Setup state the NIC will check if there is any incoming data valid data. If not it will check if the sending buffer has been filled by the glue logic and is ready to be sent. If there is incoming valid data the state is set to ReceiveSignal and the data stored in the receiving buffer alternatively if there is a package ready to be sent the state is set to SendWait after the size of the package is registered in a variable. In both situations ether the relevant sending buffer or receiving buffer in the NIC is set to NotReady meaning that the glue process are not allowed to touch the buffer.

At ReceiveSignal state the NIC will respond to the sending NIC that a connection is successfully established with the line ready signal and wait for first new piece of incoming data. When is data is received the line ready signal is removed and the data is stored in the buffer and interpreted to determine package data size. State is then set to Receive if more package data is left. If not the state is set to setup after variables and signals are reset.

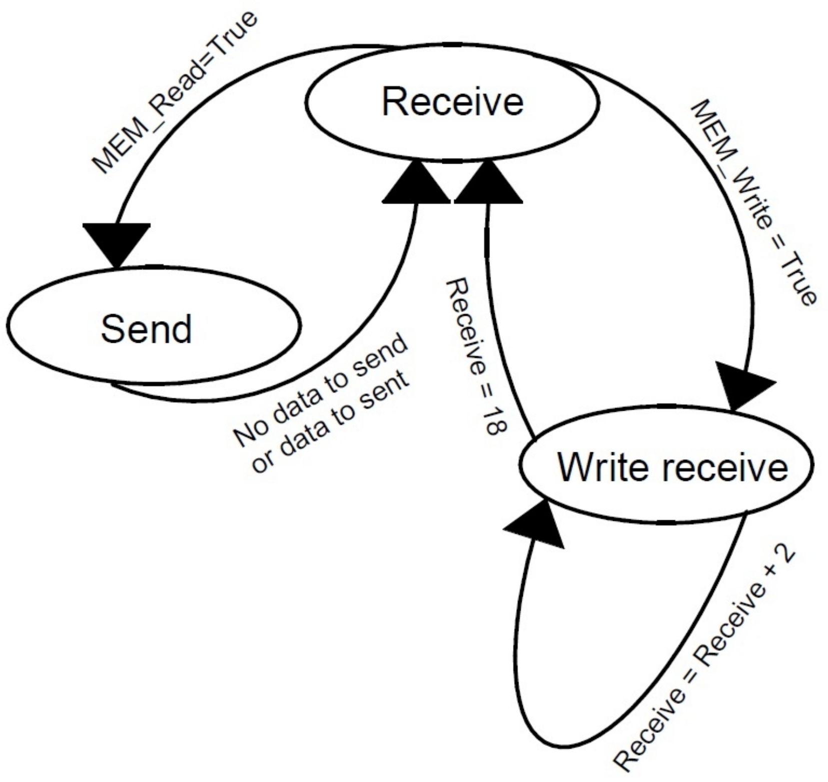
At SendWait the opposite of ReceiveSignal happens. This means the NIC here has to wait for the line ready signal to be received. When this happens the next part of the package is sent and if no more package data is left to send the state is set to setup after variables and signals are reset. If there is more data to send the state is set to Send.

In Send and Receive states the NIC is continuously sending or receiving package data until the local variable that contains the package data size is 0. Each cycle these variables are subtracted by lane width to account for how much data is send or received. When the variable reaches 0 signals and variables are reset and the state is set to setup.

Buffers are set to be filled in case of the receiving buffer after successful reception of the entire package or empty in case of the send buffer used for sending when they are done. Move flags indicating to the glue process if its allowed to touch the buffer is set to ready.

5.5 NIC interface towards Tinuso core

This is section describes how the resources process of the NIC component interfacing with the [simulated] Tinuso core is implemented. First the FSM diagram:



Starting in Receive state this process checks if there is any read or write requests from the interface. If nether of the flags are set the state is set to send.

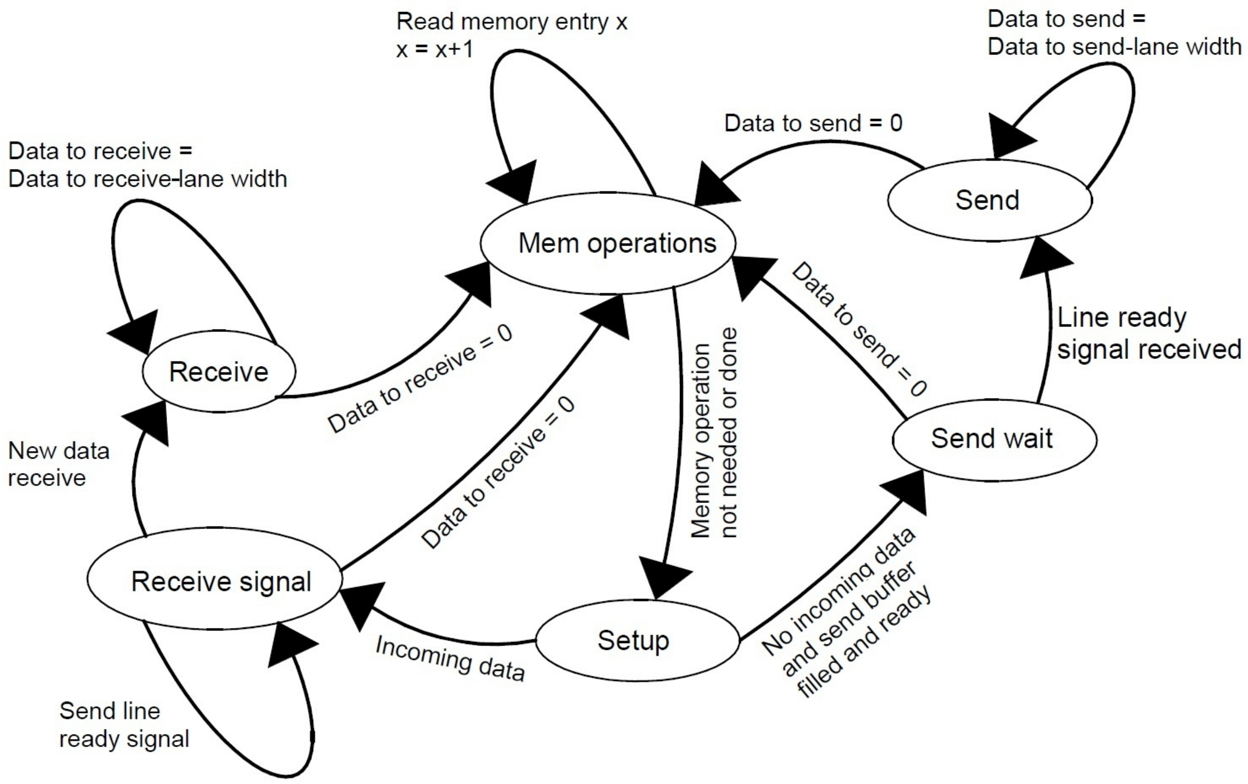
If read flag is set and the process receiving buffer is empty and ready a package is made: First the buffer is set to NotReady to keep others from using it. Then a package header is made by using the incoming core_addr signal as destination and the NIC's unique address as the sender. A read request package type is added to the buffer according to network protocol type definitions with 32 bit data size. This next 32 bit data part of the buffer is then filled with mem_addr signal from the interface which provides the intended address from which a cache line of 8x32 read and return operation is to start from. The receive buffer is then set to filled and ready so that the glue process knows it's ready for transferring to the NIC buffers. The state is set to Send.

If write flag is set a similar sequence of package building is started in the receiving buffer with the exception that the type header in the built package is then setup for write type, the buffer is not set to filled and ready just yet and in the end the state is set to WriteReceive.

In the WriteReceive state the process receives writing data of a complete 8x32 bit cache line into the receive buffer which is the data in writing type cases meant to be put into the memory at the final receiver of the NOC package. When this is done the buffer is set to "filled" and "Ready", some signals and variables is reset and then state is set to Receive.

In the Send state its check if there is any packages ready in the local sending buffer. If not state is set to Receive. If there is a ready package 32 bit of data is outputted at a time to mem_dat_read which is the signal output to the interface for ingoing data to the Tinuso core. To indicate that this process is ready for the Tinuso core to read what's in that signal a data_ready signal is toggled. In the next clock cycle this signal is then toggled back and in the following clock cycle the next 32 bit of data is sent and the data_ready signal is toggled again to indicate the new data. This repeated until all 8x32bit chunks are send to the Tinuso core and then signals and variables is reset and state is set to Receive.

5.6 Memory interface controller (mem-NIC)



The main difference between this NIC component, which has the shorted named mem-NIC, and the main NIC component is that the glue process and resource process is removed. Also in a single additional state of the NIC FSM the few interactions required to serve requests from other Tinuso cores waiting to access the memory is handled. Effectively the entire memory controller interface is handled in the MemOperations state.

This is not an interface to an external memory controller - which is part of the explanation for why it is so small. Instead this is an interface made for accessing memory on the test-bench. Originally this component in the test phase was meant to have the small memory stored directly inside it. However this would not allow in testing as the whole NOC would be optimized away. Instead the chance was ceased to implement a primitive memory interface on the test-bench.

The interface consists of a flag for requesting the data in a input lane to be written to a location in the memory defined in an address lane. Similarly there is a flag for the NIC to request a memory element which location is defined with the same address lane and the result is output from the test-bench in 32bit lane. This means a memory operation takes at least 8 clock cycles as a cache line in Tinuso is defined as 8x32bit.

To do the memory operations an address is needed. The first 32 bit of non-header data in a memory operations request package is excepting to provide this address as an integer. The next 8x32bits of the package data is then expected to be the cache line to be writing in case of writing to memory request.

As writing requests does not require a confirmation only read-return packages are made in the MemOperations state. This package is made by first locking the sender buffer of the NIC then using the sender address of the requesting package as the receiver address of the read-return package with the result data together with the local address as the sender. Each line of return data is then feed from the memory interface 32bit at a time until all 8 elements are retrieved.

All other states are exact copy of the NIC implementation – see section 5.4 for more information.

6 Test

Testing is a fundamental part of the scientific method as the guide behind making experiments to test the hypothesis. It is also fundamental to verify the function of a product and that it is within required quality parameters to be practical for the end-user. As a concept testing has several key parts and a good tester use them all to get as much data out of a hypothesis or product as possible. This data is in turn the core for evolving the hypothesis or product into a stage where it has sufficient quality and can be used as a stepping stone for other things as a completed, solid solution.

To verify the implemented designs, testing for verification are needed. The concept of testing can yield a number of data interesting related to the practicality of a product besides verification testing such as:

•Efficiency – how much energy is needed?
•Speed – how fast is the product?
•Size – is it a practical size? Is there any benefits from trying to make it smaller or larger in terms of efficiency, speed or mobility that might scale positively?

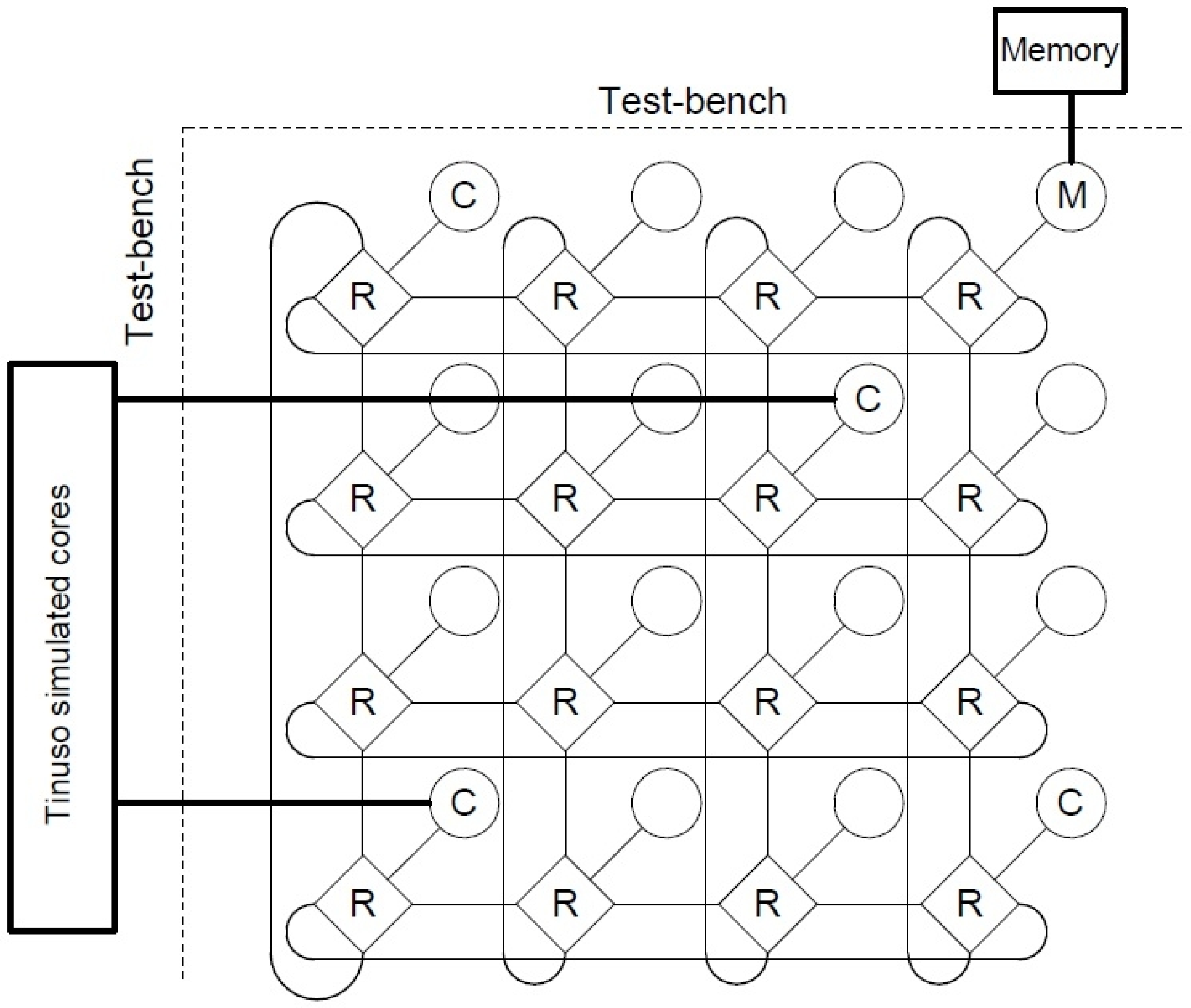
Designing tests the right way is of high importance to get credible results and useful results. In fact often its the case that results are not useful or *as* useful as needed for further progress when being done sloppy or by inexperienced testers. The test planning and design in these cases is not properly organized towards giving useful results while they most often are credible. But by being aware of some basic points tests can be planned with minimal chance of failure for the test.

The center of good testing craftsmanship is first to clearly specify what the test are testing for which is not always easy to do but easy sloppy or too loosely defined. The definition on what an expected result would be and definition of a detailed plan for how to do the test should be define. The plan has to test for specifically individual results with as few variables as possible changing unintentionally and its not uncommon to have to redo tests where parameters that can influence the test have been overlook. This is why testing can be quit time consuming. Often its also a good idea to have testing or experimental teams to work full time on such topics not at least for also removing some of the bias that all creators of a product or hypothesis have even unintentionally. Such as the confirmation bias human hypothesis creators have unintentionally at some level due to seeing things in a way that lead them to setup the hypothesis in the first place. Often separate testing also leads to testers with a different mind set than creators. This means they often spot things and make test cases the creators would never have thought of.

6.1 How the tests are designed

Testing for this thesis was done by the author that as already stated above has some issues when the author is also the implementer. Never the less with a good design one can come a long way. Especially gaining empirical data not related to verification of the hypothesis that this product I've made works for which I'm in no doubt biased. Speed of this system and such other parameters are purely readings made by other people’s tools so here the author can rid himself of more bias but even these tools have settings which could bias the measured results.

The main test-bench designed in Xilinx ISE 12.4 development tool for this system is pictured here:



What the picture is show is a 4x4 torus Network On Chip design network with multiple NIC's and simulated Tinuso cores attached to the NIC's. Also attached to the network is the special case component that is a combination of memory controller interface and a NIC shorted mem-NIC. Since mem-NIC is what provides the data for all the simulated Tinuso cores it is to be expected that close by this is where all collisions will happen. These collisions are the result of multiple NIC's trying to open lines to the men-NIC in the same period of time.
A 4x4 size network in with torus connections is also large enough to showcase the speed benefits of

the torus YX algorithm implemented. NIC's with simulated Tinuso cores attached and the mem-NIC is positioned strategically to showcase these and other test cases.
On a implementation level only 4 actual NIC's and 2 simulated cores is implemented at strategic places. The other two NIC's are intentional left idling with the possibility to easily use them or move any of the four NIC's to another router/node point. About 20 selected connections are outputted for observation in the simulation together with all input and output connections from all routers resource connections. Also observed is a ton of debug information lines for finite state machine variables and other debug relevant variables and flags.

6.2 Cross test case testing

Notice that the test cases shown in the next section can be quite similar. It’s not uncommon that one set of test actions covers multiple test cases. In this project in particular this is due to the strong interconnection between components where many features of components efficiently cannot be tested individually. This is in turn due to only the total combined system in this case has specified requirements with a high degree of freedom for implementation internally. The specification only states required behavior towards Tinuso core/pipeline, memory and by an network protocol.
That means the only way to test many features of a component is by combing it with most of the others components it needs to work with.
A functionality in component cannot be tested for failure on its own when there is no specifics on which component have to handle which task but only some general requirements they all have to serve in regards to the functionality. Such a general requirement is all components communicating in the network have to have the same lane width by network protocol design.
The protocol implementation or its design is then what fails if the test fails when a total system test is run as there is no way of saying if its component A, B or C that is to blame as the protocol does not state precisely who has to do what in implementation or it is too vague about it.

6.3 Test cases

The author has designed the following test cases:

Test target	Test steps	Expected results	Results
Speed of the entire system in MHz	1. Synthesize the system in ISE 12.4 64bit in windows 7 with a Xilinx Virtex5 XC5VLX30 at speed -3 and other settings at default 2. Record the calculated numbers	Between 100 and 400 MHz	
Functional Network Interface Controller towards network	1. Run a ISE 12.4 test-bench scenario where a full data package is send from one NIC to another via relevant number of switches. 2. Observe simulation	•No component fails. •Package is routed to destination memory and is in this at end of test. •All components return to idle.	
Functional switch/routing component	1. Run a ISE 12.4 test-bench scenario where a full data package is send from one NIC to another via relevant number of switches. 2. Observe simulation	•No component fails. •Routing and switching is done as defined in protocol. •Package is routed to destination memory and is in this at end of test. •All components return to idle.	
Functional combined memory & network interface controller (shorted mem-NIC)	1. Run a ISE 12.4 test-bench scenario where a full read line type package is send from a NIC to the mem-NIC via relevant number of switches. The mem-NIC then communicate with simulated memory and make a new package it sends back to the NIC request memory line. 2. Observe simulation	•No component fails. •Package is routed to destination mem-NIC . •Mem-NIC request correctly to memory for data line. •mem-NIC saves data line correctly in responds type package. •Mem-NIC sends responds package correctly to the NIC requesting the data. •All components return to idle.	

Functional Network Interface Controller test interfacing towards Tinuso core	1. Run a ISE 12.4 test-bench scenario where a request for a memory line is send from a simulated Tinuso core to the NIC who in turn sends a package over the network to the mem-NIC correctly. The respond line package from the mem-NIC is then correctly transformed by the NIC and send back to the simulated Tinuso core. 2. Observe simulation	•No component fails. •The NIC responds correctly to the simulated Tinuso interface request and delivers the respond data correctly. •All components return to idle.	
Functional Network on Chip total system test	1. Run a total system test with test-bench in ISE 12.4 with simulated memory and Tinuso core. 2. Look at simulation for individual component failure during simulation 3. Look for failure of components to return to idle state. 4. Check the data returned to simulated Tinuso core.	•The data returned to simulated Tinuso core is correct. •Intercommunication between components is correct. •Components does not fail. •All components return to idle state.	
Functional Multi-core Tinuso	1. Run a total system test with more than one simulated Tinuso core connected in a 4x4 or larger switch matrix network setup in torus configuration. 2. Observe simulation	•Routing of data is done as designed in protocol. •There is no loss of data and all requests are eventually served correctly. •All components return to idle.	

Multi-core Tinuso torus routing test	1. Run a total system test with more than one simulated Tinuso core connected in a 4x4 or larger switch matrix network setup in torus configuration. 2. Observe simulation for faster routing by using edge connections from torus design	•No errors by using torus edge connections routing. •Routing is faster in cases where edge connections are usable.	
Shortest cycle time for memory line read from core request to end of responds	1. Run a total system test with more than one simulated Tinuso core connected in a 4x4 or larger NOC setup in torus configuration. 2. Connect requesting core to NIC at node (1,1) and mem-NIC to top right node; (4,4) in a 4x4 NOC. 3. Observe simulation	•Measurement in integer number of clock cycles for shortest cycle - E.I. without waiting on other connections blocking	

7 Results & discussion

7.1 Results

Test target	Result
Speed of the entire system in MHz	Clock period: 7.110ns (Maximum Frequency: 140.655MHz) [T1]
Functional Network Interface Controller towards network	All Pass
Functional switch/routing component	All Pass
Functional combined memory & network interface controller (shorted mem-NIC)	Pass with conditions[T2]
Functional Network Interface Controller test interfacing towards Tinuso core	Pass[T3]
Functional Network on Chip total system test	All Pass
Functional Multi-core Tinuso	Fail[T4]
Multi-core Tinuso torus routing test	Pass with conditions[T5]
Shortest cycle time for memory line read from core request to end of responds	95 clock cycles[T6] * 7.110 ns = 675.45 ns

T1: Total system doesn't fit on the tested FPGA device. The device used was chosen at development start and was only kept for easy comparison to older revision. Also the tested Xilinx ISE version is a light license which doesn't support much new, bigger or faster FPGA's. This results means the NOC runs roughly half the speed of the Tinuso core [1].

T2: The mem-NIC has some of the slowest parts according to the synthesize report. In the known limitations section 7.2 is a discussion of a deadlock limitations that depending on how they are solved could be blamed on the mem-NIC implementation.

T3: See Tinuso implementation interface part of "known limitations" section 7.2.

T4: Fails in many cases. This is due to a deadlock issue when the more than one NIC wants to send a package to the mem-NIC. Specifically in many cases the timing will be so that when the mem-NIC tries to return a responding cache line its router is already released and at this time waiting NIC connections will cease the router. This means the mem-NIC will try to communicate out while the core will try to communicate with the mem-NIC sending both into infinite wait state. Solution is detailed in "known limitations" section.

T5: The initial data request package going from (1,1) is correctly routed to (4,1) and then to (4,4) by routers.

T6: 95 clock cycles was measured off test bench 4x4 size from simulated core on node (1,1) to mem-NIC at node(4,4). Start was from first flag set true towards the NIC at simulated core interface. End recorded was when last memory data was send to simulated core. Note that in a non-torus design this would have been the longest path but in this implementation it's one of the shortest routes but not the shortest which would be sending from NIC next to the mem-NIC. That would be the positions (1,4), (3,4), (4,1) or (4,3) in this case. Longest route could with the implemented routing algorithm go from (2,2) giving 2 more nodes to go though and 4 more clock cycles in total including the line ready signal.

7.2 Known limitations

There are a few known issues or limitations to the design and implementation that is noteworthy:

Tinuso implementation interface

The Tinuso implementation interface have 2 different directions and the cores creator, Pascal, made one for the NOC system he has used for testing that differs from the one used here – both where developed simultaneously with slightly different goals out of the same agreed design.

In the Pascal's version the core expects to be given a memory data line each cycle when the NIC sets flags for its ready to feed the core. In this project a line will be feed every second cycle and the feed ready flag is turn off and on to indicate. Also some of the signals from core to NIC are not turned false which if used with this projects implementation would in some cases confused the NIC to think there was 2 requests from the core.

This is because this projects implementation of the NIC supports multiple requests from each core before responses have been returned. I.e. This project NIC has additional features that make it better to implement the protocol in a different way. While the Tinuso core can't handle several active memory requests or core to core communications right now it’s important for future efficient core to core communications and cache coherency. Note that some of the other known limitation’s suggested solutions have to be implemented for the rest of this system to fully support these features.

Core to Core communication deadlock will occur:

As core to core communications was only prepared and not part of the specifications this is a well-known limitation. Basically the feature is not fully implemented in at the NOC level. When two cores try to send to each other at the same time or more cores in specific conditions the classic deadlock scenario will happen where cores are waiting for each other to successfully send to the core sending to them or in a circular chain. That means core A waits for core B who waits for core C who waits for core A – leaving everyone deadlocked.

This can be fixed in a number of ways. One solution would be to have a timeout counted in clock cycles in sending-wait states followed by a unique waiting period for each node (router and NIC) before the NIC try to send again. That way eventually one sender will win and force the other sender into receiving state. The timeout clock cycle counter should count at least as long as the longest possible connection time needs +1cycle.

Mem-NIC specific deadlocks

The mem-NIC related deadlocks with multicores have similarities with the core to core limitation. The issue is in part stated in T4 and T2 of the results section. There is an issue with the mem-NIC being blocked if it tries to send a responds package to a NIC and it has another NIC wanting to communicate with it at the same time.

Specifically in cases when a seconding NIC trying to also communicate with the mem-NIC has reached very close to the router mem-NIC is connected to and the mem-NIC releases its router before going to idle after receiving the data from the first NIC. At this point the second core will cease the mem-NIC router while the mem-NIC tries to send to the responds to the first NIC leading to both second NIC and mem-NIC waiting infinitely for getting the signal that their target received their initial package part (could be more than header depending on lane width).

Here is 3 solutions:

1.Implement a similar sending-wait state counter to the core to core communication deadlock that will close the line after a max normal routing time + some cycles. The number of additional cycles is a tradeoff between waiting a short time for someone else to get done and waiting too long while deadlocked. At this trigger point each involved NIC waits a unique number of cycles while routers go back idling. Due to the unique waiting period at some point, hopefully the first time, on of the sending NIC's will have broken the deadlock by getting to its end point and for it into receiving mode.

2.For requests to the mem-NIC that needs a responds a special data size option for the requesting package could be implemented together with changes to the mem-NIC and NIC's. This data size option dynamically and scale with the longest route in the NOC times some factor depending on the speed of the network and how long it takes for the mem-NIC to make and send responds package + normal request time. The mem-NIC then haves an extra state it goes to with this package type sending directly to NIC before going to idle. The NIC have to also implement a special state for this type of message to accept the data from the mem-NIC. This solution is less than elegant and claims a lot of network connections for a long time.

3.Implement a version of the second solution only instead of waiting a set number of cycles dynamically determined at VHDL synthesizes point; the NOC will not close special type connections unless the sender of the original message sends a special signal after initial normal package length.

4.Sacrifice some space and connections for a controller with 2 wires to each NIC. Before sending the problematic type of packages each NIC then has to request on one of its two wires to get allowed connection to mem-NIC. The controller then if the mem-NIC is free or the controller determines it’s the requesting NIC's turn, sends a signal back to the NIC via the other wire saying the NIC is free to send to mem-NIC. When the NIC then have received the first part of the responds package from the mem-NIC the NIC removes its request for sending to the mem-NIC on the controller. Now the controller knows it’s done. While this would limit the thought put for core to mem-NIC compared to solution 2. and 3. somewhat in that its only lets other NIC trying to open a connection several cycles later than 2. and 3. there is more sever issues. This solution uses more chip space and has potentially very long wire length connection each node with the central controller possibly slowing down the clock frequency but it also has advantages. The key advantage being that no NIC hogs NOC bandwidth and connections unnecessary waiting for a single resource that it already would know it can acquire in possible many cycles. It also distributes the potentially scares resource of mem-NIC more reasonable towards NIC far away from the mem-NIC where closer NIC's in the direct path could even starve faraway NIC's by blocking them.

8 Conclusion

In this project a NOC solution this designed for the Tinuso processor cores so that these cores can access main memory over the NOC. This was designed by analyzing current theory and design concepts based on specific prioritization of requirements. This prioritization was achieved via agile analysis of the base requirements, ideas for extensions and division of the base requirements into sub components and technologies.

The design was implemented in VHDL for FPGA's and tested towards a simulated Tinuso core interface. This interface was design in cooperation with the Tinuso core developer[1] which developed his own test-bench. As a result slight differences were found between the actual core's expectations and what was used in the simulations here. The differences are documented in results.

Clock frequency results obtained from synthesis indicates that the NOC has to run at about half the clock frequency of the Tinuso cores[1]. This was expected as the implementation has not been optimized for high clock frequency. This is the raw results from first total system tests of an experimental prototype: the system is synthesized on a FPGA that is too small to even map it in synthesis.

The implemented solution demonstrates the feasibility of the design and network protocol when tested. It also demonstrates how many challenges there are in designing and implementing deadlock free solutions in concurrent systems. Concurrent access to the main memory from multiple cores failed in many cases in the test-bench as a result of a specific deadlock situation.

Most of the deadlocks were even expected as the testing went outside the requirements for this version of the NOC system. This was done to test support for interesting extensions such as core to core communication or and cache coherency.

Several solutions for the few deadlocks situations experienced in testing has been suggested demonstrating that there is many ways, with different tradeoffs to handle deadlocks.

9 References

[1] Pascal Schleuniger and Sven Karlsson

Tinuso: A processor architecture for a multi-core hardware simulation platform

Paper from DTU IMM ESE section.

[2] www.ing.dk / www.version2.dk / Ingeniøren

The danish engineer news media organization behind the engineering news paper "Ingeniøren" have made many different articles and interviews about development models. Recently a interview was made with a bank where they use both traditional and agile development on a number of projects and teams. One of the banks project managers state they often experienced the remaining reasons for continuing developing on an agile project where all the "need to have" components was done had little value compared to the other projects the teams resources could be diverted to. Also some of teams where finished significantly faster on projects where agile development suited the project type. Danish link: http://www.version2.dk/artikel/2000-danske-bank-udviklere-arbejder-baade-agilt-og-med-vandfald-19403

[3] Aline Vieira de Mello, Luciano Copello Ost, Fernando Gehm Moraes & Ney Laert Vilar Calazans

Evaluation of Routing Algorithms on Mesh Based NoCs

Paper

Additionally the following books and papers have been used for inspiration:

Rickard Holsmark & Magnus Högberg

Modelling and Prototyping of a Network on Chip

Master of science thesis from 2002 from Ingenjörshögskolan in Jönköping

Axel Jantch and Hannu Tenhunen (Eds.)

Networks on Chip

Book from Kluwer Academic Publishers

10 Vocabulary

NOC	**- Network On Chip**. Typically a mix between a star topology and IP network custom made for the specific needs of communication between many resources inside one chip. One of its goals is to be fairly modular and easy scalable when more resources are attached. Also compatible with other chip designs of the same type is an option. See this reports theory section for in-depth details.
Resources	- In NOC terminology resources is the module the NOC services and the NIC directly talks to: E.I. The processor cores, memory attachments, DSP units or other components that need to interconnect with the other on-chip components.
NIC	**- Network Interface Controller** is the controller interfacing between the resources i.e. in this a Tinuso processor core and the network.
Core	**-** Used both in popular media and in scientific community as the major module consisting of some level of ALU's, registers etc. that more or less could be an entire processor in vintage systems. In GPU's they are often very small and light and could not live on its own being only part of the pipeline while in this Tinuso design and other generic processor designs like x86 Intel and AMD designs they are in fact best described as an entire processor just optimized in some level to take advantage of the multi- or many-core nature where each processor does not need a separate device interface etc.
Switch or router	**-** These two have different meanings depending on the context. In this these they are referring to a device who is the main distributor or handler for the network traffic in a NOC system. A policy for routing the traffic is implied and this policy is shared with the NIC's at the sender and receiver devices. The policy is a core part of the design of a network protocol for NOC's.
Node / Client	**-** In network terms this is a device which is connected to others and can send and receive. A human in the mail system is a client as it needs the services of the post distribution system for sending and receiving. A node is generally the same except it also in some network topologies can route or switch data for others.
VHDL	**- Very high speed integrated circuit Hardware Description Language** was initially developed by US Department of Defense's DARPA like the internet for their internal needs. VHDL was originally only used to describe digital circuits but is widely used for design, synthesis and research of circuits together with FPGA's. Direct sales or end user usage of FPGA's board designs running VHDL is often used as in the case of Tinuso and many small unit number commercial products where FPGA's additional size, clock top frequencies and power needs does not compromise the product as ASIC's, generic DSP's and processors and other solutions is often too expensive or too slow.
FPGA	**- Field Programmable Gate Array** is a logic device that uses static ram to store its configuration. Basically it's a very high logic capacity version of a fumble board with a set number of logic devices predefined on a single chip. FPGA's has to be reconfigured after power shutdown.

Cache coherency

- cache coherency is a technology used to synchronize all local caches of cores in a multi- or many-core processor with each other and the main memory. This is an integral part for getting high performance out of multi-cores systems. The coherency part is particular important in that not only do the memory have to be synchronized – it has to be coherent at all times. If two cores are operating on the same memory element only one of them in a defined manner can update it at a time which has to be synchronized to others using it.

Finite State Machine (FSM)

- A behavioral model and tool for designing hardware and software where the device is always in one of a finite number of states which each is predefined and defines the devices current behavior. Transition between states is defined by transition conditions which the device has to meet to transit. FSM's are very good for making predictable event driven behavior.

Process - In VHDL context a process is a section of code entered in each clock cycle which is serially executed. This is contrary to VHDL outside a process which is concurrent.

Encrypted - To use mathematical algorithms or formulas to secure data so no-one else but the intended receiver can read it as other than random garble. The intended receiver will in turn do a decryption – E.I. a reversing process which turns the data into readable material again via the receiver's knowledge of the encryption system and a shared key information that only the sender and receiver knows. This shared "key" makes sure that even if others get the encrypted data and knows the entire encryption system or even was the designer they still have no chance in reasonable time to get meaningful results out of the encrypted data. "Reasonable time" can be anything from seconds on a PC to millions of years with all the computer and manpower in the world depending on how long and how critical the data transport has value for others than they intended communicators.

End of thesis