ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

High-Level Implementation of Customized Network-on-Chip Architectures

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΔΕΣΠΟΙΝΑ-ΟΛΓΑ ΚΑΜΠΑΝΗ

Επιβλέπων : Δημήτριος Σούντρης

Επ. Καθηγητής ΕΜΠ

Αθήνα 2010

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την .... Σεπτεμβρίου 2010

.....	.....	.....
Επ. Καθηγητής	Καθηγητής	Λέκτορας
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γιώργος Οικονομάκος

.....

Δέσποινα - Όλγα Καμπάνη

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

Contents

Περίληψη

Αντικείμενο της παρούσας διπλωματικής είναι η ανάπτυξη ενός εργαλείου, το οποίο θα παράγει "Irregular Network on Chip Architectures". Οι αρχιτεκτονικές αυτές θα σχηματίζονται σύμφωνα με προδιαγραφές που θα δίνονται ως είσοδος και οι οποίες θα περιγράφουν τις ανάγκες σε πόρους και τεχνικές που θα πρέπει να διαθέτει η αρχιτεκτονική.

Στο κεφάλαιο 1 γίνεται μια εισαγωγή στα βασικά στοιχεία μιας NoC αρχιτεκτονικης, όπως η τοπολογία, ο αλγόριθμος δρομολόγησης, οι διαστάσεις των buffer, ο αριθμός των εισόδων και των εξόδων, οι τεχνικές διαιτησίας κ.λ.π.

Στο κεφάλαιο2 αναπτύσσεται η επιχειρηματολογία για την ανάγκη εύκολης τροποποίησης των χαρακτηριστικών μιας NoC αρχιτεκτονικής, η οποία θα μπορεί να συνδυάζει και να ενσωματώνει ιδιότητες από διαφορετικά μοντέλα. Επίσης παρουσιάζεται η προτεινόμενη λύση καθώς και πώς αυτή εντάσσεται σε μία ευρύτερη προσπάθεια διερεύνησης των NoC αρχιτεκτονικών.

Στο Κεφάλιο 3 αναλύονται διεξοδικά τα βήματα που χρειάστηκε να ακολουθηθούν για τη δημιουργία μιας "NoC πλατφμόρμας" γενικής χρήσης, η οποία θα μπορεί ανάλογα με τις προδιαγραφές να μεταβάλλεται. Περιγράφεται η τροποποίηση του κώδικα, ώστε να συμμορφώνεται με τις αρχές της reusability, η ενσωμάτωση των διάφορων components στην τελική οντότητα καθώς και η δημιουργία τεχνητής κίνησης που θα ελέγχει την ορθότητα της λειτουργίας.

Στο Κεφάλαιο 4 περιέχεται η περιγραφή και high-level εργαλείου αυτόματης παραγωγής NoC αρχιτεκτονικών, το οποίο θα είναι σε θέση να συνθέτει - σύμφωνα με ένα αρχείο προδιαγραφών γραμμένο σε xml - τον κώδικα που θα δίνει την τελική αρχιτεκτονική και θα την τροφοδοτεί με αιτήματα αποστολής δεδομένων.

Στο Κεφάλαιο 5 δίνονται δείγματα της λειτουργίας του εργαλείου για διαφορετικές προδιαγραφές. Επίσης παρουσιάζονται μετρήσεις καταναλώσεων ισχύος και χώρου στο chip που καταλαμβάνει η κάθε αρχιτεκτονική όταν τροφοδοτείται με κίνηση από διαφορετικές εφαρμογές.

Στο Κεφάλαιο 6 παρουσιάζονται τα συμπεράσματα της εργασίας καθώς και ιδέες για μελλοντική έρευνα.

Λέξεις-Κλειδιά

NoC, reusable, reconfigurable, specifications, xml, automatic generation

Abstract

The purpose of the present diploma thesis is the development of a tool, which will produce Irregular Network on Chip Architectures. These architectures will be formed according to input specifications, which will describe the resources and techniques that should be part of the architecture.

In Chapter 1 we make an introduction to the basic concepts of a NoC architecture, such as the topology, the routing algorithm, the buffer dimensions, the number of inputs and outputs and the arbitration techniques.

In Chapter 2 we explain the need of being able to easily modify the characteristics of a NoC arhictecture in order to combine and encompass properties of different models. Furthermore, we describe the proposed solution and how this is integrated in a more general effort to explore NoC architectures.

In Chapter 3 we analyze in detail the steps which were followed in order to create a general purpose "NoC platform", which would be reconfigurable. We describe the code modifications which were necessary to adhere to the "reusability" principle, the integration of the various components in order to create the final entity and finally the generation of artificial traffic which would test the correctness of the design.

Chapter 4 includes the description of the high-level, which will be able to produce – taking as input xml-based configuration files- the vhdl code for the NoC design and its testbench.

In Chapter 5 we present samples of the tool outputs for different configurations. Furthermore, we display the measurements of power consumption and place utilization on the chip.

In Chapter 6 the conclusions are present as well as ideas and proposals for future research.

Keywords

NoC, reusable, reconfigurable, specifications, xml, automatic generation

Ευχαριστίες/ Acknowledgments

Για την εκπόνηση της παρούσας διπλωματική θα ήθελα να ευχαριστήσω τον Επ. Καθηγητή του Ε.Μ.Π. κ. Δ. Σούντρη για την αμέριστη συμπαράστασή του, την εμπιστοσύνη του και τις πάντοτε καίριες και χρήσιμες συμβουλές του που με καθοδήγησαν στη μελέτη αυτή. Επίσης η εργασία αυτή δε θα είχε έλθει εις πέρας χωρίς την πολύτιμη συνεργασία των μεταδιδακτορικών φοιτητών Κώστα Σιώζιου, Κωνσταντίνου Τάτα, Αλέξανδρου Μπάρτζα, καθώς και τον υποψήφιο διδάκτορα Διονύση Διαμαντόπουλο, οι οποίοι βοήθησαν να ξεπεραστούν όσα σημαντικά προβλήματα προέκυψαν κατά τη διάρκεια της διπλωματικής.

Chapter 1:

Network on Chip Architectures and their various Characteristics

Abstract

In this chapter, we introduce the Network-on-Chip architectures. We make a comparison between other on-chip communication solutions and explain their advantages and disadvantages. Furthermore, we describe their basic features and techniques deployed in their design.

1.1 Introduction

The aim of this chapter is to make a brief introduction into the NoC architectures. The term NoC has a quite short history. It refers to a new approach to the design of the communication subsystem of a System On Chip (SoC).

Traditionally the design space exploration for SoCs has focused on the computational aspects. However, the number of components on a single chip and their performance increased to such an extent that the design of the communication architecture plays a major role in defining the area, performance and energy consumption of the overall system. Modern SoC architectures consist of heterogeneous IP cores such as CPU or DSP modules, video processors, embedded memory blocks etc.

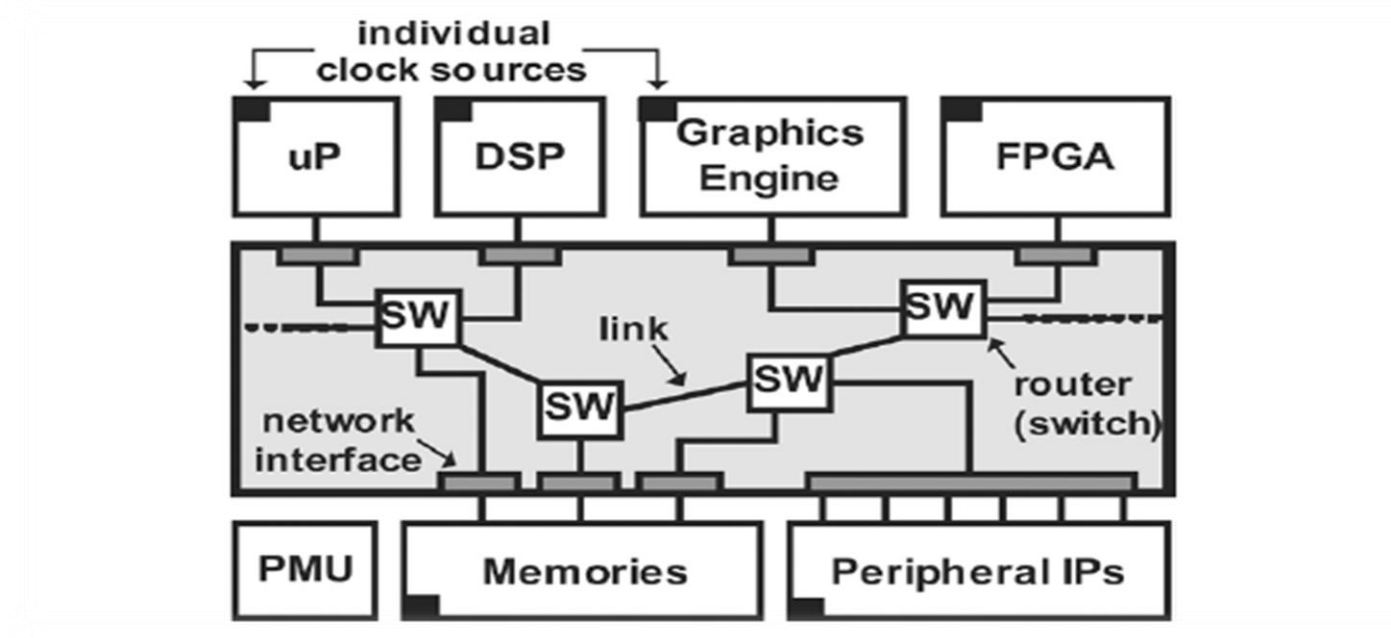

Figure 1 : **Heterogeneous NoC architecture**

Furthermore the classical bus-based and point-to-point communication solutions failed to address the new needs. Bus based architectures were abandoned for complex designs mainly because of the delay factor (bottleneck when many components are connected). The point-to-point solution is not viable for chips with many components, since the number of connections lead to a great waste of energy and space on chip. In Table 1 we can see a summary of the basic differences of bus and NoC architectures.

Bus Pros & Cons			Network Pros & Cons
Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.	−	+	Only point-to-point one-way wires are used, for all network sizes, thus local performance is not degraded when scaling.
Bus timing is difficult in a deep submicron process.	−	+	Network wires can be pipelined because links are point-to-point.
Bus arbitration can become a bottleneck. The arbitration delay grows with the number of masters.	−	+	Routing decisions are distributed, if the network protocol is made non-central.
The bus arbiter is instance-specific.	−	+	The same router may be reinstantiated, for all network sizes.
Bus testability is problematic and slow.	−	+	Locally placed dedicated BIST is fast and offers good test coverage.
Bandwidth is limited and shared by all units attached.	−	+	Aggregated bandwidth scales with the network size.
Bus latency is wire-speed once arbiter has granted control.	+	−	Internal network contention may cause a latency.
Any bus is almost directly compatible with most available IPs, including software running on CPUs.	+	−	Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems.
The concepts are simple and well understood.	+	−	System designers need reeducation for new concepts.

Table 1 : **Bus-versus-Network Arguments**

As a result, the NoC approach emerged as a promising alternative. The network approach is the evolution of former on-chip communication structures. Unlike busses and dedicated point-to-point links, a more general scheme is adapted, employing a grid of routing nodes spread out across the chip. It achieves better performance for many cores because connections between components are relatively fast for any size of chip, assuming a few hops between components.

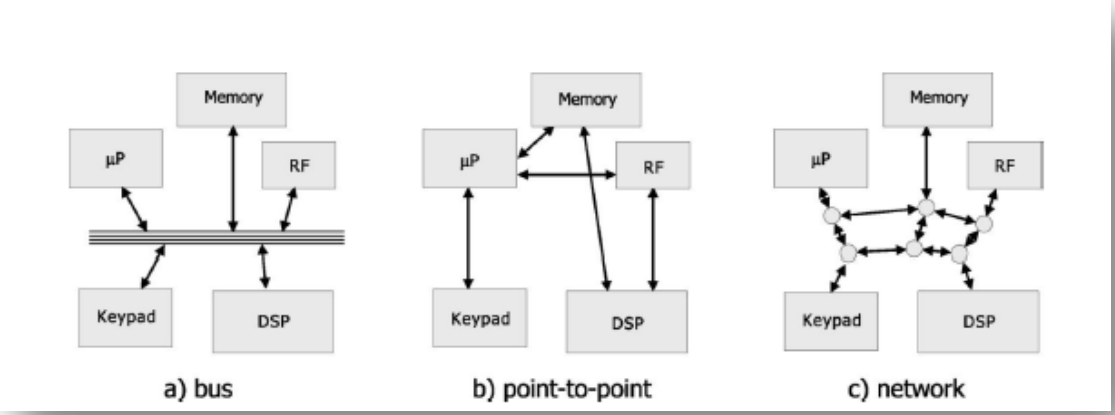

Figure 2: Common communication structures

In a NoC architecture when a source node sends a packet to a destination node, the packet is first generated and transmitted from the local processor to the attached router via a network interface (NI). The NI enables seamless communication between various cores and the network. Then, the packet is stored at the input channels and the router starts servicing it. This service time includes the time needed to make a routing decision, allocate a channel and traverse the switch fabric. After being serviced, the packet moves to the next router on its path, and the process repeats until the packet arrives at its final destination. As a result, the communication among various cores is achieved by generating, processing and forwarding packets through the network infrastructure rather than by routing global wires.

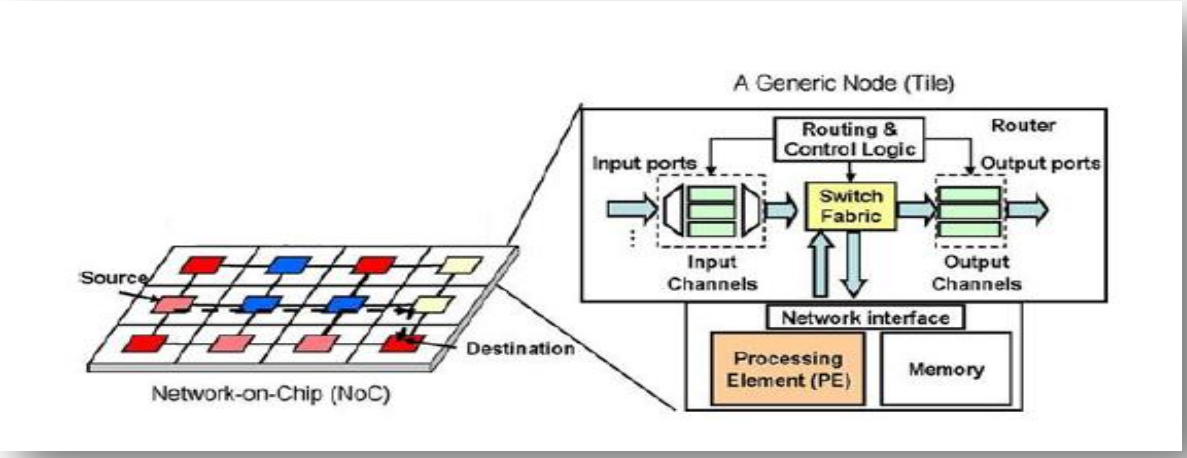

Figure 3: Generic NoC Architecture. The anatomy of a node which consists of an on-chip router, buffers and processing element(PE) is also shown on the right-hand side.

Not surprisingly, the network communication overall performance (latency, energy consumption, space overhead) depends on the characteristics of the target application (e.g., inter-task communication volume), computational elements (e.g., processor speed) and network characteristics (e.g., network bandwidth and buffer size).

1.2 NoC Communication Paradigm

1.2.1. Routing Algorithm

The routing algorithm determines the route which a message will follow from its source to its destination through the network. Its role is really crucial as it impacts all network metrics, namely latency (as the hop count is directly affected by the actual route), throughput (as congestion depends on the ability of the routing protocol to load balance), power distribution (as routing can be used to channel different message flows along distinct paths to avoid interference), and finally reliability (as the routing protocol needs to choose routes that avoid faults).

A designer has to take into consideration all the above parameters and the tradeoffs between them in order to produce the routing protocol which fits at best the application and constraints. Usually we want to minimize the average distance traveled by packets in the network, with a constraint on the maximum distance between any pairs of nodes.

A popular and simple routing technique is the dimension-ordered routing which routes packets in one dimension, then moves on to the next dimension, until the final destination is reached. (Manhattan)

While the above technique is often implemented, adaptive routing techniques are the key when we ask for better throughput and fault tolerance. Adaptive routing enables alternative paths, depending on the network congestion and run-time faults.

There have been also tested techniques, which are a hybrid and combine features of both deterministic and adaptive routing.

Furthermore there are the oblivious routing techniques, which generate routes without any knowledge of the traffic.

Finally, there is the fully customized approach of the routing table. The routing table contains the next destination of a packet (namely the output port) according to its current location and its destination. This technique is application-specific and there are works which propose thermal-(L. Shang et al : "Dynamic voltage scaling with links for power optimization and management of on-chip networks") and reliability-aware(Manolache et al : "A network traffic generator model for fast network-on-chip simulation") routing algorithms.

When it comes to irregular topologies the routing algorithm becomes a key issue. Despite the existing research, it remains difficult to find minimal routes and at the same time avoid deadlock and livelock situations. On the one side it is not always reliable to rely on dimension-ordered routing and on the other side routing tables incur delay, area and power overheads.

To conclude, routing algorithms remain a challenging area of NoC research. Although there is research into routing algorithms for off-chip interconnection networks, the different nature of on-chip communication (high frequency, low latency) leads to implications. Sophisticated solutions cannot be lightweight enough and simple ones fail to accomplish tasks as reliability and low-power design. To date, the vast majority of NoC routing solutions have focused on unicasting (i.e., sending from one PE to another).

1.2.2 Switching Technique

Switching technique, or flow control, governs the way in which messages are forwarded through the network. Typically the messages are broken into flow control units (flits) which represent the smallest unit of flow control. The switching algorithm then determines if and when flits should be buffered, forwarded or simply dropped. Mainly it addresses the issue of ensuring correct operation of the network.

Among the commonly used techniques in interconnection networks, wormhole switching seems the most promising for NoCs due to the limited availability of buffering resources and tight latency requirements.

Virtual channels have been also adopted for NoC design to improve network bandwidth and tackle deadlock.

Depending on the case, we can choose between:

- Circuit and packet switching: In circuit switching the circuit from source to destination is setup and reserved until the transport of data is complete. Packet switching on the other hand is forwarded on a per-hop basis, each packet containing routing information as well as data.

- Connection-oriented and connectionless switching: Connection-oriented mechanisms involve a dedicated logical connection path established prior to data transport. The connection is then terminated upon completion of communication. In connectionless mechanisms, the communication occurs in a dynamic manner with no prior arrangement between the sender and the receiver, Thus circuit switched communication is always connection-oriented, whereas packet switched communication may be either connection-oriented or connectionless.

1.2.3 Quality of Service and Congestion Control

Conventional packet-switched NoCs multiplex message flows on links and share resources among these flows. While this results in high throughput, it also leads to unpredictable delays per individual message flows. For many applications with real-time deadlines, this non-determinism can substantially degrade the overall application performance. Thus, there is a need for research into NoCs that can provide deterministic bounds for communication delay and throughput. We need to find a resource allocation strategy (size of output buffers of router, bandwidth of channel and/or packet injection rates in the network.

This problem is addressed adopting methods such as :

- virtual channels

- multiple priority levels for urgent traffic and regular traffic

- QoS-aware congestion control algorithms

1.2.4 Topology

Topology refers to the structure of the network and its organization. More specifically, it has to do with the number of PEs, routers, links and the graph structure interconnecting them. There are different approaches when it comes to the selection of the topology model. Parameters such as simplicity and regularity play a significant role, since regularity improves timing closure, reduces dependence on interconnect scalability and enables the use of high-

performance circuits. However, the target application traffic profile has to be taken into consideration as well, as far as the placement and interconnects are concerned.

Typically, 1-D and 2-D topologies (mesh, torus etc) are the default choices for NoC designers and constitute over 60% of cases. Mesh and torus topologies have 4 neighbor nodes but torus has wraparound links connecting the nodes on network edges and mesh does not.

The k-ary tree and the k-ary n-dimensional fat tree are two alternate regular forms of networks explored for NoC.

The Octagon NoC topology presented in Karim et al.(2001,2002) is a further example of a novel regular NoC topology. Its basic configuration is a ring of 8 nodes connected by 12 bidirectional links which provides two-hop communication between any pair of nodes in the ring and a simple, shortest-path routing algorithm. Such rings are then connected edge-to-edge to form a larger, scalable network.

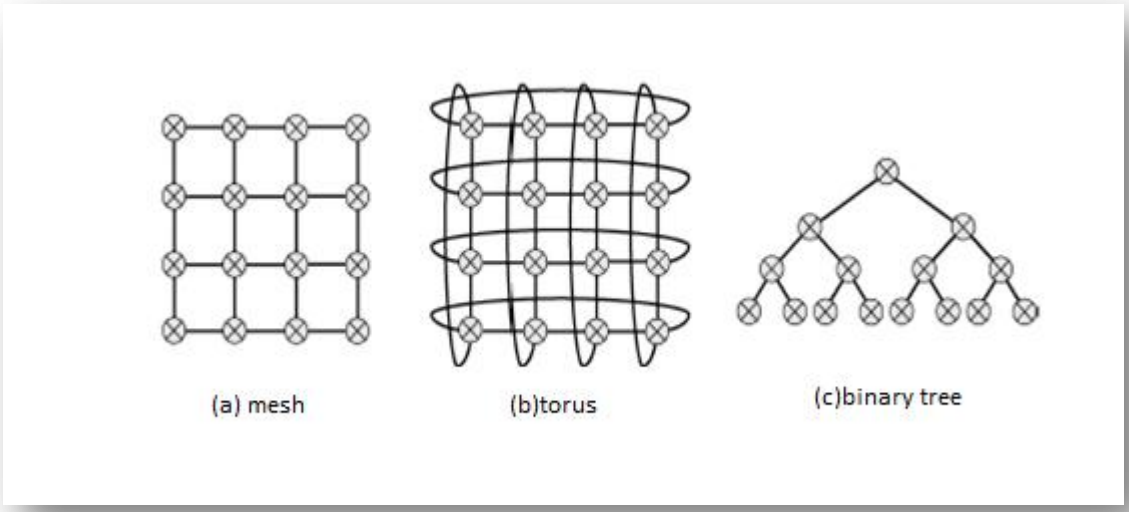

Figure 4: **Regular forms of topologies**

But so far various irregular topologies have been developed and investigated as well. They offer specific solutions for various performance, area and power tradeoffs (e.g. concentrated mesh, flattened butterfly, hierarchical star). Furthermore, there are also works which help the designer decide for the appropriate NoC topology from a given topology library for various power/performance tradeoffs. (Radu Marculescu, Keynote Paper).

Apart from 1-D and 2-D topologies, 3D architectures have been proposed as well. They emerged when the integrating ICs in 3-D fashion started becoming popular. It is true that

they solve a number of issues, but they need further exploration. In the work (Soteriou et al : "Polaris: A system-level roadmapping toolchain for on-chip interconnection networks"), it was found that 3-D mesh is the most suitable NoC in many cases.

Finally, the need for irregular and customized topologies is existent in a number of cases, where simple topologies are not applicable. For example, when we are faced with area problems, regular architectures are not the most efficient ones. In addition, for real applications, the communication requirements are not evenly distributed among the components. As a result, designing the network to meet the extreme cases leads to under-utilization of the resources and designing it to meet the average cases causes bottlenecks. There have been suggested various approaches which customize the network topology according to the target application (W.H Ho et al : "A methodology for designing efficient on-chip interconnects on well-behaved communication patterns") and the energy/performance constraints (U Orgas et al: "Energy- and performance driven NoC communication architecture synthesis using a decomposition approach").

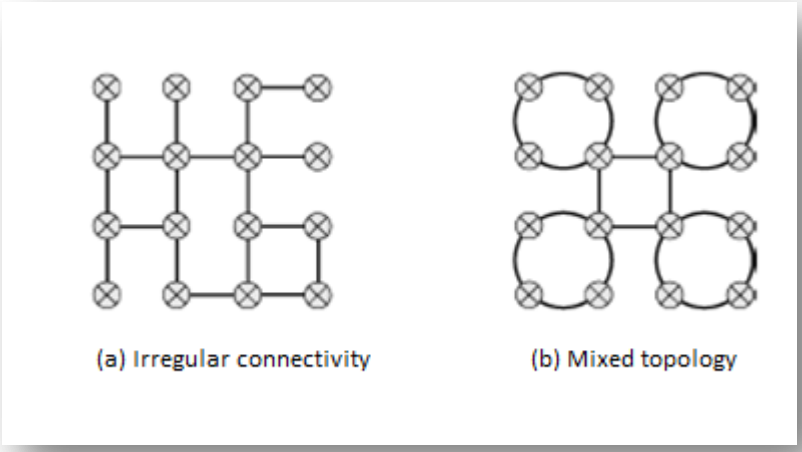

Figure 5 : Irregular Forms of topologies

Generally speaking, the theoretical problem of optimal topology synthesis for a given application does not have a known theoretical solution and it is a challenge on its own right. Apart from that, the customized architectures may need complex floorplanning and uneven wire lengths in order to function efficiently.

1.2.5 Buffering

Buffers are an integral part of any network router. In by far the most NoC architectures, buffers account for the main part of the router area. As such, it is a major concern to minimize the amount of buffering necessary under given performance requirements.

There are two main aspects of buffers (i) their size and (ii) their location within the router. Tamir and Frazier (1988) have provided a comprehensive overview of advantages and disadvantages of different buffer configurations (size and location) and additionally proposed a buffering strategy called dynamically allocated multi-queue (DAMQ) buffer. In the argument of input vs. output buffers for equal performance the queue length in a system with output port buffering is always found to be shorter than the queue length in an equivalent system with input port buffering. This is so, since in a routing node with input buffers, a packet is blocked if it is queued behind a packet whose output port is busy(head-of-the-line blocking). With regards to centralized buffer pools shared between multiple input and output ports vs distributed dedicated FIFOs, the centralized buffer implementations are found to be expensive in area due to overhead in control implementation and become bottlenecks during periods of congestion. The DAMQ buffering scheme allows independent access to the packets destined for each output port, while applying its free space to any incoming packet. DAMQ shows better performance than FIFO or statically-allocated shared buffer space per input-output port due to better utilization of the available buffer space especially for non-uniform traffic.

In Rijpkema et al.(2001), a somewhat similar concept called virtual output queuing is explored. It combines moderate cost with high performance at the output queues. Here independent queues are designated to the output channels, thus enhancing the link utilization by bypassing blocked packets.

In Hu and Marculescu (2004a), the authors present an algorithm which sizes the input buffers in a mesh-type NoC on the basis of the traffic characteristics of a given application. For three audio/video benchmarks, it was shown how such intelligent buffer allocation resulted in about 85% savings in buffering resources in comparison to uniform buffer sizes without any reduction in performance.

As a conclusion, buffer sizing and structure(location) should be thoroughly investigated if we want to avoid misusing the chip resources.

1.3 Power and Thermal Management

Due to concerns on battery lifetime, cooling and thermal budgets, power issues are at the forefront of NoC design. Indeed, several NoC prototypes show NoCs taking a substantial portion of system power, e.g. ~40% in the MIT "RAW" chip and ~30% in the Intel 80-core teraflop chip. The aim of the designer is to minimize or constraint the metrics of interests such as (peak power consumption, energy consumption and average or peak temperature). It is an optimization problem under various constraints.

There have been proposed various approaches referring to power management issue. There has been research into run-time NoC power management using DVS on links, as well as shutting links down based on their actual utilization. Globally asynchronous locally synchronous (GALS) approaches to dynamic voltage and frequency scaling further leverage the existing boundaries between various clocking domains.

Every designer needs accurate and application-aware energy models. Ideally such models should target both dynamic and static power dissipations.

1.4 Application Modeling

Traffic models refer to the mathematical characterization of workloads generated by various classes of applications. With network performance being highly dependent on the actual traffic, it is obvious that accurate traffic models are needed for a thorough understanding of the huge design space of network topologies, protocols, and implementations. Since implementing real applications is time consuming and lacks flexibility, analytical models can be used instead to evaluate the network performance early in the design process.

We are in need of stochastic traffic models and statistical parameters that describe the asymptotic properties of the network accurately and facilitate analysis.

For example, starting from real multimedia traces, one can build an analytical model that captures the long-range dependencies and then using the results, various performance and cost metrics such as packet loss probability and buffer size can be optimized.

Unfortunately, the research in this area is still lagging due to the latch of well-defined NoC benchmarks. This situation has two primary reasons. First, the applications suitable for NoC platforms are typically very complex. For instance, it is common for applications to be partitioned among tens of processes. As a result it is unclear it the benchmarks stress the NoCs effectively. Second, this research requires detailed information about the dynamic behavior of the system; this is hard to obtain even using simulation or prototyping.

Chapter 2:

Customized NoC architectures

Abstract

In this chapter, customized Network-on-Chip architectures are examined. We present existing solutions, which implement configurable architectures as well as complete design flows. Finally, we describe the proposed solution.

2.1 Introduction

As we discussed above, the designers of a NoC have to take into consideration various parameters and tradeoffs regarding the NoC features, in order to have the desirable performance. More particularly, factors like latency, throughput, power dissipation, and finally reliability are affected by the various design choices and sometimes the improvement of one ends in the deterioration of the other.

Moreover, each NoC architecture is applied on a different platform and has to adhere to a different set of constraints. The chip resources, the mapping and traffic profile of the target application, the power and thermal specifications make the design of each chip communication system a unique optimization problem.

Furthermore, the complexity of the modern chips and applications along with the need for short development times are additional reasons for the existence of parameterized and reusable vhdl code. In case of reusable code, the design is portable and easily adaptable to any alterations. As a result the designer can easily apply the changes in the value of some parameters and take many different network architectures. We are going to talk about reusable code techniques in Chapter 3.

Apart from that, when it comes to the verification of a NoC structure, the designer has to try a range of possible solutions with different parameters (topology, buffering size etc) and produce a variety of test benches, which are going to simulate the traffic flow of the target application and stress the network. This task is time-consuming and has to become automatically in order to save time and money.

To conclude, it is crucial to develop the basic platforms and tools and invest initially to build the work environment, so that we can later on generate and test various NoC models without having to manually produce different architectures and test benches.

In this Chapter, we present some samples of related research conducted both by universities and industry.

2.2 Related Work

There have been published various works, which focus on the reusability and parameterization of VHDL code of the router architecture.

These works are based on the development of generic platforms, which can support alterations in parameters such as buffer size, the channel width, the sizes of the fields of the flits, the maximum number of retransmission of a flit, the number of ports etc.

Their aim is to reach some conclusions about the set of parameters, which lead to better performance results.

We will present two of these works, which describe generic NoC platforms.

2.2.1 Spidergon NoC

In "Generic and Extensible Spidergon NoC" (Abdelkrim Zitouni et al), the writers present a GALS and generic NoC architecture based on a configurable router. This router integrates a sophisticated dynamic arbiter, the wormhole routing technique and can be configured in a manner that allows it to be used in many possible NoC topologies such as Mesh 2-D, Tree and Polygon architectures.

The proposed Spidergon NoC architecture is constructed based on an elementary polygon network which is a combination of the star and the ring architectures.

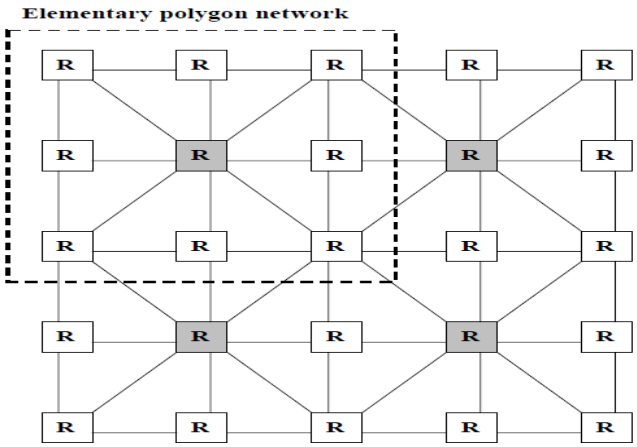

Figure 6:Example of a Spidergon architecture of valence m = 8

This elementary network is formed by 4R+1 (R=1,2 etc.) routers including a central router that is connected with the 4R peripherals routers via point to point links. The peripheral routers are connected to each other in the form of a ring. The elementary network is characterized by its valence (m = 4R) that represents the number of the peripheral routers. These routers necessitate 2m links to be connected to the central router. Each peripheral router is connected to 4 input/output ports and the central router is connected to m+1 input/output ports. A Spidergon architecture with valence m is constituted by 3m+1 routers that necessitate 7m point to point links to be connected with each other.

Furthermore, the sizes and depths of the FIFO contained in this router, the number of input/output ports, the size of the fields of a flit, the number and the time of retransmission and the maximum numbers of the requests sent to the arbiter are also generic. Moreover, it is generic in terms of supported number of cores. All these characteristics make the proposed NoC flexible and extensible according to the applicative aspect and thus improve the quality of service required by the application to be mapped on it.

The development of this network is based on a library of generic models of VHDL blocks. The files of this library contain protocol (number of retransmissions, allowed requests, time out, degrees of adaptability and size of each field forming the various types of flits) and physic (width and depth of the FIFO, number of input/output of the routers and the valence m of the network) parameters. These files also contain all the function used by the VHDL blocks like the path calculation function, the CRC checking function, etc. The generation of the Spidergon architecture is done automatically by indicating the valence m in the package file by using the VHDL GENERATE clause.

The portion of the VHDL code following shows how to generate the peripheral routers in an elementary Polygon network of valence m:

```
Generate : For I in 1 to m generate
Perif_Router : Router generic map(width,i)
                      Port map (R0=> Request_in(i),
                       Data_in0 => Input(i+1)*width-1 downto i*width),
                       ……..
                       …….
End generate ;
```

The performances of the proposed NoC were studied and compared with two other NoC with similar architectures (Mesh and Torus). A parametrised network model was constructed using HASE (Hierarchical Architectural Simulation Environment).

Figure 7 shows the evolution of average latency according to the load for two sizes of the Spidergon architecture. The Spidergon architecture of valence m =12 contains 37 routers and the Spidergon architecture of valence m = 20 contains 61 routers. It can be seein that

the latency increases with the size of the network. A larger network emits more packets. It proposes also more buffers for stoking these packets in the event of conflicts.

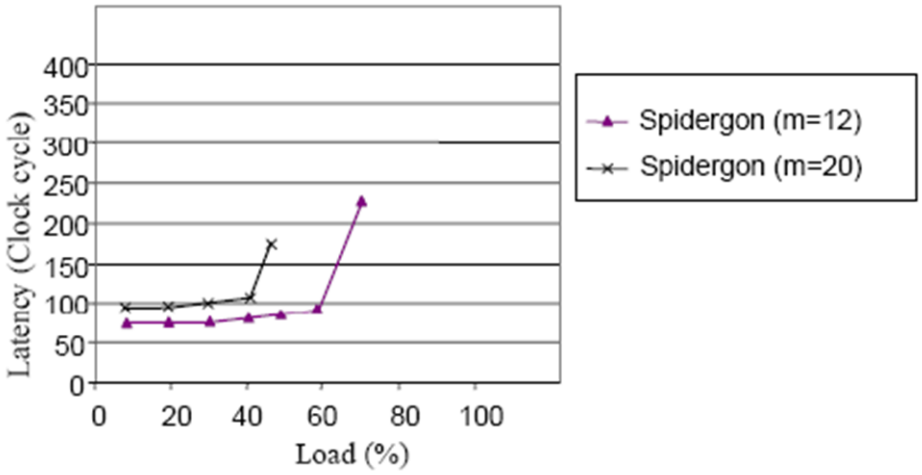

Figure 7 : Latency versus load for two Spidergon architectures

Figure 8 shows the evolution of average latency according to the load for architecture Spidergon of valence m = 12(37 routers) and two other similar architectures Mesh and Torus with 32 routers. The Spidergon architecture is characterized by a lower latency than the two other architectures. The difference is increasingly large after saturation. Also, the network Spidergon saturates later than the two other architectures. Indeed, the packets cross less routers number in the Spidergon network than in the network Mesh 2D and Torus.

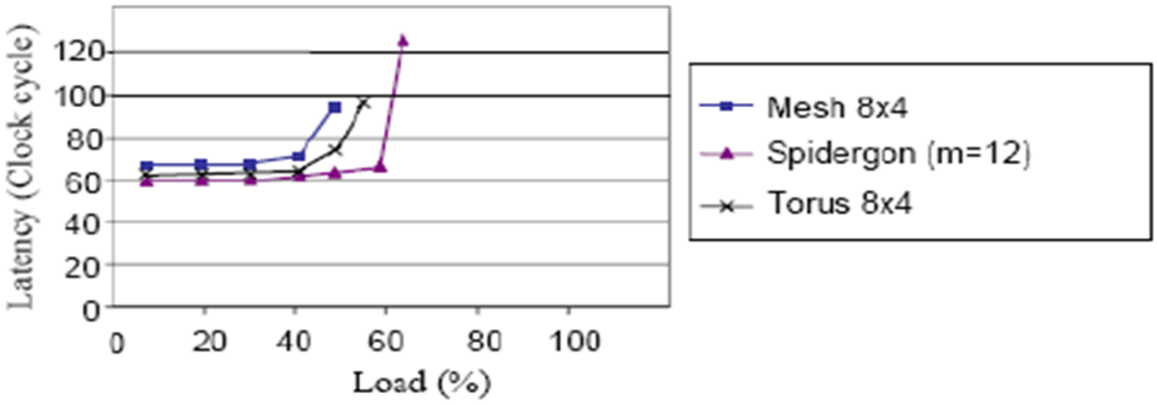

Figure 8 : Latency versus load for three architectures (Spidergon, Mesh and Torus)

Figure 9 shows the evolution on the area of the networks Mesh, Torus and Spidergon according to the number of routers in technology CMOS 0.35 um for buffers of 6 words.

The larger the network is, the greater are the differences between areas of the three networks. This is due to the central routers of the Spidergon network which have m+1 buffers, whereas in Torus architecture all the routers have 5 buffers and in the Mesh network the peripheral routers have only 3 or 4 buffers.

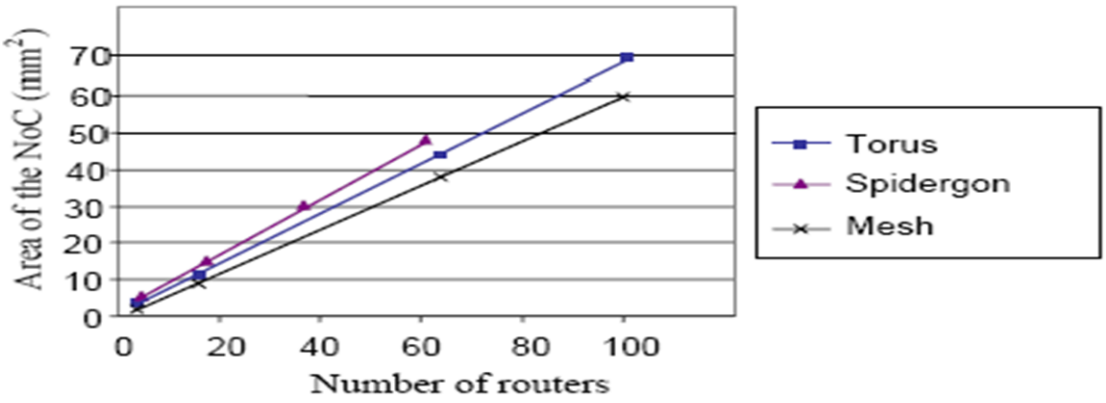

Figure 9 : **Area of NoC versus number of routers in the case of the Torus, Mesh and Spidergon architectures**

The writers conclude that the value added by the Spidergon architecture resides in its capacity to handle a suitable cost/performance compromise in the field of NoC. Spidergon is characterized by the lower latency and later saturation. The next step is the modeling of the architecture in SystemC language at TLM(Transaction Level Modeling).

2.2.2 RASoC : A Router Soft-Core for Networks-on-Chip

In the paper "RASoC : A Router Soft-Core for Networks-on-Chip" (Cesar Albenes Zeferino et al.), the design of a parametrized router is introduces. RASoC is implemented as a reusable VHDL model which can be configured with different sizes and allows the tuning of the NoC parameters in order to meet the requirements of the target application.

The paper gives a thorough description of the router structure. More specifically, the router has 5 ports maximum (North, East, South, West and Local). Depending on the position of a RASoC instance on the NoC and on the network topology, one or two of them need not be implemented, reducing the network area. RASoC ports include two unidirectional opposite channels.

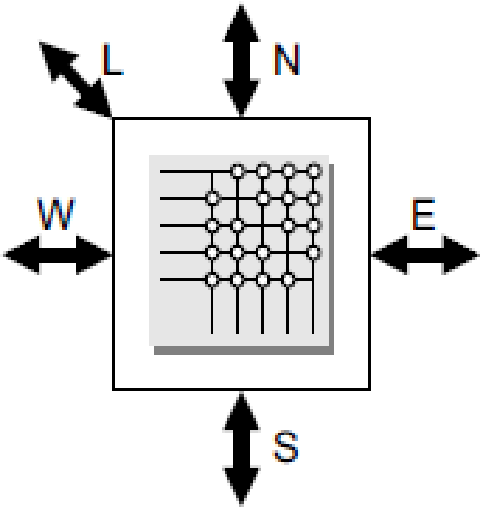

Figure 10 : **The interface of RASoC**

RASoC is implemented in a distributed way and it is composed by instances of two kinds of modules : input channel (in) and output channel (out).

The top level entity, named RASoC has three generic parameters, *n, m and p* which define the data channel width, the width of the routing information in the header and the FIFO depth, respectively. By tuning such parameters, one can synthesize routers with different cost and performance ratios. The lower-level entities receive from the higher-level entities the parameters they need to generate their architectures with the required dimensions. The acronyms in the names of the bottom level entities represent the actual name of each entity (e.g IFC is implemented by the input_flow_controller unit).

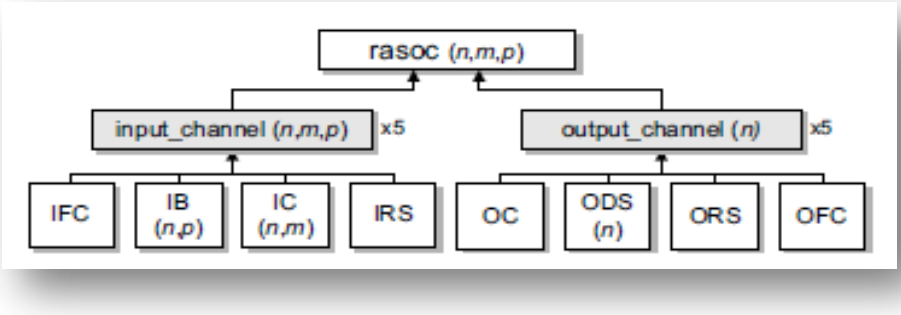

Figure 11 : **Hierarchy of entities in the model**

The RASoC model was synthesized in an FPGA of the family Altera FLEX 10KE. During the experiments, various combinations of parameters were tested and the costs in area for

the buffers and the entire architecture as well as the costs of bottom-level entities were measured.

In the tables below FF-based and EAB-based stand for the two different FIFO techniques. We are shown the number of logic cells (LC), flip-flops (Reg) and memory bits (Mem) consumed in each approach for n=8,16 and 32 bits and for p =2 and 4 flits. Each position in the buffer is n +2 bits wide.

		2 flits			4 flits		
		LC	**Reg**	**Mem**	**LC**	**Reg**	**Mem**
FF-based	**8-bit**	35	22	0	76	43	0
	16-bit	59	38	0	124	75	0
	32-bit	107	70	0	220	139	0
EAB-based	**8-bit**	13	5	20	19	8	40
	16-bit	13	5	36	19	8	72
	32-bit	13	5	68	19	8	136

Table 1 : **Costs of buffers**

		2 flits			4 flits		
		LC	**Reg**	**Mem**	**LC**	**Reg**	**Mem**
FF-based	**8-bit**	570	160	0	795	265	0
	16-bit	770	240	0	1115	425	0
	32-bit	1173	400	0	1830	745	0
EAB-based	**8-bit**	460	75	100	486	90	200
	16-bit	540	75	180	566	90	360
	32-bit	700	75	340	726	90	680

Table 2 : **Costs of RASoC**

Entities	LC	Reg	Mem
IRS - Input Read Switch	1%	0%	0%
IC - Input Controller	8%	0%	0%
IB - Input Buffer	12%	44%	100%
IFC - Input Flow Controller	1%	0%	0%
OFC - Ouptu Flow Controller	0%	0%	0%
ORC - Output Rosk Switch	1%	0%	0%
ODS - Outpu Data Switch	49%	0%	0%
OC - Output Controller	28%	56%	0%

Table 3 : **Costs of bottom-level entities**

As a conclusion, RASoC allows for the automatic building of instances with different sizes. The router has been used to enable testing of desing methodologies.

2.2.3 Æthereal Network on Chip

The Æthereal NoC was introduced by Researchers of the Philips Research Laboratories. The tenet of this NoC is that guaranteed services (GSs) (such as uncorrupted, lossless, ordered data delivery; guaranteed throughput and bounded latency) are essential for the efficient construction of robust SoCs. One reason is that many IPs have inherent performance requirements, such as minimum throughput (for real-time streaming data) or bounded latency (for interrupts).

GSs require resource reservations for the worst case. To exploit the NoC capacity unused by GS traffic, the Æthereal NoC also provides best-effort services (BESs). Furthermore the Æthereal NoC uses contention-free routing or pipelined time-division multiplexed circuit switching to avoid buffer overflow or dropping of data.

In the architecture proposed, a router with N inputs and N outputs uses a slot table to avoid contention on a link, divide up bandwidth per link between connections and switch data to the correct output. Every slot table T has S time slots (rows) and N router outputs (columns). There is a logical notion of synchronicity: All routers in the network occupy the same fixed-duration slot. In a slot s, a network node (that is a router or network interface) can read and write at most one block of data per input and output ports, respectively. In the next slot, (s+2) modulo S, the network node writes the read blocks to the appropriate output ports. Blocks thus propagate in a store-and-forward fashion and cannot deadlock. The

latency that a block incurs per router equals the duration of a slot and the slot reservations guarantee bandwidth in multiples of block size per S slots.

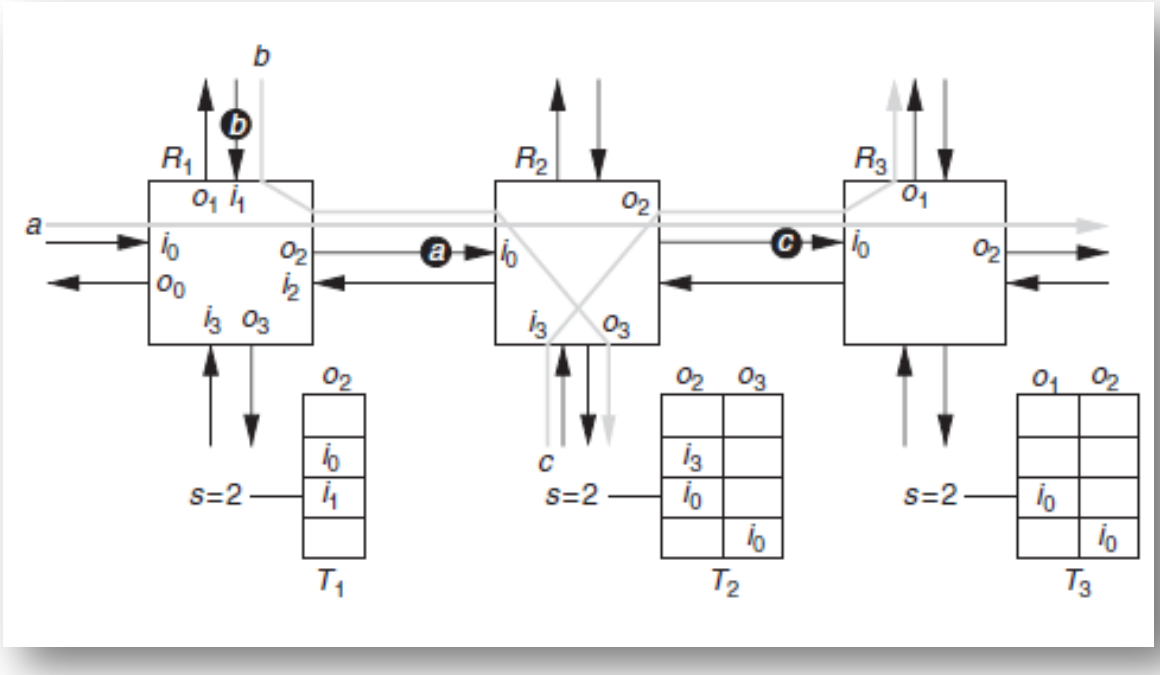

Figure 12 :Contention-free routing:
Network of three routers at slot s =2 with corresponding slot tables

The best-effort routing is a conventional wormhole-routing, input-queued router. Round-robin arbitration of the switch occurs at the granularity of three words (a flit, or flow-control unit). The capacity of the input queues is a router parameter. Bes packets use source routing. The packet header contains the path from source to destination. Each router removes as many bits from the path as necessary to determine to which output the packet must go. Because of the absence of multiple buffer classes, BES packets can deadlock. We avoid deadlock with appropriate routing strategies.

To conclude, the Ethereal NoC developed by Philips aims at achieving composability and predictability in system design and eliminating uncertainties in interconnects, by providing guaranteed throughput and latency services. It also provides run-time reconfiguration.

2.3 Proposed Solution

As we have seen above, there has been conducted extended research and experimentation regarding irregular NoC architectures. Models have been proposed which improve various aspects of performance and tools have been introduced which enable NoC development. However, there is still need for new tools and new approaches which may contribute to our understanding of NoC architectures and introduce innovative ideas.

Furthermore, the target set by the "Microprocessors Laboratory" of the Electrical and Computer Engineering School at the National Technical University of Athens is the development of a suite of tools for the support of the NoC design. The tool chain already includes some high-level platforms, which are responsible for the simulation of the NoC function, the topology optimization, the traffic profiling etc. Furthermore, a router architecture has been developed by Konstantinos Tatas, a former member of the lab.

The aim of the current master thesis is the development of an XML-based tool for the automatic generation of customized NoC architectures. The key feature of the tool is its reliability and efficiency, which help the NoC designer avoid errors and shorten the programming time.

Our first step is to extend and modify the router module architecture developed initially by Konstantinos Tatas, so that we have a reusable and generic model.

The next step is the production of a Network on Chip architecture, which has as components the aforementioned routers. The architecture is going to be generic as far as the topology, the port number and the switch depth of each router, the routing algorithm/table and the arbitration technique are concerned. These parameters are going to be integrated either in a package file or in the top-entity.

Finally, a tool, which will enable the automatic generation of the VHDL code according to XML input, is going to be developed.

Chapter 3:

General-Purpose platform

Abstract

In this chapter we explain the reusability concept, which characterizes our architecture. Furthermore, we give the structural description of the router architecture, analyzing the function of every single component.

3.1 Introduction

The first step for the development of our tool has to do with the VHDL code describing the router. This code exists already but is not in the desirable and appropriate form. It is not easily configurable, since it describes the architecture of a router with fixed port number, fixed buffer size and the only option for routing is the xy routing. The processes have been implemented for the above sizes and the dimensions of the vectors are fixed as well. As a result the code has to become reusable so that we can change the value of various parameters effectively and in a "centralized" way.

The second step is the generation of a network of routers, which will be generic in the number and topology of routers. Furthermore, we will enrich the code, adding a test bench for verification of the NoC function.

The final code will conform to the principles of reusability and thus will be easy and fast to apply changes.

3.2 Reusability

The term reusability refers to the property that a segment of source code has and which allows it to be used again with minor modifications in case we want to add new features and functionalities.

It is a characteristic, which is often adopted in industry since it reduces development times, eliminates bugs, enables the easier understanding of code and makes changes easier to apply. As far as reusability of VHDL code is concerned, there has been extended study and books about this topic.

The books "Circuit Design with VHDL" (by Volnei Pedroni) and "Reuse Methodology Manual for System on Chip Designs" by (Keating, Bricaud) gave us all the necessary information regarding reusability techniques and characteristics. They describe VHDL structures and examples, in a way which supports reusability.

3.3 Router architecture

The VHDL code for the router architecture is structural. More particularly, it consists of a top-entity (noc_switch) and its components, namely the receiver, the transmitter, the buffer, the prioritizer and the switch matrix. In Figure 13 we can see the architecture of the router.

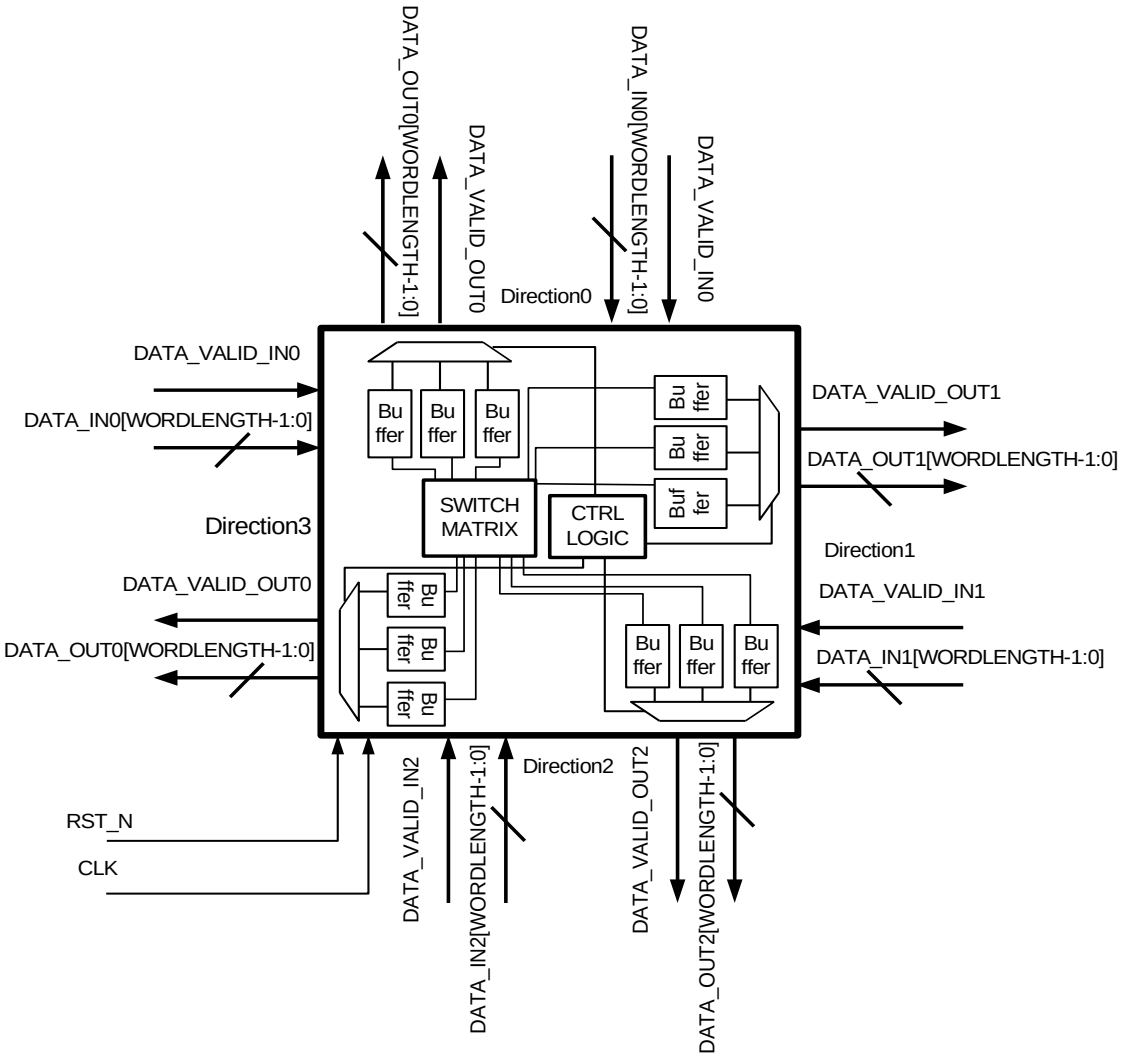

Figure 13 : Structural router architecture

We are going to briefly describe the function of each of these components and how they communicate with each other.

3.3.1 Receiver

The receiver is responsible for polling the respective data valid signal to detect incoming data from each port, selecting the output port for each packet according to its destination, and then storing it to the appropriate buffer. It is composed of a state machine and the output selection logic.

The receiver state machine has the following states:

- IDLE: The state machine is constantly polling the corresponding data valid signal. When the signal is asserted the state machine goes to the dir_check state. Idle is the default state (after reset, and for self-correction) of the state machine.

- DIR_SEL: In this state, the state machine enables the destination port select logic and goes to the writing state.

- RX: The state machine enables the write (RX) side of the Buffer FIFO increments the corresponding Buffer FIFO counter, writing input packet to the appropriate buffer. When the counter has been incremented PACKET_SIZE/WORDLENGTH times, the whole packet has been stored, and the state machine returns to the idle state.

The RX control logic diagram is shown in Fig, while a RX timing diagram is shown in Figure 14.

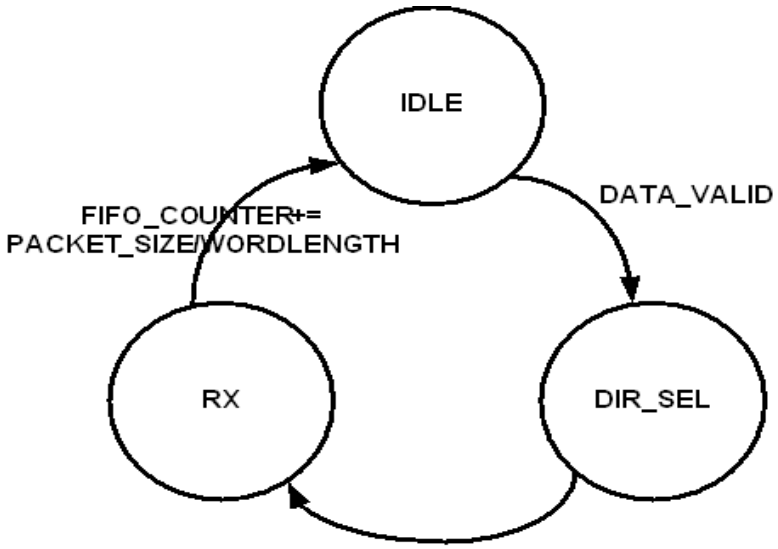

Figure 14: **Receiver Control Logic State Diagram**

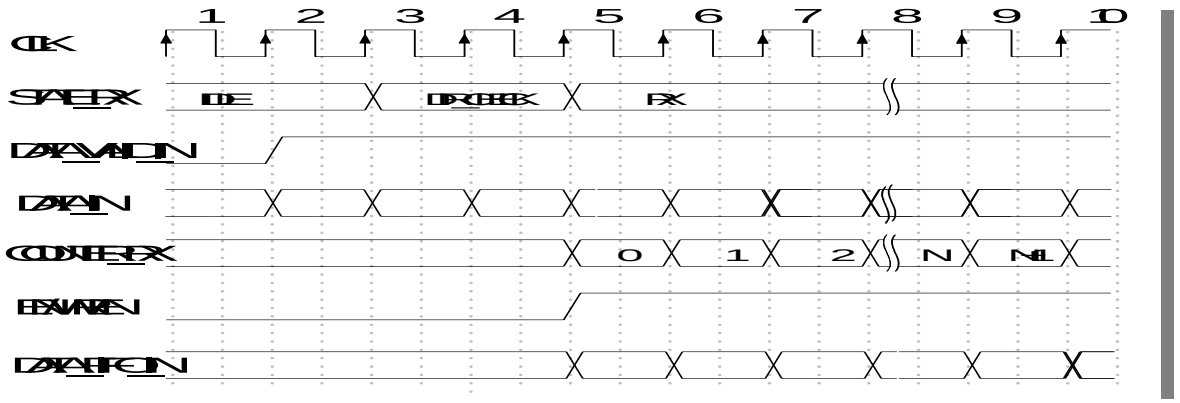

Figure 15: **Receiver timing diagram**

3.3.2 Output Port Selector

The output port selection logic is responsible for reading the destination of the incoming packet and, taking into account the switch address, forward it to the appropriate buffer by setting the switch matrix decode signal appropriately. It can be implemented either by using a ROM LUT to store the decode signals of the output port depending on the destination. The destination if the pair (x,y), which describes the location of the destination address.

The default routing algorithm is the XY but later on we introduced the additional feature of the user-defined routing table.
The routing table is a matrix whose number of elements is equal to the number of the routers on the network. For example a 2x2 NoC has a 4 element matrix. Each element of this matrix is a submatrix with another 4 elements, each of which determines the output port of the data, in case that the destination is the correspondent switch.
For example for the NoC in Figure 20, if the data has entered switch 2 and it has destination the switch 0, it will use the output port 0.

```
CONSTANT lut_array : rom_lut_type_array :=(
((4,1),(1,1)),((3,2),(4,2)), ((0,4),(1,1)), ((4,3),(0,4)) );
```

3.3.3 Buffer

The buffer is a FIFO where a maximum number of words can be stored. This size is defined as switch_depth in the code. The location of the buffer is in the output of each port.

The buffer will be implemented as a dual-port RAM, and for FPGA rapid prototyping, an appropriate number of embedded BRAMS will be used. The FIFO counters and enable signals in the write (input) side are controlled by the Buffer Control Logic, while the corresponding signals of the read (output) side are controlled by the Arbitration Logic.

3.3.4 Transmitter

The transmitter is responsible for the transmission of the packets. It communicates with the receiver (in order to be notified for the arrival of a new packet and update the fifo counter), the prioritizer (in order to get the permission for the transmission when more than one packets want to use an output port), the buffer(in order to send the enable signals for the reading of the data).

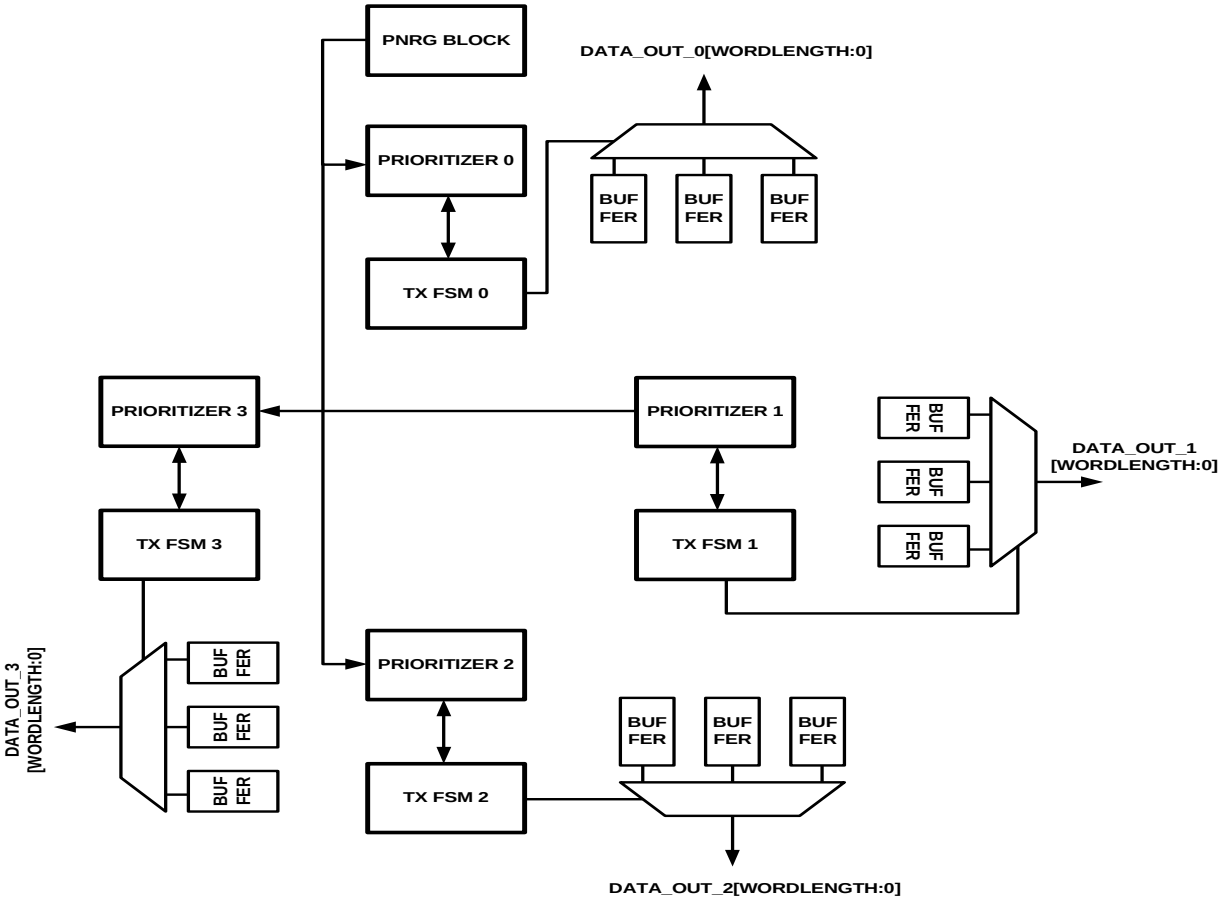

Figure 16: TX Logic block diagram

The transmitter state machine has the following states:

- IDLE: The state machine is constantly polling the FIFO buffer counters. When the write counter value is greater than the read counter value, a packet is present in the buffer, waiting to be sent. When such a condition is detected the state machine goes to the PR_CECK. IDLE is the default state (after reset, and for self-correction) of the state machine.

- PR_CHECK: In this state, the state machine checks if there are other packets pending in the same destination port and resolves their priority according to the existing priority scheme. If the output is granted, it goes to the TX state, otherwise it returns to the idle state.

- TX: The state machine enables the read (TX) side of the buffer, increments the corresponding read Buffer FIFO counter, reading the input packet out of the corresponding buffer, while enabling the appropriate 4-to-1 MUX and asserting the corresponding DATA_VALID_OUT signal, in order to send the packet out of the selected port. When the counter has been incremented PACKET_SIZE/WORDLENGTH times, the packet transmission is complete, and the state machine returns to the idle state, after disserting the appropriate signals.

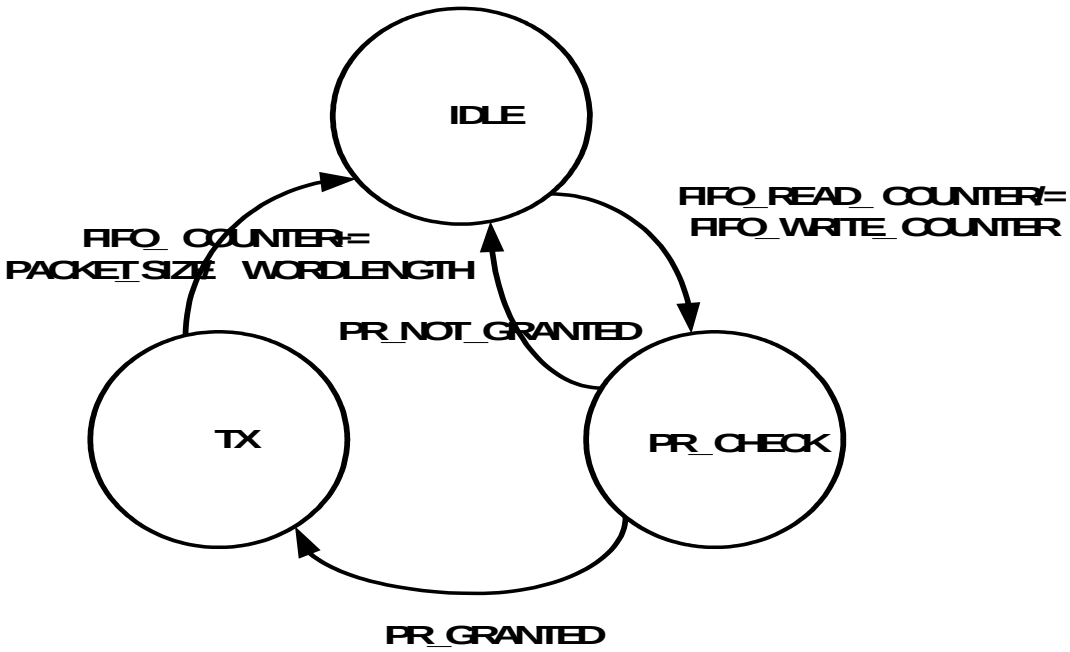

Figure 17: **Transmitter Control Logic State Diagram**

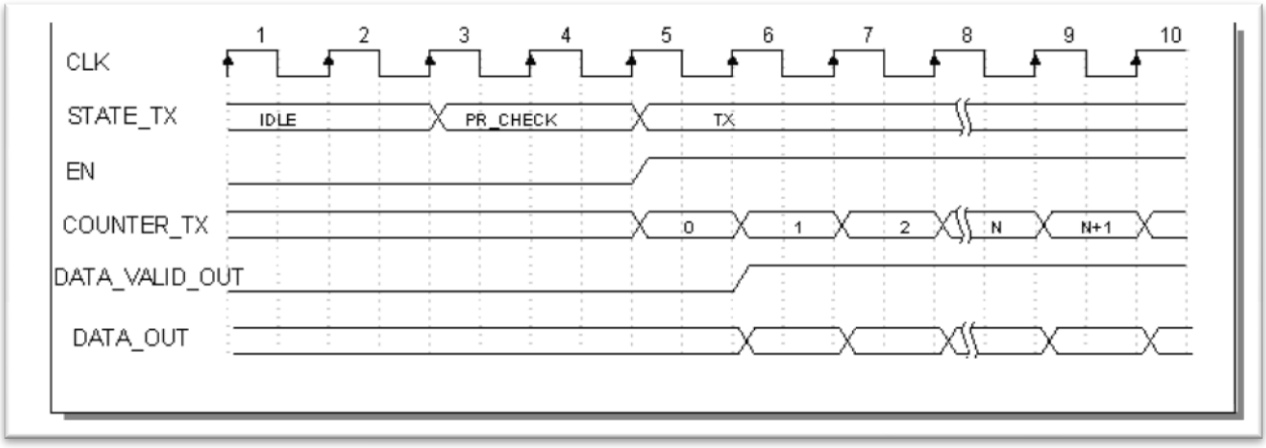

Figure 18: **Transmitter timing diagram**

3.3.5 Prioritizer

The prioritizer is the entity which accounts for the arbitration policy. More particularly, it is responsible for scheduling in the case of packets requesting to exit to the same direction(output port). Multicasting is not supported. Each packet can only exit towards a single direction, except its own (cannot return).

There are two different arbitration policies implemented in the prioritizer and it is up to the user which one will be selected.

- *Fixed priority*: Ideally, the fixed priority should be programmable, using registers. A default output port priority scheme from highest to lowest could be: Port0, Port1, Port2, and Port3. It is easy to implement, but could cause starvation of low priority ports if there is a lot of traffic on high priority ports.

- *Round-robin priority*: The packets are scheduled in a round-robin manner. For fixed-sized packets, this scheduling scheme results in virtually no congestion. It is implemented using a four bit ring counter, producing a single, circularly shifted, enable signal.

3.3.6 Router communication schema

In the following picture, we can see a rough description of the router communication schema.

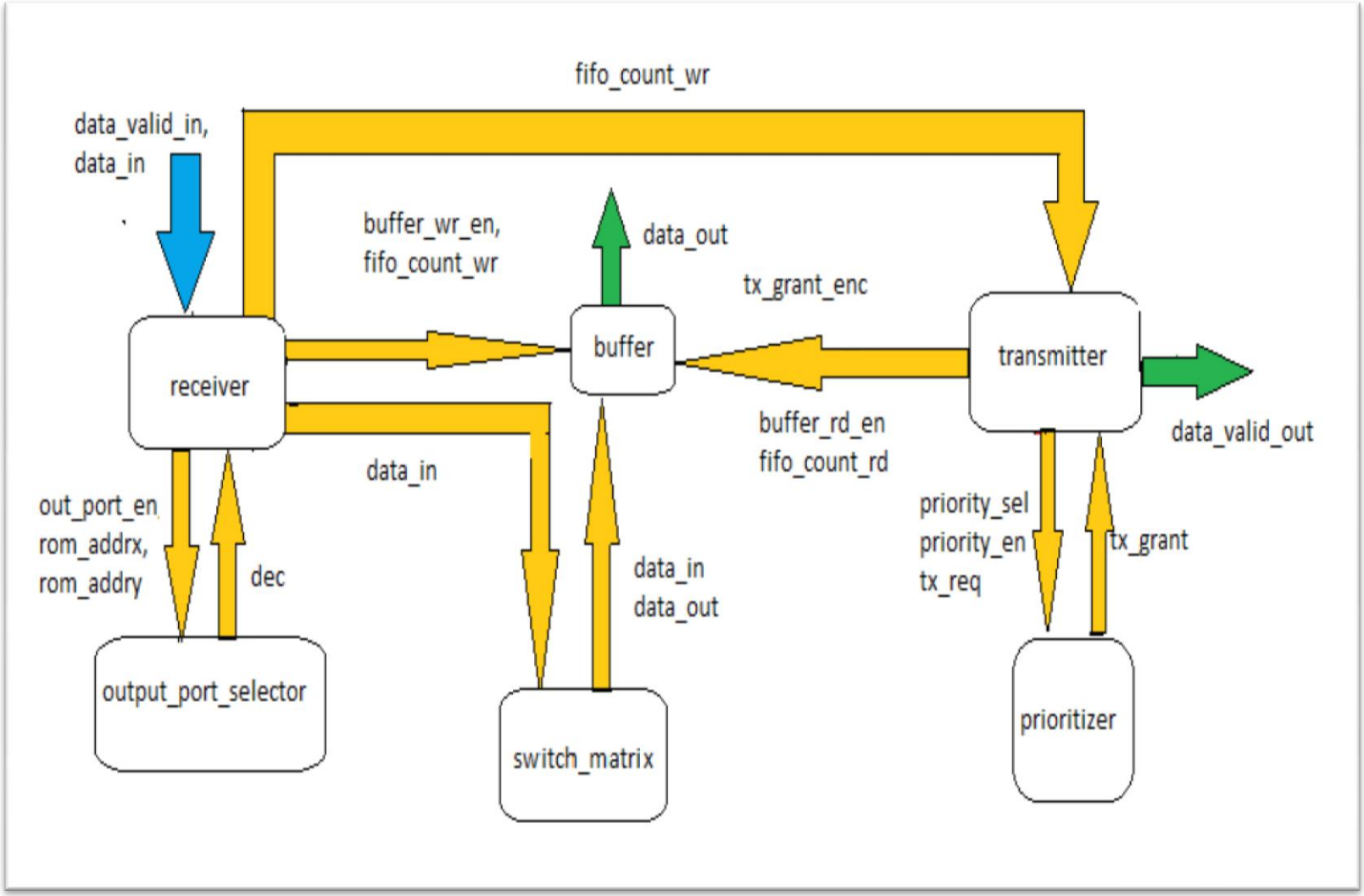

Figure 19: Communication schema of noc_switch

3.4 NoC architecture

The NoC top entity contains one or more noc_switch components and their links(topology). In order to have different switches we had to integrate all the parameters into arrays, which are assigned in a reusable package file. Furthermore, we developed a test bench, which is configurable as well.

3.4.1 Top-entity

Underneath, we present the code of the top entity noc. The signal assignments which define the links between the switches, as well as the traffic refer only to this test case and are fully configurable.

3.4.2 Package

The package includes the definitions of all configurable and standard data and data structures. It also contains the function LUT, which has two possible forms, one of which is displayed here.

3.4.3 Testbench

The testbench is a fully configurable part of the code. It includes all the signal assignments, which are responsible for the traffic in the network. We are actually simulating the traffic that the IP cores would produce if they were integrated in the design. As a result we have the chance to quickly and reliably test the network behavior.

Although there are many sophisticated tools, which offer high-level simulation (e.g Noxim), the existence of a testbench is a necessary step, in order to verify the functionality of the code.

A testbench can also be useful in case that new features will be added in the future.

The code displayed underneath is the testbench of a specific case. The testbench is a configurable part of the code and is written according to the noc specifications and the application traffic desirable.

3.4 Testcase

We are going generate a NoC like the one of Figure 20 .The links are the ones displayed in the picture and the XY routing algorithm is deployed for the servicing of the transmission requests.

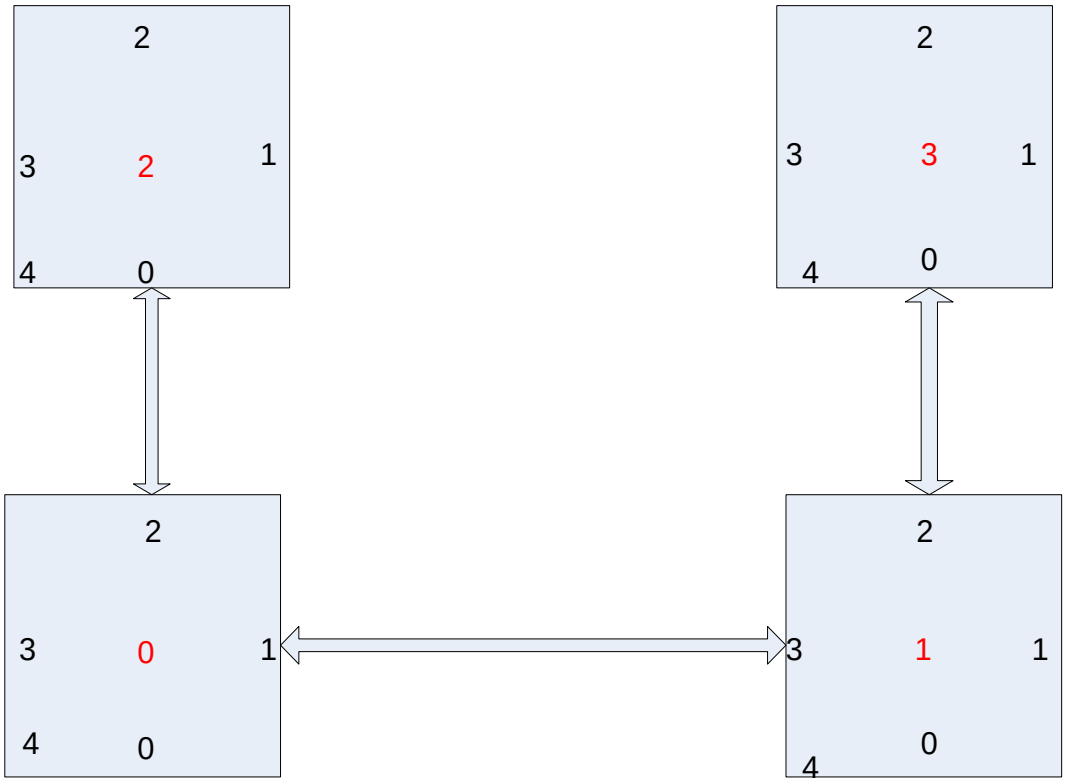

Figure 20 : 2x2 NoC

The function for the implementation of the XY routing is is the following and is located in the noc_pkg.vhd file :

```
function LUT (X_S, Y_S, p : integer) return rom_lut_type is --the port
is not taken into consideration in the implementation of the function
      variable x: integer range 0 to pe_num_x-1; --switch x address
      variable y: integer range 0 to pe_num_y-1; --switch y address
      variable port_num: integer range 0 to max_ports-1; --port number
      variable rom_lut: rom_lut_type;
begin
      port_num := p;
```

```
	for i in 0 to pe_num_x-1 loop --for every pe in the x dimension
		x := i;
		for j in 0 to pe_num_y-1 loop --for every pe in the y
	dimension
			y := j;
			if X_S > i then
				if port_num = 3 then --bouncing back packet
					rom_lut(x, y) := 0; --exit to the y
				direction
				else
					rom_lut(x,y) := 3; --exit left port
				end if;
			elsif X_S < i then
				if port_num = 1 then --bouncing back packet
					rom_lut(x,y) := 2; --exit to the y
				direction
				else
					rom_lut(x,y) := 1; --exit right port
				end if;
			else
				if Y_S > j then
					if port_num = 0 then --bouncingback packet
						rom_lut(x,y) := 1; --exit to the x
						direction
					else
						rom_lut(x,y) := 0; --exit upwards
					port
					end if;
				elsif Y_S < j then
					if port_num = 2 then --bouncingback packet
						rom_lut(x,y) := 3; --exit to the y
					direction
					else
						rom_lut(x,y) := 2; --exit downwards
					port
					end if;
				else
					if port_num = 4 then --bouncingback packet
						rom_lut(x,y) := 0; --exit to the y
					direction
					else
						rom_lut(x,y) := 4; --exit to pe
					end if;
				end if;
			end if;
		end loop;
	end loop;
	return rom_lut;
end function;
```

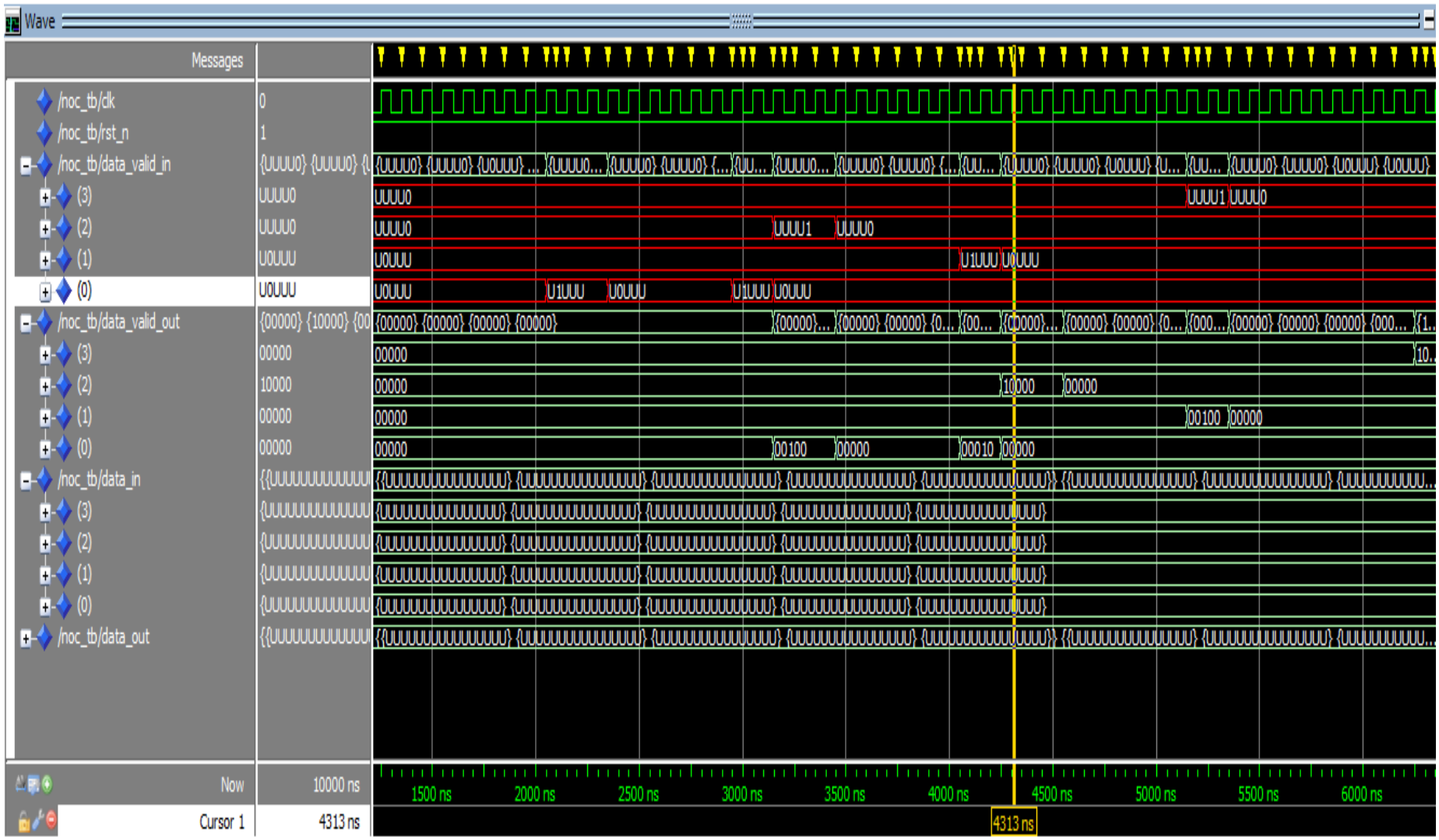

Figure 21: Waveform of the testcase (We send traffic requests from router 0 to router 3 and from router 0 to router 2).

Chapter 4:

Design Flow and Automatic Generation Tool "NoCGen"

Abstract

In this chapter, we describe the design flow for the noc platform generation. Furthermore, we explain in detail the configuration files, which are taken as input by the generation tool "NocGen".

4.1 Introduction

The design flow, which is depicted in Fig 20, includes the steps required in order to build an irregular architecture.

The first phase is the application mapping and the traffic profiling. Next, we need a topology targeted at the application and its communication needs. The topology determines how the cores are going to be distributed on the chip and interconnected with each other. Furthermore, the designer is going to give various directions, as far as the buffer size and number of ports are concerned.

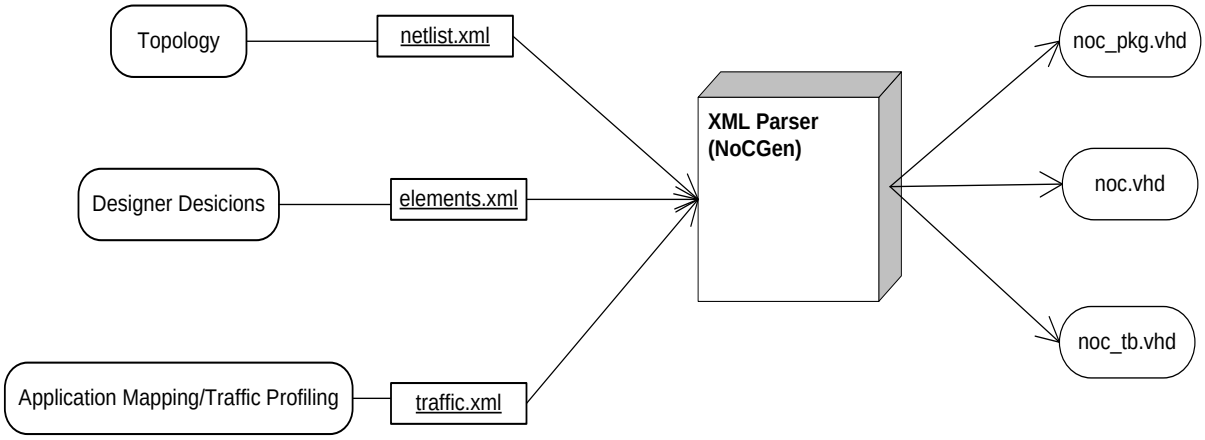

Figure 22 : Design Flow

All the above features are incorporated into three configuration files written in xml.

The first one named "elements.xml" contains information about the buffer sizes and number of output ports of each of the routers. These characteristics are determined by the designer, who has to take into consideration the traffic profile of the application. It is a decision, which will propably be reached after system-level exploration as well and consideration of the feedback coming from the network function.

The file "netlist.xml" contains information about the topology of the design, namely the way the cores are connected with each other. In the current thesis, we used the experimental

results of the work of Vourkas Ioannis. The tool developed during his thesis gives optimized topologies for the execution of various hardware applications.

The file "traffic.xml" contains some samples of the traffic generated during the execution of the above applications.

" NocGen" is an XML-based tool, which generates automatically a Network on Chip with the properties described in the above mentioned configuration files.It parses the aforementioned xml files and writes the vhdl file noc.vhd, noc_tb.vhd, noc_pkg.vhd.

These vhd files have some standard and some reconfigurable parts. The tool writes the reconfigurable part, which has to do with the topology, the traffic and some of the sizes of the router according to the specifications.

4.2 Configuration Files

Underneath, we present the XML schemas and a short example for each of them.

4.2.1 Router characteristics

The file named "elements.xml" contains the (x,y) coordinates of each router, as well as the switch depth and the number of ports. The coordinates are of use for the process of routing and the later process of floorplanning.

The XML schema is the following:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">


 The network is a sequence of the elements "switch"

 <xs:element name="ips">
 <xs:complexType>
       <xs:sequence>
             <xs:element ref="switch" maxOccurs="unbounded" />
       </xs:sequence>
 </xs:complexType>
 </xs:element>




 Each "switch" element consists of the element "switchPorts" and the
 element "switchDepth". It has as attributes the x, y coordinates.
```

```
<xs:element name="switch">
 <xs:complexType>
      <xs:sequence>
       <xs:element name="switchPorts" type ="xs:nonNegativeInteger" />
       <xs:element name="switchDepth" type ="xs:nonNegativeInteger"/>
   </xs:sequence>
   <xs:attribute  name="x"  type="xs:nonNegativeInteger"  use="required"
/>
   <xs:attribute  name="y"  type="xs:nonNegativeInteger"  use="required"
/>
 </xs:complexType>
</xs:element>
```

An xml file which corresponds to this schema is :

```
<ips>
      <switch x = "0" y = "0" >
            <switchPorts>5</switchPorts>
            <switchDepth>3</switchDepth>
      </switch>
      <switch x = "1" y = "0" >
            <switchPorts>5</switchPorts>
            <switchDepth>3</switchDepth>
      </switch>
      <switch x = "0" y = "1" >
            <switchPorts>5</switchPorts>
            <switchDepth>3</switchDepth>
      </switch>
      <switch x = "1" y = "1" >
            <switchPorts>5</switchPorts>
            <switchDepth>3</switchDepth>
      </switch>
</ips>
```

The above file describes a network of four routers, which all have 5 switch ports and a switch depth of 3 words.

4.2.2 Netlist

The file named "netlist.xml" includes the connections/links between the routers. Every port can be connected with any another and the direction of the dataflow can be defined too.

The XML schema is the following

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 The netlist contains a sequence of the elements "link"


 <xs:element name="netlist">
       <xs:complexType>
             <xs:sequence>
                   <xs:element ref="link" maxOccurs="unbounded" />
             </xs:sequence>
       </xs:complexType>
 </xs:element>




 Each "link" has a sourceRouter, a source Port, a destinationRouter and
 a destinationPort. The order is important since it determines the
 direction of the link.

 <xs:element name="link">
       <xs:complexType>
             <xs:sequence>
             <xs:element ref="sourceRouter" />
             <xs:element ref="sourcePort" />
             <xs:element ref="destinationRouter" />
             <xs:element ref="destinationPort" />
             </xs:sequence>
             <xs:attribute name="ID" type="xs:ID" use="required" />
       </xs:complexType>
 </xs:element>


<xs:elementname="destinationPort"type="xs:nonNegativeInteger"/>
<xs:element name="destinationRouter" type="xs:nonNegativeInteger"/>
<xs:element name="sourcePort" type="xs:nonNegativeInteger"/>
<xs:element name="sourceRouter" type="xs:nonNegativeInteger"/>


</xs:schema>
```

An xml document for the above schema is the following:

```
<netlist>

<link ID="link1">
      <sourceRouter> 0 </sourceRouter>
      <sourcePort> 1 </sourcePort>
      <destinationRouter> 1 </destinationRouter>
      <destinationPort> 3 </destinationPort>
</link>

<link ID="link2">
      <sourceRouter> 1 </sourceRouter>
      <sourcePort> 2 </sourcePort>
      <destinationRouter> 3 </destinationRouter>
      <destinationPort> 0 </destinationPort>
</link>
<link ID="link3">
      <sourceRouter> 0 </sourceRouter>
      <sourcePort> 2 </sourcePort>
      <destinationRouter> 2 </destinationRouter>
      <destinationPort> 0 </destinationPort>
</link>

</netlist>
```

The above xml document describes the following topology.

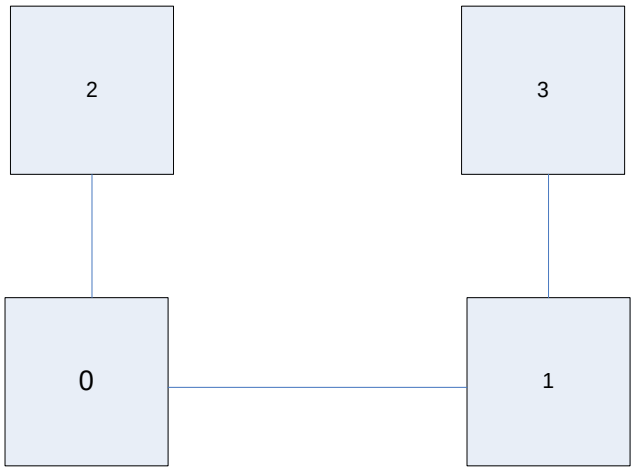

Figure 23 : The topology of the NoC described in the xml document

4.2.3 Traffic

In the file named "traffic.xml" , the data exchange between the cores is described.

Every piece of data, which is produced by a core, proceeds to the router attached to the core and its "aim" is to reach the destination with coordinates (x,y) . This destination is defined by the first digits of the data packet.

For example, in the case that we have a 4*4 NoC (pe_num_x =2, pe_num_y =2) : the first 2 (log2=2) binary digits of the packet will denote the x coordinate and the next 2 (log4=2) will denote the y coordinate. The xml data will include the source of the data packet, the data packet itself (whose header is the destination), the clock period when the transmission begins and the number of clock cycles while it is active.

We have to give the different transmissions, which come from the same IP core, one after the other with ascending time order. For example if we have continuous traffic from a network port but with different destinations, we should express like this:

```
<transmission ID="trans2">
      <inputRouter>2 </inputRouter>
      <inputPort> 4 </inputPort>
      <transBegin>35</transBegin>
      <transDuration>2</transDuration>
      <dataSequence>
            <data>0001001101110100</data>
      </dataSequence>
</transmission>
<transmission ID="trans3">
      <inputRouter>2 </inputRouter>
      <inputPort> 4 </inputPort>
      <transBegin>38</transBegin>
      <transDuration>2</transDuration>
      <dataSequence>
            <data>0101001101110100</data>
            <data>0111001101110100</data>
      </dataSequence>
</transmission>
```

Furthermore, each transmission has to hold at least 2 clock cycles, even if the packet contains only one flit (as in the case of the transmission trans1e above)

The XML schema is the following:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">


The traffic.xml consists of all the transmissions which take place in t
network.

<xs:element name="traffic">
      <xs:complexType>
            <xs:sequence>
                  <xs:element ref="transmission" maxOccurs="unbounded" />
            </xs:sequence>
      </xs:complexType>
</xs:element>



Each transmission is described by the inputPort and inputRouter,t
trasnBegin, the transBegin,the transDuration and the dataSequence.

<xs:element name="transmission">
      <xs:complexType>
            <xs:sequence>
                  <xs:element ref="inputRouter" />
                  <xs:element ref="inputPort" />
                  <xs:element ref="transBegin"/>
                  <xs:element ref="transDuration"/>
                  <xs:element ref="dataSequence"/>
            </xs:sequence>
            <xs:attribute name="ID" type="xs:ID" use="required" />
      </xs:complexType>
</xs:element>



The inputPort and inputRouter denote the input point of the traffic in t
network. Traffic is initially generated by cores attached to port routers

<xs:element name="inputPort" type="xs:nonNegativeInteger"/>
<xs:element name="inputRouter" type="xs:nonNegativeInteger"/>



The dataSequence contains the fields of data to be transmitted and t
startDuration how long it lasted.
<xs:element name="dataSequence">
<xs:element name="transBegin" type="xs:nonNegativeInteger"/>
<xs:element name="transDuration" type="xs:nonNegativeInteger"/>
            <xs:element ref="data" maxOccurs="unbounded" />
      </xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="data" type="xs:string"/>

</xs:schema>
```

An xml document for the above schema is the following:

```
<traffic>
      <transmission ID="trans0">
            <inputRouter> 0 </inputRouter>
            <inputPort> 3 </inputPort>
            <transBegin>20</transBegin>
            <transDuration>3</transDuration>
            <dataSequence>
                  <data>0111001101110100</data>
                  <data>1100000000000000</data>
                  <data>1000000000100000</data>
            </dataSequence>
      </transmission>

      <transmission ID="trans1">
            <inputRouter> 0 </inputRouter>
            <inputPort> 3 </inputPort>
            <transBegin>29</transBegin>
            <transDuration>2</transDuration>
            <dataSequence>
                  <data>1110000000000000</data>
            </dataSequence>
      </transmission>
</traffic>
```

4.3 Technical Details

The tool for the automatic generation of the vhdl code is implemented in C++. In order to parse the XML files, we used the library xerces-c-3.0.1. Xerces is a validating XML parser written in a portable subset of C++. It gives the ability to read and write XML data. A shared library is provided for parsing, generating, manipulating, and validating XML documents using the DOM, SAX, and SAX2 APIs. Xerces-C++ is faithful to the XML 1.0 recommendation and many associated standards. The parser provides high performance, modularity, and scalability.

The code of the tool can be found in Appendix.

Chapter 5:

Output Samples and Measurements

Abstract

In this chapter we study the efficiency of the NoCs-based interconnection architecture regarding four multimedia applications. We take measurements of the power dissipated, the maximum possible frequency, the slices and utilization for each of the above implementations.

5.1 Introduction

We are going to implement the network communication system of four popular hardware applications. The applications are are (a) an MPEG-4 hardware encoder/decoder, (b) a video object plane decoder (VOPD), (c) a multimedia system (MMS) and (d) a Multi-Window Display (MWD).

The number of routers is clearly defined by the application. The topology for every network will be irregular and the number of ports will be 3 or 5.

The traffic trace as well as the position of every router on chip is derived from the simulation results of previous Microlab thesis , whose task was the system-level exploration of NoCs.

The buffer depth of each router will be defined according to the sample traffic needs but we are going to investigate the changes in the overall design performance in case of its increase. It is to be expected, that increase of buffer depth leads to growth of energy on the chip.

Furthermore, we are going to take measurements of the power dissipated, the maximum possible frequency, the slices and utilization for each of the above implementations for different buffer sizes.

More particularly we are going to present the slices/slice registers/4 Input LUTs utilization for the 3 different buffer sizes in diagrams. Furthermore, we are interested in the maximum frequency and total power dissipated for the 3 different buffer sizes.

5.2 Experimental Setup

Product Version	ISE 12.1
Target Device	Virtex 4-XC4VLX15
Temperature	50, 65, 80 (°C)
Number of Ports per router	5
Links	Bidirectional
Buffer size	3, 6, 9
Topology	Mesh, Irregular
Routing algorithm	XY , routing table
Clock frequency	100 MHz

5.1.1 MPEG-4

MPEG-4 is a broadly used protocol for audio and video encoding. A hardware encoder and decoder consist of many components, so a NoC approach is suitable. The tested MPEG-4 includes various processing elements, such as a video unit, an audio unit, a risc processor, a med cpu, a binary alpha block and three SRAMS. The total number of cores needed for the application are 12, with an equal number of routers attached to them.

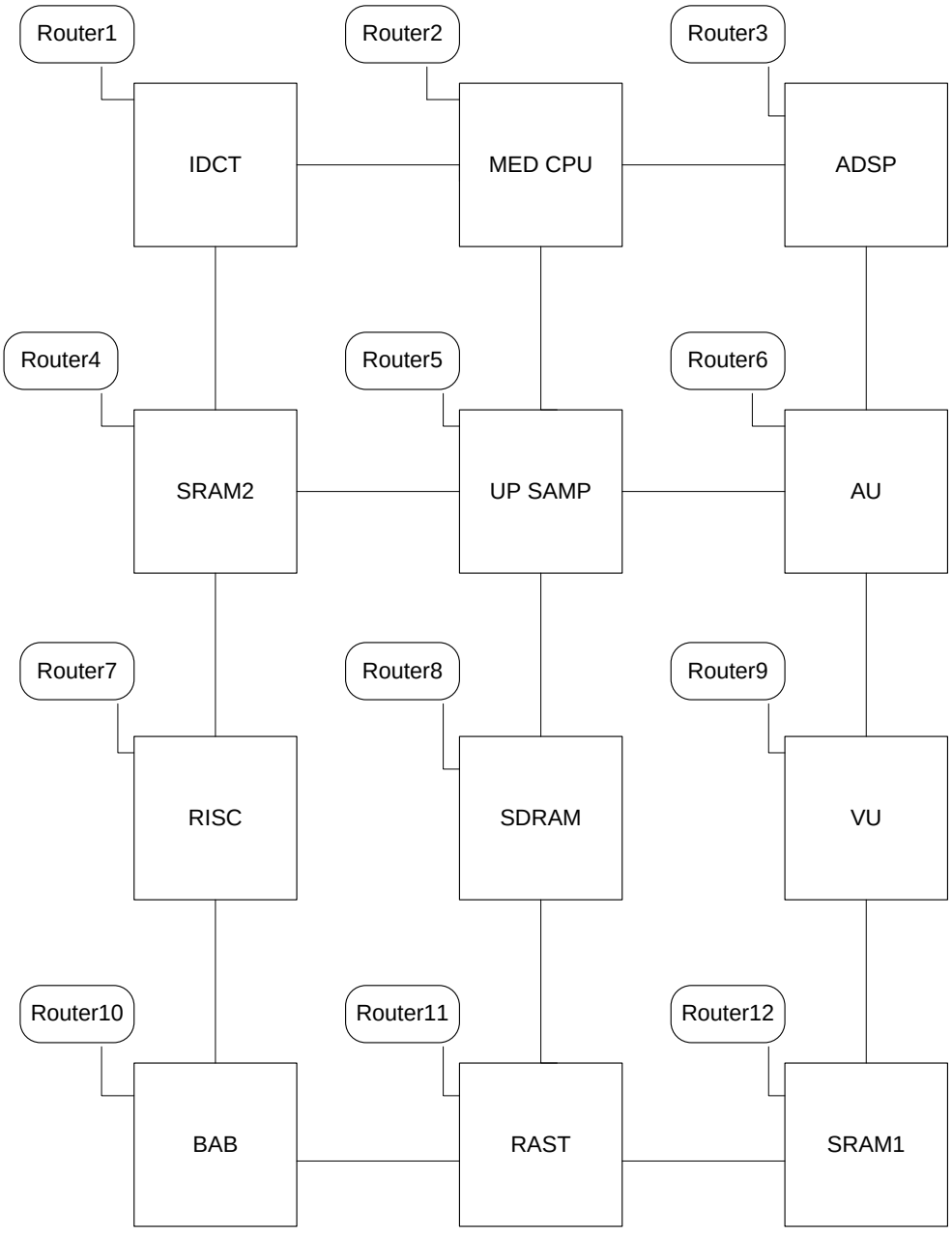

Figure 24: Block diagram of MPEG-4(12 cores)

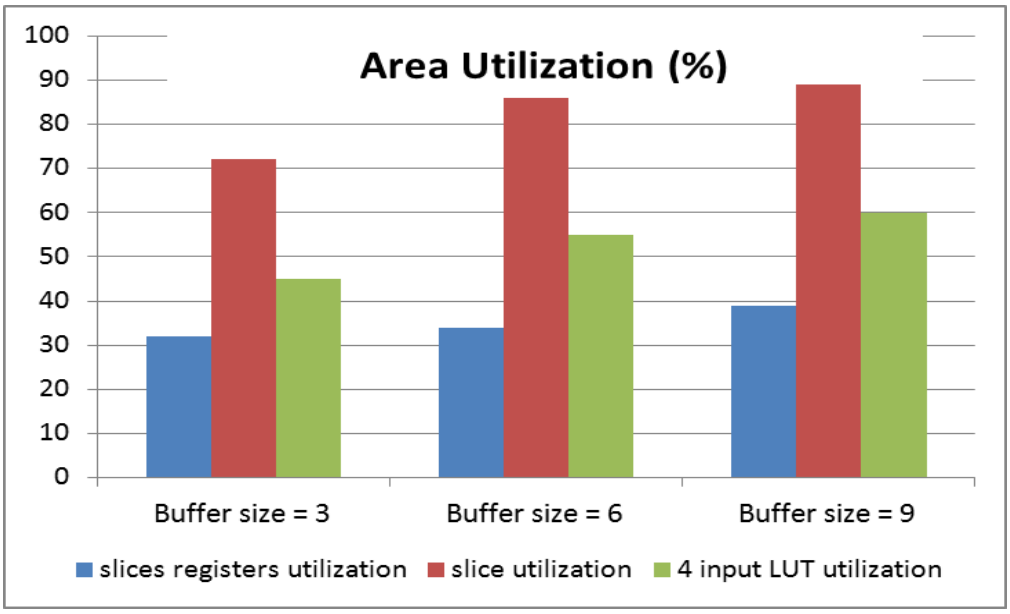

Figure 25 : MPEG-4 Area utilization

As we can see the resources of the chip required increase as the buffer size increases. In the case of buffer size 6 and 9 a significant percentage of the chip slices are used(almost 90%). A larger buffer size would take up all the chip resources.

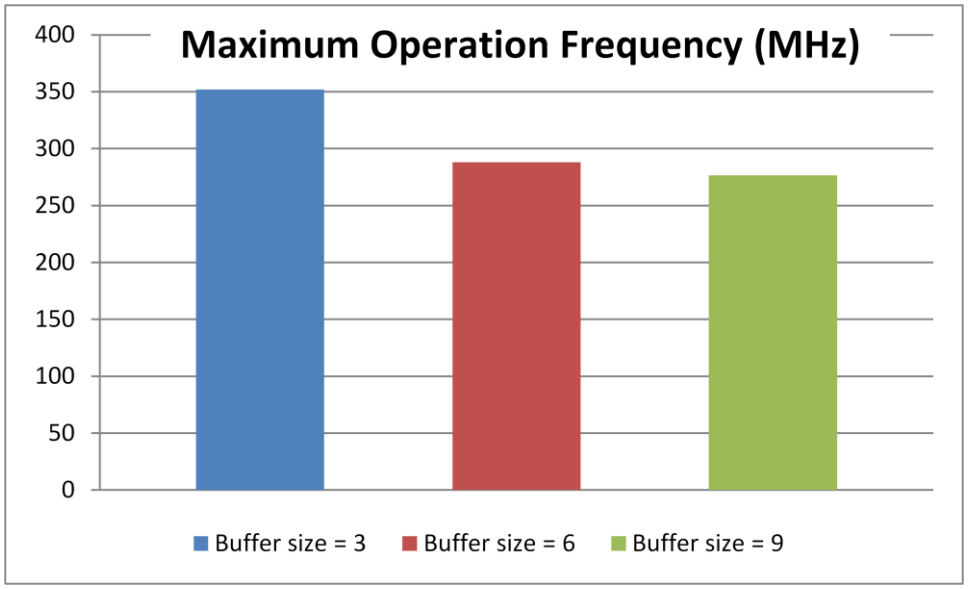

Figure 26 : MPEG-4 Maximum Frequency

The maximum frequency at which the chip can function is 276 MHz in case of buffer size 9. This frequency is well greater than our clock frequency (100 MHz). Furthermore, we notice that the

ratio in performance degradation in term of maximum operation frequency is greater from buffer size 3 to 6 than buffer size 6 to 9. This is due to the saturation, which takes place when the buffer size is 6.

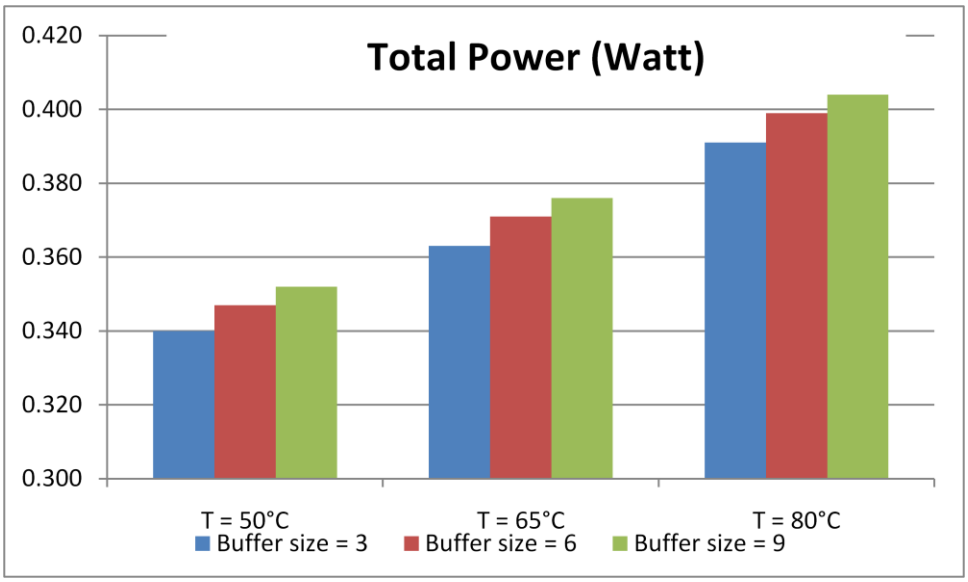

Figure 27 : MPEG-4 Total Power Dissipation

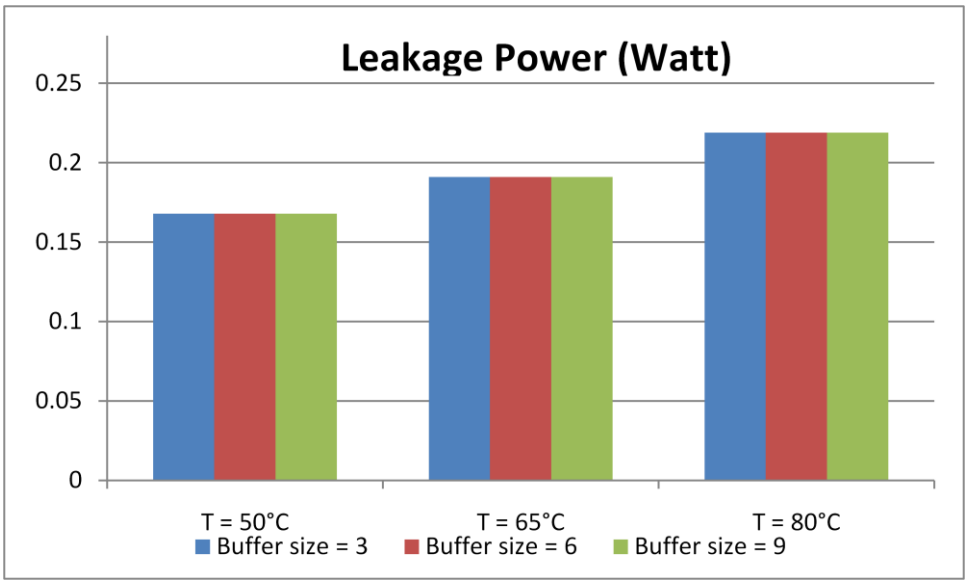

Figure 28 : MPEG-4 Leakage Power Dissipation

The total power dissipation increases as the buffer size increases. Furthermore, as expected increase of chip temperature influences significantly the power. For this application leakage

power is 0.168 at 50°C, 0.191 at 65°C, 0.219 at 80°C independent of the buffer size. The leakage power for given temperature is constant among the different buffer sizes, because it depends mainly on temperature.

5.1.2 VOPD

Video object plane decoder is another digital signal processing application that has been proposed for use on NoC and studied before (MUR, 2005). VOPD offers quality video transition with decent bandwidth performance. The tested VOPD decoder includes twelve processing elements, such as two length decoders, an AC-DC prediction, an ARM processor, two memory components and a VOP reconstructor. The cores needed for the application are 12 and so many are the routers as well.

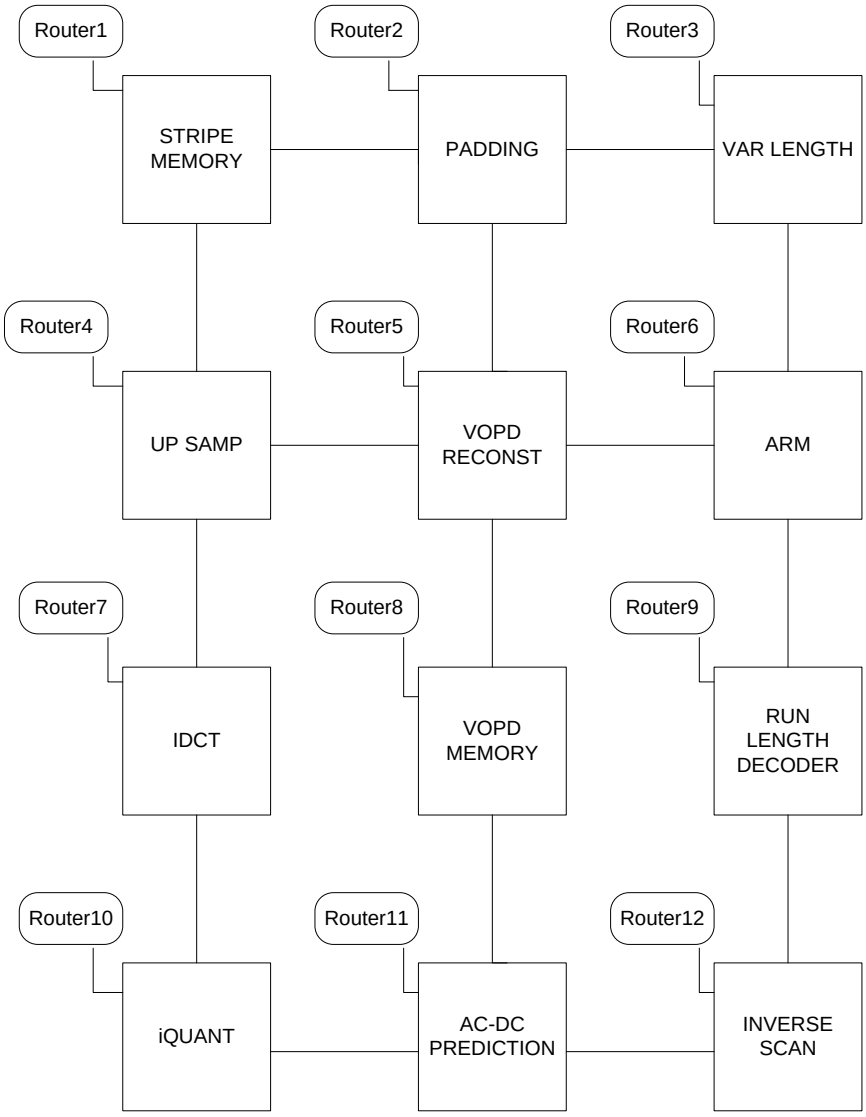

Figure 29: Block diagram of VOPD (12 cores)

Area utilization, frequency and power dissipation are depicted in upcoming figures.

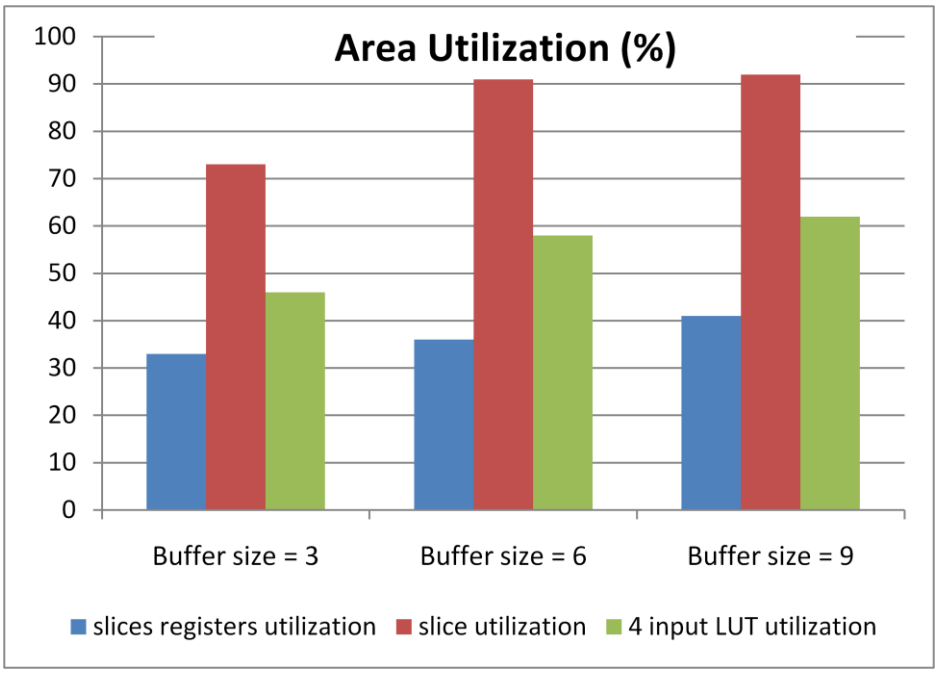

Figure 30 : VOPD Area utilization

As we can see the resources of the chip required increase as the buffer size increases. In the case of buffer size 6 and 9 a significant percentage of the chip slices are used(almost 90%). A larger buffer size would take up all the chip resources.

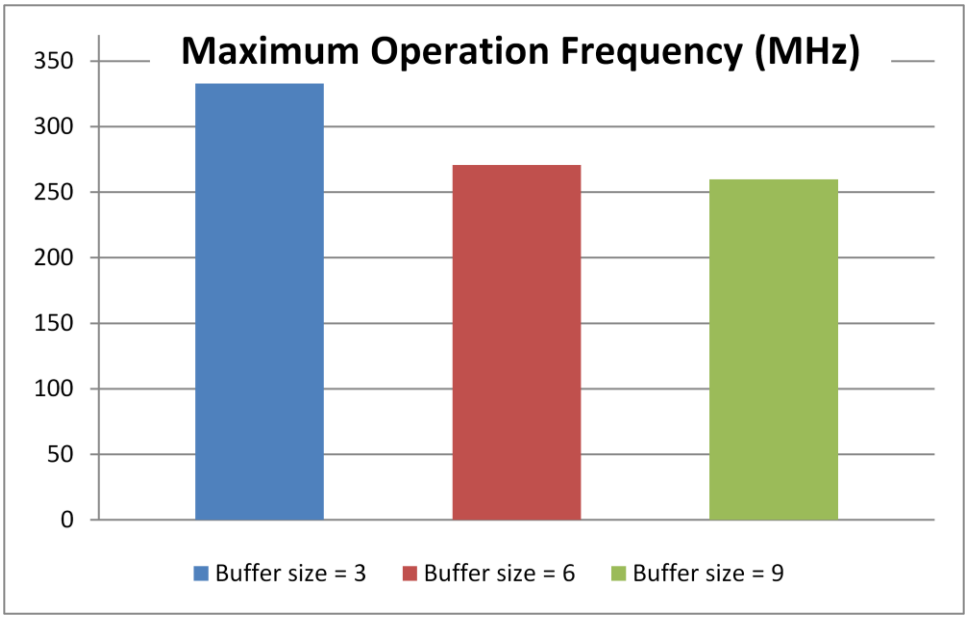

Figure 31 : VOPD Maximum Frequency

The maximum frequency at which the chip can function is 260 MHz in case of buffer size 9. This frequency is well greater than our clock frequency (100 MHz). Furthermore, we notice that the ratio in performance degradation in term of maximum operation frequency is greater from buffer size 3 to 6 than buffer size 6 to 9. This is due to the saturation, which takes place when the buffer size is 6.

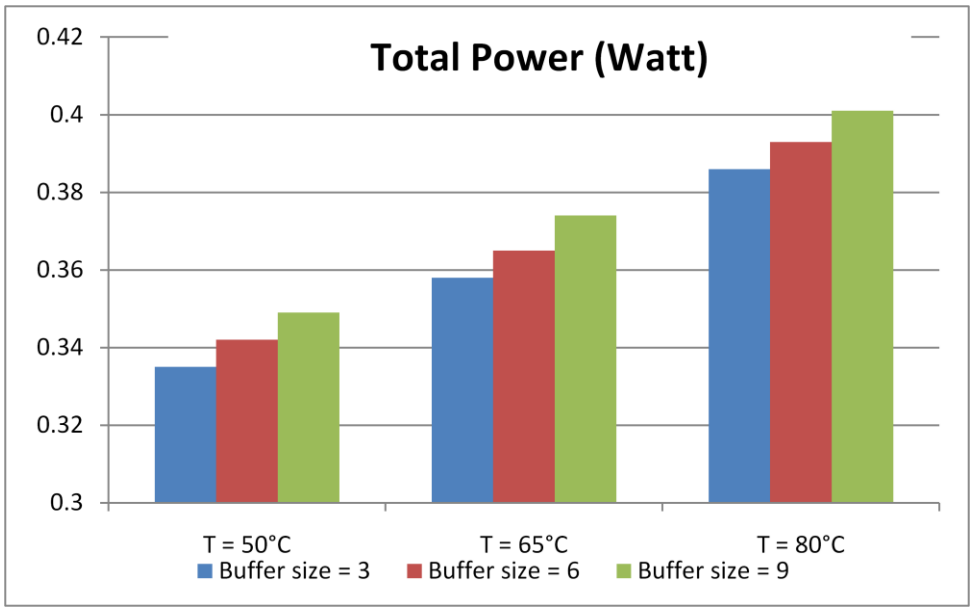

Figure 32: VOPD Total Power Dissipation

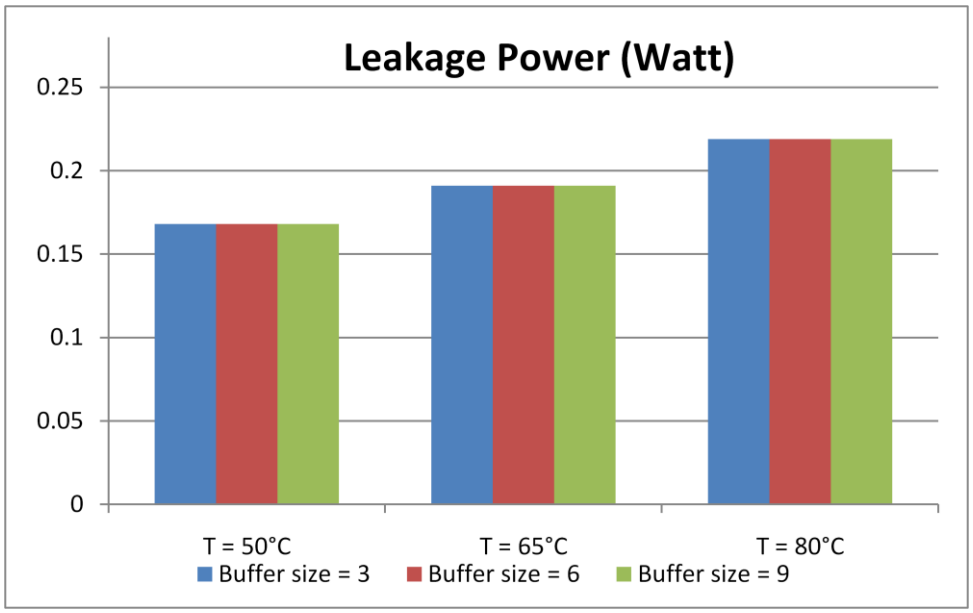

Figure 33: VOPD Leakage Power Dissipation

The total power dissipation increases as the buffer size increases. Furthermore, as expected increase of chip temperature influences significantly the power. For this application leakage power is again 0.168 at 50°C, 0.191 at 65°C, 0.219 at 80°C independent of the buffer size. The leakage power for given temperature is constant among the different buffer sizes, because it depends mainly on temperature.

5.1.3 MWD

Multi window display is another digital signal processing application (TAM, 2005), which is also suitable for NoC architectures and also uses twelve processing elements. The cores needed for the application are 12 and so many are the routers as well.

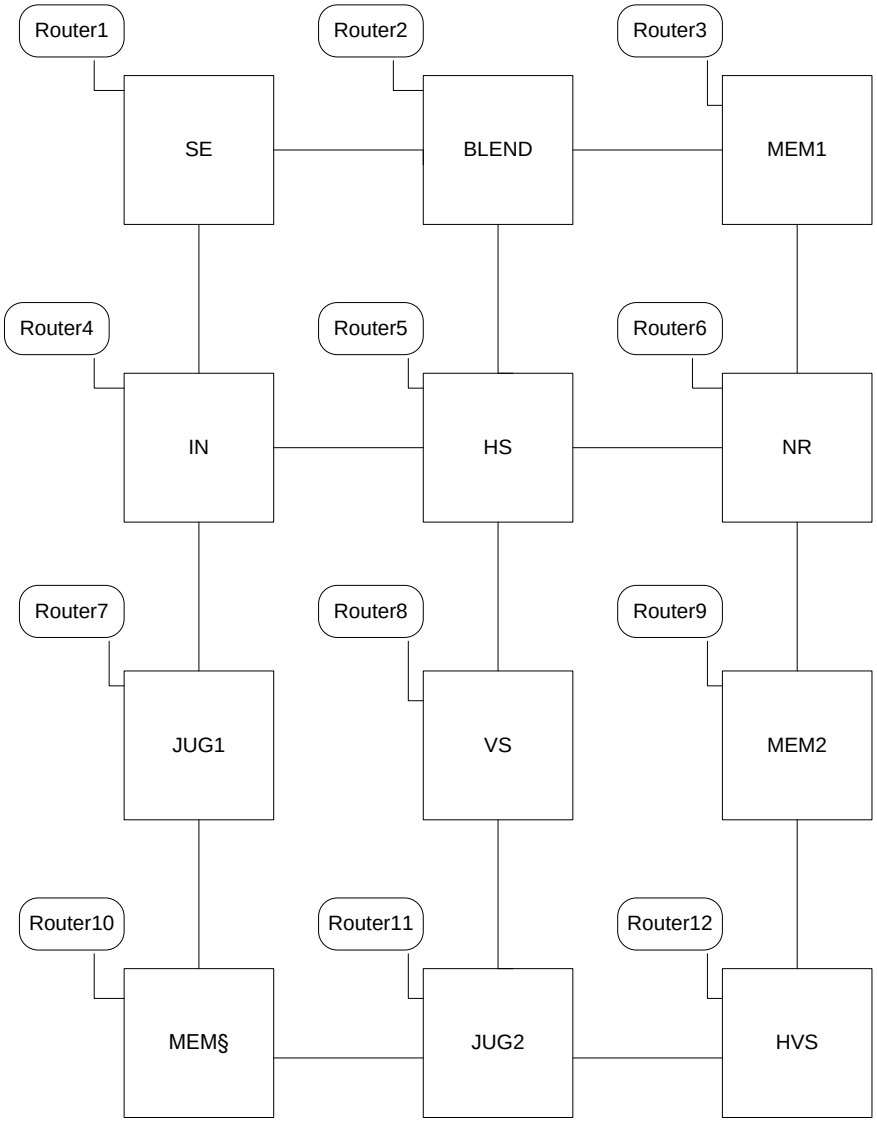

Figure 34: Block diagram of MWD(12 cores)

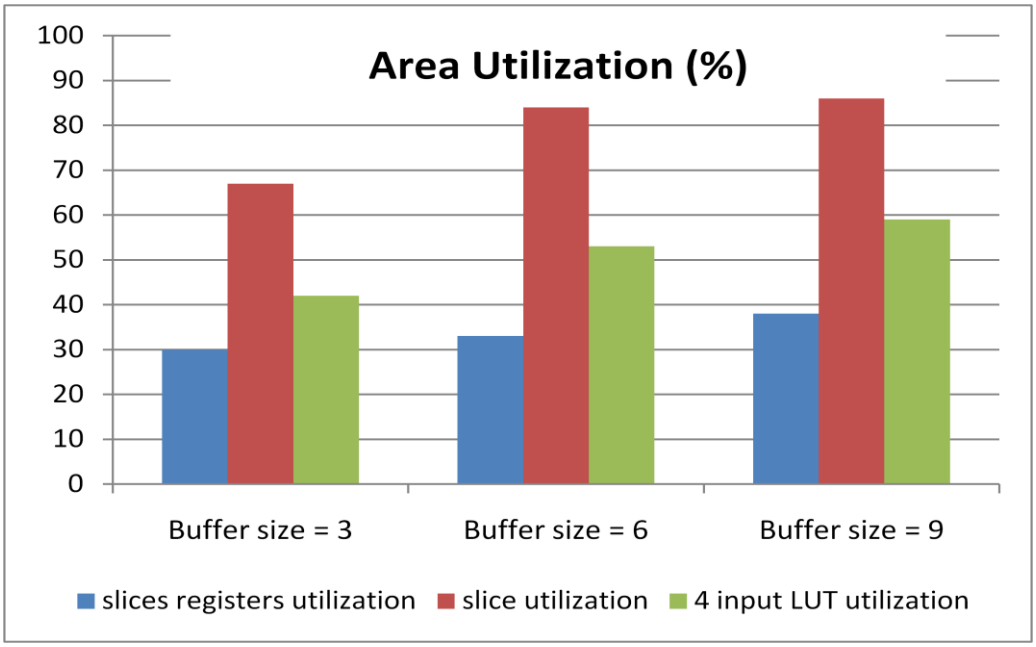

Figure 35 : MWD Area utilization

As we can see the resources of the chip required increase as the buffer size increases. In the case of buffer size 6 and 9 a significant percentage of the chip slices are used(almost 86%). A larger buffer size would take up all the chip resources.

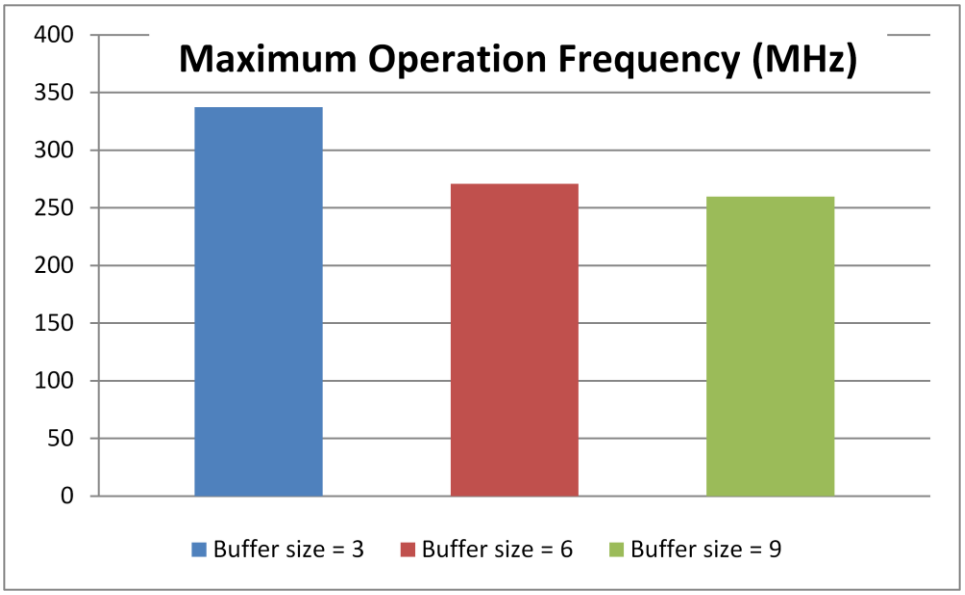

Figure 36 : MWD Maximum Frequency

The maximum frequency at which the chip can function is 260 MHz in case of buffer size 9. This frequency is well greater than our clock frequency (100 MHz). Furthermore, we notice that the ratio in performance degradation in term of maximum operation frequency is greater from buffer size 3 to 6 than buffer size 6 to 9. This is due to the saturation, which takes place when the buffer size is 6.

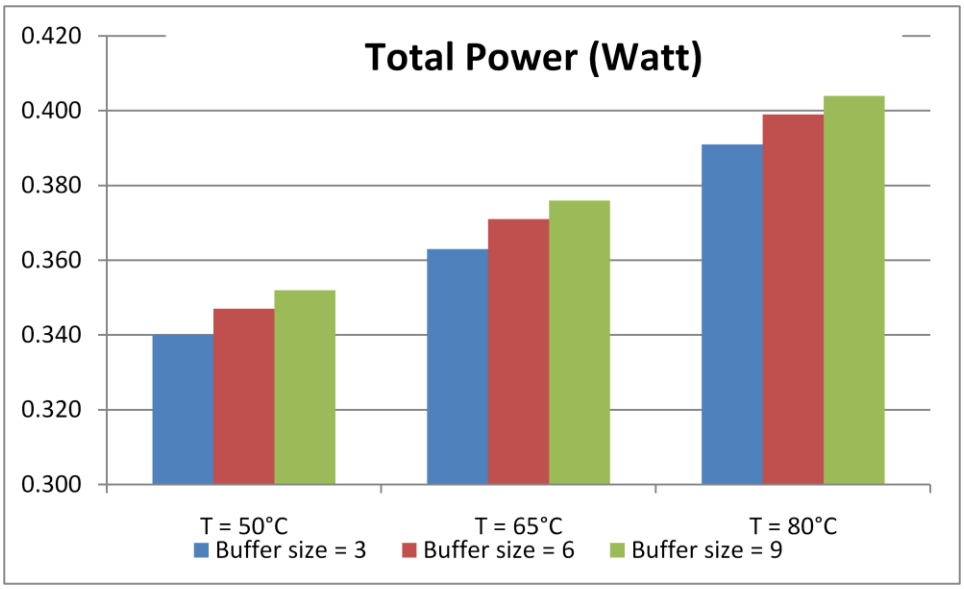

Figure 37 : MWD Total Power Dissipation

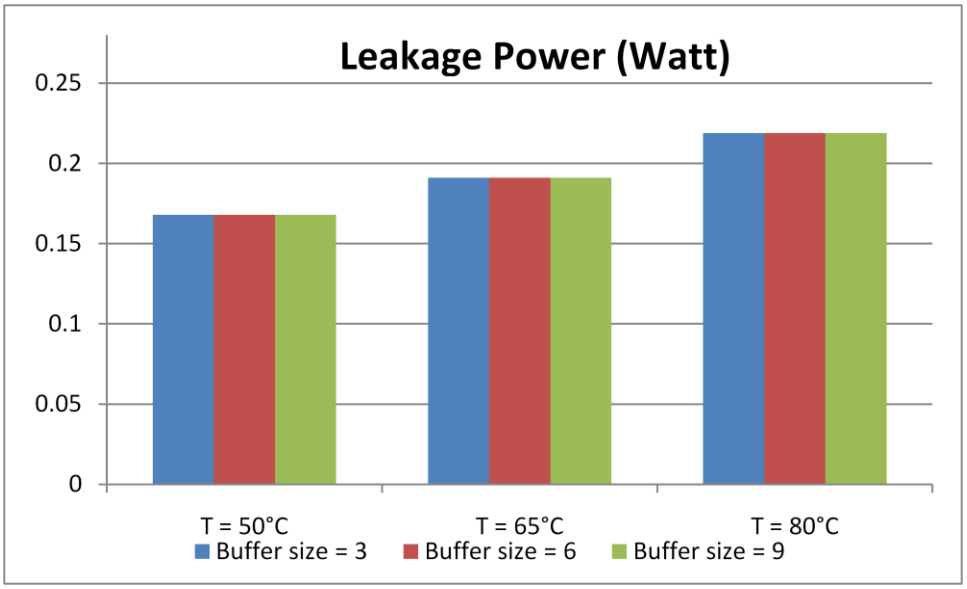

Figure 38 : MWD Leakage Power Dissipation

The total power dissipation increases as the buffer size increases. Furthermore, as expected increase of chip temperature influences significantly the power. For this application leakage power is again 0.168 at 50°C, 0.191 at 65°C, 0.219 at 80°C independent of the buffer size. The leakage power for given temperature is constant among the different buffer sizes, because it depends mainly on temperature.

5.1.4 MMS

In this section a multimedia system (MMS) is tested. The system contains 25 cores, including several memories and DSP processors. In this case, we will use 3 routers, since the strategy according to which a router attached to each core requires far too many resources. For the clustering of routers, we wanted to achieve traffic minimization. The existence of communication links between routers ensures the lack of deadlocks and livelocks. Furthermore, in this application we are going to use a user-defined look up table for the efficient routing of the data. The xy routing has no sense here since we do not have a classic mesh structure.

The diagram with the area utilization does not present percentage values as the above ones, nut the actual unit numbers. The percentage values would not be enlightening, since the chip utilization of this design does not exceed 5-6%.

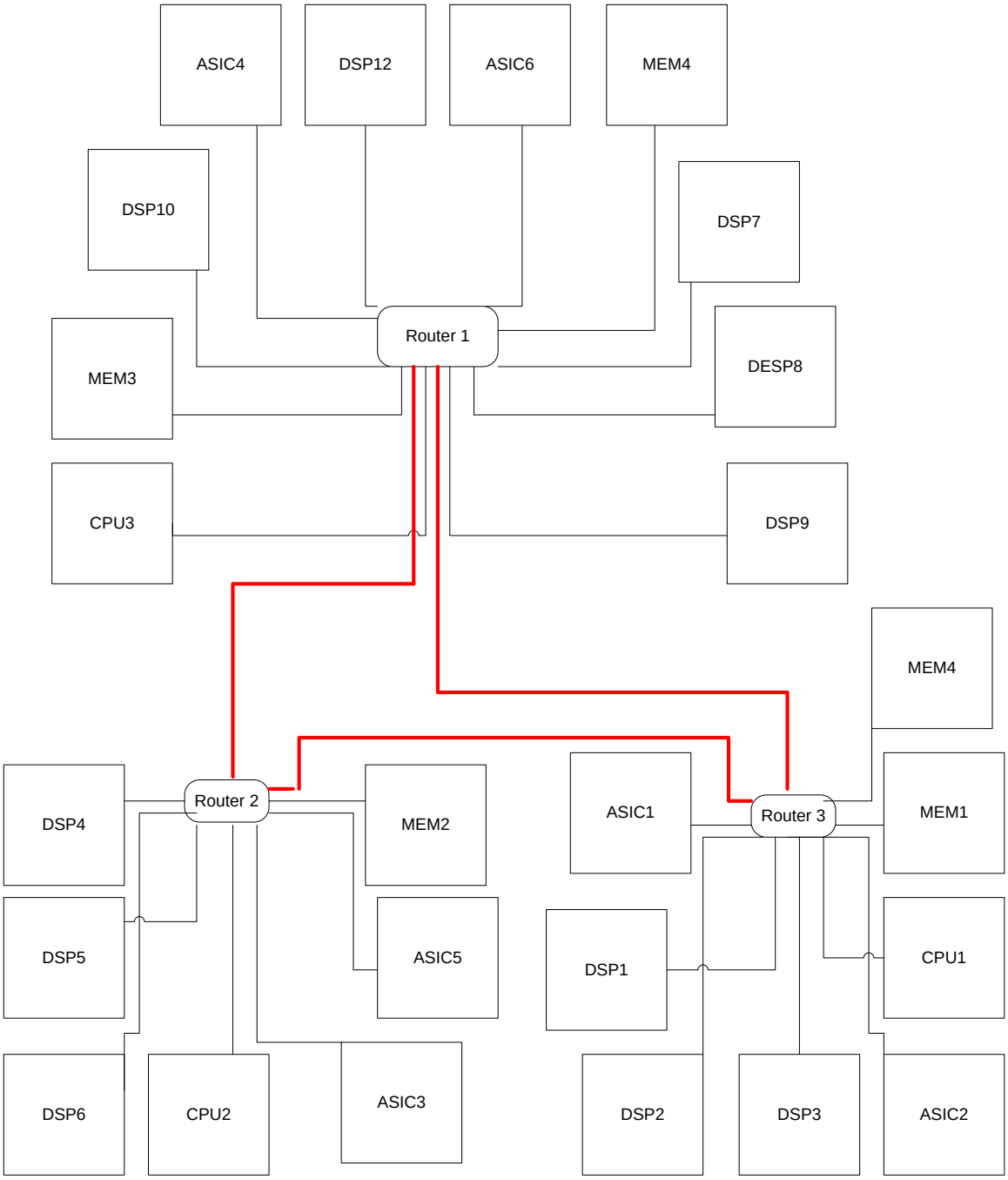

Figure 39 : Block diagram of MMS

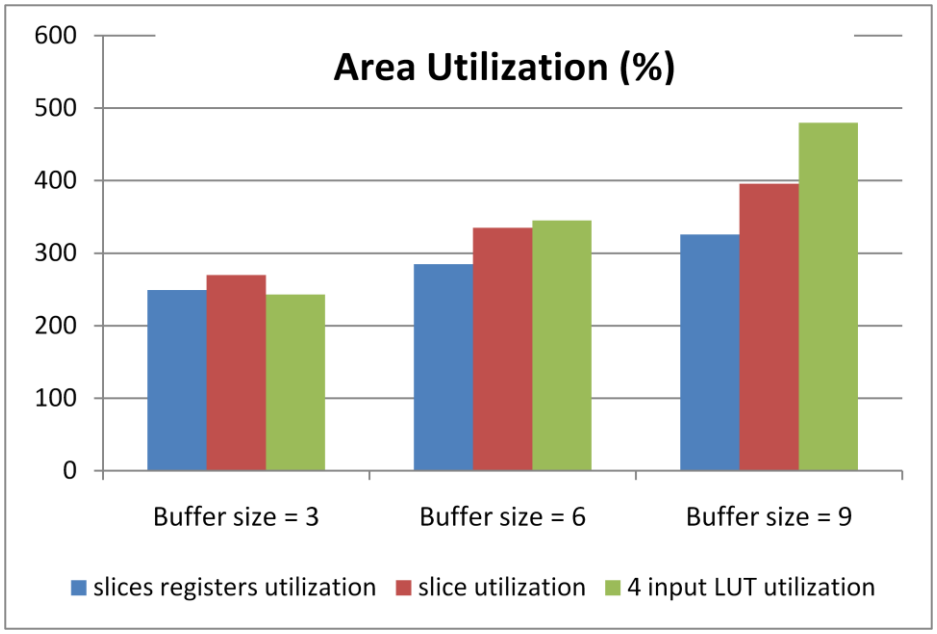

Figure 40 : MMS Area utilization

As we can see the resources of the chip required increase as the buffer size increases. But even in the case of buffer 9, only a small percentage of the chip slices are used(about 6%). A larger buffer size could be used in this case, if the traffic is to be serviced more efficiently.

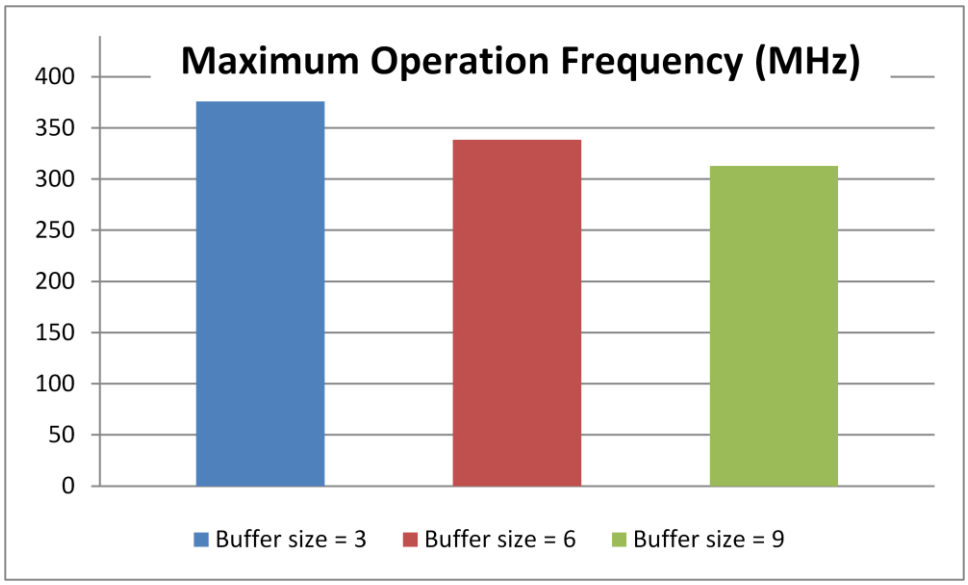

Figure 41 : MMS Maximum Frequency

The maximum frequency at which the chip can function is 312 MHz in case of buffer size 9. This frequency is well greater than our clock frequency (100 MHz).

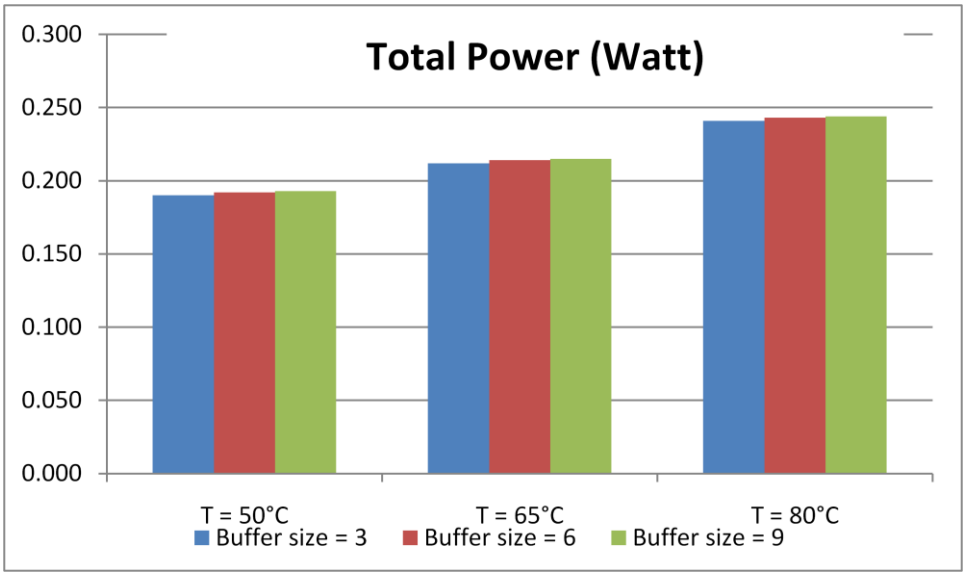

Figure 42: MMS Total Power Dissipation

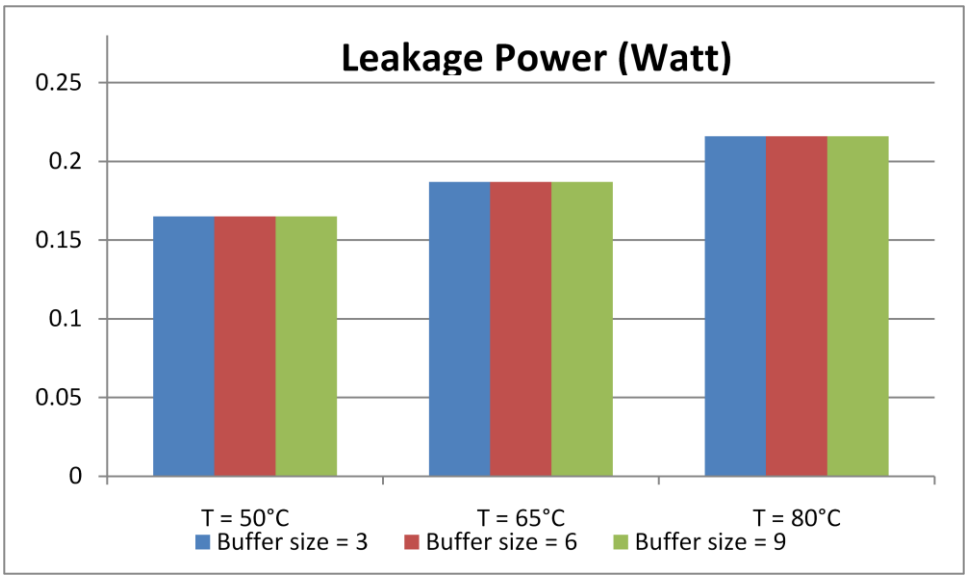

Figure 43: MMS Lekage Power Dissipation

The total power dissipation increases as the buffer size increases. Furthermore, as expected increase of chip temperature influences significantly the power. For this application leakage power is 0.165 at 50°C, 0.187 at 65°C, 0.216 at 80°C independent of the buffer size.Although the design is significantly smaller than the previous ones, the leakage values are near the previous ones.

5.2 Conclusions

As we can see, for all the applications, the utilization of all the resources on the chip increases as the buffer size increases. This behavior is expected, since the buffer component is responsible for a large proportion of the chip resources.

The power dissipated increases too, as the buffer size increases. This can be explained by the fact that the additional buffer space needs more chip area, which increases the overall power (dynamic and through leakage) produced.

As the buffer size increases, the maximum frequency at which the design can function decreases. That happens because a design with many slices requires more interconnects, which results in delays and need for a slower clock.

What we have noticed is that all the 3 applications mwd, vopd and mpeg, which consist of 12 cores have similar values as far as resources, power and frequency is concerned. That is expected since the design has similar complexity (equal number of routers). On the other side, mms has one fourth of the routers of the other designs, hence the corresponding values are significantly smaller.

Chapter 6:

Summary and Future Work

Abstract

In this chapter we present the summary of our work and suggest extensions.

6.1 Summary

In the current thesis we studied irregular NoC architectures, created a generic NoC platform and we developed a tool for their automatic generation.

The tool enables the easy and fault-free generation of the code of a NoC, which has the characteristics that the user desires. The user has only to write the requirements in some pre-defined xml files.

After the development of the tool, we had the chance to test various NoCs which were targeted for some common hardware applications. We thus examined our NoC architecture from the area, timing and power scope. The results showed the needs of each design, and give the designer a helpful first evaluation of the application requirements.

Furthermore, the tool along with the ones developed by the Microlab laboratory offer the designer the chance to explore the design communication schema and adopt its architectures to the application needs and constraints.

Finally, NoCGen contributes to the fast prototyping of NoC architectures, giving the possibility for efficient generation and debugging of NoCs.

6.2 Future work

We are going to suggest here some extensions of the work done during the current thesis.

6.2.1 Additional Functionalities

The architecture itself can be improved and enriched with additional functionalities. The code has been developed according to the principles of reusability with the aim to add functionalities in the future without having to perform extensive modifications.

Keeping the existing form of inputs and outputs, the processes may be “refined” and more sophisticated techniques for routing, buffering, arbitration and flow control could be adopted. Moreover, techniques for power-aware design from the literature could be used. The features integrated depend on the decisions of each designer.

6.2.2 Floorplanning

Each router has a unique "ID" on the chip – its coordinates. If the designer uses advanced tools for the efficient placement of the routers on the chip, this would have a significant impact on the power dissipation and interconnect delays -two topics, which are crucial for every chip. As a result, effective floorplanning will be a key in the NoC development.

References

[1] Outstanding Research Problems in NoC Design : System, Microarchitecture, and Circuit Perspectives (Marculescu et al)

[2] A Survey of Research and Practices of Network-on-Chip (Bjerregaard et al)

[3] A Reconfigurable Crossbar Switch with Adaptive Bandwidth Control for Networks-on-Chip (Kim et al)

[4] A Generic and Extensible Spidergon NoC (Zitouni et al)

[5] RASoC :A Router Soft-Core for Network-on-Chip (Zeferino et al)

[6] Power Consumption and Performance Analysis of 3D NoCs (Sharifi et al)

[7] High-Speed Buffered Crossbar Switch Design Using Virtex-EM Devices (Singhal et al)

[8] Roung-robin Arbiter Design and Generation (Shin et al)

[9] Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip (Rijpkema et al)

[10] Low-Latency Virtual-Channel Routers for On-Chip Networks (Mullins et al)

[11] Æthereal Network on Chip:Concepts, Architectures,and Implementations (Goosens et al)

[12] SPIN: a Scalable, Packet Switched, On-chip Micro-network (Adriahantenaina et al)

[13] Flit-Reservation Flow Control (Peh et al)

[14] Power-driven Design of Router Microarchitectures in On-chip Networks (Hangsheng et al)

[15] Trade-Offs in the Configuration of aNetwork on Chip for Multiple Use-Cases (Hansson)

[16] Low-Power Network-on-Chip for High-Performance SoC Design (Yoo et al)

[17] Circuit Design with VHDL" (by Volnei Pedroni)

[18] "Reuse Methodology Manual for System on Chip Designs" (by Keating, Bricaud)

Appendix

Tool Code

noc_pkg_gen.hpp

```
#ifndef XML_PARSER_HPP
#define XML_PARSER_HPP
/**
* @file
* Class "GetConfig" provides the functions to read the XML data.
* @version 1.0
*/
#include <xercesc/dom/DOM.hpp>
#include <xercesc/dom/DOMDocument.hpp>
#include <xercesc/dom/DOMDocumentType.hpp>
#include <xercesc/dom/DOMElement.hpp>
#include <xercesc/dom/DOMImplementation.hpp>
#include <xercesc/dom/DOMImplementationLS.hpp>
#include <xercesc/dom/DOMNodeIterator.hpp>
#include <xercesc/dom/DOMNodeList.hpp>
#include <xercesc/dom/DOMText.hpp>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/util/XMLUni.hpp>
#include <string>
#include <stdexcept>
#include <algorithm>
#include <math.h>
#include <fstream>
const int MAX_PENUM = 50;
const int MAX_PORTS = 10;
// Error codes
enum {
ERROR_ARGS = 1,
ERROR_XERCES_INIT,
ERROR_PARSE,
```

```
ERROR_EMPTY_DOCUMENT
};
class GetConfig
{
public:
//constructor
GetConfig();
//destructor
~GetConfig();
//functions
void readConfigFile(std::string&) throw(std::runtime_error);
void calculate();
void print();
int max(int array[]);
void binary(int number);
//variables
int x[MAX_PENUM],y[MAX_PENUM];
int switchPorts[MAX_PENUM];
int switchDepth[MAX_PENUM];
int peNum;
int peNumx;
int peNumy;
int maxPorts;
int maxDepth;
char binvalue[MAX_PORTS];//it is log2(max_ports)
int allbutiarray[MAX_PORTS][MAX_PORTS-1];
int shift_array[MAX_PORTS-1][MAX_PORTS-1];
private:
xercesc::XercesDOMParser *m_ConfigFileParser;
char* m_x;
char* m_y;
char* m_switchPorts;
char* m_switchDepth;
// Internal class use only. Hold Xerces data in UTF-16 SMLCh type.
XMLCh* TAG_switch;
XMLCh* TAG_switchPorts;
XMLCh* TAG_switchDepth;
XMLCh* ATTR_x;
XMLCh* ATTR_y;
};
```

```
#endif
```

noc_pkg_gen.cpp

```
#include <string>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <list>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include "noc_pkg_gen.hpp"
using namespace xercesc;
using namespace std;
/**
* Constructor initializes xerces-C libraries.
* The XML tags and attributes which we seek are defined.
* The xerces-C DOM parser infrastructure is initialized.
*/
GetConfig::GetConfig()
{
try
{
XMLPlatformUtils::Initialize(); // Initialize Xerces infrastructure
}
catch( XMLException& e )
{
char* message = XMLString::transcode( e.getMessage() );
cerr << "XML toolkit initialization error: " << message << endl;
XMLString::release( &message );
// throw exception here to return ERROR_XERCES_INIT
}
// Tags and attributes used in XML file.
// Can't call transcode till after Xerces Initialize()
TAG_switch = XMLString::transcode("switch");
TAG_switchPorts= XMLString::transcode("switchPorts");
```

```
TAG_switchDepth= XMLString::transcode("switchDepth");
ATTR_x = XMLString::transcode("x");
ATTR_y = XMLString::transcode("y");
m_ConfigFileParser = new XercesDOMParser;
}
/**
* Class destructor frees memory used to hold the XML tag and
* attribute definitions. It als terminates use of the xerces-C
* framework.
*/
GetConfig::~GetConfig()
{
// Free memory
delete m_ConfigFileParser;
if(m_x) XMLString::release( &m_x );
if(m_y) XMLString::release( &m_y );
if(m_y) XMLString::release( &m_switchPorts );
if(m_y) XMLString::release( &m_switchDepth );
try
{
XMLString::release( &TAG_switch );
XMLString::release( &TAG_switchPorts );
XMLString::release( &TAG_switchDepth );
XMLString::release( &ATTR_x );
XMLString::release( &ATTR_y );
}
catch( ... )
{
cerr << "Unknown exception encountered in TagNamesdtor" << endl;
}
// Terminate Xerces
try
{
XMLPlatformUtils::Terminate(); // Terminate after release of memory
}
catch( xercesc::XMLException& e )
{
char* message = xercesc::XMLString::transcode( e.getMessage() );
cerr << "XML ttolkit teardown error: " << message << endl;
XMLString::release( &message );
```

```
}
}
/**
* This function:
* - Tests the access and availability of the XML configuration file.
* - Configures the xerces-c DOM parser.
* - Reads and extracts the pertinent information from the XML config file.
*
* @param in configFile The text string name of the HLA configuration file.
*/
void GetConfig::readConfigFile(string& configFile)
throw( std::runtime_error )
{
// Test to see if the file is ok.
struct stat fileStatus;
int iretStat = stat(configFile.c_str(), &fileStatus);
if( iretStat == ENOENT )
throw ( std::runtime_error("Path file_name does not exist, or path is
an empty string.") );
else if( iretStat == ENOTDIR )
throw ( std::runtime_error("A component of the path is not a directory
."));
else if( iretStat == ELOOP )
throw ( std::runtime_error("Too many symbolic links encountered while
traversing the path."));
else if( iretStat == EACCES )
throw ( std::runtime_error("Permission denied."));
else if( iretStat == ENAMETOOLONG )
throw ( std::runtime_error("File can not be read\n"));
// Configure DOM parser.
m_ConfigFileParser->setValidationScheme( XercesDOMParser::Val_Never );
m_ConfigFileParser->setDoNamespaces( false );
m_ConfigFileParser->setDoSchema( false );
m_ConfigFileParser->setLoadExternalDTD( false );
try
{
noc_pkg_gen.cpp
m_ConfigFileParser->parse( configFile.c_str() );
// no need to free this pointer - owned by the parent parser object
DOMDocument* xmlDoc = m_ConfigFileParser->getDocument();
```

```
// Get the top-level element: NAme is "root". No attributes for "root"
DOMElement* elementRoot = xmlDoc->getDocumentElement();
if( !elementRoot ) throw(std::runtime_error( "empty XML document" ));
// Parse XML file for tags of interest: "ApplicationSettings"
// Look one level nested within "root". (child of root)
DOMNodeList* children = elementRoot->getChildNodes();
const XMLSize_t nodeCount = children->getLength();
// For all nodes, children of "root" in the XML tree.
int i=0;
for( XMLSize_t xx = 0; xx < nodeCount; ++xx )
{
DOMNode* currentNode = children->item(xx);
if( currentNode->getNodeType() && // true is not NULL
currentNode->getNodeType() == DOMNode::ELEMENT_NODE ) // is ele
ment
{
// Found node which is an Element. Re-cast node as element
DOMElement* currentElement = dynamic_cast< xercesc::DOMElement*
>( currentNode );
if( XMLString::equals(currentElement->getTagName(), TAG_switch))
{
const XMLCh* xmlch_x = currentElement->getAttribute(ATTR_x);
m_x = XMLString::transcode(xmlch_x);
x[i]=atoi(m_x);
const XMLCh* xmlch_y = currentElement->getAttribute(ATTR_y);
m_y = XMLString::transcode(xmlch_y);
y[i]=atoi(m_y);
}
DOMNodeList* grandchildren = currentElement->getChildNodes();
const XMLSize_t nodeCount_b = grandchildren->getLength();
for( XMLSize_t yy = 0; yy < nodeCount_b; ++yy )
{
DOMNode* currentNodeb = grandchildren->item(yy);
if( currentNodeb->getNodeType() && currentNodeb->getNodeTyp
e() == DOMNode::ELEMENT_NODE ) // is element
{
DOMElement* currentElementb = dynamic_cast< xercesc
::DOMElement* >( currentNodeb );
if( XMLString::equals(currentElementb->getTagName(),
TAG_switchPorts))
```

```
{
const XMLCh* xmlch_ports = currentNodeb->ge
tTextContent();
m_switchPorts = XMLString::transcode(xmlch_
ports);
switchPorts[i]=atoi(m_switchPorts);
}
if( XMLString::equals(currentElementb->getTagName()
, TAG_switchDepth))
{
const XMLCh* xmlch_depth = currentNodeb->ge
tTextContent();
m_switchDepth = XMLString::transcode(xmlch_
depth);
switchDepth[i++]=atoi(m_switchDepth);
}
}
}
}
}
peNum = i;
cout << "arithmos switches: " << peNum << endl;
}
catch( xercesc::XMLException& e )
{
char* message = xercesc::XMLString::transcode( e.getMessage() );
ostringstream errBuf;
errBuf << "Error parsing file: " << message << flush;
XMLString::release( &message );
}
}
//function max
int GetConfig::max(int array[]){
int maxvalue = array[0];
for (int i = 0; i < peNum; i++)
{
if (array[i] > maxvalue)
{
maxvalue = array[i];
}
```

```
};
return maxvalue;
}
//function binary
void GetConfig::binary(int number) {
int remainder,i=0;
for (int j=0;j<log2(maxPorts);j++)
{
binvalue[j]='0';
}
while (number>1)
{
remainder = number % 2;
number = number / 2;
binvalue[i++] = remainder + '0';
}
binvalue[i] = number + '0' ;
}
//function calculate
void GetConfig::calculate()
{
int array[peNum];
int i;//peNumx
for (i=0;i<peNum;i++){
array[i] = x[i];
}
sort(array,array+peNum);
peNumx=1;
for(i = 0; i < peNum -1 /*since we don't want to compare last element with
junk*/; i++)
{
if(array[i]==array[i+1])
continue;
else
peNumx++;
}
cout <<"peNumx: " << peNumx <<endl;
//peNumy
for (i=0;i<peNum;i++){
array[i] = y[i];
```

```
}
sort(array,array+peNum);
peNumy=1;
for(int i = 0; i < peNum -1 /*since we don't want to compare last element
with junk*/; i++)
{
if(array[i]==array[i+1])
continue;
else
peNumy++;
}
cout <<"peNumy: " << peNumy <<endl;
//maxPorts
maxPorts = max(switchPorts);
cout <<"maxPorts: " << maxPorts <<endl;
//maxDepth
maxDepth = max(switchDepth);
cout <<"maxDepth: " << maxDepth <<endl;
for(int i = 0; i < peNum /*since we don't want to compare last element wit
h junk*/; i++)
{
cout <<"x"<<x[i]<<"y"<<y[i]<< "ports "<<switchPorts[i]<< "depth"<<sw
itchDepth[i]<<endl;
}
//allbutiarray
for (int i=0; i<maxPorts;i++){
int k = 0;
for(int j=0;j<maxPorts-1;j++){
if (k==i){
allbutiarray[i][j]=++k;
}
else {
allbutiarray[i][j]=k;
}
k++;
}
}
for (int i=0; i<maxPorts;i++){
for(int j=0;j<maxPorts-1;j++){
cout << allbutiarray[i][j];
```

```
}
cout <<endl;
}
//tx_grant_enc_const
binary(6);
for(int i=0;i<log2(maxPorts);i++){
cout << binvalue[i];
}
cout << endl;
binary(5);
for(int i=0;i<log2(maxPorts);i++){
cout << binvalue[i];
}
//shift array
int temp[maxPorts-1];
for (int i=0; i<maxPorts-1;i++){
temp[i]=i;
}
for (int i=0; i<maxPorts-1;i++){
int k=0;
for(int j=0;j<maxPorts-1;j++){
if ((k+i) <= maxPorts-2) {
shift_array[i][j]=temp[k+i];
k++;
}
else{
k=-i;
shift_array[i][j]=temp[k+i];
k++;
}
}
}
cout << endl;
for (int i=0; i<maxPorts-1;i++){
for(int j=0;j<maxPorts-1;j++){
cout<<shift_array[i][j];
}
cout <<endl;
}
}
```

```
//function print
void GetConfig::print()
{ofstream myfile;
myfile.open ("vhdl/noc_pkg.vhd",std::ios::out);
myfile.seekp(0,ios::beg);
myfile << "library ieee;\nlibrary work;\nuse ieee.std_logic_1164.all;\nuse
ieee.std_logic_unsigned.all;\nuse work.noc_functions.all;\nPACKAGE
noc_pkg is\n--configurable\nCONSTANT wordlength: integer
:= 16; --width of data word in bits\n";
myfile << "CONSTANT pe_num_x: integer :="<< pe
Numx ;
myfile << ";\nCONSTANT pe_num_y: integer :=" <
< peNumy ;
myfile << ";\nCONSTANT pe_num: integer :=" <
< peNum ;
myfile << ";\nTYPE parameter_array is array(0 to pe_num -1) of integer;\n";
myfile << "CONSTANT switch_depth_array: parameter_array := (";
for (int i=0;i<peNum-1;i++)
myfile << switchDepth[i]<<",";
myfile << switchDepth[peNum-1] << ");\n";
myfile << "CONSTANT switch_ports_array: parameter_array := (";
for (int i=0;i<peNum-1;i++)
myfile << switchPorts[i]<<",";
myfile << switchPorts[peNum-1] << ");\n";
myfile << "CONSTANT x_s_array: parameter_array := (";
for (int i=0;i<peNum-1;i++)
myfile << x[i]<<",";
myfile <<x[peNum-1] << ");\n";
myfile << "CONSTANT y_s_array: parameter_array := (";
for (int i=0;i<peNum-1;i++)
myfile << y[i]<<",";
myfile <<y[peNum-1] << ");\n";
myfile << "CONSTANT max_ports: integer :=" << m
axPorts ;
myfile << ";\nCONSTANT switch_depth_max: integer :=" <
< maxDepth ;
myfile << ";\nTYPE allbuti_array is array(0 to max_ports-1,0 to max_ports-
2) of integer;";
myfile << "\nCONSTANT all_but_i :allbuti_array :=(";
for (int i=0;i<maxPorts-1;i++){
```

```
myfile << "(";
for (int j=0;j<maxPorts-2;j++){
myfile << allbutiarray[i][j] << ",";
}
myfile << allbutiarray[i][maxPorts-2];
myfile << "),";
}
myfile << "(";
for (int j=0;j<maxPorts-2;j++){
myfile << allbutiarray[maxPorts-1][j] << ",";
}
myfile << allbutiarray[maxPorts-1][maxPorts-2];
myfile << "));";
myfile << "\nTYPE tx_grant_enc_array_2 is array(0 to max_ports-2) of std_log
ic_vector(log(max_ports-1)-1 downto 0);";
myfile << "\nCONSTANT tx_grant_enc_const : tx_grant_enc_array_2
:=";
myfile << "(";
for (int i=0;i<maxPorts-2;i++){
binary(i);
myfile << "\"";
for(int j=log2(maxPorts-1)-1; j>=0;j--){
myfile << binvalue[j];
}
myfile << "\", ";
}
myfile << "\"";
binary(maxPorts-2);
for(int j=log2(maxPorts-1)-1; j>=0;j--){
myfile << binvalue[j];
}
myfile << "\");";
myfile << "\nTYPE shift_array_type is array( 0 to max_ports-2,0 to max_ports
-2) of integer;" ;
myfile << "\nCONSTANT shift_array : shift_array_type := (";
for (int i=0;i<maxPorts-2;i++){
myfile << "(";
for (int j=0;j<maxPorts-2;j++){
myfile << shift_array[i][j] << ",";
}
```

```
myfile << shift_array[i][maxPorts-2];
myfile << "),";
}
myfile << "(";
for (int j=0;j<maxPorts-2;j++){
myfile << shift_array[maxPorts-2][j] << ",";
}
myfile << shift_array[maxPorts-2][maxPorts-2];
myfile << "));";
myfile << "\nTYPE rom_lut_type is array (0 to pe_num_x-1,0 to pe_num_y-1) of
integer range 0 to (max_ports -1);";
myfile << "\n--configurable\n";
myfile.close();
myfile.open("vhdl/noc_pkg.vhd",std::ios::out|std::ios::app);
ifstream myfile2("vhdl/noc_pkg_aux", std::ios::in);
char str[2000];
while (!myfile2.eof()){
myfile2.getline(str,2000);
myfile << str << endl;
}
myfile.close();
myfile2.close();
}
#ifdef MAIN_TEST
/* This main is provided for unit test of the class. */
int main(int argc, char *argv[])
{
string s0 = "xml/";
string s1 = argv[1];
string s2 = "/elements.xml";
string configFile= s0 + s1 + s2; // stat file. Get ambigious segfault oth
erwise.
GetConfig appConfig;
appConfig.readConfigFile(configFile);
appConfig.calculate();
appConfig.print();
return 0;
}
#endif
```

noc_gen.hpp

```
#ifndef XML_PARSER_HPP
#define XML_PARSER_HPP
/**
* @file
* Class "GetConfig" provides the functions to read the XML data.
* @version 1.0
*/
#include <xercesc/dom/DOM.hpp>
#include <xercesc/dom/DOMDocument.hpp>
#include <xercesc/dom/DOMDocumentType.hpp>
#include <xercesc/dom/DOMElement.hpp>
#include <xercesc/dom/DOMImplementation.hpp>
#include <xercesc/dom/DOMImplementationLS.hpp>
#include <xercesc/dom/DOMNodeIterator.hpp>
#include <xercesc/dom/DOMNodeList.hpp>
#include <xercesc/dom/DOMText.hpp>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/util/XMLUni.hpp>
#include <string>
#include <stdexcept>
#include <algorithm>
#include <math.h>
#include <fstream>
const int LINKS_MAX = 500; //50*10( = MAX_PENUM * MAX_PORTS)
const int TRANS_MAX = 1000; //2*50*10( = 2*MAX_PENUM * MAX_PORTS)
const int MAX_NUMBER_OF_FLITS = 10; //number of flits in a packet
// Error codes
enum {
ERROR_ARGS = 1,
ERROR_XERCES_INIT,
ERROR_PARSE,
ERROR_EMPTY_DOCUMENT
```

```
};
class GetConfig
{
public:
//constructor
GetConfig();
//destructor
~GetConfig();
//variables
struct link_struct{char* sourceRouter; char* sourcePort ; char* destinati
onRouter; char* destinationPort;};
link_struct link_array[LINKS_MAX];
int NoOfLinks;
struct trans_struct{int inputRouter; int inputPort ; char* transBegin; ch
ar* transDuration; char* data[MAX_NUMBER_OF_FLITS];};
trans_struct trans_array[TRANS_MAX];
int NoOfTrans;
//functions
void readNetlist(std::string&) throw(std::runtime_error);
void readTraffic(std::string&) throw(std::runtime_error);
void print();
private:
xercesc::XercesDOMParser *m_ConfigFileParser;
// Internal class use only. Hold Xerces data in UTF-16 SMLCh type.
XMLCh* TAG_transmission;
XMLCh* TAG_destinationRouter;
XMLCh* TAG_destinationPort;
XMLCh* TAG_inputRouter;
XMLCh* TAG_inputPort;
XMLCh* TAG_transBegin;
XMLCh* TAG_transDuration;
XMLCh* TAG_dataSequence;
XMLCh* TAG_data;
XMLCh* TAG_link;
XMLCh* TAG_sourceRouter;
XMLCh* TAG_sourcePort;
char* m_sourceRouter;
char* m_sourcePort;
char* m_destinationRouter;
char* m_destinationPort;
```

```
char* m_inputRouter;
char* m_inputPort;
};
#endif
```

noc_gen.cpp

```
#include <string>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <list>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include "noc_gen.hpp"
using namespace xercesc;
using namespace std;
/**
* Constructor initializes xerces-C libraries.
* The XML tags and attributes which we seek are defined.
* The xerces-C DOM parser infrastructure is initialized.
*/
GetConfig::GetConfig()
{
try
{
XMLPlatformUtils::Initialize(); // Initialize Xerces infrastructure
}
catch( XMLException& e )
{
char* message = XMLString::transcode( e.getMessage() );
cerr << "XML toolkit initialization error: " << message << endl;
```

```
XMLString::release( &message );
// throw exception here to return ERROR_XERCES_INIT
}
// Tags and attributes used in XML file.
// Can't call transcode till after Xerces Initialize()
TAG_transmission = XMLString::transcode("transmission");
TAG_destinationRouter = XMLString::transcode("destinationRouter");
TAG_destinationPort= XMLString::transcode("destinationPort");
TAG_inputRouter = XMLString::transcode("inputRouter");
TAG_inputPort= XMLString::transcode("inputPort");
TAG_transBegin= XMLString::transcode("transBegin");
TAG_transDuration= XMLString::transcode("transDuration");
TAG_dataSequence=XMLString::transcode("dataSequence");
TAG_data=XMLString::transcode("data");
TAG_link = XMLString::transcode("link");
TAG_sourceRouter = XMLString::transcode("sourceRouter");
TAG_sourcePort= XMLString::transcode("sourcePort");
m_ConfigFileParser = new XercesDOMParser;
}
/**
* Class destructor frees memory used to hold the XML tag and
* attribute definitions. It als terminates use of the xerces-C
* framework.
*/
GetConfig::~GetConfig()
{
// Free memory
delete m_ConfigFileParser;
try
{
XMLString::release( &TAG_transmission);
XMLString::release( &TAG_destinationRouter );
XMLString::release( &TAG_destinationPort );
XMLString::release( &TAG_inputRouter );
XMLString::release( &TAG_inputPort );
XMLString::release( &TAG_transBegin );
XMLString::release( &TAG_transDuration );
XMLString::release( &TAG_dataSequence );
XMLString::release( &TAG_data );
XMLString::release( &TAG_link);
```

```
XMLString::release( &TAG_sourceRouter );
XMLString::release( &TAG_sourcePort );
}
catch( ... )
{
cerr << "Unknown exception encountered in TagNamesdtor" << endl;
}
// Terminate Xerces
try
{
XMLPlatformUtils::Terminate(); // Terminate after release of memory
}
catch( xercesc::XMLException& e )
{
char* message = xercesc::XMLString::transcode( e.getMessage() );
cerr << "XML ttolkit teardown error: " << message << endl;
XMLString::release( &message );
}
}
/**
* This function:
* - Tests the access and availability of the XML configuration file.
* - Configures the xerces-c DOM parser.
* - Reads and extracts the pertinent information from the XML config file.
*
* @param in configFile The text string name of the HLA configuration file.
*/
void GetConfig::readNetlist(string& configFile)
throw( std::runtime_error )
{
// Test to see if the file is ok.
int i=0;
struct stat fileStatus;
int iretStat = stat(configFile.c_str(), &fileStatus);
if( iretStat == ENOENT )
throw ( std::runtime_error("Path file_name does not exist, or path is
an empty string.") );
else if( iretStat == ENOTDIR )
throw ( std::runtime_error("A component of the path is not a directory
."));
```

```
else if( iretStat == ELOOP )
throw ( std::runtime_error("Too many symbolic links encountered while
traversing the path."));
else if( iretStat == EACCES )
throw ( std::runtime_error("Permission denied."));
else if( iretStat == ENAMETOOLONG )
throw ( std::runtime_error("File can not be read\n"));
// Configure DOM parser.
m_ConfigFileParser->setValidationScheme( XercesDOMParser::Val_Never );
m_ConfigFileParser->setDoNamespaces( false );
m_ConfigFileParser->setDoSchema( false );
m_ConfigFileParser->setLoadExternalDTD( false );
try
{
m_ConfigFileParser->parse( configFile.c_str() );
// no need to free this pointer - owned by the parent parser object
DOMDocument* xmlDoc = m_ConfigFileParser->getDocument();
// Get the top-level element: NAme is "root". No attributes for "root"
DOMElement* elementRoot = xmlDoc->getDocumentElement();
if( !elementRoot ) throw(std::runtime_error( "empty XML document" ));
// Parse XML file for tags of interest: "ApplicationSettings"
// Look one level nested within "root". (child of root)
DOMNodeList* children = elementRoot->getChildNodes();
const XMLSize_t nodeCount = children->getLength();
// For all nodes, children of "root" in the XML tree.
for( XMLSize_t xx = 0; xx < nodeCount; ++xx )
{
DOMNode* currentNode = children->item(xx);
if( currentNode->getNodeType() && // true is not NULL
currentNode->getNodeType() == DOMNode::ELEMENT_NODE ) // is ele
ment
{
// Found node which is an Element. Re-cast node as element
DOMElement* currentElement = dynamic_cast< xercesc::DOMElement*
>( currentNode );
if( XMLString::equals(currentElement->getTagName(), TAG_link))
{
DOMNodeList* grandchildren = currentElement->getChildNodes()
;
const XMLSize_t nodeCount_b = grandchildren->getLength();
```

```
for( XMLSize_t yy = 0; yy < nodeCount_b; ++yy )
{
DOMNode* currentNodeb = grandchildren->item(yy);
if( currentNodeb->getNodeType() && currentNodeb->ge
tNodeType() == DOMNode::ELEMENT_NODE ) // is element
{
DOMElement* currentElementb = dynamic_cast<
xercesc::DOMElement* >( currentNodeb );
if( XMLString::equals(currentElementb->getTa
gName(), TAG_sourceRouter))
{
const XMLCh* xmlch_sourceRouter = cu
rrentNodeb->getTextContent();
link_array[i].sourceRouter = XMLStri
ng::transcode(xmlch_sourceRouter);
cout << link_array[i].sourceRouter;
};
if( XMLString::equals(currentElementb->getTa
gName(), TAG_sourcePort))
const XMLCh* xmlch_sourcePort = curr
entNodeb->getTextContent();
link_array[i].sourcePort = XMLString
::transcode(xmlch_sourcePort);
cout << link_array[i].sourcePort;
};
if( XMLString::equals(currentElementb->getTa
gName(), TAG_destinationRouter))
{
const XMLCh* xmlch_destinationRouter
= currentNodeb->getTextContent();
link_array[i].destinationRouter = XM
LString::transcode(xmlch_destinationRouter);
cout << link_array[i].destinationRou
ter;
};
if ( XMLString::equals(currentElementb->getT
agName(), TAG_destinationPort))
{
const XMLCh* xmlch_destinationPort =
currentNodeb->getTextContent();
```

```
link_array[i].destinationPort = XMLS
tring::transcode(xmlch_destinationPort);
cout << link_array[i].destinationPort;
};
}
}
i++;
}
}
}
NoOfLinks = i;
cout << endl << "links: " <<NoOfLinks <<endl ;
for (int j=0;j<NoOfLinks;j++)
cout << link_array[j].sourceRouter <<link_array[j].sourcePort<<link_
array[j].destinationRouter<<link_array[j].destinationPort<<endl;
}
catch( xercesc::XMLException& e )
{
char* message = xercesc::XMLString::transcode( e.getMessage() );
ostringstream errBuf;
errBuf << "Error parsing file: " << message << flush;
XMLString::release( &message );
}
}
void GetConfig::readTraffic(string& configFile)
throw( std::runtime_error )
{
// Test to see if the file is ok.
int i=0;
struct stat fileStatus;
int iretStat = stat(configFile.c_str(), &fileStatus);
if( iretStat == ENOENT )
throw ( std::runtime_error("Path file_name does not exist, or path is
an empty string.") );
else if( iretStat == ENOTDIR )
throw ( std::runtime_error("A component of the path is not a directory
."));
else if( iretStat == ELOOP )
throw ( std::runtime_error("Too many symbolic links encountered while
traversing the path."));
```

```
else if( iretStat == EACCES )
throw ( std::runtime_error("Permission denied."));
else if( iretStat == ENAMETOOLONG )
throw ( std::runtime_error("File can not be read\n"));
// Configure DOM parser.
m_ConfigFileParser->setValidationScheme( XercesDOMParser::Val_Never );
m_ConfigFileParser->setDoNamespaces( false );
m_ConfigFileParser->setDoSchema( false );
m_ConfigFileParser->setLoadExternalDTD( false );
try
{
m_ConfigFileParser->parse( configFile.c_str() );
// no need to free this pointer - owned by the parent parser object
DOMDocument* xmlDoc = m_ConfigFileParser->getDocument();
// Get the top-level element: NAme is "root". No attributes for "root"
DOMElement* elementRoot = xmlDoc->getDocumentElement();
if( !elementRoot ) throw(std::runtime_error( "empty XML document" ));
// Parse XML file for tags of interest: "ApplicationSettings"
// Look one level nested within "root". (child of root)
DOMNodeList* children = elementRoot->getChildNodes();
const XMLSize_t nodeCount = children->getLength();
// For all nodes, children of "root" in the XML tree.
for( XMLSize_t xx = 0; xx < nodeCount; ++xx )
{
DOMNode* currentNode = children->item(xx);
if( currentNode->getNodeType() && // true is not NULL
currentNode->getNodeType() == DOMNode::ELEMENT_NODE ) // is ele
ment
{
// Found node which is an Element. Re-cast node as element
DOMElement* currentElement = dynamic_cast< xercesc::DOMElement*
>( currentNode );
if( XMLString::equals(currentElement->getTagName(), TAG_transmis
sion))
{
DOMNodeList* grandchildren = currentElement->getChildNodes()
;
const XMLSize_t nodeCount_b = grandchildren->getLength();
for( XMLSize_t yy = 0; yy < nodeCount_b; ++yy )
{
```

```
DOMNode* currentNodeb = grandchildren->item(yy);
if( currentNodeb->getNodeType() && currentNodeb->ge
tNodeType() == DOMNode::ELEMENT_NODE ) // is element
{
DOMElement* currentElementb = dynamic_cast<
xercesc::DOMElement* >( currentNodeb );
if( XMLString::equals(currentElementb->getTa
gName(), TAG_inputRouter))
{
const XMLCh* xmlch_inputRouter = cur
rentNodeb->getTextContent();
m_inputRouter = XMLString::transcode
(xmlch_inputRouter);
trans_array[i].inputRouter = atoi(m_inputRouter);
}
else if ( XMLString::equals(currentElementb-
>getTagName(), TAG_inputPort))
{
const XMLCh* xmlch_inputPort = curre
ntNodeb->getTextContent();
m_inputPort = XMLString::transcode(x
mlch_inputPort);
trans_array[i].inputPort = atoi(m_inputPort);
}
else if( XMLString::equals(currentElementb->
getTagName(), TAG_transBegin))
{
const XMLCh* xmlch_transBegin = curr
entNodeb->getTextContent();
trans_array[i].transBegin = XMLStrin
g::transcode(xmlch_transBegin);
cout << trans_array[i].transBegin;
}
else if( XMLString::equals(currentElementb->
getTagName(), TAG_transDuration))
{
const XMLCh* xmlch_transDuration = c
urrentNodeb->getTextContent();
trans_array[i].transDuration = XMLSt
ring::transcode(xmlch_transDuration);
```

```
cout << trans_array[i].transDuration
;
}
else if( XMLString::equals(currentElementb->
getTagName(), TAG_dataSequence))
{
DOMNodeList* ggrandchildren = curren
tElementb->getChildNodes();
const XMLSize_t nodeCount_c = ggran
dchildren->getLength();
int j=0;
for( XMLSize_t zz = 0; zz < nodeCoun
t_c; ++zz )
{
DOMNode* currentNodec = ggra
ndchildren->item(zz);
if( currentNodec->getNodeTyp
e() && currentNodec->getNodeType() == DOMNode::ELEMENT_NODE ) // is element
{
DOMElement* currentE
lementc = dynamic_cast< xercesc::DOMElement* >( currentNodec );
if( XMLString::equal
s(currentElementc->getTagName(), TAG_data))
{
const XMLCh*
xmlch_data = currentNodec->getTextContent();
trans_array[
i].data[j] = XMLString::transcode(xmlch_data);
cout << tran
s_array[i].data[j];
j++;
}
}
}
}
}
}
i++;
}
}
```

```
}
NoOfTrans = i;
cout << endl << "trans: " <<NoOfTrans <<endl ;
for (int j=0;j<NoOfTrans;j++)
cout << trans_array[j].inputRouter <<trans_array[j].inputPort<<trans
_array[j].transBegin<<trans_array[j].transDuration<<endl;
}
catch( xercesc::XMLException& e )
{
char* message = xercesc::XMLString::transcode( e.getMessage() );
ostringstream errBuf;
errBuf << "Error parsing file: " << message << flush;
XMLString::release( &message );
}
}
//function print
void GetConfig::print()
{
ofstream myfile;
myfile.open("vhdl/noc.vhd",std::ios::out);
myfile.seekp(0,ios::beg);
ifstream myfile2("vhdl/noc_aux", std::ios::in);
char str[2000];
while (!myfile2.eof())
{
myfile2.getline(str,2000);
myfile << str << endl;
}
myfile2.close();
myfile.seekp(0,ios::end);
//traffic
myfile << "--traffic"<<endl;
int i = 0;
while (i<NoOfTrans)
{
//for each packet
myfile << endl;
//initialisation
myfile << "data_valid_in_help("<< trans_array[i].inputRoute
r<<")("<< trans_array[i].inputPort
```

```
<<")<=data_valid_in_traffic("<<trans_array[i].inputR
outer<<")("
<< trans_array[i].inputPort<<");"<<endl ;
myfile << "data_in_help("<< trans_array[i].inputRouter<<")("
<< trans_array[i].inputPort
<<")<=data_in_traffic("<<trans_array[i].inputRouter<
<")("
<< trans_array[i].inputPort<<");"<<endl ;
i++; //next transmission
//if the next transmission is from the same port as the prev
ious one
while ((i<=NoOfTrans-2) && (trans_array[i].inputRouter == tr
ans_array[i-1].inputRouter) &&
(trans_array[i].inpu
tPort == trans_array[i-1].inputPort))
{
i++;
}
}
myfile.seekp(0,ios::end);
//links
myfile << "--links"<<endl;
for (int i=0;i<NoOfLinks;i++)
{
myfile << "data_valid_in_help(" << link_array[i].destination
Router << ")(" << link_array[i].destinationPort
<<")<=data_valid_out_help(" << link_array[i].sourceR
outer << ")(" <<link_array[i].sourcePort <<");"<< endl;
myfile << "data_in_help(" << link_array[i].destinationRouter
<< ")(" << link_array[i].destinationPort
<<")<=data_out_help(" << link_array[i].sourceRouter
<< ")(" <<link_array[i].sourcePort <<");"<<endl;
myfile << endl;
};
myfile << "end;";
myfile.close();
}
#ifdef MAIN_TEST
/* This main is provided for unit test of the class. */
int main()
```

```
{
string s0,s1,s2,configFile;
s0 = "xml/";
s1 = argv[1];
//parsing tou prwtou
s2 = "/traffic.xml";
configFile= s0 + s1 + s2;
GetConfig appConfig ;
appConfig.readTraffic(configFile);
//parsing tou defterou
s2 = "/netlist.xml";
configFile= s0 + s1 + s2;
appConfig.readNetlist(configFile);
appConfig.print();
return 0;
}
#endif
```

noc_tb_gen.hpp

```
#ifndef XML_PARSER_HPP
#define XML_PARSER_HPP
/**
* @file
* Class "GetConfig" provides the functions to read the XML data.
* @version 1.0
*/
#include <xercesc/dom/DOM.hpp>
#include <xercesc/dom/DOMDocument.hpp>
#include <xercesc/dom/DOMDocumentType.hpp>
#include <xercesc/dom/DOMElement.hpp>
#include <xercesc/dom/DOMImplementation.hpp>
#include <xercesc/dom/DOMImplementationLS.hpp>
#include <xercesc/dom/DOMNodeIterator.hpp>
#include <xercesc/dom/DOMNodeList.hpp>
#include <xercesc/dom/DOMText.hpp>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/util/XMLUni.hpp>
```

```
#include <string>
#include <stdexcept>
#include <algorithm>
#include <math.h>
#include <fstream>
const int TRANS_MAX = 3000; //2*50*10( = 2*MAX_PENUM * MAX_PORTS) antistoix
oun se kathe thira.
const int MAX_NUMBER_OF_FLITS = 10; //number of flits in a packet
const int CLK_PERIOD = 100;
const int FLIT_SIZE_IN_BITS = 16;
// Error codes
enum {
ERROR_ARGS = 1,
ERROR_XERCES_INIT,
ERROR_PARSE,
ERROR_EMPTY_DOCUMENT
};
class GetConfig
{
public:
//constructor
GetConfig();
//destructor
~GetConfig();
//variables
struct trans_struct{int inputRouter; int inputPort ;int transBegin; int t
ransDuration;
char* data[MAX_NUMBER_OF_FLITS];};
trans_struct trans_array[TRANS_MAX];
int NoOfTrans;
int NoOfFlits[TRANS_MAX] ;
//functions
void readTraffic(std::string&) throw(std::runtime_error);
void createAssignmentsArray();
void print();
int GetIntVal(std::string strConvert) {
int intReturn;
// NOTE: You should probably do some checks to ensure that
// this string contains only numbers. If the string is not
// a valid integer, zero will be returned.
```

```
intReturn = atoi(strConvert.c_str());
return(intReturn);
}
private:
xercesc::XercesDOMParser *m_ConfigFileParser;
// Internal class use only. Hold Xerces data in UTF-16 SMLCh type.
XMLCh* TAG_transmission;
XMLCh* TAG_destinationRouter;
XMLCh* TAG_destinationPort;
XMLCh* TAG_inputRouter;
XMLCh* TAG_inputPort;
XMLCh* TAG_transBegin;
XMLCh* TAG_transDuration;
XMLCh* TAG_dataSequence;
XMLCh* TAG_data;
char* m_destinationRouter;
char* m_destinationPort;
char* m_transBegin;
char* m_transDuration;
char* m_inputRouter;
char* m_inputPort;
};
#endif
```

noc_tb_gen.cpp

```
#include <string>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <list>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include "noc_tb_gen.hpp"
using namespace xercesc;
using namespace std;
/**
* Constructor initializes xerces-C libraries.
```

```
* The XML tags and attributes which we seek are defined.
* The xerces-C DOM parser infrastructure is initialized.
*/
GetConfig::GetConfig()
{
try
{
XMLPlatformUtils::Initialize(); // Initialize Xerces infrastructure
}
catch( XMLException& e )
{
char* message = XMLString::transcode( e.getMessage() );
cerr << "XML toolkit initialization error: " << message << endl;
XMLString::release( &message );
// throw exception here to return ERROR_XERCES_INIT
}
// Tags and attributes used in XML file.
// Can't call transcode till after Xerces Initialize()
TAG_transmission = XMLString::transcode("transmission");
TAG_destinationRouter = XMLString::transcode("destinationRouter");
TAG_destinationPort= XMLString::transcode("destinationPort");
TAG_inputRouter = XMLString::transcode("inputRouter");
TAG_inputPort= XMLString::transcode("inputPort");
TAG_transBegin= XMLString::transcode("transBegin");
TAG_transDuration= XMLString::transcode("transDuration");
TAG_dataSequence=XMLString::transcode("dataSequence");
TAG_data=XMLString::transcode("data");
m_ConfigFileParser = new XercesDOMParser;
}
/**
* Class destructor frees memory used to hold the XML tag and
* attribute definitions. It als terminates use of the xerces-C
* framework.
*/
GetConfig::~GetConfig()
{
// Free memory
delete m_ConfigFileParser;
try
{
```

```
XMLString::release( &TAG_transmission);
XMLString::release( &TAG_destinationRouter );
XMLString::release( &TAG_destinationPort );
XMLString::release( &TAG_inputRouter );
XMLString::release( &TAG_inputPort );
```

noc_tb_gen.cpp

```
XMLString::release( &TAG_transBegin );
XMLString::release( &TAG_transDuration );
XMLString::release( &TAG_dataSequence );
XMLString::release( &TAG_data );
}
catch( ... )
{
cerr << "Unknown exception encountered in TagNamesdtor" << endl;
}
// Terminate Xerces
try
{
XMLPlatformUtils::Terminate(); // Terminate after release of memory
}
catch( xercesc::XMLException& e )
{
char* message = xercesc::XMLString::transcode( e.getMessage() );
cerr << "XML ttolkit teardown error: " << message << endl;
XMLString::release( &message );
}
}
void GetConfig::readTraffic(string& configFile)
throw( std::runtime_error )
{
// Test to see if the file is ok.
int i=0;
struct stat fileStatus;
int iretStat = stat(configFile.c_str(), &fileStatus);
if( iretStat == ENOENT )
throw ( std::runtime_error("Path file_name does not exist, or path is
an empty string.") );
else if( iretStat == ENOTDIR )
throw ( std::runtime_error("A component of the path is not a directory
."));
```

```
else if( iretStat == ELOOP )
throw ( std::runtime_error("Too many symbolic links encountered while
traversing the path."));
else if( iretStat == EACCES )
throw ( std::runtime_error("Permission denied."));
else if( iretStat == ENAMETOOLONG )
throw ( std::runtime_error("File can not be read\n"));
// Configure DOM parser.
m_ConfigFileParser->setValidationScheme( XercesDOMParser::Val_Never );
m_ConfigFileParser->setDoNamespaces( false );
m_ConfigFileParser->setDoSchema( false );
m_ConfigFileParser->setLoadExternalDTD( false );
try
{
m_ConfigFileParser->parse( configFile.c_str() );
// no need to free this pointer - owned by the parent parser object
DOMDocument* xmlDoc = m_ConfigFileParser->getDocument();
// Get the top-level element: NAme is "root". No attributes for "root"
DOMElement* elementRoot = xmlDoc->getDocumentElement();
if( !elementRoot ) throw(std::runtime_error( "empty XML document" ));
// Parse XML file for tags of interest: "ApplicationSettings"
// Look one level nested within "root". (child of root)
DOMNodeList* children = elementRoot->getChildNodes();
const XMLSize_t nodeCount = children->getLength();
// For all nodes, children of "root" in the XML tree.
for( XMLSize_t xx = 0; xx < nodeCount; ++xx )
{
DOMNode* currentNode = children->item(xx);
if( currentNode->getNodeType() && // true is not NULL
currentNode->getNodeType() == DOMNode::ELEMENT_NODE ) // is ele
ment
{
// Found node which is an Element. Re-cast node as element
DOMElement* currentElement = dynamic_cast< xercesc::DOMElement*
>( currentNode );
if( XMLString::equals(currentElement->getTagName(), TAG_transmis
sion))
{
DOMNodeList* grandchildren = currentElement->getChildNodes()
;
```

```
const XMLSize_t nodeCount_b = grandchildren->getLength();
for( XMLSize_t yy = 0; yy < nodeCount_b; ++yy )
{
DOMNode* currentNodeb = grandchildren->item(yy);
if( currentNodeb->getNodeType() && currentNodeb->ge
tNodeType() == DOMNode::ELEMENT_NODE ) // is element
{
DOMElement* currentElementb = dynamic_cast<
xercesc::DOMElement* >( currentNodeb );
if( XMLString::equals(currentElementb->getTa
gName(), TAG_inputRouter))
{
const XMLCh* xmlch_inputRouter = cur
rentNodeb->getTextContent();
m_inputRouter = XMLString::transcode
(xmlch_inputRouter);
trans_array[i].inputRouter = atoi(m_inputRouter);
}
else if ( XMLString::equals(currentElementb-
>getTagName(), TAG_inputPort))
{
const XMLCh* xmlch_inputPort = curre
ntNodeb->getTextContent();
m_inputPort = XMLString::transcode(x
mlch_inputPort);
trans_array[i].inputPort = atoi(m_inputPort);
}
else if( XMLString::equals(currentElementb->
getTagName(), TAG_transBegin))
{
const XMLCh* xmlch_transBegin = curr
entNodeb->getTextContent();
m_transBegin= XMLString::transcode(
xmlch_transBegin);
trans_array[i].transBegin = atoi(m_
transBegin);
}
else if( XMLString::equals(currentElementb->
getTagName(), TAG_transDuration))
{
```

```
const XMLCh* xmlch_transDuration = c
urrentNodeb->getTextContent();
m_transDuration= XMLString::transcod
e(xmlch_transDuration);
trans_array[i].transDuration= atoi(m
_transDuration);
}
else if( XMLString::equals(currentElementb->
getTagName(), TAG_dataSequence))
{
DOMNodeList* ggrandchildren = curren
tElementb->getChildNodes();
const XMLSize_t nodeCount_c = ggran
dchildren->getLength();
int j=0;
for( XMLSize_t zz = 0; zz < nodeCoun
t_c; ++zz )
{
DOMNode* currentNodec = ggra
ndchildren->item(zz);
if( currentNodec->getNodeTyp
e() && currentNodec->getNodeType() == DOMNode::ELEMENT_NODE ) // is element
{
DOMElement* currentE
lementc = dynamic_cast< xercesc::DOMElement* >( currentNodec );
if( XMLString::equal
s(currentElementc->getTagName(), TAG_data))
{
const XMLCh*
xmlch_data = currentNodec->getTextContent();
trans_array[
i].data[j] = XMLString::transcode(xmlch_data);
j++;
}
}
}
NoOfFlits[i]=j;
}
}
}
```

```
i++;
}
}
}
NoOfTrans = i;
cout << endl << "trans: " <<NoOfTrans <<endl ;
for (int k=0;k<NoOfTrans;k++){
cout << trans_array[k].inputRouter << " "<<trans_array[k].inputPort<
<" "<<trans_array[k].transBegin<<trans_array[k].transDuration<<" ";
for (int l=0;l<NoOfFlits[k];l++)
cout << trans_array[k].data[l] <<"";
cout << endl;
}
}
catch( xercesc::XMLException& e )
{
char* message = xercesc::XMLString::transcode( e.getMessage() );
ostringstream errBuf;
errBuf << "Error parsing file: " << message << flush;
XMLString::release( &message );
}
}
//function print
void GetConfig::print()
{
//write the standard content
ofstream myfile;
myfile.open("vhdl/noc_tb.vhd",std::ios::out);
myfile.seekp(0,ios::beg);
ifstream myfile2("vhdl/noc_tb_aux", std::ios::in);
char str[2000];
int line=0;
while (!myfile2.eof())
{
++line;
if (line != 56)
{
myfile2.getline(str,2000);
myfile << str << endl;
}
```

```
else
{
//write the traffic values
//data_valid_in_tranffic
int i=0;
while (i<NoOfTrans)
{
//for each packet
myfile << endl;
//initialisation
myfile << "data_valid_in_traffic("<< trans_a
rray[i].inputRouter<<")("<< trans_array[i].inputPort <<")<='0',"
//first packet
<<"'1' after "<<trans_array[i].trans
Begin*CLK_PERIOD+CLK_PERIOD/2 <<" ns,"
<<"'0' after "<< trans_array[i].tran
sBegin*CLK_PERIOD+trans_array[i].transDuration*CLK_PERIOD
+CLK_PERIOD/2<< " ns";
i++; //next transmission
//if the next transmission is from the same
port as the previous one
while ((i<=NoOfTrans-1) && (trans_array[i].i
nputRouter == trans_array[i-1].inputRouter) &&
(trans_array[i].inpu
tPort == trans_array[i-1].inputPort))
{
myfile <<",'1' after "<< trans_arra
y[i].transBegin*CLK_PERIOD+CLK_PERIOD/2 <<" ns,"
<<"'0' after " << trans_arra
y[i].transBegin*CLK_PERIOD
+trans_arr
ay[i].transDuration*CLK_PERIOD+CLK_PERIOD/2 << " ns";
i++;
}
myfile << ";";
}
//data_in_traffic
i=0;
while (i<NoOfTrans)
{
```

```
myfile << endl;
myfile << "data_in_traffic("<< trans_array[i
].inputRouter<<")("<< trans_array[i].inputPort <<")<=\"";
//initialisation
for (int k=0;k<FLIT_SIZE_IN_BITS;k++)
myfile << "0";
myfile << "\"";
//packet
//for each flit
for (int j=0;j<NoOfFlits[i];j++)
{
myfile <<",\""<< trans_array[i].data
[j] <<"\" after "
<< trans_array[i].transBegin*
CLK_PERIOD+(CLK_PERIOD*j)+CLK_PERIOD/2 <<" ns";
}
myfile << ",\"";
//packet end
for (int k=0;k<FLIT_SIZE_IN_BITS;k++)
myfile << "0";
myfile << "\" after "
<< (trans_array[i].transBegin*
CLK_PERIOD)+(CLK_PERIOD*trans_array[i].transDuration)+CLK_PERIOD/2
<<" ns" ;
i++;
//if another packet for the same port
while ((i<=NoOfTrans-1) && (trans_array[i].
inputRouter == trans_array[i-1].inputRouter) &&
(trans_array[i].inputPort == t
rans_array[i-1].inputPort))
{
//for each flit
for (int j=0;j<NoOfFlits[i];j++)
{
myfile << ",\"" << trans_arr
ay[i].data[j] <<"\" after "
<< trans_array[i].tr
ansBegin*CLK_PERIOD+(CLK_PERIOD*j)+CLK_PERIOD/2 <<" ns";
}
myfile << ",\"";
```

```
//packet end
for (int k=0;k<FLIT_SIZE_IN_BITS;k++
)
myfile << "0";
myfile << "\" after "
<< (trans_array[i].transBegin
*CLK_PERIOD)+(CLK_PERIOD*trans_array[i].transDuration)
+CLK_PERIOD/2<<" ns" ;
i++;
}
myfile << ";";
}
}
}
myfile2.close();
myfile.close();
}
#ifdef MAIN_TEST
/* This main is provided for unit test of the class. */
int main(int argc, char *argv[])
{
string s0 = "xml/";
string s1 = argv[1];
string s2 = "/traffic.xml";
string configFile= s0 + s1 + s2; // stat file. Get ambigious segfault oth
erwise.
GetConfig* appConfig = new GetConfig;
appConfig->readTraffic(configFile);
appConfig->print();
delete appConfig;
return 0;
}
#endif
```