



Norwegian University of  
Science and Technology

# Network on Chip for FPGA

Development of a test system for Network on Chip

**Magnus Krokum Namork**

Master of Science in Electronics

Submission date: June 2011

Supervisor: Kjetil Svarstad, IET



# Problem Description

This assignment is a continuation of the project-assignment of fall 2010, where it was looked into the development of reactive modules for application-test and profiling of the Network on Chip realization. It will especially be focused on the further development of:

- The programmability of the system by developing functionality for more advanced surveillance of the communication between modules and routers
- Framework that will be used to test and profile entire applications on the Network on Chip

The work will primarily be directed towards testing and running the system in a way that resembles a real system at run time. The work is to be compared with relevant research within similar work.

Assignment given:            January 2011  
Supervisor:                    Kjetil Svarstad, IET

## Abstract

Testing and verification of digital systems is an essential part of product development. The Network on Chip (NoC), as a new paradigm within interconnections; has a specific need for testing. This is to determine how performance and properties of the NoC are compared to the requirements of different systems such as processors or media applications.

A NoC has been developed within the AHEAD project to form a basis for a reconfigurable platform used in the AHEAD system. This report gives an outline of the project to develop testing and benchmarking systems for a NoC. The specific work has been regarding the development of a generic module connected to the NoC and capability of testing the NoCs' properties. The test system was initiated by Ivar Ersland in 2009 and developed further by Andreas Hepsø, and Magnus Namork in the fall of 2010. The functionality and systems that are implemented are the following:

- Fully functional Hardware/Software interface which defines communication between NoC and the user
- Reactive system which responds to interaction based on package information
- MPEG example system that mimics an MPEG data stream
- Software reconfiguration of the traffic tables by sending specific packages to the system
- Cell processor example application to test simple computation and communicating modules on the network

The systems have been tested successfully, verified and implemented on a Xilinx Spartan FPGA. It has also been developed a software system written in C to read and interpret data from the Network in on-chip tests. In total these implementations have been the foundation of building a benchmarking platform for the NoC.



# Preface

*This assignment is written at the Institute of Electronics and Telecommunications (IET), NTNU in the spring of 2011. It has been done under the guidance of Professor Kjetil Svarstad and as a continuation of the AHEAD Network on Chip project. The assignment was chosen based on its practical and experimental nature, which has also been the main challenge for this project. I wish to thank Kjetil Svarstad for supervision and help during the entire process.*



# Contents

<b>1</b>	<b>The AHEAD project</b>	<b>1</b>
1.1	Reconfigurable systems . . . . .	1
1.2	Concept . . . . .	2
1.3	Areas of focus . . . . .	3
1.4	Network on Chip(NoC) . . . . .	4
1.4.1	Basic concept . . . . .	4
1.4.2	Network on Chip in reconfigurable systems . . . . .	5
1.4.3	Testing of digital circuits . . . . .	5
1.4.4	Profiling of applications . . . . .	6
1.5	Testing the NoC . . . . .	6
1.5.1	Experimental working method . . . . .	7
<b>2</b>	<b>Theory and research</b>	<b>9</b>
2.1	Design for test . . . . .	9
2.2	Benchmarking . . . . .	10
2.2.1	System properties . . . . .	11
2.2.2	Benchmarking protocol . . . . .	12
2.2.3	NoC benchmarking research . . . . .	12
2.3	Deadlocks . . . . .	12
2.4	Clustering . . . . .	13
2.5	System platforms . . . . .	14
<b>3</b>	<b>Previous work</b>	<b>17</b>
3.1	Functionality . . . . .	17
3.2	Design of the AHEAD Network on Chip . . . . .	19



3.2.1	Test system . . . . .	20
3.2.2	Reactive test modules . . . . .	21
3.3	Initial status of the system . . . . .	21
3.3.1	System properties . . . . .	22
3.4	Test system motivation . . . . .	22
<b>4</b>	<b>Development of the system</b>	<b>25</b>
4.1	Structural development . . . . .	25
4.1.1	Generic code development . . . . .	25
4.1.2	Hardware software interfacing (HW/SW) . . . . .	26
4.1.3	Other structural changes . . . . .	27
4.2	Functional development . . . . .	28
4.2.1	Specifications . . . . .	28
4.3	Application example:MPEG decoder . . . . .	30
4.3.1	Real time requirements . . . . .	31
4.4	Application example:PS3 ring bus system . . . . .	31
4.5	Other possible application examples . . . . .	33
4.5.1	RSA encryption . . . . .	33
<b>5</b>	<b>Requirements and design</b>	<b>35</b>
5.1	Design goals . . . . .	35
5.1.1	Area . . . . .	35
5.1.2	Flexible Design . . . . .	36
5.2	Hardware/Software interface . . . . .	37
5.2.1	Software accessible registers . . . . .	37
5.2.2	Hardware interfacing . . . . .	39
5.3	Traffic generator . . . . .	39
5.3.1	Sending and receiving packages with traffic tables . . . . .	40
5.4	Traffic pattern in the NoC . . . . .	40
5.4.1	Motivation . . . . .	40
5.4.2	Sendback pattern . . . . .	41
5.4.3	MPEG pattern . . . . .	42
5.4.4	Internal Design . . . . .	44

5.5	Multitasking test generator; Cell application . . . . .	44
5.5.1	Operating modes . . . . .	44
5.5.2	Data Manipulation . . . . .	47
5.6	Software based test program . . . . .	48
5.6.1	Redefining behaviour through software . . . . .	50
5.7	Surveillance and monitoring of traffic . . . . .	50
5.7.1	Measuring throughput . . . . .	51
<b>6</b>	<b>Verification and testing</b>	<b>53</b>
6.1	Simulation . . . . .	53
6.1.1	Simulation of basic functionality . . . . .	54
6.1.2	Simple traffic pattern simulation . . . . .	54
6.1.3	Patterns with crossing traffic and deadlocks . . . . .	55
6.2	Synthesis of the circuits . . . . .	56
6.2.1	Synthesis challenges . . . . .	56
6.2.2	Synthesis of the circuit . . . . .	57
6.3	FPGA implementation . . . . .	58
6.3.1	Initial testing . . . . .	58
6.3.2	Testing patterns, initial MPEG test . . . . .	58
6.3.3	Higher packet rate . . . . .	60
6.4	Cell processor application . . . . .	60
6.4.1	Streaming test of the application . . . . .	67
6.5	Summary of testing . . . . .	67
<b>7</b>	<b>Discussion</b>	<b>71</b>
7.1	Evaluation of the system . . . . .	71
7.1.1	MPEG example . . . . .	72
7.1.2	Cell example . . . . .	72
7.1.3	Interface as bottleneck . . . . .	73
7.2	Throughput results . . . . .	73
7.2.1	MPEG . . . . .	74
7.2.2	Cell application . . . . .	75
7.2.3	Stream test . . . . .	76

7.3	Application module placement . . . . .	77
7.4	Further use of the test system . . . . .	78
<b>8</b>	<b>Conclusion</b>	<b>81</b>
<b>9</b>	<b>Further work</b>	<b>83</b>
<b>A</b>	<b>Illustrations of the systems</b>	<b>89</b>
<b>B</b>	<b>Code</b>	<b>95</b>
B.1	VHDL code . . . . .	95
B.2	C-code . . . . .	114
<b>C</b>	<b>AHEAD Network on Chip-Initial words</b>	<b>127</b>
C.1	Equipment list for this project . . . . .	128
<b>D</b>	<b>Tutorial:How to implement the Network on Chip on the Suzaku-S platform</b>	<b>129</b>
D.1	Installing Xilinx in debian(Atmark Development environment(Atde3) or Ubuntu) . . . . .	129
D.2	VHDL code for the Suzaku Image(Peripheral or IP) . . . . .	130
D.2.1	Using AHDL for development . . . . .	130
D.2.2	Setting up the project in Xilinx EDK . . . . .	130
D.2.3	Synthesis . . . . .	132
D.2.4	Exporting project from EDK to ISE . . . . .	132
D.2.5	Interfacing HW/SW . . . . .	134
D.3	Downloading the generated bit file to the suzaku board . . . . .	134
D.3.1	With serial interface . . . . .	134
D.3.2	With Ethernet . . . . .	135
D.4	Compiling and creating the uCLinux image . . . . .	136
D.4.1	Known errors and solutions . . . . .	136
D.4.2	NFS . . . . .	137
D.4.3	Setting static IP . . . . .	137
D.5	Sources of error . . . . .	138
D.6	File list NoC . . . . .	138

# List of Abbreviations

AHEAD	Ambient Hardware, Embedded Architectures on Demand, page 2
ASIC	Application Specific Integrated Circuit, page 1
CLB	Configurable Logic Block, page 14
EIB	Element Interconnect Bus, page 31
FPGA	Field Programmable Gate Array, page 1
FSM	Finite State Machine, page 39
GPP	General Purpose Processor, page 1
HW	Hardware, page 1
LUT	Look up table, page 14
MIC	Memory Interface Controller, page 31
MISD	Multiple Input Single Destination, page 48
NFS	Network File System, page 27
NoC	Network on Chip, page 4
PE	Processing Element, page 48
PPE	Power Processing Element, page 31
SPE	Synergistic Processing Element, page 31
SW	Software, page 1
TG	Test Generator, page 19
USB	Universal Serial Bus, page 3

## Router table

*The routers are an essential part of the Network on Chip, but their names vary due to previous version implementation and practical use of signal names. Provided is a table of the equivalent names of the routers used in this assignment both in the text and the code:*

Router	Decimal	Binary	Hexadecimal
00	0	0000	0
01	1	0001	1
02	2	0010	2
03	3	0011	3
10	4	0100	4
11	5	0101	5
12	6	0110	6
13	7	0111	7
20	8	1000	8
21	9	1001	9
22	10	1010	A
23	11	1011	B
30	12	1100	C
31	13	1101	D
32	14	1110	E
33	15	1111	F

# List of Figures

1.1	AHEAD illustration . . . . .	2
1.2	AHEAD concept . . . . .	3
1.3	Network on Chip . . . . .	4
1.4	Design and working methodology. . . . .	8
2.1	Deadlock situation . . . . .	13
2.2	Suzaku boards . . . . .	15
3.1	Network on Chip structure . . . . .	18
3.2	Illustration of handshaking. . . . .	20
4.1	HW/SW interface vector handling. . . . .	28
4.2	MPEG traffic pattern . . . . .	30
4.3	Cell processor . . . . .	32
4.4	Cell processor in NoC . . . . .	32
5.1	Input state machine of the Traffic generator, MPEG configuration. . . . .	43
5.2	Input state machine of the Traffic generator, Cell example application. . . . .	45
5.3	Table reconfiguring through packages . . . . .	47
5.4	ALternatives FPGA test . . . . .	49
6.1	Basic simulation . . . . .	54
6.2	MPEG simulation . . . . .	55
6.3	Structural test of the FPGA . . . . .	59
6.4	Structural test FPGA . . . . .	59
6.5	Simple MPEG FPGA . . . . .	60
6.6	MPEG 100 packages simulation . . . . .	61

6.7	Software program test MPEG pattern . . . . .	61
6.8	Traffic Cell example, FPGA test . . . . .	62
6.9	Simulation table reconfiguration . . . . .	62
6.10	Simulation average, one package . . . . .	63
6.11	Calculation average FPGA Cell . . . . .	63
6.12	Graph of latency . . . . .	64
6.13	Graph of latency, second table configuration . . . . .	65
6.14	Stream tests . . . . .	69
7.1	Two interface modules . . . . .	74
A.1	Modules presented in hierarchy. . . . .	90
A.2	Test generator block. . . . .	91
A.3	Floorplan NoC . . . . .	92
A.4	Floorplan NoC description . . . . .	93
D.1	Library creation AHDL . . . . .	131
D.2	System Assembly EDK . . . . .	133

# List of Tables

2.1	FPGA data . . . . .	14
4.1	Properties Cell . . . . .	31
5.1	Address listing . . . . .	38
5.2	Register numbers and functionality. . . . .	38
5.3	Traffic table . . . . .	41
5.4	MPEG protocol . . . . .	42
5.5	Multi functional protocol . . . . .	46
5.6	change_data . . . . .	47
5.7	Average value input . . . . .	48
5.8	Cell application example table. . . . .	49
6.1	Synthesis MPEG system . . . . .	57
6.2	Synthesis TG Cell-example . . . . .	58
6.3	Test load . . . . .	66
6.4	Cell application example table. . . . .	66





# Chapter 1

## The AHEAD project

*This chapter is an introduction to the AHEAD project and gives a description of some important terms associated with it. It also describes how this report is structured and how the work has been performed.*

### 1.1 Reconfigurable systems

Development of electronic systems is in constant change and new and improved systems and algorithms are developed at a high rate. There are several ways of implementing these types of systems; one is the Application Specific Integrated Circuit (ASIC). ASICs are hard wired circuits that performs a given task, but with no or limited possibility to perform a different task after production. The second one is the general purpose processor (GPP) which performs software (SW) tasks by dividing it into predefined operations which are run on the processor. The first instance gives high speed and one efficient solution, the latter gives flexibility and the ability to do various different tasks.

Reconfigurable systems falls into the category that is between the two mentioned platforms. In these systems hardware descriptions are loaded into a platform i.e. an FPGA and the tasks are solved in hardware (HW). However; in contrast to the ASIC, Field Programmable Gate Array (FPGA) systems are completely reconfigurable, hence, it achieves flexibility not provided by any ASIC. In addition, it provides more speed to a system than what is the case with the proces-

sor. The FPGA is the foundation of the AHEAD project and adds the possibility of having extra computational power in order to serve any requirements of the system user.

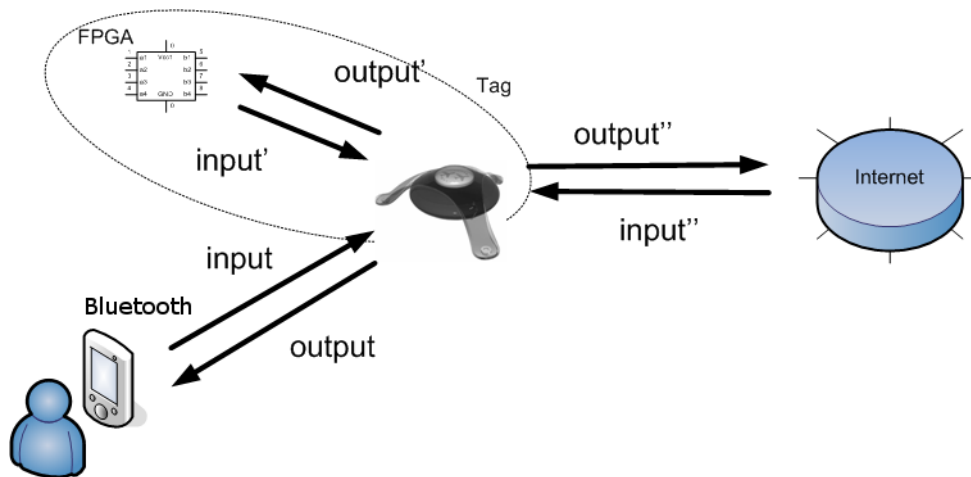
## 1.2 Concept

*Ambient Hardware, Embedded Architectures on Demand* (AHEAD) [9] is a project which was started in 2006 by Professor Kjetil Svarstad at NTNU, Department of Electronics and Telecommunications. It is based on Ambient Intelligence which is a concept describing an environment of devices which is sensitive to people. The idea is based on portable modules, i.e. PDAs or cell phones, and tags with extra computational power located within the environment of the modules [31]. This could, for instance, be an airport or a bus stop where a tag is located, as illustrated in Figure 1.1AHEAD illustrationfigure.caption.9, that detects and interacts with the portable device carried by the user when entering the environment of the tag.



**Figure 1.1:** General outline of the AHEAD concept; Tag and user with PDA [31].

This way of operation is intended to provide extra computational power for small mobile devices with limited standalone processing capabilities. It is achieved by having HW architecture descriptions located on the portable device. A wireless protocol like Bluetooth is used for communication between the portable device and



**Figure 1.2:** General outline of the AHEAD concept; Connection between user and tag, and tag and internet.

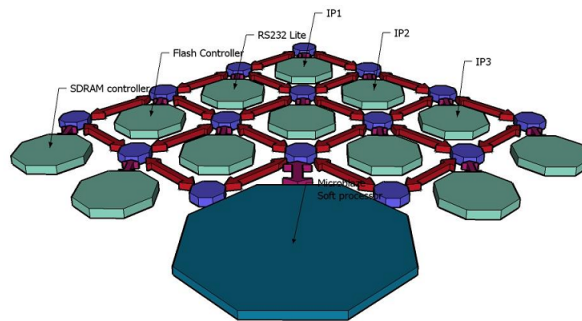
the tag on the wall. Then the portable device instantiates the architecture specific for the task on a reconfigurable co-processor, for instance an FPGA, within the tag as illustrated in Figure 1.2. The task is then run on the FPGA instead of the small processor of the user's mobile device.

### 1.3 Areas of focus

The AHEAD project has several different areas of focus. Initially external communication was investigated with assignments on USB, Bluetooth and serial communication. For the past two years there has been a focus on the Network on Chip and the self-reconfiguration and run time reconfiguration of the system. The Network on Chip provides a framework and platform for the reconfiguration and simplifies this operation while the reconfiguration provides an adaptive approach for the AHEAD system.

## 1.4 Network on Chip(NoC)

Interconnections in System on Chip(SoC)<sup>1</sup> systems have traditionally been bus based or point to point communication architectures. Some examples includes, cross bar buses and ring buses [28].However, a new paradigm within this topic is emerging and this is called Network on Chip [24]. NoC has as its goal to be a a scalable interconnection between modules, and separate the communication from computation in SoCs.



**Figure 1.3:** Network on Chip illustration [23].Shows a system with IP cores, interface modules and processor connected with an NoC and its routers.

### 1.4.1 Basic concept

Different definitions have been used to describe Networks on Chip. Predominantly, it is an interconnection system to handle communication between elements on a chip, with the usage of routers organized in i.e. a mesh topology. Its has a large resemblance to regular telecommunications systems. The concept is simply to use switching techniques to send digital packages between routers and connect modules to these routers [15]. This improves the scalability of the system and the possibility to use the system in reconfiguration because it is possible to change subparts of the system without having to change the entire system and its interconnection. This is especially interesting when partial reconfiguration is a highly demanded property.

---

<sup>1</sup>All parts of a system integrated on one chip

### 1.4.2 Network on Chip in reconfigurable systems

As mentioned, the self reconfigurable part of the AHEAD project and the Network on Chip might work together in the future. There has already been some research in the area regarding Network on Chip systems. For instance, Bobda et al. in [11], describes a system where modules are placed and connected to an NoC in different sized areas on an FPGA. Since the routers of the NoC already defines the communication between the modules it only has to adapt to the protocol used in the network to start operating. Hence, no new interconnection between the existing modules and the new module has to be added. This also provides a good platform for partial reconfiguration. If one has a method to locate a specific module connected to the network, and then in turn change only that module, the rest of the system can remain untouched. This is desirable for instance when optimizing a SoC or the new requirement of the SoC is only partially different from the previously implemented one.

### 1.4.3 Testing of digital circuits

In production of modern electronic systems, testing is a vital part of the development process. The formal definition of testing is

A test is feasible if a known set of input vectors can be applied to a circuit in a known state resulting in a response that may be compared to an expected known response

Knowing that all the modules are in place and function as intended is critical when completing a design process. This is to verify that the system will exhibit desired behaviour. Several test-methodologies exists in order to obtain this goal.

For ASICs the verification of signals and registers by using physical tests and fault models are applied. To a post fabrication FPGA test, functional tests and structural tests are more relevant. The formal definition of functional test is;

Testing that the circuit is functioning correctly using functional vectors  
and the definition of structural test is;

Testing that all the components and connections are present using special test vectors. [8]

These two test models give the relevant information when designing FPGA systems, namely is the system complete and will the output be as expected.

#### 1.4.4 Profiling of applications

A connected part to the testing of a system is the profiling of applications. If one wants to use a platform in a practical setting it is important to know what applications that are possible to implement on it. The ability to run a simple test that provides the information about how a more complex system will behave is then beneficial. Hence, profiling is merely concentrated around the possible placement of the application in for instance an NoC. This profiling could be done by mimicking its communication and use of the same resources within the system.

### 1.5 Testing the NoC

For this project it has been a primary goal *to develop a system that can be used to mimic, test and profile an NoC implemented system*. It has been a main focus to develop examples that mimics a streaming application and a small processor. In addition the environment around with communication with the NoC and surveillance of package data has been emphasized. These elements assembled forms a platform for a benchmark that is capable of measuring the performance of the network. The system also provides information about efficient placement of an implemented system on the NoC.

The assignment is connected to the NoC system developed for the AHEAD project by several participants since 2006. In this paper there will first be a brief description of some background and theory behind NoC testing and benchmarking in Chapter 2 Theory and research chapter.2. In Chapter 3 Previous work chapter.3 a brief description and outline of the system status before commencing is presented. Then it moves over to how the system development is performed in Chapter 4 Development of the system chapter.4. Further it describes how this is implemented

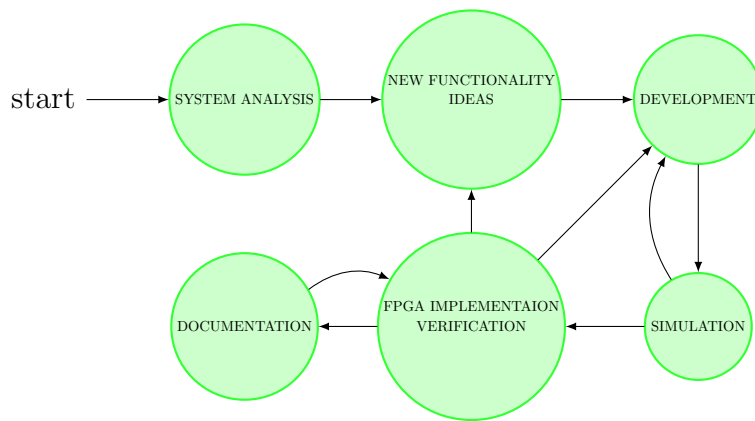
in general. And then what requirements are connected to the developed system in Chapter 5 Requirements and designchapter.5. In Chapter 6 Verification and testingchapter.6 the verification of the system is described along with details on simulation, synthesis and on-chip test-results. Finally the system and its results from testing are discussed, and important results are evaluated in Chapter 7 Discussionchapter.7, before these discussions are concluded with in Chapter 8 Conclusionchapter.8.

### 1.5.1 Experimental working method

When developing an implementation of a system without any algorithms or specification, it is necessary with an experimental approach to complete the work. In this assignment, development has been conducted by brainstorming, experimentation followed by testing to verify how new elements have applied to the existing system. The system not has been out of the box to use, hence, adding new functionality has been followed both by extensive pre implementation simulation and testing. Both to develop new functionality, but also to understand how the system works. Adapting the contribution one step at a time in the implementation stage has been a critical factor.

The challenges and solutions have been separated into different parts: *Analysis* of the system and its functionality. Under the analysis, *detecting* problem areas and the possibility of implementing new ideas and removing redundant functionality correct. Based on this analysis, *new functionality and ideas* for the system has been planned. Then *testing and verification* in simulation, and on chip, followed by *documentation* of the results has been performed. When one iteration has been run, the next iteration begins based on the previous one. In this way the previous iteration forms the platform for the next one and so forth. This methodology is depicted in Figure 1.4 Design and working methodology.figure.caption.12





**Figure 1.4:** Design and working methodology.

# Chapter 2

## Theory and research

*The Network on chip (NoC) is a rather new paradigm within the world of electronic systems. Hence little practical test-cases are commonly known, and those known to exist are mainly academic. The research done by manufacturing companies are to some extent confidential and thus not publicly available. In this chapter there will be a presentation of the theory used to compare the effect of the testing in our network on chip. Some important terms and their use for the AHEAD NoC are also included.*

### 2.1 Design for test

The main theme of this project has initially been to create a design eligible for on-chip testing. Design For Test (DFT) is one well known method to complete this task. The basis is to create a test written in for instance VHDL or C and apply it to see if the system responds as expected. In the project mentioned this forms the basis for the structural testing of the system. It is applied to verify that the circuit is correctly assembled by controlling that it gives output, when applied a known input. The other form of testing is the functional testing which aims towards verification of the functionality of the implemented system. An example of a DFT applied to a NoC is described in depth in [32].

## 2.2 Benchmarking

The ability to measure the performance of a system is essential. Benchmarking of general purpose processors has been a well-known area for many years, and has also been introduced as a way of measuring the performance of NoCs. Benchmarks are models or programs of known input that resembles or simulates a real application behaviour and measures its performance [14]. The problem with NoC architectures is that they are not, compared to general purpose processors, fixed architectures. This implies that in order to test the network it is necessary not only to specify the program code to run on the platform, but also the platform itself in order to test the network. The benchmarking of NoCs is in that way a more complex operation than benchmarking a general purpose processor. There exists a research group which has this as its main focus. The NoC Benchmarking Work group with Grecu et al. has described the benchmarking of NoCs and some proposed parameters of this process. They lists some properties a NoC benchmark should test in an attempt to reach an open standard for Network on chip benchmarks [14, p.6].

- Network size (small,medium,large)
- IP core composition (amount of processing, memory cores,other)
- Topology(regular,irregular)
- Traffic characteristics(spatial and temporal)
- QoS requirements(best effort,guaranteed bandwidth,guaranteed latency)

These are properties that directly relates to the AHEAD NoC project. However, the main focus of the testing is related to the traffic characterization and Quality of Service requirements, as initiated by Ersland in [18]. Under this area, the latency of packages and actual throughput in testing are main areas since they in many cases are the most interesting metrics [14, p.5]. When only one or few of these properties are tested, the benchmark is called a micro-benchmark [14, p.2].

### 2.2.1 System properties

There are some metrics that will be of particular importance, to get an impression of the performance. Following is a description and definition of some of the most important for the benchmarking of the AHEAD NoC.

**Latency** Latency is defined as the delay time from one point to another within a computer system. The end-to-end latency is the time it takes for a package to enter the NoC until an output arrives. In a typical microprocessing system like i.e. Cell [10, p.9], latency is defined as:

$$\textit{Latency} = \text{sending overhead} + \text{time of flight} + \text{transmission time} + \text{receiver overhead} \quad (2.1)$$

The latency could be given in either seconds or to measure it in number of clock cycles. The Cell microprocessor will be described in further detail in Chapter 4 Development of the system chapter.4

**Throughput** Throughput is a metric used to define the amount of data passing through a communication channel such as a bus. The throughput is given in bits per second (bps). The definition of the throughput in [10] defines the throughput as the amount of information being transferred over a time interval. When relating this to a package based system, the throughput would be practical to define as the amount of packages being successfully transferred. This is the definition that will be used in this assignment. Throughput is highly frequency dependent as it is measured over a given timespan.

**Bandwidth** The bandwidth of a system is given as the amount of data transmitted over a given time through a system. It is very similar to the term throughput, but refers often to the maximum information-carrying capacity of a line or a network [21]. The bandwidth is also denoted by bits per second. The relation between the throughput and bandwidth is that bandwidth is the maximum and throughput is the actual speed of the transferred data [26]. In this aspect the bandwidth is generally the theoretically achievable bit per second transfer rate, while throughput

is what is actually being transmitted. Bandwidth is, similar to the throughput, highly dependent of the frequency.

### 2.2.2 Benchmarking protocol

Any benchmark or test system applied to the NoC must have a defined communication protocol. The word protocol means "codes of correct conduct" and this is what it is for the modules connected in the system. Each module has to comply with the protocol to perform its task of communicating with the other parts of the system. An example of a suitable protocol for NoCs is the Open Core Protocol [14, p.10]. This is an open source core-centric protocol applicable to the NoC and defines a set of functionality in point to point communication between modules in a system on chip. The advantage of this protocol is the open nature of it that enables high re-usability of intellectual property by having an open interface description available for designers [1].

### 2.2.3 NoC benchmarking research

Several research groups have investigated NoCs and benchmarking of them. Salmi-nen et al describes some of the requirements connected to benchmarking in [30]. In this paper it is a focus on the open nature of benchmarking to enable comparison between different types of NoCs. As previously mentioned, a group has been formed, NoC Benchmarking Workgroup, with special emphasis on this open nature of the benchmarking and has taken the ideas further. This work group and their milestones is presented in further details in [16].

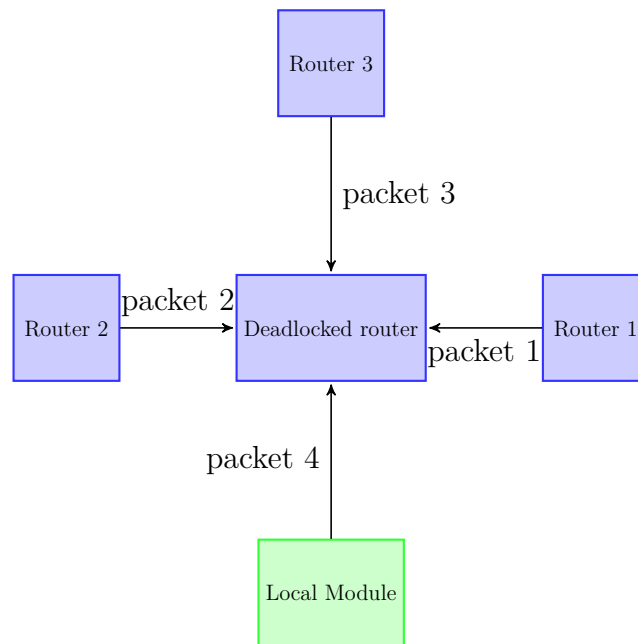
## 2.3 Deadlocks

One major problem with shared resource<sup>1</sup> systems such as the NoC is deadlocks [3, p.66]. This is a state where two or more applications wants to use the same resource and are unable to proceed because all awaits the others to move on. For the AHEAD-NoC this situation occurs when to packages arrive from three or four

---

<sup>1</sup>Shared resources in this aspect are elements such as buses, memory or input/output pins

direction at specific router at the same time. The system stalls and has to be reset to function properly. Hence, to have a running system it is essential to avoid deadlocks or make the system able to solve this problem. The situation is depicted in Figure 2.1 where the router is only able to hold two packages and route one through itself at the same time. When the third package from the local module arrives at the same time as the two others, the router is locked and unable to perform its routing operation. When this situation occurs for the AHEAD NoC is described in [25].



**Figure 2.1:** Deadlock situation with three routers and one local module sending package to the router at the same time.

## 2.4 Clustering

Clustering was introduced in the pre-project report [27] as a way of grouping elements with large communication needs together. In a system with the risk of deadlocks, to cluster objects with large communications need is essential. By using this methodology, the modules connected to the NoC avoids generating crossing traffic and the risk of deadlocks is reduced. With low crossing traffic and shorter

distance for packages to travel, latency will also be reduced.

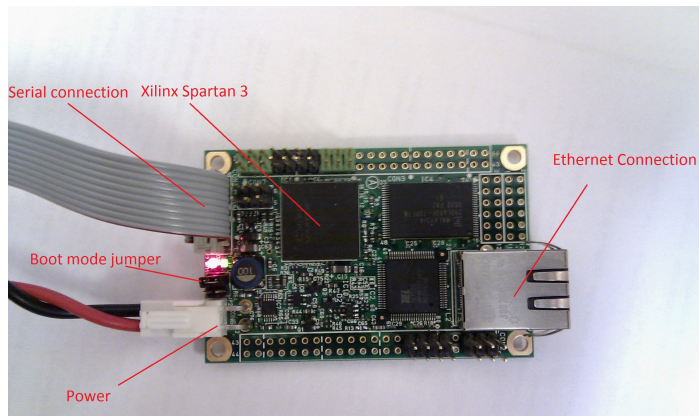
## 2.5 System platforms

The platform for this assignment has been the Suzaku-S platform. This platform includes a Spartan 3 XC3S1000 FPGA for the SZ030 version and a Spartan 3E XC3S1200 for the SZ130 version. The assignment started out with the 030 version, but changed to the 130 to have more resources in the development. The Spartan 3E is based on the Spartan 3 and is quite similar to it but has more Configurable Logic Blocks (CLB) available and more Digital Clock Managers (DCM).

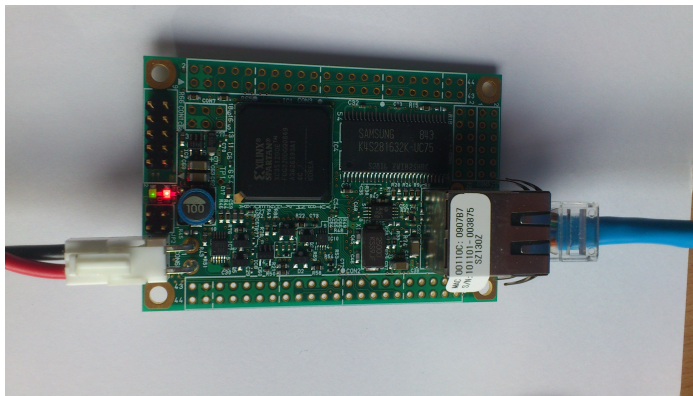
CLBs are the basic building block in Xilinx Spartan FPGAs and contains four Slices that contains two Look Up Tables (LUT) and two flip flops. DCMs assist the clock distribution on the FPGA and provides control over clock frequency, clock skew and phase shifts. The main motivation to use the 130 instead of the 030 in this project is the higher number of available CLBs. More details about the Xilinx Spartan FPGAs and their content are given in their data sheets [4, 5] and in the Xilinx dictionary [33]. List of other equipment used in this project is given in Appendix C.1 Equipment list for this project section.C.1

FPGA	CLBs	Slices	LUTs	Slice Flip Flops	Block RAM bits
XC3S1000	1920	7680	17280	17280	432K
XC3S1200E	2168	8672	19512	19512	504K

**Table 2.1:** Data for the two FPGAs used in the project.



(a) Suzaku-S 030



(b) Suzaku-S 130

**Figure 2.2:** The suzaku boards in operation [27].





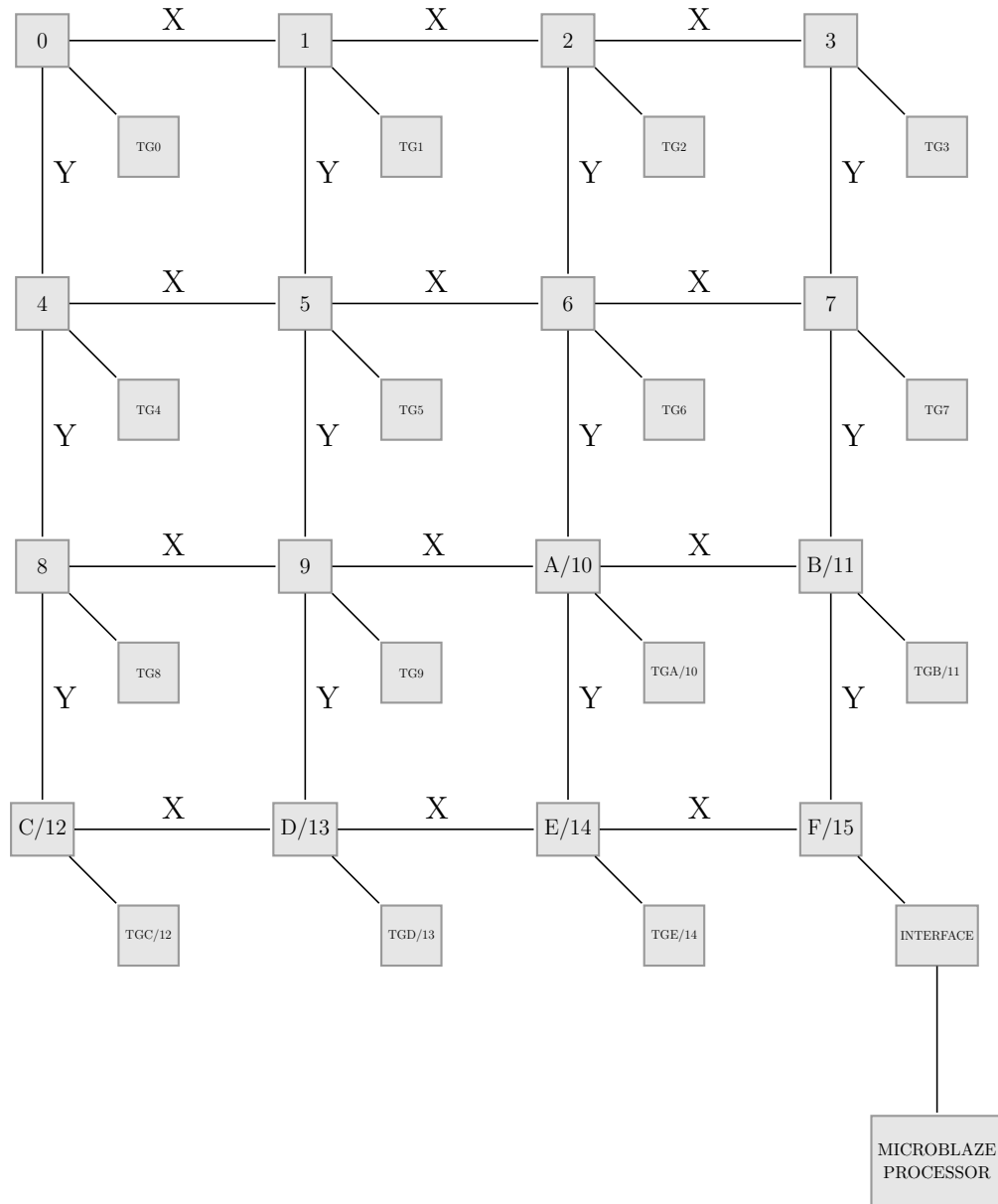
# Chapter 3

## Previous work

*The assignment is based on the work performed by Ivar Erslund (2009), Andreas Hepsø (2010) and the project by Magnus Namork in the fall of 2010. Erslund more or less finished the router design while Hepsø started the creation of an application test of the network. In the pre-project of this thesis this system was developed further and new table driven functionality was added. It is the foundation of the development of a full system test with reactive functionality. This chapter describes the previous work done with the NoC and the test system*

### 3.1 Functionality

The system consists of 16 routers with a test generator connected, as depicted in Figure 3.1 Network on Chip structurefigure.caption.19. It operates using the XY routing algorithm which is a simple algorithm that sends packages first in the correct horizontal (X) direction before it sends the package in vertical (Y) direction [17]. The NoC routers are connected through an interface module, via the on chip bus of the FPGA to a microprocessor called MicroBlaze. This microprocessor is a soft core processor implemented on the FPGA [34]. It runs the operating system uClinux which is a small variant of the Linux operating system. The uClinux adds the possibility of software communication with the network on the FPGA in run time. It is accomplished by cross compiling C programs with a specific compiler called mb-gcc which adapts the program to the MicroBlaze processor.



**Figure 3.1:** The Network on Chip in a basic mesh configuration with test generator connected to each router.

## 3.2 Design of the AHEAD Network on Chip

The system created by Erslund in [17] is the main foundation of the current system with some smaller adjustments. The system consists of a mesh of routers with buffers that routes packages based on information in the package. It uses packet-switching as transmission topology. The packages are 64 bit wide divided into eight flits<sup>1</sup> sent over an eight bit bus between each router. The routers and connected modules have a handshaking protocol that controls the sending and receiving of data between routers and from routers to modules. This protocol has the signals CTS (Cleared to send), RTS (request to send), request and grant. The latter signals are the signals on the senders side. This handshaking is used by the interface module and the test generators (TG) to be able to interact with their corresponding routers.

Sending of packages between TG and router is performed in the following operations:

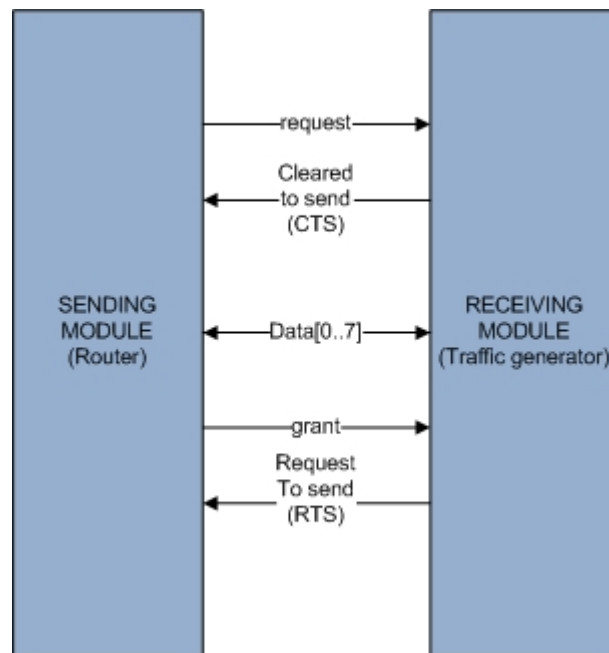
- The router has a package it wants to send
- It sends its request signal to the TG
- The TG sends the CTS signal back and the transmission commences.
- After the TG is finished and ready to send back to the router the TG sets its RTS signal high
- The router sends a grant back in response
- Transmission from TG to router is completed

This operation is illustrated in Figure 3.2Illustration of handshaking.figure.caption.20. AHEAD NoC previously had a system to handle prioritization of packages, but this was later removed in [25] because the behaviour did not function as intended.

One thing worth mentioning is the static nature of the design. The modules in the NoC are all defined as single entities several times in the HDL code. They are

---

<sup>1</sup>Flit is a part of the package transferred at the same time [18]



**Figure 3.2:** Illustration of handshaking.

not generic entities that could easily be added or removed by changing the system parameters. I.e. specifying in the top module that this system has eight TGs and eight routers instead of sixteen. This way of describing the NoC gives low degree of flexibility in modifying the system size and properties.

For more detailed information about the structure of the routing system and the routers, refer to Ersland [17].

### 3.2.1 Test system

The test-system work was initially started by Ersland, however he did not manage to complete the entire work of this system. Hepsø took this part further and introduced in [25] a more complete test system with a test generator and a system for monitoring traffic with traffic monitors. These monitors and a multiplexer module counted packages and stamped the packages with a global time based on clock cycles. In his work he also tested limitations within the system by testing different loads applied to the routers, including crossing traffic similar to Figure 2.1 Deadlock situation figure.captions.16.

### 3.2.2 Reactive test modules

The project completed in the fall of 2010, Namork [27] developed reactive functionality within the test system developed by Hepsø. The difference between these modules and the ones from previous versions was the ability to generate specified traffic and give the modules an independent set of operations to perform based on input. Based on a table that defined the behaviour of the test generator, the test generator sent traffic out on the network. It is mainly this work which is developed further and tested in this assignment.

## 3.3 Initial status of the system

When looking at the system after the project performed in 2010 one could map some areas to be looked at in further development. In brevity these were the properties of the system prior to the work on this assignment began.

- The hardware/software interface defined in the `user_logic` was not functional, the readout from the circuit was incorrect
- The previous C-code program developed for the system was not up to date with regards to the current version of the NoC system
- The area consumption was above 100% for the entire system of 16 test generators
- Reactive test generators exist, but not tested in a system
- The routing works as intended
- The routing is not deadlock free

These properties has been the foundation for detecting different areas where it is possible to achieve improvements. Especially the hardware/software interface was an area with great interest because a functional interface would mean on-chip-testing of the network would be possible. This would also open the door for developing more complex programs that could use the network on the FPGA.

### 3.3.1 System properties

The system bandwidth was determined by Ivar Erslund [18] as

$$8 \times 8\text{bit} \frac{123\text{MHz}}{12 \text{ Clock cycles}} \times 2 = 1312\text{Mbit/s} = 1,3\text{Gbit/s} \quad (3.1)$$

This is given a frequency of 123 MHz for the system without any loads i.e. test generators. In [27], a lower frequency of 50 MHz was found to be more feasible. Due to the proportional relationship between frequency and bandwidth, this leads to the following bandwidth.

$$8 \times 8\text{bit} \frac{50\text{MHz}}{12 \text{ Clock cycles}} \times 2 = 533\text{Mbit/s} \quad (3.2)$$

The number 12 comes from eight cycles to transfer a package four cycles in handshaking between routers and between routers and test generators. For this reason, 533 Mbit/s from Equation 3.2 is the reference value used for the bandwidth in this assignment.

Frequency	Theoretical Bandwidth	Bus Width flit size router<->router	Package Size
123 MHz	1,3 Gbps [18]	8 bit	64 bit
50 MHz	533 Mbps	8 bit	64 bit

## 3.4 Test system motivation

The initial status of the system gives the backdrop and motivation for the further development of the system. Since the system is not deadlock-free, the ability to use it for specific purposes and applications requires knowledge of how modules functionally could be placed in the network. To get this information it is essential to test the properties of the NoC and compare it with the applications' requirements. When knowledge of the the properties has been gained, limitations and possibilities with the NoC provides the information about what is feasible to implement in the AHEAD NoC system. In addition this provides information about what is

necessary to do to extend functionality. Some interesting factors to investigate, is how large traffic of packages the system can handle, and how long time it will use to handle the traffic in the system. These factors corresponds to the avoidance of deadlocks and the latter to the throughput and latency of the system.





# Chapter 4

## Development of the system

*The development of the system has been divided into two different areas. Since the system did not function properly from the beginning of the project, some effort had to be put into the development of a functioning system. The goal was then to run it on the FPGA in a proper manner and receive a response when testing. This is the structural development of the system. The second part considers the functionality of the test system and in particular the test generators in the system and their behaviour. This is defined as functional development. This chapter presents some of the main features connected to these two areas. Chapter 5 Requirements and designchapter.5 will describe the features from this chapter in further detail.*

### 4.1 Structural development

#### 4.1.1 Generic code development

Having a system which is built for communication between modules proposes a challenge when it comes to how the design is organized. This is because everything has to be linked properly together and several instances of the same modules has to be added to the system. By having a focus on a generic design it is possible to both achieve flexibility in the system and improve the ability to alter the functionality and testability. The implementation of this concept is described in details in Chapter 5 Requirements and designchapter.5 as a backdrop to the development of the test generator functionality.

### 4.1.2 Hardware software interfacing (HW/SW)

The HW/SW interface has been an issue in previous NoC projects, as described in the report of Erslund in [17]. It is a critical component to run proper tests of the NoC when implemented on the FPGA. This part describes the investigations made and the solutions implemented that makes the interface functional.

#### Interfacing from software

The user-interface to the circuit is described in software with an application written in C. The application, in this case the NoC, is assigned a memory space divided into registers when instantiating the peripheral in the Embedded Development Kit(EDK). In the current version of the system, nine registers are connected to the hardware, each of 32 bit . This requires a memory of  $32 \times 9 = 288\text{bit} = 0x120(\text{hexadecimal})$  which entails a memory range of 512 bits and an address range from 000 to 1FF. To use these in a simple and easily understandable manor has been one of the main focus' of the development. The C coded program has, based on this motivation, been adapted to use the registers as variables to simplify read and write operations.

#### Memory organization

A step down in the hierarchy of the design is the hardware/software interface. This is the communications channel between the FPGA hardware implementation and the software executing on the MicroBlaze processor. When communicating between these two domains there are some considerations that has to be taken into account. An area with great impact is the memory space. Spartan FPGAs comes without a Memory Management Unit(MMU) and memory read-and-write has to be done in a direct manner. Hence, none of the default C methods for handling memory read and write works. The first problem to arise with the memory space was the location of the registers in the memory block of the FPGA. The initial memory area was defined between 0x81000000 and 0x810001FF. This was not functioning properly either due to defect memory (SDRAM) or proximity/overlap of other modules' assigned memory. The SDRAM controller of the Suzaku FPGA system is located just up to 0x80FFFFFF in Spartan 3 and 0x81FFFFFF for the

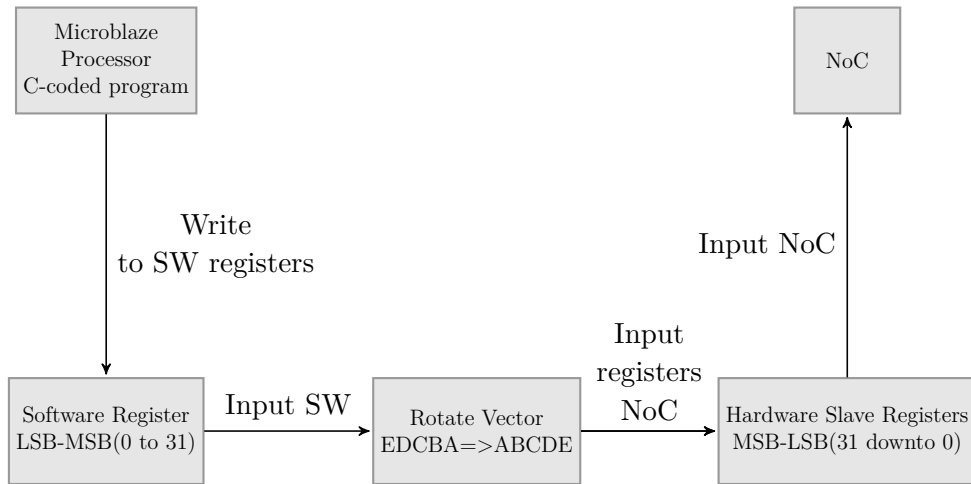
Spartan 3E and this might be the root of the problems with the mentioned memory space.

To remove these problems, the memory space designated for the Network on Chip was moved. The solution to this problem, was found using a simple adder module with simple output from [20]. The adder module was moved around to several memory ranges and tested before concluding with an area suitable for the AHEAD NoC. The NoC entity was placed in the area [0x84000000->0x8400001FF] and managed to run properly with this memory space. Hence providing a working solution for the NoC module. For further details about memory and how to organize the system, please refer to Appendix D Tutorial:How to implement the Network on Chip on the Suzaku-S platform appendix.D and Figure D.2 System Assembly EDK figure.caption.81.

## Interfacing from Hardware

When developing and implementing the HW/SW interface, another challenge surfaced. It was located a difference between the way the Microblaze processor writes to the memory registers and how the FPGA module reads from these registers. The cause was found in the Microblaze reference guide [34, p.21] that the registers are written 0:31 with the Least significant bit(LSB) at 31. However, the hardware interface and the NoC uses the common 31 downto 0 notation for VHDL, and this causes a SW input of hexadecimal x"ABCDEF00" would give an input x"00FEDCBA" to the NoC on the FPGA. This incorrect input gave no useful information to the circuit as described in [17].

The solution was adding a function to reverse the incoming vectors. This enabled the use of the common "31 downto 0" notation without having to write the vectors inversely in the C coded program. The vectors added from the C-code are in this way directly compatible with the data entering the network through the interface module. This method is illustrated in Figure 4.1 HW/SW interface vector handling.figure.caption.25.



**Figure 4.1:** HW/SW interface vector handling.

### 4.1.3 Other structural changes

There has also been done work improving the easiness of communicating with the NoC on FPGA. In general this is connected to using the Ethernet connection of the Suzaku-S together with tools such as Network File System(NFS) [12]. Having these parts in place simplifies the development of the system and the time from new functionality is developed to its FPGA implementation is reduced. The different ways of setting up and reconfiguring the NoC system on the FPGA is described in details in Appendix D Tutorial:How to implement the Network on Chip on the Suzaku-S platform appendix.D.

## 4.2 Functional development

This part of the project development is based on desired properties to test on the AHEAD NoC. It gives an idea of the direction the test-system is developed.

### 4.2.1 Specifications

To create a more generic design and to full-fill the testing motivation of the project, a set of different design implementations was developed. The motivation for these features was to improve the benchmarking of the AHEAD Network on Chip. Func-

tionality introduced for the NoC test system included:

1. Completely reactive system which reacts to the information in the package
2. Use information in packages and tables to generate specific traffic patterns
3. Ability to change traffic tables from software
4. Manipulation of data in each packet
5. Calculate the bandwidth and latency of the system
6. Ability to run software programs and interact with the system when executing on the FPGA

### **What properties to measure**

To get a precise measurement of the network it was necessary to have some properties to measure. The system already has functionality counting packages and stamping them with current number of clock cycles. It has been used further and connected to important metrics for the NoC. These metrics are useful to get an idea of the properties of the system, as introduced in Section 2.2.1 System properties subsection.2.2.1:

- Throughput
- Latency
- Bandwidth

### **How to create the tests**

To perform a test strategy that measures the given properties, it has been looked at the number of active components in the system and how they communicate. Secondly, the idea has been to develop a system with the ability to generate larger amount of data to get a test system that resembles the application one wants to evaluate. The focus was to get a good understanding of how the traffic behaves in

the circuit, therefore the modules are deterministic<sup>1</sup> in their behaviour. This was in contrast to the previous version that included a pseudo random functionality.

The functionality of the test generators are Finite State Machines(FSM) all together. This provides a good platform for performing different tasks based on different information in packages. The development of the functionality is in that matter only a question of altering the states within the modules. In the development the concept was to have one module that could fill all the roles in a benchmark or test and therefore the generic structure of each module was essential. The specific functionality of the modules was based upon two different application examples. These are:

- An MPEG pattern simulated and run on the FPGA
- A Cell processor like system with computation and communication

### 4.3 Application example:MPEG decoder

Andreas Hepsø introduced in his Master thesis, an MPEG decoder and scaler as an application applicable to the NoC [25]. This system consists of 11 different modules performing the MPEG decoding and scaling of a movie stream. In a mobile application it will be desired to be able to downscale the video in real time without delays. The MPEG decoder is fixed in its structure in the way that one sends packages in the same direction through all the steps without sending packages backwards or to several different test generators.

#### 4.3.1 Real time requirements

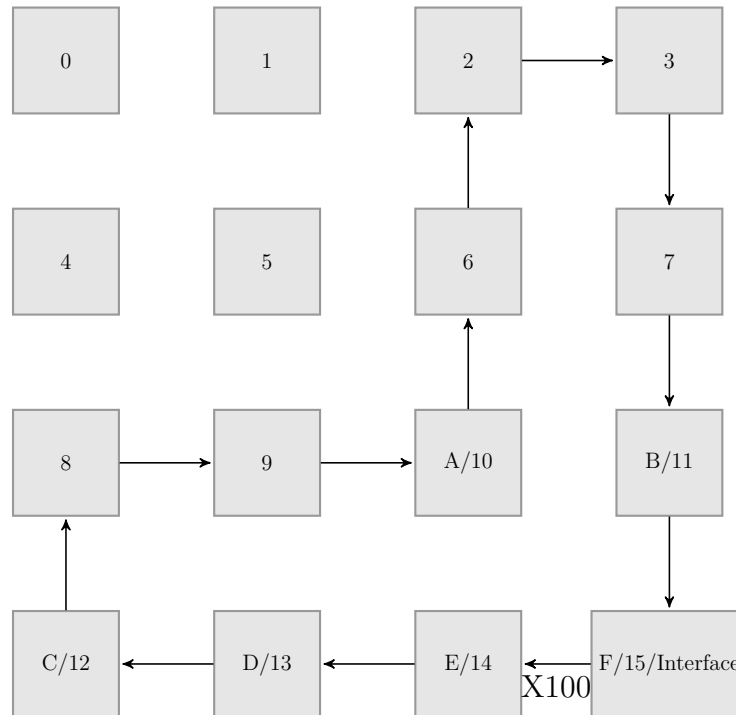
If used as an on-the-fly scaler<sup>2</sup>, as described by Hepsø in [25], some system requirements will apply. The packet stream has to be continuous and little delay will be accepted as it will lead to a reduced quality of the movie for the user. This means that such a system has soft real time requirements<sup>3</sup>. Because the system has these

---

<sup>1</sup>Deterministic behaviour means that the behaviour is predictable

<sup>2</sup>On the fly scaler means that the scaling is done in real time

<sup>3</sup>Soft real time requirements means that the output/result of the system has a deadline, but a missed deadline is not critical to the system



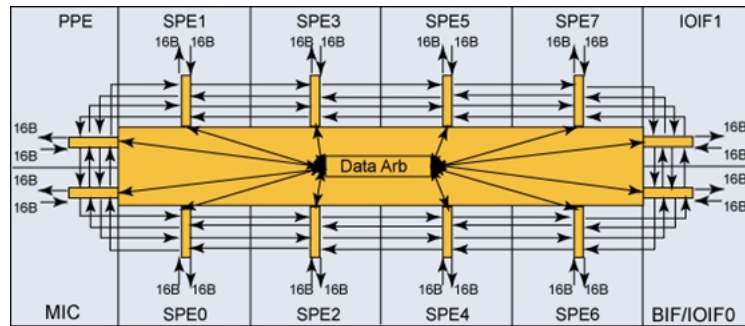
**Figure 4.2:** MPEG traffic pattern, X defines a (possible) generation of packages.

requirements it is necessary to have a high degree of determinism in the system and a highly controlled data and package flow.

## 4.4 Application example:PS3 ring bus system

Another application example is the Cell processor of the Playstation 3 [10]. The system consists of eight so called synergistic processing elements(SPE), one Power processing element(PPE) and one memory interface controller(MIC). The PPE is in control of running the operating system and coordinating data flow through SPEs. The Cell processor has the Element Interconnect Bus(EIB) as its main interconnect between elements in the processor. This system has a ring bus topology with a centralized arbiter which decides which processing element has access to the bus. The system is then capable of routing packages to and from the processing element either in a clockwise or a counter-clockwise manor. This is illustrated in Figure 4.3Cell processorfigure.caption.30. The Cell system parameters are not the





**Figure 4.3:** Illustration of the Cell processor.

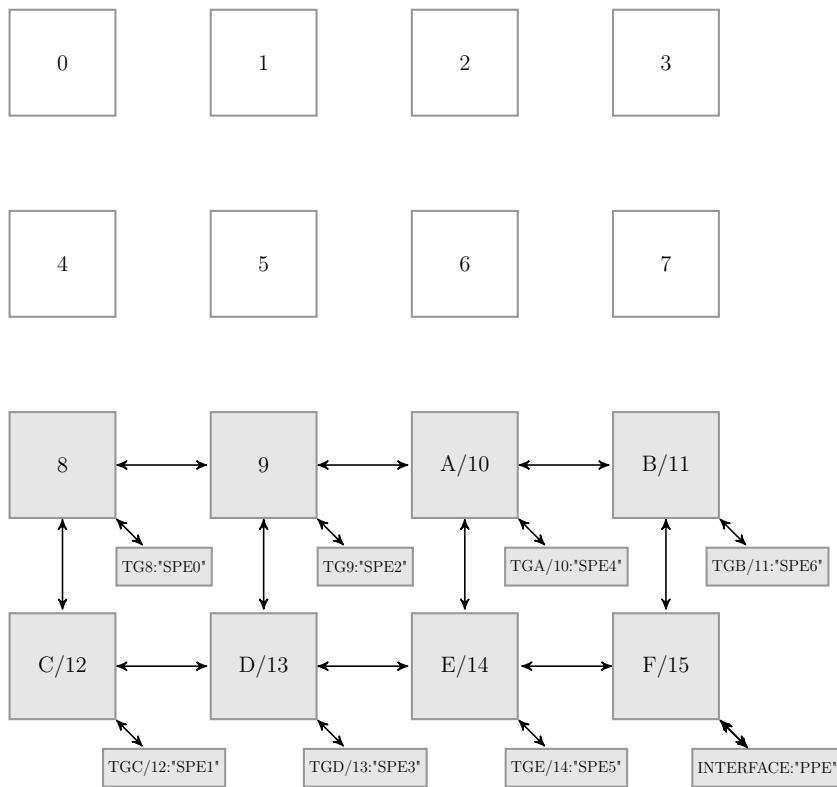
main focus of the use of this as a test example. It is not a goal to achieve the same performance from the AHEAD NoC, but to show one way of using the NoC and to visualize the essential parameters and properties that is measured by referring to a real application.

Frequency	Theoretical Bandwidth	Effective Bandwidth	Bus Width	Package Size
3.2 GHz	204.8 GB/s	78GB/s - 197GB/s	16 bytes	256 bytes

**Table 4.1:** Properties of the Cell processor [13].

With this system as a test case example, a relevant question would be; what if the PS3 system was implemented on a Network on Chip? What would the properties of the system be and would the network handle the data communication between modules? These are questions that forms the foundation for the development of a similar system for the NoC. This system is not able to do as many operations as the cell processor, but the communication and way of operating is similar [13]. The concept is to replace the EIB with NoC and see how the behaviour of the system is like. It is in this case worth noticing that the Cell EIB uses centralized arbitration<sup>4</sup> as shown in Figure 4.3Cell processorfigure.caption.30 while the NoC uses distributed arbitration in each router [18]. The packages in Cell and in the NoC are similar with regards to dimension. The Cell EIB has a width of 16 bytes and a package size of 128 bytes while the NoC has a data width out of each router of 8 bits and a package size of 64. This means that both

<sup>4</sup>Arbiter is a component using certain criteria to determine which module allowed to access the resource i.e a bus [28, p.26]



**Figure 4.4:** Cell processor with NoC instead of ring bus, White is deactivated area.

needs 8 cycles in receiving and 8 cycles in sending data out on both networks. The implementation of this example is described further in the next chapter.

## 4.5 Other possible application examples

### 4.5.1 RSA encryption

A common example well-applicable to benchmarking and testing of processors and other systems is the RSA encryption algorithm [29]. This algorithm encrypts and decrypts a message of various size. This has been used as benchmark for several applications such as the already mentioned cell processor [19]. An RSA system from a previous project at NTNU, Realization and test of digital components, was synthesized and used 10% of the area of the Spartan 3E FPGA. The implementation has, however, not been developed further since the RSA circuit is quite complex and has to be modularized and distributed in a correct way on the Network on Chip to be used as a benchmarking and test system. This would take a lot of effort and not necessarily provides more information about the systems' properties than a simpler system such as the Cell example does. However, it is a possibility that would be interesting if one wants to develop a specific practical application for later use.

# Chapter 5

## Requirements and design

*This chapter describes implementation details about the systems described in Chapter 4 Development of the system chapter.4. It describes the code behind the solution, from development of the hardware/software interface to the specifications of the traffic system on the network.*

### 5.1 Design goals

#### 5.1.1 Area

The conclusions made in [25] [17] and [27] said that the circuit-area containing the test modules is close to 100%. Due to this, the design is primarily concentrated around optimization and reduction of the circuit area. Optimization of the frequency is outside the scope for the circuit. Mainly because the focus is to see how NoC routing behaves with a certain load of data packages, and not how fast it will perform. The original code has been inspected and some areas of improvements has been detected, to optimize the design for a minimum area of the circuit.

#### **Area improvements attempts**

1. Reducing the size of large global vectors including
  - Time-stamp-vector
  - Package counter vector

2. Changing the structure of the FSMs from variables to signals in next state control logic
3. Changing from asynchronous reset to synchronous reset
4. Removing pseudo-random test functionality

The only factor contributed towards reducing the area substantially was the removal of the pseudo-random functionality. This gave a reduction of the area for the test generator of approximately 10%. The rewrite of the test generators' FSM caused a timing problem in communication with the router, hence it was not developed further. Changing the reset from asynchronous to synchronous reset, as suggested by Xilinx in [22], gave some issues with the previous implemented functionality and was not investigated further. None of the other attempts proved to give a substantial contribution of available area. This is possibly due to the structure of the FPGA.

Succeeding with the other improvements, a more thorough investigation of the design, including floor planning, would be required and has not been pursued. A simpler approach has been chosen due to the efficient use of time. The way of testing and implementing added functionality is therefore primarily based upon creating a design that is easy to alter the size when testing different properties and versions of the system.

### 5.1.2 Flexible Design

When changing the system functionality it is advantageous that minor changes of code may result in a substantial change in behaviour and size. The existing design lacked a good framework for maintainability and flexibility. It has therefore been made some improvements in the direction of creating a flexible design. This is beneficial since the area of the design is close to 100% of the FPGA, and to include more functionality in the test system requires this property. In addition, to develop it to fit different FPGA platforms might be a desired feature in future projects. For instance, it might be desired to use the NoC on a smaller platform and then it must be simple to include and exclude modules in the system.

In HW designs using VHDL this is solved by using generic mapping and generate

statements as displayed in the code in Listing 5.1. Example of how the generate statement is used to make it easier to add and remove parts of the system; the generic variable:deactivated\_tm defines the number of modules deactivated. Listing 5.1. From before, in [27], a test- and type library was developed. These libraries include types, functions and signals to be added and withdrawn from the system based on the current requirements. In that way they provide a system to maintain, develop and optimize the NoC system in an easy way in both this project and further development.

One area that has been evaluated, but not improved, is the router design. As mentioned in Section 3.2 Design of the AHEAD Network on Chip section.3.2 it was designed in a very static way, and no easy way of changing the number of routers in the system exists. The possibility of changing this in the same way as with the test generators has been looked into, but this would require an entire rewrite of the system. For testing purposes it does not provide an improvement, and will require more time to do. Therefore, the possibility to remove and add test generators and traffic monitors is present, but not to remove the connected and then redundant router connected to it. However, in later optimization of the NoC this could be a beneficial improvement of the system. Supplied code of the test generator and interface is supplied in Appendix B Code appendix.B.

```

1 NoC_TM: for n in deactivated_tm to number_of_routers-1 generate
2 TM: entity noc_v1_00_a.TM(behavioral)
3   generic map(tm_number=>n) --gives the correct index to the traffic monitor
4   port map(
5     clk      => ungated_clk,
6     reset    => reset,
7     CTS      => packet_trigger(n),
8     readout_finished => readout_finished,
9     packet_counter => packet_cnt(n)
10  );
11 end generate;
```

**Listing 5.1:** Example of how the generate statement is used to make it easier to add and remove parts of the system; the generic variable:deactivated\_tm defines the number of modules deactivated.

## 5.2 Hardware/Software interface

### 5.2.1 Software accessible registers

The communication between hardware and software is done with registers in memory defined in the synthesis tool. There are in total nine HW/SW registers currently in use and they are organized like in Table 5.2 Register numbers and functionality. [table.caption.34](#).

A specifically important register is register 4. This register is the control-register and it is used to determine transmission of the packages to the network. It also contains the reset functionality. This register and its bit values are depicted in Table 5.1 Address listing [table.caption.33](#) In total there are 20 registers in the AHEAD NoC, but as mentioned only nine of them are connected to the it. The remaining ones are added to have registers available when extending functionality, i.e. with a second interface module. Another potential use of these registers is to use them as a platform for readout of the monitoring of data on the FPGA in real time. These ideas have not been developed further in this assignment due to the focus on the functionality of the test system and that it requires more time.

Bit values	31....28	27	26	25....8	7	3	2...0
Signal names:	Mux_select	send	readout_finished	not in use	BRAM_enable	reset	not in use

**Table 5.1:** List of different bit values and their use in the control register.

### 5.2.2 Hardware interfacing

The registers are all instantiated in the user logic file of the system which serves as a high level hardware part of the HW/SW interface. The interface module is in this case the low-level interface part which is connected directly to the NoC. The registers are all 32 bit wide. The read-and-write to these are controlled by

Register	Info	Address range	Name
0	Data_in_33	0x84000000+0	slv_reg0
1	Data_in_33	0x84000000+4	slv_reg1
2	Data_out_33	0x84000000+8	slv_reg2
3	Data_out_33	0x84000000+C	slv_reg3
4	Status bits/Control bits	0x84000000+10	slv_reg4
5	BRAM address register	0x84000000+14	slv_reg5
6	Feedback register	0x84000000+18	slv_reg6
7	Counter	0x84000000+1C	slv_reg7
8	Timer	0x84000000+22	slv_reg8
9..19	Available	0x84000000..	slv_reg9..19

**Table 5.2:** Register numbers and functionality.

three different processes; read, write and a specific process to specify how these operations are connected to the NoC. By dividing these processes it is much easier to change a specific part of the NoC interface. The processes uses a vector with a "one hot" bit to determine which register to be read and written from. These are all default values when creating the peripheral in the Xilinx EDK tool but they have been altered to get a more readable and maintainable code. An important property already mentioned in Chapter 4 Development of the systemchapter.4 is the rotation of the input vector from the software registers. This is implemented with a simple function to enable a simplistic read and write to the NoC.

### 5.3 Traffic generator

The traffic generator is the core of the test and benchmarking system and is the component that produces traffic in the system. The original generator designed by Hepsø [25] provided limited functionality for producing data to the system. It was based on three Finite State Machines (FSMs) handling the receiving, throughput of the generator and sending procedures. The designed structure of the routers, interface module and test generators was the same. To remain compatible with the rest of the system it is reasonable to keep the test generators in this way. The use of FSMs is also proposed as a good way of designing test generators, this as described in [14, p.7].



The functionality has been modified and is now capable of generating traffic to the network based on a table of a defined traffic patterns. This means that the test generators operate and communicate based on information stored in each module instead of being given this information in packages. To illustrate this, in the previous version, the package contained where to send the generated package. With the new functionality, the test generators determines where to send the package based on knowing which generator it has received a package from, and the type of the package. This is the concept of the reactive functionality. The modification has been based on the work done in the fall of 2010 by Namork [27] with table driven communication and implemented in a full system. In the mentioned work, tables proved to be an efficient way of determining behaviour of the generator. However, increasing the size of the tables or adding more tables, causes area overhead. This fact will be illustrated with the Cell example implementation.

### 5.3.1 Sending and receiving packages with traffic tables

The traffic tables contains an array of 4X16 with the information the test generator needs to send packages. This information defines behaviour based on the address of the sending test generator. In Table 5.3Traffic tabletable.caption.35 this is the "SenderID" and the traffic generator then uses the corresponding "DestinationID" field to determine where to generate packages. There is a possibility of generating a continuous stream of packages from the network with this table by defining the "number of packets" column.

## 5.4 Traffic pattern in the NoC

### 5.4.1 Motivation

The motivation for looking into possible traffic patterns has been in the non-deadlock free network. There has been an acceptance that there is not possible to implement a network that can handle large crossing communication in a router. As described in Section 5.1.1Areasubsection.5.1.1 this is due to area constraints with the current FPGA. This means that the routers will not be able to function

TG#	SenderID	DestinationID	Counter	Number of Packets	VHDL
NA	0	15	1	4	(0,5,1,4)
NA	1	15	2	3	(1,4,2,3)
3	2	7	3	1	(2,7,3,1)
7	3	11	4	2	(3,11,4,2)
NA	4	9	5	4	(4,9,5,4)
NA	5	10	6	3	(5,10,6,3)
2	6	3	7	1	(6,3,7,1)
11	7	15	8	2	(7,15,8,2)
9	8	10	9	4	(8,5,9,4)
10	9	6	10	3	(9,6,10,3)
6	10	2	11	1	(10,2,11,1)
NA	11	14	12	2	(11,14,12,2)
8	12	9	13	4	(12,4,13,4)
12	13	8	14	3	(13,8,14,3)
13	14	12	15	2	(14,12,15,2)
14	15	13	16	2	(15,13,16,2))

**Table 5.3:** Traffic tables and how they are implemented, in this case a MPEG decoder pattern. NA means the testgenerator is not in use.

properly if there is traffic entering from several directions at the same time as described in Section 2.3Deadlockssection.2.3. The idea is then to develop a test system in order to map where the different modules of the network would be placed for the system to be functional in run-time. To use modules that generates a traffic-pattern it is possible in an easy way to simulate a real system without having to implement all the modules of that system.

### 5.4.2 Sendback pattern

This pattern consists of a table that defines the case where a test generator receives a package from the interface. When the package is received, the test generator returns a package to the interface. This is a simple configuration where the network gives a response to a simple input from the user. The way this is configured is simply to change all the "Destination ID" fields in the Table 5.3Traffic tabletable.caption.35 to 15. This test is in practice similar to the network admin-

istration tool "ping", used in IP<sup>1</sup> networks. [7] The intention of this test is to verify the functionality of the interface and the connections within the circuit, and this works as the Design for test for the NoC.

### 5.4.3 MPEG pattern

The MPEG example and pattern was described in the assignment by Hepsø. It defines 11 steps of an MPEG decoder that decoded and downscaled an MPEG video stream. This pattern with a slight modification of module placement is shown in Figure 4.2MPEG traffic patternfigure.caption.28.

With this as a background, the development of the traffic-pattern was built on the reactive traffic generators' table. The values in the table was set to mimic the traffic pattern of the MPEG decoding. To mimic a stream, the values for test generator 14/E was set to send 100 packages. This was to mimic a data stream similar to an MPEG stream. The reason why the entire MPEG decoding algorithm is not implemented instead is due to area constraints. In addition, the intention behind the testing is to test the network, not to test how the MPEG algorithm works. When this system proved to be successful, it was then used as a reference design in further development of the test system.

### Communications protocol MPEG

A communications protocol for the MPEG system was developed so that it defines the packages and the different operations to perform. This basic protocol is shown in Table 5.4MPEG protocoltable.caption.37. Adapting to a more advanced protocol such as the OCP-IP protocol has been looked into, but has been considered to require large structural changes of the test generator and NoC design and require too much time.

### 5.4.4 Internal Design

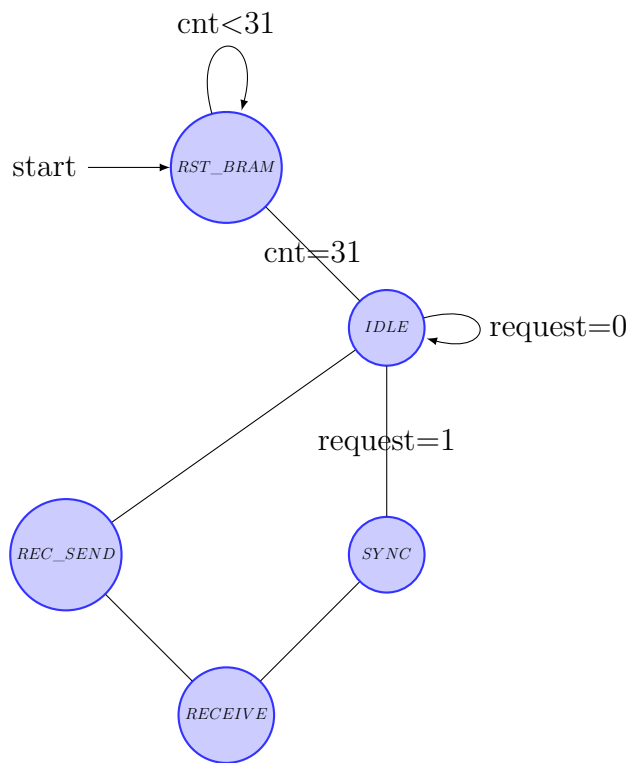
There are two parts of the test generator that forms its functionality when testing the MPEG pattern. First of all it is the protocol as in Table 5.4MPEG

---

<sup>1</sup>Internet Protocol

Bit values	63...60	59...40	39...36	35...16	15...0
Signal name:	Address	not in use	sending_router	not in use	global_time

**Table 5.4:** Communication protocol in the MPEG like system.



**Figure 5.1:** Input state machine of the Traffic generator, MPEG configuration.

protocoltable.caption.37, and secondly it is the interpretation of this protocol in the state called REC\_SEND. This state contains the receiving part, interpretation and definition of the next package to be sent. It uses a table like in Table 5.3Traffic tabletable.caption.35 that defines sender ID and number of packages. The "send" state machine handles the rest of the operation with the transfer to the local router.

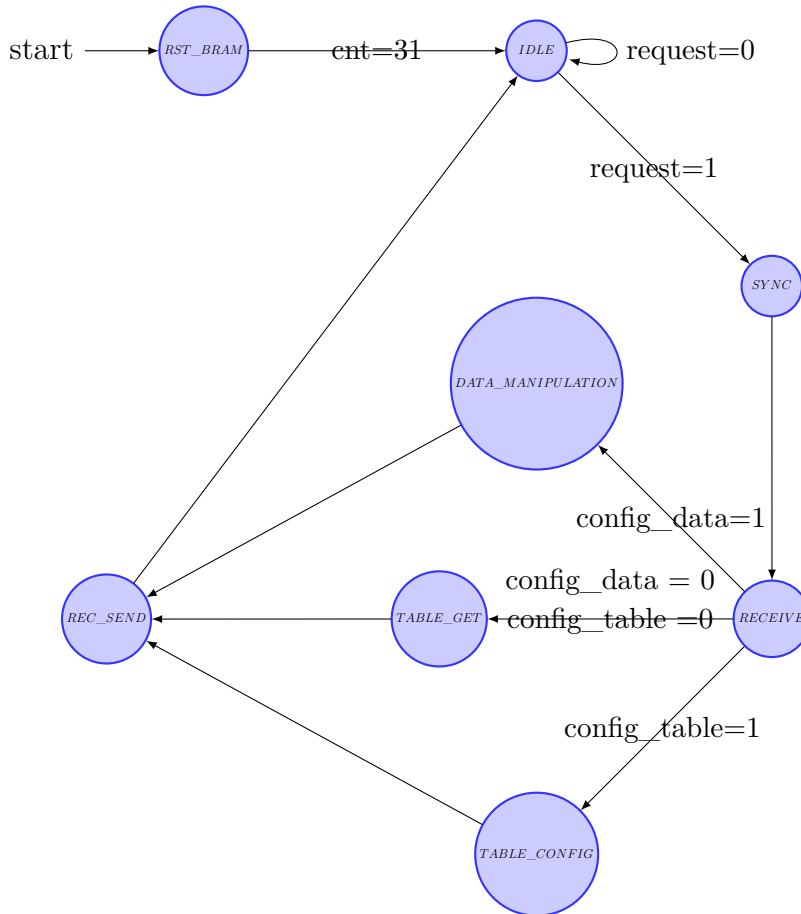
## 5.5 Multitasking test generator; Cell application

Based on the traffic generator with the MPEG pattern defined, further development was performed. In the new test generator, the essential is to perform different tasks based on the type of package. This generator is built up by the same state machine as the MPEG pattern, but has three operating modes to choose from instead of only one. It is developed on the larger Spartan 3E instead of the Spartan 3 which was used for the MPEG pattern. Although it is a larger platform, the system employs only eight test generators instead of the sixteen from the MPEG pattern. This is because of the higher area overhead with the extra functionality. Even though full functionality of the SPE in the Cell processor is not feasible to implement on a Spartan FPGA, it is possible to mimic its behaviour with data communication. This is done by dividing an operation into parts and have a need for communication with other elements to perform the operation. Calculations in the system can be done simple by sending a package to any of the seven modules, like in Figure 4.4Cell processor in NoCfigure.caption.31, in the system and the computation will be produced. By sending packages to different test generators, the latency and throughput will vary, but more than one calculation of packages can be performed simultaneously.

### 5.5.1 Operating modes

The test generator is created to perform different operations based on what tasks the specific generator is given. Information about the task is provided in the package and three modes exist. These modes are intended to provide the functionality needed to get good results with regards to the properties that are measured. The three different modes are listed below and described further in this section.

Operating Modes	Package type
Basic traffic	0
Data Manipulation	4
Table configuration	8



**Figure 5.2:** Input state machine of the Traffic generator, Cell example application.

### Basic traffic

For the new test generator, the "Basic traffic" mode is the same as the MPEG pattern described in Section 5.4.3MPEG patternsubsection.5.4.3. Packages traverse the test system from F to E and in a clockwise circle back to the interface.

Bit values:	63....60	59....56	55....40	39....36	39....36	35....16	15.. 0
Signal names:	send_to	package_type	not in use	tg_hex_number	next_operation	package_data	global_time

**Table 5.5:** Multi functional protocol.

### Software based traffic table

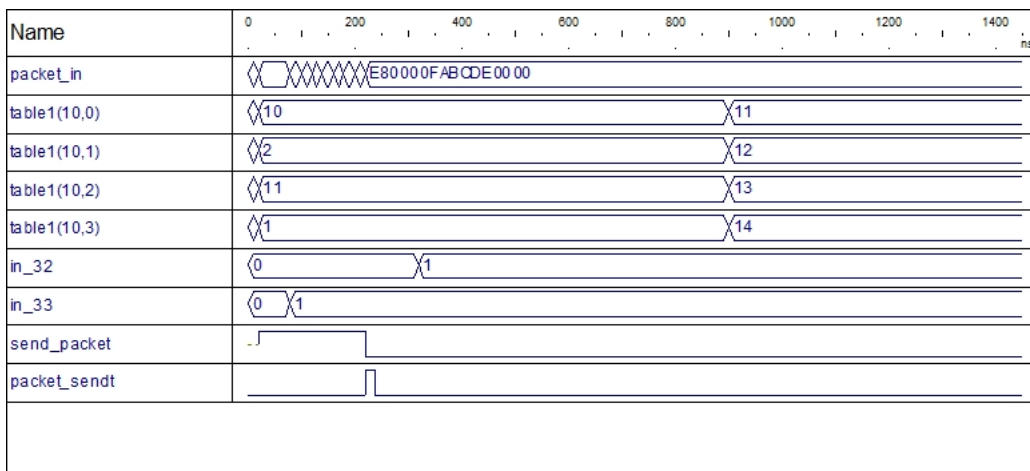
In the initial MPEG test generator the table was statically defined. The table was made dynamic in an attempt to improve the possibility to change the behaviour of each generator from software. It made it possible to send a package with a given data load and change how the test generator performs its sending. This enables the possibility to alter traffic tests in software and in a fast way, test several different patterns without FPGA reconfiguration. When it is desired to change the contents of the table, the package type, as in Table 5.5 Multi functional protocol table.captio.42, is set to 8. Having a dynamic table provided an ability to implement a table configuration mode within the test generator, as described in the next paragraph.

### Table configuration mode

The ability to alter functionality of the table without reconfiguring the system on the FPGA is interesting. With this feature it creates the opportunity to reconfigure the systems traffic and functionality in run time. To test how this behaviour could be implemented, a second state in the state machine was implemented. In this mode a package is sent to a specific test generator and tells it to change a specified row in its table. This procedure is shown in Figure 5.3 Table reconfiguring through packagesfigure.captio.45. When the table is configured the test generator sends a package back to the requiring module to confirm the reconfiguration. Next time the generator receives a traffic package, it has a new defined pattern within its table.

35..32	31..28	27..24	23..20	19..16
change_value	DestinationID	SenderID	Counter	Number of packages

**Table 5.6:** Data to configure testgenerators internal table.



**Figure 5.3:** Reconfiguring of the traffic table through software data packages.

### 5.5.2 Data Manipulation

To create a Cell processor-like system, another state called "Data manipulation" is created. This is the core of the Cell functionality and takes data input and computes a new output. The initial configuration of this system takes two hexadecimal numbers and their difference and computes the mean value of the two numbers. The operations used in each traffic generator is either increment or decrement of the input hexadecimal values. The operation designated to each generator is based on its unique number in the test system, odd numbered generators perform increment, even numbered perform decrement. Every generator contains an additional table that gives information about which test generator that has the next operation to be performed. An internal counter is used to choose a different module



Difference	Unused	High value	Low value
------------	--------	------------	-----------

**Table 5.7:** Values on input to the data manipulation mode.

each time. A limitation associated with this way of doing operations with two implemented tables was that the available resources of the FPGA. In synthesis it was not possible to implement two dynamic tables that could be read from in each test generator. Therefore the table connected to the "Data Manipulation" state was made static and thus not possible to alter with packages sent to the network.

### Design of the state

The design of the Cell example application is based on the MPEG traffic example. By adding computation, it is possible to mimic behaviour based on several communicating modules relative placement in the NoC. Hence, the alteration is the creation of the new package and the steps that does computation in the receiving state machine. In addition the modules are clustered together instead of forming a specific path like the MPEG pattern. Input of the data manipulation state is simple and is shown in Table 5.7

Generation of the packages is based on Table 5.8 Cell application example table. Each even numbered test generator has listed three odd numbered test generators to connect to in different order. The connection determines the next operation of the system. One decrement operation on the largest input value is then followed by an increment operation on the smallest value. The test generators are, due to the number of modules they communicate with, classified as a Multiple Input Single Destination(MISD) processing elements (PE) type as described in [14, p.8].

## 5.6 Software based test program

A test program or micro-benchmark was created in C and compiled for the Microblaze processor. It was based on the program developed by Ersland in [17] and Hepsø in [25]. It consists of different methods that enables functionality to test the network. One of the methods writes the type of package the user wants to send

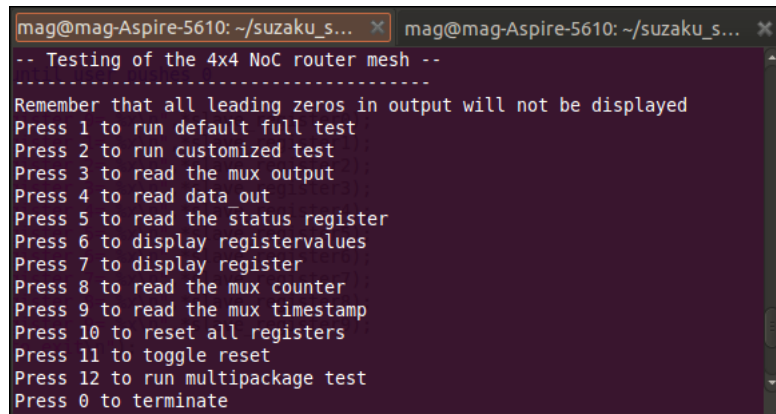
Test Generator	Next operation TG	Number of packages	VHDL table
8	9, 11,13	1	(8,9,11,13,1)
9	8,12,14	1	(9,8,12,14,1)
A	11,9,13	1	(10,11,9,13,1)
B	12,8,14	1	(11,12,8,14,1)
C	13,9,11	1	(12,13,9,11,1)
D	14,8,12	1	(13,14,8,12,1)
E	9,13,11	1	(14,9,13,11,1)

**Table 5.8:** Cell application example table.

to a specific test generator to the network and receives and interprets the output and delay. It also reads out the time stamp of each package and converts it to decimal values to get a good estimation of how long each package uses through the network. Latency values are then used to give an indication of the throughput of the test. This communication goes through the software accessible registers in the SDRAM. The interaction between the microprocessor and the FPGA is write and read operations of these registers as described in Section 5.2.1 Software accessible registers subsection.5.2.1. The code for the test is found in the Appendix B.5C program for on chip test of the Network on chip listing.B.5. In addition to the test program written in C, a Java graphing program was created to illustrate the data from the circuit. A text file was used to store the data from the C tests, and the Java program reads from this file and graph the values. It is beneficial as a tool to develop a user friendly benchmark in the future. This program will not be discussed further but is available in the supplied digital attachments.

### 5.6.1 Redefining behaviour through software

To quickly modify the test without having to reconfigure the FPGA is an interesting feature when benchmarking the Network. The idea is to change the data of traffic tables, and by that create larger and more advanced tests written in software. By providing this functionality, the testing will in later editions of the system use less time and give more and better data for analyzation. One attempted solution to this problem is the implementation within the designed test generator described in Section 5.5.1 Table configuration mode section\*.43 and in the test program. The



```

mag@mag-Aspire-5610: ~/suzaku_s... x mag@mag-Aspire-5610: ~/suzaku_s... x
-- Testing of the 4x4 NoC router mesh --
-----
Remember that all leading zeros in output will not be displayed
Press 1 to run default full test
Press 2 to run customized test
Press 3 to read the mux output
Press 4 to read data out
Press 5 to read the status register
Press 6 to display register values
Press 7 to display register
Press 8 to read the mux counter
Press 9 to read the mux timestamp
Press 10 to reset all registers
Press 11 to toggle reset
Press 12 to run multipackage test
Press 0 to terminate

```

**Figure 5.4:** The different methods listed in the program run on the FPGA.

test generator reads the value which is sent from software consisting of five hexadecimal values within the package and writes this value to its internal traffic table. In this way it changes the way it sends packages in the network. The data sent as a load to the test generator is listed in Table 5.6change\_datatable.caption.44.

## 5.7 Surveillance and monitoring of traffic

In the system the surveillance and monitoring of traffic has been handled by a packet counter and a time stamp module. These two were described in [25] but not used for testing on the FPGA. The intention behind having these modules was to get traffic information from the circuit. Some issues with these modules have been addressed. The time-stamp-module used originally 32 bit vectors that were too large for the system, as mentioned in Section 5.1.1Areasubsection.5.1.1. Package counter vector was equally large. Those were then reduced to 16 bits which should be more than enough to monitor the traffic.

It is easy to take this module in and out of the system and this has been done in development to reduce area overhead when there is no specific need for time stamp information. There is in addition a packet counter which has proven a handy tool to monitor traffic in the circuit. The practical use of this module is that it can write out data of package traffic from anywhere in the network through the interface. An example of this use is illustrated in Figure 5.3Table reconfiguring through packagesfigure.caption.45 with the signals in\_32 and in\_33. They are the

packet counters connected to router E and F.

As an example, in the MPEG pattern, using the counter on router F verifies that the correct number of packages has passed through. This is used both in simulation and on-chip testing on the FPGA. Which router to read data from can be determined both in software by setting the control register to the desired value or by setting it statically in the interface in the `user_logic` module. This determines both the time stamp one want to read out and the package counter. It could have been implemented more functionality to monitor the packages from within the network, but this is considered to be both time consuming and complex. Thus, it has been chosen not to look further on this functionality since the current implementation provides sufficient data. It has instead been a focus to develop the interpretation of the data especially in the software program.

### 5.7.1 Measuring throughput

To get a good measurement of the throughput in the circuit, the mentioned surveillance and monitoring systems are essential. The global counter implemented in the circuit is used widely to get useful data of latency from operations within the network. By stamping each packet and then reading out this time value it is possible to calculate the throughput of the system. This system is however not as precise as desired since the sending-and-receiving overhead involving the interface is not added to the measured value. A solution that has been investigated, but not implemented, is to have measurement in the interface. The reason for this is that the current time stamping gives a good approximation of the latency as it is. To change and remove this functionality would require more time to complete.



# Chapter 6

## Verification and testing

*The system has been tested in depth to verify functionality. This chapter describes the simulation, synthesis and on chip verification. The simulation has been performed with Active HDL(AHDL) 7.2 SE, the synthesis with Xilinx EDK and ISE 10.1.3 and the on chip tests are written in C and cross-compiled for the Microblaze processor on an Ubuntu PC. First the simulation of the structural development is described. Then a presentation of the MPEG and Cell examples synthesis results before the functional development testing will be presented. No manual floor planning of the design, either pre- or post synthesis have been performed.*

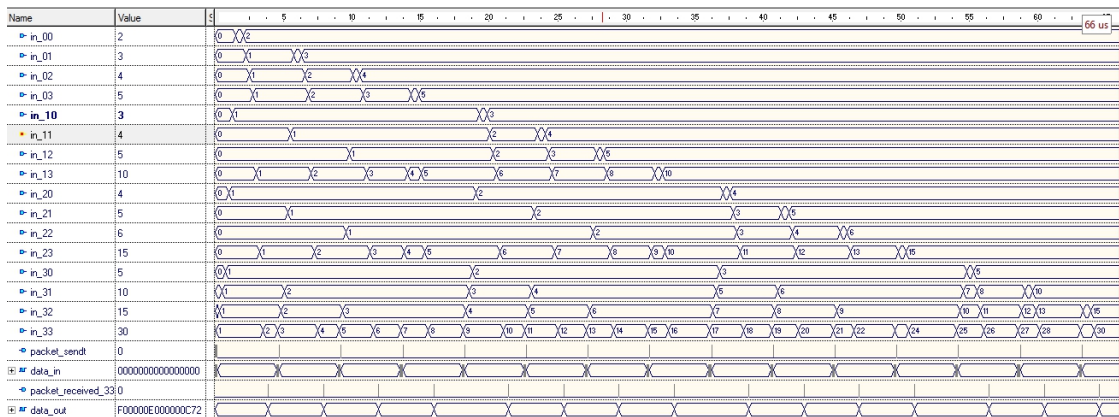
### 6.1 Simulation

The circuit was simulated to verify its behaviour in AHDL. The initial testing was done with a frequency of 50 MHz and with the interface module as top module. This was chosen instead of low level testing of each module since the interesting property to simulate was the behaviour of the entire system. Basic functionality of the table driven test generator was also simulated in [27]. The simulations based on this approach proved to give good results. In simulation the effects of the clock buffer is important. This clock buffer is previously introduced to reduce clock skew in the circuit. Because of this a second simulation clock is added in the circuit with 50 MHz. This was necessary as long as the Xilinx clock buffer libraries are not available in the simulator. More about the clock buffer module is described in

the FPGA manual [4].

### 6.1.1 Simulation of basic functionality

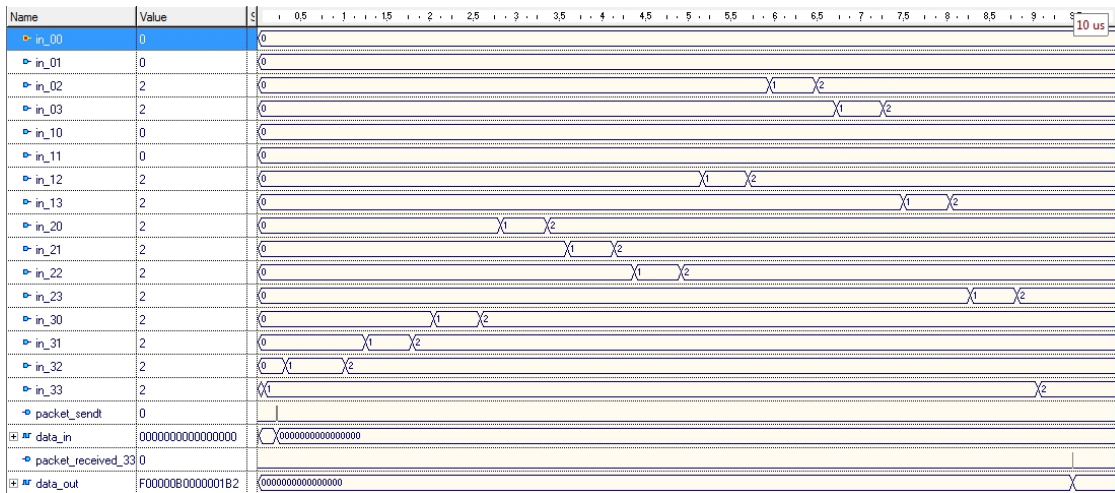
A basic test based on the traffic table was performed, to verify that the interface between hardware and software was functional. This initial test was based on a traffic pattern where all the test generators would send a package back to the interface module and the test would end when this was performed. The packages were sent in accordance with the XY routing scheme and simulated as expected. Simulation results from this test is illustrated in Figure 6.1 Basic simulation figure.captio.50. The framework for this test is a test bench similar to the one described in [27] with packages with specific data sent to different test generators. The system was post simulation, synthesized and implemented on the FPGA.



**Figure 6.1:** Simple test of the system, one package sent to every router which in order responds to the interface.

### 6.1.2 Simple traffic pattern simulation

In the assignment of Hepsø the application of an MPEG decoder was introduced. This serves as a practical implementation applicable to the NoC. The MPEG decoder consists of 11 different modules performing the decoding. For the simulation of the traffic behaviour, the traffic table was configured to be able to send packages in the same fashion as the MPEG decoder would send packages. This was the pattern given in Figure 4.2 MPEG traffic pattern figure.captio.28 and provides



**Figure 6.2:** Mpeg pattern in simulation, one package from each test generator.

a simple platform for the simulation and testing.

In this simulation sending was performed like the MPEG decoder described by Hepsø in [25]. The test was performed by sending an initial package to test generator E/14. When this was done this generator used its table to interpret its action when a package was sent and marked by generator F, in this case the interface. This was to send a package to test generator D and the operation continued around the MPEG pattern until the package arrives back at the interface again. Results from this simulated pattern is depicted in Figure 6.2MPEG simulationfigure.caption.51.

### 6.1.3 Patterns with crossing traffic and deadlocks

A large focus on deadlocks has previously given a focus on testing crossing traffic. This part is not given to much attention since it is already verified with the testing described in [25] what causes deadlocks. Because of this the system tests has mainly focused on the development and testing of applications which simulates "real" behaviour instead of knowingly trying to trigger deadlocks in the system.



## 6.2 Synthesis of the circuits

As mentioned in Section 5.1.1, the main limitation of the circuit relates to the area of the Spartan 3. For this reason it has been chosen area optimization high in synthesis of the circuit. It appears, however, that this choice of optimization has resulted in issues with regards to timing constraints. Especially for the Spartan 3E this was a problem. However, Xilinx includes an Xplore script which tries to run several iterations with different configurations and thus achieves the best timing performance for the circuit. This appears to solve the problem of the timing constraints. The Spartan 3 and 3E are also quite similar when it comes to implementation details so the the NoC is easy to move from one of the platforms to the other.

### 6.2.1 Synthesis challenges

When synthesizing the test generator for the Cell application, a problem with the tables arose. As described in Section 5.5.2, there was a resource problem on the chip that made it impossible to have two different reconfigurable tables within the circuit. This made it necessary to go back in the design and have one of them as a static table and one as a dynamic table. Due to this problem, only the traffic table is possible to change. The table with the next operation test generator as described in Section 5.5.2 is thus not possible to alter with software-generated packages.

In the test generators, arithmetic operations are performed within the circuit. When creating the IP in Xilinx EDK, the default library is a package called "std\_arith". However this is not the standardized library for synthesis given by the IEEE, with "numeric\_std" package as the standard [6]. For this reason, the "numeric\_std" is chosen in implementation, but the arith package was used in some modules from previous projects. The use of both will result in problems in synthesis and must be avoided. Refer to Appendix D for further details about the synthesis and tools.

### 6.2.2 Synthesis of the circuit

By removing some elements of the test generator such as the pseudo random functionality used in Hepsø's [25] assignment it has been possible to add the functionality for the two example systems, and fit the area on the FPGA. It is worth to mention that the systems usage of flip flops and LUTs are lower than the slice number, and this is possibly due to inefficient use of the slices. This could be solved by floor planning, but has not been investigated in the synthesis process due to its requirement of more time. An illustration of the floor-planned design made by the synthesis tool is provided in Appendix A Illustrations of the systems appendix.A. In addition hierarchical system view and the test generator module with input and output is provided.

#### MPEG system(Spartan 3)

In Table 6.1 Synthesis MPEG system table.caption.53, the synthesis results of the entire 16 TG system is listed.

Logic Utilization	Used	Available	Utilization
Number of Slices	185	7680	2%
Number of Slice Flip Flops	225	15360	1%
Number of 4 input LUTs	305	15360	1%
Number of bonded IOB	66	173	38%
Number of BRAMs	1	24	4%
Number of GCLKs	2	8	25%

Values for 16 TG

Number of Slices	2960	7680	38%
------------------	------	------	-----

Entire system

Slices	7,141	92%	7,680
Flip-Flops	7,889	50%	15,360
4-input LUTS	11,399	74%	15,360
Max Frequency	53,387%		

**Table 6.1:** Synthesis results test generator MPEG pattern;from supplied digital attachment: ../sz030-20090319/xps\_ proj.log.

### Cell application(Spartan 3E)

Table 6.2 Synthesis TG Cell-exampletable.caption.55 shows the synthesis results of the Cell example. It shows that with the new functionality, eight test generators take up approximately the same space as sixteen MPEG generators.

Logic Utilization	Used	Available	Utilization
Number of Slices	386	8672	4%
Number of Slice flip flops	402	17344	2%
Number of 4-Input Luts	542	17344	3%
Number of bonded IOBs	66	190	34%
Number of BRAMS	1	28	3%
Number of GCLKs	2	24	8%
Values for 8 Test generators			
Number of Slices 8 Tg	3088	8762	35%
Entire system			
Slices	8308	95%	8672
Flip-Flops	8765	50%	17344
4-input LUTS	12750	73%	17344
Maximum Frequency	58,928%		

**Table 6.2:** Synthesis of the test generator for cell example;from supplied digital attachment: ../sz130-20090319/xps\_ proj.log.

## 6.3 FPGA implementation

### 6.3.1 Initial testing

A simple module to verify the hardware/software interface was developed to check how input from c-code was responded to by the circuit. This is displayed in Figure 6.3 Structural test of the FPGAfigure.caption.56 and shows a sending of a package to router E and it then returns a package to back to the interface. Running this test verified the functionality of the interface.

```

package ready to be sent is e00000f00
which router is to be read from?f
Setting package value
Sending package
packet sent
Number of packages sent through the specified router is 2
Latency of the received package is 45 clock cycles
The data out is now f00000e02d
test finished

```

**Figure 6.3:** Structural test of the FPGA, package sent to test generator E that responds to the interface.

```

package ready to be sent is f00
which router is to be read from?f
Setting package value
Sending package
packet sent
Number of packages sent through the specified router is 2
Latency of the received package is 98 clock cycles
The data out is now f000000062
test finished

```

**Figure 6.4:** Structural test of the FPGA, package sent to test generator 0 that responds to the interface.

### 6.3.2 Testing patterns, initial MPEG test

In Section 4.3 Application example: MPEG decoder section.4.3 and Figure 4.2 MPEG traffic pattern figure.caption.28 MPEG pattern was introduced. This system was tested on the FPGA with a simple test. Testing the system was performed by sending one package through the network from test generator F to E and through eleven test generators back to F like in Figure 4.2 MPEG traffic pattern figure.caption.28. It proved that the time a package used to travel from interface through 11 routers/generators was a somewhat higher time than in the simulation. The difference was however not significantly large with a number of cycles of 435 for the FPGA and 434 for the simulation, a difference of 1 clock cycles. Figure 6.5 Simple MPEG FPGA figure.caption.58 shows the results from the on-chip test while Figure 6.2 MPEG simulation figure.caption.51 shows the corresponding simulation results. This test was mainly performed to get a verification of the table defined traffic and to see which improvements that could be made.

With this test one had to take in to consideration the fact mentioned in Section 5.7.1 Measuring throughputs subsection.5.7.1 that the stamping of packages occurs when leaving each test generator. This means that the number of cycles from

```

This test tests functionality of the circuit with different user input

Which test generator do you want to send a package to? (0 to E)
E
Setting send to address toe

What type of package do you want to send: (0,4 or 8 )
0
Setting type to:0
package ready to be sent is e0000f00
which router is to be read from?f
Setting package value
Sending package
packet sent
Number of packages sent through the specified router is 2
Latency of the received package is 435 clock cycles
The data_out is now f0000b01b3

```

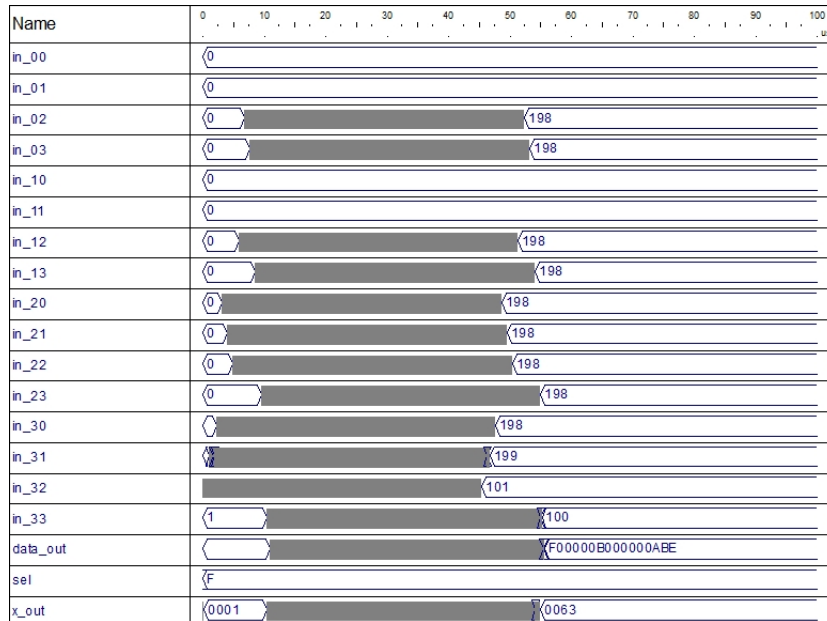
**Figure 6.5:** Simple MPEG run on the FPGA.

the package leaves the interface, until it arrives back to the interface is a larger number than the number given here. An estimation based on the basic test in Figure 6.3 Structural test of the FPGA figure.captio.56 is about 45 cycles. Since each packet transfer takes about 12 cycles through each router and the last test generator B has to send its package through two routers before reaching the interface. That is the same number as in the mentioned basic test. This number is found by looking at the initial time stamping of the global time in the closest test generator, as seen in Figure 6.3 Structural test of the FPGA figure.captio.56. It is however not a fixed number for all cases since it may increase if there is crossing traffic through the specific routers. Approximation of the latency is, however, found to be good enough.

### 6.3.3 Higher packet rate

To see how the circuit performs when applying a higher rate of packages, a table configuration that generated multiple packages in router E was enabled. That gave the opportunity to see how the system performed when sending several data packages at the same time. The procedure was as follows; a package was sent to test generator E from the interface and this test generator generated one hundred packages that followed the MPEG pattern to see if performance was affected. This test is similar to sending one hundred packages from the software to the network and

works as a test of how a data stream i.e. a video stream would behave. The test proved that the performance on the FPGA, as seen in Figure 6.6MPEG 100 packages simulationfigure.caption.59, was similar to the simulation in Figure 6.6MPEG 100 packages simulationfigure.caption.59. Almost the same latency of 2751 clock cycles measured, one more on the FPGA test than in the simulation. No deadlocks or other issues occurred in this test.



**Figure 6.6:** Mpeg pattern in simulation, one hundred packages from test generator E through the system. NOTE last for digits in data\_out ABE.

## 6.4 Cell processor application

After completing the MPEG pattern test and verification, the Cell example was developed based on the MPEG design. This system contains the same functionality as the MPEG system. As mentioned there are three operating modes. In addition to the basic traffic functionality verified in Figure 6.8Traffic Cell example, FPGA testfigure.caption.61 there is a table reconfiguration and data manipulation mode.

**Table reconfiguration** Testing the table reconfiguration mode proved to be a larger challenge than expected. Initially, the on-chip test was intended to be a

```

1
The control/status register is initially:0
Send default package E00000F000000000?(Y/N)
y

the packages sent is: e00000f0 0
Setting package value
Sending package
new latency2751
press 0 to stop input on circuit
0
Sending package
initial test with f8001c8
number of packages sent through the interface router is 100
latency of the received package is 2751 clock cycles
The status register is now f8001c8
The data out is now f00000b0abf
test ending
Reset circuit

```

**Figure 6.7:** Mpeg pattern run on the FPGA, one hundred packages from test generator E through the system NOTE data\_out last four digits ABF.

```

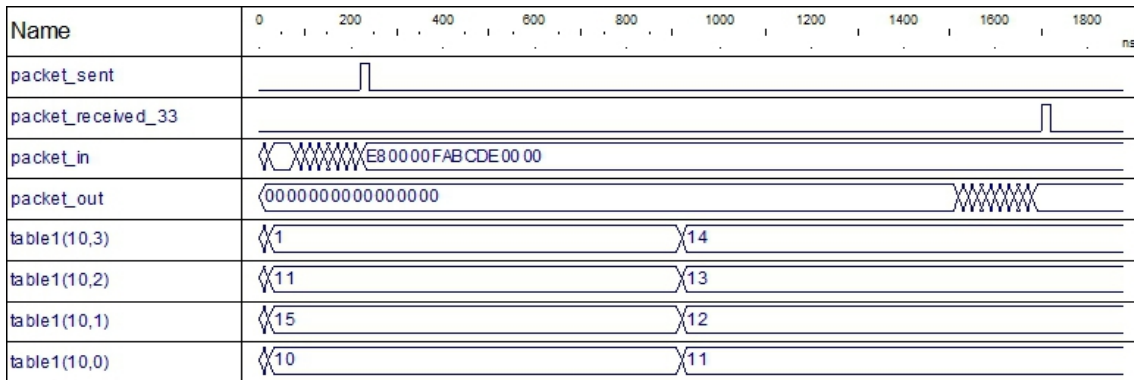
Which test generator do you want to send a package to? (0 to E)
E
Setting send to address toe

What type of package do you want to send: (0,4 or 8 ) necessary informa
0
Setting type to:0
package ready to be sent is e00000f00
which router is to be read from?F
Setting package value
Sending package
packet sent
Number of packages sent through the specified router is 2
Latency of the received package is 286 clock cycles
The status register is now f80001c8
The data out is now f00000b011e
test finished

```

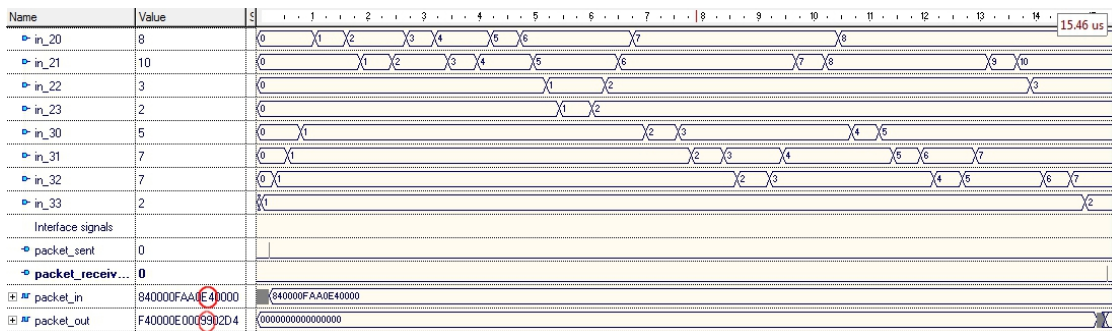
**Figure 6.8:** FPGA test of simple traffic in the cell example system; Pattern F->E->D->C->8->9->A->B->F.

test with changing traffic from one circle through the Cell system to a continuous circle with counting of the packages. The test did not function as intended due to its requirement of continuous surveillance of the NoC and this functionality is not developed in the test system. However, the functionality was verified in simulation as shown in Figure 6.9 Simulation table reconfigurationfigure.caption.63. It shows that the functionality is correct, but needs a better measuring tool to run a full scale test on-chip.



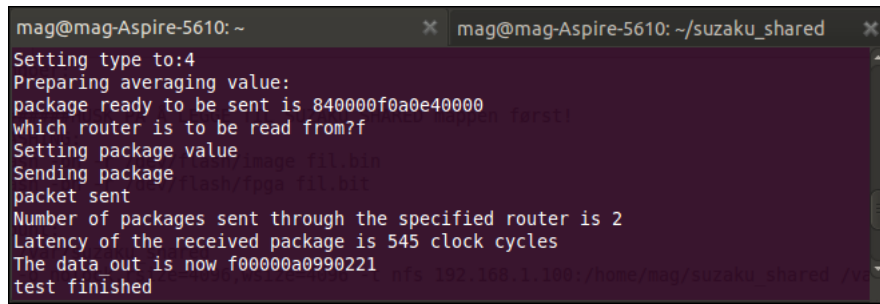
**Figure 6.9:** Simulation table reconfiguration; Package sent to generator E, table position 10 changed with new values, package returned to the interface.

**Data manipulation test** In addition to the two mentioned modes, the Cell example contains a small computer that computes the average value of two hexadecimal numbers. In this way this example works much like a co-processor that does one specific operation for the Microblaze processor. Testing was performed by sending a package with two hexadecimal values and their difference to any test generator from 8 to F, as in Figure 4.4Cell processor in NoCfigure.caption.31. The package was then forwarded to the next operation test generator and followed a path defined by the test generators internal tables. When the calculation, as described in Section 5.5.2Data Manipulationsubsection.5.5.2 was performed, the package was returned with the average value of the two input values to the interface and software. Global time was added to indicate latency of each package and then read out and interpreted by the software.



**Figure 6.10:** Simulation of the calculation of one average value by using the network and test generators, here; the number  $\frac{E+4}{2} = \frac{18}{2} = 9$ .





```

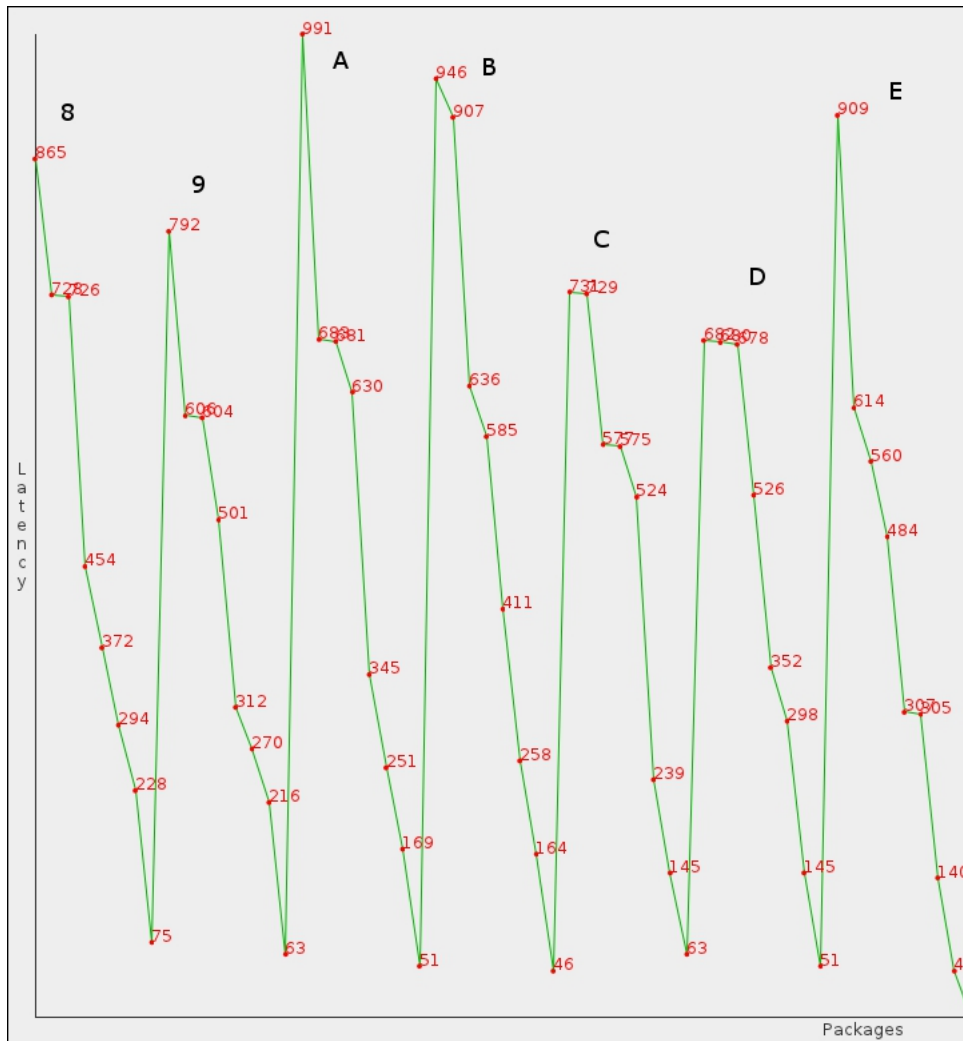
mag@mag-Aspire-5610: ~
Setting type to:4
Preparing averaging value:
package ready to be sent is 840000f0a0e40000
which router is to be read from?f
Setting package value
Sending package
packet sent
Number of packages sent through the specified router is 2
Latency of the received package is 545 clock cycles
The data out is now f00000a0990221
test finished

```

**Figure 6.11:** Calculation of one average value on the FPGA, same values as in Figure 6.10 Simulation average, one package figure.captio.65.

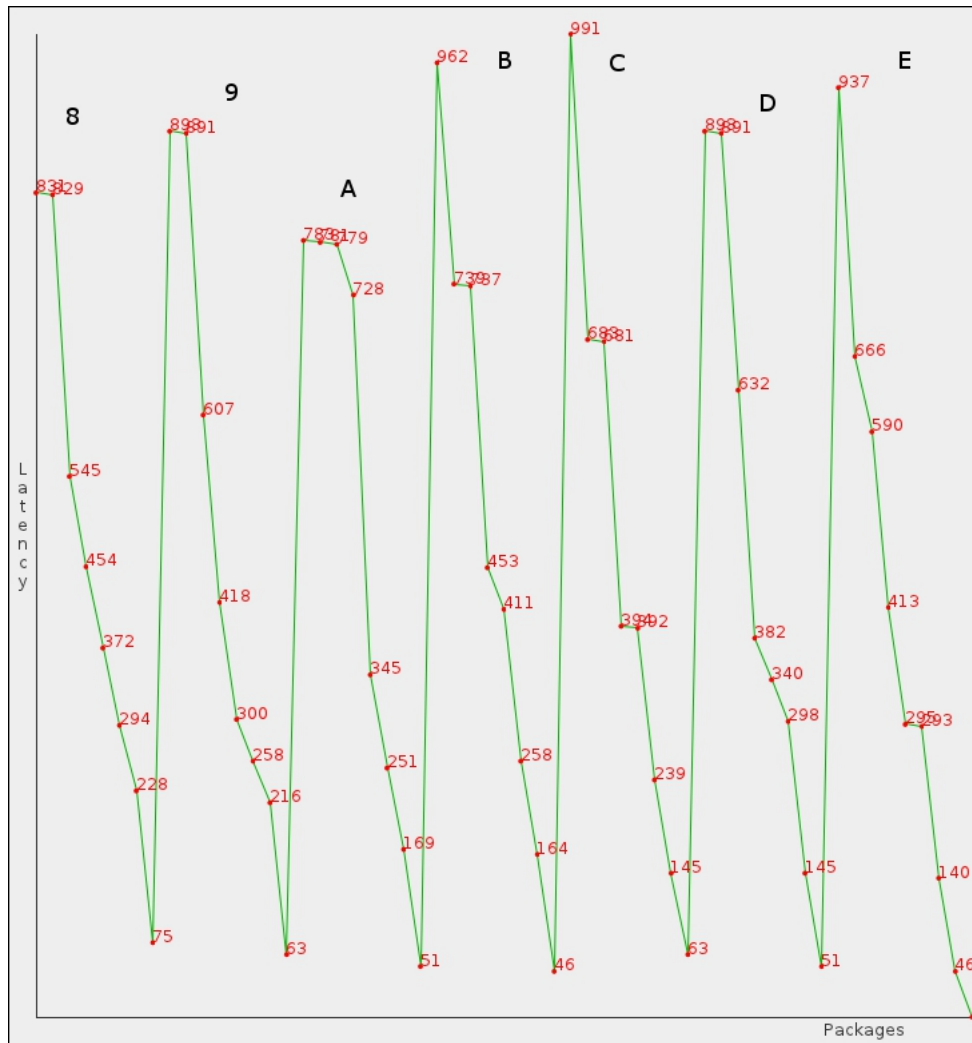
A large scale test was developed to thoroughly test the application on-chip. The test consisted of sending packages with hexadecimal numbers to each module. First, one table configuration was used with the table configured as in Table 5.8 Cell application example table.table.caption.48. In this table, test generator A was not used in the table, to see how this had an impact on the system. This one was chosen because of its central position in the system. The vectors applied as load to the circuit is listed in Table 6.3 Test loadtable.caption.69. The results from this test is displayed in the graph in Figure 6.12 Graph of latencyfigure.captio.67, based on all the latency values from the circuit. For this test, the worst case scenario with a large difference creates at the most 991 clock cycles of latency when sending to test generator A, while the best case is when there is sent a package to the test generator D. Latency for initial sending to generator D was considerably lower for the two first loads than for the rest of the loads. Differences in latency between modules are based on differences in the traffic table of the test generators. This configuration is not optimized and the distance to the next test generator might be in some cases be large. This is probably the case when sending to test generator A,B and E in this test. However, it indicates well the best- and worst case throughput of the Network on Chip with type of system.

To compare this result, a second table configuration was tested with a table configuration with minor changes. Table values for the second test is found in Table 6.4 Cell application example table.table.caption.70 and the results are illustrated in Figure 6.13 Graph of latency, second table configurationfigure.captio.68. Two test generators table values were changed and test generator A was added to



**Figure 6.12:** Data from first test run on FPGA, 8 packages with data values (Table 6.3 Test load table caption.69) sent to each test generator; Hexadecimal number indicates router, Number indicates Latency in clock cycles.

the tables. This gave a more even result for all the generators. In this test the maximum throughput is higher than for the first test with a maximum value for initial test generator A of 1005. Best case was, in this test, found when sending the initial package to test generator A. When sending the initial package to generator C the difference is largest, this indicates that the path the package follows is more optimal for the first test case. When the computation is small, the differences are smaller for all the test generators since the test generator just sends the package



**Figure 6.13:** Data from second test run on FPGA, 8 packages with data values (Table 6.3 Test load table caption.69) sent to each test generator; Hexadecimal number indicates router, Number indicates Latency in clock cycles.

in return.

Similar experiments have been performed in testing of the EIB on the cell processor with sending of packages from SPE to SPE [13, p.2].

Vector(Hexadecimal values)	Difference(Decimal values)
F1	14
E2	12
D3	10
C4	8
B5	6
A6	4
97	2
88	0

**Table 6.3:** Test loads for the cell application, on chip test.

Test Generator	Next operation TG	Number of packages	VHDL table	Change
8	9, 11,13	1	(8,9,11,13,1)	
9	8,10,14	1	(9,8,10,14,1)	X
A	11,9,13	1	(10,11,9,13,1)	
B	12,10,14	1	(11,12,10,14,1)	X
C	13,9,11	1	(12,13,9,11,1)	
D	14,8,12	1	(13,14,8,12,1)	
E	9,13,11	1	(14,9,13,11,1)	

**Table 6.4:** Cell application example table.

### 6.4.1 Streaming test of the application

To evaluate the ability to handle more than one package at the same time into the system, a stream test simulation has been performed. The stream test consist of applying a continuous stream of packages to the system in a manner similar to the full system test on the FPGA. Values to compute average from was the same as the highest difference values in Table 6.3Test loadtable.caption.69. Sending procedure was to test generator 8, then 9, A and so forth. With this test it is interesting to see whether there are any lost packages from deadlocks or not. This test forces the system to work on multiple data at the same time and to handle potentially colliding data into the routers. Hence, the test was performed with the expectation that loss of packages would occur when any router received more than three packages.

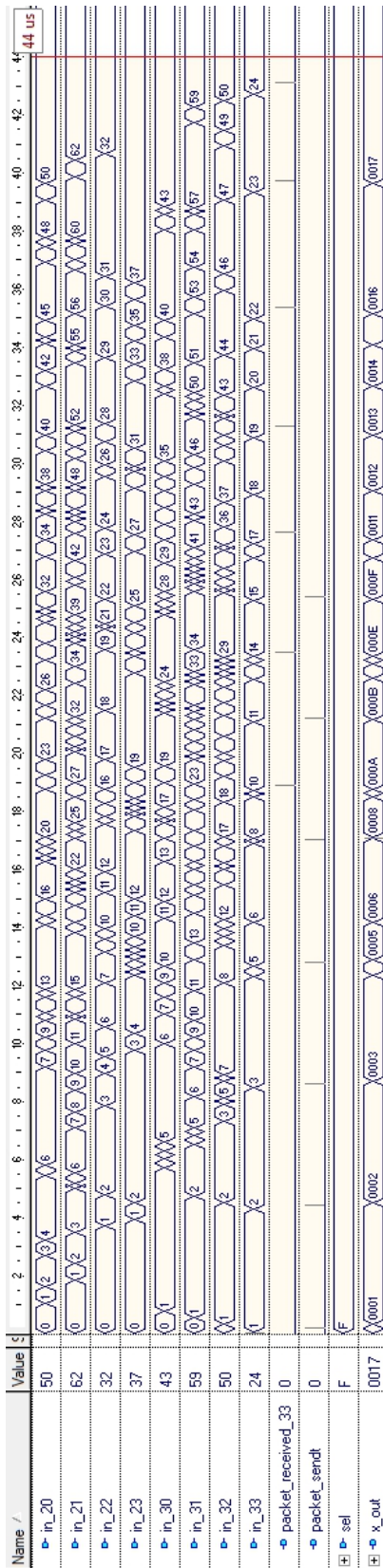
That also proved to be the case, when testing data streams with different

sending interval. It proved to be a matter of how small the interval between each package was, when the system deadlocked or not. In the Figures 6.14(b)Subfigure 6 6.14(b)subfigure.6.14.2 and Figure 6.14(b)Subfigure 6 6.14(b)subfigure.6.14.2 different intervals was chosen. In the first situation 200 clock cycles went between sending while in the latter only 20 cycles went between sending. This states the fact described previously that three packages to a single router will deadlock the system. This also proves one of the areas where this system might be a good tool. By setting parameters to correct values like in Figure 6.14(a)Subfigure 6 6.14(a)subfigure.6.14.1 one could also know what values the real system has to have in implementation. In addition the stream test shows what routers has the highest traffic flow. In the test in Figure 6.14Stream testsfigure.caption.71 the router A(in\_22) has almost twice the amount of packages routed than router 9(in\_21).

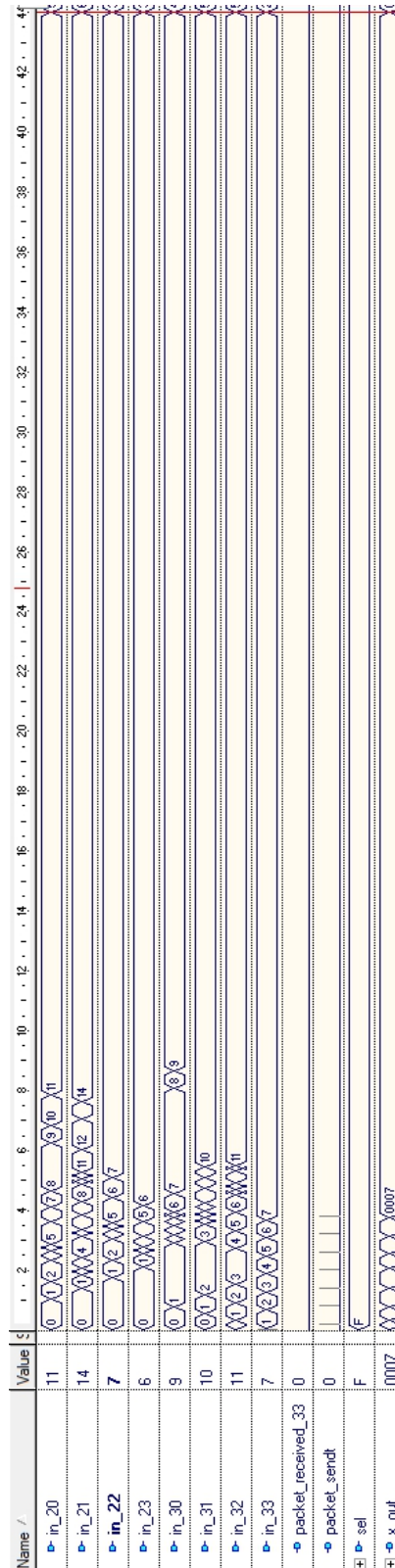
There is not found a good way to perform this test with the C program. A possible solution is introducing interruption mechanisms in to the system or organizing the sending procedure in a different way than is implemented at the moment. Both would require changes to the interface, possibly an additional interface.

## 6.5 Summary of testing

Three different systems have been tested. Initially the testing focused on structural testing verifying the interface communication. When this testing was completed, the functional testing proved how the table driven test generators mimic traffic of a real application with the MPEG and Cell example. Finally the full system tests were presented with sending to all generators and measuring of latency. The streaming test concluded the testing and illustrated some of the limitations, but also an important aspect in the test system use. Testing of the table reconfiguration mode was not fulfilled on the FPGA, but verified in simulation.



(a) Streaming test of cell example, successful run 7 packages sent and received; with 200 cycles sending interval



(b) Streaming test of cell example, deadlocked system;7 sent none received 20 cycles sending interval

Figure 6.14: Stream test of the Cell application example.



# Chapter 7

## Discussion

### 7.1 Evaluation of the system

The ambition of this project has been to develop a platform for better testing and evaluation of the NoC and applications for the NoC. By defining measurable parameters such as throughput and latency, a test system with the ability to run like a normal system has been produced. With the development of the system, two examples has been introduced to illustrate the use of the system. However, the main focus has not been to use the system to compare it with another system, but to illustrate how the system applies in testing to the NoC.

Development of the table driven test generator has proved that with the current system it is possible to imitate the behaviour and traffic pattern of several applications. By simply changing the table values, the system has been used to both mimic an MPEG decoder and a small simple processor application. This gives a new dimension to automation of the profiling of applications on the NoC. System structure and performance, and functionality has been verified in different tests. Basic functionality and traffic between modules are now in place. Programmability has been a focus of the development, new interface and enhanced use of software programs to read data from the chip, has provided increased possibility of testing the NoC. Having this in place, a framework for extensive testing of the NoC has now been in place capable of benchmarking different implementation alternatives for the NoC. Although there has been made improvements to the interface and



readout from the FPGA, the failed attempt to run a full table reconfiguration test in Section 6.4Cell processor applicationsection.6.4 indicates that it needs more development. A possibility is, as mentioned in Section 6.4.1Streaming test of the applicationsubsection.6.4.1, to implement interruption mechanisms and a second interface.

Two examples of how the test system could be used to profile applications are the MPEG example and the Cell example. Both are based on table driven functionality, but focus on different areas.

### 7.1.1 MPEG example

Results from simulation and FPGA implementation of the MPEG system proves that the NoC is capable of running a streamed application. In this aspect it is very important to emphasize the placement of modules. Using a placement like in Figure 4.2MPEG traffic patternfigure.caption.28 avoided deadlocks and managed to handle a continuous stream of data. Even though this test system did not contain any computation the test is sufficient to prove that the AHEAD NoC can handle this type of application.

### 7.1.2 Cell example

With the Cell example, introducing computation in addition to communication is the big difference from the MPEG example. Having this dimension in the test and profiling tool enables the ability to mimic a wide range of applications. Its importance became clear both in the FPGA test and the streaming simulation. By setting the parameters of the system to correct values, one could evaluate whether or not deadlock in the system will occur. In addition different table configurations, and in this aspect traffic pattern, provided information about the most efficient solution with regards to latency and throughput. To improve the details of the results, such as the latency, optimization of the tables could have been performed, but this will require more time. In any case, the possibilities of the system as a profiling tool has been illustrated with the performed tests.

## Resources

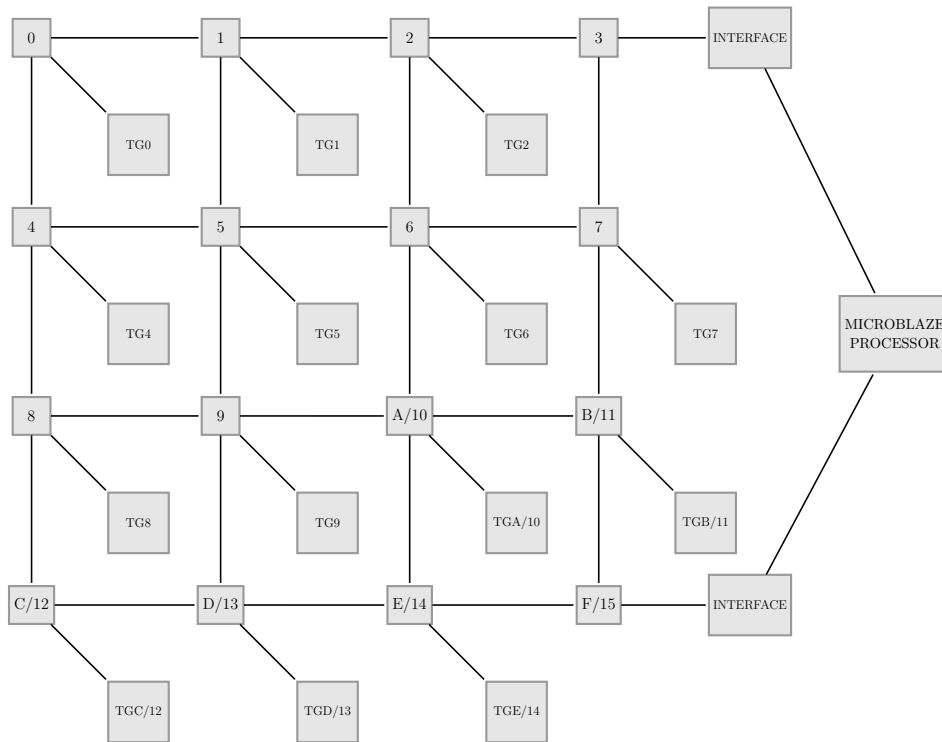
With the extension of the functionality of the test generators the area used on the FPGA was 95%. This means that the system has little or no room for more functionality. By changing the number of routers in the system this problem might have been overcome, but this improvement requires more time to investigate. An alternative of adding more functionality could be to have submodules for the operating modes, and change to a different module when testing different functionality. A final and obvious alternative is to use a FPGA platform with more resources. The test system is now scalable and with a scalable NoC, to fully exploit a new platform in a simple manner will be possible.

### 7.1.3 Interface as bottleneck

The system connections to software with only one interface has been detected as an obvious bottleneck of the system, since all data has to go through one eight bit channel. To improve and further develop the functionality of the test system it is advantageous to add one or more interfaces to the system. This will contribute to the overall bandwidth of the system since the maximum amount of data entering the NoC will be sixteen bits instead of eight. Hence potentially doubling the bandwidth for the NoC. In addition it could provide possibility of more advanced test cases. As an example, if a second interface, as in Figure 7.1Two interface modulesfigure.caption.73 two Cell systems could be implemented at the same time. The implementation could simply be done by adding a second interface module to the user logic file of the NoC project. This is not pursued further due to the focus on the test system, but will be possible to do by duplicating the interface and exchange one of the test generators with this module. Such a system will, however, complicate the communication between the MicroBlaze processor and the NoC, but with efficient organization, this will still be an improvement.

## 7.2 Throughput results

To illustrate how a desired implementation can be profiled and optimized with the test system, throughput results are essential. They give a numeric value of how the



**Figure 7.1:** Two interface modules connected to the NoC.

performance of the system is compared to its optimal potential. An illustration of this is provided in the following calculations of throughput of the MPEG system, and the worst case throughputs of the Cell example.

### 7.2.1 MPEG

Ersland [17] concluded in his Masters Thesis and preliminary project, that the system would theoretically have a bandwidth of 1,3 GB/s. The results depended on a higher frequency than what was able to achieve with the test generator system. Hence, given Equation 3.2 System properties equation.3.3.2, a more probable bandwidth was given for the loaded case of the routers. A property of the circuit which is important to evaluate is the throughput of the circuit. When sending 100 packages through the circuit the latency was found to be 2751 clock cycles. If using a frequency of 50 MHz, with a period of 20 ns per cycle, this gives the

following calculations of the throughput:

$$2751 \times 20ns = 55,02 \times 10^{-6}s \Rightarrow \frac{100}{(55,02 \times 10^{-6})} = 1817520 \frac{\text{pkg}}{\text{second}} \quad (7.1)$$

With a package size of 64 bits, the throughput of this configuration will be

$$1817520 \times 64 = 116\text{Mbps} \quad (7.2)$$

which is equivalent to

$$\frac{116}{533} = 21\% \quad (7.3)$$

of the total bandwidth.

Comparing this best unloaded case with the defined bit rate defined for HD MPEG 2 decoding at 18 Mbps for the Cell processor [13], indicates that the necessary throughput of such an application is achievable. The question is however in the loaded case how the development of the worst case throughput will be. To get an exact number for this case the same test has to be run on an MPEG decoder and scaler, but this is not feasible at the moment due to area constraints on the Spartan FPGAs.

### 7.2.2 Cell application

As shown in the loaded case for the Cell application, the worst case latency for the circuit is with a frequency of 50 MHz and a period of 20 ns:

$$991 * 20ns = 1,98 \times 10^{-5}s \Rightarrow \frac{1}{(1,98 * 10^{-5})} = 50454 \frac{\text{pkg}}{\text{second}} \quad (7.4)$$

This gives a throughput of the system of:

$$50454 \times 64 = 3,2\text{Mbps} \quad (7.5)$$

Which is, as expected, considerably lower than the throughput of the MPEG system. This is then

$$\frac{3,2}{533} = 0,6\% \quad (7.6)$$

of the total bandwidth. It is a low utilization of the NoC system, but could be enhanced if one sends several packages at the same time into the network, or optimize the sending of packages. To illustrate this fact, considering the best case scenario given for the largest difference to compute in Figure 6.12 Graph of latency figure.captio.67. Sending to test generator D gave a latency of 682 cycles which would mean, given same computations as in Equation 7.4 Cell application equation.7.2.4, a throughput of 4,7 Mbps which is the same as 0,9% of the total bandwidth of the system.

### 7.2.3 Stream test

The stream test shows this implementation for the cell example, it shows that it is possible to send more than one package into the system like in a stream manor. The MPEG example proved that the throughput of the system is considerably higher when there is a stream of data instead of one and one package<sup>1</sup>. The problem with the cell example when considering streams of data is the deadlock problem. The loss of packages due to deadlocks should be avoided. A method to counter counter this problem with the current routing in the NoC is to adapt the incoming rate of packages. Another method is to place modules and their connected test generators in an optimized way. That means using the tables actively, defining a placement that will avoid three modules sending packages to one router at the same time.

Throughput of the streaming test is another case, as described in Section 7.2.2 Cell applications subsection.7.2.2 the throughput percentage of the total bandwidth is low. The stream test forms a way of increasing the throughput for the as seen in Figure 6.14 Stream tests figure.captio.71 eight packages is completely calculated with a load in 43  $\mu s$ . With this number of packages we get the following:

$$\frac{8}{43\mu s} = 0,18\mu s = \frac{180000pkg}{second} \quad (7.7)$$

---

<sup>1</sup>A package is sent only after one has been received at the interface

$$\Rightarrow 64bit \times 180000pkg = 12,8Mbps \Rightarrow \frac{11,5Mbps}{533Mbps} = 2,1\% \quad (7.8)$$

more than three times the consumed bandwidth as when only sending one package at a time. This is an approximation since this configuration has not been implemented on the FPGA. However, previous tests have shown that there is large consistence between simulation results and on-chip tests.

The stream test indicates a good way of using the test generators to profile a system desired to implement on the NoC both with regards to throughput and of system traffic. As described in Section 6.4.1 Streaming test of the applicationsubsection.6.4.1 the test system can be applied to see what routers that have a heavy load and those that have little. By analyzing this data, system traffic could be redirected in a way that it is more evenly distributed than is the case in the provided test in Figure 6.14 Stream testsfigure.caption.71.

### **Best case vs Worst case Throughput**

With the cell application run on the FPGA, a loaded test case was introduced. This case is interesting since there is a significant difference between best and worst case throughput based on module placement. With the streaming test it also proved that the speed can be increased, without deadlocks, with correct system parameters i.e. sending interval. These tests show that there is a gain from optimization for the overall throughput of the system. It is something to bear in mind when using the system to determine desired properties and corresponding placement. A relevant question is also if it is desired with a best case latency or a low average case. It is not a given that a high throughput system for large computations is the globally optimal solution for a given system. Especially when the implemented system handles more than one package at the same time, this is very important with regards to the deadlock problem.

### 7.3 Application module placement

The basic question when implementing a distributed system on a NoC is "Which placement of the modules in the network on chip is optimal and feasible?" Based on the results from testing the MPEG decoder pattern and the mimicked version of the Cell processor, there are some interesting aspects to consider. The system has proven to function when used as a processor or co-processor by placing the modules in a cluster or close to each other as in the Cell case. Computation executed on the FPGA will in most cases outperform the processor when it comes to one single operation. The need to communicate with more than one module also strengthens the gain by having modules closely connected to each other. This clustering technique was a concept that was introduced for the AHEAD NoC in [27] and has proven to some extent valid in the cell application example test.

However, if the system is to be used for different purposes and real time applications, i.e. video decoding, a more "snake" like placement is better to consider, as proven with the MPEG decoder example. The system will not be able to handle to high crossing data traffic if there is a continuous stream of data into the system due to deadlocks. On the other hand, as shown with the stream example, it is possible to avoid the problem by streaming data with a specific interval or better distribution of traffic.

For the AHEAD system, the motivation is to provide an extra computational power to assist small devices with limited resources. In this aspect, a distributed computational resource such as a Cell processing elements is a good idea. It could assist in computing advanced functions by using several modules connected with a NoC. It could also assist the reconfiguration operation with a proper reconfiguring tool based on the NoC properties as a scalable and efficient interconnect. With a proper profiling of the system, the reconfiguration knows exactly what module to replace and where its optimal placement is.

### 7.4 Further use of the test system

The test system is, as mentioned, able to mimic and profile applications. The usage of this could be quite clear. By changing the table values of the test generator it

is be possible to test a various number of applications without having to change the design in more than one module. If it is desired to implement a specific functionality it is done by simply change the state machine and the containing states of it as described in Chapter 5Requirements and designchapter.5. Then one can implement the operating modes or operations required for a given application test. This testing also opens a new possibility with regards to the development of the NoC itself. The discussion in previous work of changing the routing itself could be taken further when a proper testing tool is at hand. By using the test generators as a benchmark, a new version of the NoC could easily be compared to the old one by simply connecting it to the test system and measure the latency.





# Chapter 8

## Conclusion

The primary motivation for conducting the work with the NoC was to take the project from a semi finished test system to a functional platform for NoC testing. The use of table driven modules to develop full scale tests for the NoC has proven successful and given answers about module placement and ideas for further development of both the test system and the NoC itself. With a simple and flexible design that is easy to use, the ability as a development platform for further use is also in place. Programmability has been taken one step further and the surveillance of the on chip traffic is possible with a functional interface.

With the MPEG pattern and the Cell example it has been shown that there is a significant connection between type of system and optimal module placement in the NoC. The MPEG test system illustrated that streaming of data is possible and that it favours a sequential placement with no possibility of crossing traffic to avoid deadlocks. With the Cell system it was illustrated that a system with communicating modules favours closeness between modules due to communication overhead. The test system is in this way a good tool to profile these different types of systems and their optimal placement.

In addition, the project has illustrated the need for further development with regards to monitoring traffic into and out of the network. The communication between the processor and the NoC has been also been detected as a bottleneck and possible improvement.



# Chapter 9

## Further work

As discussed in this report, some work is necessary to improve and to develop for the test system and the NoC. After the work on this assignment these are the most relevant improvements suggested;

- A second interface to the circuit for readout and increased bandwidth
- Enable interruption mechanism to improve readout from the FPGA
- Stream data from processor to NoC
- Extend the table to create more complex traffic possibilities
- Use test generators to profile and implement a real application
- Create a generic and flexible NoC
- Reduce the number of routers in the network

If proceeding with the two last improvements, it is possible to develop and test a router design with more advanced routing and arbitration.



# Bibliography

- [1] *Open Core Protocol Datasheet v3.0*.
- [2] Xilinx forum. <http://forums.xilinx.com/t5/EDK-and-Platform-Studio/using-simple-EDK-project-in-the-ISE/td-p/45632>.
- [3] *Operating systems: Design and principles*. Prentice Hall, 2008.
- [4] Xilinx spartan-3 fpga family data sheet. 2008.
- [5] Xilinx spartan-3e fpga family data sheet. 08 2009.
- [6] Description of numeric\_std package, June 2011. [http://www.doulos.com/knowhow/vhdl/\\_designers/\\_guide/numeric\\_std/](http://www.doulos.com/knowhow/vhdl/_designers/_guide/numeric_std/).  
\bibitem{wikiping}Descriptionofpingonwikipedia, 2011.\newblock\urlhttp://en.wikipedia.org/wiki/ping.
- [7] Einar Aas. Testing of digital circuits, 08 2009. Lecture in the course 4175 Realization and Test of Digital Component.
- [8] AHEAD. Ambienthardware. WEB, 2006. <http://www.ambienthardware.com>.
- [9] Thomas William Ainsworth and Timothy Mark Pinkston. Characterizing the cell eib on-chip network. *IEEE Micro*, 27:6–14, September 2007.
- [10] Christophe Bobda, Ali Ahmadinia, Mateusz Majer, Jürgen Teich, Sándor P. Fekete, and Jan van der Veen. Dynoc: A dynamic infrastructure for communication in dynamically reconfigurable devices. In *FPL*, pages 153–158, 2005.

- [11] B. Callaghan. *NFS illustrated*. Addison-Wesley professional computing series. Addison-Wesley, 2000.
- [12] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation; a performance view. *IBM Journal of Research and Development*, 51(5):559–572, sept. 2007.
- [13] Partha Pande Axel Jantsch Erno Salminen Umit Ogras Radu Marculescu Cristian Grecu, Andrè Ivanov. An initiative towards open network-on-chip benchmarks. 2007.
- [14] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. pages 684–689, 2001.
- [15] Zhonghai Lu Erno Salminen, Krishnan Srinivasan. Ocp-ip network-on-chip benchmarking workgroup. 2010.
- [16] Ivar Ersland. Quality of service for network on chip. Master’s thesis, NTNU, 2009.
- [17] Ivar Ersland and Kjetil Svarstad. *Quality of Service for Network on Chip*. Project report, NTNU, 2008.
- [18] B. Flachs, S. Asano, S.H. Dhong, H.P. Hofstee, G. Gervais, Roy Kim, T. Le, Peichun Liu, J. Leenstra, J. Liberty, B. Michael, Hwa-Joon Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D.A. Brokenshire, M. Peyravian, Vandung To, and E. Iwata. The microarchitecture of the synergistic processor for a cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):63 – 70, jan. 2006.
- [19] Institutt for elektronikk og telekommunikasjon. *TFE 4170 Enbrikkesystemer, Laboratorieoppgave*. NTNU, Trondheim.
- [20] B.A. Forouzan and S.C. Fegan. *Data Communications and Networking*. McGraw-Hill Forouzan Networking Series. McGraw-Hill, 2003.
- [21] Philippe Garrault and Brian Philofsky. *HDL Coding Practices to Accelerate Design Performance*. Xilinx, 1.1 edition, January 2006.

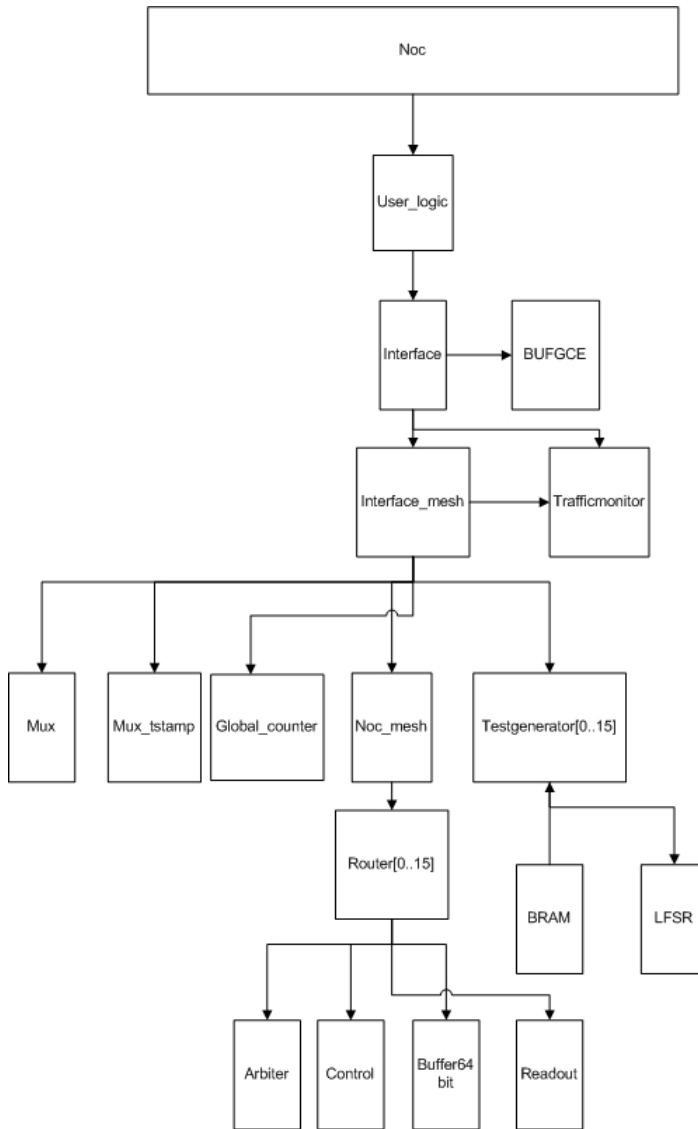
- 
- [22] Jean-Christophe Glas and Kjetil Svarstad. A noc on xilinx spartan fpga. Master's thesis, NTNU, 2006.
- [23] Ahmed Hemani, Axel Jantsch, Shashi Kumar, Adam Postula, Johnny Öberg, Mikael Millberg, and Dan Lindqvist. Network on a chip: An architecture for billion transistor era. *Proceedings of the IEEE*, 2000.
- [24] Andreas Hepsø. Utvikling av testmiljø for network on chip. Master's thesis, NTNU, 2010.
- [25] Bob Larson. Bandwidth vs throughput, 2007. Note on Bandwidth and Throughput.
- [26] Magnus Namork. Reactive test generators for network on chip. 2010.
- [27] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
- [28] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21:120–126, February 1978.
- [29] B. Salminen, T. Kangas, J. Riihimäki, and T. D. Hämäläinen. Requirements for network-on-chip benchmarking. In *23rd NORCHIP Conference 2005*, volume 2005, 2005.
- [30] Kjetil Svarstad Stig Petersen, Dag R. Rognlien. Context tag technology. 2003.
- [31] Xuan-Tu Tran, Yvain Thonnart, Jean Durupt, Vincent Beroulle, and Chantal Robach. A design-for-test implementation of an asynchronous network-on-chip architecture and its associated test pattern generation and application. *Networks-on-Chip, International Symposium on*, 0:149–158, 2008.
- [32] Xilinx. [http://www.xilinx.com/itp/xilinx5/help/xpower/html/d\\_definitions/d\\_clock\\_skew.htm](http://www.xilinx.com/itp/xilinx5/help/xpower/html/d_definitions/d_clock_skew.htm). Definitions of terms for Xilinx FPGAs.
- [33] Xilinx. *MicroBlaze Processor Reference Guide*, 9th edition, 2008.





# Appendix A

## Illustrations of the systems



**Figure A.1:** Modules presented in hierarchy.

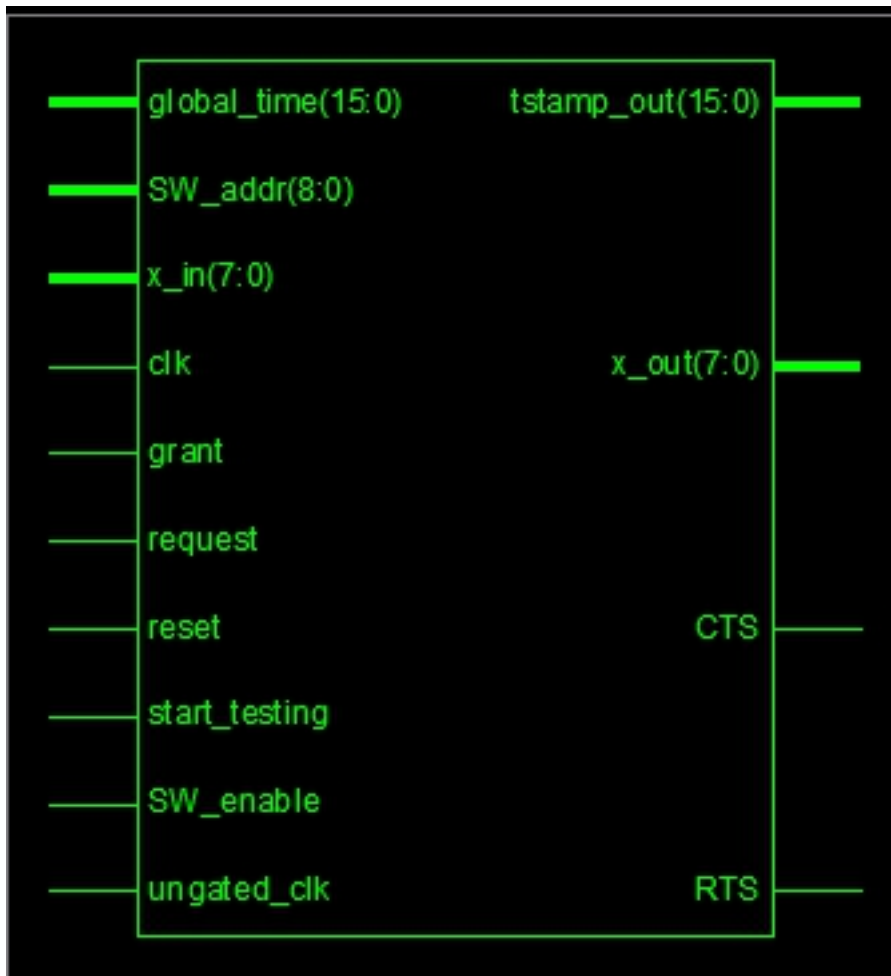
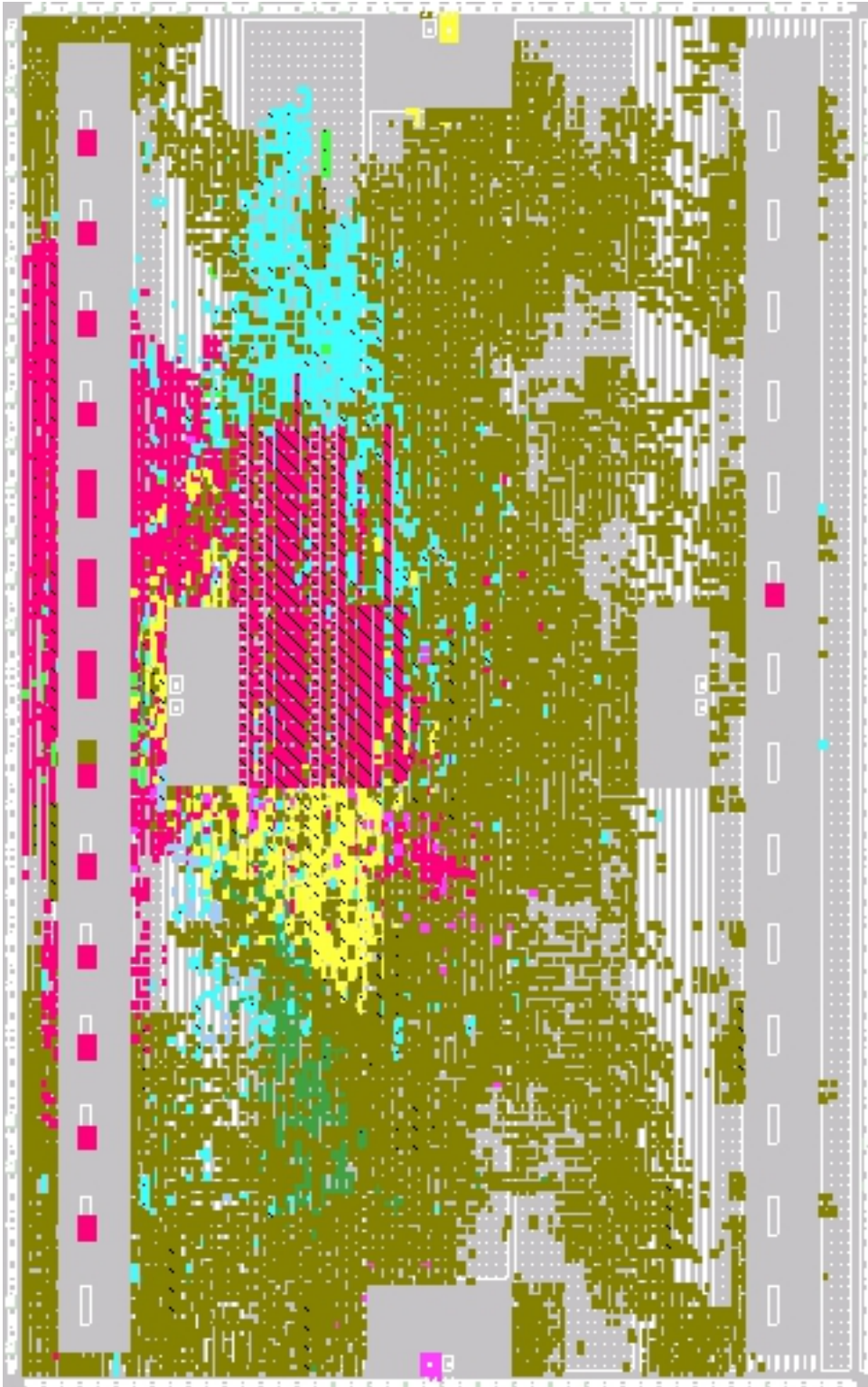


Figure A.2: Test generator block.



**Figure A.3:** Floorplan of the network on chip, from floorplanner in Xilinx ISE.

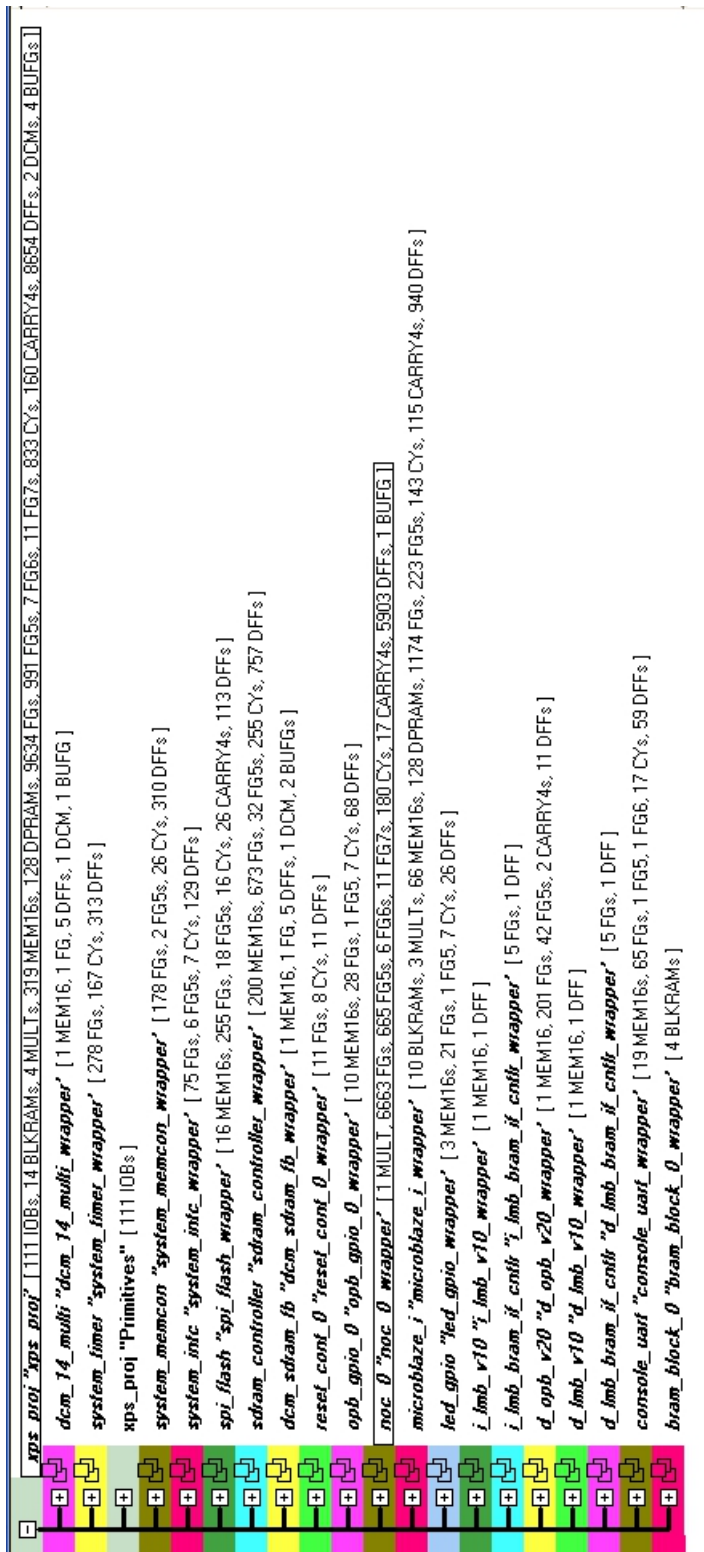


Figure A.4: Description of the floorplan with color codes, from floorplanner in Xilinx ISE.



# Appendix B

## Code

### B.1 VHDL code

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 library proc_common_v2_00_a;
7 use proc_common_v2_00_a.proc_common_pkg.all;
8 entity user_logic is
9     generic
10    (
11        -- ADD USER GENERICS BELOW THIS LINE -----
12        --USER generics added here
13        -- ADD USER GENERICS ABOVE THIS LINE -----
14
15        -- DO NOT EDIT BELOW THIS LINE -----
16        -- Bus protocol parameters, do not add to or delete
17        C_DWIDTH          : integer          := 32;
18        C_NUM_CE          : integer          := 20
19        -- DO NOT EDIT ABOVE THIS LINE -----
20    );
21 port
22    (
23        -- ADD USER PORTS BELOW THIS LINE -----
24        --USER ports added here
25        -- ADD USER PORTS ABOVE THIS LINE -----
26
27        -- DO NOT EDIT BELOW THIS LINE -----
28        -- Bus protocol ports, do not add to or delete
29        Bus2IP_Clk        : in  std_logic;
30        Bus2IP_Reset      : in  std_logic;
31        Bus2IP_Data       : in  std_logic_vector(0 to C_DWIDTH-1);
32        Bus2IP_BE         : in  std_logic_vector(0 to C_DWIDTH/8-1);
33        Bus2IP_RdCE       : in  std_logic_vector(0 to C_NUM_CE-1);
34        Bus2IP_WrCE       : in  std_logic_vector(0 to C_NUM_CE-1);
35        IP2Bus_Data       : out std_logic_vector(0 to C_DWIDTH-1);
36        IP2Bus_Ack        : out std_logic;
37        IP2Bus_Retry      : out std_logic;
38        IP2Bus_Error      : out std_logic;
39        IP2Bus_ToutSup    : out std_logic
40        -- DO NOT EDIT ABOVE THIS LINE -----
```



```

41 );
42
43 attribute SIGIS : string;
44 attribute SIGIS of Bus2IP_Clk      : signal is "CLK";
45 attribute SIGIS of Bus2IP_Reset   : signal is "RST";
46
47 end entity user_logic;
48
49 -----
50 -- Architecture section
51 -----
52
53 architecture IMP of user_logic is
54
55 --USER signal declarations added here, as needed for user logic
56 signal clk, reset                : std_logic;
57 signal data_in                   : std_logic_vector(63 downto 0);
58 signal send_packet               : std_logic;
59 signal mux_select                : std_logic_vector(3 downto 0);
60 signal BRAM_enable               : std_logic;
61 signal BRAM_addr                 : std_logic_vector(8 downto 0);
62 signal readout_finished          : std_logic;
63 signal CLK_GATE_FREQ             : std_logic_vector(3 downto 0);
64 signal start_testing             : std_logic;
65 signal data_out                  : std_logic_vector(63 downto 0);
66 signal packet_received_33        : std_logic;
67 signal packet_sent               : std_logic;
68 signal mux_o_packetcnt           : std_logic_vector(15 downto 0);
69 signal mux_o_tstamp              : std_logic_vector(15 downto 0);
70 signal readout_SW                : std_logic;
71 signal received_time             : std_logic_vector (15 downto 0);
72
73 -- Signals for user logic slave model s/w accessible register example
74 -----
75 signal slv_reg0                  : std_logic_vector(C_DWIDTH-1 downto 0);
76 signal slv_reg1                  : std_logic_vector(C_DWIDTH-1 downto 0);
77 signal slv_reg2                  : std_logic_vector(C_DWIDTH-1 downto 0);
78 signal slv_reg3                  : std_logic_vector(C_DWIDTH-1 downto 0);
79 signal slv_reg4                  : std_logic_vector(C_DWIDTH-1 downto 0);
80 signal slv_reg5                  : std_logic_vector(C_DWIDTH-1 downto 0);
81 signal slv_reg6                  : std_logic_vector(C_DWIDTH-1 downto 0);
82 signal slv_reg7                  : std_logic_vector(C_DWIDTH-1 downto 0);
83 signal slv_reg8                  : std_logic_vector(C_DWIDTH-1 downto 0);
84 signal slv_reg9                  : std_logic_vector(C_DWIDTH-1 downto 0);
85 --Signals for further development currently not connected to the NOC
86 signal slv_reg10                 : std_logic_vector(C_DWIDTH-1 downto 0);
87 signal slv_reg11                 : std_logic_vector(C_DWIDTH-1 downto 0);
88 signal slv_reg12                 : std_logic_vector(C_DWIDTH-1 downto 0);
89 signal slv_reg13                 : std_logic_vector(C_DWIDTH-1 downto 0);
90 signal slv_reg14                 : std_logic_vector(C_DWIDTH-1 downto 0);
91 signal slv_reg15                 : std_logic_vector(C_DWIDTH-1 downto 0);
92 signal slv_reg16                 : std_logic_vector(C_DWIDTH-1 downto 0);
93 signal slv_reg17                 : std_logic_vector(C_DWIDTH-1 downto 0);
94 signal slv_reg18                 : std_logic_vector(C_DWIDTH-1 downto 0);
95 signal slv_reg19                 : std_logic_vector(C_DWIDTH-1 downto 0);
96 signal slv_reg_write_select      : std_logic_vector(0 to 19);
97 signal slv_reg_read_select       : std_logic_vector(0 to 19);
98 signal data_IP2BUS               : std_logic_vector(C_DWIDTH-1 downto 0);
99 signal data_BUS2IP              : std_logic_vector(C_DWIDTH-1 downto 0);
100 signal slv_read_ack              : std_logic;
101 signal slv_write_ack             : std_logic;
102 --reversing vector
103 function reverseVector (input: in std_logic_vector) return std_logic_vector is
104 variable output: std_logic_vector(input'reverse_range);
105 begin
106 for i in input'range loop
107     output((input'length-1)-i) := input(i);

```

```

108     end loop;
109     return output;
110 end;
111 begin
112
113 interface : entity noc_v1_00_a.interface (behavioral)
114 port map(
115     clk      => Bus2IP_Clk,
116     reset    => slv_reg4(3),
117     data_in  => data_in,
118     send_packet => slv_reg4(27),
119     mux_select => mux_select,
120     BRAM_enable => slv_reg4(7),
121     BRAM_addr => slv_reg5(31 downto 23),
122     readout_finished => slv_reg4(23),
123     CLK_GATE_FREQ => slv_reg4(11 downto 8),
124     start_testing => slv_reg4(6),
125     data_out  => data_out,
126     packet_received_33 => packet_received_33,
127     packet_sent => packet_sent,
128     mux_o_packetcnt => mux_o_packetcnt,
129     mux_o_tstamp => mux_o_tstamp,
130     readout_SW => readout_SW,
131     received_time => received_time
132 );
133
134
135     slv_reg_write_select <= Bus2IP_WrCE(0 to 19);
136     slv_reg_read_select  <= Bus2IP_RdCE(0 to 19);
137 -- Assume there will never be a bus error
138 IP2Bus_Error <= '0';
139
140 -- Assume a retry will not be needed
141 IP2Bus_Retry <= '0';
142
143 -- No timeout suspend required
144 IP2Bus_ToutSup <= '0';
145
146 -- Always Acknowledge all reads and writes
147 IP2Bus_Ack <= '0' when (Bus2IP_WrCE="0000000000000000" and Bus2IP_RdCE="0000000000000000"
148 ) else '1';
149 IP2Bus_Error <= '0';
150 IP2Bus_Retry <= '0';
151 IP2Bus_ToutSup <= '0';
152 -- Reverse the data direction to and from the bus
153 -- This means we can work with the more sensible (31 downto 0) MSB:31, LSB:0
154 data_BUS2IP <= reverseVector(Bus2IP_Data);
155 IP2Bus_Data <= reverseVector(data_IP2BUS);
156
157 --Write control process, all registers from software is connected here
158 WriteControl : process (Bus2IP_Clk) is
159 begin
160     if (Bus2IP_Clk'event and Bus2IP_Clk='1') then
161         if Bus2IP_Reset = '1' then
162             slv_reg0 <= (others => '0');
163             slv_reg1 <= (others => '0');
164             --slv_reg2 <= (others => '0');
165             --slv_reg3 <= (others => '0');
166             slv_reg4 <= (others => '0');
167             slv_reg5 <= (others => '0');
168             --slv_reg6 <= (others => '0');
169             --slv_reg7 <= (others => '0');
170             --slv_reg8 <= (others => '0');
171         else
172             case slv_reg_write_select is
173             when "10000000000000000000" =>
174                 slv_reg0<=data_BUS2IP;

```

```

174     when "01000000000000000000" =>
175         slv_reg1<=data_BUS2IP;
176         --when "00100000000000000000" =>
177         --    slv_reg2<=data_BUS2IP;
178         --when "00010000000000000000" =>
179         --    slv_reg3<=data_BUS2IP;
180         when "00001000000000000000" =>
181             slv_reg4<=data_BUS2IP;
182         when "00000100000000000000" =>
183             slv_reg5<=data_BUS2IP;
184         --when "00000010000000000000" =>
185         --    slv_reg6<=data_BUS2IP;
186         --    when "00000001000000000000" =>
187         --        slv_reg7<=data_BUS2IP;
188         --    when "00000000100000000000" =>
189         --        slv_reg8 <=data_BUS2IP;
190         when others =>
191             null;
192     end case;
193 end if;
194 end if;
195 end process WriteControl;
196
197
198 --Process to determine the write and read from registers to NOC
199 network_on_chip: process (Bus2IP_Clk) is
200 begin
201     if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
202         if (Bus2IP_Reset = '1') then
203             slv_reg2 <=(others=>'0');
204             slv_reg3 <=(others=>'0');
205             slv_reg6 <=(others=>'0');
206             slv_reg7 <=(others=>'0');
207             slv_reg8 <=(others=>'0');
208         else
209             mux_select<= slv_reg4(31 downto 28); -- getting information from the SW program
210             data_in <= slv_reg0 & slv_reg1;--Assigning SLV_REG_1 & 2 Indata register
211             slv_reg7 <= x"0000" & mux_o_packetcnt;-- register number 8 counting number of packages
212                 through the mux
213             slv_reg8 <=x"FFFFFFF";-- x"0000" & mux_o_tstamp; --register number 9
214             slv_reg9<= received_time; --actual reception time
215             --when packet received flag high read out from the output of router 33 to the registers
216             if packet_received_33 = '1' then
217                 slv_reg6(27) <= '1';--Signalling the SW that package has arrived
218                 slv_reg2 <= data_out(63 downto 32); --writing data out from 33 to register
219                 slv_reg3 <= data_out(31 downto 0);
220             else
221                 slv_reg6(27) <='0';
222                 slv_reg2 <=data_out(63 downto 32);
223                 slv_reg3<=data_out(31 downto 0);
224             end if;
225             if readout_SW = '1' then
226                 slv_reg6(31) <= '1';
227             end if;
228         end if;
229     end process network_on_chip;
230     -- implement slave model register read mux
231     ReadControl : process( slv_reg_read_select , slv_reg0 , slv_reg1 , slv_reg2 , slv_reg3 , slv_reg4 ,
232         slv_reg5 , slv_reg6 , slv_reg7 , slv_reg8 ,slv_reg9 , slv_reg10 , slv_reg11 , slv_reg12 ,
233         slv_reg13 , slv_reg14 , slv_reg15 , slv_reg16 ,slv_reg17 , slv_reg18 , slv_reg19 ) is
234     begin
235         case slv_reg_read_select is
236             when "10000000000000000000" => data_IP2BUS <= slv_reg0;
237             when "01000000000000000000" => data_IP2BUS <= slv_reg1;
238             when "00100000000000000000" => data_IP2BUS <= slv_reg2;

```

```

238     when "00010000000000000000" => data_IP2BUS <= slv_reg3;
239     when "00001000000000000000" => data_IP2BUS <= slv_reg4;
240     when "00000100000000000000" => data_IP2BUS <= slv_reg5;
241     when "00000010000000000000" => data_IP2BUS <= slv_reg6;
242     when "00000001000000000000" => data_IP2BUS <= slv_reg7;
243     when "00000000100000000000" => data_IP2BUS <= slv_reg8;
244     when "00000000010000000000" => data_IP2BUS <= slv_reg9;
245     when "00000000001000000000" => data_IP2BUS <= slv_reg10;
246     when "00000000000100000000" => data_IP2BUS <= slv_reg11;
247     when "00000000000010000000" => data_IP2BUS <= slv_reg12;
248     when "00000000000001000000" => data_IP2BUS <= slv_reg13;
249     when "00000000000000100000" => data_IP2BUS <= slv_reg14;
250     when "00000000000000010000" => data_IP2BUS <= slv_reg15;
251     when "00000000000000001000" => data_IP2BUS <= slv_reg16;
252     when "00000000000000000100" => data_IP2BUS <= slv_reg17;
253     when "000000000000000000010" => data_IP2BUS <= slv_reg18;
254     when "0000000000000000000001" => data_IP2BUS <= slv_reg19;
255     when others => data_IP2BUS <= (others => '0');
256     end case;
257
258     end process ReadControl;
259 end IMP;

```

Listing B.1: HW/SW interface module, user\_logic.vhd

```

1
2
3  --Trafficgenerator module v2
4  --By Magnus Namork
5  --Based on design by
6  --Andreas Hepso and Ivar Ersland
7  --Designed to be able to detect where to put different modules in the NOC
8
9  library ieee;
10 --library noc_v1_00_a;
11 use ieee.std_logic_1164.all;
12 use ieee.numeric_std.all;
13 use ieee.std_logic_unsigned.all;
14 use work.type_lib_noc.all; --alter to noc_v1_00_a.type_lib_noc.all if implemented/used in
15   Xilinx EDK
16 --use noc_v1_00_a.all;
17
18 entity TG is
19     generic (tg_number:integer:=0;
20             tstamp_vector:integer:=15);--default value is 0 will be overwritten by interface when
21             instantiation
22
23     port(
24         ungated_clk :in  std_logic; --Necessary to read out from the BRAM when the rest of the
25         logic has stopped
26         clk :in  std_logic; --global clk
27         reset :in  std_logic; --global reset to circuit
28         start_testing :in  std_logic; --initial signal to begin the testgenerator
29         x_in :in  std_logic_vector(7 downto 0); --package data
30         request :in  std_logic; --request from the router
31         grant :in  std_logic; --grant from the local router
32         SW_enable :in  std_logic; --Activating readout from BRAM module
33         SW_addr :in  std_logic_vector(8 downto 0);--witch address in BRAM module to be read
34         global_time :in  std_logic_vector(15 downto 0);--global clock used for timestamping the
35         packages
36         x_out :out  std_logic_vector(7 downto 0); --package out data
37         RTS :out  std_logic; --synchronizing signals with the router(request to send)
38         CTS :out  std_logic; --synchronizing (cleared to send)
39         tstamp_out :out  std_logic_vector(tstamp_vector downto 0)--timestamp of the packages
40     );
41
42 end entity;

```

```

38
39 architecture behavioral of TG is
40
41 -----
42 --TYPE DECLARATION--
43 -----
44 --Table structure: SenderID; Destination ID;counter; number of packages;;
45
46 --States for the statemachines
47 type t_RecvState is (RST_BRAM, IDLE, SYNC, RECEIVE, TABLE_CONFIG, TABLE_CONFIG2, TABLE_GET, REC_SEND,
48   DATA_MANIPULATION, DATA_MANIPULATION2); --RST= Reset signal for Bram module, REC_SENDING,
49   state for sending after receiving when not in configmode
50 type t_WatchState is (IDLE, WATCHING);
51 type t_SendState is (IDLE, SYNC, SENDING);
52
53 --Definition of different variables to ease of code maintenance
54 constant module_num: quartinteger:=16;--must be 4,8,16,32 etc....
55
56 -----
57 --SIGNAL DECLARATION--
58 -----
59 --Signal to access the array
60 signal table1 :instruct_table; --defined in type_lib_noc, dynamic table possible to reconfigure
61 signal table2:instruct_table_large;--static routing table
62 --Signals for storing last received package and out-package
63 signal packet_in :std_logic_vector(63 downto 0);
64 signal packet_out :std_logic_vector(63 downto 0);
65
66 --Counter signals for use with the statemachines
67 signal rcv_counter :INTEGER range -1 to 8;
68 signal send_counter :natural range 0 to 8; --counting and controlling number of packages
69   sent in send state machine
70
71 --Control signals for controlling the response when receiving a packet
72 signal enable_timer :std_logic;
73
74 --Signals for configuration of the traffic_generator
75 --signal config :std_logic;
76 signal on_off :std_logic; --sending on or off
77 signal send_to :std_logic_vector(3 downto 0); --sending address
78 signal delay :std_logic_vector(11 downto 0); --the delay of the packet production
79 signal enable_processing_time :std_logic; --functionality
80
81 --Signals for LFSR_8 entity
82 signal enable_LFSR_8 :std_logic;
83 signal random_8bit :std_logic_vector(7 downto 0);
84
85 --Signals for BRAM entity
86 signal enable_BRAM :std_logic;
87 signal wenable_BRAM :std_logic;
88 signal addr_BRAM :std_logic_vector(8 downto 0);--address of BRAM
89 signal data_i_BRAM :std_logic_vector(tstamp_vector downto 0);--data input to BRAM
90
91 --Signals for package control
92 signal number_packets :natural range 0 to 64;
93 signal send_package :natural range 0 to 10000;
94 signal send_counting :natural range 0 to 12;
95 signal tg_hex_number :std_logic_vector(3 downto 0);
96
97 -----
98 -- END OF DECLERATIONS --
99 -----
100
101 begin
102   --Mapping of Linear Feedback Shift Register
103   --LFSR_8: entity noc_v1_00_a.LFSR_8(behavioral)
104   -- port map(
105   --   clk => clk ,
106   --   reset => reset ,
107   --   enable => enable_LFSR_8,

```

```

102 -- c_out => random_8bit
103 -- );
104 --Mapping of Bram Module within the TrafficGenerator
105 BRAM: entity work.BRAM(behavioral)
106 port map(
107     ungated_clk => ungated_clk ,
108     clk => clk ,
109     enable => enable_BRAM,
110     SW_enable => SW_enable,
111     wenable => wenable_BRAM,
112     addr => addr_BRAM,
113     SW_addr => SW_addr,
114     data_i => data_i_BRAM,
115     data_o => tstamp_out
116 );
117
118 recv: process (clk , reset , on_off , send_to , global_time , packet_in , packet_out) is
119
120 variable recv_state , nrecv_state :t_RecvState;  --state for receiving data, one in case and
121         one in control
122 variable send_state , nsend_state :t_SendState;  --state for sending data, one in case and
123         one as control
124 variable watch_state , nwatch_state :t_WatchState;  --state for monitoring Throughput of the
125         testgenerator
126 variable next_operation :std_logic_vector(3 downto 0);
127 variable trigger : natural range 0 to 4096;--specification of variable in use###
128
129 variable config_table ,config_data : std_logic;--variable which if the package is a
130         configuration package
131
132 variable packet_counter :natural range 0 to 512;
133 variable package_type :std_logic_vector(3 downto 0);
134 variable cnt :natural range 0 to 31;
135 variable table_counter :natural range 0 to 4;
136 --Variables to determine where to send package based on table
137 variable package_data :std_logic_vector(15 downto 0);--16 bits of data to be configured
138 variable received_address :std_logic_vector(module_num/4-1 downto 0);
139 variable sending_address :std_logic_vector(module_num/4-1 downto 0);
140 variable table_address :std_logic_vector(module_num/4-1 downto 0); --address for the
141         traffic table
142 variable table_num :natural range 0 to 64; --number defining the table number in the
143         static table
144 variable change_number :natural range 0 to 64; -- number to be changed in the table
145 variable sending_num :natural range 0 to 15; --number to define next send to router
146 variable number_of_packages :natural range 0 to 64; --variable to define the number of
147         packages to be sent from the TG
148
149 begin
150
151     table2<=(
152         (0,5,0,1,1) ,
153         (1,4,0,2,1) ,
154         (2,7,0,3,1) ,
155         (3,11,0,4,1) ,
156         (4,6,0,5,1) ,
157         (5,7,0,6,1) ,
158         (6,11,0,7,1) ,
159         (7,15,0,8,1) ,
160         (8,9,11,13,1) ,
161         (9,8,12,14,1) ,
162         (10,11,9,13,1) ,
163         (11,12,8,14,1) ,
164         (12,13,9,11,1) ,
165         (13,14,8,12,1) ,
166         (14,9,13,11,1) ,
167         (15,8,14,12,1)
168     ); --cell_test_array;--static data table

```

```

162     if (rising_edge(clk))then
163 --reset state, global reset
164         if reset = '0' then
165             config_table := '0';
166             config_data := '0';
167             on_off <= '0';
168             send_to <= (others=>'0');
169             delay <= (others=>'0');
170             enable_processing_time <= '0';
171             nrecv_state := RST_BRAM;
172             recv_state := RST_BRAM;
173             send_state := IDLE;
174             nsend_state := IDLE;
175             watch_state := IDLE;
176             nwatch_state := IDLE;
177             package_data :=x"0000";
178             next_operation :=(others=>'0');
179             CTS <= '0';
180             RTS <= '0';
181             package_type :=x"0";
182             x_out <= (others=>'0');
183             recv_counter <= -1;
184             send_counter <= 0;
185             table_counter :=1;
186             trigger := 0;
187             enable_LFSR_8 <= '1';
188             enable_timer <= '0';
189             packet_in <= (others=>'0');
190             packet_out <= (others=>'0');
191             enable_BRAM <= '0';
192             wenable_BRAM <= '0';
193             data_i_BRAM <= (others => '0');
194             cnt := 0;
195             send_package <= 0;
196             send_counting <= 0;
197             tg_hex_number<=std_logic_vector(to_unsigned(tg_number, 4));
198             table1<=(
199                 (0,5,1,1),
200                 (1,4,2,1),
201                 (2,7,3,1),
202                 (3,6,4,1),
203                 (4,1,5,1),
204                 (5,0,6,1),
205                 (6,13,7,1),
206                 (7,2,8,1),
207                 (8,10,9,1),
208                 (9,11,10,1),
209                 (10,15,11,1),
210                 (11,14,12,1),
211                 (12,9,13,1),
212                 (13,8,14,1),
213                 (14,12,15,1),
214                 (15,13,16,1)); --cell_ring_array;--mpeg_array;
215
216
217             else
218
219
220 --ThroughputState--
221
222 --statemachine that controls the throughput of the TG
223 case watch_state is
224 when IDLE =>
225     if on_off = '1' and start_testing = '1' then
226         enable_LFSR_8 <= '0';
227         enable_timer <= '1';
228         if enable_processing_time = '1' then

```

```

229     if trigger = to_integer(unsigned(random_8bit)) then
230         send_package <= send_package + 1;
231         nwatch_state := WATCHING;
232         enable_LFSR_8 <= '1'; --enable LFSR_8 pseudo_random counting
233         enable_timer <= '0';
234     end if;
235 else
236     if trigger = to_integer(unsigned(delay)) then
237         send_package <= send_package + 1;
238         enable_timer <= '0';
239         nwatch_state := WATCHING;
240     end if;
241 end if;
242 end if;
243
244 when WATCHING =>
245     if send_counting = 10 then
246         nwatch_state := IDLE;
247         send_counting <= 0;
248     else
249         nwatch_state := WATCHING;
250         send_counting <= send_counting + 1;
251     end if;
252 end case;
253
254 -----
255 -- Input statemachine --
256 -----
257
258 case recv_state is
259 when RST_BRAM =>
260     enable_BRAM <= '1';
261     wenable_BRAM <= '1';
262     addr_BRAM <= std_logic_vector(to_signed(cnt, 9));
263     if cnt = 31 then
264         nrecv_state := IDLE;
265     else
266         nrecv_state := RST_BRAM;
267         cnt := cnt + 1;
268     end if;
269
270 when IDLE =>
271     if request = '1' then --handshaking between router and Trafficgenerator
272         nrecv_state := SYNC;
273         CTS <= '1';
274         recv_counter <= recv_counter + 1;--reception of packages initiated
275     else
276         nrecv_state := IDLE;
277         CTS <= '0';
278     end if;
279
280 when SYNC => --synchronizing state to adapt the handshaking
281     nrecv_state := RECEIVE;
282
283 when RECEIVE =>
284     CTS <= '1';
285
286 --finished receiving packages 8 flits in total
287 if recv_counter = 8 then
288     if(packet_counter = 512) then--counts if the number of packages is the same as the space
289         packet_counter := 0;
290     end if;
291     if(config_table= '1')then
292         nrecv_state :=TABLE_CONFIG;
293         enable_BRAM<='0';
294     elsif(config_data = '1') then
295         nrecv_state:=DATA_MANIPULATION;

```



```

296     enable_BRAM<='0';
297     else
298         addr_BRAM <= std_logic_vector(to_unsigned(packet_counter, addr_BRAM'LENGTH)); --Writing
           tstamp to BRAM with packet_counter
299         enable_BRAM <= '1';--enabling the BRAM module
300         wenable_BRAM <= '1';
301         data_i_BRAM <= std_logic_vector(to_unsigned((to_integer(UNSIGNED(global_time))-
           to_integer(UNSIGNED(packet_in(15 downto 0)))),data_i_BRAM'LENGTH));
302         packet_counter := packet_counter + 1;--one more package added to the BRAM
303         nrecv_state := TABLE_GET;--going to the default send state
304     end if;
305     CTS <= '0';
306     rcv_counter <= -1;
307     else
308         --Flit with operation definition
309         if rcv_counter = 7 then
310             CTS <= '0';
311             --detects if configuration bit set in first arriving package
312             elsif rcv_counter = 0 and x_in(3) = '1' then
313                 config_table := '1'; --setting configuration bit to active
314             elsif rcv_counter = 0 and x_in(2) = '1' then
315                 config_data := '1';
316             end if;
317             rcv_counter <= rcv_counter + 1;
318             packet_in <= packet_in(55 downto 0) & x_in;
319         end if;
320
321
322         --Configuration of internal table to alter sending of data
323         when TABLE_CONFIG=>
324             CTS<='0';
325             table1<=table1;
326             change_number:=to_integer(unsigned(packet_in(35 downto 32)));--taking in first number to be
           configured in the table
327             sending_address :=packet_in(39 downto 36);--sends back a package to the router telling it
           to configure
328             number_of_packages :=1; --to confirm the configuration has been done
329             package_data:= x"AAAA";
330
331             nrecv_state:=TABLE_CONFIG2;
332
333         when TABLE_CONFIG2=>
334
335             table1(change_number,0)<=to_integer(unsigned(packet_in(31 downto 28))); --configuring
           first column in table
336             table1(change_number,1)<=to_integer(unsigned(packet_in(27 downto 24)));
337             table1(change_number,2)<=to_integer(unsigned(packet_in(23 downto 20)));
338             table1(change_number,3)<=to_integer(unsigned(packet_in(19 downto 16)));
339
340             nrecv_state := REC_SEND;
341
342         --This is the state that collects data to create the packages based on incoming package
343
344         when DATA_MANIPULATION=>
345
346             if(packet_in(31 downto 28)=x"0") then
347                 sending_address:=x"F";
348                 package_data:= packet_in(31 downto 16);
349                 nrecv_state:=REC_SEND;
350                 number_of_packages:=1;
351             else
352
353                 sending_address :=std_logic_vector(to_unsigned(table2(tg_number,table_counter),
           sending_address'length));
354             nrecv_state :=DATA_MANIPULATION2;
355         end if;
356

```

```

357 when DATA_MANIPULATION2=>
358   if (table_counter=3)then
359     table_counter:=1;
360   else
361     table_counter:=table_counter+1;
362   end if;
363   --even numbered test generator
364   if tg_hex_number(0)= '0' then
365     package_data(7 downto 4):= packet_in(23 downto 20) -x"1";
366     package_data(15 downto 12):= packet_in(31 downto 28) -x"1";
367     package_data(3 downto 0) := packet_in(19 downto 16);
368   --odd numbered test generator
369   elsif tg_hex_number(0)= '1' then
370     package_data(3 downto 0):=packet_in(19 downto 16)+x"1";
371     package_data(15 downto 12):= packet_in(31 downto 28)-x"1";
372     package_data(7 downto 4) := packet_in(23 downto 20);
373   end if;
374   --sending_address :=std_logic_vector(to_unsigned(sending_num,sending_address'LENGTH));
375   package_type:=x"4";
376   number_of_packages:=1;
377   nrcv_state:=REC_SEND;
378
379   --acquiring sending information based on package type
380   when TABLE_GET=>
381     received_address :=packet_in(39 downto 36);--taking the address from the sending TG
382     table_num :=to_integer(unsigned(received_address));--converting to integer to access
383     table
384     sending_num :=table1(table_num,1);--getting the number from the table
385     number_of_packages :=table1(table_num,3);--getting information of the number of packages
386     to be sent
387     sending_address :=std_logic_vector(to_unsigned(sending_num,sending_address'LENGTH));--
388     setting the send to address
389     nrcv_state:=REC_SEND;
390   when REC_SEND=>
391     number_packets <=number_of_packages;
392     on_off <='1';--signal to determine the sending from the TG
393     enable_processing_time <=packet_in(56);--time enable signal
394     send_to <=sending_address;--setting where to send the packet for the send to
395     statemachine
396     delay <= packet_in(51 downto 40); --12 bits delay
397     trigger := 0;
398     nrcv_state := IDLE;
399   when OTHERS =>
400     nrcv_state := IDLE;
401   end case;
402
403   -- Output statemachine --
404
405   case send_state is
406   when IDLE =>
407     RTS <= '0';
408
409     if send_package /= 0 and number_packets>0 then--initiating with number of packages to be
410     sent
411       nsend_state := SYNC;
412       --protocol of the circuit defines the packet out
413       packet_out <= send_to & package_type & x"0000" & tg_hex_number & next_operation &
414       package_data & global_time; --creating the package to send
415     else
416       nsend_state := IDLE;
417     end if;
418
419   when SYNC =>
420     --wait for synchronization with the "send_to" router.
421     RTS <= '1';
422     if grant = '1' then

```

```

418     nsend_state := SENDING;
419     send_counter <= send_counter + 1;
420     packet_out <= packet_out(55 downto 0) & packet_out(63 downto 56);
421     end if;
422
423 when SENDING =>
424     --send the generated packet out on the network
425     if send_counter = 7 then
426         if grant = '1' then
427             nsend_state := IDLE;
428             send_counter <= 0;
429             RTS <= '0';
430             number_packets<=number_packets -1;
431         end if;
432     elsif send_counter = 6 then
433         send_counter <= send_counter + 1;
434         send_package <= send_package - 1;
435         packet_out <= packet_out(55 downto 0) & packet_out(63 downto 56);
436         --send _counter [0->6]
437     else
438         send_counter <= send_counter + 1;
439         packet_out <= packet_out(55 downto 0) & packet_out(63 downto 56);
440     end if;
441
442 when OTHERS =>
443     null;
444 end case;
445
446
447 --Driving the next state signal
448 rcv_state := nrcv_state;
449 send_state := nsend_state;
450 watch_state := nwatch_state;
451 --Driving the x_out signal
452 x_out <= packet_out(63 downto 56);
453
454 end if;
455 end if;
456 end process;
457
458 end architecture;

```

Listing B.2: New trafficgenerator with table

```

1  -----
2  -- Written by Andreas Heps--
3  --Modified by Magnus Namork--
4  -----
5
6  library ieee;
7  library noc_v1_00_a;
8  use ieee.std_logic_1164.all;
9  use noc_v1_00_a.all;
10
11 entity interface is
12     generic (tstamp_vector:integer:=15);
13     port( clk      : in  std_logic;
14         reset      : in  std_logic;
15         data_in    : in  std_logic_vector(63 downto 0); --Signal containing data from
16             slv_reg0 and slv_reg1
17         send_packet : in  std_logic; --Signal which activates sending of information
18             lying on data_in, accessed by slv_reg4(1)
19         mux_select  : in  std_logic_vector(3 downto 0); --select signal for the mux.
20         BRAM_enable : in  std_logic; --Enable signal for the BRAM module in the test
21             generator

```

```

19 | BRAM_addr      : in  std_logic_vector(8 downto 0); --Write address to the bram module in test
    | generator
20 | readout_finished : in  std_logic;          --From SW, stored all values
21 | CLK_GATE_FREQ  : in  std_logic_vector(3 downto 0); --Set from SW, decides the clock gate
    | frequency
22 | start_testing  : in  std_logic;          --Starts the TM_control module, and thus the data
    | collecting.
23 | data_out       : out std_logic_vector(63 downto 0); --Signal written to slv_reg2 and
    | slv_reg3.
24 | packet_received_33 : out std_logic;          --Signal which informs that a package have
    | arrived.
25 | packet_sent    : out std_logic;          --Signal which informs that a package has been sendt.
26 | mux_o_packetcnt : out std_logic_vector(15 downto 0); --(MUX output) - Packet counters
27 | mux_o_tstamp   : out std_logic_vector(tstamp_vector downto 0); --MUX output from timestamp
    | mux
28 | readout_SW     : out std_logic ;          --readout to SW, to know the TM_control has stopped.
29 | received_time  : out std_logic_vector(tstamp_vector downto 0)
30 | );
31 | end entity;
32 |
33 | architecture behavioral of interface is
34 |
35 | type SendState is (IDLE, HOLD, SEND);
36 | type RecvState is (IDLE, RECEIVE);
37 |
38 | --declaring signals for statemachines
39 | signal recv_counter :INTEGER range -1 to 8;
40 | signal send_counter :INTEGER range 0 to 8;
41 | signal packet_in,packet_out :std_logic_vector(63 downto 0);--packj
42 |
43 | --declaring signals for 44_interface_mesh
44 | signal grant,request :std_logic; --handshaking signals
45 | signal RTS,CTS :std_logic; --handshaking signals
46 | signal x_in,x_out :std_logic_vector(7 downto 0);--router-router bus signals
47 | signal readout :std_logic;
48 | signal clock_enable :std_logic;
49 | signal i_clk :std_logic;
50 |
51 | signal global_receive_time:std_logic_vector(tstamp_vector downto 0); --time to stamp packages
    | when receiving
52 | signal prev_send_packet :std_logic; --Flank detektor signal for send_packet
53 |
54 | --Component declaration of clock buffer
55 | component BUFGCE
56 | port (
57 | O : out STD_ULONGIC;
58 | CE : in STD_ULONGIC;
59 | I : in STD_ULONGIC);
60 | end component;
61 |
62 | begin
63 | --Instantiation of clock buffer
64 | CLK_gater: BUFGCE
65 | port map(
66 | I =>clk ,
67 | CE =>clock_enable ,
68 | O =>i_clk
69 | );
70 |
71 | --trafficmonitor control module
72 | TM_control: entity noc_v1_00_a.TM_control(behavioral)
73 | port map(
74 | clk => clk ,
75 | reset => reset ,
76 | CLK_GATE_FREQ => CLK_GATE_FREQ,
77 | readout_finished => readout_finished ,
78 | start_testing => start_testing ,

```

```

79 | clock_enable => clock_enable ,
80 | readout     => readout
81 | );
82 | --entire test system and Network on chip
83 | test_mesh: entity noc_v1_00_a.interface_mesh (behavioral)
84 |   generic map (number_of_routers =>16,
85 |     bus_width     =>8,
86 |     tstamp_vector =>15,
87 |     deactivated_tg =>8,
88 |     deactivated_tm =>0)
89 |   port map(
90 |     ungated_clk => clk ,
91 |     clk         => i_clk ,
92 |     reset       => reset ,
93 |     start_testing => start_testing ,
94 |     grant_l_33  => grant ,
95 |     x_in_l_33   => x_in ,
96 |     request_l_33 => request ,
97 |     mux_select  => mux_select ,
98 |     readout     => readout ,
99 |     readout_finished => readout_finished ,
100 |     BRAM_enable => BRAM_enable,
101 |     BRAM_addr  => BRAM_addr,
102 |     x_out_l_33 => x_out ,
103 |     RTS_l_33  => RTS,
104 |     CTS_l_33  => CTS,
105 |     mux_o_packetcnt => mux_o_packetcnt ,
106 |     mux_o_tstamp => mux_o_tstamp,
107 |     global_receive_time =>global_receive_time
108 |   );
109 |
110 | HW_PROC: process (clk, reset) is
111 |   variable recv_state, nrecv_state :RecvState;
112 |   variable send_state, nsend_state :SendState;
113 |
114 |   begin
115 |     if reset = '0' then
116 |
117 |       nrecv_state := IDLE;
118 |       recv_state  := IDLE;
119 |       nsend_state := IDLE;
120 |       send_state  := IDLE;
121 |
122 |       x_in <= (others=>'0');
123 |       packet_in <= (others=>'0');
124 |       packet_out <= (others=>'0');
125 |       data_out <= (others => '0');
126 |
127 |       request <= '0';
128 |       grant <= '0';
129 |       packet_received_33 <= '0';
130 |       packet_sent <= '0';
131 |       prev_send_packet <= '0';
132 |
133 |       send_counter <= 0;
134 |
135 |       recv_counter <= -1;
136 |
137 |       elsif clk 'event and clk='1' then
138 |
139 |         -- Network Communication --
140 |
141 |
142 |
143 |         -- Receiving a packet to router 33 --
144 |
145 |

```

```

146 case recv_state is
147   when IDLE =>
148     packet_received_33 <= '0';
149     if RTS = '1' then
150       nrecv_state := RECEIVE;
151       grant <= '1';
152     else
153       nrecv_state := IDLE;
154       grant <= '0';
155     end if;
156
157   when RECEIVE =>
158     if recv_counter = 8 then
159       nrecv_state := IDLE;
160       grant <= '0';
161       recv_counter <= -1;
162       data_out <= packet_out;
163       packet_received_33 <= '1';
164       received_time <= global_receive_time;
165     else
166       if recv_counter = 6 or recv_counter = 7 then
167         grant <= '0';
168       end if;
169       recv_counter <= recv_counter + 1;
170       packet_out <= packet_out(55 downto 0) & x_out;
171     end if;
172 end case;
173
174 -----
175 -- Sending a packet in on router 33 --
176 -----
177
178 case send_state is
179   when IDLE =>
180     packet_sent <= '0';
181     if send_packet = '1' and prev_send_packet = '0' then
182       packet_in <= data_in; --read package from data_in
183       nsend_state := HOLD;
184       request <= '1';
185     end if;
186
187   when HOLD =>
188     packet_sent <= '0';
189     request <= '1';
190     if CTS = '1' then
191       nsend_state := SEND;
192       send_counter <= send_counter + 1;
193       packet_in <= packet_in(55 downto 0) & packet_in(63 downto 56);
194     end if;
195
196   when SEND =>
197     if send_counter = 7 then
198       if CTS = '1' then
199         packet_in <= packet_in(55 downto 0) & packet_in(63 downto 56);
200         nsend_state := IDLE;
201         request <= '0';
202         packet_sent <= '1';
203         send_counter <= 0;
204       end if;
205     else
206       packet_in <= packet_in(55 downto 0) & packet_in(63 downto 56);
207       send_counter <= send_counter + 1;
208     end if;
209
210   when others =>
211     null;
212 end case;

```

```

213
214 --Controlling the next state logic and input output to network
215   recv_state := nrecv_state;
216   send_state := nsend_state;
217   x_in <= packet_in(63 downto 56);
218   prev_send_packet <= send_packet;
219   readout_SW <= readout;
220   end if;
221 end process;
222 end architecture;

```

Listing B.3: The interface module with generic values defined

```

1  library ieee;
2  library noc_v1_00_a;
3  use ieee.std_logic_1164.all;
4  use noc_v1_00_a.type_lib_noc.all;--defines different types and subtypes,when used for
   simulation in ie AHDL, change to work directory,
5
6  use noc_v1_00_a.all;--library work;
7
8  entity interface_mesh is
9    generic(number_of_routers: integer:=16;
10     bus_width:integer := 8;
11     tstamp_vector:integer:=15;
12     deactivated_tg:integer:=0;
13     deactivated_tm:integer:=0);
14    port(
15     ungated_clk   : in  std_logic;
16     clk           : in  std_logic;
17     reset        : in  std_logic;
18     start_testing : in  std_logic;
19     grant_l_33   : in  std_logic;
20     x_in_l_33    : in  std_logic_vector(7 downto 0);
21     request_l_33 : in  std_logic;
22     mux_select   : in  std_logic_vector(3 downto 0);
23     readout      : in  std_logic;
24     readout_finished : in  std_logic;
25     BRAM_enable  : in  std_logic;
26     BRAM_addr    : in  std_logic_vector(8 downto 0);
27     x_out_l_33   : out std_logic_vector(7 downto 0);
28     RTS_l_33    : out std_logic;
29     CTS_l_33    : out std_logic;
30     mux_o_packetcnt : out std_logic_vector(tstamp_vector downto 0);
31     mux_o_tstamp  : out std_logic_vector(tstamp_vector downto 0);
32     global_receive_time :out std_logic_vector(number_of_routers-1 downto 0)
33    );
34 end entity;
35
36 architecture behavioral of interface_mesh is
37 --List of signal matrixes used to define interface between modules-----
38 --The positions in the matrix for each router:
39 --00=0, 01=1,02=2,03=3, 10=4, 11=5,12=6,13=7....33=15 see instantiation of router mesh for
   details
40
41 --see type library for definition of all these signals
42 signal grant_l:grant;
43
44 signal x_in_l:x_in;
45
46 signal request_l:request;
47
48 signal x_out_l:x_out;
49
50 signal RTS_l:RTS;
51

```

```

52  signal CTS_1:CTS;
53
54  signal packet_trigger:packet_trigger;
55
56  signal packet_cnt:packet_count;
57
58  signal tstamp_out:tstamp_out;
59
60  signal global_time : std_logic_vector(number_of_routers-1 downto 0);--global time signal to be
    used as time stamp in the testgenerator
61  --for all: TG use entity work.TG(behavioral);
62  begin
63
64  global_receive_time<=global_time;
65
66  --Multiplexer--
67  -----
68
69  mux_packetcnt: entity noc_v1_00_a.mux(behavioral)
70  generic map(packet_count_vect=>16)
71  port map(
72  in_00 => packet_cnt(0),
73  in_01 => packet_cnt(1),
74  in_02 => packet_cnt(2),
75  in_03 => packet_cnt(3),
76  in_10 => packet_cnt(4),
77  in_11 => packet_cnt(5),
78  in_12 => packet_cnt(6),
79  in_13 => packet_cnt(7),
80  in_20 => packet_cnt(8),
81  in_21 => packet_cnt(9),
82  in_22 => packet_cnt(10),
83  in_23 => packet_cnt(11),
84  in_30 => packet_cnt(12),
85  in_31 => packet_cnt(13),
86  in_32 => packet_cnt(14),
87  in_33 => packet_cnt(15),
88  sel => mux_select,
89  x_out => mux_o_packetcnt
90  );
91
92  --mux_tstamps: entity noc_v1_00_a.mux_tstamp(behavioral)
93  -- port map(
94  -- in_00 => tstamp_out(0),
95  -- in_01 => tstamp_out(1),
96  -- in_02 => tstamp_out(2),
97  -- in_03 => tstamp_out(3),
98  -- in_10 => tstamp_out(4),
99  -- in_11 => tstamp_out(5),
100 -- in_12 => tstamp_out(6),
101 -- in_13 => tstamp_out(7),
102 -- in_20 => tstamp_out(8),
103 -- in_21 => tstamp_out(9),
104 -- in_22 => tstamp_out(10),
105 -- in_23 => tstamp_out(11),
106 -- in_30 => tstamp_out(12),
107 -- in_31 => tstamp_out(13),
108 -- in_32 => tstamp_out(14),
109 -- sel => mux_select,
110 -- x_out => mux_o_tstamp
111 -- );
112 -----
113 --GLOBAL COUNTER--
114 -----
115 global_counter: entity noc_v1_00_a.global_counter(behavioral) --work for simulation,
    noc_v1_00_a or library for synthesis
116 port map(

```



```

117     clk    => clk ,
118     reset  => reset ,
119     start_testing => start_testing ,
120     global_time => global_time
121 );
122
123 ---4x4 RUTERMESH---
124
125 mesh: entity noc_v1_00_a.noc_44_mesh(behavioral)
126 port map(
127     clk    => clk ,
128     reset  => reset ,
129     grant_l_00 => grant_l(0) ,
130     grant_l_01 => grant_l(1) ,
131     grant_l_02 => grant_l(2) ,
132     grant_l_03 => grant_l(3) ,
133     grant_l_10 => grant_l(4) ,
134     grant_l_11 => grant_l(5) ,
135     grant_l_12 => grant_l(6) ,
136     grant_l_13 => grant_l(7) ,
137     grant_l_20 => grant_l(8) ,
138     grant_l_21 => grant_l(9) ,
139     grant_l_22 => grant_l(10) ,
140     grant_l_23 => grant_l(11) ,
141     grant_l_30 => grant_l(12) ,
142     grant_l_31 => grant_l(13) ,
143     grant_l_32 => grant_l(14) ,
144     grant_l_33 => grant_l_33 ,
145     x_in_l_00 => x_in_l(0) ,
146     x_in_l_01 => x_in_l(1) ,
147     x_in_l_02 => x_in_l(2) ,
148     x_in_l_03 => x_in_l(3) ,
149     x_in_l_10 => x_in_l(4) ,
150     x_in_l_11 => x_in_l(5) ,
151     x_in_l_12 => x_in_l(6) ,
152     x_in_l_13 => x_in_l(7) ,
153     x_in_l_20 => x_in_l(8) ,
154     x_in_l_21 => x_in_l(9) ,
155     x_in_l_22 => x_in_l(10) ,
156     x_in_l_23 => x_in_l(11) ,
157     x_in_l_30 => x_in_l(12) ,
158     x_in_l_31 => x_in_l(13) ,
159     x_in_l_32 => x_in_l(14) ,
160     x_in_l_33 => x_in_l_33 ,
161     request_l_00 => request_l(0) ,
162     request_l_01 => request_l(1) ,
163     request_l_02 => request_l(2) ,
164     request_l_03 => request_l(3) ,
165     request_l_10 => request_l(4) ,
166     request_l_11 => request_l(5) ,
167     request_l_12 => request_l(6) ,
168     request_l_13 => request_l(7) ,
169     request_l_20 => request_l(8) ,
170     request_l_21 => request_l(9) ,
171     request_l_22 => request_l(10) ,
172     request_l_23 => request_l(11) ,
173     request_l_30 => request_l(12) ,
174     request_l_31 => request_l(13) ,
175     request_l_32 => request_l(14) ,
176     request_l_33 => request_l_33 ,
177     x_out_l_00 => x_out_l(0) ,
178     x_out_l_01 => x_out_l(1) ,
179     x_out_l_02 => x_out_l(2) ,
180     x_out_l_03 => x_out_l(3) ,
181     x_out_l_10 => x_out_l(4) ,
182     x_out_l_11 => x_out_l(5) ,
183     x_out_l_12 => x_out_l(6) ,

```

```

184 | x_out_l_13 => x_out_l(7),
185 | x_out_l_20 => x_out_l(8),
186 | x_out_l_21 => x_out_l(9),
187 | x_out_l_22 => x_out_l(10),
188 | x_out_l_23 => x_out_l(11),
189 | x_out_l_30 => x_out_l(12),
190 | x_out_l_31 => x_out_l(13),
191 | x_out_l_32 => x_out_l(14),
192 | x_out_l_33 => x_out_l_33,
193 | RTS_l_00 => RTS_l(0),
194 | RTS_l_01 => RTS_l(1),
195 | RTS_l_02 => RTS_l(2),
196 | RTS_l_03 => RTS_l(3),
197 | RTS_l_10 => RTS_l(4),
198 | RTS_l_11 => RTS_l(5),
199 | RTS_l_12 => RTS_l(6),
200 | RTS_l_13 => RTS_l(7),
201 | RTS_l_20 => RTS_l(8),
202 | RTS_l_21 => RTS_l(9),
203 | RTS_l_22 => RTS_l(10),
204 | RTS_l_23 => RTS_l(11),
205 | RTS_l_30 => RTS_l(12),
206 | RTS_l_31 => RTS_l(13),
207 | RTS_l_32 => RTS_l(14),
208 | RTS_l_33 => RTS_l_33,
209 | CTS_l_00 => CTS_l(0),
210 | CTS_l_01 => CTS_l(1),
211 | CTS_l_02 => CTS_l(2),
212 | CTS_l_03 => CTS_l(3),
213 | CTS_l_10 => CTS_l(4),
214 | CTS_l_11 => CTS_l(5),
215 | CTS_l_12 => CTS_l(6),
216 | CTS_l_13 => CTS_l(7),
217 | CTS_l_20 => CTS_l(8),
218 | CTS_l_21 => CTS_l(9),
219 | CTS_l_22 => CTS_l(10),
220 | CTS_l_23 => CTS_l(11),
221 | CTS_l_30 => CTS_l(12),
222 | CTS_l_31 => CTS_l(13),
223 | CTS_l_32 => CTS_l(14),
224 | CTS_l_33 => CTS_l_33,
225 | PacketTrigger_00=>packet_trigger(0),
226 | PacketTrigger_01=>packet_trigger(1),
227 | PacketTrigger_02=>packet_trigger(2),
228 | PacketTrigger_03=>packet_trigger(3),
229 | PacketTrigger_10=>packet_trigger(4),
230 | PacketTrigger_11=>packet_trigger(5),
231 | PacketTrigger_12=>packet_trigger(6),
232 | PacketTrigger_13=>packet_trigger(7),
233 | PacketTrigger_20=>packet_trigger(8),
234 | PacketTrigger_21=>packet_trigger(9),
235 | PacketTrigger_22=>packet_trigger(10),
236 | PacketTrigger_23=>packet_trigger(11),
237 | PacketTrigger_30=>packet_trigger(12),
238 | PacketTrigger_31=>packet_trigger(13),
239 | PacketTrigger_32=>packet_trigger(14),
240 | PacketTrigger_33=>packet_trigger(15)
241 | );
242 |
243 |
244 | ---Traffic Generators---
245 |
246 |
247 | TestGen: for i in deactivated_tg to number_of_routers-2 generate --use the forloop to define
248 |     how many TG to be added to the interface
TG: entity noc_v1_00_a.TG(behavioral)-- in synthesis: noc_v1_00_a.TG(behavioral) is used, in
    simulation eg AHDL work is better

```

```
249 generic map(tg_number=>i)  --specifies identification of each test generator
250 port map(
251   ungated_clk => ungated_clk ,
252   clk        => clk ,
253   reset      => reset ,
254   start_testing => start_testing ,
255   x_in       => x_out_l(i) ,
256   request    => RTS_l(i) ,
257   grant      => CTS_l(i) ,
258   SW_enable  => BRAM_enable ,
259   SW_addr    => BRAM_addr ,
260   global_time => global_time ,
261   x_out      => x_in_l(i) ,
262   RTS        => request_l(i) ,
263   CTS        => grant_l(i) ,
264   tstamp_out => tstamp_out(i)
265 );
266 end generate;
267 -----
268 --Traffic Monitors--
269 -----
270 NoC_TM: for n in deactivated_tm to number_of_routers-1 generate
271 TM: entity noc_v1_00_a.TM(behavioral)-- in synthesis: noc_v1_00_a.TG(behavioral) is used or
    correct library, in simulation eg AHDL work is better
272 generic map(tm_number=>n)  --ID of each traffic monitor
273 port map(
274   clk        => ungated_clk ,
275   reset      => reset ,
276   CTS        => packet_trigger(n) ,
277   readout_finished => readout_finished ,
278   packet_counter => packet_cnt(n)
279 );
280 end generate;
281 end architecture;
```

**Listing B.4:** The interface mesh module with generic modules

## B.2 C-code

---

```
1 /*
2  * send.c
3  *
4  * Created on: 1. mars 2011
5  * Author: Magnus Namork
6  */
7
8 #include <stdio.h>
9 #include <sys/types.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <sys/mman.h>
```

```
13 #include <unistd.h>
14 #include <time.h>
15 #include <string.h>
16 #include <stdlib.h>
17 #include <ctype.h>
18
19 // #include "noc_methods.h"
20 // definition of addressspace according to the assigned space in Xilinx EDK
21
22
23 #define C_BASEADDR 0x84000000
24 #define C_HIGHADDR 0x840001FF
25 #define CHECK_BIT(var,pos) ((var) & (1<<(pos))) // macro to check if a single bit is
    set
26 #define BUFFER_SIZE 50
27
28 volatile unsigned int *slave_register0 = (int *) (C_BASEADDR + 0x0); // data_in_33(63
    downto 32)
29 volatile unsigned int *slave_register1 = (int *) (C_BASEADDR + 0x4); // data_in_33(31
    downto 0)
30 volatile unsigned int *slave_register2 = (int *) (C_BASEADDR + 0x8); // data_out_33(63
    downto 32)
31 volatile unsigned int *slave_register3 = (int *) (C_BASEADDR + 0xc); // data_out_33(63
    downto 32)
32 volatile unsigned int *slave_register4 = (int *) (C_BASEADDR + 0x10); // configuration
    register
33 volatile unsigned int *slave_register5 = (int *) (C_BASEADDR + 0x14); // Bram address
    register
34 volatile unsigned int *slave_register6 = (int *) (C_BASEADDR + 0x18); //
35 volatile unsigned int *slave_register7 = (int *) (C_BASEADDR + 0x1c); // mux tstamp
    counter
36 volatile unsigned int *slave_register8 = (int *) (C_BASEADDR + 0x22); // mux timer
37
38
39
```

```
40
41
42
43 void reset_all(){ //Setting all registers to zero
44 *slave_register0= 0x00000000; //register 0 data_in(63 downto 32)
45 *slave_register1= 0x00000000;
46 *slave_register2= 0x00000000;
47 *slave_register3= 0x00000000;
48 *slave_register4= 0x00000000;
49 *slave_register5= 0x00000000;
50 *slave_register6= 0x00000000;
51 *slave_register7= 0x00000000;
52 *slave_register8= 0x00000000;
53
54 }
55 void print_registers(){//printing all registers until user pushes 0
56 while(getchar()!='0'){
57     printf("slave register 0= %x\n",*slave_register0);
58     printf("slave register 1= %x\n",*slave_register1);
59     printf("slave register 2= %x\n",*slave_register2);
60     printf("slave register 3= %x\n",*slave_register3);
61     printf("slave register 4= %x\n",*slave_register4);
62     printf("slave register 5= %x\n",*slave_register5);
63     printf("slave register 6= %x\n",*slave_register6);
64     printf("slave register 7= %x\n",*slave_register7);
65     printf("slave register 8= %x\n",*slave_register8);
66     printf("press 0 to exit\n");
67 }
68 }
69
70 int run_noc(int large,int small){//method that creates a load for the average value
    calculation
71
72 int lg,sm,diff;
73 if(large>15 || small>large || small<0){
```

```
74     printf("integers to large");
75     exit(1);
76 }
77 int result= 0x0000;
78 int difference = large - small;
79 lg= (large << 4);
80 sm= (small<<0);
81 diff= (difference<<12);
82 result = (result | lg | sm |diff);
83 return result;
84
85
86
87
88
89 }
90 int filewrite(int *latency, const int length){//writes the values from the circuit to the
    latency.txt file
91     int k=0;
92     FILE *fp;
93     //int f=latency;
94
95     if((fp=fopen("/var/suzaku_shared/latency.txt", "w"))==NULL) {//which text file is
        opened w defines write operation
96         printf("Cannot open file.\n");
97         exit(1);
98     }
99     for(;k<length;k++){
100         fprintf(fp,"%d\n", latency[k]);
101     }
102     fclose(fp);
103
104     return 0;
105 }
106 void multi_package(){
```

```
107 int quit=1;
108 int tg;
109 int tab_pos, tab_val, alter_data,type,package_type, test_generator;
110 volatile unsigned int input_1= 0x00000000;
111 volatile unsigned int input_2= 0x00000000;
112 *slave_register4= 0x00000000;
113 char respons;
114 while(quit!=0){
115     *slave_register4= 0xE0001C8;
116     printf("Press a positive number to continue sending packages, press 0 to exit\n");
117     scanf("%x",&quit);
118     input_1 = (input_1 & 0x00FFFF00); //making input ready for packet configuration
119
120     printf("\n\nWhich test generator do you want to send a package to? (0 to E)\n");
121     scanf("%x",&test_generator);
122     tg = (test_generator << 28); //setting bits from 28 + 4 to the test_generator value
123     printf("Setting send to address to %x\n", test_generator);
124     input_1 = (input_1 | tg); //OR-ing the mask with the control to set the correct bits
        high.
125
126     printf("\n\nWhat type of package do you want to send: (0,4 or 8 )\n");
127     scanf("%x",&package_type); //getting the new package type from the user.
128     type = (package_type << 24); //setting the package type on the correct location
129     input_1 = (input_1 | type);
130     if(package_type == 8){
131         printf("define table position\n");
132         scanf("%x", &tab_pos);
133         input_1 = (input_1 | tab_pos);
134         input_1 = (input_1 | (0xF << 4));
135         input_2 = (input_2 & 0x0000FFFF);
136         printf("define table values\n");
137         scanf("%x", &tab_val); //which table value is to be reconfigured?
138         alter_data = (tab_val << 16); //defining the table reconfiguration of the test generator
139         input_2 = (input_2 | alter_data);
140     }
```

```
141 *slave_register0=input_1;
142 *slave_register1=input_2;
143 printf("\n\nToggeling send\n");
144 *slave_register4 = (*slave_register4 ^ 0x08000000);
145 //Masking and toggling the send bit of the control register.
146
147 printf("Do you want to monitor traffic? y/n \n\n");
148 scanf("%s",&respons);
149
150 if(respons=='y'){
151     print_registers();
152 }
153
154
155 }
156
157
158
159 }
160
161 void test_generator_config(){
162     int tg,type,router,s,l = 0;//different masking values to be added to the input of the
        network on chip
163     int test_generator,package_type,data_load,table_position,router_value,i,m;
164     int average_val_1,average_val_2;
165     volatile unsigned int input_1= 0x00000000;
166     volatile unsigned int input_2= 0x00000000;
167     volatile unsigned int latency;
168
169     printf("\n\nThis test tests functionality of the circuit with different user input\n");
170     input_1 = (input_1 & 0x00FFFF00); //making input ready for packet configuration
171     printf("\n\nWhich test generator do you want to send a package to? (0 to E)\n");
172     scanf("%x",&test_generator);
173
174     //Shifting the bits of the tg to the correct place
```



```
175 tg = (test_generator << 28); //setting bits from 28 + 4 to the test_generator value
176 printf("Setting send to address to %x\n", test_generator);
177 input_1 = (input_1 | tg); //OR-ing the mask with the control to set the correct bits
    high.
178
179 printf("\n\nWhat type of package do you want to send: (0,4 or 8 )\n");
180 scanf("%x",&package_type); //getting the new package type from the user.
181 type = (package_type << 24); //setting the package type on the correct location
182 printf("Setting type to: %x\n", package_type);
183
184 if(package_type== 0){ //simple routing
185     }
186 else if(package_type==4){ //defines that there is an averaging operation to be
    performed
187     printf("Preparing averaging value:\n");
188     input_2 = (input_2 & 0x0000FFFF);
189     //scanf("%X%X",&averages_val_1,&average_val_2);
190
191     data_load = (0xA0E4<<16); //defining the load fixed value
192     input_2 = (input_2 | data_load);
193 }
194 else if (package_type==8){ //defines a reconfiguring operation
195     printf("Preparing averaging value:\n");
196     input_1 = (input_1 | 0xA );
197     input_2 = (input_2 & 0x0000FFFF);
198     data_load = (0xBCDE<<16); //defining the table reconfiguration of the test generator
199     input_2 = (input_2 | data_load);
200
201 }
202
203 else{
204     printf("wrong value input\n");
205     exit(1);
206 }
207
```

```
208 input_1 = (input_1 | type); //adding type
209
210 s = (0xF << 4); //setting sending router, in this case interface module F
211 input_1 = (input_1 | s);
212
213 *slave_register4 = 0x00000000;
214 //Setting the control for the mux readout to zero to ensure that the mask works to it's
    intensions.
215
216 printf("package ready to be sent is %x%x\n", input_1, input_2);
217 printf("which router is to be read from?");
218 scanf("%x", &router_value);
219 router = (router_value << 28);
220
221 *slave_register4 = (*slave_register4 | router) ; //OR-ing the mask with the control to set
    the correct bits high.
222 *slave_register0 = input_1;
223 *slave_register1 = input_2;
224 *slave_register4 = *slave_register4 ^ 0x080001C8; //slv_reg_4 setting send packet high,
    mux is set to 33
225 *slave_register5 = 0x0F000000; //determining the BRAM address which is (31 downto
    23)
226
227 printf("Setting package value\n");
228
229
230 printf("Sending package\n");
231
232 printf("packet sent\n");
233
234 //checking if a package is received or not
235 if(*slave_register2 == 0x00000000){
236     printf("no packet received\n");
237 }
238 else{
```

```
239 latency = *slave_register3 & 0x0000FFFF; //anding to keep last 16 bits of latency or
      global time in circuit
240 printf("Number of packages sent through the specified router is %d\n",*slave_register7);
241 printf("Latency of the received package is %d clock cycles\n", latency);
242 //fwrite(latencies, sizeof(latencies)/sizeof(latencies[0])); //Write operation to text file
243 printf("The data_out is now %x%x\n", *slave_register2,*slave_register3);
244 printf("test finished\n");
245 }
246
247 }
248
249
250 void full_test(){ //sending initial package to the network, defines a standard testing
      procedure
251 int k,l,m,o = 0;
252 int tg_full,type_full, data_load_full;
253 int first_number,second_number,check_number;
254 int latency;
255 int latencies[100]={0};
256 volatile unsigned int input_1= 0x00000000; //default package first 32 bit
257 volatile unsigned int input_2= 0x00000000; //default package last 32 bit
258
259
260 for (k = 8 ; k<0xF;k++){ //the tg to send to
261 for (l=0xF; l>7;l--){ //value to calculate from
262 m=0xF-l; //second number of the load of the package
263 input_1= 0x00000000; //default package first 32 bit
264 input_2= 0x00000000; //default package last 32 bit
265 *slave_register4 = 0x00000000;
266
267 input_1 = (input_1 & 0x00FFFF00); //making input ready for packet configuration
268 tg_full = (k << 28); //setting bits from 28 + 4 to the test_generator value
269 input_1 = (input_1 | tg_full); //OR-ing the mask with the control to set the
      correct bits high.
270 type_full = (0x4 << 24); //setting the package type on the correct location
```

```

271     input_1 = (input_1 | type_full); //adding the type to the input
272     input_2 = (input_2 & 0x0000FFFF);
273     data_load_full = (run_noc(l,m)<<16); //defining the load
274     input_2 = (input_2 | data_load_full);
275
276     *slave_register0 = input_1; //input_1; //register 0 data_in(63 downto 32)
277     *slave_register1 = input_2; //input_2;
278     *slave_register5 = 0x0F000000; //determining the BRAM address which is (31
        downto 23)
279     *slave_register4 = *slave_register4 ^ 0xF8001C8; //slv_reg_4 setting send packet
        high, mux is set to 33
280
281     usleep(1); //wait to ensure readout ready
282     check_number=((*slave_register3 & 0xF0000000) >> 28); //shifting to take out only
        the number we want
283     first_number=((*slave_register3 & 0x00F00000) >> 20);
284     second_number=((*slave_register3 & 0x000F0000) >>16);
285     if(first_number!=second_number || check_number!=0){ //controlling that the output
        from the circuit is correct
286         printf("wrong value from output\n");
287         exit(1);
288     }
289     latency = *slave_register3 & 0x0000FFFF; //removing unnecessary information
290     latencies[o]=latency; //adding the latency to the latency table
291     o++;
292
293 }
294 }
295 fwrite(latencies, sizeof(latencies)/sizeof(latencies[0])); //Write operation to text file
296 printf("information written to file\n");
297
298 }
299 int main(int argc, char * argv[]){
300     printf("-----\n");
301     printf("-- Testing of the 4x4 NoC router mesh --\n");

```

```
302 printf("-----\n");
303 int i,j = 1;
304 reset_all();
305
306 while(j != 0){
307
308 printf("Remember that all leading zeros in output will not be displayed\n");
309 printf("Press 1 to run default full test\n");//defined in method
310 printf("Press 2 to run customized test\n");//defined in method
311 printf("Press 3 to read the mux output\n");//inline define
312 printf("Press 4 to read data_out\n");
313 printf("Press 5 to read the status register\n");
314 printf("Press 6 to display registervalues\n");
315 printf("Press 7 to display register \n");
316 printf("Press 8 to read the mux counter\n");
317 printf("Press 9 to read the mux timestamp\n");
318 printf("Press 10 to reset all registers\n");
319 printf("Press 11 to toggle reset\n");
320 printf("Press 12 to run multipackage test\n");
321 printf("Press 0 to terminate\n");
322 //reading input
323 scanf("%d",&j);
324 //press 1
325 if(j == 1){
326 full_test();
327 }
328 //press 2 runs customized test for the NOC
329 else if(j == 2){
330 test_generator_config();
331 }
332 //press 3 changes packet counter mux to a different value
333 else if(j == 3){
334 int k = 0;
335 printf("\n\nWhich packet counter do you want to read? (0 to E)\n");
336 scanf("%x",&i);
```

```
337 //Shifting the mask to the correct place
338 k = (i << 28);
339 printf("Router value is :%x\n",k);
340 //Setting the control for the mux readout to zero to ensure that the mask works to it's
    intensions.
341 *slave_register4 = (*slave_register4 & 0x0FFFFFFF);
342 //OR-ing the mask with the control to set the correct bits high.
343 *slave_register4 = (*slave_register4 | k);
344 printf("\n\nThe local packet counter of router %x has the value %x\n\n",i,*
    slave_register7);
345
346 }
347 //press 4 read data out
348 else if(j == 4){
349     printf("\n\ndata_out = %x%x\n\n",*slave_register2,*slave_register3);
350 }
351 //press 5 read status register
352 else if(j == 5){
353     printf("status = %x\n", *slave_register4);//reading out the register
354 }
355 //press 6 read out all registers
356 else if(j == 6){
357     print_registers();
358 }
359 else if(j == 7){
360     printf("status = %x\n", *slave_register4);//reading out the register
361 }
362 else if(j == 8){
363     printf("Mux counter = %x\n",*slave_register7);
364 }
365 else if(j == 9){
366     if (*slave_register8==0x00000000 || 0xFFFFFFFF){
367         printf("Mux timestamp not active\n");
368     }
369     printf("Mux timer = %x\n",*slave_register8);
```

```
370 }
371 else if(j == 10){
372     reset_all();
373
374 }
375 else if(j==11){
376     printf("\n\nToggeling reset (active low)\n");
377     *slave_register4 = (*slave_register4 ^ 0x00000008);//slv_reg_4
378     //Masking and toggling 1 bit of control register.
379     printf("Control register is now: %x\n\n", *slave_register4);
380
381 }
382 else if (j== 12){
383     multi_package();
384 }
385 else {
386     printf("incorrect value on input");
387     exit(1);
388 }
389 }
390 printf("Testing terminated\n");
391 return 0;
392 }
```

---

**Listing B.5:** C program for on chip test of the Network on chip

# Appendix C

## AHEAD Network on Chip-Initial words

*This tutorial is just a short summary of some of the necessary tools and abilities connected to the AHEAD project and in particular the Network on Chip part of it. It is based on the tutorial written in [27]. There are some initial literature which is recommended. The most essential ones are listed here and should be read before working on the project. This tutorial is only a short version of the ones found below, but should provide answers to what works and what is not possible to do. One final tip, if something works, try to stick with it then test every bits and piece of the additional elements you put in. This will avoid getting unexplainable errors.*

- Suzaku Software/Hardware manual
- TFE 4170 Enbrikkesystemer Laboratorie oppgave vår 2007
- Stian Reiersen Arnesens Master thesis-general info about the system
- Sverre Hamres Master thesis- tutorial of how to add an adder to the system
- Ivar Erslands Master thesis-assignment about the Network on Chip system
- Andreas Hepsøs Master thesis- initial development of the testenvironment



## C.1 Equipment list for this project

- ACER Aspire 5610 laptop(Ubuntu,ATDE3)
- Suzaku-S boards SZ130-U00(FPGA:Xilinx Spartan 3 XCSV1200) and SZ030-U00(FPGA: Xilinx Spartan 3 XCSV1200)
- ATDE3 with VmWare player for building FPGA project files
- Ubuntu 10.10 32bit for connection with the FPGA
- Windows XP emulated in Windows 7 64 bit for simulation
- Dropbox 1.10 for file syncing between OSes(works in all systems even ATDE3)
- Linksys WRTG54 for connection between Suzaku board and Ubuntu-PC
- Xilinx ISE/EDK 10.1.3 for synthesis(Xilinx 11.\* could be used with the sz130 system)

# Appendix D

## Tutorial:How to implement the Network on Chip on the Suzaku-S platform

### D.1 Installing Xilinx in debian(Atmark Development environment(ATde3) or Ubuntu)

- These steps describe the details with Xilinx EDK because it is most problems with this installation. Both are necessary to run EDK
- Go to location of the file downloaded from <http://www.xilinx.com> and write `:Tar xvf edk_SFD.tar -C /folder/for/installation`
- Do the same thing for the corresponding ISE install file located in the same place as the EDK(ISE should be installed first)
- Change directory to the folder you extracted and move to `/EDK/bin/linux/` write `"sudo ./setup"`.(ISE is installed in the same way as EDK)
- Follow the directions
- EDK needs to know where ISE is installed, this is done by writing in the `/etc/profile` file in ATDE3

- In the bottom of the file:

```
XILINX="path/to/ISE"
```

```
export PATH
```

```
export XILINX
```

- Go back to /EDK/bin/lin and write ./xps
- This should start the EDK program

## D.2 VHDL code for the Suzaku Image(Peripheral or IP)

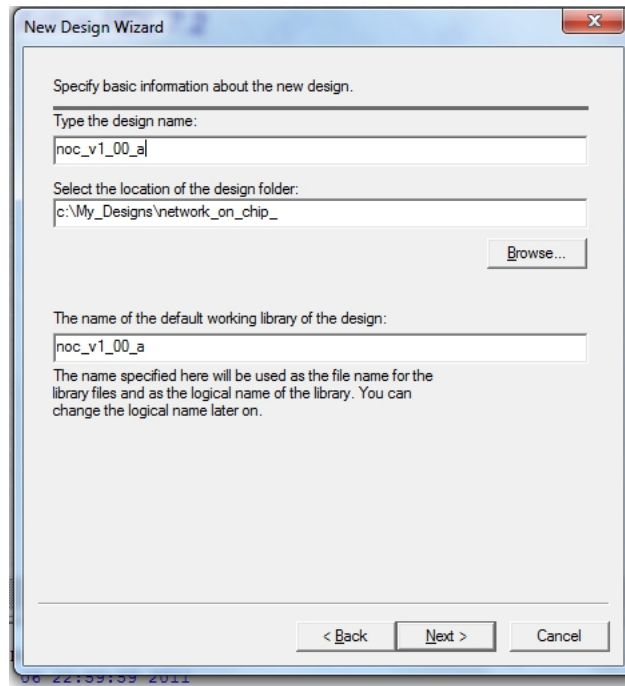
### D.2.1 Using AHDL for development

AHDL is an easy to use tool for simulation and to design the system. To be able to use AHDL together with Xilinx it is necessary to create a similar library in AHDL as the one existing in Xilinx EDK. This is done like in Figure D.1 Library creation AHDLfigure.caption.80 After this is done, add all the files from the xilinx project to the folder (typically../sz030-20090319/pcores/noc\_v1\_00\_a/hdl/vhdl) to the project and the system is set up for dual simulation and implementation.

### D.2.2 Setting up the project in Xilinx EDK

1. Start XILINX edk on the machine
2. In the introduction window choose "Open recent project" and go to xps\_proj file from the folder you will use(in folder sz030-200... which is downloaded from [http://download.atmark-techno.com/suzaku/fpga\\\_proj/10.1i/sz030/](http://download.atmark-techno.com/suzaku/fpga\_proj/10.1i/sz030/))

It is recommended to use the last version of the projectfiles since they come with an ISE file.



**Figure D.1:** How to create library similar to Xilinx library.

1. Under Hardware choose: "Create Import peripheral"
2. The name on the folder should be noc in the folder "name and version"
3. The OPB bus must be enabled , and later connected to the system in the system view
4. Choose to use 20 SW accessible registers.
5. remove User logic interrupt
6. step through the next steps
7. Select finish
8. If VHDL not included, select "IP Catalog"-> "USER"->(right)"noc"->"add IP"
9. Choose "Bus Connection" for the added noc\_0, then (right) "noc\_0" browse HDL sources. This is the folder where VHDL files must be added.

10. Change the pao file(in the data folder under pcores/noc\_v1\_00\_a) so that it contains synthesis line for all the files(supplied in appendix zip file, and listed in Section D.6File list NoCsection.D.6)
11. Add(Exchange) all the VHDL files wanted/needed in the project, if an additional file is added then the PAO file will have to be edited also. This has to be in order of synthesis ( listing the files from lowest to highest in synthesis, Figure A.1Modules presented in hierarchy.figure.caption.76)
12. One should then choose 1K in address space found under the "Addresses" tab in the System Assembly view. The 20 registers needs 20x32 bits => 640. If number of registers is edited one needs to change the amount of space as well. Address space is recommended from 0x84000000 and upwards. When the system is configured in EDK it should look something like in Figure D.2System Assembly EDKfigure.caption.81.

This step for an adder module is also described in the mentioned tutorial by Sverre Hamre.

### D.2.3 Synthesis

Synthesis in EDK is performed by doing

- "Software"->"Generate Libraries and BSPs"
- "Software"->"Build all user applications"
- "Hardware"->"Generate Netlist"
- "Device Configuration"->"Update Bitstream"

### D.2.4 Exporting project from EDK to ISE

- Open a new project in ISE.
- Then add .xps in ISE by clicking ADD SOURCE.

Instance	Name	Base Address	High Address	Size	Bus Interface(s)	Bus Connection	Lock	ICache	DCache	IP Type	IP Version
d_lmb_bram_if_ctrlr	C_BASEADDR	0x02000000	0x00001fff	8K	SLMB	d_lmb_v10	<input checked="" type="checkbox"/>			lmb_bram_if_ctrlr	1.00.b
i_lmb_bram_if_ctrlr	C_BASEADDR	0x00000000	0x00001fff	8K	SLMB	i_lmb_v10	<input checked="" type="checkbox"/>			lmb_bram_if_ctrlr	1.00.b
noc_0	C_BASEADDR	0x84000000	0x840003ff	1K	SOPB	d_opb_v20	<input checked="" type="checkbox"/>			noc	1.00.a
opb_gpio_0	C_BASEADDR	0xffffa000	0xffffa1ff	512	SOPB	d_opb_v20	<input checked="" type="checkbox"/>			opb_gpio	3.01.b
led_gpio	C_BASEADDR	0xffffa200	0xffffa3ff	512	SOPB	d_opb_v20	<input checked="" type="checkbox"/>			opb_gpio	3.01.b
system_intc	C_BASEADDR	0xffff3000	0xffff30ff	256	SOPB	d_opb_v20	<input checked="" type="checkbox"/>			opb_intc	1.00.c
spl_flash	C_BASEADDR	0xff000000	0xff0001ff	512	SOPB	d_opb_v20	<input checked="" type="checkbox"/>			opb_spi	1.00.e
system_timer	C_BASEADDR	0xffff1000	0xffff10ff	256	SOPB	d_opb_v20	<input checked="" type="checkbox"/>			opb_timer	1.00.b
console_uart	C_BASEADDR	0xff02000	0xff020fff	32M	SOPB	d_opb_v20	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		opb_uartlite	1.00.b
sdram_controller	C_MEM0_BASEADDR	0x80000000	0x81ffff	32M	SOPB	d_opb_v20	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		mch_opb_sdram	1.01.b
system_memcon	C_MEM0_BASEADDR	0xffe00000	0xffe0ffff	64K	SOPB	d_opb_v20	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		opb_emc	2.00.a

Figure D.2: System Assembly view in EDK, Spartan 3E XC3S1200.

- Then instantiate your XPS project in ISE by clicking View Instantiation Template.
- After this give UCF in your ISE (and removed that in EDK).
- Then compile your design.

Refer to the Xilinx support forum [2] for further details.

### D.2.5 Interfacing HW/SW

This part is found in the `user_logic.vhd`. It generally consists of the part with writing and reading. It defines how the slave registers are written from HW. More on this is found in the tutorial of Sverre Hamre, but as a comment it is mentionable that the commenting which is done is to determine which registers are writable and the ones who are only readable. It is possible to add registers here software accessible registers here without having to create a new peripheral. This is done by adding more `slv_reg` instances and changing the `C_NUM_CE` number in the `noc.vhd` file. The vectors in the write and read processes also has to be changed.

## D.3 Downloading the generated bit file to the suzaku board

### D.3.1 With serial interface

1. Ensure before the creation of the bit file one should set Applications->(right-click project:boot)->Set Compiler options-> Debug and Optimization->Optimization level->Size Optimized.
2. After performing Generate Libraries and BSP, build all user applications , generate netlist and update bitstream in Xilinx EDK
3. go to implementation in the project folder , typically `sz030-2008.../implementation`, copy the file named `download.bit`, paste it into the folder `bit2flash`, with the program `bit2flash`(see NOC package)

4. open the terminal and write `user@stasjon$ ./bit2flash download.bit implement.bit`, This creates the implementation file suitable for the flash memory within the suzaku board. The name of the file is possible to change.
5. open `gtkterm` or `minicom`
6. Adapt the baud rate to 115200 under configuration
7. Turn on and off the power of the suzaku board and close `gtkterm`
8. Then use `hermit` via the terminal by writing `"hermit download -i implement.bit -r fpga -force-locked"`( two lines(-) between `fpga` and `force-locked`)
9. Then something like serial: completed 0x00080000(524288) bytes in the terminal window should be seen
10. Then enter terminal and open `gtkterm`
11. Turn power on and of the suzakuboard
12. write `"b"` when it says `hermit>`
13. The system should now be running and you will have to log on to the system with `username:root` and `password:root`

### D.3.2 With Ethernet

- This method assumes that you are working with NFS. `flashw` or `netflash` has to be enabled in the `uClinux` image( second menu, under flash tools).
- Open a terminal in `ubuntu` with the `pc` and `Suzaku` connected to a router(i.e `Linksys WRTG54`)
- Write `telnet "ip address"` (i.e `"telnet 192.168.1.101"`)
- go to `var` folder and mount the shared folder in NFS.



- Instruction with netflash: "netflash -kbniCH -r /dev/flash/image image.bin" when reconfiguring fpga:netflash -kbniH -r /dev/flash/fpga fpga.bit (the b option in -bn determines whether or not to boot after the writing to flash, it could be excluded)
- If the card does not boot after reconfiguration, one must flash it with the lab computer in circuit lab 1. In those cases the reason has usually been a defect bit file.
- netflash help is located by writing netflash -h when on inside the fpga.
- it is also possible to use flashw Write# flashw -f filetotransfer.bit /dev/flash/fpga
- The board will perform the reflashing and reboot, unless b option chosen then just write "reboot" after startup

## D.4 Compiling and creating the uCLinux image

Follow the manual inside of the lab appendix for TFE4170 System-on-Chip, however one has to skip the part concerning networking support since this is needed for some applications with the Suzaku-S. When configuring the image itself, there are some things important to do: "TFE 4170 Enbrikkesystemer Laboratorieoppgave vår 2007" or Sverre Hamres tutorial explains some aspects about how this is done. To prepare the uCLinux image for the networking necessary to use ethernet, router and netflash follow these steps below:

- After selecting "Kernel/Library/Defaults Selection" choose "default all settings" then run make dep;make
- This will ensure a working image, if one wants to try to the implementation, this is now uploadable to the suzaku board
- If one wants Ethernet connection it is possible to add TCP/IP support under "Networking Options", IP Multicasting and IP kernel level autoconfiguration. Note: Not sure if this is necessary, but does not seem to do any harm to the creation of the system.

### D.4.1 Known errors and solutions

If make dep and/or make is not running properly (typically a lot of errors in final text)

Write: `echo $PATH`

check if elf-tools is included.

If not, write `export PATH=$PATH:/der/hvor/elftools er/bin`

Example: `export PATH=$PATH:/usr/local/microblaze-elf-tools/bin/`

then try `make dep;make` and `make image`

### D.4.2 NFS

NFS is a good tool to link a folder on your own computer and make it accessible on the Suzaku and it is described how this is done in

[http://no.wikipedia.org/wiki/Network\\_File\\_System](http://no.wikipedia.org/wiki/Network_File_System)

<http://ubuntuforums.org/showthread.php?t=249889>

[http://www.linuxconfig.org/HowTo\\_configure\\_NFS](http://www.linuxconfig.org/HowTo_configure_NFS) The commands needed to mount the folder when logged in to the Suzaku board (described below) is:  
`"mkdir /var/suzaku_shared" "mount -o nolock, rsize=4096, wsize=4096 -t nfs ip.to.computer:/home/folder/where/shared_folder"`

(i.e `mount -o nolock, rsize=4096, wsize=4096 -t nfs 192.168.1.100:/home/user/-suzaku_shared /var/suzaku_shared`)

If, when mounting the folder through NFS arrives permission denied, you can try to edit the `host.allow` and `host.deny` files as done in this tutorial. This is although the most thorough one with a large amount of information.

<http://nfs.sourceforge.net/nfs-howto/ar01s03.html>

additional, but more in detail info :<http://tldp.org/HOWTO/NFS-HOWTO/index.html>

### D.4.3 Setting static IP

This is described in the Software manual page 13 (version.1.3.1), but in brief: edit the `Uclinux/vendor/AtmarkTechno/SUZAKU-S.SZXXX/etc/rc/ifonfig` file and choose the preferred IP address

## C-Code

Step by step

- Compiled together with the uClinux image.Described in the tutorial written by Sverre Hamre
- Also possible to run directly from NFS, remember however to alter read-/write/execute permissions to the executable file.
- This is done by performing `chmod 777 "folderwithexecutablefile"`.
- Seems like the address range 0x84000000 is working better on the lab computer. Do not use the 0x81000000 which is specified in the tutorial because this is to close to other address areas and might cause problems.

## Java code

The java code is developed with Eclipse. Running the java program in a Linux system is performed with the command `"java -jar graphingprogram.jar"`

## D.5 Sources of error

In uClinux on the Spartan -S netflash as a reconfiguring method will not necessary work. This is most likely due to failure in the bit file. To solve this generate a new bit file after cleaning all generated files. This has been a problem with the emulated windows XP Xilinx tools but not with the tools in the ATDE3 OS.

Windows 7 og ISE10.1 is not compatible, one should rather use XP or atmark technos own system.

With Windows 7 it is easy to emulate XP, however this does not imply that the connection to HW will be possible. Ubuntu is also possible to use with Xilinx 10.1, but this is a bit more tricky and might cause more problems.

An installation key is needed to install the Xilinx tool, this is possible to get from either supervisor or Department engineer.

## D.6 File list NoC

(in compilation order for PAO file) updated pr 26.06.11:

- type\_lib\_noc vhdl
- arbiter\_v2 vhdl
- readout vhdl
- control2 vhdl
- 64bit\_buffer vhdl
- router vhdl
- mux\_tstamp vhdl
- mux16to1 vhdl
- 44\_mesh vhdl
- 44\_interface\_mesh vhdl
- global\_counter vhdl
- BRAM vhdl
- LFSR\_8 vhdl
- trafficgenerator\_v2 vhdl
- TrafficMonitor vhdl
- TM\_control vhdl
- interface vhdl
- user\_logic vhdl
- noc vhdl

Comment with regards to the files: When using create import peripheral in EDK a library called `noc_v1_00_a` is created. This is as shown for AHDL could cause some problems. If using Modelsim SE it will not handle the common "work" library where usually all the files in the same directory is found. ISE and AHDL should be capable of handling the use of only work as the referred library. The current code uses both, but following the specifications in this tutorial, it should work out of the box.

C code to be used on the board for testing:

- `helloworld.c` (just to check if c code is possible to run on the OS)
- `Send.c`
- `Makefile`
- This tutorial is more or less a result of different experiences when setting up tools to develop the NOC. Bare in mind that there might be differences between development systems and other factors so it is essential to understand limitations regarding development tools. If there are any more questions or anything not comprehend able please send an email with the question to [mnamork@gmail.com](mailto:mnamork@gmail.com).