

NoC support for dynamic FPGA pages

Eduard Warkentin

Bachelor of Computer Science, TU Darmstadt, 2007

A thesis submitted in fulfilment
of the requirements for the degree of
Master of Science

Darmstadt University of Technology,
Department of Computer Science

THE UNIVERSITY OF
NEW SOUTH WALES



SYDNEY • AUSTRALIA



March 2009

Copyright © 2009, Eduard Warkentin

Originality Statement

‘I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project’s design and conception or in style, presentation and linguistic expression is acknowledged.’

Signed

Acknowledgements

I would like to thank my supervisor, Dr. Oliver Diessel, for his great support in this project. His supervision was exceptionally good and was critical for achieving such results. From the first moment he supported me in any reasonable way. I would also like to thank my examiner Prof. Dr.-Ing. Sorin A. Huss, who gave me the chance to investigate an interesting area of research abroad. Further, he provided excellent support from afar. I would like to thank my family which supported me throughout my initiatives.

Abstract

Module-based FPGA reconfiguration offers virtualization and multitasking capabilities, but to support this, many problems need to be solved. Current methodologies are very specific and are not capable of scaling or of being reused for other applications. This thesis proposes a methodology whereby the use of dynamic reconfiguration is supported for every core independent of the communication interface and communication needs. The infrastructure introduces a general interface for attaching and detaching dynamically reconfigurable modules and a Network on Chip (NoC) to provide communication between modules and off-chip resources. The approach advocates the regular layout of modules on the device, which are connected with a network, helping to generalize the interface and share the communication lines. The NoC provides essential advantages such as scalability, increased parallelism and simplifying the process of partial dynamic reconfiguration. The interface between NoC and reconfigured modules requires a glue logic. Guidelines were composed for designing a simple, working and reusable Network Interface core (NWIF) for a general types of cores. This allows to use the regular layout with the NoC as a communication medium for multitasking or virtualization, where the modules may be easily replaced. The results of implemented examples using the guidelines are presented. The performance compared to the conventional design is worse in the NoC design and the overhead is significant, but the reusability, generalism of the interface and the scalability of the framework are improved for future applications. Several avenues for improving the prototype developed in this work are exposed.

Contents

List of Figures	x
List of Tables	xiv
1 Introduction	1
1.1 Thesis Contributions	5
1.2 Thesis Organization	6
2 Background and Related Work	8
2.1 Introduction	9
2.2 FPGA Pages - Distinct Clock Regions	10
2.2.1 The Virtex-4 FPGA Family	10
2.2.2 The Virtex-5 FPGA Family	11
2.2.3 The Virtex-6 FPGA Family	11
2.3 NoC - Previous Work	13
2.4 Support for Module-based Dynamic Reconfiguration	17
2.4.1 On-Demand Run-Time System by Ulmann	17
2.4.2 Erlangen Slot Machine (ESM) by Majer	18
2.4.3 Dynamic Network-on-Chip architecture (DyNoC) by Bobda	20
2.5 Conclusion	21

3	NoC Support for FPGA Pages	23
3.1	Requirements of the system	23
3.1.1	NoC	24
3.1.2	Page/Network Communication Interface	25
3.1.3	Inter Page Communication	27
3.1.4	Shared Access to Common Ports	28
3.2	Constraining the Degrees of Freedom	28
3.2.1	Clock and Clock Regions	29
3.2.2	IP-Cores	29
3.3	HERMES - NoC Infrastructure	29
3.3.1	Network on Chip	30
3.3.2	HERMES Router and Interface	30
3.3.3	HERMES Interface Adaptation for PDR	34
3.4	Network Interface cores - Translator between Cores and NoC .	37
3.4.1	Role of the Network Interface	37
3.4.2	Design guidelines for Network Interface	38
3.4.3	Packet Layout and different Formats	41
3.5	Summary	42
4	Custom Design for Dual AES	44
4.1	Introduction	44
4.2	AES	45
4.3	Custom Design without NoC and two AES Cores	46
4.3.1	Dual AES Interface	46
4.3.2	Dual AES Controller	49
4.3.3	Design Area and Timing Analysis	52
4.4	Host Control Application	53
5	NoC Version Design and Implementaion	55

5.1	General Design with NoC, Memory Controller and Two AES Cores	56
5.1.1	Splitting the Dual AES Interface into two Network Interfaces	56
5.1.2	Packet Formats	58
5.1.3	Operational Interface	60
5.1.4	Design of the AES Core NWIF	64
5.1.5	Design of the Memory Controller NWIF	68
5.1.6	Simulation	69
5.1.7	Design Area and Timing Analysis	71
5.2	Host Control Application	73
5.3	Conclusion	73
6	Benchmarks and Results	74
6.1	Comparison between Area and Timing Analysis for all Designs	74
6.2	Benchmark Results	76
6.2.1	Test Pattern	76
6.2.2	One AES	77
6.2.3	Two AES	78
6.2.4	NoC with three AES Cores	82
6.3	Discussion	83
6.3.1	Judgment of the Results	84
6.3.2	Differences between the designs	85
6.3.3	Potential improvements for the Designs	86
6.4	Conclusion	88
7	Conclusions and Future Work	89
7.1	Summary and Conclusions	89
7.2	Future Work	91

List of Figures

2.1	Pages in an XC4VLX15 device. It consists of 8 independent clock regions. The figure shows the bottom half of the device and indicates the smallest reconfiguration unit (row) as well distinct clock regions [25]	12
2.2	On-demand run-time system (Ullmann et al.)	18
2.3	The Erlangen Slot Machine (Majer et al.)	19
2.4	DyNoC (left: conceptual, right: implemented) by Bobda et al.	21
3.1	NoC mesh layout with intermediate Routers	27
3.2	HERMES router architecture. B indicates input buffers . . .	31
3.3	Physical interface between routers in HERMES framework . .	32
3.4	Interface between Artemis routers	35
3.5	GAPH macros: (a) Fixed to Reconfigurable area (F2R); (b) Reconfigurable to Fixed area (R2F) [20]	35
3.6	Interface between Router and page, supporting partial dynamic reconfigurable. Consists of Bus Macros from Figure 3.5 and control signals	36
3.7	General packet Layout, illustrating the layout in terms of OSI layers	41

4.1	AES core having three inputs and two outputs. Inputs are key and dataIn both 128 bits and start. The outputs are encData which is also 128 bits and ready. The AES core encrypts the data with a given key. The start signal is a control signal; the ready signal indicates when the data is finished with encryption	45
4.2	Design module implementing a custom design with memory controller, dual AES controller and two AES cores	48
4.3	Memory map of the communication model between the Host and both designs	49
4.4	Dual AES controller, it has one controll FSM and five buffer registers	50
4.5	FSM 1	51
4.6	FPGA floorplan for custom implementation for dual AES core design	52
5.1	Design of the NoC based implementation for multi AES core support. The dual AES core controller is split into two parts: the memory controller NWIF and an AES core NWIF. Custom connections are replaced by the HERMES NoC	57
5.2	Packet formats used in the current implementation. The first general packet Layout illustrates the arrangement and interpretation of each 8-bit flit	59
5.3	Sequance diagram representing an example of communication among the three pcomponents <i>Host</i> , <i>Memory Controller NWIF</i> and <i>AES Core NWIF</i> . In the example the Host writes n blocks (each 128 bits) into memory, encrypts $n - 1$ blocks and write the result back into the local memory and notifies the Host via an interrupt that it is ready	62
5.4	The AES NWIF module consists of 3 FSMs and intermediate buffers. FSM 1 receives packets, FSM2 sends packets and FSM 3 controls the AES core and estimates addresses for requesting and reading data	64

5.5	AES core NWIF FSM 1. Implements the packet receiving and disassembling routines	65
5.6	AES core NWIF FSM 3. Implements the control logic for the AES core and the next step including all relevant registers for the next packet	67
5.7	AES core NWIF FSM 2. Implements the packet assembling and sending routines	68
5.8	The memory NWIF module consists of 3 FSMs and intermediate buffers. FSM 1 receives packets, FSM2 sends packets and FSM 3 controls the memory controller	69
5.9	Timing diagram example for NoC design with two AES cores, where each AES core receives a job to encrypt 5 blocks. The rectangles marks different packets and the number in the rectangles defines the packet format according to Figure 5.2	70
5.10	FPGA floorplan of the NoC design with two AES cores	72
6.1	Benchmark for single AES core in custom and NoC Design with one and two hops	78
6.2	Benchmark for two AES core in custom and NoC designs. The core 00 and core 11 lies one hop away from the Mem Ctrl. with ID 01. The arrangement maybe obtained in Figure 6.3	79
6.3	Constellation where the <i>core 00</i> and <i>core 10</i> share, according to the XY routing algorithm, the output link. The Mem. Ctrl is the ID 01. The core 11 does not have any core attached . . .	80
6.4	Constellation where the <i>core 11</i> and <i>core 10</i> share the input link	81
6.5	Benchmark for two AES cores, where the <i>core 00</i> and <i>core 10</i> share, according to the XY routing algorithm, the output link. The Mem. Ctrl is the ID 01. The core 00 does not have any core attached	81
6.6	Benchmark for two AES cores where, the <i>core 11</i> and <i>core 10</i> shares the input link	82

6.7	Benchmark for three AES cores in the NoC design. The topology is illustrated in Figure 6.6	83
-----	--	----

List of Tables

1.1	Virtex-4 LX FPGA costs in US Dollars (source: Avnet February 2008)	4
2.1	State of the art in NoCs [22]	13
6.1	Area and timing analysis of <i>custom single AES</i> design, <i>custom dual AES</i> design and <i>NoC multi AES</i> . First column gives the names of modules, second column shows the area of each modules in slices and the third column shows the maximum speed for each module	75

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
PCI	Peripheral Component Interconnect
CLB	Configurable Logic Block
DES	Data Encryption Standard
DSP	Digital Signal Processing
DyNoC	Dynamic Network-on-Chip
ESM	Erlangen Slot Machine
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
ICAP	Internal Configuration Access Port
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IOB	Input-Output Block
LUT	Look-Up Table
NoC	Network-on-Chip
NP	Non-deterministic Polynomial time
RAM	Random Access Memory
RMB	Reconfigurable Multiple Bus
RPM	Relatively Placed Macro
VHDL	Very-high-speed integrated circuit Hardware Description Language

Chapter 1

Introduction

Field-Programmable Gate Arrays (FPGAs) are reprogrammable integrated circuits (ICs), comprising an array of logic blocks (cells) placed in an infrastructure of interconnections, which can be programmed at three distinct levels: (1) the function of the logic cells, (2) the interconnections between cells, and (3) the inputs and outputs. All three levels are configured via a string of bits, called a *bitstream*, that is loaded from an external source, either once or several times while the device is powered on. However, this potential power necessitates a suite of tools in order to design a system. Essentially, these tools generate the configuration bitstring, given such inputs as a logic diagram or a high-level functional description.

FPGA design tools start by synthesizing an application circuit specified in a design language such as VHDL [12] or Verilog [13] into a *netlist*, which is mapped to a particular FPGA, thereby creating a *circuit description*, and finally generating the configuration bitstream for this circuit description. Once the bitstream is loaded into the configuration memory of the FPGA, it be-

haves like the application circuit description that was synthesized at the start of the design flow. Thereafter, loading a different bitstream to change the circuit on the device is known as a *reconfiguration*.

FPGAs like the Xilinx Virtex-4 family [30] offers an opportunity to reconfigure the device in a partial and dynamic manner. After the bitstream is loaded and the FPGA is operating it is possible to reconfigure a part of the device, while the remaining parts are continue running. This reduces the amount of data that needs to be transfered as well the time needed for reconfiguration. In the literature ,the names for this procedure are not unique, the common variants are *runtime partial reconfiguration* or *partial dynamic reconfiguration (PDR)*.

The Virtex FPGA family supports two basic styles of partial reconfiguration: module-based and difference-based. Module-based partial reconfiguration uses modular design concepts to reconfigure large blocks of logic. The distinct portions of the design to be reconfigured are known as reconfigurable modules. Because specific properties and specific layout criteria, such as communication interface and 2D communication layout, must be met with respect to a reconfigurable module, any FPGA design intending to use partial reconfiguration must be planned and laid out with that in mind [29].

Partial dynamic reconfiguration opens up a variety of applications across many industries. It potentially increases system performance. Although a portion of the design is being reconfigured, the rest of the system can continue to operate. There is no loss of performance or functionality with unaffected portions of a design i.e., no down time. It also allows for multiple applications on a single FPGA [31].

Embedded systems often use FPGAs, because they offer the ability to provide alternative hardware components. If the system requires two or more hardware components and only one is active at the time, it may be possible to use a FPGA and reconfigure it as the situation demands. Some industries already uses this approach, e.g. in aeronautic, automotive, multimedia, industrial process control. But most designs targeted at the FPGA are static and do not change during operation.

The potential to partially reconfigure FPGAs at run time, introduces many benefits to the designer. The most relevant for this thesis is supporting so-called hardware virtualization, where a design is generally partitioned and swapped over time onto the FPGA. This allows the designer to fit a larger design onto a smaller FPGA [28]. Because of the high cost of the larger FPGAs it may be beneficial to partition the design and run it on a smaller FPGA in order to reduce part cost and power consumption. This is especially advantageous, since a larger device is almost twice the cost of the next smaller alternative, shown as the multiplicative factor in parentheses in Table 1.1, which lists the average prices of Virtex-4 devices from Avnet as of February 2008. Note that the price per logic cell also increases as the device size grows [16].

Since the size of modern FPGA scales gradually, an alternative use of partial dynamic reconfiguration in FPGA may be preferred, where by the high cost of resources (e.g. chip cost, power consumption) may be shared among many applications. This would enable multitasking systems and increase the overall utilization of a device. Multitasking on an FPGA-based processor is one possibility to explore the efficacy of reconfigurable comput-

Device	Logic Cells	Average Price	Price/Logic Cell
XC4VLX15	13,824	\$206.22	\$0.0149
XC4VLX25	23,192	\$420.63 ($\times 2.04$)	\$0.0181
XC4VLX40	41,472	\$608.13 ($\times 1.45$)	\$0.0147
XC4VLX60	59,904	\$928.75 ($\times 1.53$)	\$0.0155
XC4VLX80	80,640	\$1493.75 ($\times 1.61$)	\$0.0185
XC4VLX100	110,592	\$2816.25 ($\times 1.89$)	\$0.0255
XC4VLX160	152,064	\$4560.63 ($\times 1.62$)	\$0.0300
XC4VLX200	200,448	\$8320.00 ($\times 1.82$)	\$0.0415

Table 1.1: Virtex-4 LX FPGA costs in US Dollars (source: Avnet February 2008)

ing. Conventional computers and operating systems have demonstrated the many advantages of sharing computational hardware by several tasks over time. The ability to do run-time configuration allows the opportunities of multitasking to be investigated.

Design complexity, verification, and time-to-market pressures encourage reuse of components and designs that are tried and proven. Module-based design methodologies form a class of higher-level design methods that focus on implementing a design that is specified or described in terms of its constituent modules [6][14]. As such, dynamic reconfiguration at the module level is ideal for implementing hardware virtualization or multitasking [16].

Modular dynamic reconfiguration is currently not widely used in the industry, the reason could be the lack of practical methods and frameworks. If the vendor wants to design a partial reconfigurable system, it has to develop all the communication interface between static and reconfigurable part from scratch. This entails that the system designer must have intimate knowledge of the FPGA device architecture and partial reconfiguration, and how to best design an application for dynamic reconfiguration. If this step would

be generalized and developed into a well known procedure, maybe vendors would use it more often.

This thesis investigates a general approach for communication and a general communication interface between IP blocks. In doing so, it presents a specific Network on Chip (NoC) and proposes a top-down methodology for implementing the interfaces used by the *NoC*. This work proposes a framework in which the partial dynamic reconfiguration can be done without caring about the connections between modules. This framework is implemented on the the Virtex-4 FPGA family [30] and has some limitations, which makes it possible to focus on the investigation in a relevant scope.

1.1 Thesis Contributions

This thesis focus on difficulties using FPGAs in a multitasking manner. Concurrent usage of the input and output resources, while using an NoC infrastructure, is the main focus of this work. Another point of investigation is a general interface for communication between IP modules and the NoC, allowing partial dynamic reconfiguration of modules without errors in the NoC. The decision was made to use an already available NoC infrastructure named HERMES from the GAPH group [22] and to examine the sufficiency of this Infrastructure for the needs of the multitasking FPGA environment with highly concurrent inputs and outputs.

A general guide is presented for designing so-called glue logic or wrappers between the modules and the NoC interfaces. A top-down methodology is presented, in which a custom design is modified into an NoC design. An

extension is also made for general signals, where different signal approaches were examined and the implementation models into NoC infrastructure were worked out.

This thesis also proposes a test implementation of a custom design, which uses two AES cores for test purposes. The custom design uses conventional design methods and implements a single purpose system in which the speed is optimized. This custom system serves the role of a reference design against which the NoC infrastructure is tested. Further the thesis proposes an implementation of an NoC - based design with multiple AES core. The AES cores network interface is designed in a general manner, and this allows reuse of the implementation.

This work advocates the use of NoC infrastructure and proving the feasibility by proposing some experiments, where the performance of the custom design and NoC designs are evaluated.

This work discovers also some possible improvements, which are presented and are a target for future development.

1.2 Thesis Organization

The relevant background and state of the art is discussed in the Chapter 2. It gives an overview of the idea to use *pages* for implementing reconfigurable modules and possible advantages. Further it addresses the general discussion about benefits and drawback of an NoC for communication.

Chapter 3 presents the main methodologies and constraints in this work. It gives an overview of the general requirements for page-based communi-

cation and of the NoC infrastructure (HERMES). The methodology for for designing of network interfaces forms the main part of this Chapter.

The custom design for two AES cores is presented in details in the Chapter 4. The implementation of the interface between the memory, PCI bus and the host application is systematically illustrated.

Chapter 5 addresses the implementation design of the NoC with multiple AES cores, where the relevant NoC parameters are discussed, the particular example of the designing a network interface for an AES core ,memory controller and a PCI bus, is presented. Further, the packet formats that are proposed to be used in this particular implementation are outlined.

The Chapter 6 presents an evaluation of the effectiveness of NoC communications compared to a conventional custom design. Experimental benchmarks for assessing the performance of custom designs and NoC designs are introduced and the results of the experiments are presented and explained. Further the results are discussed and a preliminary evaluation is made. Based on the results, some suggestions for further improvement are presented.

The last Chapter 7 concludes this thesis with a summary of the work and its evaluation. Further directions for future work and improvement are proposed.

Chapter 2

Background and Related Work

This chapter presents the relevant background for the scope of the thesis: the support for pages in today's FPGAs is presented (Section 2.2), a general approach to page-based communication is described in Section 2.1, and an overview of the previous work is given.

This thesis is based on NoC communication, which has to replace the static wiring concept between modules of earlier work by Shannon Koh [15]. This chapter illustrates in the Section 2.3 the previous work done on NoC, and introduces aspects such as topology, flit size, buffering, router area, performance and implementation.

The last section presents an overview of previous work in support for module-based dynamic reconfiguration.

2.1 Introduction

To divide the FPGA into fixed pages and use the pages for virtualization of designs comes from the well known approach of virtual memory. But the concept of virtual memory is quite simple, it requires to store the data for some period of time. In the FPGA world this approach needs more capabilities, such as communication ability to other pages and to the off chip resources.

The next section analyzes the Xilinx FPGA families for feasibility for FPGA paging. One of the newest capabilities of the Virtex 4/5 families is the option to run particular parts of the FPGA with distinct clock speeds. This option allows the FPGA to be shared and run several applications with different timing, where by those can still communicate with each other. However, it is exactly this option that causes some new issues for the communication interface, since it connects different clock regions and still has to be general.

The paged FPGA approach allows the FPGA resources to be used for various applications running in parallel. Today, FPGAs are mostly used for single applications. Using pages, it becomes feasible not just to reconfigure the FPGA during the operation as the autonomous system demands, but to use the FPGA for multitasking purposes. FPGAs are playing the role of a multipurpose processor which is controlled by an operating system. The methodologies used by virtual memory approach are similar to the framework introduced by FPGA paging.

The communication between pages needs to describe a general interface in terms of supporting partial dynamic reconfiguration for swapping/programming a page. Some approaches to this communication problem are already presented by other research work, but the main problem with this is a lack of scalability and generality. In Section 2.4 some related research works are presented and discussed.

2.2 FPGA Pages - Distinct Clock Regions

Each Xilinx Virtex FPGA Family has specific characteristics, which need to be known to be able to design a page-based application framework. For example, the distinct clock regions and the smallest partial reconfiguration frame are different in each generation. This Section gives an overview about last three Virtex FPGA generations and their ability to support paging.

2.2.1 The Virtex-4 FPGA Family

Since the Virtex-4 FPGA Family [30] was introduced by Xilinx in 2004, the support for partial reconfiguration has changed and allows more flexibility for users and introduces the ability for page oriented reconfiguration.

Previous Virtex Families, such as Virtex-2, allowed reconfiguring a frame spanning the entire height of a FPGA device as a smallest unit. This makes use of pages not feasible. The Virtex-4 family introduces new architecture improvement specific to partial reconfiguration, where the smallest reconfiguration frame consists of 16 CLB in the height and the entire height consists of multiple of 16 CLB rows.

Independent clock regions were introduced first in Virtex-4 Family, this allows applications to run inside one or more clock regions with different timing constraints. So the smallest Virtex-4 FPGA has 8 such independent clock domains and the largest device has 24.

New options provided in Virtex-4 were the intention to introduce the idea of paging. So a page is an area on a chip with an independent clock region, where each row of 16 CLBs may be reconfigured dynamically. Figure 2.1 shows the bottom half of the layout in a smallest Virtex-4 FPGA, overall it has 8 distinct clock domains.

2.2.2 The Virtex-5 FPGA Family

In 2006 the Virtex-5 FPGA generation was introduced and brought some new options [32]. The architectural structure for partial reconfiguration remains the same and does not have any impact on the proposed paged-based reconfiguration scheme, except that the smallest reconfigurable unit consists of 20 CLB rows in this device family.

Important improvements to the Virtex-4 architecture included 6-input LUTs, diagonal routing and an increased maximum clock frequency of 550 MHz compared to the Virtex-4 at 500 MHz.

2.2.3 The Virtex-6 FPGA Family

In February 2009, Xilinx introduced the next generation of Virtex family - the Virtex-6 FPGA [33]. The changes made to the architectural structure

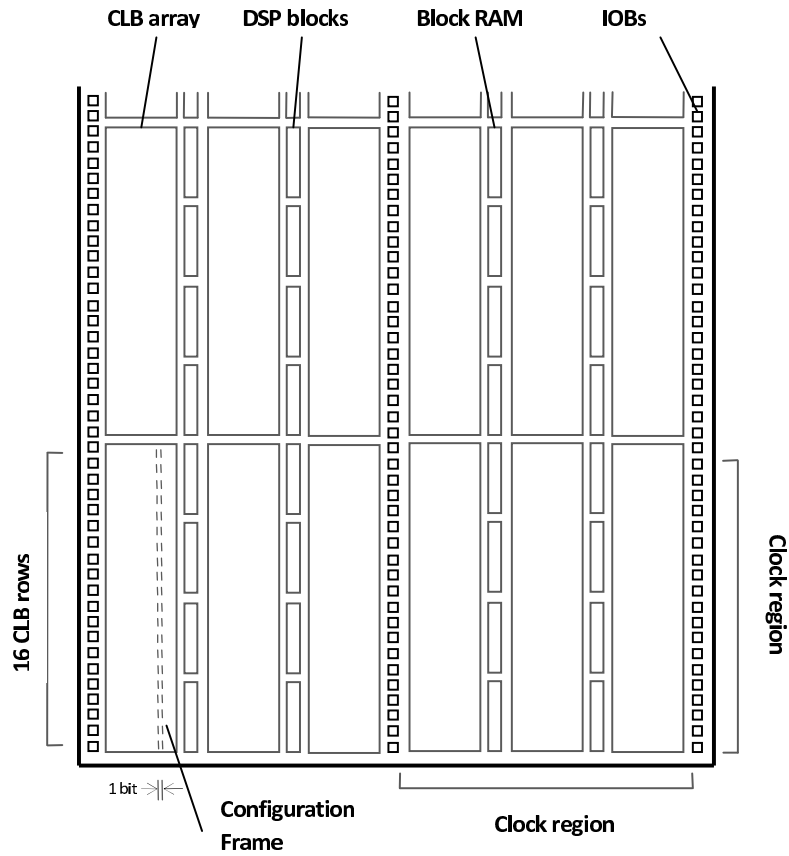


Figure 2.1: Pages in an XC4VLX15 device. It consists of 8 independent clock regions. The figure shows the bottom half of the device and indicates the smallest reconfiguration unit (row) as well distinct clock regions [25]

for partial reconfiguration is an increase to the page size, now it is doubled compared to Virtex-5 family and the smallest unit is a column of 40 CLBs.

Since the smallest unit consists of 40 CLB rows, the amount of the clock regions has slightly decreased compared to previous generations to 6 - 18. If before there were some concerns about the size of a page, now the page is quite large and it allows fitting larger cores into it.

2.3 NoC - Previous Work

NoC	Topology	Flit Size	Buffering	Router Area	Perfor.	QoS	Impl.
aSOC-2000 [17]	2D Mesh Scalable/ Determined by application	32 bits	None	50,000 transistors		Circuit switching(no wormhole)	ASIC layout CMOS 0.35m
Dally-2001 [8]	Folded 2D Torus 4x4/ XY Source	256 bits data + 38 bits control	Input queue	0.59 mm^2 CMOS 0.1m	4 Gbits/s per wire	Guarant. Throughput (virtual channels)	No
Marescaux-2002 [19]	2D Torus (scalable) / XY blocking, hop-based, deterministic	16 bits data + 3 bits control	Virtual output queue	446 slices Virtex-II(4.8% area overhead for XCV800)	320Mbits/s per virtual channel at 40 MHz	2 time multiplexed virtual channels	FPGA Virtex-II
Rijpkema-2002 [23] [24]	2D Mesh	32 bits	Input queue	0.26 mm^2 CMOS 0.12m	80Gbits/s per switch	Circuit-switching (guaranteed throughput)	ASIC layout
Hermes-2003 [22]	2D Mesh (scalable) / XY	8 bits data + 2 bits control (parameterizable)	Input queue (parameterizable)	631 LUTs 316 slices Virtex-II	500 Mbits/s per switch at 25 MHz	No	FPGA Virtex-II
MultiNoC-2005 [20]	2D Mesh (scalable) / XY	8 bits data + 2 bits control (parameterizable)	Input queue (parameterizable)	631 LUTs 316 slices Virtex-II	1 Gbit/s per switch at 50 MHz	No	FPGA Virtex-II
DyNoC-2005 [3]	2D Mesh / custom routing	32 bits data + 4 control	six 32-bit FIFO buffers	0.5% of Virtex-II 6000	Timing 391 MHz	No	Virtex-II 6000

Table 2.1: State of the art in NoCs [22]

This Section is intended to provide a big picture of the state of the art in Network on Chip (NoC) propositions. The results of the review are summarized in Table /reftab:NoCs. In this Table, each row corresponds to an NoC model that could be found in a literature. The NoC parameters considered relevant can be divided into three groups: (i) structural information, presented in the four first columns; (ii) performance information, in the fol-

lowing three columns; (iii) prototyping or silicon implementation details, in the last column.

Although the authors do not pose claims about the completeness of this review, they consider it rather comprehensive. Benini, De Micheli and Ye made important contributions to the NoC subject area in their conceptual papers [21] [2] [27]. However, none of these documents contains any NoC implementation details, except HERMES NoC [22], where MultiNoC is based on HERMES NoC

Each NoC defining parameter is described in detail below, together with an evaluation of the relative merits of each reviewed NoC proposition.

Almost all reviewed NoCs are based on packet switching and this is not stated in the table. The exception is the paper proposing the aSOC NoC [17], where the connection for each packet is fixed after the synthesis step. The first column in the Table 2.1 describes the topology of the NoC and the switching strategy. The most common topology used by many authors is a 2D mesh topology, it offers facilitated implementation using current IC planar technologies, simplicity of the XY switching strategy and network scalability. Another approach is to use the 2D torus topology, to reduce the network diameter [19]. The folded 2D torus presented by Dally [8] is an option to reduce the increased cost in wiring compared to a standard 2D torus. Concerning switching strategies, there is a clear lack of published information on specific algorithms. This indicates that further research is needed in this area. For instance, it is widely known that XY adaptive algorithms are prone to deadlock, but solutions exist to improve XY routing without causing deadlock risk.

The second important quantitative parameter of NoC switches is the flit size. From Table 2.1 it is possible to classify approaches into two groups, those focusing on future SoC technologies and those adapted to existing limitations. The first group includes the proposal of Dally [8], where switching channels are supposed to be 300-wire wide without significantly affecting the overall SoC area. However, this is clearly not feasible for today’s FPGAs. The second group comprises works with flit sizes ranging from 8 to 64 bits, a data width similar to current processor architectures. The works providing an NoC prototype, Marescaux [19] and Moeller [22], have the smallest flit sizes, 16 and 8 bits, respectively.

The next parameter in Table 2.1 is the switch buffering strategy. Most NoCs employ input queue buffers. Since input queuing implies a single queue per input, this leads to lower area overhead, justifying the choice. Another solution is to use virtual output queuing associated with time-multiplexed virtual channels, as proposed in [19].

Another important parameter is the queue size, which implies the need to solve the compromise among of the amount of network contention, packet latency and switch area overhead. A bigger queue leads to small network contention, higher packet latency, and bigger switches. Smaller queues lead to the opposite situation.

The fourth column collects results concerning the size of the switch. It is reasonable to expect that the adoption of NoCs by SoC designers be tied to gains in intra-chip communication performance. On the other hand, low area overhead when compared with e.g. standard bus architectures is another important issue. A SoC design specification will normally determine

a maximum area overhead allowed for intra-chip communication, as well as minimum expected communication performance, possibly on an IP by IP basis. Switch size, flit size (i.e. communication channel width) and switch port cardinality are fundamental values to allow estimating the area overhead and the expected peak performance for intra-chip communication.

Estimated peak performance, presented in the fifth column of Table 2.1, is a parameter that needs further analysis to provide a meaningful comparison among different NoCs. In this way, this column displays different units for different NoCs. This column must accordingly be considered as merely illustrative of possible performance values. Most of the estimates are derived from the product of three values: number of switch ports, flit size, and estimated operating frequency. The wide variation of values is due mostly to the last two values. No measured performance data could be found in any reviewed publication.

Next comes the quality of service (QoS) support parameter. The most commonly found form of guaranteeing QoS in NoCs is through circuit switching. This is a way of guaranteeing throughput and thus QoS for a given communication path. The disadvantage of the approach is that bandwidth may be wasted if the communication path is not used at every moment during the time the connection is established. In addition, since most approaches combine circuit switching with best effort techniques, this brings as consequence the increase of the switch area overhead. This is the case for NoC proposals presented in [8] and [23]. Virtual channels are one way to achieve QoS without compromising bandwidth, especially when combined with time division multiplexing (TDM) techniques. This last technique, exemplified

in [19] avoids that packets remain blocked for long periods, since flits from different inputs of a switch are transmitted according to a predefined time slot allocation associated with each switch output. It is expected that current and future SoC utilization will be dominated by streaming applications. Consequently, QoS support is regarded as a fundamental feature of NoCs by the authors. [22]

Finally, it is possible to state that NoC implementation results are still very scarce. None of the two ASIC implementations found in the literature gives hints to whether the design corresponds to working silicon. In addition, four NoCs have been reported to be prototyped in FPGAs, those proposed in [19],[22] and [3].

2.4 Support for Module-based Dynamic Reconfiguration

2.4.1 On-Demand Run-Time System by Ullmann

Ullmann et al. [26][10][11] proposed the *On-Demand Run-Time System* as shown in Figure 2.2. This system implements a custom bus system, and attached to it, four areas for dynamically reconfigured modules. It also implements bitstream decompression and self-reconfiguration via the ICAP.

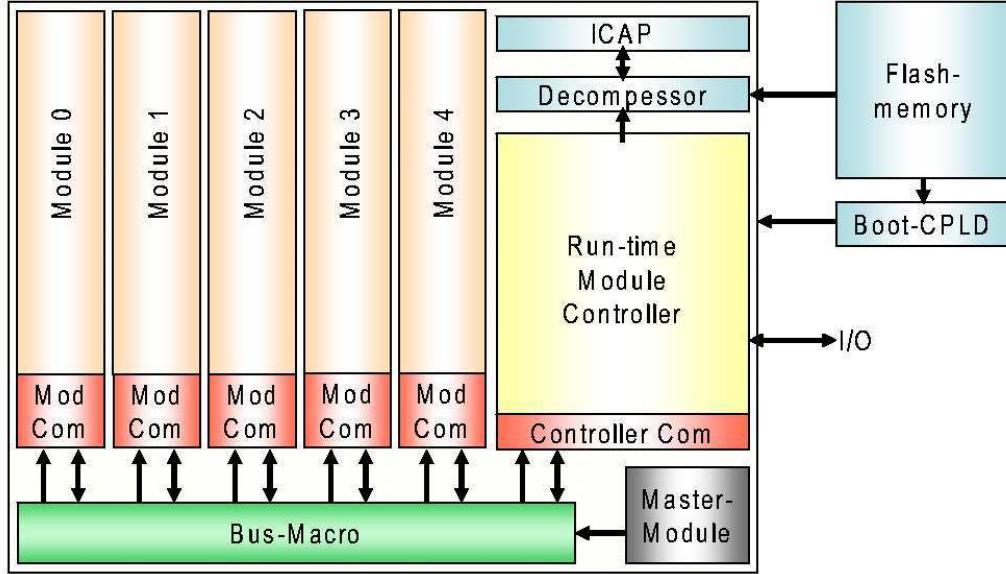


Figure 2.2: On-demand run-time system (Ullmann et al.)

The bus protocol overheads still exist. In addition, there can be long routing paths from the top of the modules to the bus interfaces at the bottom and then through the bus network (labeled as “Bus-Macro” in Figure 2.2). It is also unclear as to how one designs applications targeted to this system.

2.4.2 Erlangen Slot Machine (ESM) by Majer

One-dimensional, slot-based approach is the Erlangen Slot Machine (ESM) by Majer et al. [18]. The ESM is composed of two boards: a BabyBoard and a MotherBoard, as shown in 2.3. The MotherBoard is composed of a crossbar switch that links external I/O to the BabyBoard, and a PowerPC that runs software to control the application. The BabyBoard has a reconfigurable Virtex-II FPGA, SRAM and a reconfiguration manager responsible

for bitstream relocation and loading. Modules are loaded into fixed-sized slots M1-M3 on the Virtex-II FPGA.

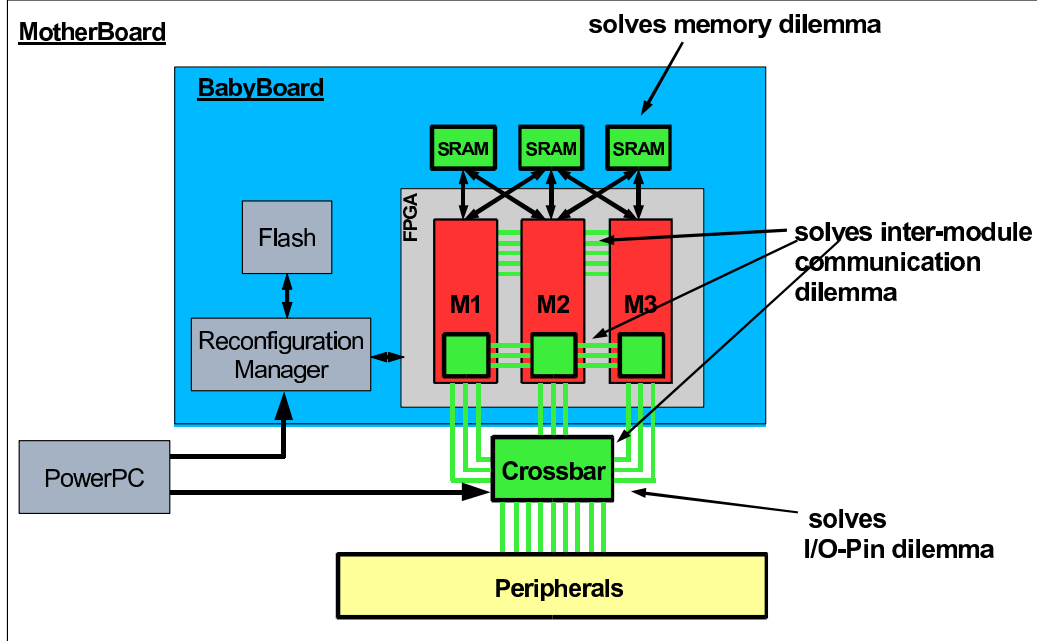


Figure 2.3: The Erlangen Slot Machine (Majer et al.)

There are four levels of inter-module communications provided on the ESM: shared SRAM gives access to two neighboring modules among each other, direct communication between adjacent modules by using bus macros, non-neighboring modules can communicate via a modified version [1] of a Reconfigurable Multiple Bus (RMB) [9] and the crossbar is used for off-chip communication.

This approach also suffers from area and timing overheads required to manage the system and its communications. The RMB requires crosspoint modules consisting of a controller, FIFOs and data network. External I/O is very expensive as it has to go off-chip into another FPGA, which in turn has to be routed through the crossbar and then off-chip again to the actual

peripherals or off-chip logic. The delay of the crossbar itself is 15 ns with an 18 clock-cycle setup time.

Programming such a system is an extremely complex task. The multiple levels of communication and application management add a high degree of complexity at the application design level.

2.4.3 Dynamic Network-on-Chip architecture (DyNoC) by Bobda

Two-dimensional approaches offer more flexibility in placement and thus the possibility of shorter intermodule wiring paths. The Dynamic Network-on-Chip architecture (DyNoC) proposed by Bobda et al. [3] [4] is a two-dimensional network-on-chip with routers laid out in a grid on an FPGA (see Figure 2.4). Rectangular modules of any size can be placed onto the network. Routers are disabled if they are temporarily obscured by a module placed over them, and are re-enabled when the module is removed.

It is a complex task to determine the appropriate set of shapes and placements of the modules in the application such that area and timing constraints are met. This is accentuated by the fact that it is very difficult to provide network bandwidth guarantees. The complete temporal communication patterns of the application must be known. Even then, doing so requires that multiple NP-hard problems be solved, as determined by Chan et al. [5] in addressing a similar problem in NoC topology generation and module shaping.

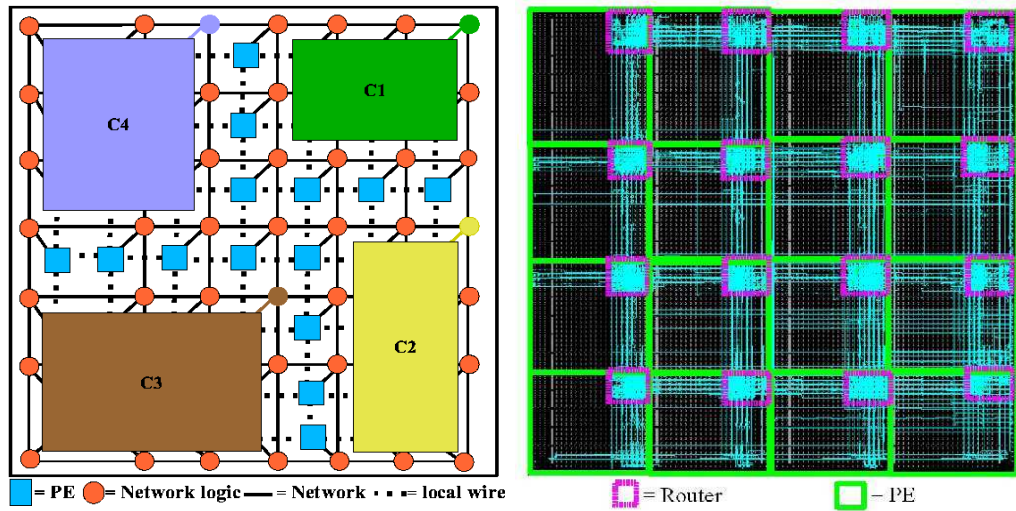


Figure 2.4: DyNoC (left: conceptual, right: implemented) by Bobda et al.

Finally, some area of the chip is underutilized because a ring of unobserved routers must surround every module in order to ensure routability.

Technically speaking, a two-dimensional layout such as this is counter-intuitive to the reconfiguration mechanism of the Virtex and Virtex-II devices. Since a configuration frame spans the entire height of the device reconfiguring a module that is wider than it is high will take longer than one of the same size that is rotated 90 degrees. This tends to indicate that the best way to lay out modules would be to pack them into the narrowest possible rectangles, thereby reverting to a one-dimensional layout.

2.5 Conclusion

This Chapter discussed the background found in the literature about NoC implementations and their support for partial dynamic reconfiguration. The approaches provide quite complex frameworks with impracticable limitations.

Further, the approaches provide an overview of how they are supposed to be implemented, but do not give application examples where they would be suitable.

The scope of paged design in newer FPGA families was also analyzed and the benefits stated. The main advantages offered by pages is virtualization and multitasking ability, but the proposed requires a general interface and suitable NoC support. After the research of previous work, it was concluded to use the HERMES [22] NoC framework because of its simplicity, low level of overhead, provided examples, and available simulation framework.

Chapter 3

NoC Support for FPGA Pages

The concept of "pages" supported by today's FPGAs is the main idea of this thesis and it is explored in this chapter. In Section 3.1 this chapter presents the requirements of the system for supporting pages. The NoC HERMES, configured as a suitable communication media, is presented in Section 3.3. The degree to which the design freedom was constrained is explained in Section 3.2. Furthermore, Section 3.4 presents the importance and design guidelines followed for the *Network Interface Core* (NWIF) in this work.

3.1 Requirements of the system

This Section gives an overview of the requirements imposed by the "paged" use of an FPGA. Paged use of an FPGA refers to capability of modern FPGAs to reprogram independent, rectangular areas. Further, each of such rectangles forms an independent clock domain (See Section 2.2.1).

One of the main requirements of a paged dynamic reconfiguration method is a generalized communication approach. This requirement is presented in Section 3.1.1. The appropriate requirements of the interfaces supporting pages is described in Section 3.1.2. The model of communication, required to be supported by interpage and off-chip communications, is presented in Sections 3.1.3 and 3.1.4.

3.1.1 NoC

Network on Chip (NoC) is a new approach to System on a chip (SoC) design. NoC-based systems can accommodate multiple asynchronous clocking as used in many of today's complex SoC designs. The NoC solution brings a networking methodology to on-chip communication and brings notable scalability and performance improvements over conventional bus systems.

The paged design introduced in Section 2 requires an appropriate communication framework to be able to exchange data between pages and off-chip resources. Previously a point-to-point communication scheme based on static wiring between modules was investigated and implemented [16]. The results were not sufficient in terms of scalability and simplifying the process of partial dynamic reconfiguration.

The need for a more general communication method and a general support for partial dynamic reconfiguration were the reasons why the NoC concept was chosen to implement the page-based communication. NoC provides enhanced scalability in comparison with previous communication architectures such as point-to-point and shared buses communications. Further, on one hand it allows the wires in the links of the NoC to be shared, which reduces

the demand for wiring resources, and on the other hand it is capable of operating simultaneously on different data packets on independent links, which achieves a high level of parallelism.

The NoC selected in this Thesis is static and so it does not need to be reprogrammed during run time, but it has to provide a general interface to each communication partner such as each page and off-chip resource, via an interface, as described in the next section.

3.1.2 Page/Network Communication Interface

Since the communication infrastructure is defined once at the outset, a general communication interface has to be defined, which is suited to every connection between network nodes and pages as well as off-chip communication port. We want the pages and off-chip communication ports to be partially reconfigurable FPGA areas. To decrease the complexity, the ports could be also predefined and configured once at the beginning.

To delimit the partial reconfiguration area from the network the interface between them has to use a reliable attach/detach mechanism. There are several approaches for realize this. One of the common methods is provided by Xilinx, but this approach is superseded. The *bus macros* from Xilinx are based on tri-state buffers, which are scarce resources in Xilinx FPGAs. Another approach, presented in Section 3.3.3, is proposed by the group GAPH from Catholic University of Rio Grande do Sul, Brasil [20].

The communication interface has to provide support for partial dynamic reconfiguration, where during a partial reconfiguration process the network

should not be affected by glitches in the interface between page and network. These glitches may introduce spurious data into the network, causing malfunctions or even circuit blocking. In addition, packets transmitted to an area undergoing reconfiguration, must be discarded, since it is typically impossible to know if these packets are targeted to the previous configuration in this area or to the next one.

Another requirement of the network and interfaces is the ability to scale with future devices and provide the same operational utilization. For this purpose the NoC is supposed to be fully parametrized and provide a high level simulation framework.

Figure 3.1 shows a lower half of the XC4VLX15 Virtex-4 device. The XC4VLX15 Virtex-4 device has overall 8 pages. The figure shows just the lower 4 pages, which are connected to each other using an NoC. The topology of the NoC is a mesh, chosen because of its simplicity and small implementation overhead [22]. The routing algorithm used here is XY Routing, according to which all routers have a unique, fixed address. An XY Routing algorithm requires a strict mesh topology to perform well and in a deterministic, deadlock and starvation free manner [20]. In our case it is required, that intermediate routers are located on the Y-axis to satisfy this requirement. The Figure 3.1 also shows the routers with address 30 and 31, which are connected to the IO blocks. To achieve low area overhead and high performance, the off-chip ports have to be chosen well.

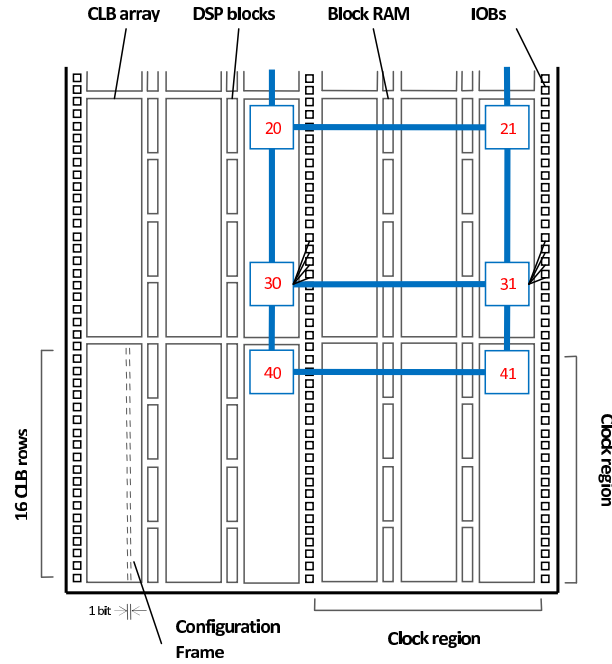


Figure 3.1: NoC mesh layout with intermediate Routers

3.1.3 Inter Page Communication

This section introduces some general communication scenarios, imposed by paged-based architecture, which need to be supported by the NoC and which need a deeper investigation to find optimal solution.

The first scenario is page to page communication in which one page communicates with another one and the NoC does not have any congestion on the link between two pages. This is a best case in which the communication is possible with the full bandwidth of the NoC.

If two or more pages want to communicate with one specific page over one link, then congestion occurs in the network, since the link has to be shared between many pages. At this point, the order in which access to the link is

provided is determined by an arbiter in a particular NoC router, where one page needs to wait (is blocked) and another can communicate. Since the link is shared and the bandwidth is fixed, the data of both pages needs more time to arrive at its destination. To prioritize one of the pages, the communication infrastructure can provide *QoS* and give more priority to a specific task.

If a task has a realtime constraint and relies on communication with another component it is very hard to predict the behaviour of the network, except when QoS is implemented and used. It is possible to allocate the page in a cluster with no link sharing and so it is also possible to predict the run time of the communication.

3.1.4 Shared Access to Common Ports

Another scenario is the communication between many pages and one off-chip resource. The resource is accessible through some dedicated I/O pins. This is similar to the scenario above, but the potential communication need is higher. To avoid congestion and blocking of some applications a solution could be increasing throughput to the off-chip resource or choosing a large buffer for critical ports. This decision can be made if the Interface of external resources is known. The goal is to maximize the utilization of the off-chip resources such as memory read and write operations

3.2 Constraining the Degrees of Freedom

To assess the main problems, as outlined in the previous sections some simplifications are made to the model and described below.

3.2.1 Clock and Clock Regions

The first simplification is the clock speed, which can be set independently for each page region, and ideally the clock speed is allowed to vary for every page region according to the IP. This requires the ability to operate different routers at different clock speeds. In this work it is assumed that page IP operates with the same clock speed as the NoC model and constrains the whole FPGA to operate at one clock speed for the sake of prototyping. The network as well the cores operate with the same clock speed and are limited by the slowest component.

3.2.2 IP-Cores

In this work it is assumed that the page is large enough to fit any core on it, and not constrained to a fix region or timing constraints on the device, which would be needed for a mesh structure. Further, this work uses just cores which provide a built-in blocking mechanism, or does not need such communication type.

3.3 HERMES - NoC Infrastructure

HERMES is the name of a NoC infrastructure developed by the group GAPH from Catholic University of Rio Grande do Sul, Brasil [20] and proposed in by Moares et al. in [22]. This section gives a top-down overview of important details of this NoC. Section 3.3.1 gives a role of the HERMES NoC and provided services. The routers and interface implemented in HERMES are

described in detail in Section 3.3.2. Section 3.3.3 presents an adaptation of the interface to support partial dynamic configuration.

3.3.1 Network on Chip

HERMES is an infrastructure used to generate NoCs with packet switching, which is adaptable for different topologies, flit sizes, buffer depths and routing algorithms. The HERMES name is also employed to refer to the NoCs implemented with this infrastructure and to the other components of this network, like routers and buffers.

With the HERMES infrastructure it is possible to implement the three lower levels of the ISO OSI Reference Model: (i) physical - corresponding to the definition of the router physical wiring interface; (ii) link - which defines the data transfer protocol between routers (the HERMES infrastructure adopts an explicit handshake protocol for sending and receiving data reliably) (iii) network - corresponding to the level at which the switching mode employed by the NoC is defined.

3.3.2 HERMES Router and Interface

The HERMES infrastructure assumes wormhole packet switching is used. The main component that implements this characteristic is the HERMES router shown in Figure 3.2. This router contains two parts: control logic and a set of up to 5 bidirectional ports: *East*, *West*, *North*, *South* and *Local*. Each port contains a queue to temporarily store packet flits, and whose size is parameterizable at design time. The Local port establishes the commu-

nication path between the processing core and the NoC, and from the local router to any other core in the system. The remaining ports connect routers together. The control logic is composed of two modules: routing and arbitration. The routing module implements one of the algorithms made available by the HERMES infrastructure. The arbitration module determines which packet must receive priority to be switched inside the router when more than one packet arrives simultaneously at the router requiring the same output port. A dynamic arbitration scheme(*round-robin*) is assumed by the HERMES infrastructure [22].

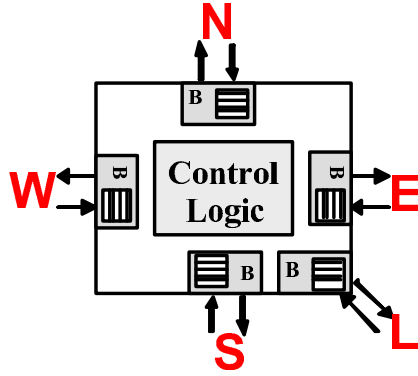


Figure 3.2: HERMES router architecture. *B* indicates input buffers

The routing algorithm defines the path taken by a packet between the source and the destination. The deterministic XY routing algorithm is taken. The HERMES NoC follows a mesh topology, justified to facilitate routing, IP cores placement and chip layout generation. The routers use an 8-bit flit size in parallel (bandwidth of the link $n = 8$), and the maximum number of flits in a packet is fixed at $2^{(flitsizeinbits)}$. The first and the second flits of a packet are header information, being respectively the address of the target router, named header flit, and the number of flits in the packet payload. An asynchronous handshake protocol is used between neighboring routers. The

physical interface between routers is shown in Figure 3.3 and is composed of the following signals:

- *tx*: control signal indicating data availability;
- *data_out*: data to be sent;
- *ack_tx*: control signal indicating successful data reception.
- *rx*: control signal indicating data availability;
- *data_in*: data to be received;
- *ack_rx*: control signal indicating successful data reception.

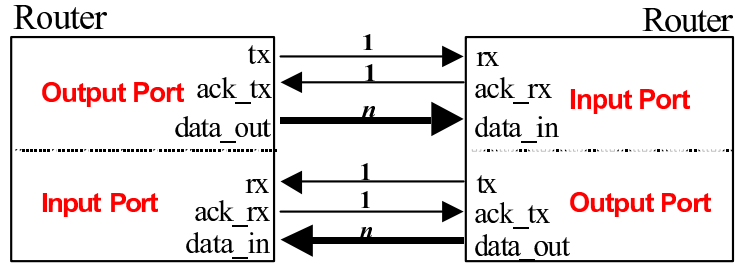


Figure 3.3: Physical interface between routers in HERMES framework

The control logic implements the routing and arbitration algorithms. When a router receives a header flit, arbitration is performed, and if the incoming packet request is granted, an XY routing algorithm is executed to connect the input port data to the correct output port. If the chosen port is busy, the header flit, as well as all subsequent flits of this packet, will be blocked in the input buffers. The routing request for this packet will remain active until a connection is established in some future execution of the procedure in this router. When the XY routing algorithm finds a free output port

to use, the connection between the input port and the output port is established. After routing all flits of the packet the connection is closed. At the operating frequency of 50MHz, with a word size (flit) of 8 bits the theoretical peak throughput of each HERMES router over one link is 1 Gbits/s [22]. This is because a router can establish up to five connections simultaneously (Lin->Eout, Ein->Nout, Nin->Wout, Win->Sou, Sin->Lout, as referred in Figure 3.2).

Arbitration logic is used to grant access to an output port when one or more input ports require a connection at the same time. At the first step round-robin arbitration scheme is used to avoid starvation. Thereafter a prioritized arbitration scheme is implemented, to grant priorities to different packets, depending upon type or source. When a flit is blocked in a given router, the performance of the network is affected, since several flits belonging to the same packet may be blocked in several intermediate routers. To lessen the performance loss, a 4-flit buffer is added to each input router port, reducing the number of routers affected by the blocked flits. Larger buffers can provide enhanced NoC performance. Buffers work as circular FIFOs. The minimal latency in clock cycles to transfer a packet from source to destination is given by:

$$latency (min) = (\sum_{i=1}^n R_i) + P * 2$$

where: n is the number of routers in the communication path (source and target included), R_i is the time required by the routing algorithm at each router (at least 7 clock cycles), and P is the packet size. This number is

multiplied by 2 because each flit requires 2 clock cycles to be sent in case the path is free, due to the handshake protocol.

3.3.3 HERMES Interface Adaptation for PDR

NoCs are good choices due to their scalability, increased parallelism and short-range wires that reduce power consumption. This section proposes Artemis, a NoC that supports specific reconfiguration services and is based on the HERMES NoC [22]. This Section describes the modifications carried out in HERMES to allow its use in dynamically reconfigurable systems. Section 3.1 presented the requirements of a partial reconfiguration process. To fulfill these, a set of services is added to the NoC.

Three services are implemented in Artemis: (i) reconfigurable area insulation; (ii) packet discarding; (iii) reconfigurable area reconnection. HERMES added of two functionalities to support these services: (i) definition of control packets, enabling IPs to send packets to routers, not only to other IPs; and (ii) the capacity to disconnect/connect routers to its associated reconfigurable area.

The addition of two sideband signals per port to the original HERMES router serves to differentiate control packets from data packets. These signals, depicted in Figure 3.4, are *ctrl_in* and *ctrl_out*. For each flit sent by *data_out*, the *ctrl_out* is asserted together with *tx* if the flit is a control packet. The target router receives flits analogously, using *data_in*, *rx* and *ctrl_in* signals. When the reconfigurable area is insulated, the router discards any data packets sent to the area under reconfiguration. Insulation also protects the network, since during reconfiguration transients can occur in the reconfig-

urable interface. Once the new IP is configured, a control packet reconnects IP and router, enabling normal operation.

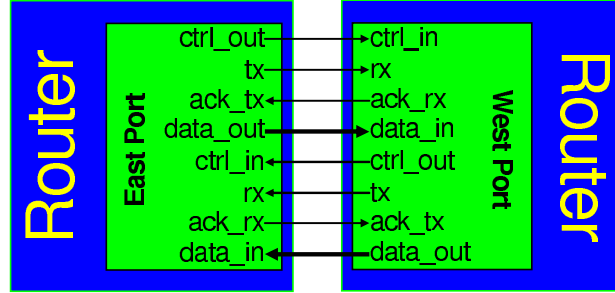


Figure 3.4: Interface between Artemis routers

The reception and forwarding of control and data packets is similar. The major change in the router is the addition of one bit at each position of the input buffer. This is required to propagate the value of the *ctrl_out* signal to the reconfigurable IP router. When the control packet arrives at its destination router, it decodes and executes the corresponding operation.

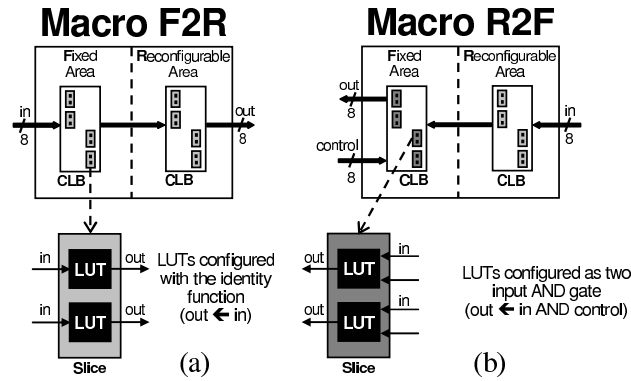


Figure 3.5: GAPH macros: (a) Fixed to Reconfigurable area (F2R); (b) Reconfigurable to Fixed area (R2F) [20]

To delimit the static region and the partial reconfiguration region GAPH macros are used, it is a new reconfigurable interface that does not impose the use of a specific communication infrastructure. This interface uses LUTs.

Two unidirectional macros compose the reconfigurable interface, as depicted in Figure 3.5. The first one, named F2R, is responsible to send data from the fixed part of the system to a reconfigurable IP, while the second one, named R2F, implements the communication in the reverse direction. Both macros allow the simultaneous transmission of 8 data bits. The F2R macro is an identity function, while the R2F uses special logic to avoid transient glitches during the reconfiguration process from reconfigurable to fixed areas [20].

The complete interface between the Artemis router and a reconfigurable IP appears in Figure 3.6. It uses two R2F macros to connect 10 bits from right to left and two F2R macros to connect 11 bits in the opposite direction. The interface between the router and the reconfigurable IP does not contain the *ctrl_in* and *ctrl_out* signals because reconfigurable IPs neither send nor receive control packets. The reset is a global signal used to initialize the entire system. The router asserts the *reconf* signal to initialize the reconfigurable core connected to the local port. The *reconf_n* signal in Figure 3.6 connects to the control signal in Figure 3.4, controlling the connection from the router to the reconfigurable core.

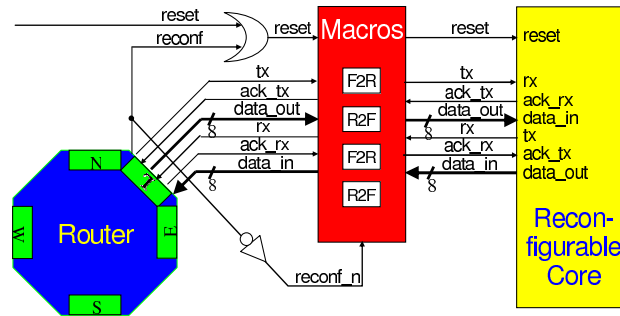


Figure 3.6: Interface between Router and page, supporting partial dynamic reconfigurable. Consists of Bus Macros from Figure 3.5 and control signals

3.4 Network Interface cores - Translator between Cores and NoC

This section addresses the network interface core, which has the central role in interpreting signals coming from the NoC to a custom core or page and vice versa. In the literature common names for it are *glue logic*, *adapter* or *wrapper*. This thesis will use the more specific NoC name *Network Interface Adapter* (NWIF). Section 3.4.1 lists all the responsibilities and issues involved in creating a NWIF. A guideline for designing an NWIF is presented in Section 3.4.2, where various aspects which need to be kept in mind are presented and discussed.

3.4.1 Role of the Network Interface

The NWIF plays the role of glue logic between a router and a logic page. The page logic is producing some serial or parallel data: this data has to be assembled by the adapter into packets accepted by the router. Furthermore, the NWIF is responsible for the packet format and its destination. In the opposite direction the NWIF disassembles arriving packets from the NoC and translates them to serial or parallel data for the core/page. Another important role of the NWIF is the handling of blocking, where the core/page is blocked and the state of the core/page is recorded in the NWIF for further processing after unblocking.

The complexity of a NWIF is related to the interface of a given core/page, because it has to implement all translations from NoC to the core and back

for a given set of functionalities. Usually the NWIF needs to be implemented by the designer who wants to use a custom core in an NoC environment.

The interface between NWIF and router is well known (see Figure 3.6), but the interface between NWIF and core/page may vary. At this point it is desirable to develop a method for automatically generating the NWIFs for a set of IP's or a guideline for designing NWIF to assure reusability, functionality and complexity reduction.

3.4.2 Design guidelines for Network Interface

The design guide is split into two parts: the first part describes general construction suggestions, such as the role of different FSMs, blocking mechanisms and decoupling of control, the second part in this guide is concerned with a set of communication types and how they are supposed to be handled in the NWIF core.

It is very important to design a NWIF in a reusable manner and to try to keep the complexity of the design as low as possible to optimize the power and space consumption. These ideas were implemented and showed good performance and reusability:

- *Decoupling* - this is a very important property for a well-designed NWIF. The state of the core is supposed to be saved and managed strictly by the NWIF of a particular core. For example if there is a core which writes/reads data to/from memory the NWIF must save all relevant data locally and the memory NWIF must provide the capability to write and read to or from memory. So the address of data is

kept at the local NWIF of a core and a request is sent to the memory controller (controlled by memory NWIF) to read data from a particular address. This allows the complexity to be kept low, since every NWIF implements just the capabilities of the local core.

- *Receiving FSM* - senses the NoC for new packets addressed to the local core. If the local core is required to process a new packet, this FSM receives the packet, determines the format of the packet and finally disassemble/interprets it. To be able to do so it needs to implement all packet formats intended for the associated core and needs routines for disassembling the packets. The role of the *Receiving FSM* is over when the FSM has received a packet, written all relevant data into registers, and given a sign to the *Control State FSM* that new data needs to be processed. Since it plays a role in the blocking mechanism, it is supposed to receive a packet only if the local core can handle it, this is determined in the *Control State FSM*.
- *Control State FSM* - is the FSM which communicates directly with the local core. The role of this FSM is to control the local core, by implementing the core side of the interface. The *Control State FSM* also saves relevant state of the local core and according to this chooses the next control action. All the data coming from the core is buffered in registers, where the data is assembled into packet form. A Further responsibility is to provide signals to other FSMs: blocking signals from the *Receiving FSM* during processing of current packet and signaling the *Sending FSM* when packet is ready for sending.

- *Sending FSM* - senses the ready signal set by *Control State FSM* for sending a packet into NoC. It assemble the packet from registers preset by other FSMs and sends the packet. During sending it also releases a signal for the *Receiving FSM* to be able to receive the next packet.

The following describes the more detailed NoC support for different types of communication. Here the target is some design rules to handle some signals in the NoC NWIF. Each type of communication needs to be modeled by an asynchronous, blockable communication construct.

Synchronous communication causes problems when blocking needs to be implemented. The easiest way is to implement clock disable, which would allows the core to hold in a particular state and wait on the further activation through the *Control State FSM*. The same is the case for other signal types which absolutely needs to be translated into asynchronous communication type.

An example with synchronous communication would be a pipelined divider which needs each clock cycle an input and after a specific number of cycles the corresponding results are at the output, which needs to be read in a synchronous manner. For NWIF it means that either the operation is made at once synchronous and the results are transmitted over the NoC or the core is blocked after each results is read and waiting for the next request for the next result. The same techniques can be used for all other signal types such as level sensitive, edge sensitive.

To support different signal types the NWIF needs to implement a set of packet formats which have the role of transmitting data and controlling in-

formation flow between cores/pages. The general layout and custom example set of packets is described in the next section.

3.4.3 Packet Layout and different Formats

As previously described, the wormhole mode implies dividing packets into flits. The flit size for the Hermes infrastructure is parameterizable, and the number of flits in a packet is fixed at $2^{(flitsizeinbits)}$. Figure 3.7 shows a general layout, where the first and the second flit of a packet are header information for the IP layer, being respectively the address of the target switch, named header flit, and the number of flits in the packet payload. The third and fourth flits are packet format and sender information for the application layer and are evaluated only in the NWIF.

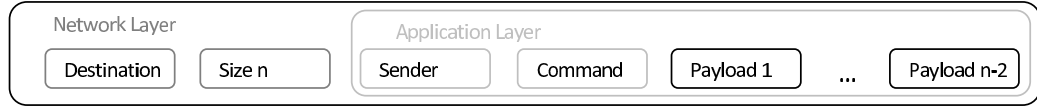


Figure 3.7: General packet Layout, illustrating the layout in terms of OSI layers

A set of general formats could be:

1. *read from memory*, is used to request data from memory controller from a specific location specified in the payload;
2. *write data to memory*, is used to write the data to memory at a specific location, this format may also be used to deliver different results on different requests;

3. *return data*, delivering data from one core to another, some subformats could be implemented as payload header to achieve general formats which are supported by all NWIFs;
4. *new request*, start a new process in the core, here the relevant information is also in the payload, which may vary between different cores;
5. *request ready*, signaling that a process from a particular core is ready;

3.5 Summary

This chapter described the requirements of a paged FPGA design which are necessary for partial dynamic reconfiguration support. The main requirement is for a general communication method, which supports inter-page communication and off-chip communication. The best technique to support this requirement, is a NoC with fixed, general infrastructure. As a part of the communication method a general interface is needed for detaching and attaching a reconfigured page to the communication method. A general interface needs to be implemented by both sides: NoC and a dynamic reconfigurable part. This makes it possible to reprogram the pages while the device is running and without any further adaptation of the communication interface.

HERMES NoC was determined to be a suitable communication framework, as it offers a network which has a low area overhead and offers parametrized implementation. Further, it offers an generic interface which supports partial dynamic reconfiguration and which protect the network during reconfiguration by insulating it from unwanted data.

Since glue logic is required for cores to support easy communication, some guidelines are presented in this chapter as well as a rough structure of such an Network Interface (NWIF) with a standardized set of packet formats. The main idea here is to decouple the control logic and allocate it directly to the NWIF responsible for a particular core/page. Overall, the design of NWIF is supposed to consider only the functionalities of a given core. This simplifies the design complexity and reduce potential pitfalls.

Chapter 4

Custom Design for Dual AES

4.1 Introduction

The decision was made to implement a conventional design as a reference and to then examine the performance against a NoC based implementation. This chapter is about implementation details of the custom design for two AES cores. The role of one AES core is to encrypt 128 bit data with a 128 bit key. The design contains two AES cores, so as to examine the concurrency behaviour on the off-chip communication. Section 4.2 gives an overview of the function and interface of an AES cores. In Section 4.3, the custom design implementation and module interaction is described. The interface between the host computer, as well the implementation of the host application, is presented in Section 4.4.

4.2 AES

Advanced Encryption Standard (AES) [7] is an encryption standard, adopted in 2000 by National Institute of Standards and Technology (NIST) as a successor to the Data Encryption Standard (DES). Each AES cipher has a 128-bit block size, with key sizes of 128, 192 or 256 bits, respectively. The AES ciphers have been analyzed extensively and are now used worldwide.

This work uses an AES core, provided by the ISS institute of Darmstadt University of Technology, as a prototypical core for simulating a real application in the reference design. The interface of the AES core is shown in Figure 4.1. The core has three inputs and two outputs. The inputs are *dataIn* and *Key*, respectively 128 bits, and a *start_aes* signal, which asserts the start of the encryption with the given data. The outputs are *aes_ready* signal, which signals that the encryption is finished, and the encrypted data is available on the output *encData*.

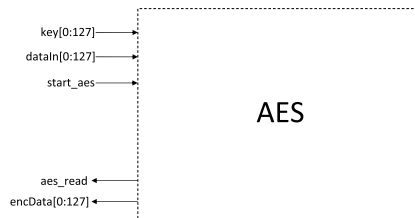


Figure 4.1: AES core having three inputs and two outputs. Inputs are key and dataIn both 128 bits and start. The outputs are encData which is also 128 bits and ready. The AES core encrypts the data with a given key. The start signal is a control signal; the ready signal indicates when the data is finished with encryption

The AES core needs 20 clock cycles after the *aes_start* is asserted to encrypt the data and rise the *aes_ready* signal.

4.3 Custom Design without NoC and two AES Cores

The custom design is implemented using conventional techniques, which are usual for implementing in today's FPGA applications. The modules are connected via point-to-point connections and the design is optimized for a single task application which has the role to encrypting parallel two blocks of data with two AES cores. Section 4.3.1 provides a top view of the module organization and communication scheme and presents the memory allocation approach. Section 4.3.2 describes in detail the implementation of the dual AES controller with memory interface. Finally Section 4.3.3 gives an overview of the timing and area for the given implementation.

4.3.1 Dual AES Interface

The Dual AES Interface is the central control unit which connects and controls three different modules: the local bus which constitutes the connection to the host application over the PCI bus, the memory controller which allows application modules to write and read from/to memory and two AES cores, which perform the encryption.

The intermodule interconnection is shown in Figure 4.2. The Sequence of performing encryption is as follows:

1. *Direct Memory Access (DMA) burst*: The host application transmits a block of data to the memory located locally on the Alpha-Data board.

The data is transferred via the *PCI Bus* and *Local Bus* over the *Dual AES Controller* and *Mem Controller* to memory.

2. *Control Registers*: The control registers on the FPGA may be written and read from the host application, so the host application controls the start of encryption and is able to sense the end of encryption caused by an interrupt. Usually after the data is written to the memory the host controller sets appropriate registers on the FPGA which provide the memory address and the length of data, and asserts the start of the encryption.
3. *Read data*: The *Dual AES Controller* first reads the key and data from the memory by communicating appropriate with the *Mem Controller* and stores these values into registers connected to the *key* and *dataIn* inputs of the AES core.
4. *Encryption start*: After the data is read *aes_start* is asserted and so the encryption process is started.
5. *Encryption ready*: The signal *aes_ready* is asserted when the encryption is ready, when the data on the AES output *encData* is valid and is written to a buffer register.
6. *Write data*: after the encryption is finished the data is written back to the memory to an appropriate location.
7. *Loop* if there is more data to be encrypted, the procedure is repeated from step 3, except only the next data is read, otherwise there is no more data to be encrypted and processing continue with the next step.

8. *Interrupt the Host* After encryption is done an interrupt is set in a control register and the host application may read the encrypted data.

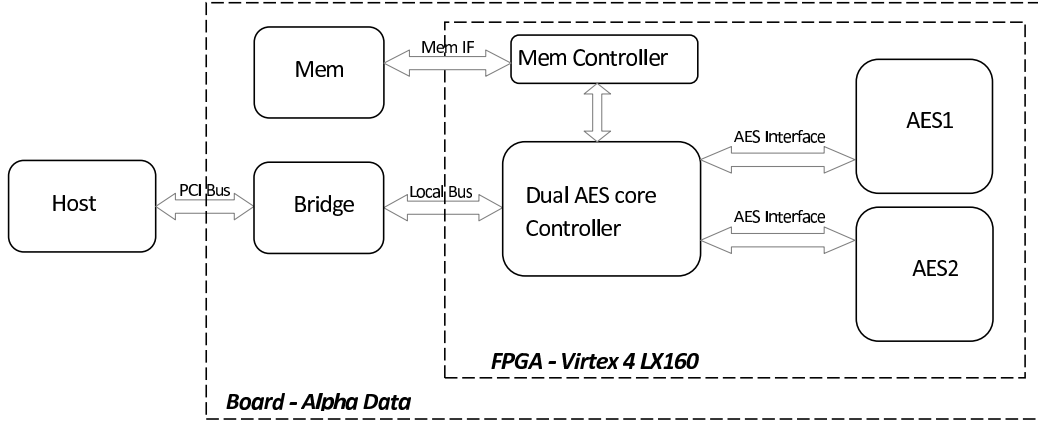


Figure 4.2: Design module implementing a custom design with memory controller, dual AES controller and two AES cores

The address of the data and the length of the block of data to be encrypted is stored in the *Dual AES Controller* and is automatically adjusted with each encryption step. The address and the length of data is used for storing encrypted data back into the memory.

The memory arrangement is very simple. A number of blocks (each 128 bits long) is written over DMA to a free memory address and area. The first block is the key for AES encryption, all following blocks are data blocks for encryption. The encrypted blocks are stored directly at the end of the last input block. The detailed presentation of the data arrangement is shown in Figure 4.3, where two areas for an example using two AES cores is drawn. The memory storage concept may be expanded to any number of AES cores.

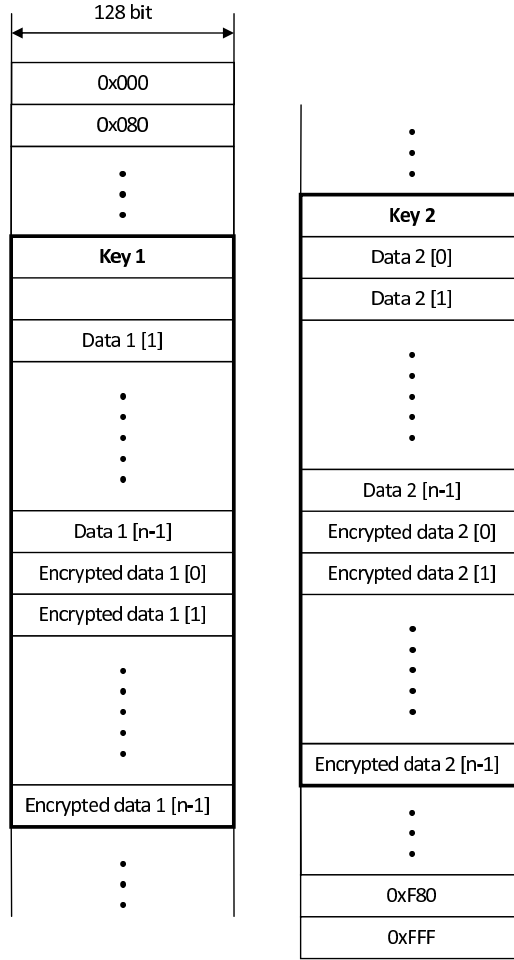


Figure 4.3: Memory map of the communication model between the Host and both designs

4.3.2 Dual AES Controller

The *Dual AES Controller* forms the central control unit. It is connected to all crucial modules and also directly accesses each module except the Host. The internal structure is shown in Figure 4.4. The module consists of one Finite State Machine (FSM) and a couple of buffer registers. In the 32-bit register *DataAddr* the address of the memory location is stored where the first block is located. The 32-bit register *Length* contains the length of the data block

array. The 128-bit registers *Key* and *Data* are the buffer registers storing the current key and data for encryption connected directly to the AES core. The register *EncData* buffers the encrypted results before it is stored into the memory. Since an equivalent set of registers is needed for both AESs Figure 4.4 only shows one set.

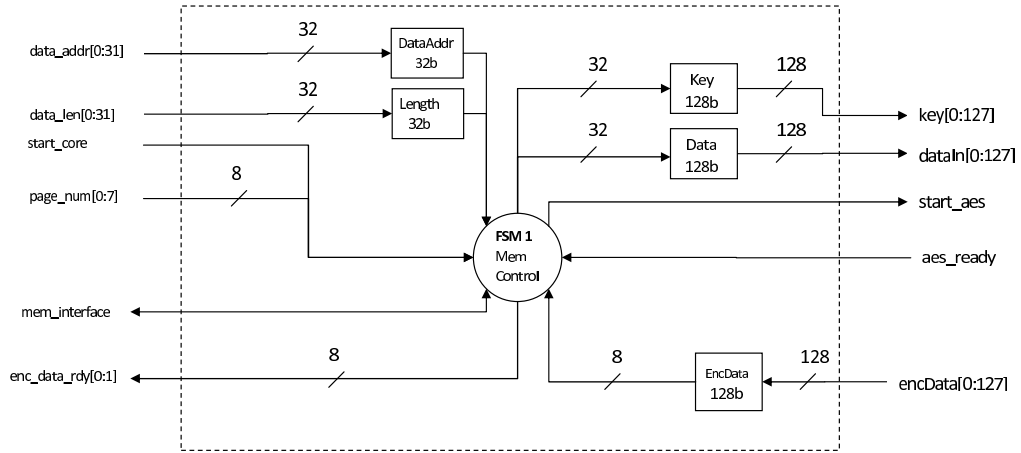


Figure 4.4: Dual AES controller, it has one control FSM and five buffer registers

The central control unit of the *Dual AES Controller* is the FSM1 (see Figure 4.4). The details of the FSM are illustrated in Figure 4.5. The FSM is started with the top left state, which is dependent on a control register accessed by the host. Basically, the FSM implements steps 3 to 8 from the Section 4.3.1. It also has an extension in the state *Change AES*, this state simply changes between processing data from the first AES core to the second, e.g. key read states are composed of two states: *Read Key* and *Read Status*, after the Key for the first AES is read the state *Change AES* changes relevant data, such as `data_addr` and reads the key for the second AES. This happens also by reading data and storing data back to the memory.

The termination of the FSM and going to the IDLE state back, is located in the state *Next data*, if the block counter is less than the length it indicates that not all blocks were encrypted yet, so the next block is encrypted and stored to the memory. If the counter is equal length, it means, that the encryption is completed and the FSM changes the state to the IDLE state, where it can be started again from the host.

FSM1 for custom dual AES Design
Initially n = "0"
len = # of blocks

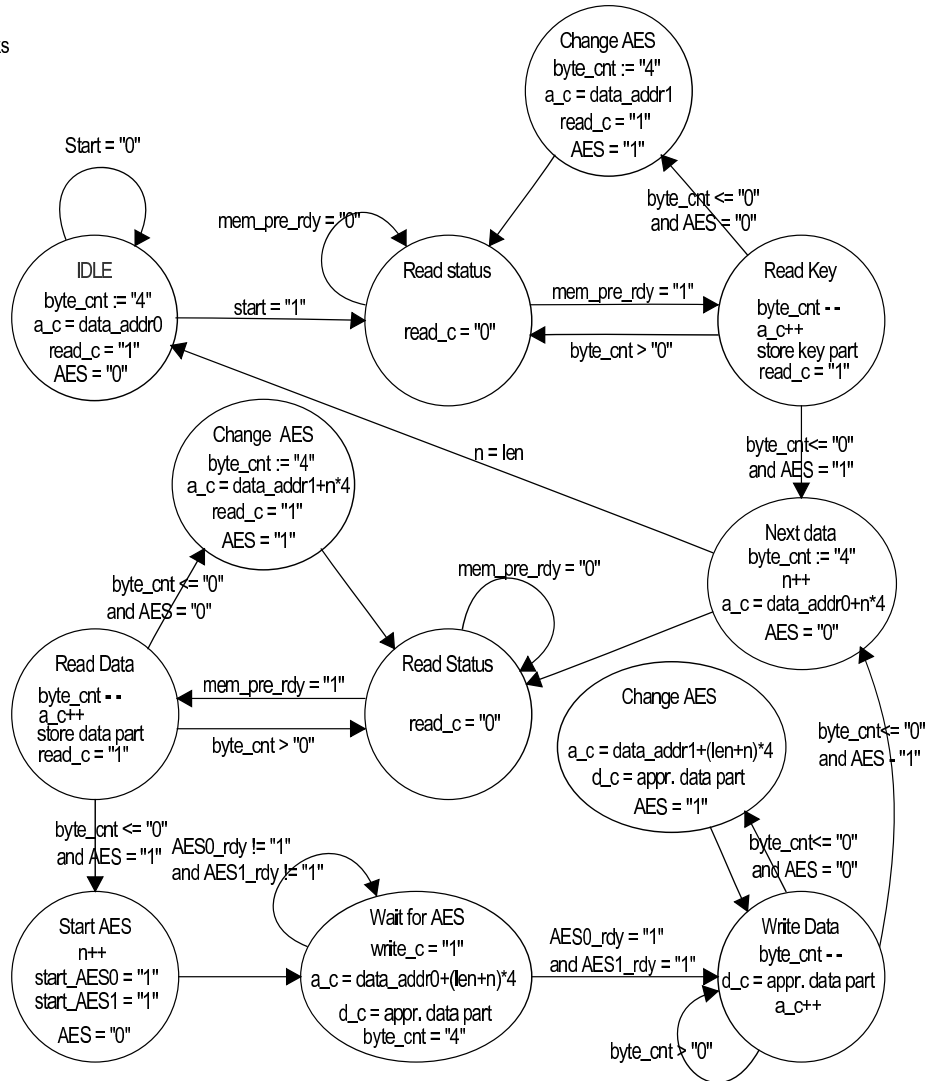


Figure 4.5: FSM 1

4.3.3 Design Area and Timing Analysis

The custom design, represented in this Section, serves as a reference for timing performance and for area consumption. The timing of the current design is 153 MHz, which is the best speed performance compared to the NoC design. The critical path lies in the implementation of the dual AES controller, all other components may run faster. This provides an opportunity to increase the performance, but it was not the goal of this work.

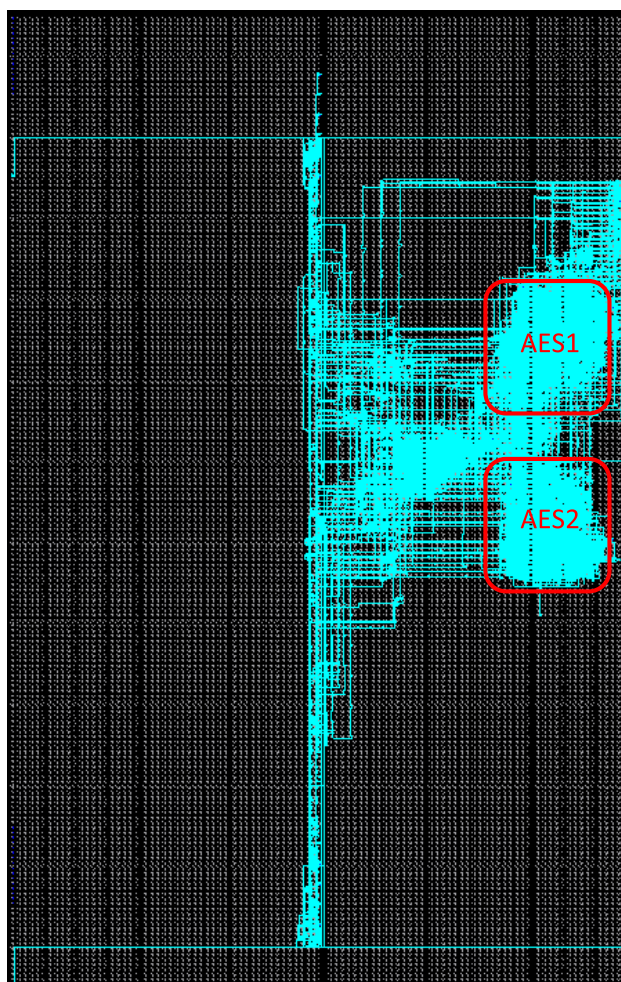


Figure 4.6: FPGA floorplan for custom implementation for dual AES core design

The area of the design is 1787 slices, where the Dual AES controller allocates 258 slices. The floorplan of the dual AES custom design is shown in Figure 4.6, in which only the AES cores are marked.

4.4 Host Control Application

Alpha-Data provides a Software Development Kit (SDK), where a library of routines is available, so for example it is possible to load a bitstream directly from the PCI bus. Another option of the SDK is to map the board registers to an address and access it from the host. After the mapping is successful it is possible to access the local board registers as an array, of course it requires a small VHDL code on the FPGA to save them.

So a couple of control registers were defined to build an interface between the custom design and the host application. Registers which are written from the host to the FPGA are as follows (the number in the brackets identifies the offset to the mapped address):

- AES_DATA_REG (8): register which contains the address of the local memory where a block array is stored for encryption.
- AES_LENGTH_REG (9): register which contains the length of the array.
- PAGE_REG (10): defines the ID of the AES core, which is supposed to perform the encryption
- AES_START_REG (11): registers which asserts the start of a given encryption job.

The appropriate set of the registers above will start the AES encryption on a given core with a given address and length. After encryption is done, the following registers are sensed from the host to receive the information:

- AES_RDY (14): causes an interrupt on the host side.
- AES_NUM (15): provides the ID of the AES core which finished with an interrupt.

The initialization of the FPGA and establishing a connection to the board is also provided by the Alpha-Data SDK. For the memory access it offers some functions which makes it possible to write and read to/from a given address, this simplifies the access to the local board memory.

The access to the memory and also the communication infrastructure to the local bus requires a special FPGA design, which is also provided by the Alpha-Data SDK as an example. They were adopted in both designs.

Chapter 5

NoC Version Design and Implementaion

Following the discussion about different communication approaches in Chapter 2, an NoC framework was identified to be suitable for the requirements presented in Chapter 3. The HERMES NoC framework was chosen to implement the custom design by swapping the point-to-point communication interface to an NoC framework and corresponding core interfaces. This chapter presents an example how, based on a custom design, an infrastructure is built using an NoC as the communication method.

Section 5.1 presents a top-down review of the design implementation: the NoC parameters, packet formats for NoC design, and the procedure for partitioning the interface. Section 5.2 provides an overview of the host application and communication approach between the host and the Alpha-Data board.

5.1 General Design with NoC, Memory Controller and Two AES Cores

The design involves several steps, including: (i) identifying appropriate NoC framework parameters such as the bandwidth, the buffer size and the topology (presented in Section 3.3.2), (ii) the allocation of the control logic (Section 5.1.1) and specifying of the control sequence (Section 5.1.3) with appropriate packet formats (Section 5.1.2), and (iii) implementation, simulation and testing as presented in Sections 5.1.4 to 5.1.7.

5.1.1 Splitting the Dual AES Interface into two Network Interfaces

The method for transforming a custom Dual AES design into an NoC design, in which the interconnections are fixed and are shared between different cores/pages, necessitates a split of the functions covered by the Dual AES Controller. The functions of the custom controller are as follows: (i) controlling the communications to the host, (ii) providing read/write access to the local memory, (iii) controlling relevant address and counter registers for memory access, and (iv) controlling the AES cores.

The functions are split or partitioned across two NWIF cores, namely, the *Mem. Ctrl. NWIF* and the *AES Core NWIF*. Both of these controllers are attached to a router in the NoC. Basically the original custom controller is split into two controllers and the connections between these controllers

is provided by the NoC. The Figure 5.1 represents a high level view of the resulting design.

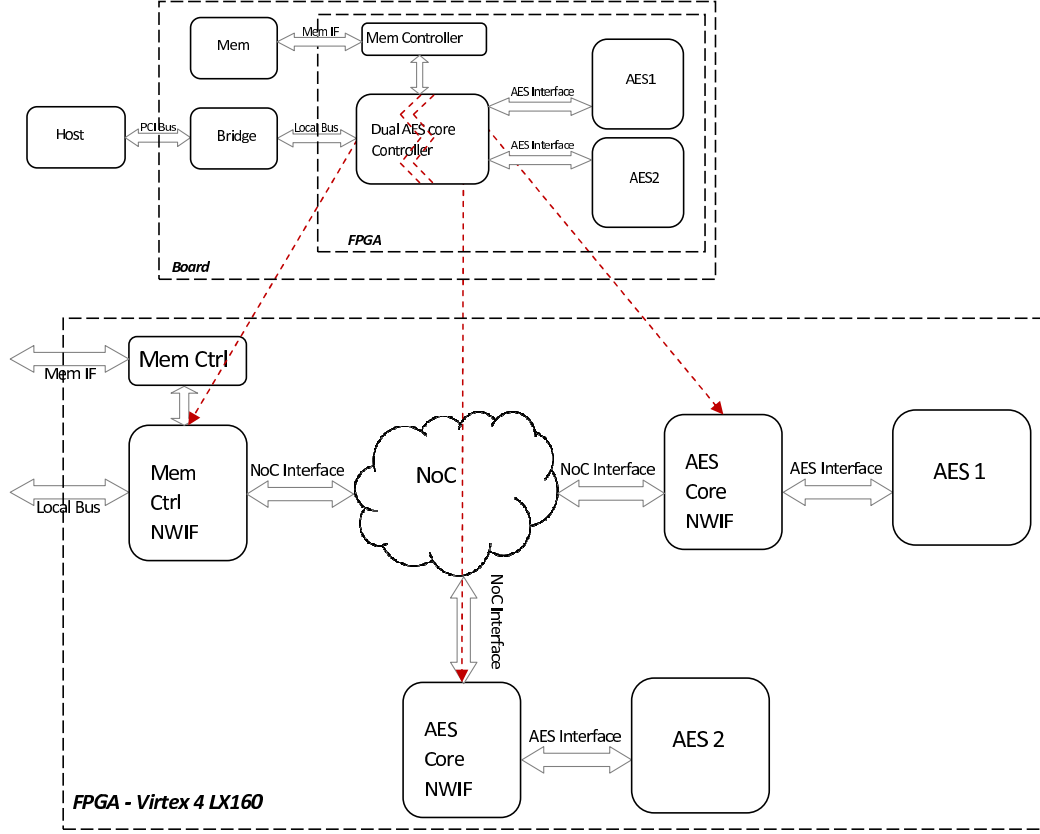


Figure 5.1: Design of the NoC based implementation for multi AES core support. The dual AES core controller is split into two parts: the memory controller NWIF and an AES core NWIF. Custom connections are replaced by the HERMES NoC

The functions of the original controller are distributed over both NWIF cores. The *Mem. Ctrl. NWIF* implements the functions (i) controlling the communications to the host and (ii) providing read/write access to the local memory and the *AES Core NWIF* implements (iii) controlling relevant address and counter registers for memory access, and (iv) controlling the AES cores. The partitioning of functions in such a way allows the relevant

control functions to be decoupled and decentralized. This leads to a lower complexity for each core. In this design the *Mem. Ctrl. NWIF* only has the functionality to communicate with the host and to access the memory, but does not store any control information locally. The *AES Core NWIF* stores all relevant data such as the address and the length of the block array, and computes the control information necessary to access the memory before transmitting it to the *Mem. Ctrl. NWIF*.

By designing the cores in a manner in which the control information is decoupled, it becomes possible to reuse the implementation. In particular, it becomes possible to add more routers to the design and to reuse the design of the *AES Core NWIF* to attach further AES cores without any further design efforts. This was done during the evaluation phase and is described in the Chapter 6.

5.1.2 Packet Formats

The *AES Core NWIF* stores control information, such as the memory address where the first block of the block array is stored and the length of the array. This information is sent from the host application to the *Mem. Ctrl. NWIF*, which sends it on to the *AES Core NWIF*: this type of message forms the first packet format and is shown in Figure 5.2 as *Send new Request*. It has the *cmd* number 00 and the payload contains the address and the length of the array, which are both 32 bits long.

The *AES Core NWIF*, after receiving the control information, requests a key from the *Mem. Ctrl. NWIF* with a packet format called *Request Key* as

shown in Figure 5.2. It has the *cmd* 01 and the payload contains only the 32 bit address of the requested key.

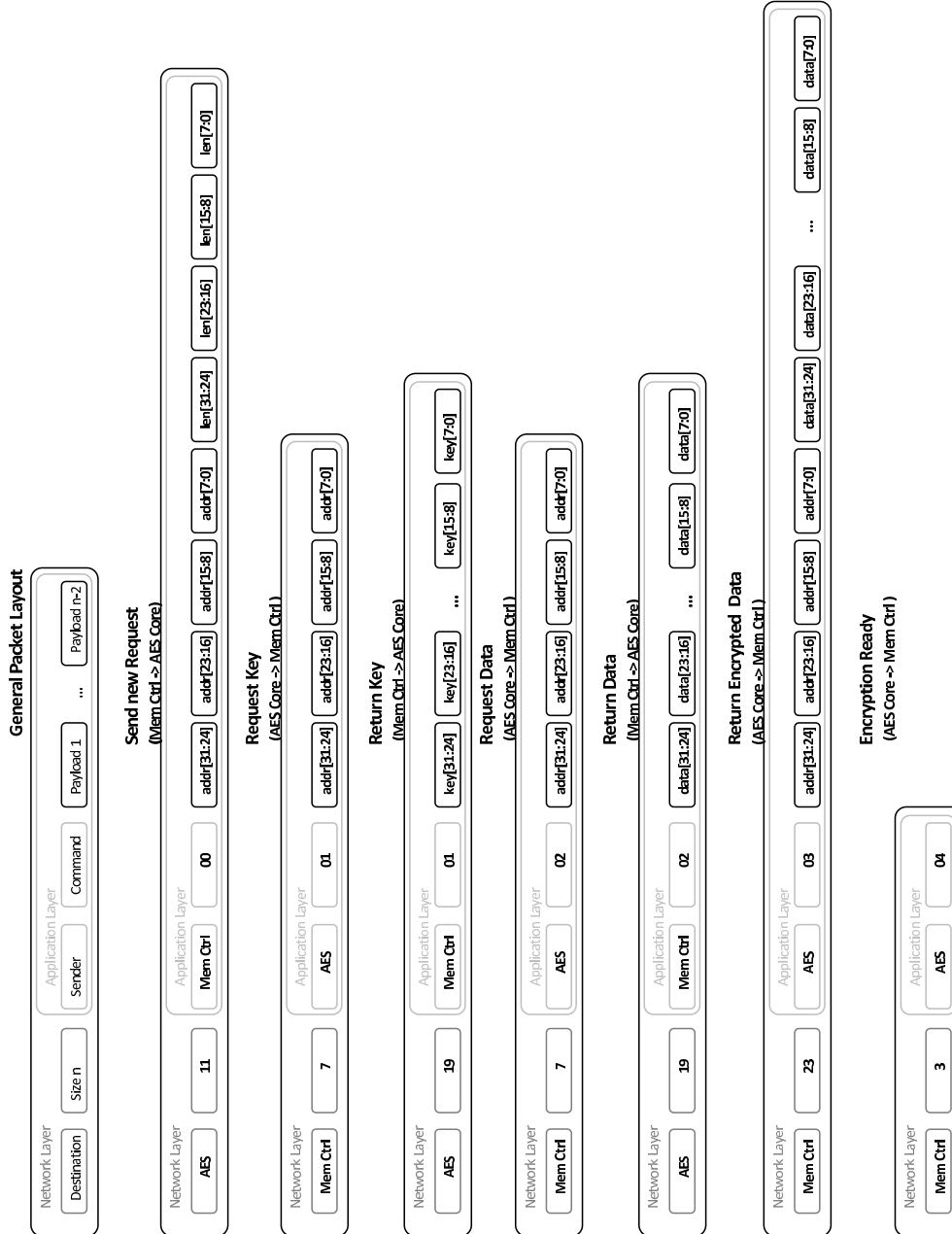


Figure 5.2: Packet formats used in the current implementation. The first general packet Layout illustrates the arrangement and interpretation of each 8-bit flit

The *Mem. Ctrl. NWIF* provides the functionality to access the memory, but only after a request such as a *request key* command. The packet format *Return Key* is a format which delivers the requested information after the request is evaluated and the appropriate data from the memory has been read. This packet format has also *cmd* 01 and contains the 128 bit key as a payload.

The two following packet formats are designed to make the key available for the *Mem. Ctrl. NWIF*. The packet formats *Request Data* and *Return Data* from Figure 5.2 are doing exactly the same as *Request Key* and *Return Key*, except they have the *cmd* 02 to be able to distinguish between key and data. After the key and data are available, the *AES Core NWIF* controls the AES core and performs the encryption. The encrypted data is sent back to the *Mem. Ctrl. NWIF* with an appropriate address where to store it in the memory. The packet format for this transaction is the *Return Encrypted Data* shown in Figure 5.2. It has a *cmd* 03 and the payload contains the encrypted data with an address.

The last packet format indicates an end of encryption job. It is sent from the *AES Core NWIF* to the *Mem. Ctrl. NWIF* to inform the host that processing is finished. The packet format used is *Encryption Ready*, which has the *cmd* 04 and does not have any payload.

5.1.3 Operational Interface

The communication protocol for encrypting some data on one or more AES cores is designed to support the NoC communication and is derived from the packet formats stated above. The sequence of communication is very similar

to the sequence used by the custom design, except that it is standardized into a sequence of messages which share the NoC.

The memory allocation and arrangement is described in detail in Chapter 4 and a diagram presentation is shown in the Figure 4.3.

The UML sequence diagram of Figure 5.3 represents an example of a communication between *Host*, *Mem. Ctrl. NWIF* and one *AES core NWIF*. The example is generic, in so far as it describes a communication method for a general size of problem in which the amount of data is $n - 1$.

The first step for encrypting data is initiated by the host application, which writes an array of data with a certain length to the local Alpha-Data board memory into a defined area of memory. This is done via DMA. After the data is written, the start address the length of the data array and the AES ID of the core that is supposed to perform the encryption are transferred from the host application to the *Mem. Ctrl. NWIF*. After this step is done, the host application starts the encryption by asserting the start signal, by writing to the appropriate FPGA register. The host application starts a timer with the last action and waits for an interrupt signaling the end of the encryption.

After the *Mem. Ctrl. NWIF* receives a start signal from the host application, it sends a packet *Request new Job* with the appropriate memory address and length of the data array. This packet arrives at the *AES Core NWIF* and the information is stored in local registers. The *AES Core NWIF* starts a routine for performing the job, the first step in this routine is to request a key from the *Mem. Ctrl. NWIF* and in response a packet with such an request is sent containing the memory address of the key.

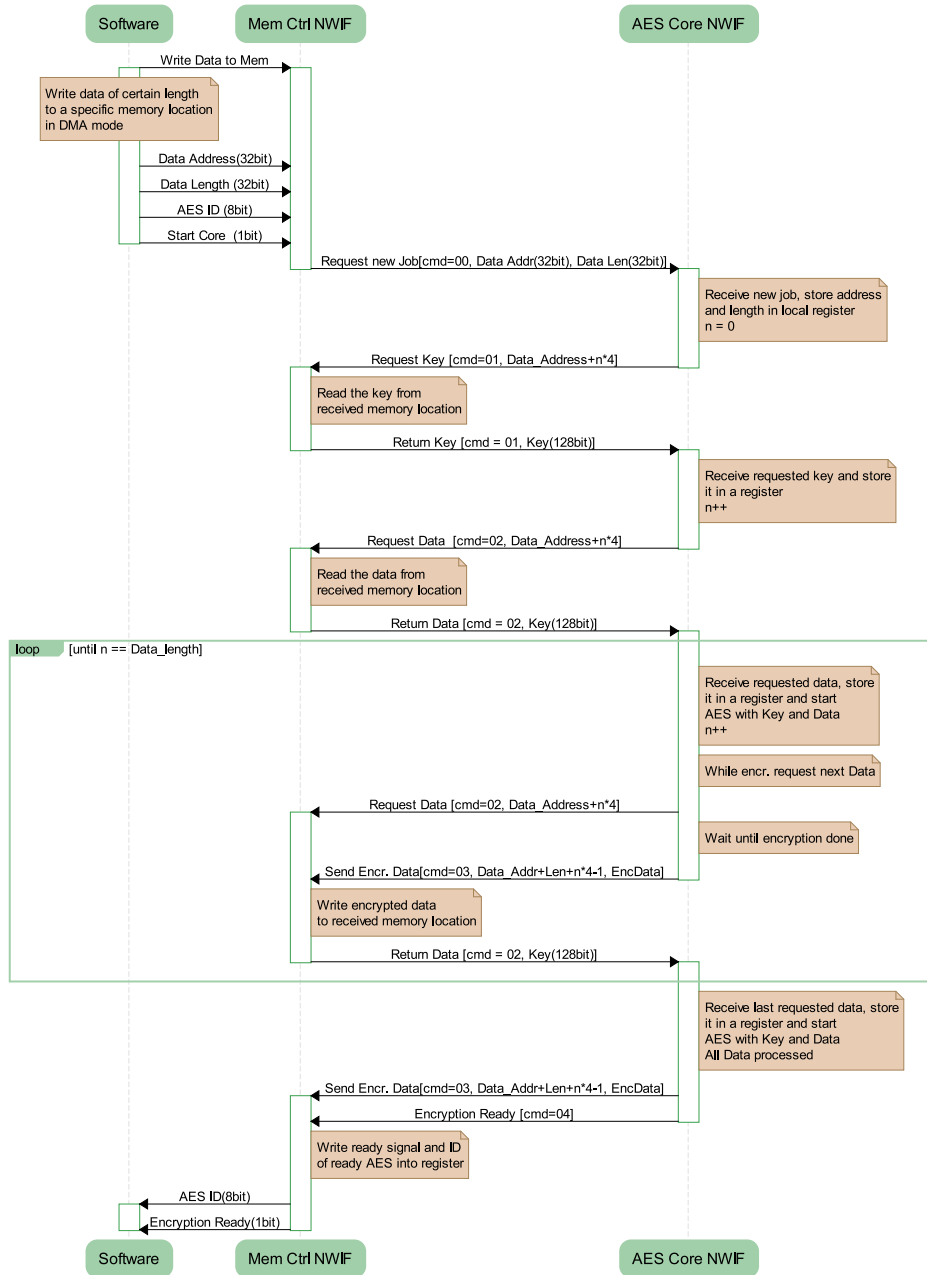


Figure 5.3: Sequence diagram representing an example of communication among the three pcomponents *Host*, *Memory Controller NWIF* and *AES Core NWIF*. In the example the Host writes n blocks (each 128 bits) into memory, encrypts $n-1$ blocks and write the result back into the local memory and notifies the Host via an interrupt that it is ready

After receiving the key request, the *Mem. Ctrl. NWIF* reads the key from the given address and sends it back to the requested *AES Core NWIF*. After the key is received, the routine requests a datum from the *Mem. Ctrl. NWIF*. The data is read and transferred back to the requesting *AES Core NWIF*.

Now the key and data are available on the *AES Core NWIF*, so the AES core is started to perform the encryption. During the encryption phase, the next block of data is requested so as to make use of the link. After the encryption is finished, the encrypted data is sent back to the *Mem. Ctrl. NWIF* with the target address where the encrypted data is to be stored in the local memory.

Previously requested data arrives at the *AES Core NWIF* next, which again is going to be encrypted with AES, where by during the encryption, another block is requested. This represents a loop, which ends when the last datum is received for encryption. After all the data has been encrypted and sent to the *Mem. Ctrl. NWIF* for storage, the *AES Core NWIF* sends an *Encryption Ready* packet to the *Mem. Ctrl. NWIF*, which accordingly asserts an interrupt to the host application.

After the host application receives an interrupt, it stops the timer, reads a register from the FPGA which indicates which AES has finished the job and can upload the encrypted data.

5.1.4 Design of the AES Core NWIF

This section presents the detailed implementation of the *AES Core NWIF*. The interface consists of three Final State Machines (FSM), buffer registers, an interface to the NoC router and an interface to an AES core.

The role of an *AES Core NWIF* is to receive and disassemble packets from the NoC, to control the AES core and to assemble new packets and to send them. The first FSM depicted in Figure 5.4 is responsible for receiving new packets and storing the payload in the appropriate registers. The registers *Address* and *Length* are both 32 bits long and are used to store the memory location and the length of the block array. The registers *Cmd_in* and *Source* are used to interpret the packet format and to buffer the source address. The registers *Key* and *Data* are buffers for current key and address and are directly connected to the appropriate input pins of the AES core.

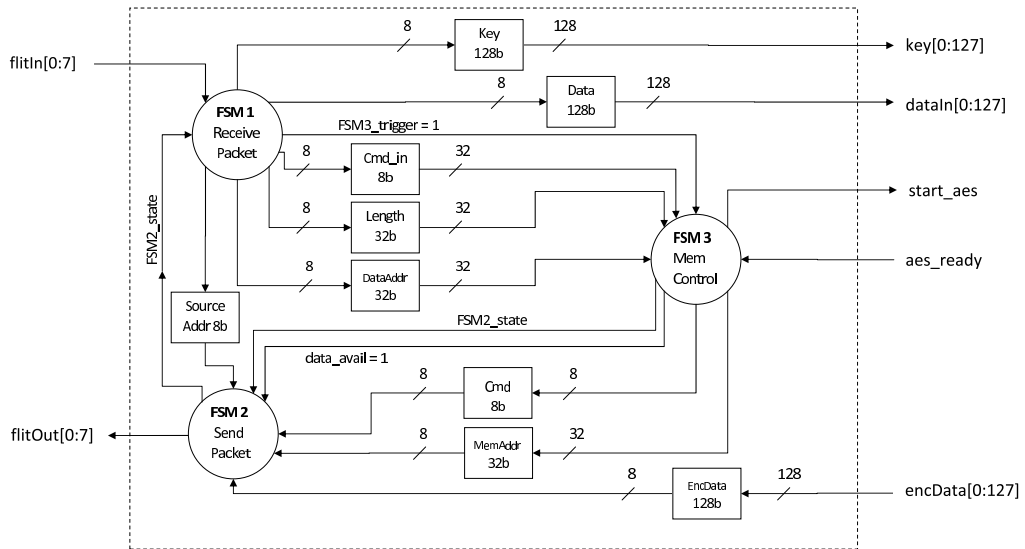


Figure 5.4: The AES NWIF module consists of 3 FSMs and intermediate buffers. FSM 1 receives packets, FSM2 sends packets and FSM 3 controls the AES core and estimates addresses for requesting and reading data

FSM3 is the heart of the *AES Core NWIF*. After a packet is received, the FSM3 decides what to do next according to the packet type: either to request a new key or to request data, to start the encryption, or to send the encrypted data or to send the Encryption ready packet. FSM3 controls the AES and sets the *Cmd* and *MemAddr* registers, which indicate the packet type and the address to be used for the packet.

FSM1 (Receiving) for AES Core NWIF

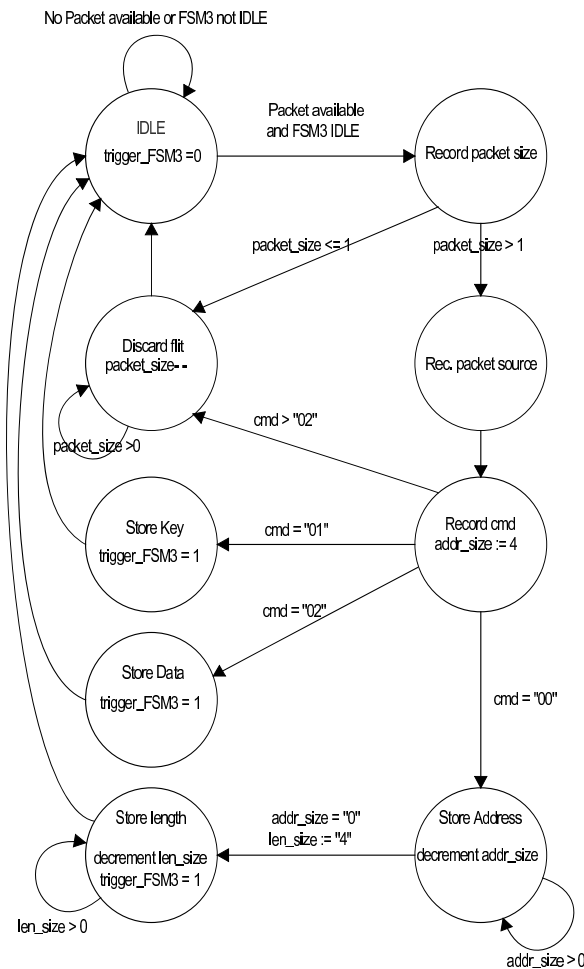


Figure 5.5: AES core NWIF FSM 1. Implements the packet receiving and disassembling routines

The *Cmd* and *MemAddr* and *EncData* registers store information for the next packet to send. *EncData* is directly connected to the output of the AES core. The sending and assembling of packets is the responsibility of FSM2, which reads the relevant registers for a given packet format and sends the packet to the NoC.

Figure 5.5 shows the detailed implementation of FSM1. Initially, the FSM is in state *IDLE*. After a packet becomes available the packet size flit is received. Packets without a valid size are discarded. If the size is valid the next flit is received and stored into the Source register. The next flit is the packet format which is processed in the state *Record cmd* and stored in the register *Cmd_in*. Depending on the packet format, the next state stores the key or the data or the address and the length in appropriate registers. After the data is stored, FSM3 is triggered and the receiving routine is done. FSM1 reenters state *IDLE*, where it waits for the next packet.

After a packet is received, FSM3, which is shown in Figure 5.6, is triggered and changes according to the packet format received in the *IDLE* state to the *Request Key* or *Request Data* state in which the appropriate registers are set and FSM2 is triggered to assemble and send a particular request packet. If the *cmd_in* register is 2 then the *Start AES* state is entered, and the *start_aes* signals are triggered.

If there is more data to encrypt, FSM3 changes state to *Request Data*, where relevant registers are set and FSM2 is triggered to request new data. After the request is done, FSM3 changes to a state where it waits for the completion of the encryption process on the AES core. After the encrypted

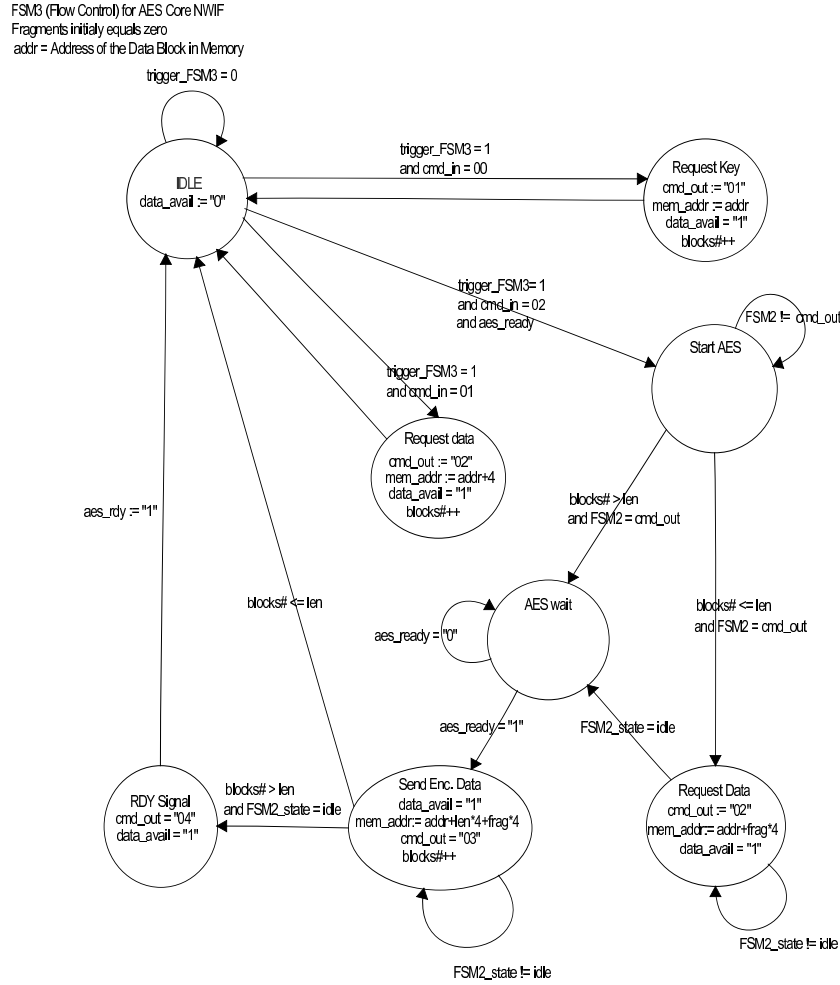


Figure 5.6: AES core NWIF FSM 3. Implements the control logic for the AES core and the next step including all relevant registers for the next packet

data is available the data is stored in intermediate registers and other registers are set and FSM2 is triggered to transmit the encrypted data.

FSM2 is initially in the *IDLE* state and is started with a trigger signal from FSM3. If FSM2 is triggered, it sends according to the registers set by FSM3 and the Source register of the packet. The detailed send behaviour is showed in Figure 5.7. The most relevant register is the *cmd* register, which

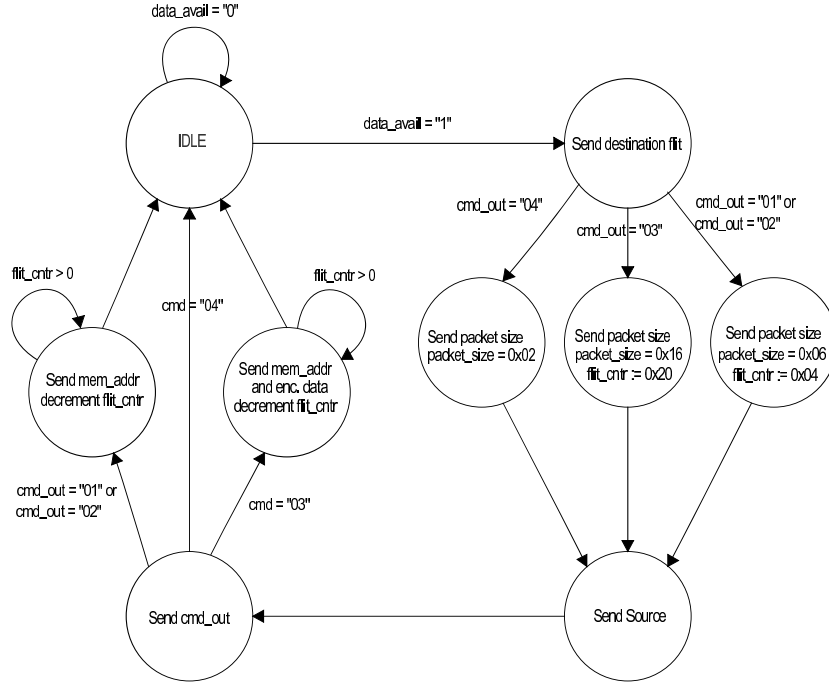


Figure 5.7: AES core NWIF FSM 2. Implements the packet assembling and sending routines

indicates the format of the next packet, according to which the length of the packet is set, and the data is sent.

5.1.5 Design of the Memory Controller NWIF

The *Memory Controller NWIF* has two key responsibilities: it implements the interface to registers which are written by the host application and controls the memory controller. Figure 5.8 shows the internal structure of the core. It also consists of three FSMs like the AES Core NWIF. FSM1 receives the packets, FSM2 sends the packets and FSM3 controls the memory controller.

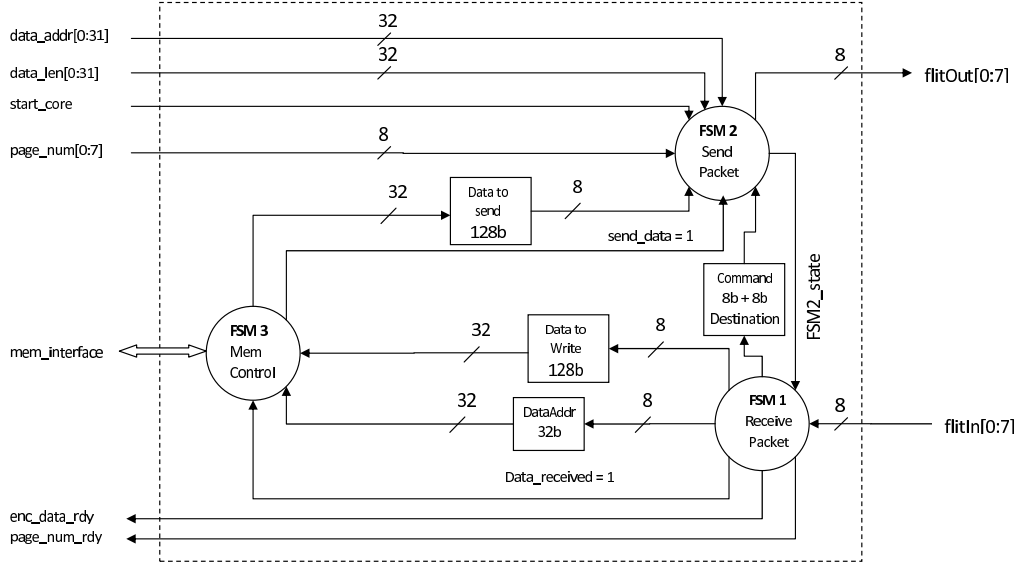


Figure 5.8: The memory NWIF module consists of 3 FSMs and intermediate buffers. FSM 1 receives packets, FSM2 sends packets and FSM 3 controls the memory controller

The detailed implementations of the FSMs are not listed here, since they are very similar to those described above. FSM1 and FSM2 are almost identical to FSM1 and FSM2 from the *AES Core NWIF* and FSM3 is basically described in Section 4.3.1. The implementation of FSM3 in the Memory Controller NWIF is less complex than for custom design and implements only the write and read access to the memory controller.

5.1.6 Simulation

Figure 5.9 shows a simulation of the complete system executing two encryption jobs in parallel. Each encryption job involves encrypting 5 blocks of data. Figure 5.9 combines three different simulation traces aligned on the time axis.

The widest part of the figure contains the simulation of the *Mem Ctr. NWIF*; both smaller parts are respectively the *AES Cores NWIF* for first AES and second AES cores.

The Simulation of each NWIF is organized aligned to the interface of the NWIF: the output link is attendant with a *tx* and *ack_tx* signals to show the detailed handshake communication protocol. The input link is attendant respectively with the *rx* and *ack_rx* signals. The *Mem Ctr. NWIF* simulation part contains the signals *read_c* and *write_c* to see the memory access utilization.

The simulation shows concurrent encryption on two AES cores. The *Mem Ctr. NWIF* provides memory access for both encryptions in a specific, deterministic manner, mediated through requests from each *AES Cores NWIF*. The packets are depicted in Figure 5.9 through dotted rectangles. Some of these rectangles are very wide and this is to be interpreted as a waiting state, because of the network congestion induced by the concurrent access to the *Mem Ctr. NWIF*.

The numbers in the rectangles represent the command numbers corresponding to the packet types presented in Figure 5.2 and the sequence presented in Figure 5.3.

5.1.7 Design Area and Timing Analysis

The NoC consists of an overhead compared to the custom design presented in Chapter 4. An analysis of the timing as well as the area consumption is provided in this section.

The design runs with clock speed of 97 MHz, which is constrained by the implementation of the NoC. Other components such as the AES core, NWIF's allow for better timing performance.

The area required by the design is 4281 slices, where the largest portion is used by the two AES cores, followed by the NoC and the NWIF's. A more detailed evaluation of the area consumption and timing analysis is presented in Section 6.1

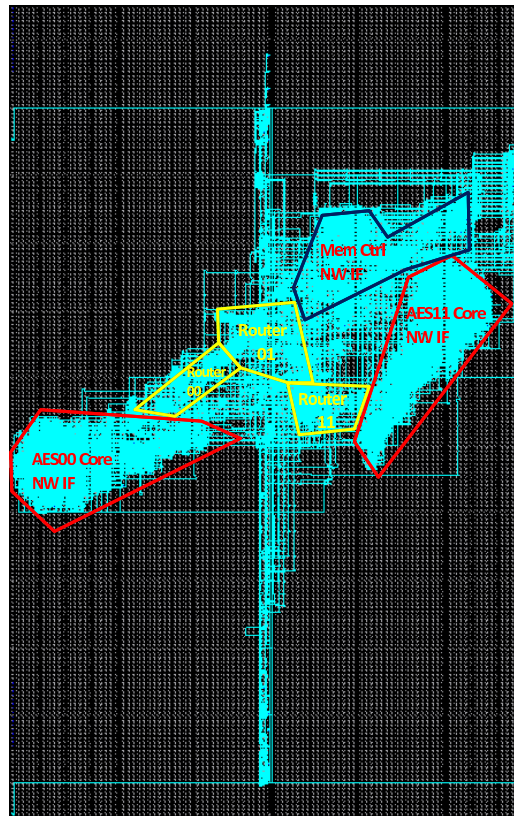


Figure 5.10: FPGA floorplan of the NoC design with two AES cores

Figure 5.10 shows the floorplan of the NoC design. The floorplan shows the allocation of FPGA area to individual system components. It shows one

Mem Ctr. NWIF, two *AES Cores NWIF* including their associated AES cores, and three routers which are attached to each NWIF.

5.2 Host Control Application

The Host application is the same and uses the same interface as for the custom design and is described in Section 4.4.

5.3 Conclusion

This chapter proposed an NoC-based design implementing an AES encryption application. A top-down description shows the top view of the design followed by the detailed representation of individual components and control sequences.

The custom design was partitioned according to the guidelines proposed in Section 3.4.2 and implemented. Different packet formats were defined and built and a deterministic control sequence of communication was developed. The simulation of the design confirmed the proposed functionalities were successfully designed.

After the design was implemented a variety of tests were conducted to assess the performance of the design. The results and evaluation of the NoC based design are presented in Chapter 6, which also contains some proposals for improvement.

Chapter 6

Benchmarks and Results

This Chapter gives an overview about evaluation of the results, their judgment and their potential improvements, which are given by area and timing analysis in Section 6.1, by representing of benchmarks results in Section 6.2, by results discussion in Section 6.3 and by conclusion in Section 6.4.

6.1 Comparison between Area and Timing Analysis for all Designs

An Area and timing analysis of a *custom single AES* design, a *custom dual AES* design and an *NoC multi AES* design are presented in this section. For a better overview of the results the analysis covers every single module of each design as well as the entire designs.

Table 6.1 reports on the area and timing of all three designs as described in Chapters 4 and 5. All designs share the components: *Mem and Bus Ctrl.*

and *AES Core*, which have good timing behavior, and do not limit the overall design performance. The best timing of all submodules was achieved by the *Mem and Bus Ctrl.* at 291 MHz. The AES Core has the second best timing of 223 MHz.

All designs have different timing performances, whereby the best timing was achieved by the implementation of the custom design for a single AES core with 163 MHz. The custom design for dual AES cores is the second fastest implementation with timing of 153 MHz. And the Implementation of NoC design for three AES cores is the slowest design with timing of 97 MHz. It is constrained by the timing behavior of the NoC framework (107 MHz) as well as by the *AES Core NWIF* (108 MHz). The *Mem Ctrl. NWIF* has a timing performance of 152 MHz and like router, with timing of 195 MHz, there is room for improvements.

Module	Area in Slices	Max. Timing
Mem and Bus Ctrl.	686	291 MHz
AES	586	223 MHz
Custom Single AES	1787	163 MHz
AES Ctrl.	183	216 MHz
Custom Dual AES	2656	153 MHz
AES Ctrl.	258	211 MHz
NoC 3 AES	6567	97 MHz
NoC	876	107 MHz
Router	205	195 MHz
AES Core NWIF	723	108 MHz
Mem Ctrl. NWIF	528	152 MHz

Table 6.1: Area and timing analysis of *custom single AES* design, *custom dual AES* design and *NoC multi AES*. First column gives the names of modules, second column shows the area of each modules in slices and the third column shows the maximum speed for each module

The second column in the Table 6.1 represents the area in slices for each design and module. The areas of *Memory and Bus Controller* (686 slices) and AES core (586 slices) are fixed for all designs.

The area differs between the modules involved in controlling the AES and NoC communication. The *AES Ctrl.* in the custom single AES design uses 183 slices, the *AES Ctrl.* for the custom dual AES design allocates 258 slices. The area used by the NWIF controllers is significant bigger and needs for *AES Core NWIF* 723 slices and for *Mem Ctrl. NWIF* 528 slices. The area allocated by the NoC including the routers and communication is 876 slices, where one router uses 205 slices.

6.2 Benchmark Results

After implementation it were necessary to evaluate the performances of all designs. For this purpose some test patterns were formulated, which are described in Section 6.2.1. In Sections 6.2.2 and 6.2.3 benchmark results for one and two AES cores, in custom and in NoC designs, are represented. Section 6.2.4 represents the results for three AES cores in an NoC design.

6.2.1 Test Pattern

This Section introduces some test patterns, which are used in the benchmarks. Defining concrete test patterns allows us to observe and to compare the performance of different designs. The pattern is a simple linear progression: $5K, 10K, 15K \dots 50K$ Blocks. A limitation is made for the benchmark with three simultaneously running AES cores in NoC design. Because the

memory size, used by all designs, is constrained to 4096 Kb, the test pattern for the benchmark with three cores ends at 40K.

The benchmark sends a job to the particular design and start the time measurement. After the encryption is complete, as signalized by an interrupt, the encryption time is recorded in milliseconds. This happens iteratively for each value in the progression, afterwards a table is produced for evaluation. Based on this table, the results are plotted and presented in the next Sections.

6.2.2 One AES

The first test is made with one AES core. The AES core was running in two different designs: the custom design and the NoC design. The test in NoC is split into two variations, whereby, in the first variation the AES core is located one hop away from the Mem Ctrl. NWIF, and in the second variation the AES core is located two hops away. This should demonstrate delays dependent on the hop distance between communication partners. For one hop one flit needs at least 2 clock cycles, this is the case if the NoC does not have a congestion on the particular path. The results are plotted in the Figure 6.1.

The main characteristic of the plot, is the strong linear nature of all graphs. The time needed by AES for encryption in the custom design is significantly less (by approx. a *factor* of 4) than the time needed in the NoC designs. It can be seen from the plot, that the one hop version needs less time (a *factor* of 1.12) compared to the twohop version.

The evaluation of the results is presented in Section 6.3

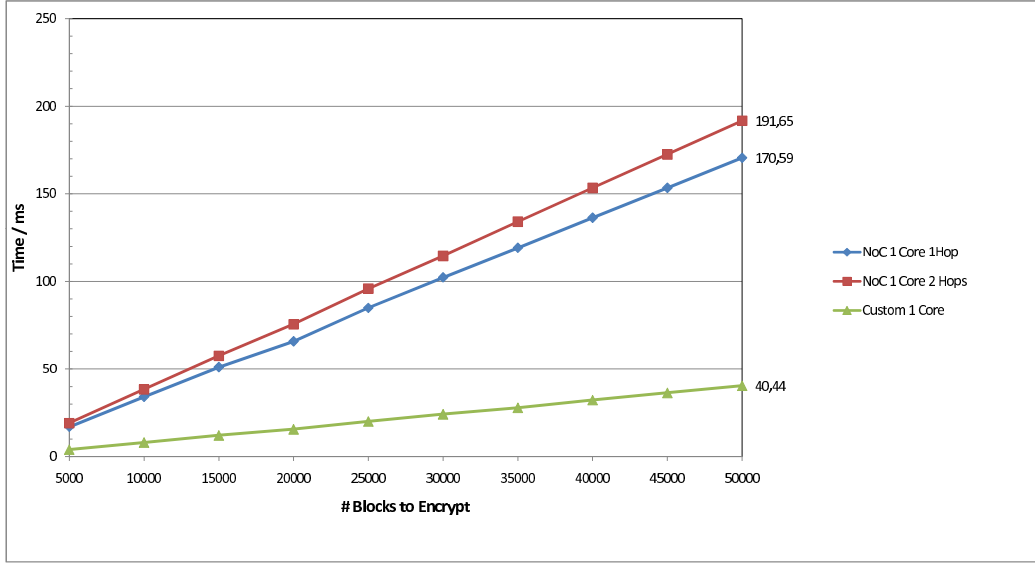


Figure 6.1: Benchmark for single AES core in custom and NoC Design with one and two hops

6.2.3 Two AES

This section presents the benchmarks results for two simultaneously operated AES cores: The first benchmark (Figure 6.2) is designed to compare the time needed by a custom and NoC designs to perform the encryption, with two AES cores, on the test patterns. The second benchmark (Figure 6.5, 6.6) is intended to obtain the difference due to the XY routing approach as the allocation of the two AES cores within the 2x2 NoC mesh is varied.

The plot in Figure 6.2 shows the same behaviour as the previous test, it has a strong linear slope in all three graphs. The performance of the custom design is significantly better (by approx. a *factor* of 3,5) than the performance of the NoC design. The performance within the NoC for *core 11* is better by a *factor* of 1,23 compared that of *core 00*.

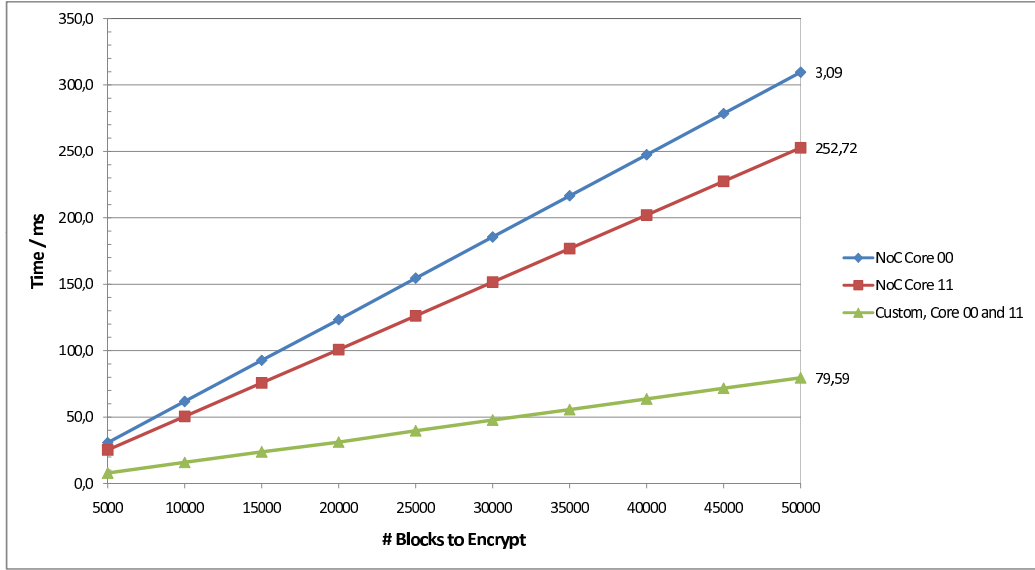


Figure 6.2: Benchmark for two AES core in custom and NoC designs. The core 00 and core 11 lies one hop away from the Mem Ctrl. with ID 01. The arrangement maybe obtained in Figure 6.3

Figures 6.3 and 6.4 represent the constellations for the next test, where the *core 00* is one hop away from Mem Ctrl. NWIF and the *core 10* is two hops away, the routing for the second core differs for input and output. It can be seen in the Figure 6.3, that the output connection (in red/blue) for *core 00* shares the output communication link with the out-traffic produced by *core 10*.

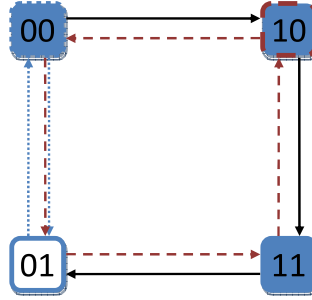


Figure 6.3: Constellation where the *core 00* and *core 10* share, according to the XY routing algorithm, the output link. The Mem. Ctrl is the ID 01. The core 11 does not have any core attached

The opposite constellation is shown in Figure 6.4, where the input link for *core 11* shares the in-traffic for the *core 10*.

The payload of packet formats presented in Section 5.1.2 differs according to the format. If the traffic of an AES NWIF is considered, then the amount of output data is 1.5 times more than the input data since the output also contains a data request. This means, that to perform encryption of one block it sends 256 bits in two packets and receives 192 with one packet.

Since the input data for one AES core, which flows over the network, is less than the output data by a *factor* of 1,5 the performance of these two constellations needs to be evaluated. This results are plotted in the Figures 6.5 and 6.6.

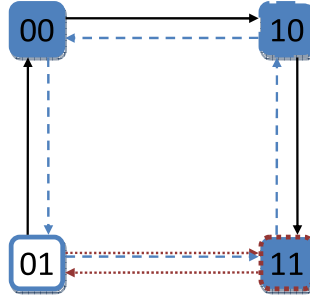


Figure 6.4: Constellation where the *core 11* and *core 10* share the input link

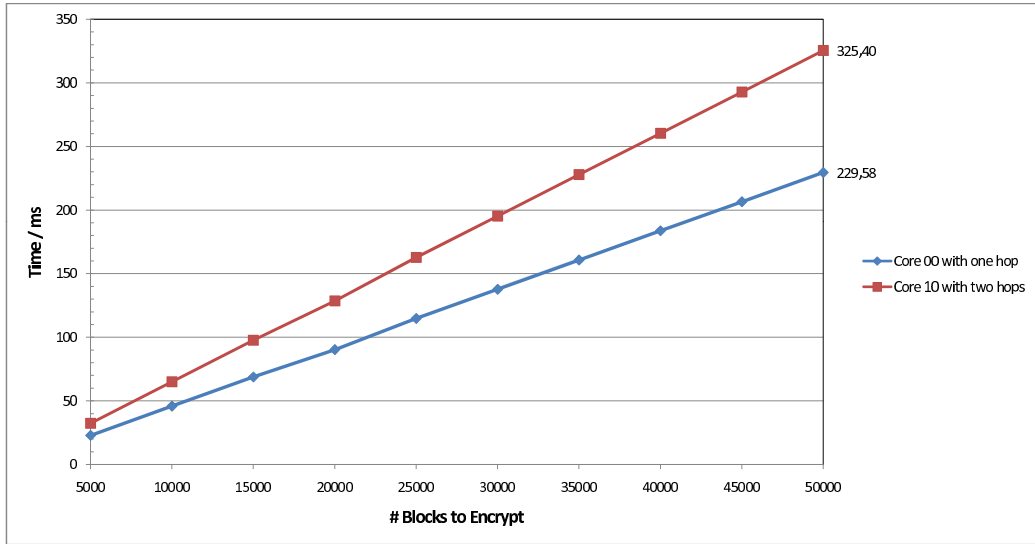


Figure 6.5: Benchmark for two AES cores, where the *core 00* and *core 10* share, according to the XY routing algorithm, the output link. The Mem. Ctrl is the ID 01. The core 00 does not have any core attached

The plot on Figure 6.5 shows, that the encryption made by core 10 needs more time than the core 00. This behaviour was expected when one AES core is two hops away and another one hop from the Mem Ctrl. NWIF. To observe the results in both constellations it is necessary to compare both plots (Figure 6.5 and Figure 6.6) where it can be seen that the constellation in Figure 6.6 has worse performance compared to the plot in the Figure 6.5:

the performance (Figure 6.6) is better for the core 10 than in the plot in the Figure 6.5, but the performance of the second core is more worse. By combining results of both cores the constellation shown in the Figure 6.3 has a slightly better performance.

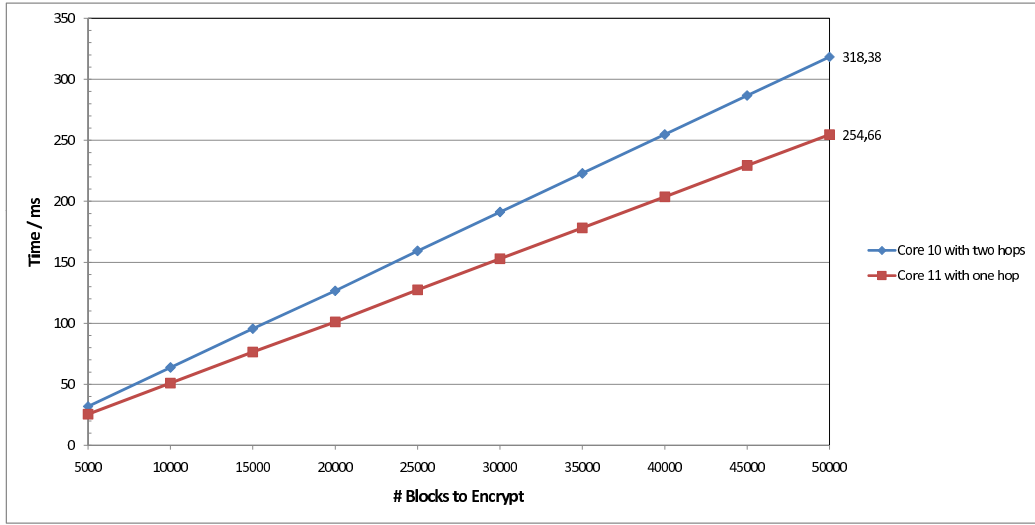


Figure 6.6: Benchmark for two AES cores where, the *core 11* and *core 10* shares the input link

6.2.4 NoC with three AES Cores

This section shows the results for the test with three AES cores, running simultaneously, in the NoC design. As previous mentioned the test pattern for this benchmark is limited to 40K, because the memory is limited to 4096 Kb.

The most conspicuous feature here is the performance of the *core 00* and *core 11*, which are both one hop away from the Mem Ctrl. NWIF. The Performance of those cores is exactly equal during the entire benchmark and like in other benchmark the slope for each core is also linear. The *core 10*

needs in this benchmarks more time (*factor 1,21*) to encrypt the test pattern than both other cores.

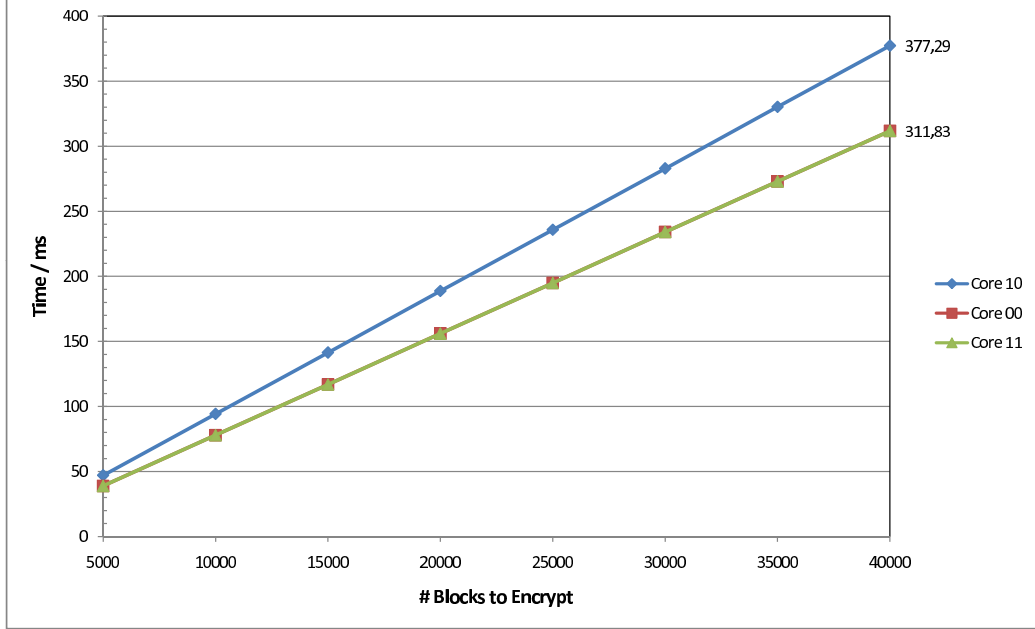


Figure 6.7: Benchmark for three AES cores in the NoC design. The topology is illustrated in Figure 6.6

6.3 Discussion

This section deals with the evaluation of the results presented in Sections 6.2 and 6.1. It is organized as follows: the evaluation of the results is presented in Section 6.3.1, for better understanding the results Section 6.3.2 composes the differences between the designs, which are responsible for the performances. Afterwards, Section 6.3.3 introduces some potential improvements.

6.3.1 Judgment of the Results

As can be seen in Section 6.2, the performances vary between designs. The custom design has in every benchmark better results and it was never the goal to achieve a better performance with NoC. The key points in the NoC framework are the standardized interfaces, small overhead in performance and reusability.

The benchmarks show, that the performance of the NoC implementation is worse by a *factor* of 3, 5 – 4, which causes a significant overhead compared to the custom design. A good characteristic of the benchmarks is, that all graphs have a linear slope, that means that every improvement on the NoC will linearly affect the benchmark results, for each test.

The AES core specification is described in the Section 4.2, it has an 128-bit for data in and out. In the NoC implementation this data is serialized into 8-bit flits, so the custom design needs 2 cycles for input and output, where as the NoC design needs at least 32-bit cycles. Then the NoC design has routers, which also cause delays that are constant. But the biggest delay potentially comes from the congestion in the network. Considering these facts, the results are very good. Of course, the NoC will have in every design worse performance than the custom design, but it is possible to improve the performance and decrease the gap.

The results are also only relevant for a small subset of problems. The results will be better for different problems, where, for example, the data transfered over NoC would only be one flit. Another scenario, which would cause an increase of performance, would be when there is less concurrency

on a communication link, this can be the case when two cores communicate with each other and handshake, rather than contend for the link.

In terms of area, for NoC design, the implementation presented in this work, is also a trade off between performance and area. The design was designed to have a small area overhead and to be as general as possible, so this had an impact on the performance.

6.3.2 Differences between the designs

Each benchmark for the NoC design needs more time for encryption compared to the custom design, but except for the flit size and the subset of problems being processed there are other differences in the designs, which are also responsible for the gap in the performance and we may introduce improvements by adopting them.

The first difference is described in Section 6.1; it is the different timing in all designs. The clock speed of the NoC design is slower by approximately a *factor* of 1,5. If the NoC could be improved, it would potentially have better performance by a *factor* of 1,5.

The second significant difference is the utilization of the memory controller. In the custom design it is possible to achieve a higher utilization for both read and write operations. The reason is the determinism for accessing the memory; for the custom design, it is designed in advance and the memory access is predefined. The first operation is to read a block from memory, afterwards it will be encrypted and written back. This happens in a loop until the encryption is done for all blocks. The memory controller needs 4

cycles to read 32 bits and is able to write every cycle 32 bits back. To read 128 bits, it takes 12 cycles, the encryption needs 20 cycles and to write back the result it takes 4 cycles. The utilization of the read channel is so 33,3% in the custom design and respectively the utilization for the write channel is 11,1%.

The utilization for the NoC is not predictable, since it is dependent on the rate of the packets and they are dependent on the rate of the sender and on the congestion in the network. The optimal utilization for a NoC design with one AES core is approximately 8,5% for read and 2,8% for write (Obtained through the simulation). This is significantly less than the utilization in the custom design. Utilization of the mem controller is just possible if the NoC delivers the memory requests faster. This could be done either by increasing the bandwidth, or processing the data in a pipelined mode (see next section for more details).

6.3.3 Potential improvements for the Designs

In this section some improvements which do not have much area overhead, but could have a great impact on the performance, are described. Some of the improvements were mentioned already in the previous section, here they are described in more detail.

The performance could be improved not only in the NoC design. During the tests some new ideas were developed as well for the custom design:

- Since the timing of the AES core and memory and bus controller is higher, it could be possible to design the custom design to be able run

with higher clock speed. This would lead to improvement, based on the overall speed. The maximum could be around 210 MHz, this would be an improvement by a *factor* of 1,3.

- The custom design does not have pipeline steps in the design for the design with two AES cores. The inserting of the pipeline step could save 20 out 60 of cycles in every block processed by the design and so could potentially decrease the encryption time by a factor 0.66

The improvements for the NoC design are similar to the improvements for the custom design, some of them are completely different and dependent on the specifics of the NoC;

- The clock speed is a big issue in the NoC design, it is 1,5 times worse than the timing in the custom design. If the timing of NoC design could be increased, the performance would be significant better. If the timing could be increased around 200 MHz, what is potentially possible, than the improvement would be doubled compared to the current design.
- The data is first read in the Mem Ctrl. NWIF and then send to the network, it is possible to send first flits already when the first 32 bits are read, this would save 12 cycles out of 144 cycles. That would cause an improvement by a factor of 0,91. The area overhead would be small for realization.
- NoC specific improvement would be the size of the buffers, which are located in each router. This is very crucial for the performance, since if the buffer is small and all flits are spread over the network, the network become potential more congestion. Some cores would be blocked

as a consequence. The optimal size of buffers is dependent on the rate, the size of the packets and on the area requirements. So for this optimization it is necessary to constrict the set of problems which will use the NoC for communication. Furthermore the area overhead caused by additional buffers has to be considered in the decision. This problem may be not solvable for all problems, so each subset of problems needs a separate simulation and estimation.

- Increase the link width from 8 bits to 32 bits. This would reduce the number of flits needed by a *factor* of 4 and therefore significantly reduce communications overhead.

6.4 Conclusion

The results, described in this chapter, are very positive. The performance in the custom AES design is better than the results of the NoC design, as expected originally. The deterioration of the performance for the NoC design compared with the custom design is linear and has a factor 3,5-4. The benchmarks with a different hop amount show very good performance, the time overhead is very small.

Further this chapter described some design improvement opportunities, which could boost the performance considerably. Three modifications, which would bring the most improvements are: the timing needs to be optimized, increasing the link width to reduce the number of flits needed for messages, and the buffer size needs to be estimated for the similar subset of the problems.

Chapter 7

Conclusions and Future Work

7.1 Summary and Conclusions

Module-based dynamic reconfiguration of FPGAs enables the virtualization and multitasking of todays devices to be enhanced. It also provides greater flexibility by allowing user-dependent loading of modules when they are needed.

This thesis discussed and identified a feasible NoC approach to support general and simplified methodologies for implementing applications that use module-based dynamic reconfiguration. The novel approach of using fixed FPGA pages to virtualize and support multitasking was proposed and initially explored by *Shannon Koh* in the PhD Thesis: *Generating the Communications Infrastructure for Module-based Dynamic Reconfiguration of FPGAs* [16]. The idea of using fixed FPGA pages and supporting fast general module-based dynamic reconfiguration for those pages enables new opportunities for using FPGA devices. The reconfiguration ability of the device

becomes highly dynamic, since it supports fixed, general communication interfaces and communication methods. The only part which needs to be designed for an application is the Network Interface core (NWIF) which matches given core interfaces to the NoC interface.

To implement such an infrastructure, the methodology uses a fixed module slot layout on the FPGA with an NoC infrastructure providing inter-module and off-chip communication. The layout takes advantage of the two-dimensional paged reconfiguration architecture of the Virtex-4 FPGA family by allowing independent reconfiguration of each slot in each page.

This thesis proposed requirements for the paged FPGA design which are necessary for partial dynamic reconfiguration support. The main requirement is a general communication method, which supports inter-page communication and off-chip communication. The best technique to support it, is NoC with fixed, general infrastructure. As a part of the communication method a general interface is needed for detaching and attaching a reconfigured page to the communication method. A general interface needs to be implemented by both sides: the NoC and the dynamically reconfigurable part. This makes it possible to reprogram the pages while the device is running without any adaptation of the communication interface.

The thesis also presents guidelines for designing a good NWIF with a rough structure with a standardized set of packet formats. The main idea here is to decouple the control logic and allocate it directly to the NWIF of a particular core/page. The design of a specific NWIF is supposed to consider just the functionality of a given core. This simplifies design complexity and reduces potential pitfalls.

The NoC HERMES used in this work is developed by the group GAPH from Catholic University of Rio Grande do Sul, Brasil [20] and proposed by Moares et al. in [22]. It offers a network which has a low area overhead and parametrized implementation. Further, it offers a generic interface which supports partial dynamic reconfiguration and protects the network during reconfiguration by insulating it from unwanted data.

Two designs were proposed: one, as a reference, of a conventional custom design, and a second NoC-based implementation using the guidelines. The custom design was implemented first and afterwards a partitioning process was applied to the design to create a new design using and NoC as the central communication method for inter-page and off-chip communication.

An experimental framework was developed and the custom and NoC designs were assessed using various benchmarks. The results show that such an implementation is feasible. Although the performance of the NoC design was clearly worse compared to the custom, reference design, the benefits of scalability, increased parallelism and a simplified partial reconfiguration process are of great values to future applications.

The implementation process and evaluation identified some improvement potential for both designs, so they were also stated in this work.

7.2 Future Work

Several immediate directions for further investigation can be identified from the methodology proposed in this thesis and from the results that were obtained.

This thesis investigates NoC support for dynamic pages, where a fixed application with fixed interfaces and needs were chosen for implementation. This limits the set of problems to a small subset. It would be desirable to state a general set of problems, which need to be supported by the NoC and their interfaces. This would allow designers to choose good parameters for NoC implementation, such as bandwidth and buffer size, in order to reach an optimal support for a wide set of problems.

The implementation in this thesis does not have any constraints regarding allocation to lower the complexity. The next step would be to implement a general mesh structure over the entire device and to constrain the position of each router with a general interface for dynamic reconfiguration.

The assumption was made in this work to use a single clock speed for all components as limited by the slowest part. Each page needs to have the capability to run with a distinct clock speed, this also affects the router attached to a specific page. The behaviour of the NoC needs to be investigated by such an scenario, despite the asynchronous handshake protocol of the NoC.

The guidelines were proposed for creating a NWIF between the NoC interface and a core. These guideline are the first step for potential automation of creating such NWIFs. This problem needs further investigation.

The NoC represented in this work uses simple components for easier investigation of a general problem, but the possibilities for improvement are manifold. Other routing algorithm exists which provide better performance. The arbitration could implement Quality of Service which would be beneficial for many applications.

The timing of the design needs to be improved. Relevant details and potential improvements are stated in Chapter 6.

All future work stated above potentially increases the area of the implementation. This point also needs to be investigated, since the area of a page is limited.

Bibliography

- [1] Ahmadinia, A., Bobda, C., Ding, J., Majer, M. and Teich, J. A practical approach for circuit routing on dynamic reconfigurable devices. In *International Workshop on Rapid System Prototyping*, pages 84–90, Montréal, Canada, 2005.
- [2] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, 2002.
- [3] Bobda, C., Ahmadinia, A., Majer, M., Teich, J., Fekete, S. and Veen, J.v.d. DyNoC: A dynamic infrastructure for communication in dynamically reconfigurable devices. In *International Conference on Field Programmable Logic and Applications*, pages 153–158, Tampere, Finland, 2005.
- [4] Bobda, C., Majer, M., Koch, D., Ahmadinia, A. and Teich, J. A dynamic NoC approach for communication in reconfigurable devices. In *International Conference on Field-Programmable Logic and Applications*, pages 1032–1036, Antwerp, Belgium, 2004.
- [5] Chan, J. and Parameswaran, S. NoCOUT : NoC topology generation with mixed packet-switched and point-to-point networks. In *Asia*

and South Pacific Design Automation Conference, COEX, Seoul, Korea, 2008.

- [6] Chaouat, L., Garin, S., Vachoux, A. and Mlynek, D. Rapid prototyping of hardware systems via model reuse. In *IEEE International Workshop on Rapid System Prototyping*, pages 150–156, Chapel Hill, North Carolina, USA, 1997.
- [7] J. Daemen and V. Rijmen. Aes proposal: Rijndael. 1999.
- [8] William J. Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 684–689, New York, NY, USA, 2001. ACM.
- [9] Elgindy, H., Schröder, H., Spray, A., Somani, A.K. and Schmeck, H. RMB — a reconfigurable multiple bus network. In *International Symposium on High-Performance Computer Architecture*, pages 108–117, San Jose, CA, USA, 1996. IEEE.
- [10] Hübner, M., Becker, T. and Becker, J. Real-Time LUT-Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration. In *Symposium on Integrated Circuits and Systems Design*, pages 28–32, Lafayette, Los Angeles, USA, 2004.
- [11] Hübner, M., Schuck, C., Kühnle, M. and Becker, J. New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits . In *IEEE Computer Society Annual Symposium on VLSI: Emerging VLSI Technologies and Architectures*, pages 97–102, Karlsruhe, Germany, 2006.

- [12] IEEE. IEEE Standard VHDL Language Reference Manual. *IEEE Standard P1076 2004-10*, 2004.
- [13] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Standard 1364 -2005*, 2006.
- [14] Janac, G., Poltronetti, T., Herbert, A. and RuDusky, D. IP supply chain-the design reuse paradigm comes of age. *Integrated System Design*, 13(141):66–70, 2001.
- [15] Koh, S. and Diessel, O. COMMA: a communications methodology for dynamic module reconfiguration in FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 273–274, Napa Valley, California, 2006.
- [16] Koh, Shannon. Generating the Communications Infrastructure for Module-based Dynamic Reconfiguration of FPGAs, PhD Thesis. *University of New South Wales*, 2008.
- [17] Jian Liang, Sriram Swaminathan, and Russell Tessier. aSOC: A Scalable, Single-Chip Communications Architecture. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:37, 2000.
- [18] Majer, M., Teich, J., Ahmadinia, A. and Bobda, C. The Erlangen Slot Machine: A dynamically reconfigurable FPGA-based computer. *The Journal of VLSI Signal Processing*, 47(1), 2007.
- [19] Théodore Marescaux, Andrei Bartic, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Interconnection networks enable fine-grain dynamic multi-tasking on fpgas. In *FPL '02: Proceedings of the Reconfig-*

- urable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 795–805, London, UK, 2002. Springer-Verlag.
- [20] Aline Mello, Leandro Moeller, Ney Calazans, and Fernando Moraes. Multinoc: A multiprocessing system enabled by a network on chip. *Design, Automation and Test in Europe Conference and Exhibition*, 3:234–239, 2005.
 - [21] Giovanni De Micheli and Luca Benini. Powering networks on chips: Energy-efficient and reliable interconnect design for socs. *System Synthesis, International Symposium on*, 0:33–38, 2001.
 - [22] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integr. VLSI J.*, 38(1):69–93, 2004.
 - [23] E. Rijpkema, K. G. W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10350, Washington, DC, USA, 2003. IEEE Computer Society.
 - [24] Edwin Rijpkema, Kees Goossens, and Paul Wielage. A router architecture for networks on silicon. In *In Proceedings of Progress 2001, 2nd Workshop on Embedded Systems*, pages 181–188, 2001.

- [25] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in virtex fpgas. *Computers and Digital Techniques, IEE Proceedings-*, 153(3):157–164, 2006.
- [26] Ullmann, M., Hübner, M., Grimm, B. and Becker, J. On-demand FPGA run-time system for dynamical reconfiguration with adaptive priorities. In *International Conference on Field Programmable Logic and Applications*, pages 454–463, Leuven, Belgium, 2004.
- [27] Mário P. Véstias and Horácio C. Neto. Area and performance optimization of a generic network-on-chip architecture. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 68–73, New York, NY, USA, 2006. ACM.
- [28] Villasenor, J., Jones, C. and Schoner, B. Video communications using rapidly reconfigurable hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(6):565–567, 1995.
- [29] Xilinx. Two flows for partial reconfiguration: Module based or difference based. *Xilinx Application Note 290*, 2003.
- [30] Xilinx. Virtex-4 family overview. *Datasheet DS112*, 2004.
- [31] Xilinx. Early access partial reconfiguration user guide. *User Guide UG208*, 2007.
- [32] Xilinx. Virtex-5 family overview: LX, LXT and SXT platforms. *Xilinx Datasheet DS100*, 2007.
- [33] Xilinx. Virtex-6 family overview: LX, FXT and SXT platforms. *Xilinx Datasheet DS150*, 2009.