MICRONETWORK BASED
SYSTEM-ON-FPGA (SoFPGA) ARCHITECTURE

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

ABDUL-NASSER AL-ARAJE, M.S.

* * * * *
The Ohio State University

2005

Dissertation Committee:

Professor Joanne E. DeGroat, Advisor
Professor Mohammed Ismail
Professor Chang-Gun Lee

Approved by

Graduate Program in Electrical & Computer Engineering
ABSTRACT

In today’s world of advanced technology, numerous applications are computational intensive. This led to the development of new System-on-Chip (SoC) design techniques to allow for integration and reuse of Intellectual Property, *IP Cores*, under time-to-market pressure. A wide range of these newly emerging design platforms is now drifting towards highly integrated SoC designs with many on-chip processing resources like processors, DSPs, and memory. Using this technique, designers can build SoC by integrating dozens of IP cores. As the number of IP cores increases, the on-chip communication and physical interconnections become a bottleneck.

Originally, SoC was built based on a Bus-Centered approach. The Bus-Centered approach is a shared-medium architecture in which all IP cores share the same transmission medium. The Bus architecture is widely used in current SoC implementation and will continue to be the right design platform for small designs that integrate a few IP cores. However, for designs targeting FPGA with a large number of IP cores, we will propose a Micronetwork based System-on-FPGA (SoFPGA) architecture.
The problems with a SoC bus-centered approach using FPGAs grow as the size of the FPGAs and the complexity of the associated IP grows. The bus does not scale with the system size as the bandwidth is shared by all the components attached to it. Furthermore, smaller technologies are more subject to random errors and hence verification of physical faults will become increasingly more difficult. Also, as the number of IP cores increases, the capacitive load increases and the electrical performance degrades. Moreover, Synchronizing SoC with a single clock source and eliminating clock skew becomes extremely difficult or even impossible.

In this work, we will present a router architecture to be used as the basic building block of low overhead cost SoFPGA. This router is implemented as a VHDL model. We will address the interconnecting issues in SoFPGA design methodology built in a single FPGA device. Mainly, we will consider the problem of achieving efficient NoFPGA (Network-on-FPGA) performance through investigating the best topology. The hardware overhead induced by the 2D Mesh NoFPGA will be compared to the hardware overhead of the 2D Torus.
ACKNOWLEDGMENTS

This Ph.D. dissertation project was carried out at the Department of Electrical and Computer Engineering at the Ohio State University.

I would like to thank my advisor Dr. Joanne DeGroat for her continuous support and constructive feedback. All the things I learned from her especially her commitment to excellence in research and teaching, will be very useful for the rest of my life. I would like also to thank the committee members Dr. Mohammed Ismail, Dr. Chang-Gun Lee, and Dr. Stephen Milne for their help and continuous productive support. Their invaluable suggestions were crucial in improving the clarity and readability of this dissertation.

I am indebted to my parents for their support and prayers throughout my life. I am grateful to my wife Sahar for patiently facing the rigors of life for the past few years. This dissertation would not have been possible without her constant love and support. My brothers and sisters provided much needed support and prayers. Finally, I would like to thank my lovely daughter Lana for bringing so much joy into my life.
VITA

July 1, 1966 .............................................. Born, Gaza Strip, Palestine

1990 ....................................................... B.S. Electrical Engineering
                                          Irbid, Jordan

1996 ....................................................... M.S. Electrical Engineering
                                          Columbus, Ohio

1997 ...................................................... Electrical Design Engineer
                                          Lucent Technologies

2001 ...................................................... Chief Hardware Engineer
                                          vLogix, Loveland, Ohio

FIELDS OF STUDY

Major Field:   Electrical Engineering

Specialty:     Computer Engineering
# TABLE OF CONTENTS

- **ACKNOWLEDGMENTS** .................................................................................. iv
- **VITA** ........................................................................................................... v
- **LIST OF TABLES** ......................................................................................... x
- **LIST OF FIGURES** ....................................................................................... xii

## CHAPTER 1 ....................................................................................................... 1
  1.1 Problem Formulation................................................................................... 2
  1.2 Statement of Work....................................................................................... 4
  1.3 Dissertation Contribution ........................................................................... 5

## CHAPTER 2 ....................................................................................................... 8
  2.1 SRAM-FPGA .............................................................................................. 8
  2.2 Re-configurable Hardware ......................................................................... 10
  2.3 Previous work on Re-configurable Hardware ............................................. 14
  2.4 Network-on-Chip Concept ......................................................................... 19
  2.5 Bus versus NoC Model ............................................................................... 20

## CHAPTER 3 ..................................................................................................... 25
  3.1 Xilinx Virtex-II Pro Architecture .................................................................. 26
  3.2 Vertix-II Pro Embedded IP Core ............................................................... 28
  3.3 Looking into future of Platform FPGAs ..................................................... 32

## CHAPTER 4 ..................................................................................................... 33
LIST OF TABLES

Table 6.1 16-bit FIFO Implementation Cost ................................................................. 68
Table 6.2 32-bit FIFO Implementation Cost ................................................................. 69
Table 6.3 64-bit FIFO Implementation Cost ................................................................. 70
Table 6.4 16-Bit EBR FIFO Cost .............................................................................. 74
Table 6.5 32-Bit EBR FIFO Cost .............................................................................. 75
Table 6.6 64-Bit EBR FIFO Cost .............................................................................. 76
Table 6.7 16-Bit Torus IPRouter PFU FIFO Cost ....................................................... 77
Table 6.8 32-Bit Torus IPRouter PFU FIFO Cost ....................................................... 78
Table 6.9 64-Bit Torus IPRouter PFU FIFO Cost ....................................................... 79
Table 6.10 16-Bit EBR Based Torus IP Router Cost .................................................... 79
Table 6.11 32-Bit EBR Based Torus IP Router Cost .................................................... 80
Table 6.12 64-Bit EBR Based Torus IP Router Cost .................................................... 81
Table 6.13 16-Bit PFU Based Mesh IP Router Cost ..................................................... 82
Table 6.14 32-Bit PFU Based Mesh IP Router Cost ..................................................... 83
Table 6.15 64-Bit PFU Based Mesh IP Router Cost ..................................................... 84
Table 6.16 16-Bit EBR Based Mesh IP Router Cost ..................................................... 85
Table 6.17 32-Bit EBR Based Mesh IP Router Cost ..................................................... 86
Table 6.18 64-Bit EBR Based Mesh IP Router Cost ..................................................... 87
Table 6.19 Cost analysis of opencores IP Cores .......................................................... 90
Table 7.1 Torus Maximum Operating Frequency ....................................................... 93
Table 7.2 Mesh Maximum Operating Frequency ...................................................... 94
Table 7.3 Torus Link Bandwidth .............................................................................. 95
Table 7.4 Mesh Link Bandwidth .............................................................................. 95
LIST OF FIGURES

Figure 3.1 Virtex-II Pro Layout ................................................................. 30
Figure 4.1 FPGA Floor Plan Partitioning .................................................. 34
Figure 4.2 NoC Topology: Mesh and Torus ............................................. 35
Figure 4.3 NoFPGA TAB ...................................................................... 36
Figure 4.4 Packet Format (wormhole Routing) ........................................ 38
Figure 4.5 X-Y routing from source to destination .................................... 39
Figure 4.6 Asynchronous Communication (Handshake Signals) .............. 41
Figure 4.7 Header Routing Information .................................................. 43
Figure 5.1 Block Diagram of NoFPGA IP router .................................... 46
Figure 5.2 Mesh IP Router block diagram ............................................. 47
Figure 5.3 Mesh IP Router Entity .......................................................... 49
Figure 5.4 Detailed Mesh Router Block Diagram ................................... 51
Figure 5.5 Mesh Edge Node (Horizontal and Vertical) ............................ 54
Figure 5.6 Block Diagram of NoFPGA Torus IP Router ......................... 55
Figure 5.7 Torus IP Router ................................................................. 56
Figure 5.8 Torus IP Router Entity ........................................................ 58
Figure 5.9 Detailed Torus Router Block Diagram ................................... 60
Figure 6.1 FIFO BLOCK DIAGRAM .................................................... 67

xii
Figure 6.2 16-bit FIFO cost ................................................................. 71
Figure 6.3 32-bit FIFO cost ............................................................... 72
Figure 6.4 64-bit FIFO cost ............................................................... 72
Figure 6.5 FIFO PFU Cost ................................................................. 73
Figure 6.6 Torus and Mesh IP Router Cost with EBR FIFO ................. 88
Figure 6.7 Torus and Mesh IP Router Cost with PFU FIFO .................. 89
Figure 7.1 Communication Link Bandwidth ....................................... 96
Figure 7.2 Architecture Total Bandwidth .......................................... 99
Figure 8.1 GbEOS Block Diagram .................................................... 104
Figure 8.2 Mapping of GbEOS on 3 x 3 Torus NoFPGA ..................... 105
CHAPTER 1

INTRODUCTION

In today’s world of advanced technology, numerous applications are computational intensive. This created an opportunity for the development of new System-on-Chip (SoC) design techniques to allow easy IP cores (Intellectual Property cores) re-use and integration under time-to-market pressure. A wide range of these newly emerging design platforms is now drifting towards highly integrated System-on-Chip designs with many on-chip processing resources like processors, DSPs, and memory. Using this technique, designers can build System-on-Chip (SoC) by integrating dozens of IP cores. As the number of IP cores increases, the on-chip communication and physical interconnections become a bottleneck.

Originally, SoC was built based on a Bus-Centered approach. The Bus-Centered approach is a shared-medium architecture in which all IP cores share the same transmission medium. Since there will be multi-master of the shared bus, a bus arbitration mechanism is necessary. When one Bus-master needs to communicate across the bus, it first requests to gain the ownership of the bus and subsequently
releases the bus when it is done. The Bus architecture is widely used in current System-on-Chip (SoC) implementation and will continue to be the right design platform for small designs that integrate a few IP cores. As the number of IP cores increases, the Bus-Centered approach becomes an inappropriate design choice because of its limitations as a shared medium. First, the bandwidth of the bus is shared among all of the attached IP cores. Second, as the number of IP cores attached to the centered bus grows, the capacitive load will increase which will result in a performance degradation.

Due to the Bus-Centered approach limitations as an interconnecting medium for large SoC, there is a trend towards the use of packet-switched based interconnecting network. The NoC based architecture has the advantages of scalability, reusability, and parallelism, these are the key factors for implementing SoC. We will propose a NoFPGA based SoC design with FPGA as a target design.

1.1 Problem Formulation

The applications utilizing FPGA (Field Programmable Gate Array) as a design medium are predominant. Current FPGAs have a largely homogeneous architecture, i.e. a homogeneous fabric of Programmable Logic Cells (PLCs) with regularly distributed memory blocks and other specialized logic (e.g. multiplier blocks). This homogeneous architecture is maintained as the devices scale in size. FPGA architecture has evolved from just a glue logic device to more of a logic platform device with many hard and soft building blocks such as DSP block, a memory block,
and an embedded processor. As VLSI feature size shrinks, FPGA devices get larger and IP implementations that can be mapped to an FPGA become more complex.

The problems with SoC bus-centered approach using FPGAs (SoFPGA) grow as the size of the FPGAs and the complexity of the associated IP grow. The bus does not scale with the system size as the bandwidth is shared by all the components attached to it. Furthermore, smaller technologies are more subject to random errors and hence verification of physical faults will become increasingly more difficult. Also, as the number of IP cores increases, the capacitive load increases and the electrical performance degrades with this growth. Moreover, the task of synchronizing SoC with a single clock source and eliminating clock skew becomes extremely hard or even impossible [6]. Current System-on-Chip (SoC) concepts can be used to resolve many of these issues.

1.1.1 Problem Statement

According to ITRS (International Technology Roadmap on Semiconductors), we are approaching a four billions transistors by the end of this decade. Hence, FPGA will be larger and can house a very complex high performance system. The bus approach of current platform FPGA will become the bottleneck to achieve a high performance system, a new mechanism of interconnecting the embedded IP cores is necessary, the packet-switched communication network (NoFPGA) probably will be an excellent choice to address the communication needs for future platform FPGA.
NoC concept is not new, it is based on borrowing interconnection network concept of high performance parallel computing and adopt it for SoC interconnection network with FPGA as our design medium. NoFPGA based design is a structural and a modular approach. This will shorten the design cycle phase and introduce the product faster to the market. Also, by de-coupling applications and system architectures, the design cycle is further shortened. Our most important goal is to design a highly scalable NoFPGA, this will enable a re-use of already available IP cores. If the design of the NoFPGA interface is properly implemented, it will make IP integration easier with little cost and effort.

The problem to be investigated is how to apply SoC concepts to large FPGA designs. Specifically, we will investigate the effectiveness of applying SoC NoFPGA to resolve issues associated with the bus-centered approach and other possible benefits. A large FPGA is architected as a NoFPGA of small FPGA fabric tiles with each FPGA tile representing an IP block. This architecture makes it easier to partition and isolate functional IP blocks, allowing fine-grain dynamic partial reconfiguration and making hardware multi-tasking a reality. Also, it will make it easier to manage and verify the design.

1.2 Statement of Work

The problem to be investigated is a new reconfigurable FPGA architecture composed of small FPGA fabric tiles interconnected with a NoFPGA. This research focuses on
the NoFPGA optimal topology and the various routing algorithms for interconnecting FPGA tiles. This communication layer will be implemented on Lattice Semiconductor’s new high performance FPGA called Lattice FPSC. The NoFPGA is a 2D mesh and uses wormhole routing. The hardware overhead induced by the 2D mesh NoFPGA will be compared to the hardware overhead of the 2D torus. Also, the maximum bandwidth between two adjacent routers of both topologies, i.e. 2D mesh and 2D torus, will be measured and compared. Moreover, we will address the area overhead of the network logic in each tile and the size of the FPGA tile will be estimated.

1.3 Dissertation Contribution

In this dissertation, a new communication architecture, the NoFPGA, for future SoFPGA has been presented. The currently used Bus-Centered approach becomes an inappropriate choice because of its limitation as a shared medium that restricts the scalability of the communication architecture. Also, long bus wires result in performance degradation due to increased capacitive load. The long wires also consume more power to drive all of IP Cores on the bus. The following are the main contribution of this dissertation:

1. A low cost scalable architecture implemented in VHDL as a re-usable soft core is presented. The communication interface has been designed to support the packet-switched scheme so that the bus communication
bottleneck is resolved. The router architecture results in a highly scalable NoFPGA that enables a re-use and easy integration of already available IP Cores.

2. FPGA is a 2D device in its nature. As a result, the most appropriate topology to apply is either a 2D Mesh or a 2D Torus. The two candidate topologies are studied and the cost associated with each topology is investigated.

3. This dissertation work investigates the best topologies for future SoFPGA. The cost synthesis results showed that the hardware overhead of 2D Torus experiences a 40% logic drop when compared to 2D Mesh. The drop in logic usage is due to the fact that Torus topology only routes packets in two directions while Mesh topology routes packets in four directions.

4. The buffer is the most highly expensive block of the router and it should be highly optimized by using available resources of FPGA, especially the Embedded Block RAM.

5. The cost synthesis of available opencores IP Cores showed that most of IP Cores fits between 18 Kgates and 35 Kgates. The Torus cost
represents a range of 4.2% to 17.5% of the total cost of the IP core. On the other hand, the Mesh cost represents a range of 7.1% to 26% of the total cost of IP Core.

6. The cost synthesis analysis of NoFPGA showed that the size of the Tile Architecture Block (TAB) could be chosen in the range of 40 to 50 K gates.

7. Simulation results showed that the Torus topology has a higher communication link bandwidth than the Mesh topology. The total bandwidth of NoFPGA depends on the communication link bandwidth and the total number of links of network topology.
CHAPTER 2

BACKGROUND AND PREVIOUS WORK

The applications utilizing FPGA (Field Programmable Gate Array) as a design medium are predominant. FPGAs have been used extensively not only in logic emulation but also in custom-computing machines. The re-programmability nature of SRAM-FPGA makes it the workhorse of many new reprogrammable applications. One such example is in multi-mode hardware applications [1] where an FPGA with a set of different configuration files stored in a ROM can be used to support different functionalities. Also, multiple-FPGA systems are used extensively in logic emulation. Multi-FPGA system was an area of research for so many years where the heart of research was mainly focusing on the best way of interconnecting multi-FPGAs.

2.1 SRAM-FPGA

FPGA-based re-programmable logic design became more attractive during the last decade, the use of FPGA in digital logic design is increasing rapidly. SRAM-FPGA’s are the most popular and becoming the workhorse of many re-programmable applications. SRAM-FPGAs’ re-programmability feature makes it more attractive
since it can be completely changed by the same electrical process. A microprocessor can be configured to run different applications, the configuration of SRAM-FPGA can be changed for bug fixes or even upgrades, this makes them an ideal prototyping medium.

FPGA can be used to implement operational algorithms for certain applications. This can offer extremely high performance compared to general-purpose microprocessor machines. This was the driving force for many research machines like Splash system that provides high performance on genetic string matching. The Splash system is almost 200 times faster than any other supercomputer [1].

SRAM-FPGAs are volatile and need reprogramming on system power up. Reprogramming is done through configuration files that are stored in a ROM. Control circuitry is used to automatically load the configuration files from the ROM. This in-circuit re-programmability feature makes them more flexible since their programming can be changed without any rewiring or re-fabrication. There is no need to continually reprogram the ROM on power-ups since it retains its content when the power is off.

The memory cells of SRAM-FPGA are scattered throughout the FPGA. The SRAM programming bit can be connected to a multiplexer to form a lookup table (LUT). A look-up table is a logic block that can implement any combinational boolean function of its input. The number of inputs classifies the look-up table; as the number of inputs
increase, so does the number of Boolean functions implemented. This, in turn, will affect the number of logic blocks necessary to implement a circuit.

The complexity of the logic block grows exponentially as the number of the inputs grows. However, it is more practical to group a set of logical blocks together and provide local routing to interconnect them [42]. The new logic block will have more functionality than the basic logic block. This new logic block is called a logic cluster [42]. We will utilize this concept to partition the floor plan of the FPGA device into a set of Tile Architecture Blocks (TAB), and the size of the TAB should be large enough to implement the corresponding IP Cores.

### 2.2 Re-configurable Hardware

Re-configurable Architectures are based on Field-Programmable Gate Arrays (FPGAs). FPGAs are structured as an array of programmable logic blocks embedded in a programmable routing structure. Both logic blocks and routing resources are programmable. A custom digital circuit can be mapped to the re-configurable logic blocks and interconnect them to form the intended circuit.

Current re-configurable processors are viewed as a combination of re-configurable processing unit (FPGA) interconnected with a general-purpose microprocessor. The re-configurable logic is used to accelerate algorithms by mapping the compute-
intensive portions of the algorithm to the re-configurable hardware. The behavior of the re-configurable processing unit is specified by its configuration. Two modes of re-configuration are available [3]:

- **Static or compile-time re-configuration (CTR):** The configuration of the re-configurable processing unit is done once and does not change during the execution of the task.

- **Dynamic or run-time reconfiguration (RTR):** The configuration of the re-configurable processing unit can change during run time. The re-configuration time has to be minimal to improve the potential performance of the re-configurable processors over general-purpose microprocessors. Recent research is focused on minimizing the reconfiguration time.

Re-configurable processors have gained increasing attention in the research community, and a large number of re-configurable architectures have been developed, some of them entering the commercial market. Re-configurable architectures can be classified based on several different parameters. The following subsections list the distinguishing architecture parameters that can be used to classify re-configurable architectures:
2.2.1 Granularity of the re-configurable processing unit

The granularity of the re-configurable processing unit defines the size of the smallest building block that is addressed by the mapping tools of the re-configurable unit. It can be either fine-grain or coarse-grain:

- Fine-grain blocks provide more flexibility in adapting hardware to the computation structure. On the other hand, they suffer from lower performance due to larger delay when constructing larger blocks from fine-grain blocks. An example of such architecture is PiCoGA [4].

- Coarse-grain blocks are intended for the implementation of word-length data path circuits. The coarse-grain data path, proposed by Galanis et al. [9], is an example of this type of architecture. Coarse-grain blocks are better suited to implement regular arithmetic operations, like DSP applications.

2.2.2 Microprocessor coupling

The coupling between the microprocessor and re-configurable processing unit determines the overhead in utilizing the re-configurable logic to speed up computations. The degree of coupling affects the re-configuration and the data access costs. The degree of coupling can be classified into three classes [5]:

- Loose board-level coupling: This is the lowest method of coupling, the re-configurable processing unit communicates with the microprocessor (host)
through an I/O interface, for example, PCI interface. Initial re-configurable architectures were constructed with this degree of coupling. The instruction set of the microprocessor is not extended.

- Loose chip-level coupling: The re-configurable processing unit has a direct communications port to the microprocessor (host) and acts as a coprocessor.
- Tight on-chip coupling: With this architecture, the re-configurable processing unit is integrated on-chip with the microprocessor (host) as a System-On-Chip configuration. This arrangement significantly reduces the communication overhead. Such architectures include PiCoGA [4].

The commercial re-configurable processors (for example, Altera Excalibur ARM and MIPS based, Xilinx Virtex-II Pro) are on the border between loose-chip level and tight on-chip coupling. There are no customized instructions for the re-configurable processing unit.

### 2.2.3 Instruction Set Extension (Extensible Processor)

The re-configurable processing logic is viewed as an extension to the host microprocessor. The microprocessor instruction set has a customized instruction set for the re-configurable logic. The microprocessor fetches, decodes, and issues customized instructions to the re-configurable processing logic to perform certain tasks, either to start the reconfiguration or to execute the re-configurable logic function.
2.2.4 Programming Model and Compiler

Programming model is defined as the process of code construction, automatic code generation, etc. The current programming model consists of separate tool flows for the hardware and software. This area of research has not received enough attention yet. A significant amount of research should address the compilation technology and code generation for re-configurable processor architecture [5]. The work done by [10] shows promising results to combine both hardware and software design path.

2.3 Previous work on Re-configurable Hardware

Lodi et al. [4] proposed a Pipelined Configurable Gate Array (PiCoGA) architecture to fit the re-configurable processing unit of a re-configurable processor. This architecture is a tightly coupled architecture and composed of an array of rows. Each row represents a pipeline stage and can process 32-bit operands. Each row is connected to the other rows via configurable interconnect channels. Also, each row connects to the host microprocessor register file with 6 32-bit busses, that span the whole array. The building block of each row is a re-configurable logic cell (RLC), 16 per each row.

The authors verified their design by prototyping a re-configurable processor in 0.18 um XiRisc (eXtended Instruction set Risc). XiRisc is an open core VLIW (Very Long Instruction Word) processor with 5 stages of pipeline. It is a Harvard architecture built with separate instruction and data cache. The XiRisc instruction set
has been updated to support PiCoGA operations. This architecture was tested on several signal-processing algorithms, including CRC, Median filter, motion estimate. Their results show a speedup factor of 4.3X to 13.5X ranges when compared to standard VLIW microprocessor. The most powerful gain was in power consumption reduction up to 93% due to less memory access when compared to standard VLIW microprocessor [4].

The new trends in this architecture are basically two. First, this architecture introduces the idea of re-configurable cache to improve re-configuration latency. Second, the pipeline nature of the re-configurable processing unit (PiCoGA) improves the throughput.

Cong et al. [6] addressed Instruction Set Extension automation for re-configurable architecture. The idea is to extend the base instruction set of a general-purpose microprocessor with a set of customized instructions supported by the specific re-configurable hardware resources. Their work is based on using a fully performance driven, automated compilation flow to generate application specific instructions. Their approach has three phases:

- Enumerate all candidate patterns
- Instruction set selection based on a cost function
- Mapping data flow into selected patterns.
This algorithm was implemented in C++. The authors used a re-configurable system (Altera’s Nios) to test their algorithm. Their results show a speedup over standard processor with no special instruction for the re-configurable logic. The use of execution time as a metric when selecting special instruction produces a reasonable speedup over non-special instruction set processor.

Kujuth et al. [7] proposed a new architecture of clustered programmable re-configurable processors. This architecture integrates multiple re-configurable processors and logic blocks onto a single chip. The communication among different clusters is done through shared memory. This architecture is very similar to a multi-processor architecture. Their idea was to achieve better performance when compared to a single cluster architecture. Their experimental work relied on SPICE simulation for power and latencies. On the other hand, the area estimate was captured using hand layout. Similar work was done by Soares et al. [15] to represent a coarse grain re-configurable multiprocessor.

Galanis et al. [9] proposed a coarse-grain re-configurable data path. This data path is tailored for computational intensive DSP kernel and is permanently implemented in hardware in ASIC technology. There is no need to configure the hardware block, which reduces the reconfiguration overhead. The key aspect of this research is the capability to map the intended application onto the data path. The coarse-grain component (CGC) is part of a general mixed granularity re-configurable
platform including microprocessor, fine-grain re-configurable blocks, and memory. CGC is an array of nodes containing multiplier, ALU, and steering logic, which allow easy realization of the desired application. This research presented an ASIC coarse-grain data path that reduces the re-configuration overhead and improves performance as a result. Moreover, the authors proposed a methodology for mapping the computational intensive part onto the CGC.

Kafafi et al. [11] compared using soft cores versus hard cores when implementing a re-configurable processing unit. Their results show that for a small amount of logic, a synthesizable soft core is more attractive since it is easier to integrate, and to make upgrades without a need for re-layout. The authors used standard synthesis tools, cell libraries, and design flow for their experimental work.

Austin et al. [13] proposed a system level simulator for CPU and memory architecture called the SimpleScalar simulation suite. It is written in C and provides a widely parameterized CPU model. It is open source and allows modification for academic purposes. This simulator is based on a 32-bit RISC architecture; the instructions are extended to 64 bit to allow experimental instruction set extensions. The parameter of the superscalar execution core includes a number of functional units (ALU, integer unit, floating-point unit, and multiplier). This simulator model allows performance evaluation of a broad spectrum of CPUs. An important feature of SimpleScalar is its extensibility, which allows the adaptation or extension of the
simulated processor with new customized instruction set. SimpleScalar has become popular among researchers and academia [13].

Taylor et al. [8] proposed the RAW microprocessor. It is composed of multiple identical tiles. Each tile incorporates an ALU and fine-grain re-configurable logic. The authors claimed that the small amount of re-configurable logic in each tile will be selected by software for the data path, supporting multi-granular operation. However, the mechanism of how the ALU cooperates with the re-configurable logic is not elaborated.

Shanon et al. [12] introduced a standard benchmark suite called the Re-configurable Architecture TEsting Suite (RATES). The driving force to develop this standard benchmark is the lack of commonly standard performance evaluation of re-configurable processors. The new trend in this benchmark is to be totally independent from the architecture and language. It is also applicable to either general-purpose or re-configurable processors, making the comparison among different architectures more meaningful. This paper addressed an important issue of standardizing benchmarks to evaluate different re-configurable processors. The contribution of this study comes from exploring the possibility of architecture independent re-configurable processors evaluation benchmarks.
Perri et al. [14] presented a SIMD 2-D convolver for fast FPGA-based image and video processor. The authors utilized the re-configurability nature of FPGA to introduce a new SIMD image and video processor. This processor can dynamically adapt itself to perform 2-D convolutions between variable bit resolution. The results showed efficient solution for image convolution when compared to traditional DSP.

Slade et al. [10] presented a new application framework for re-configurable processor design. The authors first discussed the current design tools and the overhead cost since they are not tailored for re-configurable processors. One of the limitations of the current design tools is being ASIC-oriented since they only focuses on hardware design and debug and not supporting software development. To make the design cycle more efficient, a combined design path is necessary to streamline the development of both hardware and software. As a result, a tighter integration of hardware and software component occurs.

2.4 Network-on-Chip Concept

The future System-on-Chip design methodology will heavily utilize the Network-on-Chip concept as the interconnection mechanism for integrating IP cores. According to ITRS, we are expecting a four billion transistor chip by the end of this decade, this technology scaling will allows hundreds of IP cores to be integrating on the same chip and NoC will be the only interconnection mechanism that can handle the communication requirements for future System-on-Chip. NoC concept is not new, it is based on borrowing interconnection network concept of high performance parallel
computing and adopt it for SoC interconnection network. NoC is an array of IP-Routers interconnecting an array of IP-Blocks. The IP-Router directs communications between the IP-Block and the neighboring nodes or between the neighboring nodes themselves. Each IP-Block will be encapsulated into a fixed communication layer, which is identical among all IP-Blocks within the NoC.

2.5 Bus versus NoC Model

The authors [71] studied and compared two interconnection architecture models for Systems-on-Chip, the Bus-Centered approach and the Network-on-Chip approach. Their work focused on mathematical analysis of the performance of busses versus NoC. The authors tried to answer the question of when NoC will become the preferred interconnecting architecture for Systems-on-Chip. Their work was divided into two parts. In the first one, the target system topologies were introduced, both the Bus model as well as the NoC model. The comparison modeling criteria such as silicon cost, operating frequency, and latency were discussed in the second part.

2.5.1 Bus-Model

The considered Bus model follows simple Bus-model architecture to interconnect Intellectual Properties (IP) Cores in a SoC with the following features:

- De-multiplexed 32-bit address/data buses.
- Multi-master capability.
- Bus Control Unit (BCU) for bus arbitration and decoding.
**Bus-Model Propagation Delay:**

The propagation delay of a Bus depends on the number of IP Cores attached to the wires as well as the wire length.

\[ T_{bus} = (N_c \cdot C_L + L_{w,bus} \cdot C_m) \cdot K_{tec} \]

Where:

- \( N_c \): the number of IP Cores interconnected to the wire.
- \( L_{w,bus} \): Bus wire length, which grows with the number of IP Cores in the system.
- \( C_L \): IP Core parasitic capacitance.
- \( C_m \): Wire parasitic capacitance (F/m)
- \( K_{tec} \): Technology parameter

**The zero-load latency model of Bus-Centered approach:**

\[ t_{msg, bus} = T_{arb} + T_{bus} \cdot m \]

Where:

- \( T_{arb} \): bus arbitration time; one bus cycle = \( T_{bus} \)
- \( m \): message length
- \( T_{bus} \): one bus cycle

### 2.5.2 NoC Model

The NoC modeled in this work was based on 2-D torus topology with wormwhole packet switching. Each packet consists of 32-bit data. The routing protocol is deterministic with X-Y dimension order.
**NoC-model Propagation Delay:**

Since each wire connects a driver to a single receiver, the propagation delay for NoC is:

\[
T_{noc} = (1 \cdot C_L + L_{w,noc} \cdot C_m) \cdot K_{tec}
\]

Where:

\(L_{w,noc}\): NoC wire length, which depends on network topology. For example, in 2-D grid, it is a constant value.

---

**The zero-load latency of NoC approach:**

\[
t_{msg,noc} = D \cdot (t_r + t_s + t_w) + \max(t_s, t_w) \cdot \left[ \frac{L}{W} \right]
\]

Where:

\(L\): represents message length

\(W\): represents channel width

\(D\): represents locality and defines the number of hops between sender and receiver, small \(D\) means great locality.

\(t_s, t_r\): represents routing time and switching time, \(t_s\) equals \(2 \cdot T_{noc}\), \(t_r\) equals \(T_{noc}\)

\(t_w\): represents inter-router delay, the propagation delay across the wires of external channels equal \(T_{noc}\)

\(L/W\): represents pipelined packet payload.

**The zero-load latency of NoC can be rewritten as follows:**

\[
t_{msg,noc} = 4 \cdot T_{noc} \cdot D + T_{noc} \cdot \left[ \frac{L}{W} \right]
\]
To accommodate for the overhead related to packing and unpacking of message, two more cycles are added:

\[ t_{msg, noc} = 2 \cdot T_{noc} + 4 \cdot T_{noc} \cdot D + T_{noc} \cdot \left[ \frac{L}{W} \right] \]

In case of non-zero load and assuming each node will send a message to all other nodes (\textit{worst case scenario}), the total number of messages is expressed by:

\[ N_{msg} = N_c (N_c - 1) \]

To represent a normal load, a more lighter workload where each IP Core sends message to a smaller number of IP Cores, a reduction factor is applied as follows:

\[ N_{msg} = \frac{N_c}{R_d} \left( \frac{N_c}{R_d} - 1 \right) \]

To re-write the total latency of both Bus and NoC models for a lighter workload:

\[ t_{msg, bus} = \frac{N_c}{R_d} \left( \frac{N_c}{R_d} - 1 \right) \cdot T_{bus} (1 + m) \]

\[ t_{msg, noc} = \frac{N_c}{R_d} \left( \frac{N_c}{R_d} - 1 \right) \cdot T_{noc} \cdot \left[ 2 + 4 \cdot D + \frac{L}{W} \right] \cdot \frac{1}{\rho_{noc}} \]

Where:

\( \rho_{noc} \): represents parallelism offered by NoC and depends on the number of IP cores as well as the locality:

\[ \rho_{noc} = \frac{2}{D} \cdot \frac{N_c}{R_d} \]
The authors mathematically compared both architecture models, their results showed:

- The total capacitance load of the Bus-Centered approach increases with the number of the interconnecting IP Cores, \( N_c \). On the other hand, the total capacitance of NoC approach is constant since each wire connects a driver to a single receiver.

- The die size effect on operating frequency for Bus-Centered approach is more than NoC approach since it is a factor of 1.414 for NoC compared to a factor of 2 for Bus model.

- The authors derived the allowed hardware overhead of NoC before it starts to degrade the clock cycle. This done by making:

\[ T_{bus} = T_{noc} \]

- For a short message and at a maximum workload, the NoC is effective with a great locality (D small) or when the system grows (\( N_c \) increases)

- If the message size increased, NoC takes advantage of its pipelined structure and its performance outperforms Bus-Centered approach for \( N_c \geq 16 \)

- NoC is becoming a preferred choice for Systems-on-Chip interconnection with more than two dozens of IP Cores and/or running applications with good locality.
CHAPTER 3

PLATFORM FPGAs

FPGAs have evolved from only a glue logic device to platform-based design medium. According to ITRS, we are approaching a four billion transistors chip by the end of this decade, this will allow building a very complex, high performance systems on FPGA. As the technology scaling continues, more and more logic will be available on chip, this creates a great avenue for SoFPGA that provides the platform for IP Cores re-use.

IP Core can be either a hardcore or softcore. For example, Xilinx Virtex-II Pro is incorporated with IBM PowerPC hardcore RISC processors [72]. On the other hand, soft IP Core represents a more flexible synthesizable core that can be used on as needed basis.
Many FPGA vendors provide families of platforms suitable for SoFPGA implementation. Example of available platform FPGA is Xilinx Virtex-II Pro and Lattice Semiconductor Lattice-SC FPGA.

### 3.1 Xilinx Virtex-II Pro Architecture

The Virtex-II Pro from Xilinx represents a high performance platform FPGA solution with up to four embedded 32-bit PowerPC RISC processors. The Virtex-II Pro is manufactured in 0.13 um CMOS using 90-nm transistors and nine-layer copper process. Virtex-II Pro families are equipped with various programmable architecture that makes them a key for high performance system design and re-use.

#### 3.1.1 Configurable Logic Block (CLB)

The basic building blocks of FPGAs are CLBs. The Virtex-II Pro is architected as a two-dimensional array of CLBs, each CLB includes four slices and two tri-state buffers. Each slice contains two function generators that is configured as a 4-input lookup table (LUT). Also, each slice contains two storage elements, arithmetic logic gates, large multiplexer, and a fast look-ahead carry-chain. The CLB contains the routing resources to connect its internal resources to a switch matrix in order to access the general routing resources.
3.1.2 **Block Select RAM**

The Virtex-II Pro has a large memory resource of 18 Kbit of synchronous dual-port RAM. The memory is programmable in various width and depth, it can be cascaded to implement a large embedded storage block.

3.1.3 **Multiplier**

There is a dedicated 18-bit x 18-bit two’s complement multiplier associated with each memory block to be used for operations on memory contents. The multiplier can also be used independent of the memory block to provide Read/Multiply/Accumulate operations for DSP applications.

3.1.4 **Digital Clock Manager (DCM)**

There are up to twelve DCM blocks to provide clock-de-skew necessary for high-speed design solutions. The DCM also provides phase-shifted versions of its output clocks. Also, the DCM provides frequency synthesis that permits accurate clock multiplication and division.

3.1.5 **Input/Output Block**

The input/output block supports a total of twenty-two single-ended I/O standards, and five differential signaling standards. The largest device has almost 1200 user I/O pins. The I/O block provides a maximum speed of 840 Mbps for low voltage differential signal (LVDS) mode. The I/O output block is characterized by a Digitally
Controlled Impedance that is necessary to provide on-chip differential termination for LVDS standards.

3.1.6 Routing

The array of building blocks (CLB, Block RAM, Multiplier, …) are embedded in a global routing structure, each of the building blocks has the same access to the global routing matrix, the global routing matrix is composed of a set of programmable short, medium, and long-range routes as well as switch boxes, the routing structure is symmetrical along both the horizontal and vertical directions. The embedded IP Cores have the same access to the global routing matrix as well.

3.2 Vertix-II Pro Embedded IP Core

The Virtex-II Pro family has two embedded hard IP Cores as part of its architecture. The Two IP Cores are the PowerPC 405 processor core from IBM and the RocketIO serial transceiver that is based on Mindspeeds’s SkyRail technology.

3.2.1 PowerPC 405

The Virtex-II Pro FPGAs provide up to four embedded 32-bit IBM PowerPC 405 RISC processors, each processor delivers over 420 Dhrystones MIPs at 300 Mhz. The main features of the PowerPC 405 are:
• 16KB instruction cache unit and 16KB data cache unit, the cache unit is two-way, set-associative.

• Embedded memory management unit (MMU) with translation look-aside buffers, the MMU supports multiple page sizes from 1KB to 16MB.

• Five-Stage data path pipeline

• Thirty-two 32-bit general-purpose registers (GPRs)

• Integer multiply/divide unit for fast integer arithmetic, the PowerPC is not equipped with floating point unit.

• Dedicated on-chip memory interfaces

• Timer functionality

• Debug support

• Advanced power management support.

The PowerPC 405 only takes up as little as 2% of the total die area of the Virtex-II Pro XC2VP50. The PowerPC is also integrated with the surrounding CLBs, Block RAM, and embedded in the general routing resources of longlines and hex lines that travel across the core both vertically and horizontally allowing the signals to route through the processor block. Figure 3.1 shows the layout of the Virtex-II Pro.
The on-chip memory controller (OCM) represents the dedicated interface between the block RAM and the PowerPC Core.

The PowerPC implements the IBM CoreConnect bus architecture that consists of three separate buses, the Processor Local Bus (PLB), the Device Control Register (DCR), and the On-chip Peripheral Bus (OPB). The PLB connects the core to high-
speed system resources, it provides 32-bit address/64-bit data bus. The DCR bus is used to manage the status and control registers of the peripherals external to the PowerPC.

Figure 3.1 shows that the PowerPC is embedded in the logic fabric and does not interrupt the routing structure. As a result, the process of automatic routing during the placement and routing design phase is not impacted.

3.2.2 RocketIO Multi-Gigabit Transceiver (MGT)

The Virtex-II Pro is equipped with a hard core RocketIO Multi-Gigabit Serial Transceiver (MGT), the MGT is capable of operating at a data rate in the range of 622 Mbps to 3.125 Gbps. The Virtex-II Pro family can be equipped with up to 24 transceivers. The RocketIO MGT has the following features.

- Full-duplex serial transceiver (SERDES) operating at a rate from 622 Mbps to 3.125 Gbps.
- Supports Fiber channel, Gigabit Ethernet, and Infiniband-compatible transceivers.
- Monolithic clock and data recover (CDR)
- 8-bit, 16-bit, and 32-bit selectable internal interface to FPGA.
- 50/70 Ohm on-chip selectable transmit and receiver terminations
- Five level of output differential voltage
- 8B/10B Encoder/Decoder
• Rate matching via insertion/deletion characters.

The RocketIO MGT consists of the physical media attachment (PMA) and physical coding sublayer (PCS). The PMA contains the 3.125 Gbps serializer/deserializer (SERDES), the Tx/Rx buffer, the clock generator, and the clock recovery circuitry.

The DCM and MGT are placed on the top and bottom of each block RAM as shown in figure 3.1, this layout will place no limitations on the routing process during place and route design phase. The number of DCMs and RocketIO MGTs is twice the number of block RAM.

3.3 Looking into future of Platform FPGAs

According to ITRS, we are approaching a four billions transistors by the end of this decade. Hence, FPGA will be larger and can house a very complex high performance system. The bus approach of current platform FPGA will become the bottleneck to achieve a high performance system, a new mechanism of interconnecting the embedded IP cores is necessary, the packet-switched communication network (NoFPGA) probably will be the only choice to address the communication needs for future platform FPGA. Also, analog building blocks will appear as a basic building block such as analog-to-digital and digital to analog converters.
NoFPGA (Network-on-FPGA) ARCHITECTURAL MODEL

The NoC is characterized by a set of parameters as follows: topology, routing algorithm, packet format, and packet switching. The NoC topology defines the arrangements of the NoC nodes while routing algorithm defines the path the message will take to go from sender node to receiver node. On the other hand, packet format specifies the message structure as two parts: header and payload. Packet switching describes how the message payload will follow the message header when it is selected to switch the message from input port to output port.

4.1 NoFPGA Architecture

A large FPGA is architected as a SoC NoFPGA of small FPGA fabric tiles with each FPGA tile representing an IP core and IP router. Figure 4.1 shows how FPGA floor plan can be architected of small fabric tile called FPGA Tile Architecture Block (TAB). This architecture makes it easier to partition and isolate functional IP blocks, allowing fine-grain dynamic partial reconfiguration and making hardware multi-
tasking a reality. Also, it will make it easier to manage and verify the design since the IP core development can run independent of the NoFPGA implementation which will shorten the development cycle and reduce the time-to-market (TTM) pressure.

Each TAB represents a NoFPGA node that communicates with its neighbors over the NoFPGA communication links. The IP router is responsible of routing packets to NoFPGA neighbors, this routing logic should only represent a small amount of logic gates, since this routing logic is considered as an overhead, it has to be optimized for area, the architecture of IP router affects this logic overhead and it is the scope of this research.
4.2 NoFPGA topology

Since an FPGA is 2-dimensional in its nature, the most appropriate topology is a 2D mesh. Figure 4.2 shows 2D mesh topology where the router has the capability to route in all four directions: East, West, North, and South. To simplify the complexity of the router, it is possible to use a 2D torus topology, as shown in figure 4.2. The figure shows a 2D mesh folding along the horizontal and the vertical directions. This makes the routing easier and only in two directions: East and South since any destination can be reached by going only in the two directions. Figure 4.5 shows the routing path between source (S) and destination (D): First you go east, then south where you reach your destination (D).

![Mesh and Torus](image)

*Figure 4.2 NoC Topology: Mesh and Torus*
4.2.1 NoFPGA Node

Each NoFPGA TAB node is composed of two functional parts, the first one is the IP core itself which can be implemented independent of the NoFPGA communication layer. Secondly is the IP router which has a set communication links to connect with its neighbors as well as the IP core. Figure 4.3 shows a block diagram of the NoFPGA node and how each node has IP core and IP router.
4.3 NoFPGA Packet Switching Algorithm

The Wormhole switching algorithm [40] that will be used for the proposed NoFPGA is a packet switching approach that has been used extensively in high-performance parallel computer networks. In wormhole switching algorithm, a router can forward a packet as soon as it receives the packet header. This simplifies the router design since it does not have to buffer the entire message and hence reducing the hardware overhead. A message is broken into flits (flow control unit) which are the smallest units over which the flow control is done. The message packets are pipelined through the network, each message is composed of a header flit which contains routing information and data flits which contains the message itself, figure 4.4 shows the packet format. Once the message header has been selected to be switched from input port to an output port, the message payload follows the message header in a pipelined fashion. However, if the required output port is not available, the IP router must buffer the header as well as the rest of the payload flits in a local buffer, this is similar to store and forward packet switching but the message is not stored in the local IP Core memory. The Buffer of the IP router is an expensive building block and needs to be highly optimized for logic usage.
4.4 NoFPGA Packet Routing Algorithm

Routing packets determines the path the message choose from source to destination, in this architecture we will use X-Y routing algorithm which is deterministic and source based meaning the source will determine the path the packet will follow to reach the destination. The packets are first routed along the X direction then along the Y. The routing information is part of the message header flit which have the number of channels that must be traversed respectively in the X and Y directions, the X value is decremented each time a router is traversed. When it reaches zero, the Y direction routing starts. Also, the Y value is decremented each time a
router is traversed. When Y value reaches zero, the message is delivered to its destination. Figure 4.5 shows the X-Y routing mechanism on 2-D mesh NoFPGA, for example, the sender node [S] wants to send a packet to Destination node [D], since X-Y routing is source based, the sender generates the routing header information, the routing header information has:

- The number of hops to traverse in the X direction, xd_addr = 2
- The X directional bit, if moving in positive x direction, x_flag = 1, else x_flag = 0, in this example x_flag = 1
- The number of hops to traverse in the Y direction, yd_addr = 2
- The Y directional bit, if moving in positive Y direction, y_flag = 1, else y_flag = 0, in this example, y_flag = 0.

![Figure 4.5 X-Y routing from source to destination](image)
4.5 NoFPGA Communication Link

The data path width is an important factor in determining the NoFPGA throughput and has to be chosen carefully; it cannot be set too narrow causing a bottleneck nor cannot be set too wide.

The NoFPGA links are asynchronous structures. Each NoFPGA node is connected to its four neighbors and local IP core through point-to-point communication links, the communication link is a bi-directional link, one in each direction. The communication link has two parts:

- Data part, the channel width is the number of parallel bits that can be transmitted across the communication link, which will be translated in the number of wires connecting neighboring nodes. We will study the impact of three data link width, 16-, 32- and 64-bit data width. The message will be segmented into 16-, 32-bit and 64-bit flits.

- Handshake signals for flow control and communication between neighboring nodes. The handshake signals are used to validate data being transmitted and acknowledging the receipt of data. Signals RDY_In and Latch_In functioning as input handshake signals and RDY_Out and Latch_Out as output handshake signals. Figure 4.6 shows the handshaking communication signals between neighboring nodes, Node A and Node B.
When Node A wants to send message to Node B. If Node B RDY_In signal is asserted, Node A asserts Latch_In, causing the header flit to be latched into the Node B dimension buffer (FB).

Node B de-asserts RDY_In if the buffer is Full and start processing the header packet information.

When the message is selected for an output port, Node B sets output handshake signals, Latch_Out, and RDY_Out.

The message is then passed in a pipeline fashion, with RDY_In and Latch_In functioning as input handshake signals and RDY_Out and Latch_Out as output handshake signals.

**Figure 4.6 Asynchronous Communication (Handshake Signals)**
4.6 Communication Protocol

The communication protocol is based on a request-ack cycle; each packet is transmitted after a negotiation session between handshake signals. Each IP core in the NoFPGA can be either a sender or a receiver. When a message enters a router, it can either be forwarded along the current direction, sent orthogonally or delivered to the local IP-block.

The functionality of a node can be viewed as multiplexing and de-multiplexing five different data flows: The North-direction, East-direction, South-direction, West-direction, and local IP-direction. Each direction has a FIFO buffer. The IP interface to the NoFPGA translates the interface signals of the functional IP core to the NoFPGA packet format. All IP cores are implemented as independent IP cores.

The IP core will work independently from the NoFPGA. The interface of the IP core to the IP-router will decouple the IP core from the NoFPGA, this can be done through using a DualPort-RAMs (DP-RAM) as buffers for communicating messages. As a result, the IP core can use independent data-width and clock frequency resulting in an easy IP integration. Moreover, the ability to design IP cores in parallel is an attractive design methodology to sustain the quick time-to-market pressure.
The IP Interface of the NoFPGA node contains a routing logic table that is configurable to decide on which port to send the packets forwarded by the IP interface.

IP Core can insert a packet into the NoFPGA when the designated output port is free, this free output port will be used for new packets originating from the IP Core. If there are packets on all four inputs and no packet is to be forwarded to the IP Core, a packet from the IP Core is unable to be released on the NoFPGA. The NoFPGA node will direct the different packet flows and will interface the synchronous and asynchronous clock domains.

4.7 Packet Header Routing Information

The message will be segmented into 16-, 32-bit and 64-bit flits. The header includes routing information used by the IP router to decide where to send the incoming input packet. Figure 4.7 displays the header routing information as follows:

<table>
<thead>
<tr>
<th>Xd_addr</th>
<th>yd_addr</th>
<th>X_flag</th>
<th>y_flag</th>
</tr>
</thead>
</table>

Figure 4.7 Header Routing Information
- **Xd_addr**: the number of hops the message needs to be routed in the X direction, it is 4 bits field, so the total number of hops in the X direction is $2^4 = 16$.

- **Yd_addr**: the number of hops the message needs to be routed in the Y direction, it is 4 bits field, so the total number of hops in the Y direction is $2^4 = 16$.

- **X_flag**: it is a 1-bit flag to determine if the message is routed in the positive X direction or the negative X direction.

- **Y_flag**: it is a 1-bit flag to determine whether the message is routed in the positive Y direction or the negative one.

Since the xd_addr and yd_addr are both 4-bit field routing header information, the maximum supported 2-D mesh or 2-D Torus is 16x16, and the total number of NoFPGA nodes are 256 nodes. The xd_addr and yd_addr can be considered as a configurable NoFPGA parameter to support larger dimensional topologies.
CHAPTER 5

IP ROUTER ARCHITECTURE

The IP router is the heart of NoFPGA, each NoFPGA node is composed of two functional parts, the IP core and the IP router. Each IP Core will be encapsulated into a fixed communication layer, the IP router, which is identical among all IP Cores within the NoFPGA. This will provide an efficient way of hardware reuse of the IP Core by integrating them through our proposed NoFPGA.

NoFPGA based design is a structural and a modular approach. This will shorten the design cycle phase and introduce the product faster to the market. NoFPGA is a Platform-based design (PBD) approach to reduce the design cost by an IP-reuse. PBD represents a key design methodology of SoC and provides a high system IP reuse at both block level as well as at system level. Also, by de-coupling applications and system architectures, the design cycle is further shortened. Our most important goal is to design a highly scalable NoFPGA, this will enable a re-use of already available IP cores. If the design of the NoFPGA interface is properly
implemented, it will make IP integration easier with little cost and effort. Also, NoFPGA structured implementation enable faster design cycle and shorter time-to-market product release. The design can be divided into sub-modules that can be implemented in parallel. The integration of the design will happen next resulting in a shorter development cycle.

5.1 Mesh Topology Router Architecture

IP router is responsible for switching incoming packets on any of the five input ports, named North, East, South, West, and IP, to any of the five output ports based on header routing information. Figure 5.1 shows a block diagram of the Mesh topology IP router architecture, it shows the 5 input/output ports, North, East, South, West, and local IP.

Figure 5.1 Block Diagram of NoFPGA IP router
The functionality of an IP router can be viewed as multiplexing and de-multiplexing five different data flows: The North, East, South, West, and IP-direction. Each direction has a FIFO buffer. The IP interface to the NoFPGA translates the interface signals of the functional IP core to the NoFPGA packet format. All IP cores are implemented as independent IP cores.

The IP router is composed of a node controller and five port modules, one in each direction. The IP node controller is responsible for multiplexing incoming five data flow on the outgoing five output ports, it also generates the handshaking signals for flow control. Figure 5.2 shows the block level diagram of Mesh IP router, it shows the node controller as well as the five ports modules.

![Figure 5.2 Mesh IP Router block diagram](image)
The five I/O port modules connect the IP core to its four neighboring nodes; East, South, West, North, and the local IP core. The links are bi-directional and data flows in both direction. The X-Y routing algorithm is used with direction flag bit to define whether data is moved in positive X-direction or negative X-direction, as well as positive Y-direction or negative Y-direction.

5.1.1 Node Controller

The function of the node controller is to multiplex the five input data flows into the five output data flows. Data flow on one input port can not be multiplexed on the output of the same input port, for example, data flow incoming on the North input port can not be switched on the North output port since in real application this is not possible, data can not be bounced back on the same port.

The design is implemented in VHDL as a soft core IP router that can be used and re-used in low cost NoFPGA architecture.
Figure 5.3 Mesh IP Router Entity

Figure 5.3 shows the entity declaration of the mesh IP router, it shows the five input/output data flows as well as the handshaking signals.

In order to maintain minimum logic overhead, the complexity of the IP router should be minimized, this means simplifying the design of IP router building blocks, the node controller and port module, the hardware overhead of mesh topology IP
router architecture will be evaluated and compared to other possible topologies, mainly Torus topology.

The node controller receives a request from port modules to assign an output port to incoming packet, when two or more packets request the same output port, a resolution mechanism based on predefined priority will be used as follows; packets incoming on North input will be assigned the highest priority followed by East, South, and West. Other arbitration techniques can be implemented but the idea is to keep node controller simple. The node controller is responsible for making the connection between external handshaking flow control signals and the corresponding port controller.

The node controller will adjust the incoming packet header when switching the packet on output port. The node controller will update the header to reflect the remaining number of hops that needs to travel before it reaches its destination. The node controller will update:

- Xd_addr if packet is switched a long X-direction
- Yd_addr when packet is switched along Y-direction
Figure 5.4 Detailed Mesh Router Block Diagram
The node controller is also responsible for generating the control for flow control handshaking signal (Latch_Out, RDY_In) associated with the data flow to communicate with the next hop on the data path.

5.1.2 Port Module

The port module is the input/output port. It consists of the port controller and the buffer as shown in figure 5.4. The buffer is a FIFO buffer and holds the incoming packets until it can be switched on its destination output port. The output port is composed of three 5X5 crossbar-switches. The first crossbar is for Data flow switching, the other two crossbars are for output handshaking signals pairs (Latch_OUT_XX, RDY_OUT_XX).

The port controller will examine the header routing information of the incoming packet resides in the buffer. The port controller will determine the output port based on routing information in the X and Y header as follows:

- If both X and Y headers are Zero, the packet is scheduled to be delivered to the Local IP Core.
- If the packet is coming along X-direction and X header is nonzero, the port controller will generate a request to the node controller for the destination output port along X-direction.
• If the packet is coming along X-direction and X header is Zero but Y header is nonzero, the port controller will generate a request to the node controller for the destination output port along Y-direction.

• If the packet is coming along Y-direction and Y header is nonzero, the port controller will generate a request to the node controller for the destination output port along Y-direction.

The port controller is responsible for monitoring the buffer status and generating the input handshaking signals to allow data to be latched in the input buffer. If the buffer is full, the port controller will block incoming packets from being latched in the input buffer by de-asserting RDY_In handshake signal. As a result of de-asserting RDY_In, the neighboring nodes on that direction will be blocked from sending packets across this input port.

5.2 Mesh Edge Node

The Mesh nodes that are physically located on the NoFPGA edges are special nodes and are considered as boundary nodes. To make the NoFPGA more homogenous architecture, the issues of unconnected links of edge nodes should be resolved and not left without being connected. There are two types of edge nodes as follows:
5.2.1 Horizontal Edge Node

These nodes have one link that is not connected, either the north port if the node is in the top row, or the south port if the node is in the bottom row. To make the topology more homogenous, input port on unconnected link is connected to output link of the same port as shown in figure 5.5.

![Figure 5.5 Mesh Edge Node (Horizontal and Vertical)](image-url)
5.2.2 Vertical Edge node

These nodes are either located on the most right column of NoFPGA or the most left column of NoFPGA. Unconnected link on input port will be connected to output link of the same port as shown in figure 5.5.

5.3 Torus Topology Router Architecture

In 2-D Mesh topology, the IP Router routes packets in all directions, North, East, South, and West. To simplify the routing logic, we will implement a simpler routing topology, the Torus. Torus is simply a 2-D Mesh folded along both vertical and horizontal directions. Packets will be routed only in two directions, East-West and North-South in addition to local IP direction. Figure 5.6 shows a block diagram of the Torus topology IP router architecture. It shows the two input/output ports, X (East-West) and Y (North-South), and local IP.

![Figure 5.6 Block Diagram of NoFPGA Torus IP Router](image-url)
The Torus IP router is composed of a node controller and three port modules, one in the X direction, one in the Y direction and one in the IP direction. The IP node controller is responsible for multiplexing incoming three data flow on the outgoing three output ports. It also generates the handshaking signals for flow control. Figure 5.7 shows the block level diagram of Torus IP router, it shows the node controller as well as the three ports modules.

![Torus IP Router Diagram](image)

Figure 5.7 Torus IP Router
The three I/O port modules connect the IP core to its four neighboring nodes; East, South, West, North, and the local IP core by daisy-chaining the X-port module to form a horizontal circular ring and Y-port module to form a vertical ring. The same routing algorithm used for Mesh topology IP router will be used with Torus IP router, the X-Y routing, but only along two directions, East and South, which in turn will reduce the complexity of the IP router.

5.3.1 Node Controller

The function of the node controller is to multiplex the three input data flows into the three output data flows. Data flow on one input port can not be multiplexed on the output of the same input port, for example, data flow incoming on one input port can not be switched on the same output port since in real application this is not possible. Data cannot be bounced back on the same port.

The design is implemented in VHDL as a soft core IP router that can be used and re-used in low cost NoFPGA architecture.
Figure 5.8 Torus IP Router Entity

Figure 5.8 shows the entity declaration of the Torus IP router. It shows the three input/output data flows as well as the handshaking signals.

The IP router is composed of a one input/output module for each direction; one controller for the X-direction, another for the Y-direction, and a third input/output
module for the local IP direction. There is also the node controller to schedule requests from each one of the previous port modules.

When Torus IP router is compared with Mesh IP router, less routing decision is necessary since only routing is done across two directions. As a result, the routing logic overhead will be smaller which makes Torus topology as an attractive choice for low cost NoFPGA. The synthesis results of both topologies will be addressed in chapter 7 of this dissertation.

As in Mesh topology, the node controller receives a request from port modules to assign an output port to incoming packet, when two or more packets request the same output port, a resolution mechanism based on predefined priority will be used as follows; packets incoming along East direction will be assigned the highest priority followed by packets incoming along South direction. Other arbitration techniques can be implemented but the idea is to keep node controller simple. The node controller is responsible for making the connection between external handshaking flow control signals and the corresponding port controller.

The node controller will adjust the incoming packet header when switching the packet on output port. The node controller will update the header to reflect the remaining number of hops the needs to travel before it reaches its destination. The node controller will update:
- Xd_addr if packet is switched along X-direction
- Yd_addr when packet is switched along Y-direction

The node controller is also responsible for generating the flow control handshaking signal (Latch_Out, RDY_Out) associated with the data flow to communicate with the next hop on the data path.

Figure 5.9 Detailed Torus Router Block Diagram
5.3.2 Port Module

The port module is the input/output port. It consists of the port controller and the buffer as shown in figure 5.9, the buffer is a FIFO buffer and holds the incoming packets until it can be switched on its destination output port. The output port is composed of three 3X3 crossbar-switches. The first crossbar is for Data flow switching, the other two crossbars are for output handshaking signals pairs (Latch_OUT_XX, RDY_OUT_XX)

The port controller will examine the header routing information of the incoming packet resides in the buffer. The port controller will determine the output port based on routing information in the X and Y header as follows:

- If both the X and the Y headers are Zero, the packet is scheduled to be delivered to the Local IP Core.
- If the packet is coming along the X-direction and the X header is nonzero, port controller will generate a request to the node controller for the destination output port along the X-direction.
- If the packet is coming along the X-direction and the X header is Zero but the Y header is nonzero, the port controller will generate a request to the node controller for the destination output port along the Y-direction.
• If the packet is coming along the Y-direction and the Y header is nonzero, the port controller will generate a request to the node controller for the destination output port along the Y-direction.

The port controller is responsible for monitoring the buffer status and generating the input handshaking signals to allow data to be latched in the input buffer. If the buffer is full, the port controller will block incoming packets from being latched in the input buffer by de-asserting RDY_In handshake signal. As a result of de-asserting RDY_In, the neighboring nodes on that direction will be blocked from sending packets across this input port.

5.3.3 Torus Communication Scenario

The communication protocol is based on a request-ack cycle; each packet is transmitted after a negotiation session between handshake signals. Each IP core in the NoFPGA can be either a sender or a receiver. When a message enters a router, it can either be forwarded along the current direction, sent orthogonally or delivered to the local IP-block. Suppose a neighbor X wishes to communicate with a neighbor Y through this node shown in figure 5.9:

• Neighbor X asserts Latch_in_X, causing the header flit to be latched into the dimension X-flit buffer (FB)
• The X-Control Module de-asserts RDY_in_X, examines the header flit and requests output channel Y via REQ-X

• The Node Control asserts ACK_X in response if channel Y is Free and sets each of the 3X3 CrossBars to appropriately connect Out_Y, Latch_Out_Y, and RDY_Out_Y

• The message is then passed in a pipeline fashion, with RDY_In_X and Latch_In_X functioning as input handshake signals and RDY_Out_X and Latch_Out_X as output handshake signals.

• Once the tail of the message passes, X-Control de-asserts REQ_X, allowing node control to relinquish the Y channel.
CHAPTER 6

NoFPGA EXPERIMENTAL SYNTHESIS RESULTS

The proposed NoFPGA is modeled using VHDL. VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a computer-aided design language used for design and verification of hardware circuits. It is used to describe hardware circuits from an abstract level to the gate level.

The NoFPGA architecture is modeled as a software core that can be used in implementing low cost NoFPGA systems. Both Mesh based NoFPGA and Torus based NoFPGA were implemented in VHDL as an off the shelf softcore that can be highly utilized and customized when implementing SoFPGA.

6.1 Design Tool

The NoFPGA architecture was modeled in VHDL using Lattice Semiconductor ispLEVER version 4.2 tool suite. The ispLEVER supports multiple
design entries such as VHDL, Verilog, and Schematic Capture. It is used with Lattice FPGA device families and supports Leonardo Spectrum Synthesis environment from Mentor Graphics as well as Synplify from Synplicity. For HDL simulation, ispLEVER is integrated with ModelSim from Mentor Graphics.

The ispLEVER design tool provides the tool to produce a configured FPGA. In the design flow, the model is implemented in VHDL, this design entry stage is followed by map, place, and route tools that translate the netlist into a routed FPGA. The ispLEVER has a floor planner tool to support layout manually. A bit stream generator is used to generate the configuration data that is necessary to program the FPGA.

Both Mesh and Torus NoFPGA were synthesized in ORCA FPSC (Field Programmable System-on-a Chip) FPGA family. The device used for synthesis is ORCA4 ort8850FPSC-1, it is a 600 Kgate with 16 K LUT and 148K of Embedded Block RAM [73].

6.2 NoFPGA Architecture Model

The NoFPGA architecture has been modeled first by modeling the IPRouter of both Topologies, Mesh and Torus, in VHDL as described in chapter five. Second, the Mesh and Torus NoFPGA was modeled as a top level entity by grouping 3X3 and
4X4 IPRouters and interconnect them in both topologies. All of the IPRouter can send and receive packets and run independent of the NoFPGA interconnection network.

The design cost of IPRouter Buffering, the most expensive building block, is evaluated based on two different implementation approaches. First, IPRouter buffering based on distributed memory. Second, IPRouter buffering based on Embedded Block RAMs. Also, the design cost of both Mesh and Torus topologies is compared in reference to the total number of logic cells.

6.3 IPRouter Buffering

The IPRouter buffering mechanism is implemented as FIFO. The FIFO size is determined by the packet format. The width of the FIFO corresponds to the packet data width. On the other hand, the depth of the FIFO represents the total number of messages that can be buffered at incoming port. Both FIFO width and FIFO depth are tuning parameters and affects the cost associated with FIFO implementation. Consequently, it affects the implementation cost of IPRouter. The buffer size is the most critical part of the NoFPGA IPRouter and should be highly optimized. Otherwise, the hardware overhead will be high and the implementation of NoFPGA approach will be an expensive approach.
The implementation of FIFO depends on utilizing resources of FPGA. There are two possible ways to implement FIFO as either sysMEM Embedded Block RAM (EBR) or as a distributed RAM across Programmable Function Units (PFU). The cost associated with both implementation will be compared.

Figure 6.1 shows the block diagram of FIFO with its input and output ports, the tuning parameters are both the data width that is represented by DataIn and DataOut ports as well as the FIFO depth.
6.3.1 PFU (Distributed Memory) FIFO

The Tuning parameters of the PFU FIFO are the data width and the FIFO depth. Evaluation of synthesis cost associated with FIFO implementation is described in the following subsections. We shall adjust both parameters and evaluate the cost impact on the synthesis results. Three different data width will be evaluated as follows: 16-bit, 32-bit, and 64-bit with FIFO depth of both 32 and 64.

16-bit data width, 32 and 64 depth FIFO

When designing FIFO with 16-bit and 32 depth, table 6.1 shows the cost represented in the number of PFUs, number of registers, and number of LUTs. Table 6.1 shows that the total number of PFUs to implement a FIFO based on distributed memory blocks increases when FIFO depth increases. With only 32 depth FIFO, 11 PFU is necessary to synthesize the FIFO, this number experience almost 54% increase when FIFO depth changes to 64 entries instead of 32 entries.

<table>
<thead>
<tr>
<th>FIFO Depth</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>11</td>
<td>13</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>64-depth</td>
<td>17</td>
<td>15</td>
<td>40</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1 16-bit FIFO Implementation Cost
32-bit data width, 32 and 64 depth FIFO

Table 6.2 shows that FIFO’s depth is a tuning parameter when implementing FIFO, for example, when comparing depth of 32 to depth of 64, 64 depth FIFO consumes 25 PFUs versus only 15 PFUs for 32-depth FIFO, both represents 0.7% and 1.2% of the total available PFUs of FPSC respectively.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>15</td>
<td>13</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>64-depth</td>
<td>25</td>
<td>15</td>
<td>40</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2 32-bit FIFO Implementation Cost

64-bit data width, 32 and 64 depth FIFO

By implementing a wider data of 64-bit instead of 32-bit, table 6.3 shows that almost 50% increase in the number of PFUs due to multiplexing increase when implementing FIFO. The number of PFU is also doubled by the fact that both data width and FIFO depth affect the consumed logic to implement FIFO based on PFU Distributed Memory.
As a result, the total number of PFUs to implement a FIFO based on Distributed Memory Block increases when both data width as well as FIFO depth increases.

**Putting it all together**

As shown from the synthesis results of PFU based FIFO implementation, the number of PFU increases when both data width and FIFO depth increase. Figure 6.2 shows the consumed logic of 16-bit Data width FIFO.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>23</td>
<td>13</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>64-depth</td>
<td>41</td>
<td>15</td>
<td>40</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.3 64-bit FIFO Implementation Cost
Figure 6.2 16-bit FIFO cost

On the other hand, figures 6.3 and 6.4 show the number of PFUs, LUTs, Registers, and EBR Block RAM for 32-bit FIFO as well as 64-bit FIFO. It is important to study the effect of increase data width when keeping the FIFO depth remains constant. Figure 6.5 shows the total cost increase when data width changes from 16-bit to 32-bit to 64-bit while the FIFO depth is kept at 32. Only the number of PFU increases. For example with 16-bit, the total number of PFUs is 11 while for 32-bit FIFO, this number goes up to 15 PFUs reflecting an increase of 36 %.
Figure 6.3 32-bit FIFO cost

Figure 6.4 64-bit FIFO cost
The cost effect of increasing the data width is less than the cost effect of increasing the FIFO depth. To optimize the FIFO logic cost, the best configuration is with 64-bit data width and 32 FIFO depth, the cost is as follows:

- PFUs Cost: 23
- LUTs Cost: 13
- Registers Cost: 36

![PFU Cost](image)

**Figure 6.5 FIFO PFU Cost**

### 6.3.2 EBR (Embedded Block RAM) FIFO

We have studied the design cost of FIFO implementation based on Distributed Memory PFU. In this section, we will study the design implementation of FIFO
utilizing the FPSC EBR blocks. The PFUs will be saved to implement the logic of IPRouter functionality and IP Cores themselves.

FPSC has 512X18 dual-port RAM blocks to significantly increase the amount of memory. These EBR blocks support multiple memory configuration including FIFO. Utilizing the 16 EBR blocks available in FPSC, FIFO can be implemented and FPU logic can be saved. The tuning parameters are again the data width and FIFO depth, We will evaluate the cost associated with EBR FIFO and compare results to PFU Distributed Memory FIFO.

**16-Bit EBR FIFO**

When synthesizing FIFO using available EBR, only the Block RAM will be used leaving the PFU for implementing the other building blocks of IPRouters as well as the IP Cores themselves. Table 6.4 shows the cost associated with 16-bit EBR based FIFO. Only one block is necessary to implement 16-bit EBR FIFO with both 32 and 64 depth.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>64-depth</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.4 16-Bit EBR FIFO Cost
32-Bit EBR FIFO

To see the impact of implementing 32-bit FIFO, we synthesized the EBR FIFO with 32-bit and 32 depth as well as 64 depth. Synthesis results show again that one EBR can support both FIFO either with 32-depth or with 64-depth FIFO. Table 6.5 shows the synthesis results.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>64-depth</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.5 32-Bit EBR FIFO Cost

64-Bit EBR FIFO

Table 6.6 shows the cost associated with EBR FIFO with the following tuning parameters, 32-depth and 64-depth while data width is kept at 64-bit. Two memory blocks are necessary to implement 64-bit FIFO which an increase of 100 % more than 32-bit FIFO. Considering that only 16 EBR blocks are available, the most cost
effective implementation is a 32-bit data width and 32-depth FIFO. This configuration will only consume one EBR of the available 16 EBR blocks.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>32-depth</th>
<th>64-depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PFU 1</td>
<td>LUT 1</td>
</tr>
<tr>
<td></td>
<td>Registers 0</td>
<td>EBR Block 2</td>
</tr>
<tr>
<td></td>
<td>Registers 0</td>
<td>EBR Block 2</td>
</tr>
</tbody>
</table>

Table 6.6 64-Bit EBR FIFO Cost

6.4 Torus IP Router Synthesis Cost

We will study the cost impact of IPRouter with the possible FIFO implementation. First we will study the cost of Torus IPRouter with PFU FIFO. Second, we will evaluate the cost impact of using EBR FIFO.

6.4.1 16-Bit Communication Link PFU FIFO Based IP Router

The Torus IPRouter was implemented with 16-Bit data width of each of the three ports, X, Y, and local IP port. Table 6.7 shows the cost associated with implementing the IPRouter based on PFU FIFO, with a communication link of 16-bit and 32-depth FIFO, only 43 FPUs are needed which represents only 2.3% of the available logic. If we consider using the gate size as a unit to describe the logic usage, the ratio of LUTs usage to gate size is usually 1:10. As shown in table 6.7, the cost of
IPRouter is 178 LUTs that is equivalent to 1780 gate or 1.78 Kgate. FPSC is a 600 Kgate and IPRouter only represents 0.3 % of the available logic on device. As a result of this low cost NoFPGA customizable IPRouter, NoFPGA represents an excellent candidate architecture for future SoC platform design.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>43</td>
<td>174</td>
<td>119</td>
<td>0</td>
</tr>
<tr>
<td>64-depth</td>
<td>55</td>
<td>178</td>
<td>127</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.7 16-Bit Torus IPRouter PFU FIFO Cost

6.4.2 32-Bit Communication Link PFU FIFO Based IP Router

When changing the communication links to 32-bit instead of 16-bit and synthesize the IPRouter based on the new configuration, table 6.8 shows the logic cost associated with this IPRouter configuration, it shows that for a 32-bit and 32-depth PFU FIFO, the total cost is 70 PFU while 64-depth FIFO utilize 90 PFUs. To represent the total cost in terms of gate size, as shown in table 6.8, the Torus IPRouter consumes 322 LUTs which is equivalent to 3.22 Kgate. As a summary, the cost of 32-bit Torus IPRouter represents 0.53 % of the total FPSC available logic.
6.4.3 64-Bit Communication Link PFU FIFO Based IP Router

We repeated the previous experiment with tuning the communication link to 64-bit instead of 32-bit, table 6.9 represents how much logic is necessary to implement the Torus IPRouter with this configuration, it shows that the total cost of 64-bit 64-depth PFU FIFO based Torus IPRouter is 177 PFUs, or 615 LUTs. The cost size represents 6.15 Kgates that is about 1.0 % of the total FPSC available logic.
6.4.4 16-Bit Communication Link EBR FIFO Based IP Router

To evaluate the cost of Torus IPRouter with EBR based FIFO, table 6.10 shows the cost of a 16-Bit communication link IPRouter, only 23 PFUs is necessary to implement with a 150 total number of LUTs. The synthesis results show that the size of this IPRouter is 1.5 Kgates that represents 0.25 of the total FPSC available logic.

<table>
<thead>
<tr>
<th>FIFO Depth</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>23</td>
<td>150</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>64-depth</td>
<td>23</td>
<td>150</td>
<td>47</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.10 16-Bit EBR Based Torus IP Router Cost
As shown from table 6.10, one EBR is consumed to synthesize the FIFO. This EBR block is enough to implement either a 32-depth or a 64-depth FIFO leaving more logic for implementation of other building blocks of the IPRouter.

6.4.5 32-Bit Communication Link EBR FIFO Based IP Router

With a wider communication link, the IPRouter cost is shown in table 6.11. As shown from the table, the cost of 32-Bit IPRouter with EBR FIFO is 294 LUTs that is equivalent to 2.94 Kgates.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>42</td>
<td>294</td>
<td>95</td>
<td>1</td>
</tr>
<tr>
<td>64-depth</td>
<td>42</td>
<td>294</td>
<td>95</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.11 32-Bit EBR Based Torus IP Router Cost

Only one EBR block is necessary to implement the IPRouter with either 32-depth or 64-depth FIFO.
6.4.6 64-Bit Communication Link EBR FIFO Based IP Router

When using a communication link of 64-bit wide, table 6.12 shows that the total cost of implementing the IPRouter based on 64-bit EBR FIFO is 587 LUTs, this represents 5.87 Kgates. According to the cost analysis of implementing low cost Torus IPRouter, the size of the IPRouter is 5.87 Kgates that is about 1.0% of the total FPSC gate size.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>97</td>
<td>587</td>
<td>191</td>
<td>2</td>
</tr>
<tr>
<td>64-depth</td>
<td>97</td>
<td>587</td>
<td>191</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.12 64-Bit EBR Based Torus IP Router Cost

As shown from table 6.12, two EBR blocks are necessary to implement the IPRouter EBR FIFO with either 32-depth or 64-depth.

6.5 Mesh IP Router Synthesis Cost

The cost impact of Mesh IPRouter will be evaluated. First, we will study the cost impact of Mesh IPRouter based on PFU (Distributed Memory) FIFO
implementation and compare synthesis results with the cost impact of Mesh IPRouter based on EBR FIFO implementation.

6.5.1 16-Bit Communication Link PFU FIFO Based IP Router

In Mesh topology IPRouter, five ports are allocated, one in each direction, North, East, South, West, and Local IP. The Mesh IPRouter routes packets in all five directions, when compared to the Torus IPRouter, only three ports are allocated for the Torus IPRouter. Table 6.13 shows the cost of a 16-Bit IPRouter with PFU FIFO, the total cost of 32-depth is 80 PFUs. This represents a cost of 298 LUTs or 2.98 Kgates equivalent. The cost shows an increase of 67% when compared to the Torus IPRouter with same configuration.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>80</td>
<td>298</td>
<td>223</td>
<td>0</td>
</tr>
<tr>
<td>64-depth</td>
<td>104</td>
<td>306</td>
<td>239</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.13 16-Bit PFU Based Mesh IP Router Cost
6.5.2 32-Bit Communication Link PFU FIFO Based IP Router

The impact of a higher communication link on the cost of the Mesh IPRouter is studied, a 32-bit communication link IPRouter is synthesized. The synthesis results show an increase on the number of PFUs necessary to implement a wider communication link IPRouter. Table 6.14 shows that the cost of IPRouter is 123 PFUs, it shows a cost size of 527 LUTs that is equivalent to 5.27 Kgate. The total cost represents an increase of 63% when compared to equivalent configuration of Torus IPRouter. Since we implemented FIFO in PFU distributed memory, zero EBR memory block is consumed with this configuration.

<table>
<thead>
<tr>
<th>FIFO Depth</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>123</td>
<td>519</td>
<td>303</td>
<td>0</td>
</tr>
<tr>
<td>64-depth</td>
<td>163</td>
<td>527</td>
<td>359</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.14 32-Bit PFU Based Mesh IP Router Cost
6.5.3 64-Bit Communication Link PFU FIFO Based IP Router

The same experiment is repeated with a larger communication link of 64-Bit instead of 32-Bit. Table 6.15 shows the cost of the new configuration of the IPRouter. It shows that 969 LUTs is the size of the new IPRouter that represents 9.69 Kgates. When compared to equivalent Torus IPRouter, the Mesh IPRouter costs 58% more the Torus IPRouter. Also, synthesis results show a wider communication link cost 4.42 Kgates more compared to 32-bit implementation. This represents almost 83% increase compared to IPRouter cost with 32-bit communication link.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>208</td>
<td>961</td>
<td>463</td>
<td>0</td>
</tr>
<tr>
<td>64-depth</td>
<td>280</td>
<td>969</td>
<td>479</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.15 64-Bit PFU Based Mesh IP Router Cost

To sum up, the communication link and PFU FIFO depth affect the cost size of the synthesizable IPRouter. The cost size is linearly increased with the size of the communication link and the depth of the PFU FIFO respectively.
6.5.4 16-Bit Communication Link EBR FIFO Based IP Router

The cost of the Mesh IPRouter with EBR FIFO based will be cheaper due to the fact that EBR FIFO will be synthesized in one of the available EBR memory blocks on the FPGA. Table 6.16 shows the synthesis cost of the 16-bit EBR FIFO based IPRouter. The table shows only 40 PFUs is needed to implement the design with one EBR consumed to synthesize the FIFO, the total cost is 250 LUTs that represents 2.5 Kgate. The cost shows a 66% increase over the equivalent Torus IPRouter cost with same configuration. It also shows that EBR FIFO based IPRouter is 22% cheaper than PFU FIFO based IPRouter.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>40</td>
<td>250</td>
<td>79</td>
<td>1</td>
</tr>
<tr>
<td>64-depth</td>
<td>40</td>
<td>250</td>
<td>79</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.16 16-Bit EBR Based Mesh IP Router Cost
6.5.5 32-Bit Communication Link EBR FIFO Based IP Router

The Mesh IPRouter is configured with a 32-bit communication link. The results of synthesis show a cost increase. One EBR is enough to synthesize a 32-bit with either 32-depth or 64-depth FIFO. The total cost as shown in Table 6.17 is 71 PFUs that is 471 LUTs or 4.71 Kgate. This cost is almost 59% higher than the equivalent Torus IPRouter with the same configuration.

<table>
<thead>
<tr>
<th>FIFO DEPTH</th>
<th>PFU</th>
<th>LUT</th>
<th>Registers</th>
<th>EBR Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-depth</td>
<td>71</td>
<td>471</td>
<td>159</td>
<td>1</td>
</tr>
<tr>
<td>64-depth</td>
<td>71</td>
<td>471</td>
<td>159</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.17 32-Bit EBR Based Mesh IP Router Cost

6.5.6 64-Bit Communication Link EBR FIFO Based IP Router

The cost of the Mesh IPRouter will continue to increase linearly with the communication link increase. Table 6.18 shows that a 64-bit communication link IPRouter costs 120 PFUs or 913 LUTs, this cost shows an increase of 93% over 32-bit IPRouter and also shows a 55% cost increase over the Torus IPRouter with equivalent configurations.
As a result of the cost analysis of the Mesh IPRouter, the cost experiences a linear increase with the increase of the communication link bandwidth to support higher bandwidth. It also experiences an increase of the LUTs usage when compared to equivalent Torus IPRouter due to the fact that Torus IPRouter only routes in two directions and simpler logic can be used to perform the routing functionality. Table 6.18 shows the cost of EBR FIFO is two EBR blocks, this represent an increase from the only one EBR block to synthesize a 32-bit IPRouter.

### 6.6 Putting it all together

The cost of the Torus IPRouter experiences a decrease in the consumed logic when compared to the equivalent Mesh IPRouter. As shown in figure 6.6 and figure 6.7, the Torus topology showed a 40% logic drop when compared to the Mesh topology. The drop in logic usage is due to the fact that the Torus IPRouter only routes packets in two directions, east and south. On the other hand, the Mesh IPRouter routes packets in all eight directions.
packets in four directions, North, East, South, and West. Also, the Mesh Topology has five ports compared to only three ports for Torus IP Router.

Figure 6.6 Torus and Mesh IP Router Cost with EBR FIFO
6.7 IP Core Size and TAB (Tile Architecture Block)

To be able to evaluate the logic cost associated with NoFPGA implementation, it is important to relate the size of either Torus or Mesh IPRouter to the size of individual IP Cores. To do so, a set of available opencores IP cores are synthesized in a Lattice ORCA FPSC FPGA. In order to estimate the Size of IP Cores, the opencores IP source code is used as our based design.
The design was run through the ispLEVER ver4.2 tools, the target device is Lattice Semiconductor ORCA FPSC FPGA family. Table 6.19 shows the size of a set of individual IP Cores picked from opencores.org [74]. The IP Cores chosen are:

- Ethernet MAC 10/100 Mbps
- Floating Point Unit
- USB Controller
- Memory Controller
- I2C Controller

<table>
<thead>
<tr>
<th></th>
<th>Size in LUTs</th>
<th>Size in Kgate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet MAC 10/100 Mbps</td>
<td>2540</td>
<td>25.4</td>
</tr>
<tr>
<td>Floating Point Unit</td>
<td>3500</td>
<td>35</td>
</tr>
<tr>
<td>USB Controller</td>
<td>3182</td>
<td>31.8</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>1840</td>
<td>18.4</td>
</tr>
<tr>
<td>I2C Controller</td>
<td>209</td>
<td>2.09</td>
</tr>
</tbody>
</table>

Table 6.19 Cost analysis of opencores IP Cores
As shown from table 6.19, most of the IP Cores have between 18 K
gates and 35 K
gates with exception to I2C controller which has a small synthesizable core. If we consider the largest IP Core (Floating Point Unit) as our reference, the Torus IPRouter cost represents a range of 4.2\% to 17.5\% of the total cost of IP Core. On the other hand, the Mesh IPRouter cost represents a range of 7.1\% to 27.6\% of the total cost of IP Core. The range of the Torus and the Mesh IPRouter is due to the mechanism of FIFO implementation as well as different configuration parameters such as the communication link. As a results of the cost estimate of IP Cores and IPRouter, the size of the Tile Architecture Block (TAB) can be chosen to be in the range of 40 K
gates to 50 K
gates, the TAB will be large enough to fit both the IP Core as well as the IPRouter.
CHAPTER 7

PERFORMANCE ANALYSIS

7.1 Torus & Mesh IP Router Performance

We studied the performance analysis of the IPRouter of both Torus and Mesh topologies. First, the maximum operating frequency is determined. Second, the maximum link bandwidth between adjacent IPRouters is found. Also, the maximum bandwidth of 3 x 3 Torus NoFPGA is compared to 3 x 3 Mesh NoFPGA.

7.1.1 Torus Operating Frequency

The Torus IPRouter is simpler and consumes less logic compared to the Mesh IPRouter, this will results in higher operating frequency of Torus IPRouter when compared to equivalent. Table 7.1 shows the maximum operating of Torus IPRouter when implemented with 16-bit, 32-bit, and 64-bit communication link. For example, the 64-bit communication link Tours IPRouter has a maximum operation frequency of 86 Mhz.
### Table 7.1 Torus Maximum Operating Frequency

<table>
<thead>
<tr>
<th>Data Link</th>
<th>Maximum Operating Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit</td>
<td>79.8 Mhz</td>
</tr>
<tr>
<td>32-bit</td>
<td>87 Mhz</td>
</tr>
<tr>
<td>64-bit</td>
<td>85.9 Mhz</td>
</tr>
</tbody>
</table>

#### 7.1.2 Mesh Operating Frequency

The Mesh IPRouter consumes more logic compared to the Torus IPRouter. Table 7.2 shows the maximum operating of the Mesh IPRouter when implemented with 16-bit, 32-bit, and 64-bit communication link. For example, the 64-bit communication link Mesh IPRouter has a maximum operation frequency of 57.1 Mhz.
As shown from table 7.1 and table 7.2, the Torus IPRouter experiences a higher operating frequency when compared to the equivalent Mesh IPRouter due to the fact that the Torus IPRouter is simpler and consumes less routing logic.

### 7.1.3 Communication Link Bandwidth

Since the maximum operating frequency of the Torus IPRouter is higher than the corresponding Mesh IPRouter. As a result, the Torus IPRouter will experience a higher link bandwidth than the Mesh IPRouter. Table 7.3 shows the maximum link bandwidth of the Torus IPRouter. As can be shown from table 7.3, the wider the communication link, the higher the bandwidth it supports.
Maximum Link Bandwidth

<table>
<thead>
<tr>
<th>Data Link</th>
<th>16-bit</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>159.6 Mbyte/Sec</td>
<td>348 Mbyte/Sec</td>
<td>687.2 Mbyte/Sec</td>
</tr>
</tbody>
</table>

Table 7.3 Torus Link Bandwidth

On the other hand, table 7.4 shows the link bandwidth of the Mesh IPRouter. The Mesh IPRouter has less link bandwidth when compared to the equivalent Torus IPRouter.

<table>
<thead>
<tr>
<th>Data Link</th>
<th>16-bit</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>116 Mbyte/Sec</td>
<td>253.2 Mbyte/Sec</td>
<td>456.8 Mbyte/Sec</td>
</tr>
</tbody>
</table>

Table 7.4 Mesh Link Bandwidth
Figure 7.1 shows that Torus IPRouter always has a higher communication link bandwidth compared to the Mesh IPRouter, for example with 32-bit communication link, the Torus IPRouter experiences a 27% higher bandwidth than the equivalent Mesh IPRouter.

7.2 NoFPGA Total Bandwidth

The total bandwidth of both Torus and Mesh NoFPGA topologies is estimated. The total bandwidth of both topologies depends on the communication link bandwidth
as well as the total number of link of each topology. The total bandwidth of 3 x 3 NoFPGA of both topologies is estimated and compared.

7.2.1 3 x 3 Torus NoFPGA

It has been found that the 32-bit Torus IPRouter has a link bandwidth of 348 Mbyte/Sec. In order to estimate the 3 x 3 Torus NoFPGA total bandwidth, the number of links needs to be calculated as follows:

\[ N_{\text{link}} = 2N \]

Where:

\[ N_{\text{link}} \]: Total Number of links
\[ N \]: The number of Nodes (for 3 x 3 NoFPGA, \( N = 9 \))

As a result, 3 x 3 Torus NoFPGA has 18 links. To find the total bandwidth, \( B_{\text{Torus}} \),

\[ B_{\text{Torus}} = N_{\text{link}} \cdot L_B \]

Where:

\[ B_{\text{Torus}} \]: The total bandwidth
\[ L_B \]: The link Bandwidth = 348 Mbyte/Sec for 32-bit Torus

As a result, the 3 x 3 Torus NoFPGA has 6.3 Gbyte/Sec.
7.2.2 3 x 3 Mesh NoFPGA

The number of links of Mesh topology is less than the number of links of Torus topology. Consequently, the total bandwidth of the 3 x 3 Mesh NoFPGA is less than the total bandwidth of the Torus NoFPGA due to the fact that the Mesh IPRouter has less communication link too. It has been found that the 32-bit Mesh IPRouter has a link bandwidth of 253 Mbyte/Sec. In order to estimate the 3 x 3 Mesh NoFPGA total bandwidth, the number of links needs to be calculated as follows [75]:

\[ N_{\text{link}} = 2N - 2r \]

Where:

- \( N_{\text{link}} \): Total Number of links
- \( N \): The number of Nodes (for 3 x 3 NoFPGA, \( N = 9 \))
- \( r \): The number of node in each direction (\( r \times r = N \)), \( r = 3 \)

As a result, 3 x 3 Mesh NoFPGA has 12 links. To find the total bandwidth, \( B_{\text{Mesh}} \)

\[ B_{\text{Mesh}} = N_{\text{link}} \times L_B \]

Where:

- \( B_{\text{Mesh}} \): The total bandwidth
- \( L_B \): The link Bandwidth = 253 Mbyte/Sec for 32-bit Mesh

As a result, the 3 x 3 Mesh NoFPGA has 3 Gbyte/Sec.
7.2.3 **Bus versus NoFPGA Total Bandwidth**

If the one link bandwidth is used to represent the maximum bandwidth of the Bus architecture. Figure shows that the 3 x 3 Torus NoFPGA delivers eighteen times higher bandwidth than the Bus architecture. On the other hand, the total bandwidth of the 3 x3 Mesh NoFPGA represents twelve times higher bandwidth when compared to Bus architecture maximum bandwidth. The 3x3 Mesh NoFPGA experience a 50 % drop in total bandwidth when compared to the equivalent Torus NoFPGA.

![Total Bandwidth Chart](image)

**Figure 7.2 Architecture Total Bandwidth**
7.3 Power Consumption estimate of Torus and Mesh NoFPGA

The achieved higher bandwidth of communication link of Torus NoFPGA when compared to the equivalent Mesh NoFPGA comes at the expense of power dissipation. We will estimate analytically the power cost associated with both Torus and Mesh NoFPGA.

The total power of NoFPGA is represented by the power consumed by the IP Router and the power consumed by communication links. We will focus on the power consumption of the communication link since it dominates the power consumption by the IP Router. The power per link is given by:

\[ P_{\text{link}} = \alpha \cdot C_{\text{link}} \cdot V_{dd}^2 \]

Where:

- \( P_{\text{link}} \): The total power per link.
- \( C_{\text{link}} \): Link parasitic capacitance (F/m)
- \( \alpha \): The switching factor on the link.

\[ C_{\text{link}} = C_m \cdot L_{w,noc} \]

Where:

- \( C_m \): Link parasitic capacitance per unit length (F/m)
NoFPGA wire length that depends on the network topology, we will assume both the Mesh and Torus NoFPGA have the same wire length ($L_{w,\text{Mesh}} = L_{w,\text{Torus}} = L_{w,\text{noc}}$).

Next we will estimate the total power in the communication links for both Mesh and Torus NoFPGA as follows:

$$P_{\text{Link,Mesh}} = \alpha \cdot C_{\text{link}} \cdot l_{\text{dd}}^2 \cdot (2N - 2r)$$

$$P_{\text{Link,Torus}} = \alpha \cdot C_{\text{link}} \cdot l_{\text{dd}}^2 \cdot (2N)$$

$$P_{\text{Link,Diff}} = \alpha \cdot C_{\text{link}} \cdot l_{\text{dd}}^2 \cdot (2r)$$

Where:

$P_{\text{Link,Mesh}}$: The Total Mesh NoFPGA links power

$P_{\text{Link,Torus}}$: The Total Torus NoFPGA links power

$P_{\text{Link,Diff}}$: The Torus power overhead

$N$: The number of Nodes (for 3 x 3 NoFPGA, $N = 9$)

$r$: The number of node in each direction ($r \times r = N$), $r = 3$

The power overhead introduced by the Torus NoFPGA ($N = 9$, $r = 3$) is:
The Mesh NoFPGA experiences a 30% drop in power consumption when compared to the equivalent Torus NoFPGA configuration. This is a mathematical analysis of the power estimate of both Mesh and Torus NoFPGA. To have more accurate results, simulation based results are necessary, both analytical and experimental analysis should be as close as possible. The error between analytical and experimental results should be a reasonable value.

7.4 Put it all together

The Torus IPRouter has a higher communication link bandwidth compared to the Mesh IPRouter. As a result, the total bandwidth of NoFPGA built with Torus IPRouter is higher than NoFPGA built with Mesh IPRouter. The higher bandwidth and the lesser cost size of the Torus IPRouter comes at the expense of more power. Power estimate analysis showed that the Mesh NoFPGA represents a 30% power drop compared to the equivalent Torus NoFPGA. As a result, if the power dissipation is critical, the Mesh NoFPGA is a better candidate.
CHAPTER 8

NoFGPA TARGET APPLICATIONS

Current needs for high scalable and re-usable SoFPGA continue to grow. Such system will require integrating a set of IP cores and requiring a bandwidth of Gbit/sec, this comes as a result of multimedia applications on portable devices such as mobile phones and emerging consumer electronics products. Telecommunication applications have high bandwidth requirements with system scalability and re-usability is must. In this chapter we will give a high level description of Gigabit Ethernet over SONET (GbEOS), it is a telecommunication application that will utilize NoFPGA communication architecture.

8.1 Gigabit Ethernet over SONET (GbEOS)

The GbEOS transports Gigabit Ethernet (GbE) frames over existing SONET optical frame. It multiplexes four GbE traffic into a single SONET frame. This will
deliver a high Internet traffic over SONET optical infrastructure. The GbEOS is an end-to-end solution and interconnects the two networks together, the SONET and GbE networks. Figure 8.1 shows a block diagram of the building blocks of GbEOS system.

Figure 8.1 GbEOS Block Diagram
The ingress data flow is controlled by the Ethernet to SONET Multiplexer (ESMUX) IP core, it multiplexes the four Ethernet traffic into the SONET frame. On the other hand, the egress data flow is controlled by the SONET to Ethernet De-Multiplexer (SEDEMUX) IP core, it de-multiplexes the traffic from the SONET frame into four Gigabit Ethernet traffic and routes packets to their corresponding ports. Both the ingress and egress SDRAM are used to implement the flow control and provide the buffering of the four Gigabit Ethernet traffics in each direction. Figure 8.2 shows the mapping of the GbEOS onto 3x3 Torus NoFPGA.

Figure 8.2 Mapping of GbEOS on 3 x 3 Torus NoFPGA
Each Ethernet traffic represents 125 Mbyte/Sec. The SONET frame represents 500 Mbyte/SEC bandwidth requirements. Referring to table 6.3, the 3 x 3 Torus NoFPGA with 64-bit data link satisfies the GbEOS bandwidth requirements.
CHAPTER 9

CONCLUSION AND FUTURE WORK

This dissertation addresses the problem of low cost synthesizable scalable on-chip communication network for near future SoFPGA. The currently used Bus-Centered approach becomes an inappropriate choice because of its limitation as a shared medium that restricts the scalability of the communication architecture. Also, long bus wires result in performance degradation due to increased capacitive load. The long wires also consume more power to drive all of IP Cores on the bus. In this dissertation, a new communication architecture, the NoFPGA, for future SoFPGA has been presented. The following are the main contribution of this dissertation:

1. A low cost scalable architecture implemented in VHDL as a re-usable soft core is presented. The router is the basic building block of low cost NoFPGA. The router architecture address the interconnecting issues in SoFPGA design methodology built in a single FPGA. The communication interface has been designed to support the packet-
switched scheme so that the bus communication bottleneck is resolved. The router architecture results in a highly scalable NoFPGA that enables a re-use and easy integration of already available IP Cores.

2. FPGA is a 2D device in its nature. As a result, the most appropriate topology to apply is either a 2D Mesh or a 2D Torus. The two candidate topologies are studied and the cost associated with each topology is investigated.

3. This dissertation work investigates the best topologies for future SoFPGA. The cost synthesis results showed that the hardware overhead of 2D Torus experiences a 40% logic drop when compared to 2D Mesh. The drop in logic usage is due to the fact that Torus topology only routes packets in two directions while Mesh topology routes packets in four directions.

4. The buffer is the most highly expensive block of the router and it should be highly optimized by using available resources of FPGA, especially the Embedded Block RAM.

5. The cost synthesis of available opencores IP Cores showed that most of IP Cores fits between 18 Kgates and 35 Kgates. The Torus cost
represents a range of 4.2% to 17.5% of the total cost of the IP core. On the other hand, the Mesh cost represents a range of 7.1% to 26% of the total cost of IP Core.

6. The cost synthesis analysis of NoFPGA showed that the size of the Tile Architecture Block (TAB) could be chosen in the range of 40 to 50 Kgates.

7. Simulation results showed that the Torus topology has a higher communication link bandwidth than the Mesh topology. The total bandwidth of NoFPGA depends on the communication link bandwidth and the total number of links of network topology.

The current status of this dissertation has proposed a low synthesis cost IP Router architecture for near future SoFPGA and provides a packet-switching communication network. The floor plan of the FPGA is architected as an array of TABs. In addition, the dissertation opens some possible direction for future research interests as follows:

1. While the timing performance and synthesis cost analysis of NoFPGA has been addressed in this dissertation, the power consumption that is another critical issue for future SoFPGA has not been addressed. The
Torus topology has a higher number of communication links that results in higher power consumption when compared to the Mesh topology. The long wire data communications consume a considerable portion of the power in future SoFPGA. Further research can be done to quantify the power cost associated with both target topologies.

2. The NoFPGA architecture looks at FPGA floor plan as a set of TABs. As a result, Fault-Tolerant computing can be an incremental advantage of this architecture. Future research can address the issue of identifying bad FPGA TABs and remove them from the available pool of resources.

3. Having the best FPGA architecture is critically important but it has to be complemented with a new software CAD tool that aid in mapping of the applications onto the NoFPGA. The new CAD tools should match the new FPGA architecture. It should also allow dynamic partial re-configuration, this will deliver hardware re-use of the NoFPGA and provides a more generic hardware system.
REFERENCES


