Evaluation of FPGA-based Implementations of Interconnection Networks

Master Thesis by

Carsten Harms

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science (M.Sc.)

Supervisors:
Prof.Dr. Klaus Schneider
M.tech. Tripti Jain

University of Kaiserslautern
Embedded Systems Group

2016

Declaration

I hereby declare and confirm that the thesis at hand is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and are properly acknowledged. Furthermore, I hereby declare that this or similar work has not been submitted for credit elsewhere.

Kaiserslautern, ____________________ ______________________________

(Carsten Harms)

Abstract

The newly developed *Synchronous Control Asynchronous Dataflow* (SCAD) processor architecture exposes its processing elements and data paths to the compiler, allowing it to better exploit *Instruction-Level Parallelism* (ILP). The processing elements called functional units are communicating with each other via the *Data Transport Network* (DTN). Hence, this interconnection network is critical for the overall performance of the SCAD machine.

The main contribution of the thesis at hand was the implementation and evaluation of two different recent network topologies, namely the fat-tree and flattened butterfly network, considering the special characteristics of the SCAD architecture and the targeted *Field Programmable Gate Array* (FPGA). The very strict limitations in terms of resource utilization were satisfied and performance was optimized within the design space.

Contents

Abstract **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 2

1.3 Outline 2

2 Preliminaries **3**

2.1 SCAD Architecture 3

2.2 Network Topologies 5

2.2.1 Fat-Tree 5

2.2.2 Flattened Butterfly 7

2.2.2.1 2-Dilated Flattened Butterfly 8

2.3 Routing 9

2.3.1 Routing Problems 9

2.3.2 Flow Control 10

2.3.3 Virtual Channels 11

2.3.4 Routing on Fat-Tree 12

2.3.5 Routing on Flattened Butterfly 14

2.3.5.1 Dimension-Ordered Routing (DOR) 14

2.3.5.2 Valiant's Algorithm (VAL) 15

2.3.5.3 Non-minimal global adaptive routing (UGAL) 15

3 Implementation **17**

3.1 Target Device 17

3.2 Requirements 18

3.3 Control Unit 19

3.3.1 Move-Instruction-Bus 20

3.4 Functional Units 21

3.4.1 Input Queue 21

3.4.2 Output Queue 23

3.4.3 Branch Unit 24

3.4.4 Universal ALU 25

3.4.5 Load and Store Unit 26

3.4.6 Miscellaneous Functional Units 27

3.5 Network Interface 27

3.5.1 Splitter 28

3.5.2 Merger 29

3.6 Round-Robin Arbiter 29

3.7 Network Protocol 31

3.8 Fat-Tree Router 32

3.8.1 Input Port 33

3.8.2 Interconnection 34

3.9 Flattened Butterfly Router 36

3.10 Flattened Butterfly Router with Virtual Channel Support 37

3.10.1 Input Port 38

3.10.2 Interconnection 39

4 Evaluation & Analysis **42**

4.1 Network Performance 42

4.1.1 Traffic Patterns 42

4.1.2 Methodology 43

4.1.3 Analysis of Fat-Tree Performance 45

4.1.4 Analysis of Flattened Butterfly Performance 48

4.1.5 Direct Comparison 51
4.1.6 Impact of Network Size 53
4.2 FPGA Resource Utilization 54
4.2.1 SCAD Components 54
4.2.2 Interconnection Networks 55
4.3 Related Work 57

5 Conclusion **59**

Bibliography **61**

List of Figures

2.1 Organization of the functional units of a SCAD machine 4
2.2 Clos topology for eight nodes 6
2.3 Butterfly network for eight terminals 7
2.4 Demonstration of how virtual channels are utilized. 11
2.5 A fat-tree network for eight terminals 13
2.6 A flattened butterfly network for eight terminals 14

3.1 Basic block diagram of a complete functional unit 21
3.2 Finite state machine implemented by the Splitter control logic. 28
3.3 Finite state machine implemented by the Merger control logic. 29
3.4 Timing diagram of required behavior of a round-robin arbiter. 29
3.5 Router timing protocol without support for virtual channels 31
3.6 Simplified block diagram of a fat-tree router. 32
3.7 Connections between two routers. 33
3.8 Equivalent circuit schematic of a fat-tree router without flow control logic. 35
3.9 Simplified block diagram of inter-router connections of two routers with virtual channel support. 37
3.10 Finite state machine implemented by the flattened butterfly input port control logic with support for virtual channels. 38
3.11 Equivalent schematic of the flattened butterfly virtual channel router internals 39
3.12 Simplified block diagram of the router internals 40

4.1 Block diagram of the simulated functional unit. 44
4.2 Total Latency vs. Injection Rate on a Fat-Tree network using the worst-case traffic pattern. 46

4.3 Total Latency vs. Injection Rate on a Fat-Tree network using benign unified random traffic pattern. 46
4.4 Throughput vs. Injection Rate on a fat-tree network with two different flit sizes and traffic patterns. 47
4.5 Total Latency vs. Injection Rate on a flattened butterfly network utilizing the DOR algorithm under the worst-case traffic pattern. 49
4.6 Total Latency vs. Injection Rate on a flattened butterfly network utilizing the DOR algorithm on benign unified random traffic pattern. 49
4.7 Throughput vs. Injection Rate for the 32 terminal flattened butterfly (DOR) network with two different flit sizes and traffic patterns. 50
4.8 Latency vs. Injection Rate comparison for the 32 terminal networks on unified random traffic pattern. Flit size is 16 bits and the flattened butterfly network utilizes DOR. 52
4.9 Latency vs. Injection Rate comparison for the 32 terminal networks on worst-case traffic pattern. Flit size is 16 bits and the flattened butterfly network utilizes DOR. 52
4.10 Throughput vs. Injection Rate comparison for the 32 terminal networks on unified random traffic and worst-case traffic. Flit size is 16 bits. 53

Chapter 1

Introduction

1.1 Motivation

Recent research in the Embedded Systems department of the University of Kaiserslautern led to a new processor architecture called *Synchronous Control Asynchronous Dataflow* (SCAD) [BJS16]. This processors exposes its processing elements and the data paths to the compiler allowing it to better exploit instruction-level parallelism (ILP).

The SCAD architecture works in principle like a highly parallel queue automaton and consists of a great number of interconnected *Functional Units* (FUs), each implementing a specific function (e. g. branching, load-and-store or a simple addition). Universal functional units[1] are equally possible and can each be used to emulate a complete queue machine on their own. Every FU has a number of input and output queues. Depending on the implemented function of a particular unit, at least one to all input values are consumed and zero to arbitrary many output values are produced. This data-flow oriented design does not dictate *when* a unit is active, thus it is asynchronous. Afterwards, the produced output values must either be transferred to an input queue local to the unit or an input queue of another unit. Again, the model of computation between multiple units is asynchronous as well.

To realize the transport to remote units, all FUs are connected to each other via the *Data Transport Network* (DTN). To enable fast operation of the new processor architecture, this network should be as fast as possible while also using the least amount of resources. This is because the available resources on a *Field Programmable Gate Array* (FPGA) must be shared between the network and the rest of the

[1] that basically implement an extended arithmetic logic unit

SCAD implementation. Since both desired properties cannot possibly be fulfilled at the same time, certain tradeoffs are necessary.

The main focus of the thesis at hand is to explore several possible interconnection network implementations. Important design decision are discussed in great detail. Additionally the feasibility, performance and resource requirements of the implementations are evaluated and recommendations regarding what network to chose under which circumstances are given.

1.2 Contribution

Because research for the motivated SCAD architecture is still on-going, the first contribution of this thesis is a prototype implementation of the SCAD for 32 functional units operating on 32-bit data. The main contribution consists of implementing two different suitable interconnection networks fulfilling the requirements imposed by the SCAD machine. Finally, theses network implementations are extensively evaluated considering both performance and resource utilization.

1.3 Outline

The remainder of this thesis is organized as follows: Section 2 introduces required network basics such as implemented topologies and routing methods. Section 3 discusses in great detail the implementation of all SCAD machine components with a larger focus on the interconnection network. Section 4 evaluates the implemented interconnection networks in terms of performance and resource demands and compares the obtained results with related works. Finally, Section 5 concludes the thesis.

Chapter 2

Preliminaries

At first, this chapter introduces the motivated architecture of the SCAD machine in more detail. Next, the implemented network topologies and routing algorithms are discussed. Additionally different routing modes and typical problems as well as their solutions are explained.

2.1 SCAD Architecture

As shortly introduced in Section 1.1, the SCAD machine consists of many interconnected FUs. These are organized as illustrated in Figure 2.1. Each queue is a *First-In First-Out* (FIFO) buffer which store tuples in the form of $(address, data)$. Input queues store the address of the source output queue in order to match incoming data to the corresponding entry. Output queues store the address of the target input queue in order to ultimately transfer data to the destination.

To guarantee a sequential program flow, a central *Control Unit* (CU) issues move-instructions (src, tgt) over the *Move-Instruction Bus* (MIB). This bus tells the corresponding queues that data from the source must be transferred to the target. To ensure the correct order, both addressed queues push a new tuple if possible, otherwise the CU is notified that the current instruction must be repeated. As mentioned previously, the input queue adds the tuple $(src, \perp)$ and the output queue adds $(tgt, \perp)$ where $\perp$ denotes an empty value which will be filled at a later point in time.

While input queues always contain tuples either in the form of $(src, \perp)$ or $(src, data)$, output queues are additionally able to contain tuples in the form of $(\perp, data)$. These can occur whenever the processing unit of the FU produces more data than there are recipients currently. For example, a duplicator can be

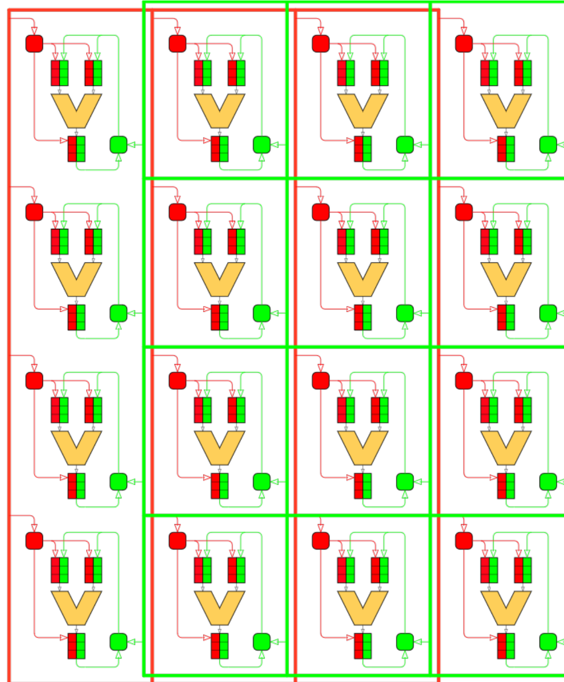


Figure 2.1: Organization of the functional units of a SCAD machine (Source: [BJS16]).

ordered to produce ten copies. Not all of these must have a valid target address. Those that do not have one can receive a destination at a later point in time.

Tuples from the DTN have the form $(tgt, src, data)$. The input queue addressed by tgt matches all buffer entries $(src, \bot)$ with $(src, data)$ beginning from the front. The matching process stops as soon as the first match is found. Completed entries are considered valid and are ready for consumption (popping the entry of the queue).

The connected processing unit may consume valid data *asynchronously* at any time. While enough space in the output buffer(s) is not necessarily required, it is usually encouraged. One exception is the duplicator, which can be issued to produce more copies than can physically fit in the output queue. Similar to input queues, valid entries of the output queues can arbitrarily be consumed by the DTN. Thus, the interconnection network of the SCAD machine is also asynchronous.

2.2 Network Topologies

There are many different network topologies proposed in the literature which range from the well-known mesh, torus, Clos and butterfly explained in [DT04] to lesser known topologies like the Octagon published in [KNDR01].

There are also many comparisons of the published topologies concerning performance or resource requirements. After careful consideration the two most promising topologies, fat-tree and flattened butterfly were chosen for this thesis.

To provide as much bandwidth as possible while also minimizing the required resources, only two functional units are connected to each router. Connecting every functional unit to its own router would deliver the best performance at higher resource cost. Although connecting more units to a single router increases the local bandwidth, all connected units share the limited bandwidth between routers. Doubling the amount of terminals halves the theoretical bandwidth for each router. Hence, only binary router connections are considered. The remainder of this section will discuss the selected topologies and why they were chosen in detail.

2.2.1 Fat-Tree

The fat-tree (or folded-Clos) topology was first published in 1985 by Leiserson in [Lei85]. It was designed for a "a parallel computation engine composed of a set of processors interconnected by a routing network" which used that interconnection network to exchange data. At that time it was meant for super-computers, but modern technology made it possible to place multiple processors and the network[1] into single *integrated circuit* (IC) called System-On-Chip (SoC). However, the underlying principle of the network still holds today.

The binary fat-tree is similar to a normal binary tree. The only difference is that it gets wider (or *fatter*) while traversing up the tree. The reason for this is that the nodes closer to the root node must accommodate for the complete traffic to and from it's children nodes. The leaves of the fat-tree are connected to terminals, or in the case of SCAD the functional units.

The fat-tree topology is obtained by *folding* the output stage of a Clos network on top of the input stage along the dashed mirror line shown in Figure 2.2. The originally unidirectional connections then become bidirectional. Additionally, the inputs and their corresponding outputs are placed on the same

[1] Also called Network-On-Chip (NoC)

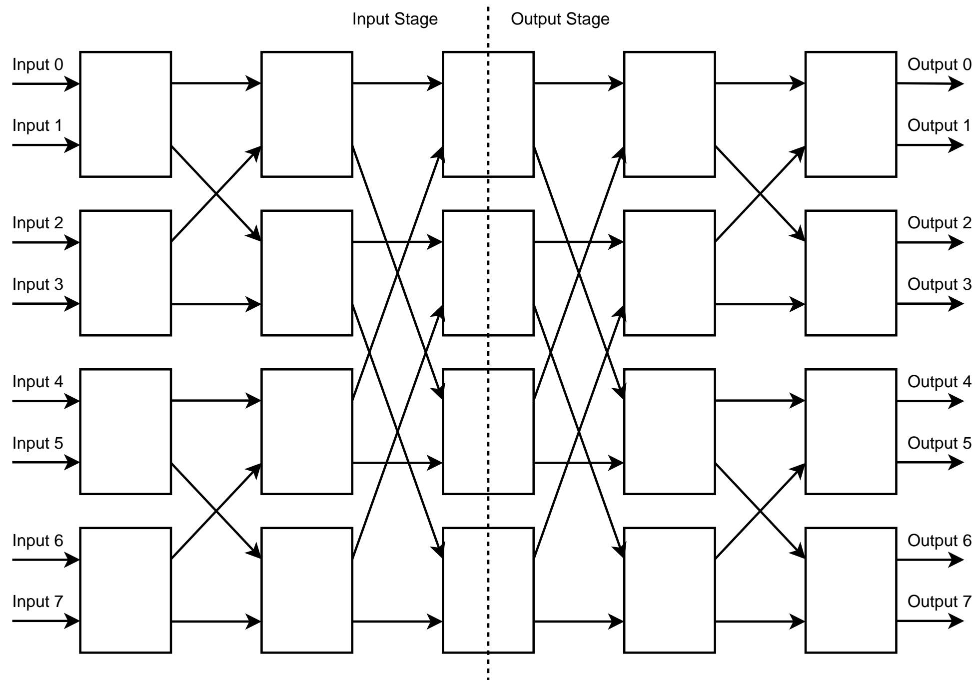


Figure 2.2: Clos topology for eight nodes. Note that the output stage is simply the mirrored input stage.

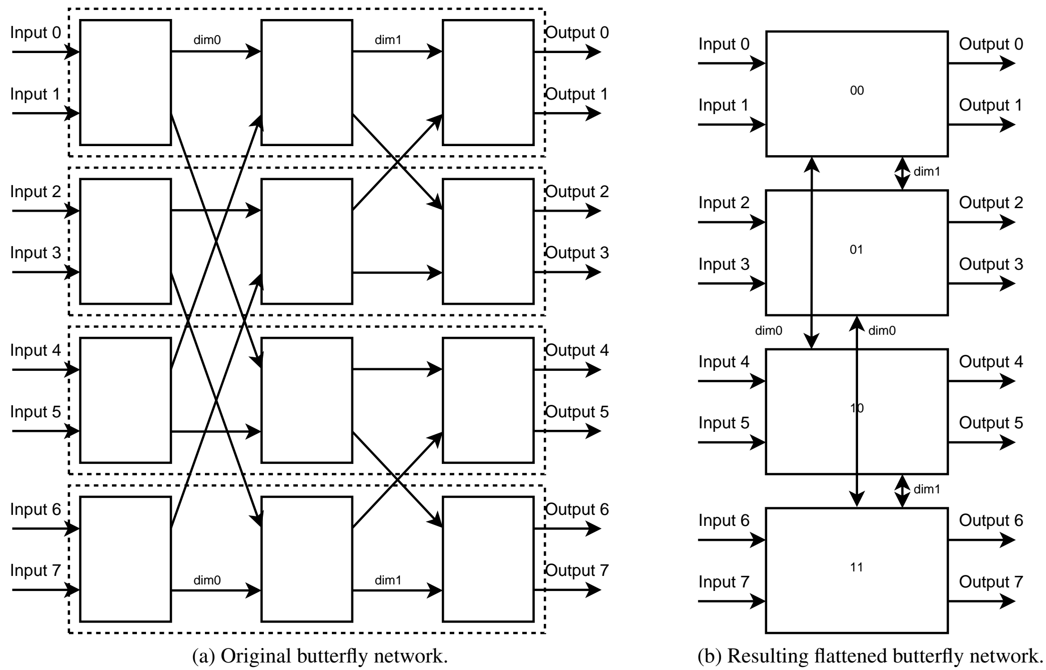
routers.

Note that the root-nodes created by the aforementioned folding (routers along the dashed line) can be omitted because these do not contain any logic. Instead, all input and output ports are wired directly to the corresponding output and input ports. The remaining routers of the fat-tree each have four bidirectional ports. Internally, the connections are incomplete, meaning that not every input port is connected to every output port. Connections that can be omitted are discussed in Section 2.3.4.

The main advantage of the fat-tree topology is it's *non-blocking* nature. While in theory it provides enough physical channels to enable all processors to send data to another unique processor at the same time. However, this non-blocking property requires rearrangement of established routes depending on other routes. Unfortunately, rearrangement during the routing process is unfeasible due to the used protocol which will be explained in Section 3.7. Another advantage of this topology is the existence of multiple paths (*path-diversity*) and additionally the existence of multiple minimal paths.

The main disadvantage of the fat-tree topology is the large worst-case hop-count[2] which leads to high latency. In the worse-case, a message must travel through the height of the tree twice to reach

[2]Passing through a router is commonly called hop.



(a) Original butterfly network.

(b) Resulting flattened butterfly network.

Figure 2.3: Butterfly network for eight terminals. The routers enclosed by the dotted rectangles in (a) are flattened into one corresponding router per row (b).

the destination. As the network size increases, the height of the fat-tree grows logarithmically. Another disadvantage of this topology is its high resource requirements. Figure 2.5 in Section 2.3.4 depicts a very small network with only $N = 8$ terminals where already 8 routers are necessary. As the network size N increases linearly, the number of necessary routers increases exponentially.

Ultimately, the deciding factor was Leiserson's proven theorem that "for a given physical volume of hardware, no network is much better than a fat-tree." As motivated for this thesis, resource efficiency is very important considering the goal of utilizing this topology as the interconnection network for the SCAD machine on a FPGA.

2.2.2 Flattened Butterfly

The flattened butterfly topology published by Kim et. al in [KDA07] and [KBD07] improves the well-know butterfly topology by providing better path-diversity. Instead of routing a network packet through a fixed order of dimensions imposed by the butterfly topology, the flattened butterfly topology grants freedom of choosing the order by the routing algorithm. Figure 2.3(a) depicts a general binary butterfly network. For example, assume that terminal zero wants to transfer data to terminal six. First, the data must

be routed within the first dimension (*dim0* in the figure) and then within the second dimension (*dim1*) to reach the destination. In contrast, the flattened butterfly network in Figure 2.3(b) allows the router `00` the choose in which dimension the data is routed first.

Figure 2.3 also illustrates how the topology is obtained. Basically, all routers of a single row of the butterfly (framed by the dashed rectangle) are *flattened* into a single router. The two unidirectional connections between the previous rows are kept, resulting in a single bidirectional connection between two flattened butterfly routers.

The main advantage is the small amount of necessary routers. To support N terminals, only $n/2$ routers are required when considering the initially discussed reason to limit each router to exactly two terminals. The small amount of routers also leads to another advantage: the hop-count and therefore the expected latency is significantly lower compared to the fat-tree network discussed in Section 2.2.1.

Disadvantages of this topology are twofold. For one, flatting many routers into a single router increases the complexity and therefore the amount of required resources per router. This is especially apparent when considering not only two dimensions as shown in the simple example but four which are necessary for the interconnection of a SCAD machine with 32 functional units. Secondly, the flattened butterfly topology—just like the original butterfly topology— is blocking. For example, consider that terminals zero and one both want to transfer data at the same time to terminals four and five. It is obvious that the single connection cannot be used by both at once. Section 2.2.2.1 shortly discusses a variant of the flattened butterfly network that solves the blocking problem.

In the end, similar to the fat-tree decision, the claimed performance and resource efficiency led to choosing the flattened butterfly topology as a suitable interconnection network for the SCAD machine.

2.2.2.1 2-Dilated Flattened Butterfly

The 2-dilated flattened butterfly [TC10], [TC11] is a double wide variant of the previously discussed flattened butterfly. All physical inter-router channels are doubled to provide enough bandwidth to be a *non-blocking* topology[3]. While the non-blocking property is very desirable, it has some huge disadvantages, especially when the limitations imposed by the goals of this thesis are considered. First of all, doubling every inter-router channel almost doubles the amount of necessary buffers for every router. Furthermore, in order to exploit the second channel, more complex routing algorithms have to be used which require even more resources (see Section 2.3.5). Given the very limited resource of the targeted FPGA,

[3] Channels to processors are not doubled

implementing this variant appeared to be impossible.

2.3 Routing

In general, the task of a routing algorithm is to find a *path* from one terminal of any given network topology to any other terminal. The algorithms can be classified into the following categories:

Oblivious: The routing decision does *not* depend on any additional information such as network congestion. Decisions are solely based on the routing information contained within the network message itself. Clearly, this category contains the least complex algorithms both in terms of logic (performance) and resources.

Locally Adaptive: The routing decision uses aforementioned additional information, but only for its *direct neighbors*. Algorithms in this category not only are more complex logically, but additionally are more prone to problems that will be discussed in Section 2.3.1.

Globally Adaptive: The routing decision uses aforementioned additional information from *all nodes* of the network (or a subset larger than only direct neighbors). Unsurprisingly, the complexity and necessary resources are even higher while also sharing the same problems.

In theory, adaptive algorithms should perform better compared to oblivious ones. However, the cost-to-implement may either not be worth it or even prevent implementation in for example entry-level FPGAs.

Another distinction of routing algorithms is whether they consider only *minimal* paths or additionally look at *non-minimal* paths. Non-minimal paths have the chance to load-balance the traffic across the network better. See Section 2.3.5.2 for an example of a non-minimal algorithm.

Concerning this thesis, only algorithms which are both applicable to the topologies introduced in Section 2.2 and which are viable for implementation are discussed further.

2.3.1 Routing Problems

For networks in general, there are some problems which require the attention of the designer. As such, this sections shortly introduces these:

Deadlock: The situation where multiple routers try to access the same shared channel and are blocking each other *permanently* in a cyclic dependency (confer the dining philosophers problem). In general, there are two possible solutions. The first solution is to avoid deadlocks entirely by guaranteeing dependency graph to be cycle-free. The other solution is to detect the occurrence of a deadlock and break it by dropping the causing messages.

Livelock: The situation where *no progress* towards the destination is made while the message is actively routed through the network. This can only occur using non-minimal routing algorithms. It can easily be solved by limiting the number of non-minimal routing decisions.

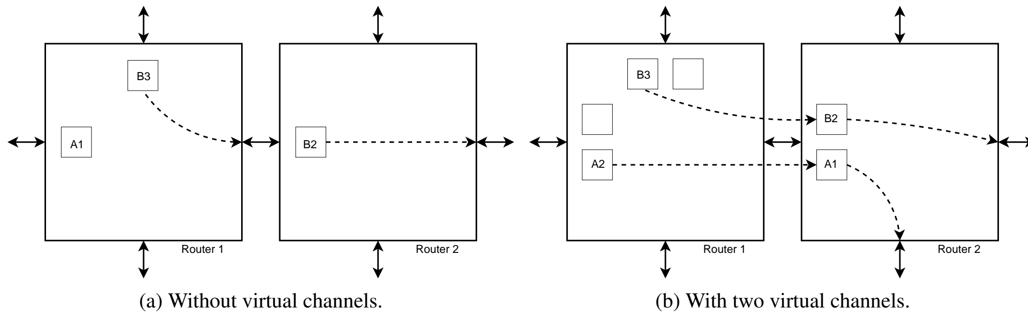
Starvation: The situation where some messages are permanently blocked by other messages, possibly higher priority ones. The simplest way of solving this issue is by using a fair allocation algorithm.

2.3.2 Flow Control

The flow control (or routing mode) of a network specifies how messages are send from and received by routers in a way that ensures not overloading the input buffers. To this end, receiving input buffers need a mechanism to signal their buffer status to the sender.

In the following, only *packet-switching* routing modes are discussed. In general for packet-switching, *messages* are split into one or more *packets*. Depending on the chosen flow control method, these packets in turn are split further into equally sized small *flits*(flow control digits). Flits may be even further split into *phits* (physical transfer units). For the remainder of this thesis, message and packet will be used interchangeably since all messages send through the interconnection network of the SCAD machine are equally sized and comparatively small. Additionally, flits have the same length as phits. Further implementation details are discussed in Chapter 3.

Store-and-Forward Routing As the name suggests, this routing mode requires a router to receive and *store* all flits of a packet completely before it is allowed to *forward* it to the next router. The first advantage of this mode is its simplicity. An additional advantage is that at most one channel within the whole network is used at a time by one packet which makes deadlock avoidance easy. However, this mode suffers from high resource requirements, because the complete message has to fit inside the router's buffer. Additionally, the latency is very high due to the long waiting period where no progress towards the destination is made.



(a) Without virtual channels.

(b) With two virtual channels.

Figure 2.4: Demonstration of how virtual channels are utilized.

Virtual Cut-Through Routing Improving on the latency issue of the store-and-forward mode, routers must no longer wait until the complete packet is inside the buffer. Instead, they are allowed to forward the flits as soon as the receiving router's buffer is ready. While a message can span multiple routers at the same time as it traverses through the network, it can be stored completely inside a router to free the used channels. Unfortunately, the buffer size remains the same as in store-and-forward routing.

Wormhole Routing Further improvements towards smaller buffer sizes lead to the development of this routing mode. Routers are no longer required to store every flit of a packet. Instead storing only a few flits down to one single flit is possible. While obviously reducing the necessary buffer sizes, this routing mode allocates multiple channels for a single packet, which makes it very prone to deadlocks. To solve this issue, virtual channels (see Section 2.3.3) have been introduced.

The high resource requirements from both store-and-forward and virtual cut-through routing make them unfeasible for the implementation on an FPGA. Thus, wormhole routing with a buffer size of exactly one flit has been used in the implementation and will only be considered in the remaining part of this thesis.

2.3.3 Virtual Channels

Virtual channels not only help avoiding deadlocks [DS87] [DA93], but also help to increase the performance of a network by enabling *bypassing* of blocked messages [Dal92]. Figure 2.4 demonstrates this with a simple example. Two arbitrary routers are connected via a single bidirectional connection (channel) and utilize the previously discussed wormhole routing with atleast three flits per message. The first

router receives one message A from its western channel and another message B from its northern channel. Assume both messages must be routed through the second router. Further, assume message B is now blocked in the second router because its destination in the east is full.

Without virtual channels (Figure 2.4(a)) the message B blocks message A since it was received at an earlier point in time[4]. However, with virtual channels (Figure 2.4(b)) the message A is able to bypass message B since it is destination is further south. Similar, virtual channels allow to avoid the cyclic dependency causing deadlocks simply by providing more resources (virtual channels) to the routing algorithm.

In order to implement virtual channels, a lot of additional logic is necessary. Apart from requiring new buffers at each input port, a demultiplexer between the real input port and buffers is necessary to distinguish the virtual channels. This, however, also requires sending identification information alongside the flit which increases the necessary channel bandwidth. Both the router-internal interconnection and arbitration is more complex due to the increased channel count.

2.3.4 Routing on Fat-Tree

The routing on a fat-tree can be divided into two separate phases:

Up-routing phase: In this phase, the network message is routed towards the (non-existential) root of the tree until the message arrived at the first node from where the target terminal is reachable (*common ancestor*). Routing upwards any further would result in a non-minimal path. As mentioned in Section 2.2.1, fat-trees contain multiple minimal paths. This grants freedom for the routing decisions during this phase.

Down-routing phase: In this phase, the network message is routed towards the destination leaf of the tree, starting from the node where the up-routing phase ended. In this phase, the routing decision is completely deterministic as there is no freedom anymore.

Figure 2.5 illustrates a fat-tree routing network supporting eight terminals. Suppose terminal `001` wants to transfer data to terminal `100`. Then there are four different minimal paths for the up-routing phase: *left-left*, *left-right*, *right-left*, *right-right*.

Individually, each router input is able to compute to which output port a message must be passed. In Figure 2.5 the router labels contain the configuration consisting of address and number of significant bits.

[4] also identifiable by the flit number next to the message letter

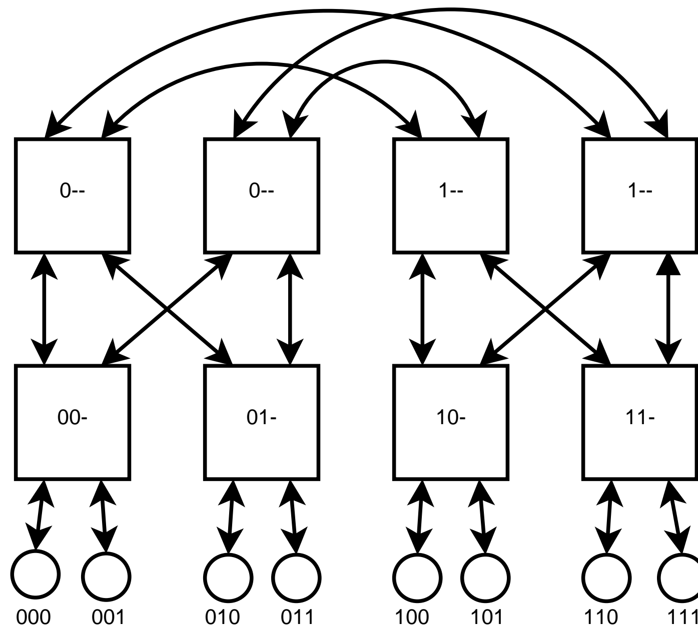


Figure 2.5: A fat-tree network for eight terminals. The bit-vector labels of the routers depict their configuration where - implies *don't care*.

Depending on the height of the router more or less bits are considered (*significant*). Going back to the example, router `00-` compares its own address with the destination address `100`. Since the addresses do not match, it is routed upwards. The same is true for any of the `0-` routers. After the message reached the `1-` router, the addresses match and the down-routing phase is entered. As already explained, routing here is completely deterministic and can be deduced from the first non-significant bit of the target address. In the example it is zero and as such the message is routed downwards to the left. In case of a one, it would have been routed downwards to the right. Likewise the last router on the path of the message routes as described above and the destination is reached.

Kim et al. have studied in [KDDA06] adaptive routing on high-radix fat-trees in great detail. They compared different variations of the greedy (locally adaptive) and sequential[5] (globally adaptive) algorithm in terms of performance and cost. They concluded, that the greedy_r(2)[6] variant performed only slightly worse than the sequential algorithm with limited buffering while being less complex. On a low-radix fat-tree, such as the one that was implemented for this thesis, they report almost 100% throughput. For this reason, only the greedy algorithm was implemented in addition to a deterministic, oblivious one.

[5]Routing decisions are made one after another and later decisions are based on previous ones

[6](only 2 valid, randomly picked routing decisions were offered)

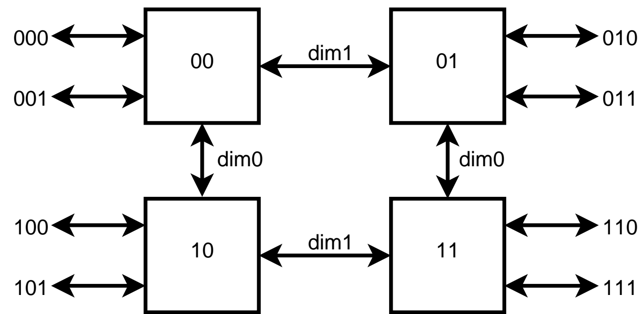


Figure 2.6: A flattened butterfly network for eight terminals. The bit-vector labels of the routers depict their address.

However, the achieved results were not convincing and thus the implementation was scrapped.

2.3.5 Routing on Flattened Butterfly

While there are dozens of routing algorithms for the flattened butterfly topology in the literature, this list is heavily restricted by the requirements of network size (32 nodes) and the resources of the targeted FPGA. Thus, while appearing relatively simple to implement on the first glance, more sophisticated routing algorithms than the chosen ones require either sequential logic that would slow the routing process disproportionately down[7] or need too many resources for deadlock avoidance. For example, the number of virtual channels necessary for a *simple* greedy algorithm on a flattened butterfly equals the number of dimensions between the routers.

For an example consider Figure 2.6. Suppose terminal `101` wants to transfer data to `010`. In order to find a (not necessarily free) path to the destination, the router simply applies a bitwise `XOR` of the destination address and it's own address. The resulting difference bit-vector (here `11`) represents the possible dimensions which lead closer to the destination. As mentioned in Section 2.2.2, the flattened butterfly topology allows routing in any order of dimensions. However, enabling such freedom also requires special care to avoid deadlocks.

2.3.5.1 Dimension-Ordered Routing (DOR)

The simple dimension-ordered routing algorithm is a representative of oblivious and minimal routing algorithms. As the name suggests, a network packet is routed towards the destination in any *fixed* order.

[7] The message sizes are really small compared to other packets, for example internet traffic.

Fixing the order from most significant bit to least significant bit results in a path from `10` $\rightarrow$ `00` $\rightarrow$ `01` in the previous example.

The advantage of this algorithm is its simplicity both in required logic and other resources. It is deadlock free without using virtual channels. However, the disadvantage is the poor utilization of the network as the path diversity offered from the flattened butterfly topology is not fully exploited.

2.3.5.2 Valiant's Algorithm (VAL)

Valiant's algorithm is a representative of oblivious non-minimal routing algorithms. It improves upon the poor network utilization of DOR and load-balances network traffic across the whole network. This is accomplished by splitting the routing process into two phases:

Phase 1: In the beginning of this phase, an *intermediate* router is (quasi-)randomly picked. The network packet is then routed to the intermediate router using any minimal routing algorithm, here DOR.

Phase 2: Starting from the intermediate node, the packet is routed minimally to the destination.

The resulting route including phases may be non-minimal, because the choice for the randomly picked intermediate router is unrestricted. Suppose the terminal `000` wants to transfer data to terminal `010`. The randomly selected (with a chance of $1/4$) intermediate router is `11`. Using DOR as the minimal routing algorithm, the resulting non-minimal path is `00` $\rightarrow$ `10` $\rightarrow$ `11` $\rightarrow$ `01`.

While routing through a random intermediate node load-balances the network, there are disadvantages: Both best- and worst-case hop-counts are increased, which in turn increases latency. Furthermore it does not exploit locality. Two neighboring nodes could be sending their data through a distant intermediate node, even if the direct channel would be free.

The variant used in this thesis is deadlock free with exactly two virtual channels. Each phase uses a unique virtual channel and since the DOR algorithm is used to route through the network, no further virtual channels are necessary to guarantee freedom of deadlocks. Valiant's algorithm is also livelock-free, because the number of mis-routings is bound by the dimension count of the network.

2.3.5.3 Non-minimal global adaptive routing (UGAL)

UGAL [Sin05] is a representative of a globally adaptive non-minimal routing algorithm. Depending on the global network state, the algorithm decides whether it is more efficient to route non-minimally via

an random intermediate node or minimally directly to the destination. Similar to VAL, routing to and from the intermediate router is accomplished by another minimal routing algorithm. Depending on the choice of the used routing algorithm, the calculation whether it is actually beneficial to route non-minimally can be very complex. Choosing the earlier discussed DOR algorithm, computation still involves determining the expected latency for both the non-minimal and minimal path and comparing them. This requires total knowledge of all buffer states, which is increasingly costly for larger network sizes.

Similar to VAL, UGAL used in conjunction with DOR requires two virtual channels to be deadlock free. It is also livelock-free for the same reasons as VAL. While this algorithm was originally planned to be implemented, the previously discussed cost made implementation unfeasible.

Chapter 3

Implementation

This chapter discusses the implementation of all SCAD machine components in great detail. First, the target device and its operation is introduced. Next, the requirements are discussed. Last but not least, the implementation of every component, their sub-components as well as their connections among themselves is described.

3.1 Target Device

The targeted prototype platform is the AVNET™ZedBoard™[1]. It contains a Xilinx Zynq®-7000 All Programmable System-on-Chip, which in turn contains an Artix®-7 *Field Programmable Gate Array* (FPGA) equivalent. This FPGA provides $53,200$ Lookup Tables (LUTs) to realize any combinational logic function. Utilizing the additionally provided $106,400$ Flip-Flops, any suitable sequential logic function can be realized.

To save flip-flops resources for larger memory instances such as a scratch-pad, the FPGA also contains $140 * 36$ kilobit dual-port block RAM blocks. The dual-port functionality allows two independent operations at the same time on the same block.

The chosen hardware description language is VHDL. The source code only describes the register-transfer-level behavior and is applicable for the implementation on a FPGA for prototyping purposes or on an *Integrated Circuit* (IC) for production. The synthesizer of the Xilinx Vivado™2015.3 is used to generate a bit-stream configuration from the VHDL description of the implemented SCAD machine for

[1] `http://zedboard.org/`

the aforementioned FPGA.

3.2 Requirements

The available resources of the aforementioned target FPGA must be partitioned to fit both every non-networking SCAD component with the necessary network interfaces and the routing network itself. The implementation of the interconnection network should not use more than roughly 15% of the available resources. This bound was determined in an early development phase after implementing the addition functional unit[2] and extrapolating its resource requirements to a SCAD machine with 32 units.

The requirements of the interconnection network are as follows: It is of highest priority that no packet-loss occurs. Each packet send through the network is expected to arrive at the destination and no acknowledgement or retransmission protocols are necessary. To fulfill this requirement, the network must be fair and free of deadlocks, livelocks and starvation. Deadlock *avoidance* is necessary because when a deadlock happened, the only solution would be to drop the packets causing the issue. However, this would violate the no packet-loss requirement.

When work was started on the SCAD implementation in the beginning of this thesis, exactly 32 non-universal functional units with at most four queues each were planned. This meant, that all addresses were seven bits wide, composed of $log_2(32) = 5$ bits for addressing the unit itself and $log_2(4) = 2$ bits for addressing the queue within the unit. These addresses fit in a 8-bit flit with one reserved bit. This bit is later used for the implementation of VAL (see Section 2.3.5.2).

In order to transfer the complete triple *(target, source, data)*, the partition depicted in Table 3.1 is used. The target address needs to be contained in the first flit (*header flit*) to enable routing. Flits 2–6 are only *payload* flits, the ordering here is arbitrary. Since all triples have exactly the same size, the number of flits is fixed and implicit. Doubling the flit size to 16 bits allows for transferring two 8-bit flits at the same time. Pairing (#1, #2) as the first 16-bit flit[3] requires only adjusting the expected flit count in the router implementations.

However, further development in the Embedded Systems department decreased the required number of functional units to just eight. In return, the number of addressable queues increased. With only $log_2(8) + 1 = 3$ bits used, this provides room for up to $2^4 = 16$ input queues as well as 16 output queues, because a target queue can only be an input queue, while a source queue can only be an output queue. As

[2] which is one of the least complex functional units

[3] followed by (#3, #4) and (#5, #6)

Table 3.1: Partition of any triple for the SCAD machine into 8-bit flits.

#	flit[7:3]	flit[2:1]	flit[0]
1	target.unit	target.queue	0
2	source.unit	source.queue	0
3	Data[31:24]		
4	Data[23:16]		
5	Data[15:8]		
6	Data[7:0]		

such, smaller DTN versions with support for exactly 8 and 16 functional units were also implemented. These versions are also interesting to determine the scaling of these networks.

3.3 Control Unit

The control unit (CU) is the heart of the SCAD machine. It contains the program counter (PC) and controls the Move-Instruction-Bus (MIB). The instructions stored within the instruction memory of the CU are processed sequentially. When an instruction targets the branch unit, the CU enters the *branch mode*. While in this mode, further instructions are processed until an instruction is encountered which does not target the branch unit. In terms of program code, this marks the end of the branching operation. As such, the CU halts operation (*stalls*) until it is allowed to continue (*unstalled*) by the branch unit. While the CU is stalled, the branch unit is allowed to set the program counter to a new value, from which the normal operation will continue upon unstalling.

Instructions are only move-operations and describe the intended transport of data from an output queue (source) to an input queue (target). There are two different types of instructions: normal and immediate. In case of an immediate instruction, the source queue is implicitly the CU. Move-instructions are formatted according to Table 3.2 and are ordered for simple processing (inspired by RISC instructions).

To support *immediate* instructions, the control unit also contains a *feeder* component. It consists of a normal output queue[4] and a single network splitter (see Section 3.5.1) to connect to the interconnection network.

[4]Optimization is possible here, since data and addresses always arrive at the same time.

Table 3.2: Move instruction format (40 bits).

Type	Opcode	Target	Remainder
Normal	0	Target address (7 bits)	Source address (7 bits)
Intermediate	1	Target address (7 bits)	Intermediate data (32 bits)

Table 3.3: Signals included in the move-instruction-bus. Directions are from CUs point-of-view.

Signal	Direction	Description
`valid`	out	Signals whether any other bits are valid or must be ignored.
`source`	out	The address of the source queue of the current instruction.
`target`	out	The address of the target queue of the current instruction.
`repeat`	out	A flag that signals whether the current addresses of the MIB are repeated because any queue signaled `full` in the previous clock cycle.
`source_full`	in	Feedback signal from the addressed source queue whether it is full.
`target_full`	in	Feedback signal from the addressed target queue whether it is full.

3.3.1 Move-Instruction-Bus

The Move-Instruction-Bus (MIB) contains the signals described in Table 3.3. Despite its name, it is not implemented as a traditional bus, since data flows in both directions at the same time. The `full` feedback signals described as a bus in VHDL. All input queues are connected to the same signal, but only drive the signal `HIGH` when they are targeted. Otherwise their output floats (`'Z'`). Although FPGAs in general only have floating capable ports to the outside of the chip, the described behavior is translated by the synthesizer to simply `OR`ing the corresponding signals together.

The repeat signal is necessary, because the full-signal from the addressed queues arrives one clock cycle later. Because this is clearly too late, the corresponding queues must keep track whether they were full last cycle in order to determine whether they must ignore the MIB this clock cycle or not.

Another way of solving this issue would be by enabling the CU to check each individual full-state of every single queue. Worst-case scenario are 31 functional units with four queues each, resulting in a $31*4 = 124$-bit input at the CU. Then the states can be checked in time and new instructions are send over the MIB only when both addressed queues are ready. Which solution is better in terms of performance and area was not further tested. The implemented method was a design choice early on and optimization in non-interconnection related components is not within the scope of this thesis.

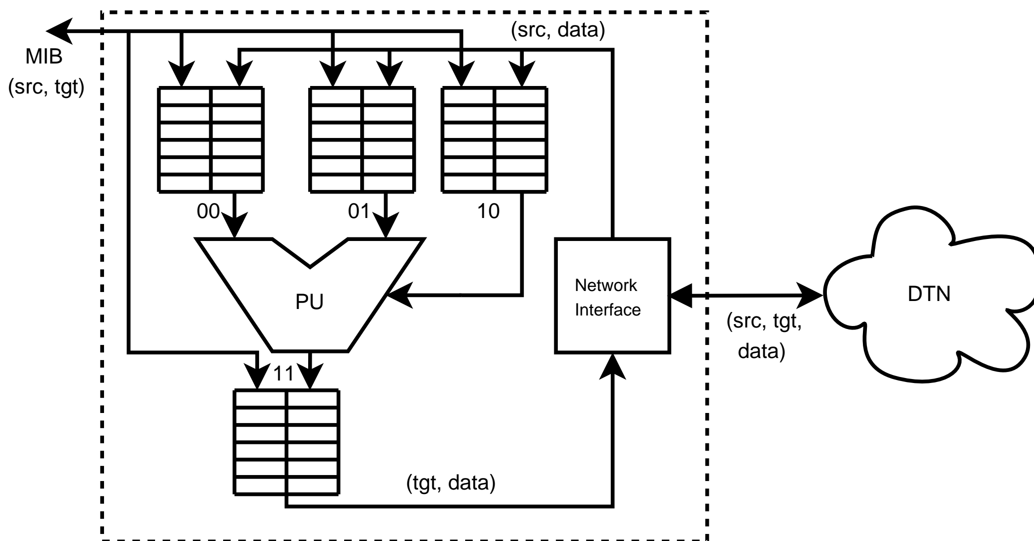


Figure 3.1: Basic block diagram of a complete functional unit with three input ports `00`, `01` and `10` and a single output port `11`. Only data flow and MIB are shown, other signals omitted for readability.

3.4 Functional Units

Functional Units (FUs) are responsible for processing data. They consists of potentially arbitrarily many input queues, arbitrarily many output queues, one processing unit and a network interface. Figure 3.1 depicts the restricted older version of functional units with at most three input queues and at most one output queue.

The task of a functional unit is to the consume valid data present the head(s) of the input queues and process them internally. After processing the input data, the processing unit may produce output data which can then be transferred over the DTN to any other input queue. These include the input queues of the functional unit where the processing unit is a part of.

3.4.1 Input Queue

An input queue entry consists of two separately filled arrays, source address and data, as well as their corresponding valid flags. Two separate flags are necessary, because at first, the source address field is filled via the MIB. At least one clock cycle later, the data fields of such incomplete but not empty entries are filled via tuples *(source address, data)*.

Realizing the required behavior in hardware is more complex than a traditional First-In First-Out (FIFO) queue. While the addresses incoming from the MIB are stored in the FIFO way, the incoming

Table 3.4: Input queue values over six clock cycles during different operations.

(a) Incoming address from MIB

#Entry	Address	V	Data	V
1 (H)	(2,3)	1		0
2		0		0
3		0		0

(b) Incoming address from MIB

#Entry	Address	V	Data	V
1	(5,3)	1		0
2 (H)	(2,3)	1		0
3		0		0

(c) Incoming tuple $((5, 3), 42)$ from DTN

#Entry	Address	V	Data	V
1	(5,3)	1	42	1
2 (H)	(2,3)	1		0
3		0		0

(d) Incoming tuple $((2, 3), 24816)$ from DTN

#Entry	Address	V	Data	V
1	(5,3)	1	42	1
2 (H)	(2,3)	1	24816	1
3		0		0

(e) Consume head

#Entry	Address	V	Data	V
1 (H)	(5,3)	1	42	1
2	(2,3)	0	24816	0
3		0		0

(f) Consume head

#Entry	Address	V	Data	V
1	(5,3)	0	42	0
2	(2,3)	0	24816	0
3		0		0

data tuple with the source address is not. It is matched with the address fields of incomplete entries until the first match is found starting at the head entry within one clock cycle.

To realize the required behavior in hardware, the tail of the FIFO is always the first array element while the head pointer is allowed to move. Entries above the head must be invalid. Table 3.4 demonstrates this with the aid of an example. The reason for this is that basic software loops with varying bounds are not synthesizable. Thus the matching process must start at the last entry of the queue and ends at the first entry (tail). The head entry is somewhere in between, but since all entries above the head must be invalid, it is guaranteed the the actual matching starts at the head of the FIFO.

The input queue signals `full` to the control unit, when two conditions are met: First, the unit must be addressed as the target of the current move-instruction and secondly, the queue's header position already points to the last entry of the array.

Mapping the implementation to the block RAM of the FPGA is not feasible. The use of multiple

($|queue| > 2$) read/write operations per clock cycle prevents the use of available dual-port block RAM. Independent of that issue, it would also be a very inefficient usage of block RAM resources, which are minimally 36 kB in size.

The number of queue entries per queue is configurable via a VHDL generic and defaults to six. This enables optimal usage of available FPGA resources, depending on the remaining configuration of the SCAD.

3.4.2 Output Queue

Output queues are placed downstream of the processing unit and operate asynchronously. The processing units output is controlled by an `output_enable` signal which is only `HIGH` when there is space for new data.

The main difference between input and output queues is the lack of needing to match previously stored addresses with the incoming tuples. However as already shortly mentioned in Section 2.1, it is possible for a queue entry to only have valid data but not yet have a target address: $(\perp, data)$. In this case, when a new target address is issued over the MIB by the control unit, it is not pushed into the FIFO buffer. Instead, it is matched with the first occurrence of a $(\perp, data)$ entry starting from the oldest such entries. Additionally, it is possible that a new data entry and a new address occur at the same time, but those do not necessarily need to correspond. Further complicating the matter, completed entries might get consumed at any time.

Assuming an empty queue, target addresses from the MIB are pushed into the FIFO buffer. Now, valid data might also be present. In this case, a completed entry $(tgt, data)$ is pushed, otherwise only $(tgt, \perp)$.

Thankfully, VHDL features not only *signals* but also *variables*. When used within a process[5], the main difference between the two types is that new values are instantly assigned to variables whereas the assignment to signals takes time. Basically, all signal assignments of a clocked process are executed in *parallel*. Using variables, it is possible to describe sequential code, which makes implementation of the required output queue behavior less complex. Instead of describing the $2^3 = 8$ different parallel cases[6], only three sequentially executed cases are necessary.

First, it is checked whether the output queue should consume the head entry. Consuming only invalidates both valid fields and decrements the head pointer.

[5] which resembles a hardware component

[6] (incoming address, incoming data, consumption)

Afterwards, input from the MIB is checked. If it is both valid and cannot be ignored due to a repeat the tail entry of the queue is examined. In case it contains at least a valid address, the complete queue contents are shifted upwards, the old tail entry is replaced by $(tgt, \bot)$ and the head pointer is incremented. In the other case, the address is matched to the first entry of the form $(\bot, data)$ starting at the highest queue entry (greater or equal to the head).

Last but not least, valid input from the attached processing unit is processed. Similar to the MIB input step, if the tail entry contains valid data, the whole queue content is shifted upwards, the old tail is replaced by $(\bot, data)$ and the head is incremented. Otherwise, the data is matched to the first entry with a valid address similar to the aforementioned step.

Note that it is guaranteed that the head position from one clock cycle to the next can only change by at most one step in either direction. This is because both head-incrementing cases cannot occur at the same time, because if the MIB input step increments the head position, the following step will find $(tgt, \bot)$ in the tail entry, preventing it from further incrementing.

While using variables in VHDL enables a very clean description of the behavior, the resulting circuit is very abstract due to synthesizer operation. Unfortunately, translating this complex behavior to a simple block diagram is not possible and as such no internal schematic is given.

3.4.3 Branch Unit

The branch unit is a special functional unit for two reasons: First, it has a direct connection to the previously discussed control unit and as such there can only be one branch unit within a SCAD machine. However, this is not a disadvantage, because the semantics of any given move-program is sequential. While data processing can be distributed across many functional units, control flow can not.

Instead of implementing the direct connection to the control unit, another method proposed in [BJS16] is possible: There, the control unit receives the new PC value via the DTN. As soon as a move-instruction targets the control unit, it stalls itself and the incoming data is interpreted as the new PC value. While this solution uses the already existing architecture, it certainly requires more clock cycles for the new PC value to reach the destination. Additionally, the control unit needs a complete network interface that supports both directions, requiring more FPGA resources than a direct connection. However, some resources can be saved by having a simpler stall mechanism.

The branch unit has its own set of instructions, supporting up to two operands. As such, three input queues are necessary, two for the operands and one for the operation. The output queue is mainly used

Table 3.5: List of branch unit instruction opcodes which can affect the PC value.

Opcode	Shorthand	Description
00000	JUMP	Jumps unconditionally to the destination contained in the operation.
01000	JUMP_OP1	Jumps unconditionally to the destination given by the first operand. Allows jump addresses to be calculated at run-time.
10000	BLESS	Jumps to destination contained in the operation *if* first operand is less than the second operand.
10001	BLEQ	Jumps to destination contained in the operation *if* first operand is less than or equal to the second operand.
10010	BEQ	Jumps to destination contained in the operation *if* first operand is equal to the second operand.
10011	BNEQ	Jumps to destination contained in the operation *if* first operand is not equal to the second operand.

to loop back to the inputs to enable more complex branching behavior. A branch instruction consists of the opcode and, depending on the instruction type, a target PC value. Table 3.5 list the opcodes of the implemented branch instructions[7]. Note that this list can be easily extended to implement additional instructions. Additionally, the branching unit supports general logic operations like `AND`, `OR` and `XOR` to allow local chaining of multiple branching conditions.

3.4.4 Universal ALU

The universal *Arithmetic Logic Unit* (ALU) has three input queues and one output queue. Two input queues contain the data of the operands whereas the data of the third input queue determines the operation. Table 3.6 gives an overview of the implemented operations, their opcodes and a short description.

Note that this list can easily be extended to implement missing operations such as bit shifts and incrementation and decrementation. Immediate types could also be implemented when utilizing the unused bits of the 32 bit operation to store a smaller immediate.

The universal ALU corresponds roughly to the universal functional units described in [BJS16] with the exception of the integrated duplication functionality. To implement this behavior, some of the unused bits of the operation could contain the number of requested copies. Another possibility would be by utilizing an additional input queue which determines the number of copies. However, this would mean that for example a simple `NAND` operation requires four values from four input queues. When these values

[7] Again, inspired by RISC-style instructions.

Table 3.6: List of implemented instruction binary opcodes of the universal ALU.

Opcode[4:0]	Shorthand	Description
`00000`	ADD	Signed addition.
`00001`	ADDU	Unsigned addition.
`00010`	MUL	Signed multiplication. Result is truncated to 32 bits.
`00011`	MULU	Unsigned multiplication. Result is truncated to 32 bits.
`01000`	AND	Bitwise `AND`.
`01001`	NAND	Bitwise `NAND`.
`01010`	OR	Bitwise `OR`.
`01011`	NOR	Bitwise `NOR`.
`01100`	XOR	Bitwise `XOR`.
`01101`	XNOR	Bitwise `XNOR`.

originate from a remote unit, additional traffic increases the load of the interconnection network further. Thus, I would recommend the first solution.

3.4.5 Load and Store Unit

The *Load and Store Unit* (LSU) is another singleton functional unit. It grants both read and write access to a scratch-pad memory on the FPGA. As a simple proof-of-concept implementation, only one memory operation at a time is allowed, although the utilized dual-port block RAM allows two independently addressed accesses at the same time. Two such LSUs could share a common scratch-pad memory. However, synchronizing the accesses would require more complex logic outside the scope of this thesis.

At least two input queues are necessary for providing writing access, one containing the memory addresses and the other containing the data. To provide reading access, only an input queue containing the memory addresses is required in addition to an output queue which will be filled by the data read at given the memory addresses. To differentiate between the two possible operations, several solutions are possible.

The clearly worst solution would be an additional input queue to select the operation. This would require another data tuple for each operation, similar to the situation discussed in Section 3.4.4. A better solution would be to have separate address input queues for each operation. This requires no further data tuple. The implemented solution uses this idea, however employs a *virtual* input queue. This queue that does not exist, but still can be addressed. By addressing either the real or virtual input queue storing the

Table 3.7: Implemented miscellaneous function units.

# Input Queues	Function	Description
2	ADD	Consume a value from both input queues, perform addition and store the result in the output queue.
2	MUL	Consume a value from both input queues, perform multiplication and store the result in the output queue.
2	DUP	Consume a value from both input queues, and store n duplicates of the first operand in the output queue, where n is given by the second operand.
1	SWAP	Consume two values from the input queue and store them in reverse order in the output queue.

addresses, the operation is selected. To store the selected operation, the given data bits containing the address are shifted to the left by one. The now freed least significant bit is then used indicated whether the given address is used for a read or write operation.

Because the components used are unaware of the non-existence of the virtual queue, some signal modifications between the network interface and the first input queue are necessary. In particular, the `valid` signal must be `HIGH` when either the first or virtual second queue are selected by the network interface. The incoming data is modified as described above. Last but not least, the target queue address must be changed to the first queue whenever the virtual queue is addressed. These modifications only require very few LUTs, certainly less than another instance of a real input queue.

3.4.6 Miscellaneous Functional Units

The implementation of every other functional unit is straight forward. Only the processing unit and the number of input queues differ from each other. Table 3.7 lists all such implemented functional units.

3.5 Network Interface

As the name suggests, one task of the network interface is to provide a conversion between triples and flits and vice versa. This task is divided into two sub-components, the *splitter* and the *merger*. The splitters's task is to split the incoming triple from the output port(s) of a functional unit into flits for the network using the partition shown in Table 3.1. The merger's task is the reverse: merging the incoming flits from the network into triples for the input queues.

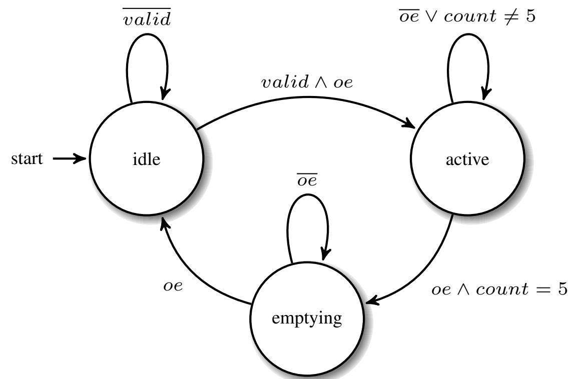


Figure 3.2: Finite state machine implemented by the Splitter control logic.

Another task of the network interface is to route local traffic from the output queue(s) of a functional unit to the targeted local input queue. This might result in a conflict as both the network and the local output queue could try to send data tuples at the same time. This is resolved by always granting the network precedence. The reasoning behind this design decision is that there must be at least one clock cycle between flits of different network packets (see Section 3.7).

3.5.1 Splitter

Internally, the splitter mainly consists of a 6:1 multiplexer for the 6 different flit contents. The order is given by the partition of Section 3.2. The multiplexers select-input is controlled by a counter from 0–5. This counter is increased whenever a flit has been send successfully (receiving router did not signal `full`). The final state machine implemented by the splitter is depicted in Figure 3.2. The signal `output_consume` is only `HIGH` for one clock cycle when entering the state `emptying`.

The splitter implementation for VAL is slightly different. It is connected to a pseudo-random number generator implementation, here a *Linear Feedback Shift-Register* (LFSR) of length 4 which can be implemented efficiently in a FPGA. As the name suggests, a LFSR is a chain of shift registers where the last register and some registers in between feed back into the first through a XOR gate. Here, the last and second to last registers are used. This realizes a polynomial of the form of $x^4 + x^3 + 1$ and generates a repeating quasi-random sequence of length $2^4 - 1$. Using the current random number (which is the same for all splitters) and `XOR`ing it with the `SEED` VHDL generic, creates a pseudo-random intermediate target. The first flit that is produced contains this target and has its length bit set to `HIGH`, signaling a message length of seven flits instead of just six.

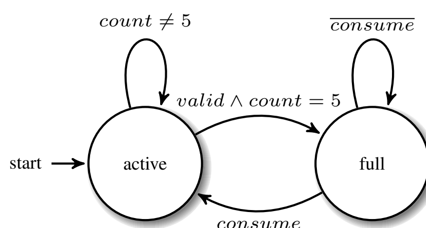


Figure 3.3: Finite state machine implemented by the Merger control logic.

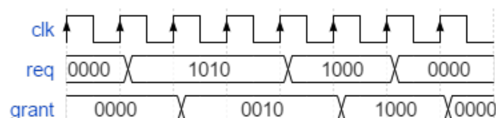


Figure 3.4: Timing diagram of required behavior of a round-robin arbiter.

In order to implement the UGAL routing algorithm, the splitters would also need access to the status signals of the input ports of the interconnection network. The required decision logic would also be implemented in here.

3.5.2 Merger

The merger works in reverse to the splitter. Incoming flits from the interconnection network are send into a demultiplexer which is used to store the flit contents into the corresponding triple part of its internal buffer. The demultiplexers select-input is controlled by a counter from 0–5. This counter is increased whenever a valid flit has been received. The final state machine implemented by the merger is depicted in Figure 3.3. Counter value is reset to zero in when entering the state `active` from `full`. The output triple is only valid when the state machine is in the `full` state.

3.6 Round-Robin Arbiter

The task of a round-robin arbiter is to grant access to the client with the highest priority. Clients that were granted receive the lowest priority in the next cycle.

In general, a clocked[8] round-robin arbiter has a `request` bit-vector of length $n = \#clients$ as an input and a `grant` bit-vector of length n as an output. The output encodes the winner of the arbitration in

[8]Otherwise switching glitches occur.

an *one-hot*[9] encoding. Every cycle a new winner is determined by finding the next set bit starting from the position of the old winner. The resulting arbitration is fair since all clients will eventually be granted access.

Unfortunately, this behavior is not yet useful for the implemented protocol between routers (see Section 3.7). Instead, the desired behavior is that the winning input port keeps winning until it is done sending its packet. Figure 3.4 characterizes this behavior. It is called “grant-hold” in the literature [DT04] and is achieved by only selecting a new winner when the corresponding `request`-bit of the previous winner is set to `LOW`.

To implement this behavior, small tricks from Benjamin Krill[10] are used. To isolate the least significant set bit of an arbitrarily large bit-vector, the following formula results in a bit vector where at most one bit is set to `HIGH`:

$$grant_i = request_i \wedge (\overline{request_i} + 1) \tag{3.1}$$

where the subscript indicates the clock cycle. Correctness can trivially be shown. For an example, consider $request_i = 1001$. Then

$$grant_i = 1001 \wedge (0110 + 0001) = 0001 \tag{3.2}$$

The critical case where the addition overflows can only appear when $request = 0000$. However, the `AND` takes care that $grant = 0000$ as desired.

To test whether the grant is released, the $grant_{i-1}$ of the previous clock cycle is `AND`ed with the current $request_i$. If the result is zero, the previous winner has released the grant. The previous winner is masked of by this formula:

$$mask_i = request_i \wedge \overline{((grant_{i-1} - 1) \vee grant_{i-1})} \tag{3.3}$$

For an example, lets assume $grant_{i-1} = 0100$ and $request_i = 1100$. Then

$$mask_i = 1100 \wedge \overline{(0011) \vee 0100)} = 1100 \wedge 1000 = 1000 \tag{3.4}$$

Again, correctness is trivial. The only critical case is the underflow of the subtraction which results in $mask_i = 0000$. Therefore, in this case the unmasked $request$ must be used.

[9] Meaning at most one bit is set to `HIGH`.

[10] `http://www.krll.de/portfolio/round-robin-arbiter/`

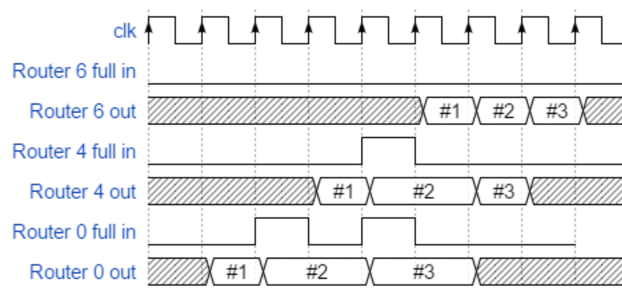


Figure 3.5: Router timing protocol without support for virtual channels. Number of flits per packet is three and flit size is 16 bit.

While the round-robin arbiter is fair in the sense that eventually every request is granted, it is not particular fair with respect to the ordering. For an example, in a system with four clients, both the first and last client request access, resulting in `request = 1001`. Assuming without loss of generality no previous winner, the first client is granted access (`grand = 0001`). Later but while the first client is holding the grant, the second client also requests access (`request = 1011`. Now, as soon as the first client releases the grant, the second client will receive the grant despite the last client waiting longer.

Clearly, when transferring this general disadvantage to router port arbitration, this behavior increases the worst-case latency. Dally discusses in [DT04] two more complex arbiters which solve this particular problem. The first being a *matrix arbiter* which grants access to the least recently granted client. The second being a *queuing arbiter* which grants access in the exact order clients requested access using time stamps. Unfortunately, due to the complexity and the number of necessary instances, they are not feasible for implementation and thus are not discussed any further.

3.7 Network Protocol

For networks without virtual channels, the routers involved in transferring a particular message across the network are expected to implement the following protocol:

While the receiving router input port is not signaling `full`, send new flits of the message. When `full` is `HIGH`, repeat the last send flit. This behavior results in a continuous stream of valid flits and enables the receiver to work as fast as possible. There is never the situation where an input port would be able to transmit, but has no valid flit to send.

Figure 3.5 illustrates this. Here, router zero is sending the three flits of a packet through router four to router six (input of router four equals output of router zero etc.). The one clock cycle delay from input to

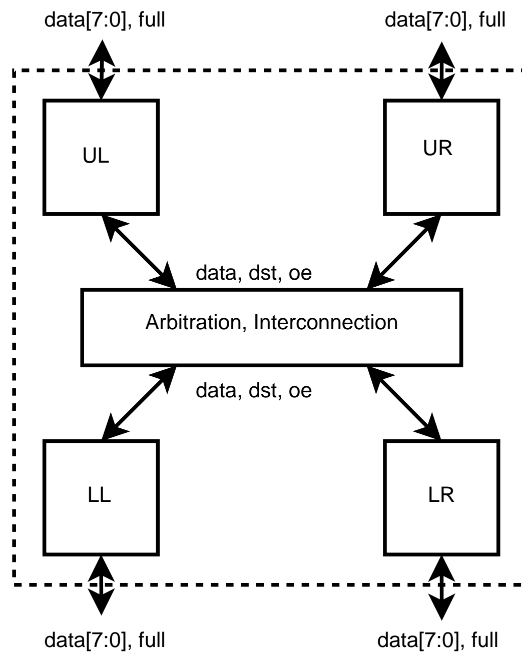


Figure 3.6: Simplified block diagram of a fat-tree router.

output is due to arbitration. No further arbitration induced waiting period is generated for router six as it is connected to network interfaces which are able to hold the compete network message. Note that there is no synchronization necessary, because all network parts operate on the same clock.

Due to the requirements of the previously discussed round-robin arbiter implementation (see Section 3.6), one clock cycle space between two flits of different network packets is necessary to guarantee that the request of the corresponding input queue will be cleared.

This protocol is identical for networks with support for virtual channels. However, due to the multiplexing of multiple virtual channels to one physical channel, it is no longer a guaranteed continuous stream of valid flits. Participating routers must be able to handle cases where no new valid flit is present at the input port.

3.8 Fat-Tree Router

All fat-tree routers share the same architecture, whether the lower ports are connected to the network interfaces of the corresponding functional unit or some upper ports of other fat-tree routers. The necessary positional information for correct routing operation is provided by a VHDL generic called

Table 3.8: Incoming and outgoing signals for a bidirectional router port (external view).

Signal	Direction	Description
data_in[7:0]	in	Incoming flits.
data_in_valid	in	Valid flag for incoming flits.
data_in_full	out	Flag which indicates whether this router port is full.
data_out[7:0]	out	Outgoing flits.
data_out_valid	out	Valid flag for outgoing flits.
data_out_full	in	Flag which indicates whether the receiving router port is full.

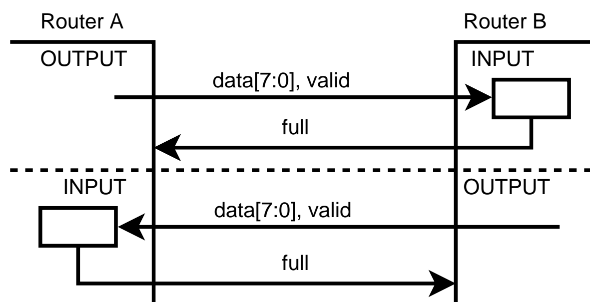


Figure 3.7: Connections between two routers.

ROUTER_CONFIG. It is a tuple consisting of the router address and a small integer which is used to identify the significant bit of the routing operation.

The basic architecture of such a router is depicted in Figure 3.6. It shows four bidirectional ports called UL, UR, LL and LR. Each port consists of an input port and an output port. In between all ports is the internal interconnection component that is responsible for the interconnection, arbitration and flow control. Incoming and outgoing signals for each port are listed in Table 3.8. Figure 3.7 illustrates these signals graphically.

3.8.1 Input Port

The routing on the fat-tree topology consists of two distinct phases, the up-routing phase followed by the down-routing phase (recall Section 2.2.1). The lower input ports which are involved in the first phase are slightly different to the upper input ports that are responsible for the deterministic second phase. They differ only in the method used to determine the inner router destination, which is either UP or DOWN for lower inputs and DOWN_LEFT and DOWN_RIGHT for the upper inputs.

The finite state machine implemented by any input port is identical to the finite state machine depicted

in Figure 3.2 for the splitter. Upon entering the active-state, the input port determines the destination of the next router towards the final destination of the packet. Depending on the type of input router, it is implemented as follows:

For the upper ports, this is achieved by comparing the correct bit of the target unit address in the incoming header flit. As mentioned beforehand, the correct bit to use is determined by the ROUTER_CONFIG. When this bit equals zero, the destination is DOWN_LEFT, otherwise it is DOWN_RIGHT. There is no up-direction for these ports since U-turns are strictly forbidden. A U-turn means that the target terminal is identical to the source terminal. While a U-turn is generally allowed for the SCAD machine, it must be handled locally to the functional unit and is not allowed to transmit anything into the interconnection network.

For the lower ports, the destination is determined by comparing the significant part of the router address with the target unit address. If they differ, the up-routing phase continues and the destination is UP. However, if both are identical, it means that a common ancestor was found and the down-routing phase begins. Thus, the destination is just DOWN since U-turns are not allowed.

The flow control output bit full is determined as follows. It is set to LOW either when the current state is idle or the output is enabled by the interconnection while not being in the emptying state. Otherwise it is set to HIGH. If the finite state machine is in the emptying state, the router is not allowed to accept new flits until at least one idle cycle was completed in order to comply with the network protocol.

3.8.2 Interconnection

The task of the interconnection component of the fat-tree router is not only to connect input ports to output ports but also to handle the flow control. The easiest way of realizing the internal interconnect between input ports and output ports is a full crossbar. However, this is very inefficient as not every input port can access every output port. As such, only necessary connections are implemented.

For both lower output ports arbitration is necessary to guarantee conflict-free access to them. Each port has its own round-robin arbiter with three clients. The request bit-vector is generated from the destination outputs of the input ports that can access the corresponding output port, depicted as D2R in Figure 3.8. The grant bit-vector is used to control the multiplexer whose outputs are directly connected to the output ports of the router.

The output_enable signals for the input ports are computed as follows: For all possible directions, corresponding bit in the grant bit-vectors of the round-robin arbiters is ANDed with the negation of the

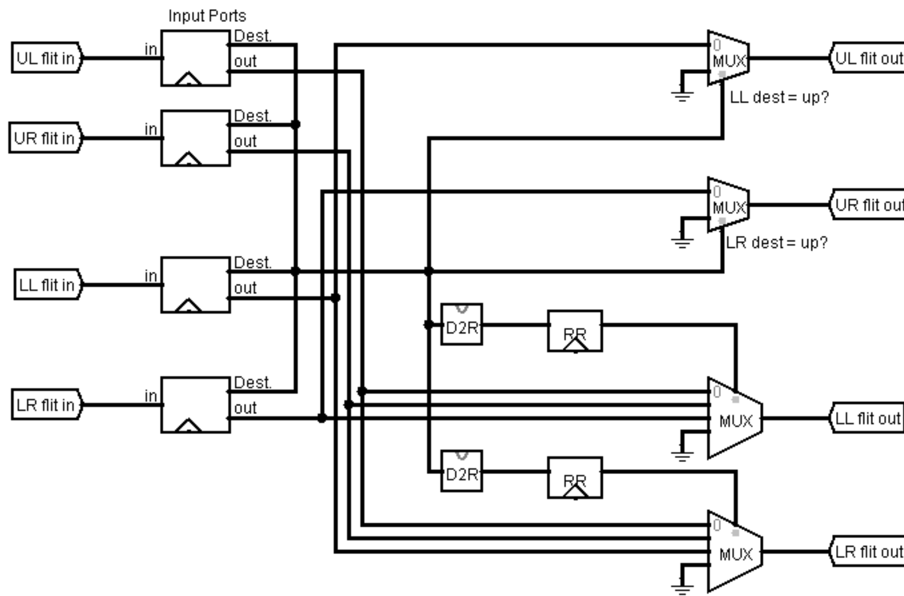


Figure 3.8: Equivalent circuit schematic of a fat-tree router without flow control logic.

corresponding `full` signal of the requested port. These results are than `OR`ed to obtain each `output_enable` signal.

$$
\begin{aligned}
output_enable_UL = \; & (rr_LL_grant(0) \wedge \overline{LL_out_full}) \\
& \vee (rr_LR_grant(0) \wedge \overline{LR_out_full})
\end{aligned}
\tag{3.5}
$$

Here is an example: Assume the upper left input port wants to transfer flits to the lower right output port. Thus, its destination output is `DOWN_RIGHT`. With this, the corresponding bit in the `request`-vector of round-robin arbiter responsible for the lower right port is set to `HIGH`. Eventually, the corresponding bit in the `grant` vector is set to `HIGH`. According to the formula stated in Equation 3.5, the output will only be enabled when the input port connected to the lower right output port signals `NOT full`.

To implement support for an adaptive routing algorithm, both lower input ports need to be connected to both upper output ports. Additional logic is necessary to handle the situation where both input ports want to access the same upper output port. For testing purposes an additional `waiting` state was introduced with the attached meaning that the header flit was not yet transmitted successfully. This allowed the input port to try the other output port.

Figure 3.8 illustrates the equivalent schematic of the VHDL description. Input ports are located on

the left side whereas the output ports are located on the right side.

Note that both upper output ports have only one connection with an input port. The reason for this is, that a locally adaptive (greedy) routing algorithm for the up-routing phase proved to be ineffective. The one clock cycle delay due to the arbitration adds up quickly for larger fat-trees. By not having to wait for one clock cycle for the arbitration, the total latency is reduced by the number of hops during the up-routing phase. Additionally, a globally adaptive routing algorithm would be necessary to further improve performance of the fat-tree network.

3.9 Flattened Butterfly Router

The architecture of flattened butterfly routers is very similar to the fat-tree design. Each router has two bidirectional ports solely for connecting to the network interfaces of two functional units. Additionally there are $D' = log_2(N) - 1, N > 0$ bidirectional ports depending on the number of network size N. For a interconnection network supporting 32 functional units, the number of additional ports amounts to $D' = log_2(32) - 1 = 4$.

The design and behavior of the input ports are very similar to a input port of a fat-tree router. Here, all input ports of a single router are exact copies of each other. Destinations are all available dimensions, D' inter-router dimensions and the last dimension which involves the two local ports. For a network size of 32, the destinations are `dim0`, . . . , `dim3`, `dim4_0` and `dim4_1`. For minimal routing algorithms, the dimensions which still have to be traversed are computed by `XOR`ing the D' most significant bits of the incoming header flit target address field with the `ROUTER_ADDRESS` generic. If the result is $\vec{0}$, the message is at the target router and can be handled locally.

The signals between external router ports and their connections are identical to the ones described in Section 3.8. Of course, the flattened butterfly routers are connected to different routers according to the topology discussed in Section 2.2.2.

Flow control is handled by the input ports themselves. To this end, they have reading access to the port states of all neighboring ports. Input ports can expect that the transmission of the current flit was successful if both the receiving buffer did not signal `full` and `output_enable` signal from the interconnection component is `HIGH`. This is the case when the round-robin arbiter granted access to the corresponding port.

To implement dimension-ordered routing, the bits of the resulting vector are checked in descending

Table 3.9: Incoming and outgoing signals for a bidirectional router port with virtual channel support (external view).

Signal	Direction	Description
data_in[7:0]	in	Incoming flits.
data_in_valid	in	Valid flag for incoming flits.
data_in_vp[v':0]	in	Containing the destination virtual port of this router.
data_in_full[v-1:0]	out	Flag which indicates whether the virtual ports of the corresponding router port are full.
data_out[7:0]	out	Outgoing flits.
data_out_valid	out	Valid flag for outgoing flits.
data_in_vp[v':0]	out	Containing the destination virtual port of the other router.
data_out_full[v-1:0]	in	Flag which indicates which virtual ports of the receiving router port are full.

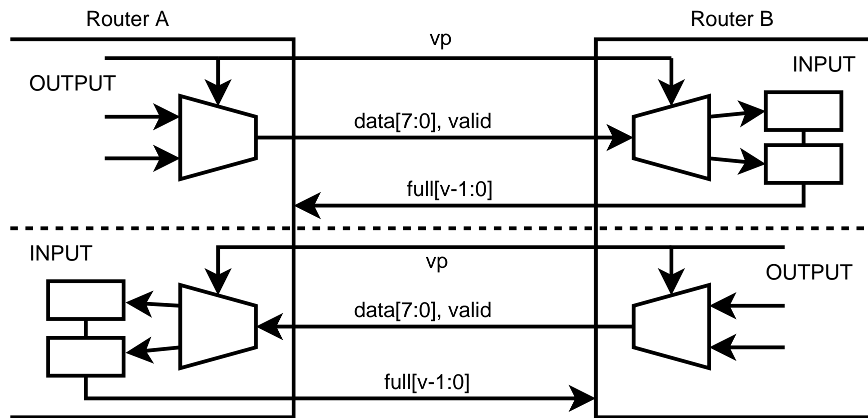


Figure 3.9: Simplified block diagram of inter-router connections of two routers with virtual channel support.

order. The first bit set to HIGH determines the internal destination. The adaptive greedy-algorithm additionally ANDs this vector with the corresponding input port states of the neighboring routers to determine the best (local) route. However, because the greedy algorithm needs D' virtual channels, it is discussed further in Section 3.10.

3.10 Flattened Butterfly Router with Virtual Channel Support

Adding v virtual channels to the aforementioned flattened butterfly router architecture is relatively simple in theory. However, in practice this task required a lot more effort than originally thought.

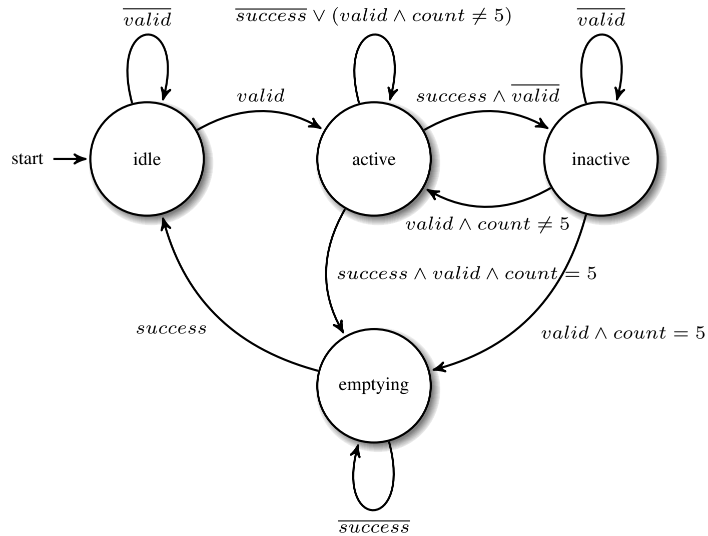


Figure 3.10: Finite state machine implemented by the flattened butterfly input port control logic with support for virtual channels.

First of all, routers have to send $v' = log_2(v)$ virtual channel addressing bits alongside every flit. These are used to select the correct virtual input port of the receiving router. Table 3.9 lists all necessary external router signals connecting one port to another in one direction. Figure 3.9 illustrates the aforementioned signals.

Next, the number of destinations is multiplied by v, which in turn means a huge blow-up in the size of the interconnection network. Every virtual input port needs to be connected to every other virtual output port and thus the required arbiters are multiplied by v, too.

Last but not least, any physical inter-router channel is multiplexed between multiple virtual ports. This breaks the simple protocol described in 3.7. To fix this, the routers needed adjustments to their logic which is described in the next section.

3.10.1 Input Port

Because of the preceding demultiplexer, incoming flits are already destined for the given router, thus no changes were necessary in that respect. In order to handle the intermittent stream of flits a new state for the finite state machine was necessary (see Figure 3.10). The new `inactive` state denotes that the internal buffer is empty. It is entered whenever data was *successfully* send the previous clock cycle, but no valid incoming flit is present this clock cycle. Successfully means that both the target buffer was not

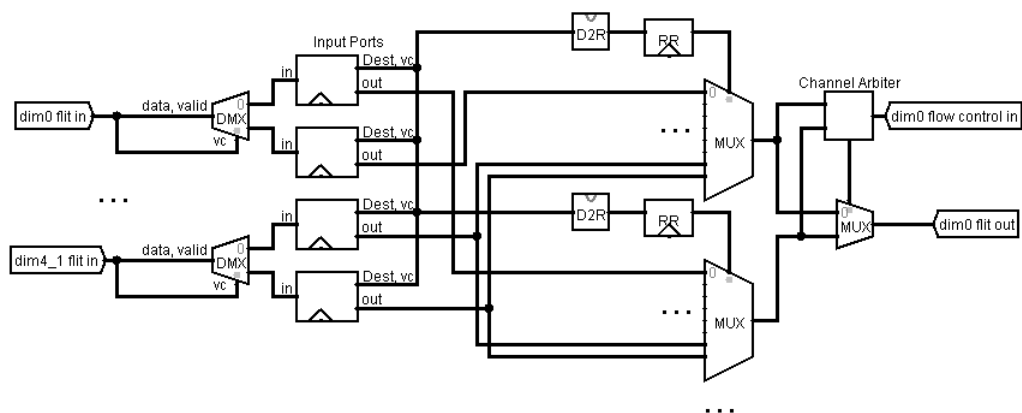


Figure 3.11: Equivalent schematic of flattened butterfly router internals with support for 2 virtual channels. Here, only two physical input ports and one physical output port are shown for space and clarity reasons. Other ports are similar. Also note, that flow control signals connected to all virtual input ports are hidden.

full and that the output of the transmitting input port was enabled. The counter can be increased in both the `active` and `inactive` state whenever a valid flit is received. Output signals now additionally include the aforementioned virtual channel addressing bits, which are used alongside the destination output for the internal arbitration.

To implement the non-minimal VAL routing algorithm, input ports have to check the least significant bit of the header flit whether the packet is an additional flit long. Afterwards, the counter either starts at zero or one to accommodate for the longer packet. As usual, the first flit contains the (immediate) destination. In case the immediate destination is reached, the additional flit is simply dropped and the state machine remains in the `idle` state. This results in interpreting the actual second flit as the new header flit and entering the second phase. DOR is used to route minimally in both phases as described in Section 3.9. Routing during the first phase uses exclusively the first virtual channel of all involved ports whereas routing in the last phase uses exclusively the second virtual channel. Fixing the use of virtual channels to the described fashion avoids deadlocks as discussed in Section 2.3.3.

3.10.2 Interconnection

The design published by Kavaldjiev et. al in [KSJ04] was used due to the advantage that no blocking of flits occurs at the input ports. Otherwise, when using a more traditional design like the one used in [MWM04] where the multiplexers used for the virtual channel allocation are placed in between input

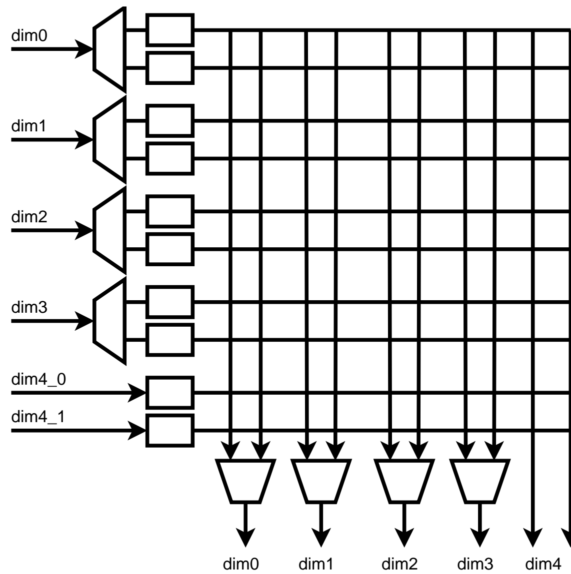


Figure 3.12: Simplified block diagram of the router internals. Flits from the corresponding physical channels enter on the left hand side and exit after successful arbitration through the bottom output ports. Flits are only stored within the (virtual) input ports, here shown as rectangles.

port and interconnection only allows one flit of a input port at the same time to enter the smaller crossbar. In hindsight, the later design should scale better than the chosen design. However, at the time of this decision I severely underestimated the required resources for large crossbars and had mainly performance in mind.

Thus, the size of the interconnection component of a flattened butterfly router with virtual channels is relatively huge in terms of necessary logic. Every virtual input port of the neighboring routers are potential destinations, thus each need their own round-robin arbitration. One arbitration per physical port is not enough, because then the multiplexing would not work. Additionally,'U-turns to and from the same physical channel must be allowed as long as a different virtual channel is used. This is necessary, because VAL for example might route non-minimally through the actual destination. Then the intermediate router has to send the flits back the same physical channel they came from.

Figure 3.11 depicts the block diagram of the interconnection. Only two physical input channels and one physical output channel is shown to reduce complexity and increase readability. Missing ports are connected similarly. The resulting interconnection is illustrated in Figure 3.12 as a simplified block diagram.

The additional *Channel Arbiter* (CA) grants access to one of the virtual ports of the receiving router at

a time over the single physical channel. The simplest way of implementing a CA is by utilizing the well known *Time-Division Multiplexing* (TDM) technique. Each clock cycle another virtual port is granted access. And since every component is perfectly synchronized, there is no need to send virtual channel identification bits as long as all routers use the same TDM schedule. However, the main disadvantage is the poor utilization of the physical channel when only a single virtual port wants to send data. In the best case the utilization is now only $1/2$ for networks with two virtual channels, but with four virtual channels the utilization is only $1/4$ in that case.

To fix this, a modified TDM scheme is implemented to only include virtual ports that want to transmit, i. e. with valid data. It could be implemented by a round-robin arbiter without grant-and-hold (see Section 3.6) when the previous winner is allowed to win again if it is the only request. The problem with this method is, that due to the protocol (see Section 3.7), virtual input ports always keep sending, whether the current flit is accepted or not. In case of blocks, the (productive) utilization of the physical channel is not ideal. Thus, to fix this disadvantage, the `full`-signals of the corresponding input ports are taken into account.

The final implementation of a CA works as follows: As long as the virtual buffer is both not `full` and receives valid data, it is continuously granted access. As soon as this condition is not fulfilled anymore, access is granted to another virtual channel which is able to receive in round-robin fashion.

Chapter 4

Evaluation & Analysis

This chapter discusses the evaluation of the most important metrics of the interconnection network, latency, throughput and resource-cost (area). Additionally, the area requirements of the remaining SCAD machine components are listed in order to estimate whether the complete architecture fits on the target FPGA.

4.1 Network Performance

The best way of determining the performance of the interconnection network of the SCAD machine would be by compiling benchmark programs to this architecture and compare the run-times of the different implementations. Unfortunately, research towards such a compiler is still ongoing and additionally targets only SCAD machines with universal functional units [BJS16]. Furthermore, the partition of code to functional units is critical for the performance of the SCAD architecture. Using synthetic traffic patterns help deciding which partitions should be avoided and which partitions should be striven for.

4.1.1 Traffic Patterns

There are two types of traffic patterns for network stress testing, *benign* and *adversarial*. One representative of benign traffic patterns is called *Unified Random* (UR). Here, each output queue has an equal chance of targeting any function unit as long as it was not chosen as a destination yet. This balances the load across the network. Completely random traffic is also benign, however there the possibility exist that

a unit is targeted multiple times. This is undesired, because the resulting block is not caused by the network itself rather than the poor choice of targets. As such, only random traffic patterns that assign unique target addresses are considered for latency testing.

Table 4.1: Permutation traffic patterns for 5-bit addresses used for latency testing.

Pattern	Source Address	Target Address
Bit Complement	$(b_4, b_3, b_2, b_1, b_0)$	$(\neg b_4, \neg b_3, \neg b_2, \neg b_1, \neg b_0)$
Bit Reverse	$(b_4, b_3, b_2, b_1, b_0)$	$(b_0, b_1, b_2, b_3, b_4)$
Bit Rotation	$(b_4, b_3, b_2, b_1, b_0)$	$(b_3, b_2, b_1, b_0, b_4)$
Bit Shuffle	$(b_4, b_3, b_2, b_1, b_0)$	$(b_0, b_4, b_3, b_2, b_1)$
Bit Transpose	$(b_4, b_3, b_2, b_1, b_0)$	$(b_1, b_0, b_2, b_4, b_3)$

Table 4.1 lists all traffic patterns generated by permuting the bits of the source unit in the described fashion. These were calculated from the formulas found in [DT04]. Note that when the source address is equal to the target address, the packet is not offered to the network since it is handled locally. For example, reversing the bits of the source address `00000` results in the target address `00000`. The traffic pattern for the smaller networks with shorter addresses are similar.

4.1.2 Methodology

Figure 4.1 depicts the block diagram of the simulated functional unit. The *injector* directly interacts with output queue and imitates both the control unit and the processing unit. The target address is determined according to the traffic patterns introduced in the previous Section 4.1.1. The outgoing data contains the current cycle count which is used by the *reporter* of the receiving simulated functional unit to calculate the latency. This way, the total latency does not only include the latency caused by the interconnection network but also includes the latency caused by going through the output queue. Furthermore, according to [DT04] measurements without such a queue are incorrect. This is because in loaded blocking networks, most of the total latency is due to waiting in the source queue.

Note, however, that the simulated output queue is only able to store six entries by default until it is full. To simulate the behavior of any real processing unit accurately, the Injector does not try to push more entries into the queue than possible.

Injection timing is determined stochastically(Bernoulli). Instead of having all injectors use the same periodic *injection rate* r, each injector produces a new packet with a probability of $1/r$, independent of

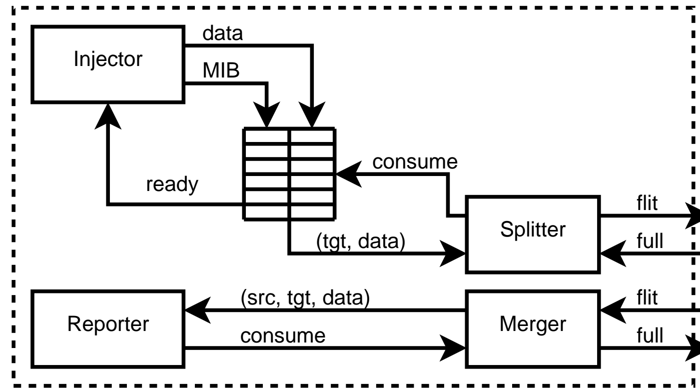


Figure 4.1: Block diagram of the simulated functional unit.

the others. The pseudo-randomly generated numbers using the non-synthesizable[1] random functions of VHDL are sufficient for this purpose.

Typically, the performance of a network is specified by the *steady-state* performance. Using stationary traffic sources, the network is brought to an equilibrium where the networks average queue lengths have been reached (cf. [DT04]). To reach the aforementioned equilibrium, at first the simulation must be *warmed up*. After the steady-state has been reached, performance *measurements* can be taken. Throughput is determined by counting all successfully received packets during this phase. Last but not least, all queues must be *drained* of all entries. Only after all queues have been drained, the latency for each individual packet can be determined. Utilizing the contained cycle count in the received packet, the total latency is calculated as follows:

$$Latency = cycle_{start} - cycle_{current} \tag{4.1}$$

The interconnection networks are tested using the Xilinx Vivado® 2015.3 VHDL simulator on the implemented hardware description. This implies a fully detailed behavior simulation on the register-transfer level (RTL), instead of more abstract simulations written in C/C++.

The carefully designed testbench used for performance measurements is compatible with all implemented networks, network sizes, flit size and traffic patterns. The injection rate is configurable via an external file to avoid the long compilation times of the simulator. Instead, the simulation is simply restarted after the injection rate has been changed.

The aforementioned phases are executed as follows: At first the warm-up phase is carried out for

[1]For simulation only.

Table 4.2: Performance measurements of traffic patterns under full load for the fat-tree network

Traffic Pattern	Latency [cycle] Average	Latency [cycle] Maximum	Absolute Throughput
Bit Complement	43	57	22760
Bit Reverse	140	169	5715
Bit Rotation	77	112	12477
Bit Shuffle	86	176	11431
Bit Transpose	140	169	5715
Unified Random	65	152	15647

1000 cycles. Then, the measurement phase runs for 5000 cycles and last but not least the drain phase is executed for another 500 cycles. While the number of cycles seem very small, they are large enough to allow accurate measurements. An order of magnitude higher cycles for all phases yielded comparable results while requiring proportionally longer run-times of the simulator.

The up to 32 reporter components each write to their own report-file recording the arrival of packets to them and calculating the latency according to Equation 4.1. Afterwards, these reports are automatically processed by a small parser written in C# that collects all values and computes the maximum and average latencies as well as counts the number of packets arrived during the measurement phase.

4.1.3 Analysis of Fat-Tree Performance

To determine the worst-case traffic pattern for any given network, it is simply stress-tested under maximum load for each pattern. The results are listed in Table 4.2. Unsurprisingly, the bit complement traffic represents the best-case scenario. This is because this pattern results in a non-blocking configuration of the implemented fat-tree network, which is obvious when comparing the achieved performance to the rest. Going forward, one of the worst-case traffic patterns of bit reverse and bit transpose are considered further, since they are best suited to evaluate the performance.

Clearly, some effort of the compiler partition should go towards the bit complement pattern and avoid the similar bit reverse or bit transpose patterns. Partitioning the code randomly appears to be very effective while requiring low compiler effort. As such, the unified random traffic is also considered.

Figure 4.2 depicts the latency vs. injection rate using the adversarial worst-case traffic pattern. Dou-

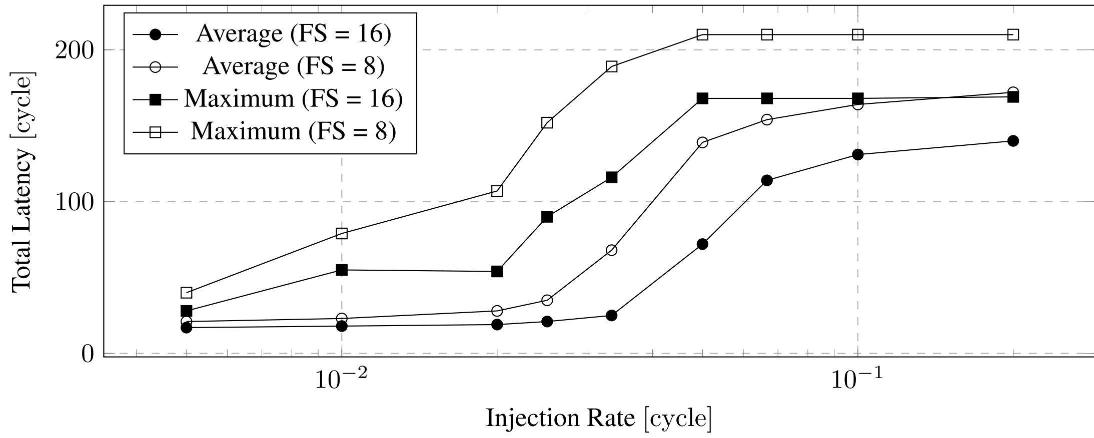


Figure 4.2: Total Latency vs. Injection Rate on a Fat-Tree network using the worst-case traffic pattern.

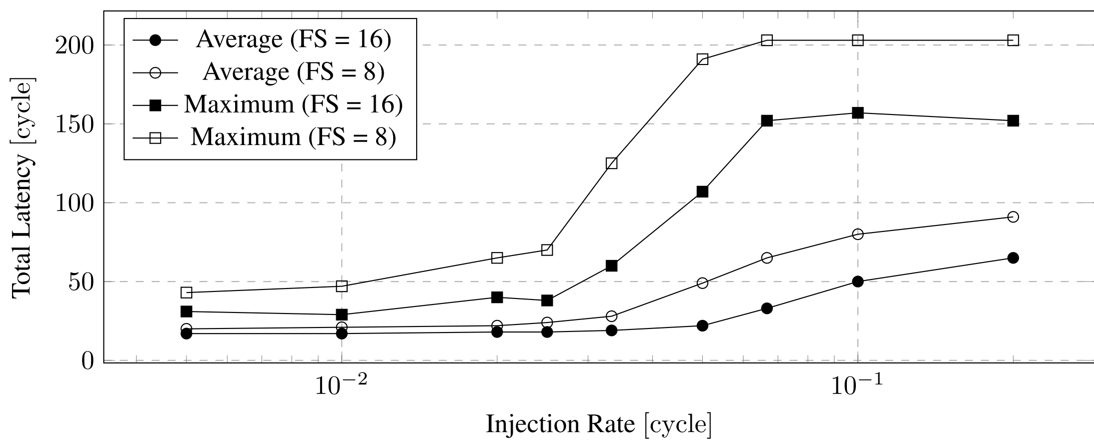


Figure 4.3: Total Latency vs. Injection Rate on a Fat-Tree network using benign unified random traffic pattern.

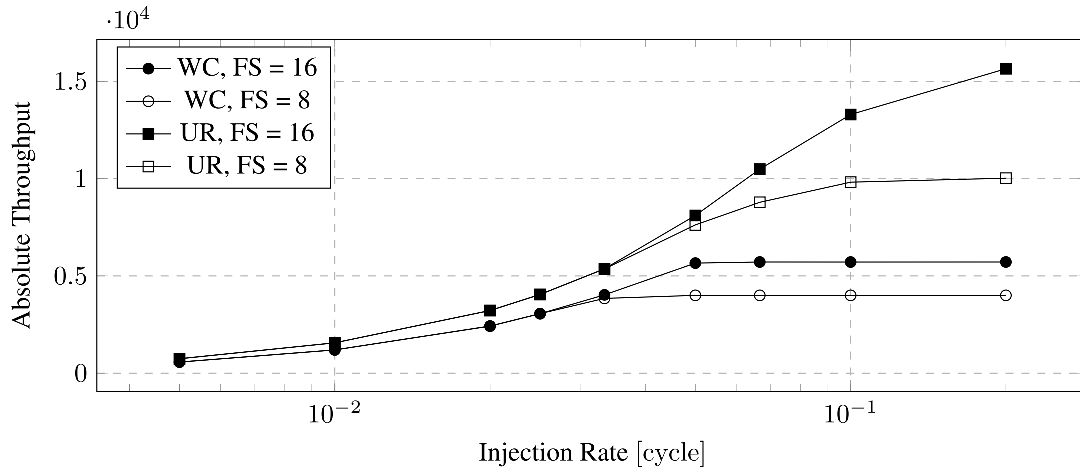


Figure 4.4: Throughput vs. Injection Rate on a fat-tree network with two different flit sizes and traffic patterns.

bling the flit size and thus halving the number of necessary flits to transmit the same amount of data, delays the inevitable sharp rise of total latency for both the average and maximum case. It also decreases average and maximum latency under full load by 18.6% and 19.5% respectively. Considering the unified random traffic pattern, latency is reduced by 28.6% on average and by 25.1% on maximum. Across all measured injection rates a resembling decrease in latency is observable.

Due to the behavior of the injectors described in Section 4.1.2 and the finite size of the source output queue, the latency is capped at 210 cycles for a flit size of 8 bits and capped at 169 cycles for a flit size of 16 bits. Assuming an infinite queue size, the latency would very quickly approach infinity beyond network saturation.

Lowering the injection rates even further, each latency curve will approaches the zero-load latency. With the smaller flit size, the zero-load latency is obviously higher compared to the larger flit size due to requiring more cycles to transmit the same packet in general.

The results of the simulation with the benign unified random traffic pattern are illustrated in Figure 4.3. Similar to the adversarial traffic pattern, the maximum and average latencies achieved with a smaller flit size are worse than the ones achieved with a flit size of 16.

Figure 4.4 plots the absolute throughput vs. injection rate. Considering the maximum achieved throughput, doubling the flit size results in an increase of 42.8% in throughput when using the worst-case pattern (56.1% when using the unified random pattern). Table 4.3 summarizes all discussed impacts of the flit size on the performance of the implemented fat-tree.

Table 4.3: Impact of the flit size on the performance of the fat-tree implementation under full load.

Traffic Pattern	Flit Size	Latency [cycle] Average	Latency [cycle] Maximum	Absolute Throughput
Worst-case	16	140	169	5715
Worst-case	8	172	210	4000
Unified-Random	16	65	152	15647
Unified-Random	8	91	203	10022

Table 4.4: Performance measurements of traffic patterns under full load for the flattened butterfly network using DOR.

Traffic Pattern	Latency [cycle] Average	Latency [cycle] Maximum	Absolute Throughput
Bit Complement	76	93	11425
Bit Reverse	49	83	12347
Bit Rotation	99	254	7901
Bit Shuffle	38	57	19398
Bit Transpose	69	120	8888
Unified Random	52	90	15761

Furthermore, interpretation of the curvature proves that the implemented network is fair. This is because the throughput does not decrease rapidly after saturation is reached. Instead, it stays constant.

4.1.4 Analysis of Flattened Butterfly Performance

Likewise to the fat-tree analysis, the worst-case pattern for the flattened butterfly network utilizing DOR is determined by measuring the maximum throughput for each pattern using 16 bit flits. The results are listed in Table 4.4. This network performs the worst using the adversarial bit rotation pattern and as such is used to stress-test it. Again, similar to the fat-tree case, the unified random pattern performs well and is only beaten by the bit shuffle pattern.

The latency vs. injection rate diagrams (Figures 4.5 and 4.6) for both considered traffic patterns show similar results to those discussed in Section 4.1.3. Across the whole range of measured injection rates, the latency achieved with 16 bit flits is lower than the counterpart. Considering the full load case, doubling

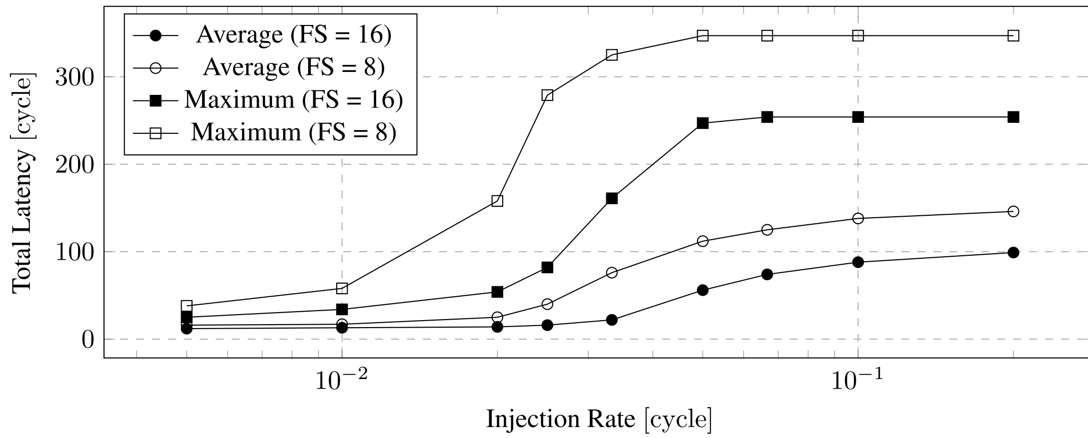


Figure 4.5: Total Latency vs. Injection Rate on a flattened butterfly network utilizing the DOR algorithm under the worst-case traffic pattern.

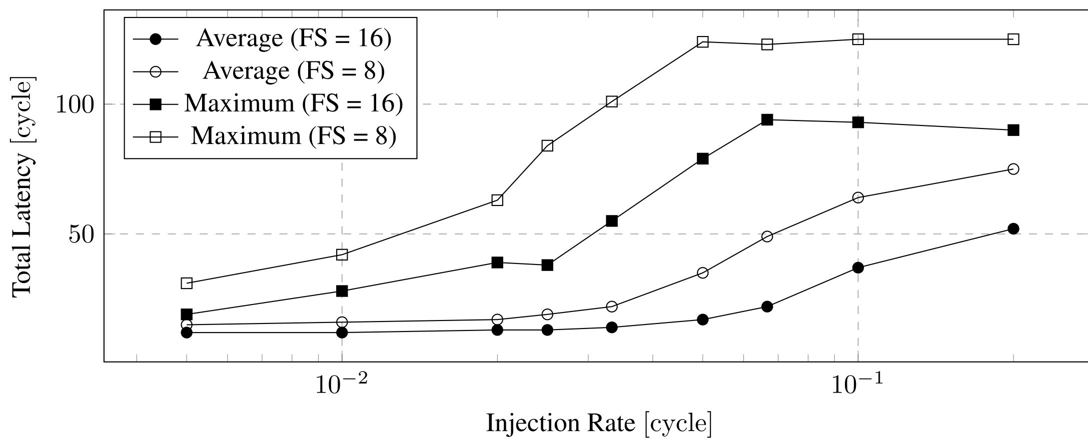


Figure 4.6: Total Latency vs. Injection Rate on a flattened butterfly network utilizing the DOR algorithm on benign unified random traffic pattern.

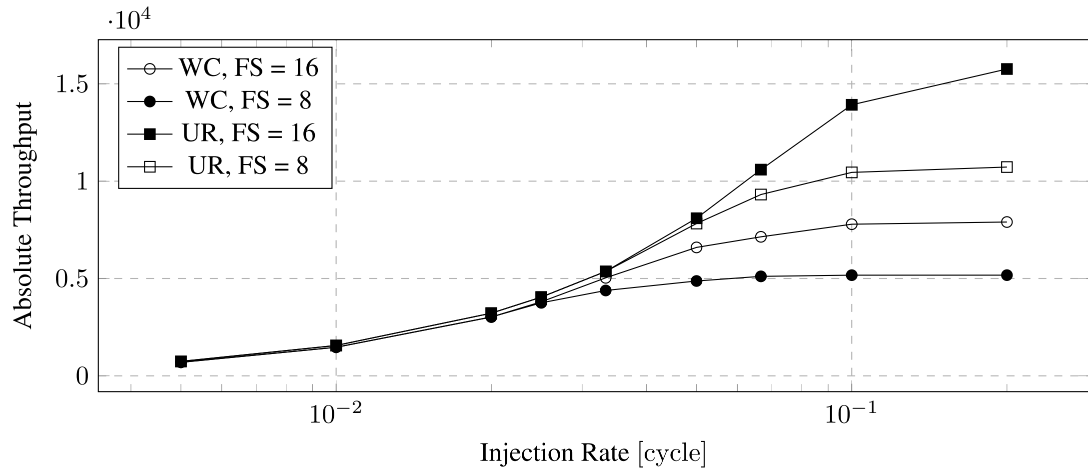


Figure 4.7: Throughput vs. Injection Rate for the 32 terminal flattened butterfly (DOR) network with two different flit sizes and traffic patterns.

Table 4.5: Impact of the flit size on the performance of the flattened butterfly implementation under full load.

Traffic Pattern	Flit Size	Latency [cycle] Average	Latency [cycle] Maximum	Absolute Throughput
Worst-case	16	99	254	7901
Worst-case	8	146	347	5172
Unified-Random	16	52	90	15761
Unified-Random	8	75	125	10725

the flit size lowers the latency by 32.2% on average and 26.8% on maximum using the worst-case pattern. With unified random traffic, average latency is decreased by 30.6% and maximum latency decreased by 28.0%.

Throughput is increased by 52.7% using the worst-case pattern. When using the unified random pattern, throughput is boosted by 47.0%. Table 4.5 summarizes all discussed impacts of the flit size on the performance of the implemented flattened butterfly utilizing DOR.

Again, analyzing the curvature of the achieved throughput proves the fairness of the implemented network for the same reason as discussed in 4.1.3.

Table 4.6 displays the measured performance of the flattened butterfly network utilizing the VAL

Table 4.6: Performance of the flattened butterfly network using VAL on two virtual channels under full load. Flit size is 8-bit.

Traffic Pattern	Latency [cycle] Average	Latency [cycle] Maximum	Absolute Throughput
Bit Complement	240	667	3476
Bit Reverse	193	549	3252
Bit Rotation	232	743	3396
Bit Shuffle	233	674	3364
Bit Transpose	186	449	3388
Unified Random	250	712	3348

routing algorithm and two virtual channels under full load with 8-bit flits[2]. Unsurprisingly, the obtained results were disheartening. When the reported throughput is considered in isolation, the algorithm works perfectly fine—the throughput is practically identical regardless of the traffic pattern. However, when comparing it to the worst-case performance of DOR the achieved throughput is less than half of it. Even worse is the achieved latency. With an average of 222 cycles on average latency and an average of 632 cycles on maximum the usage of this algorithm is not feasible. However, the working VAL routing algorithm proves that the implemented support for virtual channels on the flattened butterfly network was successful.

4.1.5 Direct Comparison

Comparing the two implemented interconnection networks directly, the following results are observable: First of all, using the benign unified random traffic pattern both the average and maximum latencies of the flattened butterfly are lower than the corresponding fat-tree latencies across all tested injection rates (see Figure 4.8). At an injection rate of $1/5$, the difference amounts to 13 cycles for the average latency and 62 cycles for the maximum latency.

Figure 4.9 plots the latencies vs the injection rate when using the respective worst-case traffic patterns. Surprisingly, the flattened butterfly network reaches much higher maximum latencies (254 cycles compared to 169 cycles) even though its average latency is still lower than the average latency of the fat-tree network.

[2] 8-bit flits were used, because there the additional flit necessary for VALs operation has less of an impact on overall performance.

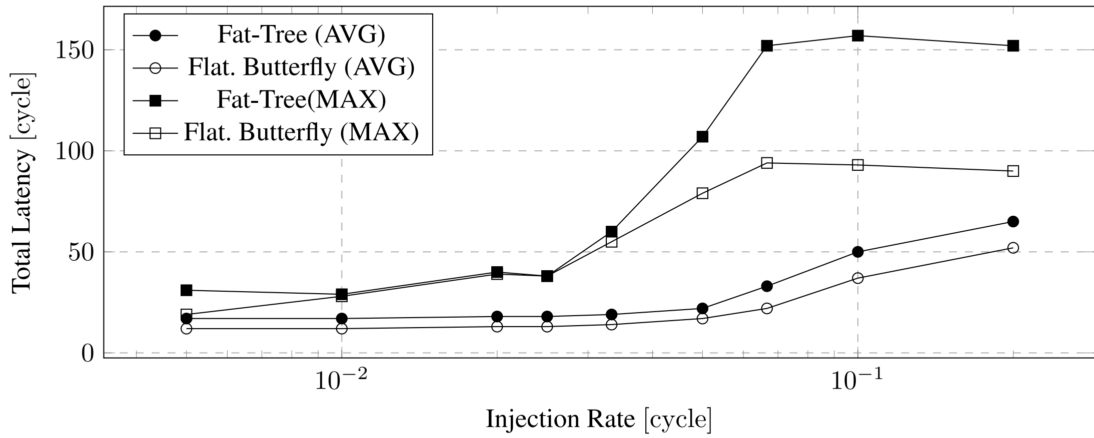


Figure 4.8: Latency vs. Injection Rate comparison for the 32 terminal networks on unified random traffic pattern. Flit size is 16 bits and the flattened butterfly network utilizes DOR.

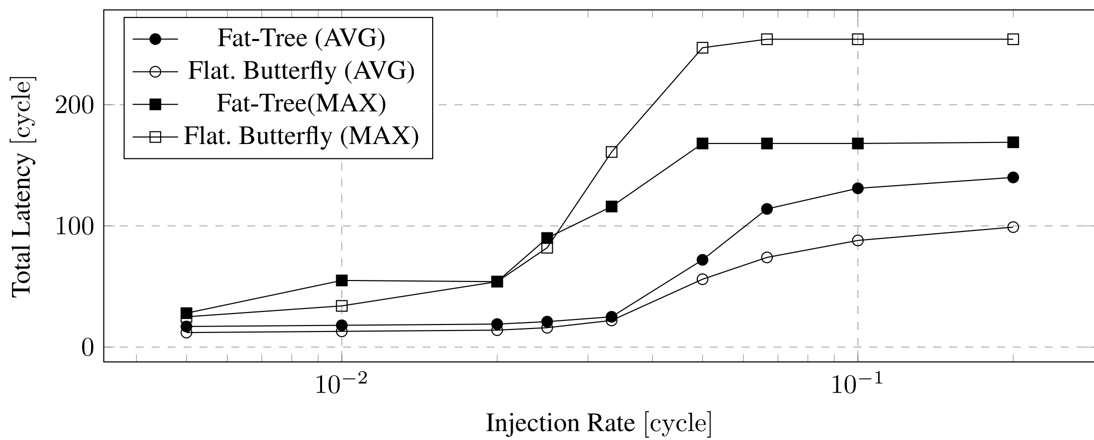


Figure 4.9: Latency vs. Injection Rate comparison for the 32 terminal networks on worst-case traffic pattern. Flit size is 16 bits and the flattened butterfly network utilizes DOR.

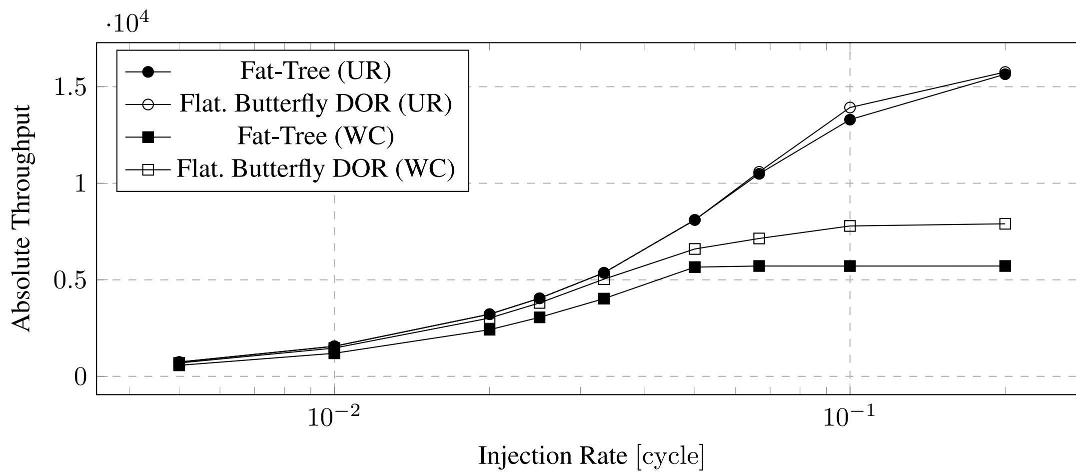


Figure 4.10: Throughput vs. Injection Rate comparison for the 32 terminal networks on unified random traffic and worst-case traffic. Flit size is 16 bits.

The comparison between the two networks with respect to throughput is depicted in Figure 4.10. On uniform random traffic they perform almost identical. Regarding the individual worst-case traffic patterns, the throughput of the flattened butterfly network is always higher than the throughput of the fat-tree network.

Considering the huge difference of the maximum latency for unified random pattern with comparable throughput, the flattened butterfly is much more suited as the interconnection network for the SCAD machine — provided the compiler is able to partition the given code to obtain a favorable traffic pattern. Is this not the case, the fat-tree network should be used because it reacts more generous to worst-case traffic patterns. Maximum latencies are considerable lower than on the flattened butterfly while having only slightly less throughput.

4.1.6 Impact of Network Size

To evaluate the impact of the network size on the performance, both networks have been implemented in smaller versions with support for 16 and eight terminals. Latency and absolute throughput (TP) were measured using the the same patterns used beforehand on the respective networks, an injection rate of 1/5 and a flit size of 16 bits. The flattened butterfly implementation only utilizes DOR since VALs performance was not convincing. Table 4.7 presents the obtained results.

Note that the fat-tree network with support for eight terminals appears to be very good. However, the used bit reverse traffic pattern for this size generates only four remote destinations where the remaining

Table 4.7: Impact of different network sizes on latencies and throughput under full load.

(a) Fat-Tree on WC pattern

	Latency		
Size	Avg.	Max.	TP
32	140	169	5715
16	108	129	3335
8	27	42	3237

(b) Fat-Tree on UR pattern

	Latency		
Size	Avg.	Max.	TP
32	65	152	15647
16	60	129	7745
8	35	69	5467

(c) Flattened Butterfly on WC pattern

	Latency		
Size	Avg.	Max.	TP
32	99	254	7901
16	79	214	4545
8	55	71	2727

(d) Flattened Butterfly on UR pattern

	Latency		
Size	Avg.	Max.	TP
32	52	90	15761
16	47	82	8429
8	37	66	4937

four target local destinations. Larger versions of both networks yield usual results comparable to those obtained in previous sections.

4.2 FPGA Resource Utilization

4.2.1 SCAD Components

Table 4.8 lists the reported FPGA resource utilization of chosen SCAD machine components except the interconnection network itself after synthesis. According to the synthesis report, the amount of required LUTs is often lower after physical optimization and full implementation by the tool chain. Unfortunately, the complete design was never run on the target FPGA and thus more accurate reports could not be obtained.

The almost more than double amount of required LUTs for the output queue is due to the complexity of the mechanism discussed in Section 3.4.2. Here, optimization potential is high. For example, the output queue behavior could be restricted to not allow three operations at the same time anymore. It is also the prime target for optimization since it is a subcomponent of every functional unit.

As expected, the difference between the network interfaces for the different flit sizes is only minimal. For 16-bit flits, less LUTs are required because only three different cycles are necessary compared to the

Table 4.8: Area consumption post-synthesis of non-interconnection related SCAD machine components. The subscript of components indicates the flit size (8-bit or 16-bit).

Component	LUTs	Flip-Flops
Input Queue	269 (0.51%)	287 (0.27%)
Output Queue	669 (1.26%)	294 (0.28%)
Network Interface$_8$	113 (0.21%)	142 (0.13%)
Network Interface$_{16}$	106 (0.20%)	150 (0.14%)
PU-ALU	113 (0.21%)	33 (0.03%)
FU-ALU	1601 (3.01%)	1142 (1.07%)
FU-ADD	1322 (2.48%)	1043 (0.98%)
Load&Store Unit	1173 (2.20%)	875 (0.82%)

six cycles for 8-bit flits. In return, a few more flip-flops are required to store the larger flit.

The individual listings of every component of the universal ALU functional unit allows to breakdown its resource requirement. It contains roughly 2.6 input queues[3], one output queue, the network interface and the processing unit itself. 86% of the required LUTs are used up by the queues. The remaining 14% are shared between the network interface and processing unit.

A similar result can be observed for the addition unit as well as other not listed units. The flip-flop count of the load-and-store unit is low because the contained memory is completely mapped into the FPGAs block RAMs. The same is true for the instruction memory not shown here. These two components use the block RAM resources exclusively.

Extrapolating the reported results for the 32 functional unit SCAD machine estimates nearly 90% used FPGA resources for non-interconnection related parts. Considering that not much time was spend on optimizing these components, changes the estimation to roughly 85% which leaves the remaining 15% for the interconnection implementation.

4.2.2 Interconnection Networks

Table 4.9 contains the FPGA resource utilization of both the fat-tree and flattened butterfly implementations without virtual channels. Increasing the flit size for the fat-tree network with support for 32

[3]The unused bits of the operation input queue are optimized away by the synthesizer.

Table 4.9: Resource utilization post-synthesis of different interconnection networks of different sizes.

(a) Fat-Tree

Net.	Flit	LUTs	Flip-Flops
8	8	564 (1.06%)	560 (0.53%)
8	16	616 (1.16%)	784 (0.74%)
16	8	1744 (3.28%)	1680 (1.58%)
16	16	1944 (3.65%)	2352 (2.21%)
32	8	4400 (8.27%)	4480 (4.21%)
32	16	5104 (9.59%)	6272 (5.89%)

(b) Flat. Butterfly (DOR)

Net.	Flit	LUTs	Flip-Flops
8	8	629 (1.18%)	368 (0.35%)
8	16	708 (1.33%)	480 (0.45%)
16	8	2050 (3.85%)	887 (0.83%)
16	16	2532 (4.76%)	1167 (1.10%)
32	8	5042 (9.48%)	2112 (1.98%)
32	16	6538 (12.29%)	2784 (2.62%)

(c) 32 Flat. Butterfly with two virtual channels (VAL)

Flit Size	LUTs	Flip-Flops
8	15912 (29.91%)	5120 (4.81%)
16	20041 (37.67%)	6240 (5.86%)

terminals, the LUT requirements increase by 16.0% and 40.0% for the amount of necessary flip-flops. On the flattened butterfly, increasing the flit size results in 29.6% and 31.8% higher utilization respectively.

Scaling the fat-tree network size down to 16 terminals, LUT utilization drops by about 61% regardless of the flit size. The same is true for the amount of necessary flip-flops which drops by 62.5%. Scaling this network down even further to just 8 terminals, the resource requirements are decreased by another roughly 67% on average for both flip-flops and LUTs.

Applying the same scaling to the flattened butterfly network without support for virtual channels, the LUT utilization is lowered by similar percentages to the fat-tree. However, the amount of required flip-flops drops only about 58% on average for 16 and 8 terminal networks regardless of flit size.

Table 4.9(c) also contains the amount of necessary resources for the flattened butterfly network with two virtual channels. It is obvious that the stated utilization of LUTs will not fit inside the target FPGA simultaneously with the remaining SCAD machine. The more than double volume of flip-flops was expected due to the effective doubling in input ports. Yet roughly thrice the amount of required LUTs was unexpected. These are mainly due to more and larger (de-)multiplexers. Perhaps the 2-dilated flattened butterfly shortly discussed in Section 2.2.2.1 is a possible solution. It would dispose the (de-)multiplexers required for the channel multiplexing and would additionally provide more bandwidth. However, the

main internal interconnection of a router would still require very big multiplexers. Because of the discussed problems, the flattened butterfly network with virtual channels is not considered any further.

Comparing the two network implementations in general, the fat-tree uses less LUTs and considerable more flip-flops across the all network and flit sizes. However, this is not problematic since there are much more flip-flop than LUT resources remaining after the previously discussed SCAD components have been implemented. When considering only the resource utilization with no regards to performance, the fat-tree implementation is the clear winner. However, since performance is also a very important factor, the choice between the two implementations is more difficult. If the required resources for the flattened butterfly without virtual channels are available, it is the better choice based on the measurements of Section 4.1. Otherwise the fat-tree implementation is no consolation prize and performs well-enough on average.

4.3 Related Work

Mello et al. published in [MTCM05] their findings about implementing virtual channels in the Hermes NoC (Network on Chip) architecture. This NoC utilizes the mesh topology and routing is done via packet switching. They augmented the Hermes NoC with a variable number of virtual channels (0–4) and prototyped a smaller 4x4 instance of the network on a FPGA evaluating the required resources depending on the number of used virtual channels. The implemented router architecture is very similar to the architecture discussed in Section 3.10. Differences are the use of adaptive TDM to multiplex the virtual channels as explained in Section 3.10.2 and larger buffers for the input ports of the routers.

For their testing they used a flit size of 16-bit and buffer lengths of eight which scale down as the number of virtual channel increases. For routing the XY dimension-ordered algorithm is utilized.

They reported an almost linear increase in resources when increasing the number of virtual channels both in required LUTs and flip-flops. The same results were observed in the area analysis of the previous section. Another conclusion was the smaller networks do not benefit enough of virtual channels to be worthwhile.

Gratz et al. discussed in [GKM^{+}06] the implementation and evaluation of the TRIPS NoC. This network uses a 2D 4x10 mesh topology utilizing X-Y dimension-ordered routing. It implements four virtual channels and wormhole routing to connect two processors with dozens of shared level 2 (L2) caches on a single *Application-Specific Integrated Circuit* (ASIC). They used a flit size of 16 bytes. One to five flits are necessary to transfer the up to 80 bytes long messages. Overall, each router has five ports

where each port is able to store two flits per virtual channel.

The implemented architecture differs to the virtual channel implementation of this thesis in that the input queues contain the demultiplexer necessary for the selection of the virtual channels. This allows a significantly smaller crossbar. They reported an area consumption of 11% of the chip die for the interconnection network, which is close to the percentage used to restrict the implemented networks.

They opted to increase the number of virtual channels instead of utilizing adaptive routing, trading area for speed. This also allowed lower complexity routers which are able to achieve a hop latency of just one cycle. Finally, they concluded that synthetic traffic patterns are not enough to evaluate interconnection networks. This is because “real workloads tend to be bursty and not evenly distributed in destination, altering the desired network operating parameters.” While this is true, unfortunately there are no real workloads for SCAD architecture yet to evaluate the performance this way.

Chapter 5

Conclusion

The main focus of the thesis at hand was to implement and evaluate suitable interconnection networks for the SCAD machine. To this end, the two most promising network topologies, namely fat-tree and flattened butterfly, have been implemented and extensively evaluated using minimal deterministic routing algorithms. Depending on the remaining resources on a FPGA, either network is applicable in its current form. Furthermore, the evaluation of both networks confirmed the respective claims of the authors in terms of performance and resource requirements which initially led to choosing them.

Additionally, the foundation for any adaptive routing algorithm on the flattened butterfly network was realized in the form of virtual channels. However, due to very restrictive resource utilization requirements, further development was hindered. The working and deadlock-free VAL routing algorithm proved that the virtual channel router design is operational in practice.

A minor focus of this thesis was also to prototype the SCAD machine in its initial form with 32 functional units. After the implementation of all components were finished, another SCAD machine concept with only eight universal functional units was introduced which rendered most of the already implemented parts obsolete. Moreover, a functioning compiler for the initial architecture was never completed in favor of the new design. This resulted in the implementation of improvised synthetic traffic generators to evaluate the performance of the implemented networks.

Fortunately, the realized interconnection networks are able to operate mostly independent of the remaining SCAD architecture. The utilized network interfaces transform the conceptional used tuples into a sequence of flits, transmits them to the destination unit and transforms them back. The only necessity

is that the destination address must be contained in the header flit. Thus, the implemented networks are abstract enough to allow simple modifications only in the network interface in order to adjust to the new SCAD machine design.

Although the implemented networks obtained similar throughput per cycle, it remains to be seen which maximum clock frequency can be achieved with either network. Based on the lower complexity of the fat-tree router which in turn should result in a shorter critical path, I would expect the fat-tree network to achieve higher clock speeds than flattened butterfly one, putting the achieved cycle based latency and throughput into a new perspective.

Furthermore, the decoupling of functional units and interconnection network via the network interface allows them to operate on different clock frequencies. The significantly lower complexity of routers— especially the fat-tree ones— should enable them to operate much faster and therefore the network should not bottleneck the SCAD machine.

Less ambitious requirements such as a SCAD machine with just 16 units would grant a lot of room for improvement. The addition of further virtual channels is expected to increase the general performance of the network assuming the routing algorithm exploits them. Speaking of, implementing adaptive routing algorithms would be feasible which should further increase performance. The input ports of the routers can also be improved by utilizing an input queue instead of the used one-flit buffer.

All things considered, this thesis proved that even with the severe imposed limitations not only one but two different interconnection networks for the SCAD architecture with 32 functional units are realizable.

Bibliography

[BJS16] A. Bhagyanath, T. Jain, and K. Schneider. Towards code generation for the synchronous control asynchronous dataflow (SCAD) architectures. In R. Wimmer, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 77–88, Freiburg, Germany, 2016. University of Freiburg.

[DA93] William J. Dally and Hiromichi Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE transactions on Parallel and Distributed Systems*, 4(4):466–475, 1993.

[Dal92] William J Dally. Virtual-channel flow control. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):194–205, 1992.

[DS87] William J Dally and Charles L Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *Computers, IEEE Transactions on*, 100(5):547–553, 1987.

[DT04] W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[GKM+06] Paul Gratz, Changkyu Kim, Robert McDonald, Stephen W Keckler, and Doug Burger. Implementation and evaluation of on-chip network architectures. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 477–484. IEEE, 2006.

[KBD07] John Kim, James Balfour, and William Dally. Flattened butterfly topology for on-chip networks. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182. IEEE Computer Society, 2007.

[KDA07] John Kim, William J Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. *ACM SIGARCH Computer Architecture News*, 35(2):126–137, 2007.

[KDDA06] John Kim, William Dally, J Dally, and Dennis Abts. Adaptive routing in high-radix clos network. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 7–7. IEEE, 2006.

[KNDR01] Faraydon Karim, Anh Nguyen, Sujit Dey, and Ramesh Rao. On-chip communication architecture for oc-768 network processors. In *Proceedings of the 38th annual Design Automation Conference*, pages 678–683. ACM, 2001.

[KSJ04] Nikolay Krasimirov Kavaldjiev, Gerardus Johannes Maria Smit, and Pierre G Jansen. A virtual channel router for on-chip networks. 2004.

[Lei85] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, 100(10):892–901, 1985.

[MTCM05] Aline Mello, Leonel Tedesco, Ney Calazans, and Fernando Moraes. Virtual channels in networks on chip: implementation and evaluation on hermes noc. In *Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 178–183. ACM, 2005.

[MWM04] Robert Mullins, Andrew West, and Simon Moore. Low-latency virtual-channel routers for on-chip networks. In *ACM SIGARCH Computer Architecture News*, volume 32, page 188. IEEE Computer Society, 2004.

[RLP06] Ville Rantala, Teijo Lehtonen, and Juha Plosila. Network on chip routing algorithms. 2006.

[Sin05] Arjun Singh. Load-balanced routing in interconnection networks. 2005.

[TC10] Ajithkumar Thamarakuzhi and John A Chandy. 2-dilated flattened butterfly: A nonblocking switching network. In *High Performance Switching and Routing (HPSR), 2010 International Conference on*, pages 153–158. IEEE, 2010.

[TC11] Ajithkumar Thamarakuzhi and John A Chandy. 2-dilated flattened butterfly: A nonblocking switching topology for high-radix networks. *Computer Communications*, 34(15):1822–1835, 2011.

[Xil15] Xilinx. *Vivado Design Suite User Guide Synthesis*, 2015.