Acknowledgements

I would like to take the opportunity and thank my advisor, Prof. S.K. Nandy, whose insights, encouragement and suggestions helped tremendously to finish this work successfully. Further I am grateful for his patience and support he showed towards me during my administrative and legal obligations such as visa applications and extensions.

Secondly I express my gratitude towards Dr. Ranjani Narayan, CTO of Morphing Machines, for her moral and technical support during my stay in India. I am sure this work would have not been possible without her help.

Besides I would like to thank my lab mates, especially Mythri Alle, Keshavan Varadarajan, Ganesha K. Garga, S. Balakrishnan and many more for testing my implementations and reporting bugs. They always had time to listen to problems that occurred. Their (sometimes unconventional) suggestions from a different point of view helped solving problems on many occasions. This thesis would not have been possible without the help of Niraj Sharma of Bluespec, Inc. During the realization of RECONNECT he always patiently listened to big and small implementation problems alike. Some of his suggestions for workarounds can still be found in the code and the forwarding the bugs reported to him, improved the Bluespec System Verilog (BSV) compiler.

This acknowledgement would not be complete without the mentioning of the International Relation Cell (IRC) of IISc whose guidance and experience in visa related issues and organizational skills made my stay a very pleasant one. I was able to learn many interesting qualities about the Indian culture and I am sure that this exposure effectively changed my perception of life.

Lastly I would like to thank all my friends from within and outside the institute for welcoming me into their families to give me an insight view into their lives. Further they were always present for an exhausting game of basketball to release excessive energy.
Abstract

In this thesis a Network on Chip (NoC) router implementation called RECONNECT realized in BSV is presented. It is highly configurable in terms of flit size, the number of provided Input Port (IP)/Output Port (OP) pairs and support for configurations during runtime, to name a few. Depending on the amount of available IP/OP pairs, the router can be integrated into different topologies. Due to the ability to be configured during runtime, the router can even support multiple topologies. A developer is then able to choose among the available topologies the one that promises the highest performance for an application. However this work only concentrates on tessellations like toroidal mesh, honeycomb and hexagonal.

Routing algorithms that were needed to be developed or adapted, are presented. In addition a step-by-step example of the routing algorithm development for the honeycomb topology is included in this thesis. This enables a system designer who wishes to use RECONNECT for any other topology that is not discussed such as a hypercube or a ring, to develop the required routing algorithm easily and fast.

The impact of the chosen topology on the execution time of several real life algorithms has been analyzed by executing these algorithms on a target architecture called REDEFINE, a dataflow multi-processor consisting of Compute Elements (CEs) and Support Logic (SL). For this purpose an NoC comprising of RECONNECT routers establishing communication links among the CEs, has been integrated into REDEFINE. It has been found out that for very small algorithms, the execution time does not depend on the choice of topology, whereas for larger applications such as Advanced Encryption Standard (AES) encryption and decryption, it becomes evident that the honeycomb topology performs worst and the hexagonal one best. However it is observed that in many cases the additional links that are provided by the hexagonal topology, when compared with the mesh, are not utilized due to the topology unawareness of the REDEFINE SL. Hence the algorithm execution time for mesh topology is often on par with hexagonal ones.

In addition to the chosen topology it is investigated, how the size of the flit affects these algorithms. As expected the performance of the NoC decreases, if the flit size is reduced so that the packets have to be segmented into more flits. Further it is analyzed, if the NoC performance is sufficient
to support high level algorithms such as e.g. the H.264 decoder through which data is streamed. These algorithms require to perform the necessary computations not only within a time constraint, but also the data needs to be fed to the Processing Elements (PEs) fast enough. In H.264 the time constraint is the frame rate meaning that each frame need to be processed in a specified fraction of a second. The current RECONNECT implementation does not qualify to deliver the data within this requirement. As a result, the necessity for a pipelined router version is presented.

To allow a fair comparison of network performance with implementations found in current literature and to validate this approach, the NoC has been put under stress by artificial traffic generators which could be configured to generate uniform and self-similar traffic patterns. Further different destination addresses generation algorithms such as normal (randomly selecting a destination located anywhere in the network), close neighbor communication, bit complement and tornado, for each of these traffic patterns have been developed. It could be observed that in general the honeycomb topology performs worst, followed by the mesh and topped by the hexagonal topology. From the artificial traffic generators it can be concluded that the richer the topology, the higher the throughput.

The different router designs have been synthesized to gain approximate area and power consumption details. Depending on the flit size the single cycle router which is able to forward an incoming flit in the next clock cycle, if no congestion occurs, dissipates between 13 and 35mW for honeycomb topology operating at a frequency of 450MHz. The power increases by approximately 25% for each IP/OP pair that is added to the router integrated in a honeycomb topology. The area that is required for a router in a honeycomb network, has been found out to be between 96167 and 301339 cells depending on the flit size. A router supporting a mesh or a hexagonal topology needs respectively 50% or 91% more area than the honeycomb router.

Depending on the flit size the pipelined version of the router dissipates between 70 and 270, 75 and 294, and 85 and 337mW for the honeycomb, mesh and hexagonal topologies respectively. The area that is required for a single router, is between 213898 and 839334 for honeycomb, 238139 and 957548 for mesh, or 286328 and 1182129 cells for hexagonal router configurations. The tremendous increase of both power dissipation and area consumption is caused by the additional buffers that are required for each stage. The maximum clock frequency of the pipelined version has reached 1.4GHz.
# Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xv</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>xvii</td>
</tr>
</tbody>
</table>

1 Introduction                         | 1   |
| 1.1 Motivation                        | 1   |
| 1.2 Definition of commonly used Terms | 4   |
| 1.3 A short Introduction into Bluespec System Verilog (BSV) | 5 |
| 1.3.1 Implicit and Explicit Conditions| 8   |
| 1.4 Organization                     | 9   |
| 1.5 Summary                           | 10  |

2 Related Work                         | 11  |
| 2.1 Topologies                        | 11  |
| 2.2 Routing Algorithms                | 12  |
| 2.2.1 Deterministic Routing Algorithms| 13  |
| 2.2.2 Adaptive Routing Algorithms     | 13  |
| 2.3 Flow Control                      | 14  |
| 2.4 Impact on RECONNECT               | 16  |
| 2.5 Summary                           | 16  |

3 Architecture                          | 17  |
| 3.1 Architectural Overview of the Router| 17  |
| 3.2 Configuration Parameters          | 18  |
| 3.2.1 DEBUG_NOC[1-3]                  | 19  |
| 3.2.2 ASMUNIT_DEBUG                   | 19  |
3.2.3 MULTIFLITSUPPORT ........................................ 19
3.2.4 STAGES_OF_ROUTER ........................................ 20
3.2.5 IP_VC, FIFO_DEPTH_VC_IP .................................. 20
3.2.6 PORTS, PORTS_MESH, PORTS_HC .......................... 21
3.2.7 NWFLITSIZE .................................................. 21
3.2.8 NUM_ADDRESSES_ROUTER .................................... 21
3.2.9 OPSLENGTH .................................................... 22
3.2.10 HEADERSIZE, BITS_SPACE_IN_HEADFLIT ............... 22
3.2.11 MULTITOPOLOGY ........................................... 22
3.2.12 XBAR ......................................................... 23
3.2.13 EJECTPORT and (.*)(_FABRIC)? ......................... 24
3.2.14 FLTINW_FIFOSIZE_ASMUNIT ............................... 24
3.2.15 CE_CLK and SUPPORTLOGIC_CLK ....................... 24
3.3 Input Port (IP) (InputPort.bsv) .............................. 25
3.3.1 Assembly Unit (AU) (AssemblyUnit.bsv) ............... 26
3.3.2 IP connected to OP ........................................ 29
3.3.2.1 Matrix Arbitrators (Arbiter([0-9]+).bsv) ............ 29
3.4 Crossbar ....................................................... 33
3.5 Output Port (OP) (OutputPort.bsv) .......................... 33
3.6 Pipelined Routers ............................................ 33
3.6.1 Pipelining of the Single Cycle Router Implementation .. 34
3.6.2 Changes in the Implementation ........................... 36
3.7 Summary ....................................................... 38

4 Routing Algorithms (Routing(.*).bsv) and Topologies .......... 39
4.1 Preliminaries .................................................. 39
4.1.1 Example .................................................... 40
4.2 Topologies ..................................................... 42
4.2.1 Flattened Butterfly ....................................... 44
4.2.2 Spidergon and Stargon Topology .......................... 44
4.3 Virtual Channels (VCs) ...................................... 47
4.4 Honeycomb Topology .......................................... 49
4.4.1 Algorithm .................................................. 53
4.4.1.1 Behavioral Observations ............................... 53
4.4.1.2 if Branch Aggregation .................................. 56
4.4.1.3 Virtual Channel Optimization .......................... 58
4.4.1.4 Input Port Optimization ................................ 60
4.4.2 Limitations of the Routing Algorithm ..................... 62
4.5 Mesh Topology ................................................ 62
4.6 Hexagonal Topology .......................................... 64
4.6.1 Proof ....................................................... 65
4.7 Summary ....................................................... 66
List of Figures

1.1 A point-to-point connection always between a pair of PEs. . . 2
1.2 A pipelined bus system . . . . . . . . . . . . . . . . . . . . . . 3
1.3 Logic block to calculate $b = a \times 4 + 3$. . . . . . . . . . . . . 6

3.1 The processing steps a flit encounters while traversing a router. 17
3.2 An architectural overview of the several modules of a router. . 25
3.3 An example of a packet. Here the packet type is an Instruction Packet of REDEFINE. The payload bits at the LSB and the template bits at the MSB are separated by unused bits (gray field). Bluespec initializes unused bit with 0xa in the simulator. 27
3.4 In this example multiple flits are generated from an Instruction Packet type. The flit size is set to 18 bits and the router are operating in an environment in which `NUM_ADDRESSES_ROUTER are set to 2 and the routing algorithm does not depend on VCs. 28
3.5 3-way matrix arbiter: The figure shows, how the grant for request0 is calculated. . . . . . . . . . . . . . . . . . . . . . . . 30
3.6 How a deadlock situation can occur, if the arbiter have the notion of strictly serving the oldest VC first. Although the original implementation seems to be fairer, it certainly does look so locally. However in a global point of view, unfortunate situations such as this one, occur. Without additional hardware like fixing the VC for a specific amount of cycles, increases hardware complexity. This needs to be avoided, since the arbitration is in the critical path. . . . . . . . . . . . . . . . . . . . . . . . 32
3.7 The muxes and demuxes of a stateless crossbar [60]. . . . . . 33
3.8 The internal structure of the butterfly crossbar allowing the implementation of pipelined routers. . . . . . . . . . . . . . . . . . . . 35
3.9 While flit A is in transit, flit B is sent by another IP to the same OP and VC like flit A. Assuming the VCs can store one flit only and after flit A is stored, flit B has to wait in the last pipeline stage till flit A can proceed further. . . . . . . . . . . . . . . . . . . . . . . . 36

4.1 Different layouts of the honeycomb and hexagonal topology. . 43
4.2 The mapping of a honeycomb and mesh topology into a hexagonal one. The thick lines are the ones used for the honeycomb topology. The gray nodes are the Access Routers (ARs) providing connectivity to the Fabric. ...

4.3 A 4 × 4 flattened butterfly topology in which all nodes are fully connected row and column wise. ...

4.4 The logical and physical layouts of Spidergon and Stargon topologies. ...

4.5 Two examples for cyclic dependencies that can occur in honeycomb topologies. ...

4.6 Prohibited turns according to the Turn-Model. ...

4.7 Two examples how the Turn-Model increases the latency of a message significantly. ...

4.8 Two layers of the network each of them with a different routing algorithm. ...

4.9 Bidirectional links for the toroidal structure create 2 cyclic dependencies additionally which are broken by increasing the number of VCs and by introducing a date line at link $T_{horizontal}$. ...

4.10 A 4 × 4 non-toroidal honeycomb topology. Some of the honeycombs are incomplete. ...

4.11 The turns that are forbidden in the mesh topology are marked. This results in routing rules in which the west direction has to be considered first. The dotted lines show the location of the date lines. ...

4.12 Multiple possibilities are provided to describe the position of $P$. ...

5.1 Overview of the flow for compiling an application written in C for REDEFINE by generating Hyper Operations (HyperOps) ...

5.2 An overview of REDEFINE and its major modules and their relationship among each other. ...

5.3 The design of the topology in the early stages of REDEFINE. ...

5.4 The impact on the number of flits and the amount of unused bits in the payload field, if the flit sizes varies. Here the number of flits that traverse through the network during the execution of the CRC application is shown. ...

5.5 The mapping of the HyperOps of the CRC application onto the Fabric. CRC consists of 3 HyperOps: 2 of the are mapped onto one CE only (striped area around (0,0)) whereas the larger one occupies 2 CEs (grayed out area). The thick links exist for the honeycomb topology, the mesh topology consists of the thick links and the thin link, whereas hexagonal also includes the dotted link. ...

5.6 The Fabric Execution Time against the maximum flit size for CRC in various topologies. ...
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7</td>
<td>A snapshot of the location of multiple HyperOps of the AES-D on the Fabric. Like in figure 5.5 the format of the interlinks represent the different topologies they belong to.</td>
<td>78</td>
</tr>
<tr>
<td>5.8</td>
<td>The Fabric Execution Time against the maximum flit size for AES decryption in various topologies.</td>
<td>79</td>
</tr>
<tr>
<td>5.9</td>
<td>The Fabric Execution Time against the maximum flit size for SOBEL edge detection in various topologies.</td>
<td>80</td>
</tr>
<tr>
<td>5.10</td>
<td>The row and column wise distances of nodes as seen from the black one. In a toroidal $6 \times 6$ Fabric the distances for each dimension cannot exceed three hops.</td>
<td>83</td>
</tr>
<tr>
<td>6.1</td>
<td>Latency for uniform traffic patterns and various address generation methods.</td>
<td>88</td>
</tr>
<tr>
<td>6.2</td>
<td>Throughput for uniform traffic patterns and various address generation methods.</td>
<td>89</td>
</tr>
<tr>
<td>6.3</td>
<td>Latency for self similar traffic patterns and various address generation methods.</td>
<td>90</td>
</tr>
<tr>
<td>6.4</td>
<td>Throughput for self similar traffic patterns and various address generation methods.</td>
<td>91</td>
</tr>
<tr>
<td>6.5</td>
<td>Latency for self similar traffic patterns and various address generation methods.</td>
<td>93</td>
</tr>
<tr>
<td>6.6</td>
<td>Throughput for self similar traffic patterns and various address generation methods.</td>
<td>94</td>
</tr>
<tr>
<td>6.7</td>
<td>Latency for self similar traffic patterns and various address generation methods.</td>
<td>95</td>
</tr>
<tr>
<td>6.8</td>
<td>Throughput for self similar traffic patterns and various address generation methods.</td>
<td>96</td>
</tr>
<tr>
<td>7.1</td>
<td>Power and area consumption of a single router for various flit sizes. The drop at 116 bit is caused by the removal of the AU whose functionality of segmenting packets into multiple flits is not required at these sizes anymore.</td>
<td>102</td>
</tr>
<tr>
<td>7.2</td>
<td>Power and area consumption, if the VC has to hold one packet eventually comprising of multiple flits.</td>
<td>104</td>
</tr>
<tr>
<td>7.3</td>
<td>Power and area consumption of a single pipelined router using a butterfly as a crossbar for various flit sizes. The drop at 116 bit is caused by the removal of the AU whose functionality of segmenting packets into multiple flits is not required at these sizes anymore.</td>
<td>107</td>
</tr>
<tr>
<td>A.1</td>
<td>AES Encryption algorithm</td>
<td>117</td>
</tr>
<tr>
<td>A.2</td>
<td>Elliptic Curve Point (ECP) Addition (ECPA)</td>
<td>118</td>
</tr>
<tr>
<td>A.3</td>
<td>Elliptic Curve Point (ECP) Doubling (ECPD)</td>
<td>118</td>
</tr>
<tr>
<td>A.4</td>
<td>GIVENS algorithm</td>
<td>119</td>
</tr>
</tbody>
</table>
A.5 Lower upper matrix factorization .......................... 119
A.6 Matrix multiplication ......................................... 120
A.7 MIRIQ algorithm .............................................. 120
A.8 Secure Hash Algorithm (SHA) version 1 ............... 121
## List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Properties of commonly used topologies</td>
<td>5</td>
</tr>
<tr>
<td>4.1</td>
<td>Routing function naming convention</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>The size of the different packet types used by REDEFINE in the Fabric. Displayed is the required bus width to transfer the packet as a whole, which apart from the payload itself also includes the address tuples (3 × 4 bits), VC number (2 bits) and template/union bits stating which packet type is valid (4 bits).</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Occurrences of the various packet types during the execution of a CRC against the sizes of the packets.</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>Occurrences of the various packet types during the execution of an AES decryption against the sizes of the packets.</td>
<td>79</td>
</tr>
<tr>
<td>5.4</td>
<td>Distances the flits have to travel during the execution of various applications</td>
<td>82</td>
</tr>
<tr>
<td>7.1</td>
<td>Maximum clock frequency of the router reported by the design compiler in pre-synthesis phase.</td>
<td>109</td>
</tr>
</tbody>
</table>
List of Acronyms

**ACK**  Acknowledgement

**AES**  Advanced Encryption Standard

**ALU**  Arithmetic Logical Unit

**AR**  Access Router

**ASIC**  Application Specific Integrated Circuit

**AU**  Assembly Unit

**BB**  Basic Block

**BSV**  Bluespec System Verilog

**CE**  Compute Element

**CRC**  Cyclic Redundancy Check

**DFG**  data flow graph

**degree**  The degree of a router is defined as the number of links, it can establish to its neighbors and excludes the injection and ejection port.

**ECP**  Elliptic Curve Point

**ECPA**  Elliptic Curve Point (ECP) Addition

**ECPD**  Elliptic Curve Point (ECP) Doubling

**FIFO**  first-in-first-out

**flit**  A flit is the largest amount of data that can be transmitted between two routers in a clock cycle. The size of the flit depends on the amount of available wires laid out between the routers.

**HL**  HyperOp Launcher

**HyperOp**  Hyper Operation
IHDF Inter-HyperOp Data Forwarder
IP Input Port
IPC Intellectual Property Controller
LIFO last-in-first-out
LSU Load/Store Unit
NoC Network on Chip
OP Output Port
PE Processing Element
pHyperOp partial HyperOp
radix The radix is the total amount of IPs and OPs a router can establish not only to its neighbors, but also to the attached units as well. Hence it includes also the injection and ejection port.
RB Resource Binder
SHA Secure Hash Algorithm
SL Support Logic
union A union in BSV is comparable with a union known in the programming language C. It can contain several structures such as packet structures, but only one is valid in any point in time. To distinguish which packet type is valid, the BSV compiler automatically add template bits according to the entries in the union.
VC Virtual Channel
VLSI Very Large Scale Integration
Chapter 1

Introduction

In this chapter the evolvement of the Network on Chips (NoCs) originating from point-to-point connections and bus systems including the motivation for this work is described. After defining common terms that are used throughout this thesis, the reader will be exposed to a short introduction into Bluespec System Verilog (BSV) in which RECONNECT has been implemented.

1.1 Motivation

Within a chip, complex systems and multi-core architectures consist of many units called Processing Elements (PE) that are either highly specific to perform a single task very efficiently, or comprise Arithmetic Logical Units (ALU) for generic operations. The high level of integration of multiple (in an order of tens or even higher) PE promise to satisfy the demand for computation power needed as of today. On the other hand these systems and architectures require a high speed communication system to exchange data among the PE within the chip. One method is to analyze the traffic patterns of the executed application and building a communication system exactly matching these patterns by directly connecting the data exchanging PE as shown in figure 1.1. While this point-to-point communication system is considered to be the fastest, it restricts the system designer to a few applications only. In addition depending on the richness of the system, the wiring requirements explode for n-to-n connections.

A solution of this dilemma is rearranging the PE to be connected via a cost efficient bus system, a shared resource in which only one PE can transmit data at a time. A control logic manages access granted to the bus. While this communication system gives the desired flexibility, it lacks scalability. In a magnitude of a thousand PE, which can be integrated easily in modern VLSI technology, the wires of the bus become very long and have a high capacitance resulting on long delays and high power consumption [39]. In the view of the allowed access of a single PE at a time, the throughput is
very low. Thus bus systems are only used, if a few tens of PEs need to communicate.

To increase the utilization level bridges basically consisting of a set of first-in-first-out (FIFO) buffers, are inserted into the bus system, dividing the bus into several subsets (refer to figure 1.2a). If two PEs within a subset communicate, they can do so provided no other PE of the same group uses the bus at the same moment. With these bridges multiple PEs in different subsets are able to transmit data at the same time, since the bridges are opaque and the traffic cannot cross them. If the destination of the data resides in a subset outside of the current one, the bridge turns transparent and lets the traffic pass through it. If the bus consists of multiple bridges it can also be considered as a pipelined bus with each bridge representing one pipeline stage as shown in figure 1.2d. Each subset has its own arbitration control granting access to the bus segment either to one PE or to one bridge.

However the time a message requires to cross the bus system from one side to the logically other, is immense. Depending on the amount of bridges and the size of the system, it can be in the order of thousands of clock cycles. In addition orchestration of the access to each subset and keeping track of the traffic crossing through several bus segments, increases the complexity in the global arbitration control.

By rearranging the PEs into a more beneficial pattern such as a grid to cut short the long distances (refer to figure 1.2c), and by localizing the access control, the disadvantages of pipelined buses can be avoided. By now the access control not only regulates the grants to the shared bus resource, but also forwards the traffic into one of the multiple directions available.
1.1 Motivation

(a) The bus is divided into two subsets separated by a bridge which is opaque, if the source and destination of a communication pair is within the same subset. Otherwise the bridge is transparent allowing the traffic to pass through it.

(b) The bus is further divided by using more bridges.

(c) Rearrangement of the PEs to shorten the distances and decentralizing the access control logic.

Figure 1.2: A pipelined bus system
1.2 Definition of commonly used Terms

The pattern in which the bridges and their connections among each other are arranged is henceforth called topology. The accumulation of the access control including the direction decision logic is called a router. Topology and routers form an Network on Chip (NoC). The NoC including the PEs among which the connectivity is established, is referred to as Fabric.

An encapsulated piece of information outside the Fabric is called a packet. If it is inserted into the NoC it is converted into a protocol that the router understands. This might include the necessity to divide the packet into several flits. Flits represent the largest amount of bits that can be transmitted at one instance of time between two routers. For instance a packet of a size of 80 bits is divided into 4 flits of 20 bits each. To transmit the flit, the connection between two routers need to consist of 20 wires at least.

The number of connections that are established from one routers to all its neighbors, is referred to as degree. In figure 1.2c the routers located in the corners have a degree of two whereas the routers in the middle a degree of three. Usually the maximum degree only is given while listing the specifications of the NoC. The degree depends on the topology that a router is integrated into, and it differs from the term radix, which represents the total number of connections a router provides including the links to all connected PEs. Again referring to the same figure the radix of the corner router is three whereas the radix of the routers in the middle is four. Assuming each router has 4 PEs connected to it, the radix becomes 7 and 8 respectively.

Besides the characteristics for a router, the NoC can be described by the bisection bandwidth which is obtained by dividing the network into two disjoint sets of nearly equal size. The partition with the lowest number of connections originating in one set and ending in the other one, determines the bisection bandwidth.

For the system designer deciding on the topology, it is important to know the maximum number of hops a flit has to travel to the source and destination that are farthest apart. This characteristic is called diameter of the network. Table 1.1 compares these characteristics in an overview.

The time a flit traverses through the network is referred to as latency. Usually system designers are interested not only when the flit arrives, but also when the packet consisting of multiple flits, is ejected from the network and available for processing. Hence the latency also includes the time needed for segmentation and reassembly of the packet (serialization latency). The higher the number of flits and hence the higher the network load, the higher is the latency due to the occurrence of congestions, in which flits have to share a common path. The network load can reach a point in which the throughput saturates. If the saturation point has been reached, the router does not always accept injected packets anymore and these packets need to be queued. Throughput depends on the underlaying topology, the clock
### 1.3 A short Introduction into Bluespec System Verilog (BSV)

RECONNECT is implemented in the high-level language BSV\[6, 12, 11\] which allows to compile it into a clock accurate simulator that can be executed on ordinary end user PCs, but also to compile it into Verilog files for further processing by e.g. synthesis tools. By using one code for simulation and synthesis, code maintenance is minimized, since consistency does not need to be ensured among several implementations. In addition BSV allows to use certain constructs which are hard to understand for developers coming from hardware design background, but are well known to software programmers. On the other hand many times the software programmer lacks the experience and knowledge of hardware programming. As an example, if

\[
b = a \times 4 + 3
\]  

needs to be calculated, a software developer will most likely write the equation directly into the program. However the hardware engineer knows that the multiplication is a simply shift of \(a\) by 2 bits to the left and the addition by 3 means that the last 2 bits of \(b\) are set after assignment of \(a\) to \(b\). Equation \[1.1\] is equivalent to

\[
b = (a << 2) | 0x3
\]  

#### Table 1.1: Properties of commonly used topologies

<table>
<thead>
<tr>
<th>Topology</th>
<th>Degree (deg)</th>
<th>Diameter (dia)</th>
<th>Bisection Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh</td>
<td>4</td>
<td>(2\sqrt{n})</td>
<td>(\sqrt{n})</td>
</tr>
<tr>
<td>Honeycomb</td>
<td>3</td>
<td>(1.16\sqrt{n})</td>
<td>0.82(\sqrt{n})</td>
</tr>
<tr>
<td>Honeycomb (rectangular)</td>
<td>3</td>
<td>(2\sqrt{n})</td>
<td>0.5(\sqrt{n})</td>
</tr>
<tr>
<td>Hexagonal</td>
<td>6</td>
<td>(1.16\sqrt{n})</td>
<td>2.31(\sqrt{n})</td>
</tr>
<tr>
<td>Hexagonal (rectangular)</td>
<td>6</td>
<td>(2\sqrt{n})</td>
<td>(2\sqrt{n} - 1)</td>
</tr>
</tbody>
</table>

#### Toroidal Topologies

<table>
<thead>
<tr>
<th>Topology</th>
<th>Degree (deg)</th>
<th>Diameter (dia)</th>
<th>Bisection Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh</td>
<td>4</td>
<td>(\sqrt{n})</td>
<td>(2\sqrt{n})</td>
</tr>
<tr>
<td>Honeycomb</td>
<td>3</td>
<td>0.81(\sqrt{n})</td>
<td>2.04(\sqrt{n})</td>
</tr>
<tr>
<td>Honeycomb (rectangular)</td>
<td>3</td>
<td>(\sqrt{n})</td>
<td>(\sqrt{n})</td>
</tr>
<tr>
<td>Hexagonal</td>
<td>6</td>
<td>0.58(\sqrt{n})</td>
<td>4.61(\sqrt{n})</td>
</tr>
<tr>
<td>Hexagonal (rectangular)</td>
<td>6</td>
<td>(\sqrt{n})</td>
<td>(4\sqrt{n} - 2)</td>
</tr>
<tr>
<td>Hypercube</td>
<td>(\log n)</td>
<td>(\log n)</td>
<td>0.5 \times n</td>
</tr>
</tbody>
</table>

frequency of the router and the flit size.
avoiding the synthesis of power intensive multiplication and addition logic potentially requiring multiple clock cycles for calculation (refer to figure \[1.3\]). BSV tries to fill the gap between hardware and software developers by providing very abstract high level language constructs, but also allows the hardware engineers to do the operations that they are used to. The implementation of the router makes use of these high level abstractions. Hence this section introduces and explains some of them to ease the understanding of the code of the routers.

As described in chapter \[3\] the router needs differently behaving routing algorithms depending on its location in the Fabric. It would be programming overhead and cause a considerable effort spent for code maintenance, if the code of the routers is duplicated. Hence the routing algorithms as functions are given to the router module as parameters. The following listing shows the module declaration of the router:

```verilog
module mkNoCRouterXBar#(
    function Tuple2#(Bit#('STAGES_OF_ROUTER), UInt#(TLog#('IP_VC)))
        getRouteHoneycomb(
            Vector#('NUM_ADDRESSES_ROUTER, Int#('ADDRLENGTH)) addr,
            UInt#(TLog#('IP_VC)) _vcNo,
            Integer number),
    parameter int routerNumber, Bool northHoneycomb
    (RouterXBar_IFC);
```

In this example in total three parameters are given to the module `mkNoCRouterXBar`: a function which can be invoked by the name `getRouteHoneycomb` from within the module, and two parameters, `routerNumber` and `northHoneycomb`. The list of arguments given to the function (`addr`, `_vcNo` and `number`) is a prototype so that the BSV compiler becomes aware of, how to handle that function. Since giving a function as parameter for a module is not a Verilog construct, this module cannot be synthesized. That is why it is not recommended to instantiate `mkNoCRouterXBar` directly, but through a trick to make it synthesizable. BSV allows to return modules that are instantiated from a parent module which assumes the same functionality:

![Figure 1.3: Logic block to calculate $b = a \times 4 + 3$.](image)
1.3 A short Introduction into Bluespec System Verilog (BSV)

(* synthesize *)
module mkNoCRouterXBarNormalS#(parameter int routerNumber)
        (RouterXBar_IFC);
        let router <- mkNoCRouterXBar(
            getRouteHoneycombS2DVC,
            routerNumber,
            False);
        return router;
endmodule

As it can be observed the function that is supposed to be used as get-
RouteHoneycomb inside the module mkNoCRouterXBar is named getRoute-
HoneycombS2DVC. It can be any function to change the behavior of the mod-
ule, as long as it follows the list of arguments and return parameter format
of the prototype. Since there is no longer an unknown Verilog construct in
the header of mkNoCRouterXBarNormalS, it can be synthesized by inserting
(* synthesize *) before the module keyword.

Not only functions can be given as parameter to a module, but also
modules themselves. This functionality is used for instantiating the crossbar
(mkXBar):

XBAR_IFC xbar <- mkXBar(
        'STAGES_OF_ROUTER,
        mkMerge2x1_multipleFlit,
        routerNumber
);

The module header of mkXBar is similar to the one of mkNoCRouterXBar:

module mkXBar#(Integer logn,
        module #(Merge2x1_IFC) mkMerge2x1, parameter int id)
        (XBAR_IFC);

In that case mkMerge2x1_multipleFlit is a module which instantiates
one node of the crossbar. Depending on the required functionality, mk-
Merge2x1_multipleFlit can be replaced. Illustrations are given in the file
XBar.bsv.

XBar.bsv is an example of one more specialty of BSV: Depending on the
number of input and output ports, multiple stages need to be instantiated
(refer to figure 3.8 on page 35). By generating one more stage, the previous
stages are instantiated two times and the upper stage is connected to the
lower nodes of the newly created stage, whereas the lower previous stage
is connected to the upper ones. This is repetitive for every stage. Hence
depending on the number of required stages, mkXBar is instantiating itself
recursively \([\log_2 n]\) times with \(n\) being the number of required ports.
Apart from the well known constructs in the world of software engineering, BSV provides functionality for e.g. linked lists of yet unknown size at the time of development. It is possible to walk through those lists, append elements and remove them. However at the time of compilation, everything needs to be fixed, since hardware obviously cannot be invoked during runtime.

1.3.1 Implicit and Explicit Conditions

The functionality a BSV module has to provide, is implemented in rules and methods. Rules are executed whenever conditions allow, whereas methods are more comparable with function calls in C taking parameters as inputs and returning values.

For instance the following rule is automatically executed every clock cycle, since there is no condition preventing the execution:

```
rule incrementCounter;
    counter <= counter + 1;
endrule
```

with `counter` being a register. If desired, the rule can be extended to execute only, if certain conditions are met. As an example the rule above has been extended to increase the counter value only, if a register called `start` is set to true.

```
rule incrementCounter(start);
    // or rule incrementCounter(start == True);
    counter <= counter + 1;
endrule
```

The added condition is called an explicit condition and is enforced by the system designer. Many times more conditions are required which are not immediately visible to the designer. For instance a rule that writes to a FIFO buffer cannot do so, if the buffer is already full. These conditions are automatically inserted by the BSV compiler and are called implicit conditions.

While these implicit conditions are very helpful and for instance can be used to check, if an Input Port (IP) is ready to receive a flit, they sometimes create misunderstandings resulting in long debug sessions. Consider the following rule as an example:

```
rule getData; // get data and store in register reg
    if(getDataFromOne) begin
        reg <= fifo1.deq(); // dequeue the data from fifo 1
    end else begin
        reg <= fifo2.deq(); // dequeue the data from fifo 2
    end
endrule
```
Somewhere else in the code, `getDataFromOne` is calculated and the first value from either `fifo1` or `fifo2` shall be stored in the register `reg`. However in many cases the rule is not executed at all. The reason is that the implicit condition states that to dequeue data from a FIFO, it must not be empty. Hence the implicit conditions for this rule is that `fifo1` and at the same time also `fifo2` must have some data stored. If only one FIFO contains data, these conditions are not met and rule does not fire.

### 1.4 Organization

This thesis describes the development and architecture of the routers called RECONNECT. As it is discussed in the next chapter (chapter 2) containing a literature survey in which break-through and recent developments regarding NoCs are introduced, the major difference between current publications and RECONNECT is the unique feature that RECONNECT can be easily adapted and configured to be integrated in various Fabrics. The motivation for this work originates in the presented implementations which until now are either too rigid because they serve only a very particular purpose, or closed source and commercialized. However some of the techniques mentioned in the papers have been implemented in RECONNECT. This chapter gives an overview of these ideas and which can be used and which are despite the fact that they are interesting, discarded.

The routers can be easily adapted to provide different configurations or even functions by often just defining a constant. In chapter 3 these configuration parameters and their effects are explained. Further the currently implemented functionality of the several router modules such as Input Port and Crossbar, is elaborated, while the point of view of a flit is taken as it traversals through the router. In many cases the section title includes a filename stating in which file the described functionality is encapsulated. The purpose of each file that are used in RECONNECT, is tabulated in appendix B.

The routing algorithms and their tightly connected topologies are explicated in chapter 4. Three popular topologies viz. honeycomb, mesh and hexagonal are introduced and the routing algorithms explained. Especially the honeycomb topology required the development of a new routing algorithm. How this task has been accomplished is demonstrated via a step-by-step description which is also applicable to develop routing algorithms for other topologies not considered in this work such as the Flattened Butterfly, Spidergon and Stargon which are only briefly analyzed.

In chapter 5 the performance of several router configurations is presented by executing applications on REDEFINE, a dataflow oriented, reconfigurable
architecture which is also briefly described in this chapter. In addition the design of RECONNECT has been validated and compared by running artificial traffic generators under different traffic loads (refer to chapter 6). The router has been compiled into Verilog code and findings of pre-synthesis are tabulated in chapter 7 before concluding the thesis in chapter 8.

1.5 Summary

In this chapter the motivation for an NoC has been explained by pointing out the disadvantages of point-to-point communication systems and buses. Clearly an NoC is not always the ultimate choice. Especially in small systems, a bus is cheaper and easier to implement. However in systems that are scaled, a bus soon becomes a bottleneck and an NoC promise to give a higher performance.

Further the reader has been exposed to some of the common terms that are used in literature regarding NoC. This minimal set of terms is required to understand the basics of NoC. If specifications and requirements require to alter the code in which the routers are implemented, novices to BSV will find section 1.3 most interesting. The strength of the rule management found in BSV is explained with the help of code snippets directly taken from the code itself.
Chapter 2

Related Work

Network on Chips (NoCs) is lively investigated by uncountable research groups throughout the world with many dedicated conferences and journals conducted every year. Some of the results have been commercialized into a product such as Arteris [38, 5], the Teraflops Research Chip by Intel [29, 28] or Æthereal by Philips Research [21]. While these implementations are intellectual property and hence detailed information is rare or patented, NoCs have also been developed by academically oriented research groups resulting for instance in Nostrum [42], MANGO [10] or CLICHÉ [37].

Currently research regarding NoCs can be roughly divided into three different categories each handling other aspects:

1. **Topologies**: Arrangements of the nodes and their interconnects among them.

2. **Routing Algorithms**: Algorithms that determine how flits are forwarded.

3. and **Flow Control**: How the router forwards flits internally by for instance applying novel arbitration rules or controlling the buffer management.

In the following sections, details of the break through and recent research work sorted into one of the categories, will be summarized and finally its impact on RECONNECT discussed.

2.1 Topologies

Since the invention of NoCs around two decades ago [16, 24, 50], numerous suggestions for topologies were introduced ranging from fully connected topologies to arrangements of nodes in three and more dimensions [9]. A lot of attention has been received by the mesh topology and variants thereof due to its easy and natural understanding of arranging nodes in a grid which
eases layout efforts at the same time. In recent years the topology is often
tailed to specific designs in which large Processing Elements (PEs) are
interconnected. In [25] the authors are describing an NoC in which each
node is not connected to its direct neighbors, but to the neighbors next
to them. Hollis and Jackson observed that in many cases the source and
destination are two hops apart in their application specific traffic patterns
and establishing this kind of connectivity results in a latency decrease.

Assuming a predictable traffic pattern the authors in [55, 44] describe
a method to estimate the traffic demands between a communication pair.
In a second step the throughput capacity of the links is adapted to these
requirements resulting in an irregular NoC in which the bandwidth between
two routers differ. In both publications the NoC is customized for a specific
set of traffic traces and synthesized.

If, however, the application is not fixed or not yet known, it could be
beneficial to be able to change the topology according to the requirements.
Different traffic patterns can be accommodated by these topologies which are
also called Polymorphic On-Chip Networks [36]. While this approach seems
to be interesting, problems such as scalability and extremely long wiring
occur apart from a significant overhead in area consumption.

Similar to Polymorphic On-Chip Networks the routers of RECONNECT
have the ability to be integrated into one topology such as mesh and to
mimic other topologies. However in RECONNECT the Input Port (IP)/Output
Port (OP) pairs of the routers are deactivated for power conservation limiting
the number of supported topologies. Thus for example a mesh can be
transformed into a ring, but not into a fully connected network in which
the distance from any source to any destination is one hop only, because
of the unavailability of required connections. By limiting the number of
supported topologies, RECONNECT can be simplified considerably compared
to Polymorphic On-Chip Networks.

Primarily the purpose of this ability of RECONNECT is to prevent the
topology being dictated to the system designer. The difference between
Polymorphic On-Chip Networks and RECONNECT is that the designer is not
confronted with a fixed fully connected network which then can be configured
to mimic other topologies. Instead RECONNECT offers the freedom of
choice by letting the designer picking the desired topologies selectively, while
maintaining its scalability and generality. If the topologies are to be fixed to
e.g. a ring and mesh, a fully connected network whose unused links only
contribute to a tremendous area and power consumption, is not required.

2.2 Routing Algorithms

Tightly bound to the topology are the routing algorithms which can be further
divided into two categories: deterministic and adaptive algorithms.
2.2 Routing Algorithms

2.2.1 Deterministic Routing Algorithms

Deterministic or oblivious algorithms always choose the same route between any source to any destination. Although they are comparatively easy to make deadlock free, they do not consider the current state of the router. For instance a particular link can be used heavily, whereas the other links are underutilized.

A example of a deterministic routing algorithm is a look up routing table in each source for any destination. The information into which direction the router needs to forward the flit, is stored in the its header. These look up tables do not scale efficiently nullifying the very purpose of using an NoC and are usually not used. Deterministic routing algorithms can be easily made deadlock free, since all situations are predictable [17].

2.2.2 Adaptive Routing Algorithms

Compared to deterministic routing algorithms, adaptive algorithms take several factors into consideration such as link utilization but also thermal information [52], while calculating the further path. In addition they can avoid a particular route, if faults occur and a link or node becomes dysfunctional [18].

Adaptive routing algorithms does not need to send flits onto a path that lead them one hop closer to their destination. In case of a heavily utilized link, flits can get deflected and might reach the destination after additional hops, but in a shorter time by avoiding regions of heavy traffic congestions [46]. The algorithms distribute the load in the NoC automatically so that no hot spots occur. In [8] the authors describe an adaptive algorithm called “Hot Potato Routing”. Like a hot potato the aim of this algorithm is to pass the flit on, preferably closer to its destination. In case of a congestion, if multiple flits desire to use the same shared link, the flits has to be forwarded nevertheless. Except from being sent back into the direction where the flit just came from, it is deflected into an arbitrary direction. Thus storing of the flit into a buffer and keeping it waiting till resources become available, is not intended by this routing algorithm.

However adaptive routing algorithms are difficult to be proved being deadlock free due to the unpredictability. A flit that is continuously deflected does not reach its destination although it is not stuck and still traverses through the network. If this situation occurs, the flit experiences in a livelock.

Adaptive routing algorithms can be combined with look ahead routing and speculative allocation [35]. In look ahead routing the path a flit needs to take, is calculated in the previous hop. Depending on the factors the algorithm relies on, this assumes that the previous hop is aware of the situation in the current router such as the link utilization for instance. In speculative allocation a flit is tried to be forwarded regardless, if there is a
competitor for the same resource. The router assumes that no congestion within it occurs and that all flits requesting resources, can get the required resource granted. This skips arbitration steps and hence shortens the critical path. In case of a congestion, no flit can proceed and the arbitration step is executed.

Both adaptive and deterministic routing algorithms have advantages. Especially in heavily congested regions an adaptive routing algorithm will perform better. In periods in which the NoC is not highly utilized, the deterministic routing algorithm does not have any impact on the performance when compared with an adaptive counterpart, but requires less complexity such as a history about link utilization. In [26] a router is introduced providing both, an adaptive and a deterministic routing algorithm. Depending on the congestion conditions in the network one of the algorithm is chosen to forward the flits.

Although adaptive routing algorithms sounds promising, in RECONNECT only a deterministic approach is implemented. Apart from a higher complexity, adaptivity comes with the disadvantage that packets can arrive in an out-of-order sequence and the necessity to put them back into the right order, before they can be passed on to the sink at the destination. Especially in applications in which data is continuously streamed into the target architecture such as in decoders for example, the order becomes important and a tremendous task, because a method to distinguish packets is required. Finding a compromise between the speed up promised by the adaptive routing algorithms, and the hardware complexity and its inherited power and area consumption, depends on the applications that are executed on the architecture which the NoC is integrated into. It might as well happen that the required serialization latency nullifies all advantages of the adaptivity.

2.3 Flow Control

Flow Control defines, how the flits are passed on to the next hop. For instance in [33] a crossbar has been implemented that is able to boost the bandwidth temporarily between an Input Port (IP) and Output Port (OP) by integrating a bus connecting all IP and OP together. A similar approach has been introduced in [41] in which the authors observed a performance increase at the cost of a higher power dissipation and area consumption by increasing the radix of the router. It can provide multiple links to the attached sink resulting that packets can be ejected faster avoiding competition for resources.

On the other hand, Virtual Channels (VCs) [14] are an important mechanism to prevent deadlocks of flits and achieve a higher throughput. The routing algorithms presented in chapter 4.2 depend on VCs to break up cyclic dependencies. In section 4.3 the theoretical background is explained in detail.
In case of packets which are divided into an unequal number of flits due to differences of their payload sizes, a Virtual Channel Regulator called ViChaR [45], can adapt the required channel input buffer sizes according to the sizes of the incoming packets. In wormhole switching however there is no need to store a packet completely before it can be forwarded to the next hop [15]. The header flit is followed by multiple flits carrying additional payload, and the whole packet traverses the network like a train. A packet segmented into multiple flits can therefore be spread over multiple routers. Compared to wormhole switching in which only a fraction of the packet is stored in the traversed routers, a packet needs to be able to be stored as a whole in virtual cut-through techniques [32]. Although also forwarded immediately after arrival, if the availability of resources permits, flits belonging to a packet, are accumulated into a buffer, if a congestion occurs somewhere on the path. The advantage of virtual cut-through is that in case of a congestion, the packet does not consume valuable resources of several routers such as in wormhole switching. However the power and area consumption is comparable to the store-and-forward switching in which the packet is always stored in the buffer completely before it is considered to be forwarded, hence resulting in large buffers. A comparison between wormhole switching and virtual cut-through is summarized in [54].

The problem of identifying flits which belong together and form one packet, is solved by adding a locally unique identity tag to each flit in [51]. Each flit can be considered separately and intermixed whenever a higher link utilization can be achieved. Consider the example in which one IP sends multiple flits to a specific OP but after some time, the IP runs out of data before the tailing flit arrives due to some congestion in the previous routers. In the meantime this OP which is currently not utilized, could be assigned to another IP till the congestion is resolved and first IP is able to send some data again. Flits from both IPs can be stored into the same VC which makes it necessary to distinguish them by a unique ID. However during the execution of real life applications on the test architecture introduced in [5], only very brief congestions of a short duration were observed. Thus the additional hardware for flit distinction cannot be justified currently.

A hot topic in NoC are optical links which promise a higher bandwidth and longer wires without paying delay penalty. Research is concentrated on optical networks (ONoC) such as in [58] that explains routing of packets without the need to convert them into electrical signals first. However in real life ONoC has not been observed in any implementation yet due to the complexity of optical links within the chip, and a high entrance barrier, because of its novelty in chip design.
2.4 Impact on RECONNECT

From most of the mentioned publications it is evident that the underlaying communication and its requirements was known prior an NoC implementation was considered. Extensive traffic trace analyses have been performed and the best compromise between power, area and performance calculated before an NoC was developed exactly fitting the specifications. Although some research work has been done, giving the flexibility of e.g. changing the topology even during runtime, it comes at huge power and area costs and are not considered for productive architectures.

However one goal of RECONNECT is not to restrict its use unreasonably resulting in a flexible router architecture that can be either be tailored towards specific demands or kept flexible to run under several conditions. The aim is to give the system designer a ready made solution that can be adopted for a variety of domains by plugging in or stripping of various functions which are either customized or readily taken out of a pool. As mentioned in the next chapter many ideas which were introduced in the previous sections, are currently implemented in RECONNECT such as support for wormhole as well as virtual cut-through switching, VCs or routing algorithms which were adapted to support various and multiple topologies (even post-synthesis). It is the responsibility and the freedom of the system designer to use the functionality that is appropriate.

2.5 Summary

In this chapter some of the research work published by other research groups, has been introduced. Usually a bottom-up approach has been taken by extensively analyzing the traffic patterns and developing an NoC meeting the specifications. In this work another approach is taken (top-down): A ready made NoC called RECONNECT is provided and stripped off functionalities to also meet the specifications. Since RECONNECT is generic, many published methods are implemented into RECONNECT. However due to the flexible nature of RECONNECT the system designer is able to extend the functionality of the routers with techniques which are not mentioned in the sections above, or even with yet unpublished work.
Chapter 3

Architecture

This chapter describes the architecture of a RECONNECT router and its configurations. Multiple configuration parameters provide an easy way to alter the behavior and functionality of RECONNECT by either changing values of constants (configuration flexibility) or by replacing modules (module flexibility). The effects of some of the parameters are not only restricted to RECONNECT or the Fabric, but also depend on other modules of the architecture in which the Fabric is integrated into. Since this architecture is not known at the time of writing, it is henceforth referred to as target architecture.

3.1 Architectural Overview of the Router

The task a router has to fulfill, can be easily described: Take the data from the incoming Input Port (IP), check which direction it needs to be forwarded to, and pass it on to the next router. A more detailed description is depicted in figure 3.1.

![Figure 3.1: The processing steps a flit encounters while traversing a router.](image)

1. The incoming data is stored in one of the Virtual Channel (VC). The VC has been calculated by the routing algorithm in the previous router. It is important to note that the VC cannot be chosen arbitrarily as described in detail in chapter 4. Before the flit is stored, it is checked, if the new flit is a header flit. In this case, the next Output Port (OP) and VC
number is calculated by the routing algorithm and stored along with the flit.

2. In the next step the neighboring routers report their states of the VC. This step ensures that the IP only considers flits which have a chance to be forwarded. For instance, if IP 2 of the next router reports that all its VCs are full, all flits which want to be routed towards IP 2 do not need to be considered. This step creates a bit array of VCs that contain data and have a chance to be forwarded.

3. An arbiter residing inside the IP chooses one of the requesting VCs and the IP reports the desired OP to the router.

4. Multiple IP might request for the same route, hence another arbitration step (IP/OP arbitration step) is required to resolve this conflict.

5. The IP that won the IP/OP arbitration, transmits its flit and deletes it from the chosen VC. In case of multifit environment the OP is bound to the IP for the entire duration of the transmission of the packet to prevent interleaving of flits from other IPs to the same OP.

6. The flit traverses through the crossbar and is received at the OP in which its relative address is updated in case it is a header flit. In the relative addressing scheme, the address tuple represents the distance from the current node to the destination. Since the distance changes when the flit traverses the Network on Chip (NoC), it needs to be updated at every node the flit passes. If all elements of the address tuple equal 0, the flit reached its destination and is ejected from the network.

   After the address has been updated, the flit is passed on towards the next router.

In the following section the parameters for the configuration flexibility are described. From section 3.3 onwards a point of view of the flit is taken and the modules which a router consists of, are described in detail as it traverses through the router.

### 3.2 Configuration Parameters

This section describes the constants (‘defines mentioned in ./trunk/includes/define.bsv) which are evaluated by the Bluespec System Verilog (BSV) precompiler. With corresponding ‘ifdef and ‘else branches the precompiler removes paths which are never executed. However in some cases such as the evaluation of the number of flits a packet comprises of, the value of a defined constant cannot be calculated during the time of precompilation.
and the if-else branches are not removed. In the next compilation step the following Bluespec compiler evaluates static variables and removes the part of the branch that is never true.

### 3.2.1 DEBUG_NOC[1-3]

If one of the DEBUG_NOC is defined, the routers will print out debugging messages during simulation. This is achieved by the $display command. Obviously it is only relevant during simulations and does not have an equivalent component in hardware. The higher the number of this define, the more messages are printed. During simulations printing messages has a serious impact on the performance. Hence the system designer can speed up the execution time by disabling and commenting out messages which are of no interest.

### 3.2.2 ASMUNIT_DEBUG

Similar to DEBUG_NOC[1-3] this define is used to print out debugging messages for the Assembly Unit (AU) (refer to section 3.3.1). This along with the provided test bench is particular useful, to check, if the conversion from the union representing different packet types, into the bit structure of flits and back, has been implemented correctly. One of the major cause of bugs and errors is the miscalculation of the bit field sizes of the structures within the union. If this define is not set, the AU is silent.

### 3.2.3 MULTIFLITSUPPORT

If the router are integrated in an environment that splits the packet into multiple flits, this define needs to be set. By doing so, the AU is automatically added to the injection and ejection ports of the router ensuring that the packets are divided into appropriate sizes of NWFLITSIZE bits (see section 3.2.7). Depending on this setting the buffers and the logic controlling the flow of packets comprising of multiple flits, are adjusted.

Currently the router supports two flow control mechanisms:

1. The wormhole flow control mechanism means that a header flit will determine the path and can be immediately forwarded even without waiting for the tail flit [15]. The flits of the corresponding packet traverse the routers similar to a train. The advantage of wormhole routing is that the buffers do not need to be able to hold the complete packet once a congestion occurs. A packet comprising of several flit can therefore be spread over multiple routers.

   Since the destination address is carried only in the header flit and hence only with that flit the required OP can be calculated by the routing
algorithm, the information were the following flits need to be sent to, must be retained. Thus the VC that just got selected to serve the head flit, must be bound to the output of the IP module. Further the IP needs to be fixed to the OP so that all successive flits can follow their head flit. However this binding cannot be broken, if the channel runs out of data due to e.g. a congestion in previous hops. The route from the VC to the OP within the router cannot be restored, if data is available again. If the channel does not contain data, the resources are blocked and other flits are not able to utilize them, till a tail flit terminates the bound.

2. Virtual cut-through solves this problem be reassigning unused resources resulting in a higher utilization of the available bandwidth. However this might lead to packets whose flits got interleaved. The destination is currently not able to distinguish flits belonging to different packets. Secondly the buffer sizes required for virtual cut-through will be in an order of magnitude larger compared to wormhole routing.

If \texttt{MULTIFLITSUPPORT} is set, the format of the flit is converted into a union that provides header, headtail, payload and tail flits. Headtail flits are used, if the header provides some bits for payload and the whole payload of the packet fits into it. If \texttt{MULTIFLITSUPPORT} is not set, all packets become headtail flits and hence the union becomes superfluous.

In theory and for later requirements this mechanism of binding IP/OP pairs could be exploited to provide virtual circuits between a source and destination, since the channel is never terminated as long as no tail flit is sent.

### 3.2.4 \texttt{STAGES\_OF\_ROUTER}

The given value defines the number of stages in the butterfly crossbar (see section \ref{sect:3.6}). The number of stages determine how many input and output ports the crossbar has to provide. In the current implementation that supports topologies up to hexagonal structures, the number of required ports is 6 plus 1 for the attached sink/generator. Hence the number of stages is $\lceil \log_2 7 \rceil = 3$ and defines how many bits are required to determine the route of a flit through the crossbar as described in section \ref{sect:3.6.1}.

### 3.2.5 \texttt{IP\_VC, FIFO\_DEPTH\_VC\_IP}

Here the total of required VCs (refer to section \ref{sect:4.3}) for each IP is defined. This number is closely related to the routing algorithm (refer to section \ref{sect:4}) which needs VCs to provide a deadlock free traversal of flits through the network. In case of a routing algorithm that does not depend on VCs to
be deadlock free, this define can be commented out and the provision of VCs completely deactivated. Since the number of VCs is transmitted in the flit header, these bits will be saved and can be used for other purposes such as an increased payload field in the header flit or a reduced flit size. The IP that is connected to the sink/generator does not provide any VCs by default, since a flit is always injected in VC0.

Since the Fabric and its routers are modularized, modules can be replaced easily as long as the interface format is compatible. In another, alternate implementation the define SIZE_VIRTUAL_CHANNEL was used for the same purpose. To not to break the compilation of this earlier implementation, the defines are kept in place.

Another value that is related to VCs is the depth or how many flits it can hold of each channel. To ease the analysis of performance impacts for various buffer sizes, its depth can be changed by alternating the value of FIFO_DEPTH_VC_IP.

3.2.6 PORTS, PORTS_MESH, PORTS_HC

This define determines the number of ports that the router have. It allows an efficient programming style by e.g. looping through the ports arranged in an array for initialization purposes. The system developer defines the topology and the degree of the router here. However it has to be kept in mind that with each port added, the complexity of the router implementation increases. In the implementation described in this work, only PORTS is used and the remaining defines are ignored. They become important for alternative router implementations.

3.2.7 NWFLITSIZE

The value given for this define, determines the flit size in bits that is sent through the network and becomes important in multiflit environments (see section 3.2.3). If MULTIFLITSUPPORT is not defined, NWFLITSIZE is ignored, since the flit size will be equal to the packet size. It is of advantage to determine a flit size that allows a few number of bits as payload to be transmitted in the flit header itself to allow very small packets such as the Topology Configuration Packet (refer to section 3.2.11) to be transmitted in one single flit. A method to determining the appropriate flit size is presented in section 5.4.

3.2.8 NUM_ADDRESSES_ROUTER

In the environment outside the Fabric the coordinates for each node within it comprises of two tuples, since the nodes are arranged in a two dimensional grid. However other topologies with different addressing schemes may
have other requirements. This value reserves space in the header flit for the number of dimensions as depicted in figure 3.3. A function to convert the 2D address coming from the target architecture into the appropriate format is necessary and can be included easily into the AU (refer to 3.3.1). A conversion back into a 2D address is not required, since a flit is ejected from the network only, if its relative address is \((0,0)\). The sink does not need to examine this address.

### 3.2.9 OPSLENGTH

The type of a packet is stored within a union. Bluespec automatically determines the number of bits required to distinguish each packet type. If the packet is stored in registers, the size will be

\[
\text{size} = \max \text{size}(\text{packet\_type}) + \lceil \log_2(\text{number\_of\_packet\_types}) \rceil.
\]

Currently there is no method to extract this number of additional bits and make it available to the system designer. So when it becomes necessary to convert the packet from the bit field back into a union, the number of bits that are needed to distinguish the packet type must be known. This value is defined in \text{OPSLENGTH} and needs to be recalculated, when the number of packet types in the union changes.

### 3.2.10 HEADERSIZE, BITS_SPACE_IN_HEADFLIT

Whereas \text{HEADERSIZE} accumulates the size of the address tuples and eventually also the number of bits needed to store the VC number (refer to 3.2.5), \text{BITS_SPACE_IN_HEADFLIT} returns the number of bits which are left to store payload in the header flit.

### 3.2.11 MULTITOPOLOGY

The router supports multiple topologies and can change them according to the desired configuration. After a reset all routers examine incoming packets for the Topology Configuration Packet. Once they are configured or the first packet that arrives, is not a Topology Configuration Packet, they lock themselves and hence cannot change the topology configuration again till the next reset. The reason is that the unused ports are clock gated and hence switched off. In case of a fully dynamic topology change during runtime, a switched off \([IP,OP]\) pair will not accept any data nor forward data that is stored in the buffers eventually. Waiting applications cannot continue with their execution and stay on the Fabric occupying computation resources.

The current implementation does not expect to run multiple different topologies on the Fabric at the same time. This restriction can be relaxed,
if it is guaranteed that traffic does not cross from one topology to another one. The Fabric is logically divided into subsets and the traffic caused by an application is restricted to a particular area of the Fabric. It needs to be kept in mind, that the supporting logic which is responsible to launch applications onto the Fabric has to be aware of the topology to maximize its utilization. Otherwise the impact of a richer topology might be negligible as observed in the test case REDEFINE, in which the Resource Binder (RB) is not topology aware to not to increase its complexity (refer to section 5.2).

In addition the NoC does not have the notion of a broadcast. Hence after the arrival of a Topology Configuration Packet, the routers change their topology, but at the same time also route the packet according to the routing algorithm of the honeycomb topology, which is the default one. If the Fabric has \( n \) columns, only \( n \) packets with the destination address \((x, y) = (\text{number of cols}, 0)\) that are injected at the Access Routers (ARs), are required to configure all routers. So at maximum currently only a row and not a section wise configuration is allowed.

If MULTITOPOLOGY is not set, the necessary logic to change the topology is not synthesized and the Topology Configuration Packet does not exist. The router falls back to the default, currently honeycomb, topology. The main purpose for supporting multiple topologies is to have the ability to check uncomplicated the impact of a topology on the execution time of an application without the necessity of recompilation of the whole architecture. Although it is expected that in the final design the routers serve only in a single topology, the functionality has been preserved to allow system designers to test future topologies and applications.

### 3.2.12 XBAR

There are different implementations from several authors of NoC routers available. If the routers that are described in this work shall be used, the define needs to be set. It ensures that the Python script multiTopo.py is executed, generating the Bluespec code which in turn establishes all links among the routers and Processing Elements (PEs) (.bluespec/Fabrique/Fabric/Fabric.bsv). A look into multiTopo.py reveals that the Fabric can be further configured:

- Generate a Fabric that allows to connect artificial traffic generators. The Fabric will become a self sustaining stand alone module allowing testing of new modules such as e.g. routing algorithms. Artificial traffic generators written in C code and compiled into the Bluespec code allow a minute configuration of test cases. These traffic generators read the traffic patterns from configuration files hence avoiding a time consuming recompilation of the whole Fabric for other test cases. Due to the C code generators, this Fabric cannot be synthesized into Verilog
and runs only as a Bluesim simulator (i.e. an executable for an i686 or x64 PC architecture).

- Usage of routers which comprises multiple pipelined stages as described in 3.6
- Single cycle routers running at the same clock speed as the remaining modules of the target architecture.

### 3.2.13 EJECTPORT and (.*)_FABRIC?

These defines provide a human readable representation of the direction numbering to ease the implementation and avoid programming mistakes. There are two different sets of defines:

1. Defines that represent the direction within the router.
2. and defines ending with _FABRIC which are only used in the Fabric and which are decremented by one compared to the corresponding direction definition for the router.

Within the router the routing algorithm determines, where flits have to be forwarded to. To accomplish this the algorithm returns a number which represents the heading of the OP. All OPs that are connected to neighbors are stored in an array. Logically the returned number by the routing algorithm represents the index of this array. However the OP to which the sink/generator is connected, differs and is not included in that array. This OP is exported by the router by a separate interface. Hence the index for a particular OP outside the router does not match with the array index for the same OP used inside the router and its routing algorithm. To connect the routers to form a specific topology, defines ending with _FABRIC, but to implement a new routing algorithm, defines not ending with _FABRIC, need to be used.

### 3.2.14 FLITNW_FIFOSIZE_ASMUNIT

It determines the depth of the first-in-first-out (FIFO) buffer that stores the flits of the segmented packet in the AU. The larger the buffer, the better a congestion in the NoC can be compensated by allowing the attached generator to continue sending packets. A smaller FIFO will result in lower area and power numbers of the AU.

### 3.2.15 CE_CLK and SUPPORTLOGIC_CLK

Since the routers are comparatively small logical units, they can probably run at higher clock frequencies than the remaining modules of the target
architecture. Higher frequencies will cover up the latency that is experienced after submission of a packet into the NoC till it reaches its destination.

Both defines represent clock dividers meaning that the given value is used as a divisor to derive a new clock from the original one. This setting highly depends on the target architecture. In case of REDEFINE CE_CLK is the clock for the Compute Elements (CE) whereas SUPPORTLOGIC_CLK is the one for the Support Logic (SL) respectively. The original clock which is at the highest frequency, is given to the NoC. However synchronizers are not yet implemented at the boundary of the modules so that any setting of these two define values does not have any effect.

3.3 Input Port (IP) (InputPort.bsv)

![Figure 3.2: An architectural overview of the several modules of a router.](image)

As it can be observed in figure 3.2 depicting an architectural overview of the several modules of a router, the first module a flit encounters, is an IP. There are two different kind of IPs that provide connectivity to the router:

1. One is designed to be connected to the sink/generator which can be a PE or other modules of the target architecture. It includes the segmentation and reassembly of packets into flits and vice versa in multifit environments. This functionality has been implemented in a separate module called AU (refer to section 3.3.1).
2. The other kind of IP is directly connected to the OPs of the neighboring routers (see section 3.3.2). It provides the functionality of VCs including the arbitration by Matrix Arbiters. If VCs are not required, each IP is reduced to provide a simple buffer of a predefined depth and a routing algorithm. Since there are no VCs anymore, obviously a VC arbitration step also becomes superfluous.

3.3.1 Assembly Unit (AU) (AssemblyUnit.bsv)

The AU is always instantiated regardless of the configuration depending on the defines (refer to section 3.2). However the provided functionality differs, if the router is not used in multiflit environments. In these, the AU basically only provides an interface that is compatible to the target architecture. As mentioned in section 1.3 the NoC works merely on the provision that a rule does not fire, if any of the implicit and explicit conditions that leads to the firing of the rule, is false. For example if a rule could fire and it contains a component that writes into a FIFO, but at the same time the FIFO is full, the rule will not fire. The readiness of the FIFO buffer becomes an implicit condition for firing this particular rule. This mechanism is used to e.g. transfer data from the OP to the IP. The rule representing the OP will not transmit data, if the IP cannot accept it. Thus sending Acknowledgements (ACKs) back and forth can be omitted.

As described earlier the IP has an additional mechanism to choose only those flits in the VCs that can be routed (i.e. space is available in the receiving IP of the next router). Hence the implicit condition in which a rule transmits a flit to a full IP, does not occur in the first place.

However it might be the case that the surrounding support logic or the attached PE do not follow this protocol. For instance in REDEFINE (refer to chapter 5) the SL and CE write to wires first, which are always writable and cannot block rules. Hence the SL expects an ACK if the storage of data was successful in the intended module. The AU translates the protocol supported by the SL and CE into a compatible protocol for the router and vice versa.

If the router is embedded in a multiflit environment the task of AU is extended into segmenting a packet into flits. Depending on the available bus width and the size of the packet, multiple flits are generated. As long as the flits are generated, the AU marks itself as busy and does not send back ACKs, if new data is intended to be stored in the FIFO at its input port.

As it can be observed in figure 3.3 the packet structure is not necessarily fortunately chosen to be split into multiple flits: The header is aligned to the MSB of the packet whereas the payload is aligned to the LSB leaving bits unused right in the middle. Hence it is not a simple shift operation till the whole payload of the packet is converted into flits. The problem that forces this format, was the Bluespec compiler. Bluespec is type sensitive meaning that a variable declared as int (a signed integer of 32 bits width) cannot be
3.3 Input Port (IP) (InputPort.bsv)

used as an Int#(4) (a signed integer of 4 bits width). The type int needs to be casted into Int#(4) by converting it into a bit field of 32 bits (unpack function), truncating it (truncate) and casting it back into the Int#(4) type (pack function). This is merely a change of data representation not effecting any hardware eventually generated. Bluespec also allows to overload the pack/unpack functions by self defined functions to e.g. align the flit header and the payload to the MSB. However it has been observed that in that case additional hardware is indeed generated. To avoid this unnecessary hardware, the default unpack/pack functions are used leaving unused bits inconveniently in the middle.

The AU consists of a state machine to process incoming packets and the procedure is depicted in figure 3.4. After a packet has been accepted by the AU depending on the type of packet first from a look-up table it is chosen, how many flits should be generated. This value is stored in counter. The look-up table is static and has been precalculated during compilation of the target architecture such as REDEFINE (refer to section 5) in FlitStructureSizes.bsv. If the packet format changes and architecture is recompiled, the number of required flits is adapted automatically.

In the second stage the head flit or headTail flit, if the payload of the packet fits into the payload field in the header flit completely, is generated. If ‘NUM_ADDRESSES_ROUTER (refer to section 3.2.8) is greater than the dimension of the address array that is used by the target architecture, the additional address fields in the flit header are set to 0. In a second step the template bits, which store the type of packet of the union, are stored in the payload field of the head flit. If there is any space left, the payload located at the LSB of the packet, is stored in the payload field and the packet is right shifted accordingly. In addition the variable counter is decremented. That completes the stage of the header flit. The new header flit is stored at the output port of the AU and can be picked up by the router as soon as resources are available.

Depending on the value of counter, multiple payload flits can be generated. Payload flits do not carry any header information and the whole space can be used to carry payload. After each generation, the payload of the
(a) The head flit is generated. The first 2 bits are the template bits to distinguish between the flit types (head, headTail, payload and tail). The header is filled with the given information of the packet. If space is left such as in this example are 8 bits left to carry payload, the LSB of the packet are taken and the payload field of the packet is right shifted accordingly.

(b) Payload flit generation. This step continues till there is only sufficient payload data left to generate one more packet.

(c) Generation of the tail flit. Since the number of bits is not sufficient to fill the whole payload field in the flit, the bits of the unused field are taken.

Figure 3.4: In this example multiple flits are generated from an Instruction Packet type. The flit size is set to 18 bits and the router are operating in an environment in which ‘NUM_ADDRESSES_ROUTER’ are set to 2 and the routing algorithm does not depend on VCs.

packet is right shifted by the value of ‘NWFLITSIZE’ (refer to section 3.2.7) and counter is decremented by one each cycle.

If counter is equal to 1, a tail flit is generated. As the payload flit, it does not carry any additional information apart from the payload. As soon as the tail or headTail flit is generated, the AU is able to accept the next packet from the attached generator.

The procedure for incoming flits that must be reassembled to a packet is similar. The NoC guarantees in-order delivery, hence it is not required to keep a record about the incoming sequence. The payload of the head flit is partly stored at the MSB of the payload field to determine the packet type. The remaining bits are stored at the LSB of the packet. All following payload flits including the tail flit are stored in a last-in-first-out (LIFO) buffer. After the
tail flit has arrived, the buffer is read and the data is inserted into the payload field of the packet. Since some data has already been stored by the head flit, the location of insertion is not at the LSB, but at ‘BITS_SPACE_IN_HEADFLIT − ‘OPSLENGTH’, which is $8 - 4 = 4$ in the example in figure 3.4.

### 3.3.2 IP connected to OP

If ‘IP_VC’ (refer to section 3.2.5) is set, the IPs that are connected to the OPs of the neighboring routers provide VCs which are instantiated in the module `mkInputPort[XBar]`.

For selecting among the filled VCs a Matrix Arbiter [48] has been implemented. It provides a similar scheduling of VCs such as round-robin. However there are a few limitations:

1. In single cycle routers it is obvious to implement an arbiter that only considers flits that can be routed, since the buffers in the next IP are not full and can still accept data. Flits that cannot be routed, are not considered, hence increasing the throughput. In a second step multiple IPs might compete for an OP. If the flit looses this arbitration, it remains in the IP. Hence considering the state of the next IPs does not guarantee any proceeding to the next hop.

2. In multiflt environments after the arbiter chose a head flit, the following flits are taken from the same VC and the arbiter is deactivated till a tail flit passed. Even if the chosen VC runs out of data, the arbiter does not choose another VC. Assume there are two IPs which have data (flits belonging to packet $a$ and $b$) that needs to be routed to OP 0 and the same VC number. The arbiter of each IP chooses the head flit and it starts moving through the crossbar. However the VC of flit $b$ runs out of data. To achieve a higher utilization of the available hardware, it would be desirable that the flit $a$ can proceed till the empty VC is filled with data again. However the receiving IP has no means to distinguish both flits. From its point of view a head flit and a payload flit from flit $a$ arrives followed by the head flit of $b$ for the same VC. The following payload flit could belong to both packets.

By binding the chosen VC to the output port of the IP, the interleaving of packets is avoided at the cost of a potentially lower hardware utilization which is compensated by the lack of hardware for distinguishing and reordering of incoming flits.

### 3.3.2.1 Matrix Arbitrators (Arbiter([0-9]+).bsv)

One efficient way for an arbiter implementation is the Matrix Arbiter [48], which calculates the winner among all requests by a priority queue stored in...
a matrix. Immediately after selecting the request with the highest priority, its request line is set to the lowest urgency to allow others to win. The logic and implementation can be done very efficiently.

A basic matrix arbiter is partly shown in figure 3.5. The figure displays only the logic which handles the granting of request0. The logic for all other grants is similar. The rows and columns of the matrix for the connection of the inputs to the OR gates are calculated as followed: Since in this logic request0 is handled, only the registers located in column number 0 are considered. For the input to the OR gate the row number corresponds to the number of the request. In this example the inverted value in register (1,0) is directed to the OR gate for request 1 and (2,0) is used for request 2 (see \(m'_0\), \(m'_2\), and dotted wire).

To save power and area it can be observed that the values stored in the registers above the diagonal of the matrix, are the inverted values of the values below that diagonal. Hence \(m_0\) and \(m_2\) can be used instead which are the same compared to the complements of \(m'_0\) and \(m'_2\).

In general \(\left\lceil \frac{\rho(\rho+1)}{2} \right\rceil\) registers are needed for \(\rho\) requests (with \(\rho \geq 0\)). Let \(req_i\) the \(i\)th request and \(grant_i\) its grant respectively. \(m_{ij}\) represents the \(i\)th row and the \(j\)th column in the matrix. The following equation calculates \(grant_i\) for a request:

\[
grant_i = \left\lceil \frac{\rho(\rho+1)}{2} \right\rceil
\]
3.3 Input Port (IP) (InputPort.bsv)

\[
grant_i = \text{req}_i \prod_{j=0}^{j<i} (\text{req}_{ij} + m_{ij}) \prod_{j=i+1}^{j<\rho} (\text{req}_{j} + m_{ij}) \tag{3.1}
\]

The initialized status of the matrix is also shown in this figure. After a successful grant the status of the registers has to be changed to allow other requests to win this arbitration. In this example after the winning of request0, in column 0 all registers are set (so nothing changes, since there are none in column 0) and row 0 is reset.

An advantage is that due to the combinational logic, the grant can be calculated within a cycle. More advanced arbiters which also supports e.g. prioritized requests, get more complex and will not be able to generate the granting signal within a single cycle. For that reason a kind of speculative arbiter is introduced in [43], which tries to predict the next winning request based upon the previous grants.

There are two arbitration steps in the router in total: VC and IP/OP arbitration both operating with Matrix Arbiters. As mentioned in [40], at least a weak, but fair arbitration is desirable meaning that each flit will be served eventually. In other words, in both arbitration steps it is guaranteed that no flit starves in any point in time. However a stronger arbitration would be desirable guaranteeing not only the service to a request, but also considering other factors such as the age of a request. This would assume some kind of communication between the two arbitration steps increasing their complexity. One method would be to overrule or deactivate the VC arbiter in case a flit could not make it through the IP/OP arbitration step. The same flit tries it once again in the next cycle till it could traverse the router successfully. From a local point of view of the router this seems to be much fairer compared to choose the next flit no matter, if the previous one was successfully routed. However from a global point of view of the whole network, this could lead to a deadlock. Consider the situation depicted in figure [3.6]

In the current implementation the arbitration is weakly fair which in some cases might result that a flit resides in the VC longer than necessary, easing the complexity of the arbiter. On the other hand this causes a degradation of the throughput saturation point as shown in figure [6.2d] in section 6.2. Since arbiters are in the critical path in single cycle routers one aim is to keep them as simple as possible to not to prolong the path reducing the clock frequency.
(a) Flit A and B need to be routed to the east. VC0 in the next router is reported to be free, and no other VC competes with A nor B. However IP0 and IP1 are both competing for the same OP and IP1 is chosen to proceed.

(b) Flit A is stored in the next router. It still needs to be forwarded into an eastern direction and in one point in time, it will use one of the toroidal links and shifts from VC0 to VC1. Since VC1 is free, flit A progresses and is followed by the first flit from packet B, because the IP/OP arbiter in router A grants the least served request first.

(c) Since the IP1 lost the arbitration in the previous cycle, the VC arbiter has been deactivated. Hence the same VC of IP1 requests resources to be forwarded. However the VC at the next hop is still occupied by B. Although A0 could be routed because of the availability of buffer space, the VC arbiter cannot choose another VC. A deadlock situation occurs.

Figure 3.6: How a deadlock situation can occur, if the arbiter have the notion of strictly serving the oldest VC first. Although the original implementation seems to be fairer, it certainly does look so locally. However in a global point of view, unfortunate situations such as this one, occur. Without additional hardware like fixing the VC for a specific amount of cycles, increases hardware complexity. This needs to be avoided, since the arbitration is in the critical path.
3.4 Crossbar

There are different ways to implement a crossbar. In single cycle routers the crossbar is stateless and comprises of muxes which are ideally arranged in the order shown in figure 3.7 or in [60]. In the Bluespec implementation the crossbar is hidden in the rule transferFlit and it is not visible in the Verilog code.

There are two points of view, how a flit traverses the router: An IP can send the flit towards an OP or an OP can request one specific IP to transmit its data. The latter one is implemented. For each connection from any IP to any OP exists a rule that fires upon a pulse. The pulse is generated by the arbiter that is instantiated once for each OP. The requests are coming from the IPs which wish to send their flits to this particular OP. After the arbitration step the OP is aware what IP won and request it to send its data.

Another implementation of a crossbar is a butterfly crossbar, which is described in section 3.6.1.

3.5 Output Port (OP) (OutputPort.bsv)

Since the Fabric uses a relative addressing scheme, at each hop the address is updated based on the location of the OP. Each one consists of an adder to increment or decrement a signed integer value of ADDRLENGTH bits (currently 4 bits). By definition the vertical axis or the axis from north east to south west as described in section 4.2 represents the y-component, the horizontal axis the x-component and the axis from north west to south east the z-component of the address tuple.

3.6 Pipelined Routers

Currently the router run in synchronization with the target architecture, since multiple clock domains are not yet explored nor implemented. As discussed in the next chapter, the current clock rate of 450MHz suffices. However for
future and in streaming applications in which each CE performs a specific task repeatedly and passes the values on to the next CE the throughput provided by the current router implementation, might not be high enough. Especially if the PEs are replaced by large functional blocks that can be commonly found in Application Specific Integrated Circuits (ASICs), the NoC can become a major bottleneck in the view of the large amount of data that is transmitted among the blocks. For instance in [4] the authors calculated a required bandwidth requirement of 607MBps and in [64] the bandwidth requirements including the overhead has been estimated to be 637MBps both for H.264 decoder implementations.

One method to boost the bandwidth of the NoC is to increase the bus width connecting two routers which does not impact the clock frequency as elaborated in section 7.5. However this method will increase the wiring complexity and the efforts in post-synthesis. Another method is to increase the frequency by pipelining the router which is discussed in this section.

### 3.6.1 Pipelining of the Single Cycle Router Implementation

The maximum effect is achieved by pipelining the critical path and by implementing buffers located exactly in the middle of the path. However since Bluespec is an abstract language describing the hardware, it is not always obvious, where the critical path is. Especially in big designs, in which instantiated modules take functions and other modules as parameters and are hence marked as not synthesizable (refer to section 1.3), the desired functionality is compressed into one large Verilog file which can be several megabytes in size. Finding the path that is reported by the design compiler is just a matter of searching in the Verilog file since the naming convention in the Verilog file will be the same. However matching the found path in functionality with the corresponding Bluespec code which the Verilog code got compiled from, is challenging.

Another method of pipelining though not as efficient as described above, is the logical segmentation of the functionality of a design (shown in figure 3.1 on page 17) into pipeline stages. Currently there are two different implementations of pipelined routers available:

1. A pipeline stage was inserted after the IP/OP arbitration step. Hence VC and IP/OP arbitration has been separated from crossbar traversal, relative address update and insertion of the flit into the VC of the next IP

2. To introduce more pipeline stages, the crossbar with its multiplexers has been replaced by a butterfly crossbar (refer to figure 3.8) comprising of multiple stages. Since the butterfly will resolve any conflict for competing IPs for one OP automatically, the IP/OP arbitration step can
be removed. Each node comprises of a FIFO buffer at its input port, a decision logic determining, if the direction of the flit is flipped or if the flit has to be routed straight, and a FIFO at its output port. In case of a congestion, a node arbiter chooses the flit to continue. In multilfit environments, the arbiter is deactivated, if a head flit passed till the tail flit arrived at the node. This prevents the interleaving of flits belonging to different packets. It uses the same method as the IP is binding a VC to its output port (refer to section 3.3.2).

Figure 3.8: The internal structure of the butterfly crossbar allowing the implementation of pipelined routers.

The dotted lines and boxes symbolize unused elements. The crossbar is generated by the recursive invocation of the BSV module mkXBar. As input parameters is given the internal decision logic as another module (mkMerge2x1), the ID of the router and the number of the current stage. To allow a higher readability the code is not adapted to skip the generation of the unused seventh port. It is assumed that unconnected ports are removed by design tools down the chain.

The routing algorithm fills the flit header with information at which
stage the flit needs to be send straight to the node in the next stage of the crossbar or if the direction needs to be flipped. \( \log_2(n) \) (n being the number of ports) additional bits are necessary. If a bit is set, its flit is flipped, otherwise it goes straight. Depending on the routing algorithm and its location in the corresponding IP, the directions to the OP differ. For instance the flit header of a flit going from the eastern port to southeast needs to be set to 'b111, whereas from the west to the same OP the bit array would be 'b100. Please note that the LSB represents the operation that is executed at the first (left) stage.

### 3.6.2 Changes in the Implementation

Currently the IP receives information about the VC occupation states from the attached IPs of the neighboring routers. Flits which do not have any chance of being forwarded due to the non-readiness of the receiving VC are not even considered for routing. However in multi-pipelined routers the VC occupation state that is currently received, might differ from the state that the receiving VC will be, when the flit leaves the router and needs to be stored after multiple clock cycles. It is already outdated by the time, the state is used for the VC arbiter. Consider the situation depicted in figure 3.9 in which an IP sends out a flit (A) destined for one particular OP and VC in the receiving IP at \( t = 0 \). There are 10 pipeline stages in the router. At time index 5 another IP sends a flit (B) to the same destination as flit A. Since flit A is still in transit and the receiving IP is not aware of it, it still reports the receiving VC as available. Once flit A left the router and is stored in the VC, it becomes full and cannot store any data further leaving flit B “hanging” in one of the pipeline stages blocking valuable resources. Especially in the butterfly crossbar implementation in which many paths are shared by distinct IP/OP pairs, this situation can lead to deadlocks.

![Figure 3.9](figure.png)

**Figure 3.9:** While flit A is in transit, flit B is sent by another IP to the same OP and VC like flit A. Assuming the VC can store one flit only and after flit A is stored, flit B has to wait in the last pipeline stage till flit A can proceed further.
3.6 Pipelined Routers

There are two methods to prevent this situation to occur that are discussed further:

1. Once an IP sends out data to a specific OP this OP is marked as occupied and in use. Hence no other IP can send data through that OP till a tail flit passed through it. While this is easy to implement, since the original IP/OP arbiter is extended by a simple state machine examining the flit types leaving the crossbar, it comes at the disadvantage of introducing bubbles into the pipeline. An IP which wishes to send data to an occupied OP has to wait till this OP is available again. Hence the flit that is currently traversing the crossbar, neither precedes nor is succeeded by another flit resulting in a period in which the IPs are in a waiting state.

Consider the example in figure 3.9. For any flit that enters the pipeline, the IPs which have to send data to the same OP have to wait for 10 cycles. Assuming the router runs at speeds of 10MHz before pipelining and the introduction of pipeline stages was optimal so that the clock speed is at 100MHz now. The flit size and hence the bus width is 100 bits. In the single cycle implementation the bandwidth is therefore \( \frac{10^7 \times 100 \text{bits}}{s} = 1 \text{Gbps} \). Potentially the pipelined version of the router can transmit 10 times the data than the original non-pipelined version. However the effects of a faster clock and the promising higher throughput are nullified by the forced introduction of bubbles in the pipeline.

2. To achieve a higher throughput the bubbles in the pipeline need to be eradicated. To do so, it needs to be guaranteed that the receiving VCs can store a complete packet comprising of multiple flits. In addition the receiving VCs needs to accept any possible packet that might be under way in the pipeline stages, after it reports itself as occupied. A water mark mechanism needs to be implemented. The first condition ensures that no packet is spread among several routers in case of a congestion. The latter condition prevents the occupation of valuable resources in the router when the VC becomes full. The VC depth needs to be

\[
\text{VC Depth} = \left\lceil \frac{L}{B} \right\rceil \times P
\]  

(3.2)

with \( L \) and \( B \) being the largest packet and the flit size respectively and \( P \) represents the number of pipeline stages. Since buffers consume a comparatively large amount of power and area (refer to section 7), pipelining to achieve higher frequencies might not always be an option. Especially in the view of for mobile devices geared towards power savings.
3.7 Summary

In this chapter the internal architecture of the RECONNECT router and its configuration parameters has been explained in detail. In many cases the system designer is able to either choose among several modules out of a pool each providing different functionality (module flexibility), or changing the behavior of the router just by altering a constant (configuration flexibility). A common mistake by network designers is to consider the router to be a physical and logical entity completely separated and encapsulated. An example is that the flit always enters through the IP and is stored here. However another perspective is a shift of the relative address update logic into the IP literally moving the OP of the previous router into the IP of the current one. Hence some caveats in the case that new functionality is implemented, are also discussed such as the global impact onto the NoC, if a module is supposed to be very much locally restricted to the router, is changed (refer to section 3.3.2.1).
Chapter 4

Routing Algorithms
(Routing(.*).bsv) and Topologies

In this chapter the deterministic routing algorithms for three popular topologies viz. honeycomb, mesh and hexagonal topology and their development are described. Each routing algorithm is encapsulated as a function given to each router as a parameter. Due to the encapsulation, the routing algorithm can be replaced without effort. If RECONNECT is integrated into a topology not mentioned here, a new routing algorithm needs to be developed. How this can be accomplished, is described by a step-by-step example for the honeycomb topology.

4.1 Preliminaries

Every Input Port (IP) has access to its own instantiation of the routing algorithm. Normally it returns the next heading a flit has to be forwarded to as a unique identifier for each Output Port (OP). This identifier is the index of the array that contains the OP modules. For the butterfly crossbar (refer to section 3.6), the routing algorithm returns a bit array instead, stating at which stage the flit needs to be forwarded straight or flipped.

The rules of the algorithm are implemented in functions which are given to the module mkNoCRouter as a parameter. By doing so, the implementation remains flexible and the algorithms can be easily exchanged. For instance the router located in the right corner at the bottom of the Network on Chip (NoC) requires a different algorithm with date lines than a router in the middle of the Fabric.

In general during the implementation the following naming scheme for the algorithms has been followed (refer to table 4.1): Each function name
Routing Algorithms (Routing(.*).bsv) and Topologies

starts with getRoute followed by the topology the algorithm is designed for (Honeycomb, Mesh, Hexa). After that a name for the algorithm preceded by an underscore (_) follows to allow multiple different routing rules for the same topology. For instance Mesh is followed by _WestFirst and Hexa by _NegFirst to describe the rules of routing flits first to west in meshes and into the negative direction in the hexagonal topologies respectively. This is followed by the number of elements in the address tuple. For two elements, 2D is used, for three elements 3D. Since there is only one routing algorithm currently implemented for Hexa and it requires always three tupled addresses, this field is omitted here. The next column defines, if the algorithm requires Virtual Channels (VCs), and the last column shows, in which direction the dateline is located.

4.1.1 Example

Assuming the router in the upper right corner is instantiated. The following routing algorithms are used (see figure 4.2):

- getRouteHoneycombDateLineN2DVCE, because in Honeycomb topologies, this router has a Dateline towards the east (E). The algorithm depends on VCs to be deadlock free, it has an IP/OP pair to neighboring routers in the north (N) and uses only \((x,y)\) coordinates (2D) for addressing.

- getRouteMeshDateLine_WestFirst2DVCE for the Mesh topology that provides a Dateline towards the north and east (N_E) and also uses two tupled addresses (2D), while depending on VCs.

- getRouteHexaDateLine_NegFirstVCNE_NW_E. Note that the direction N and NE are the same as shown in figure 4.2.
Table 4.1: Routing function naming convention.
4.2 Topologies

The router is designed to support honeycomb, mesh and hexagonal topologies each with a different routing algorithm. It is able to switch the topology after each reset meaning that the router must have at least a degree of 6 (excluding the port to the generator/sink) for the hexagonal topology. There are multiple options available on how all three topologies can be merged into a single one. The options are discussed in this section.

For the honeycomb topology a rectangular brick structure is used as shown in figure 4.1b. It is obvious to see that the brick structure is very similar to the mesh topology with each alternating vertical link missing. Hence merging honeycomb and mesh topologies can be done easily.

However if a mesh or honeycomb topology are mapped on a hexagonal one, it becomes problematic, since there simply are no vertical links (refer to figure 4.1c). By logically shifting every second row of the hexagonal topology by half a hop distance so that the vertical nodes are aligned seems to solve the problem as done in figure 4.1d. The same hexagonal cells are marked with thick lines. However beside the fact that every router has an additional link to the north west and south west for even rows and to the north east and south east for the odd ones, the y and z axis are no longer straight lines. For instance, if a flit needs to travel two hops to the south, it does so by traveling in z- direction (marked by z+ in the figure) and the continues on the y+ axis. Though it went straight south, the axis changed. Further, if the node is located in an even numbered row, the y+ axis is the one that points into south west direction, whereas z- is heading directly for the southern router.

This leads to an overhead in programming and is prone to errors, since the routing algorithms for the hexagonal and mesh topologies need to compensate for the changes in the axis.

Instead of superimposing a hexagonal topology onto honeycomb and mesh, the other way around would be to merge honeycomb and mesh topologies into the hexagonal one. Arranging the nodes in this pattern simplifies the mapping which is depicted in figure 4.2. The thick lines are the connections that are used for honeycomb.

The gray nodes represent the Access Routers (AR) which provide connectivity to and from the Fabric and which are not attached to a Processing Element (PE). The figure also shows the numbering of the x and y axis. As it can be observed the x-axis is horizontal whereas the y axis is tilted by 30° to the right. Hence if a flit is routed north, it is actually routed north-east, but due to the mapping of the routers in the Fabric both directions are the same. By definition the AR are located at the logical right boundary of the Fabric. However since currently only relative addressing is supported and in addition the opposing boundaries are connected by toroidal links (of which only the ones that are significant for the honeycomb topology are shown), it is irrelevant where the routers are located physically. Hence the lower two
4.2 Topologies

(a) Regular honeycomb topology consisting of three combs

(b) Same topology as in figure 4.1a converted into a brick structure.

(c) Rectangular hexagonal topology

(d) Same topology as in figure 4.1c. However every second row has been shifted by half of a hop distance to the left.

Figure 4.1: Different layouts of the honeycomb and hexagonal topology.
nodes on the right side are logically situated at the left side of the same row converting this Fabric into a parallelogram.

4.2.1 Flattened Butterfly

There are other topologies worth a further investigation: One of them is the Flattened Butterfly by [34] connecting all nodes in a row and column, reducing the distance from any source to any destination drastically to two hops only. The reduction is paid by an increased radix to 10 - a concentrator connects four sinks/generators to the router. Further six links are used to establish connectivity to all routers in the row and column (refer to figure 4.3). In addition to a high radix, long links reaching across the network are required making it necessary to strengthen the signal either by repeaters or by using optical routers [23] [19] which are, though an emerging research field, not yet common in today’s technology.

4.2.2 Spidergon and Stargon Topology

Spidergon has the same degree when compared with the routers integrated into a honeycomb topology namely three. As an advantage it requires only two VCs to be deadlock free [13], whereas the routing algorithm for the honeycomb topology needs four. The logical structure of Spidergon is depicted in figure 4.4a and its physical layout in 4.4b. It also can be scaled to accommodate more nodes: The physical layout can be stretched horizontally leaving the radix constant.

Another similar topology is Stargon [63] which solves the problem of Spidergon that the network diameter does not scale efficiently, when the number of nodes grows. The degree is extended to four as shown in figure
4.2 Topologies

Figure 4.3: A $4 \times 4$ flattened butterfly topology in which all nodes are fully connected row and column wise.

4.4c. Although an implementation especially of the physical layout (figure 4.4d) looks promising, it is highly depending on how applications are run on the target architecture and if the traffic pattern can be altered in a way to match this topology.
(a) Logical layout of the Spidergon NoC with a node degree of 3.

(b) Physical layout the same NoC shown in figure 4.4a

(c) Logical layout of the Stargon NoC with a node degree of 4.

(d) Physical layout the same NoC shown in figure 4.4c

Figure 4.4: The logical and physical layouts of Spidergon and Stargon topologies.
4.3 Virtual Channels (VCs)

It is compulsory to ensure that there cannot be any cyclic dependency in any situation in the network to avoid deadlocks. For instance in honeycomb toroidal networks are 2 different kind of dependencies possible: A cycle that comprises a comb and a cycle that circles once around the network as depicted in figure 4.5. The arrows indicate the direction a flit want to be forwarded to. It is important to note that the data stored in the buffer can belong to entirely different messages and do not have to have a common source and destination. For instance the flit marked with (*) could have come from the generator that is located in the far east of the same row.

To break cyclic dependencies the routing algorithm might depend on additional physical network layers which a flit is forwarded to. Each layer is equipped with a different routing algorithm and the paths from a lower layer to an upper layer are unidirectional. A message that traverses in a higher layer, cannot reenter a lower layer. Though physical separation is a commonly used implementation of complex bus systems such as in [31], it results in a tremendous overhead on a chip. If performance or wiring complexity is a constraint, VCs provide a solution to reduce multiple layers to a single physical layer thus flattening the topology [61]. The input ports are equipped with a buffer for each layer. A change in a layer translates in a change of the buffer. E.g. if a message traverses from layer 0 to layer 1, it changes from buffer 0 to buffer 1. Hence each physical connection (called channel) among the routers consists of multiple Virtual Channels [14].
(a) The flit in the buffer (gray box) is waiting to be routed to the next router as indicated by the arrow. However it cannot proceed, because the buffer in the next router is full as well. Its flit also waits to be routed. This continues till the cycle is complete.

(b) An additional cyclic dependency, if toroidal topologies are considered.

Figure 4.5: Two examples for cyclic dependencies that can occur in honeycomb topologies.
4.4 Honeycomb Topology

To avoid deadlocks in the network, dependencies among several messages must be prevented by breaking the cycle in which a message waits for the resources currently occupied by another message [20]. The dependency cycle can be broken by forbidding a turn (figure 4.6a). Since the connection links among the routers are bidirectional, a second turn must be removed such that no cyclic graph can be built. If the example depicted in figure 4.6a is transformed into a brick structure (figure 4.6b), it can be observed that the turns north to east and east to north are prohibited. This results in a routing algorithm which restricts messages that have to be routed north, to go always alternating to the west and north. However the algorithm does not restrict messages needed to be forwarded into southern direction.

(a) Allowed turns in a honeycomb topology. Two turns need to be prohibited to avoid deadlocks. A message that is sent north can only be routed to north-west and not to north-east. (2) marks this forbidden turn. A message that travels north-east cannot be routed to north, since turn (3) is forbidden.

(b) Same method of the Turn-Model applied to the brick structure. The “turns” marked with (1) are no turns anymore, since the message continues to travel into the same direction where it came from. (2) and (3) represent the forbidden turns.

Figure 4.6: Prohibited turns according to the Turn-Model.

If a destination is located in the north-east (exactly the direction that is prohibited to be taken directly), the destination must be either approached from the west requiring a toroidal topology, or the row in which the destination is located, must be passed so that the message can be sent south and east (example 1 in figure 4.7a). Both approaches result in additional latency. Even a more careful selection of the turns to forbid, result in an increase of hop counts by approximately 50% in average as it can be observed in
Routing Algorithms (Routing(.*).bsv) and Topologies

(a) Example 1: The restriction of turns leads to a significant increase of hops, if the destination is located in the north-east, a direction that is prevented by the Turn-Model (turns (3) and (4) are forbidden). Either the row in which the destination (DST) is located must be passed and the message needs to return to the row to be routed to east (refer to the dotted line marked with (1)) or a toroidal topology becomes mandatory, so that the destination can be approached from the west (2).

(b) Example 2: Compared to the shortest path ((1) with 5 hops), the Turn-Model forces a message to go a longer path (2) with 7 hops. First the message is forwarded to the west till the distance to the diagonal (refer to section 4.4.1) is less or equal than one. Then the message is routed along the diagonal to its destination. In this example the turns north to west (3) and south to west (4) are prohibited.

Figure 4.7: Two examples how the Turn-Model increases the latency of a message significantly.

element 2 depicted in figure 4.7b in which the turns north to west (3) and south to west (4) are prohibited.

In honeycomb topologies the number of possible outgoing links for any incoming message is only two, provided that a message cannot be send back into a direction where it came from. In addition only one of the possible outgoing links forwards the message into a good direction which leads to a hop closer to the destination. Hence a deterministic routing algorithm is the preferable option, since the number of choices offered to the routing algorithm in each node is very limited and any forwarding of a message into a bad direction due to forbidden turns or congestion in case of adaptive routing algorithms, results in a tremendous penalty to be paid in form of latency.

If the Turn-Model of example 1 in figure 4.7a is applied to a routing algorithm in layer 0 (or VC0), the algorithm can be described to favor
4.4 Honeycomb Topology

The routing algorithm of layer 0 favors north-west or south-east directions. Messages going that way, can be forwarded very efficiently towards their destinations (refer to figure 4.8a). Messages which need to traverse in north-east or south-west cannot be routed in this layer. Thus they will enter a higher layer in which a different routing algorithm is applied favoring north-east or rather south-west (refer to figure 4.8b). The additional routing algorithm also follows the Turn-Model by prohibiting the turns west to north and north to west. Hence it is deadlock free.

The routing algorithm is embedded in a toroidal square honeycomb topology. Hence apart from the dependency cycles discussed so far, additional dependency cycles must be broken (refer to figure 4.9) resulting in the necessity of 4 VCs in total. The VC is changed, if a message crosses the date line (that is if a message uses one of the links that creates the torus - linkT_{vertical} or linkT_{vertical}). Since the message is forwarded in a step-like pattern, it is not necessary to introduce an additional horizontal date line at linkT_{vertical}. If the destination is far away in the north (or south) automatically the vertical date line will be crossed.

The following rules are imposed to determine the VC of a message:

1. If a message crosses the date line from either direction, the VC number is incremented by 1.

Figure 4.8: Two layers of the network each of them with a different routing algorithm.
2. Messages are always inserted in layer 0 (VC0). They can either be routed north and west or, if the destination is in southern direction, they are forwarded to the east and south alternating.

3. In VC0 and VC1 messages are routed towards the diagonal (dotted line in figure 4.7b) on the shortest path. A message is only moved to VC2, if the diagonal has been reached. That is when the shortest path to the destination is to north-east or south-west. In VC2 and VC3 (latter one in case the message crossed the date line) a different routing algorithm is applied in which the turns west to north and north to west are prohibited. Hence this routing algorithm favors the directions north-east and south-west and the message is routed along this diagonal towards its destination.

Let $dy(a)$ be the vertical and $dx(a)$ the horizontal distance from the current node to the destination $a$. A node is on the diagonal with respect to the destination node, if $dx(a) = -dy(a)$.

4. The VC number is only incremented, i.e. a message can only move up in the layers.

Though routing algorithms for honeycomb topologies which are independent of VC, are introduced in [59, 53], they come with several restrictions:

1. The algorithm of both publications depends on complete combs as shown in figure 4.1a, but in many implementations the Fabric is not equipped with it. Consider figure 4.10 in which only one link leads to the node in the upper left corner. An unaware routing algorithm might send a flit onto that link assuming that the comb is complete. Since the router is not able to return flits on the same link they came from and if the node is not the destination, the flit gets stuck there. Since the honeycomb topology is not integrated in its natural way, it depends on toroidal links to complete the half comb in the upper left corner with the help of the upper two nodes on the right side. Toroidal links make it necessary to introduce VC to avoid deadlocks.
2. The routing algorithm in [59] does not take the shortest route and is similar in its behavior shown in figure 4.7b path (2).

4.4.1 Algorithm

In this section the stages are demonstrated, how the routing algorithm for the honeycomb topology has been developed. According to the Turn Model [20] an adaptive routing algorithm is deadlock free, if no circular dependency can be built. Since deterministic routing algorithms are a subset of the adaptive counterparts, the same technique can be applied. In figure 4.8 such a constellation is depicted and referred to as an example.

4.4.1.1 Behavioral Observations

First the rules for each IP are defined. Since different turns are forbidden in different layers or VCs, they need to be looked at separately. The complete rule set is listed in listing 4.1 and as it can be observed, they depend on the location (dir) of the IP and the current vcNo (number of the VC) of the flit. dx and dy are stored in the message header and represent a relative addressing scheme. In every node a message passes, the address is updated according to the direction which a flit is forwarded to. If $dx = dy = 0$, the destination has been reached and the message is ejected from the network. The boolean value northPort is a static constant that is independent from the message. It determines, if the current router has a neighboring router attached to the north port (true) or to its southern port (false). Thus, the routing algorithm consists of either the if (starting from line 6) or the else branch (line 14). This constant corresponds to the coloring of the nodes by Ivan Stojmenovic in [30]. For a better overview, datelines are not considered and omitted.
direction routingHoneycomb2D_VC(int dx, int dy, int *vcNo ←
, const bool northPort, const direction dir) {
    direction opNo = INVALID;

    if(*vcNo <= 1) {
        if(dir == SOUTH) {
            if(dx==0 && dy==0) opNo = EJECT;
            else {
                if(dx >= dy*(-1)) {
                    *vcNo = 2;
                    opNo = EAST;
                } else opNo = WEST;
            }
        }
        if(dir == WEST) {
            if(dx == 0 && dy == 0) opNo = EJECT;
            else {
                if(dy > 0) opNo = SOUTH;
                else opNo = EAST;
            }
        }
        if(dir == NORTH) {
            if(dx == 0 && dy == 0) opNo = EJECT;
            else {
                if(dx > dy*(-1)) opNo = EAST;
                else opNo = WEST;
            }
        }
        if(dir == EAST) {
            if(dx == 0 && dy == 0) opNo = EJECT;
            else {
                if(northPort) {
                    if(dy < 0) opNo = NORTH;
                    else opNo = WEST;
                } else {
                    if(dy > 0) opNo = SOUTH;
                    else opNo = WEST;
                }
            }
        }
        if(dir == EJECT) {
            if(northPort) {
                if(dy < 0) {
                    opNo = NORTH;
                } else if(dy > 0) {
                    if(dx >= dy*(-1)) opNo = EAST;
                    else {
                        *vcNo = 2;
                        opNo = WEST;
                    }
                } else {
                    opNo = INVALID;
                }
            } else {
                if(dir == SOUTH) {
                    if(dx == 0) {
                        opNo = NORTH;
                    } else {
                        *vcNo = 2;
                        opNo = WEST;
                    }
                }
            }
        }
    }
}
```c
Listing 4.1: Full rule set defining where flits need to be forwarded to depending on the relative address, the location of the IP, the VC and if the router has a port to the north.
This information has been obtained by merely observing, what directions are allowed. If there is only one outgoing port (for instance a flit in VC0 in the IP located in the SOUTH, cannot be routed to the east), the necessary action is obvious and the port should include only one possible outgoing direction. Since the algorithm consists of multiple layers, the eastern direction is only allowed, if the VC number is changed as done in line 9. The flit changes into another VC if it reaches the diagonal. This can only happen, if flit is traversed the NoC in a step like pattern and is stored in the southern buffer of the node on the diagonal.

4.4.1.2 if Branch Aggregation

As a first step of optimization conditions which are common for all if branches, are aggregated. For instance almost all branches include the condition for the ejection port. The only branch that does not include a condition for ejection, is the injection port itself. It is assumed that no flit is sent back into the same direction where it just came from. Hence it is safe to include the ejection as a condition into the injection port.

As it can be observed further is the similarity of the appropriate if branches. For instance the conditions for the northern IP are the same regardless the VC number a flit is currently residing. Listing 4.2 shows the result of the routing algorithm after the first optimization step.

```cpp
direction routingHoneycomb2D_VC(int dx, int dy, int *vcNo <-
    , const bool northPort, const direction dir) {
    direction opNo = INVALID;

    if(dx == 0 && dy == 0) opNo = EJECT;
    else {
        if(dir == SOUTH) {
            if(*vcNo == 0) {
                if(dx >= dy*(-1)) {
                    if(*vcNo <= 1) *vcNo = 2;
                    opNo = EAST;
                } else opNo = WEST;
            } else opNo = EAST;
        }

        if(dir == WEST) {
            if(*vcNo == 0) {
                if(dy > 0) opNo = SOUTH;
                else opNo = EAST;
            } else {
                if(dy > 0) opNo = SOUTH;
                else if(dy < 0) opNo = NORTH;
                else opNo = EAST;
            }
        }
    }
}
```
4.4 Honeycomb Topology

Listing 4.2: Rule set after the first optimization step

```c
}  
if (dir == NORTH) {
    if (dx > dy*(-1)) opNo = EAST;
    else opNo = WEST;
}

if (dir == EAST) {
    if (*vcNo == 0) {
        if (northPort) {
            if (dy < 0) opNo = NORTH;
            else opNo = WEST;
        } else {
            if (dy > 0) opNo = SOUTH;
            else opNo = WEST;
        }
    } else {
        if (dy > 0) opNo = SOUTH;
        else opNo = WEST;
    }
}

if (dir == INJECT) {
    if (northPort) {
        if (dy < 0) opNo = NORTH;
        else if (dy > 0) {
            if (dx < 0 && abs(dx) >= dy) {
                if (*vcNo <= 1) *vcNo = 2;
                opNo = WEST;
            } else opNo = EAST;
        } else {
            if (dx > 0) opNo = EAST;
            else if (dx < 0) opNo = WEST;
        }
    } else {
        if (dy > 0) opNo = SOUTH;
        else if (dx >= dy*(-1)) {
            if (*vcNo <= 1) *vcNo = 2;
            opNo = EAST;
        } else opNo = WEST;
    }
}
return opNo;
```
4.4.1.3 Virtual Channel Optimization

One aim is to optimize the routing algorithm till it is independent of VC, assumed this can be accomplished at all. First the conditions in the western port are examined. As it can be observed in the listing above, that in VC2 an additional condition \( dy < 0 \) has been inserted. Now the question that is being asked is: Can this condition be included into VC0 so that both VCs become the same? Or in other words: Why is this additional condition not required in VC0?

It is not necessary, because the turn from west to north is prohibited in VC0. The previous router to the west which has a port to the south, must have intentionally sent a packet to the east that needs to go to the north. This can happen, if a flit came from the south or from the west in that previous router. The flit could not have came from the south, since the turn from south to east is also prohibited in VC0. However this turn is allowed in VC2, hence the packet is already in that VC which does not forbid the link west to north in the current router.

If it came from the west in the previous router, then it should have been forwarded north in the router in the west of the previous one which did not happen. Hence that is not possible, the situation that a flit is in west and has to go north, cannot occur and inclusion of \( dy < 0 \) does not have any affect in VC0 on the behavior of the algorithm.

In the southern port the VC condition can be removed as well. The VC is changed, if the flit reaches the diagonal from south after which the flit is has to go east. The same happens, if the diagonal is located more to the right of the current location. In both cases the condition \( dx \geq dy*(-1) \) is true. Only if the diagonal has not been reached, the flit stays in its VC and is routed to the west as long as the diagonal or the destination is reached. In that case the condition \( dx \geq dy*(-1) \) is false and the VC is not changed.

In the eastern port a flit in VC2 is either forwarded to the west or south whereas a flit in VC0 is sent either north, south or continues to the west. Since the turn east to north is prohibited in VC2, we have to check, if the situation can occur in the first place before removing the VC condition.

A flit that is in VC2 of the eastern port and wants to go to north, is sent to this router by the previous router with a southern port. In the latter router the flit could have come from the east or south.

1. If it had come from the east, a change to VC2 would have had occurred only if it had come from the north in the previous router. But since flits do not go back into the same direction, this situation is impossible.

2. It cannot come from the south, because the turn south to west is prohibited in VC2. So a flit residing in VC2 of the eastern port, and which wants to go to the north, does not exist.
4.4 Honeycomb Topology

The optimizations result in a VC-independent routing algorithm as shown in listing 4.3.

```c
direction routingHoneycomb2D_VC(int dx, int dy, int *vcNo, const bool northPort, const direction dir) {
    direction opNo = INVALID;
    if(dx == 0 && dy == 0) opNo = EJECT;
    else {
        if(dir == SOUTH) {
            if(dx >= dy*(-1)) {
                if(*vcNo <= 1) *vcNo = 2;
                opNo = EAST;
            } else opNo = WEST;
        }
        if(dir == WEST) {
            if(dy > 0) opNo = SOUTH;
            else if(dy < 0) opNo = NORTH;
            else opNo = EAST;
        }
        if(dir == NORTH) {
            if(dx > dy*(-1)) opNo = EAST;
            else opNo = WEST;
        }
        if(dir == EAST) {
            if(northPort) {
                if(dy < 0) opNo = NORTH;
                else opNo = WEST;
            } else {
                if(dy > 0) opNo = SOUTH;
                else opNo = WEST;
            }
        }
        if(dir == INJECT) {
            if(northPort) {
                if(dy < 0) opNo = NORTH;
                else if(dy > 0) {
                    if(dx >= dy*(-1)) opNo = EAST;
                    else {
                        *vcNo = 2;
                        opNo = WEST;
                    }
                } else {
                    if(dx > 0) opNo = EAST;
                    else if(dx < 0) opNo = WEST;
                }
            } else {
                if(dy > 0) opNo = SOUTH;
                else if(dx >= dy*(-1)) {
```
Listing 4.3: Rule set after the second optimization step

4.4.1.4 Input Port Optimization

As last step the dependency on the location of the IP is removed by aggregation the conditions in which a flit is sent in a particular direction. If the following boolean expression is true, a flit has to be forwarded into the northern direction:

\[
\text{dy} < 0 \land \text{dir} = \text{WEST} \lor \text{dy} < 0 \land \text{dir} = \text{EAST} \lor \text{northPort} \lor \text{dy} \leftarrow < 0 \land \text{dir} = \text{INJECT}
\]

This expression is equivalent to

\[
\text{dy} < 0 \land \text{northPort} \land (\text{dir} = \text{WEST} \lor \text{dir} = \text{EAST} \lor \text{dir} = \text{INJECT} \lor \text{dir} = \text{NORTH} \lor \text{dir} = \text{SOUTH})
\]

\text{dir} = \text{NORTH} and \text{dir} = \text{SOUTH} can be safely added, because if a flit has to go north, the router will not have a southern port nor does it come from the northern direction. Hence both additions are always false not influencing the outcome of this expression. Since all ports have been included, the condition does no longer depend on the location of the IP. Hence the flit takes the northern OP if

\[
\text{dy} < 0 \land \text{northPort}
\]

Similarly a flit is routed south, if the following expression is true:

\[
\text{dy} > 0 \land \lnot \text{northPort}
\]

The aggregation of the eastern and western directions is simplified, if the amount of prohibited turns is increased. Strictly speaking a flit that needs to reach destination (1,3) from (1,0) can do so without changing the VC, since it does not encounter any forbidden turns. This results that the desired step-like pattern for the different VCs is not followed. By adding forbidden turns to VC0, the routing algorithm becomes more restricted by reducing the available routing options, but it stays deadlock free. To further simplify the routing algorithm, the turns south to west and east to south are
prohibited to enforce the step like pattern. As a result the VC is now changed for certain paths and a higher utilization of VC2 occurs that did not happen earlier. By now the same technique that was used to reduce the complexity for the western and easter directions, can be also applied to the northern and southern ones that gives the optimization shown in listing 4.4. As it can be observed, the dependency on the location of the IP could be completely eliminated. The algorithm only depends on the address of the flit and the availability of the northern port.

```cpp
1 direction routingHoneycomb2D_VC(int dx, int dy, int *vcNo
2 , const bool northPort) {
3     direction opNo = INVALID;
4     if(dx == 0 && dy == 0) opNo = EJECT;
5     else {
6         if(northPort) {
7             if(dy < 0) opNo = NORTH;
8             else {
9                 if(dx <= dy*(-1)) {
10                     if(*vcNo <= 1) *vcNo = 2;
11                     opNo = WEST;
12                 } else opNo = EAST;
13             }
14         } else {
15             if(dy > 0) opNo = SOUTH;
16             else {
17                 if(dx >= dy*(-1)) {
18                     if(*vcNo <= 1) *vcNo = 2;
19                     opNo = EAST;
20                 } else opNo = WEST;
21             }
22         }
23     }
24     return opNo;
25 }
```

Listing 4.4: Rule set after the final optimization step

The algorithm needs to be extended by invoking the function `getRouteHoneycombDataLine2DVC` for the routers located next to the dateline. The routing algorithm is in such a way, that only messages in VC0 or VC2 cross the dateline. Consider the situation in which a flit starts to travel north-west in VC0 and crosses the date line. Hence the VC is changed to VC1. However to reach its destination, the flit may turn into a north-east direction (VC2), if necessary. Eventually it will cross the dateline again and the VC will change to VC3. If the flit had continued to travel in north-west direction, it must cross the dateline again to violate this statement. However to travel that many hops into this direction will result in a crossing of the diagonal. Hence
Routing Algorithms (Routing(.*) .bsv) and Topologies

this situation cannot occur.

Listing 4.5: The dateline increasing the VC in case the flit uses a toroidal link.

4.4.2 Limitations of the Routing Algorithm

Since each message is allowed to cross the date line twice at maximum on the way to its destination, a quadratic topology with nodes\(_{\text{rows}} = \text{nodes}_{\text{cols}}\) is preferable. In rectangle configurations either the number of VCs needs to be incremented resulting in a design that requires more area and power, or the date line needs to be shifted. As shown in figure 4.9 the date line is located at the horizontal links creating the torus (\(\text{linkT}_{\text{vertical}}\)) since \(|\text{linkT}_{\text{vertical}}| < |\text{linkT}_{\text{vertical}}|\).

The number of nodes in the rows and columns cannot be chosen arbitrarily in a toroidal rectangular honeycomb topology. The routing algorithm requires a homogeneous network topology consisting of honeycomb cells only. Thus \(\text{nodes}_{\text{rows}}\) and \(\text{nodes}_{\text{cols}}\) must be even.

4.5 Mesh Topology

To avoid deadlocks in mesh topologies the same technique of prohibiting turns is applied. The turns that a flit cannot take are depicted in figure 4.11 which results in an algorithm that can be described as west first given in listing 4.6. Once a flit has been routed to the north or south, it cannot be forwarded towards the west anymore which means, if a packet has a destination in the west, it must first go to west before it can head anywhere else.
4.5 Mesh Topology

Figure 4.11: The turns that are forbidden in the mesh topology are marked. This results in routing rules in which the west direction has to be considered first. The dotted lines show the location of the date lines.

```
if(dy == 0 && dx == 0) return EJECTPORT;
if(dx < 0) return WEST; // west first
if(dx > 0) return EAST;
if(dy > 0) return SOUTH;
return NORTH;
```

Listing 4.6: The west-first routing algorithm for mesh topologies in C language

Originally the west-first algorithm does not depend on VCs to ensure deadlock freeness. However since the same algorithm is applied in a toroidal network, a vertical and horizontal dateline is introduced. This algorithm is efficient and routes the packets onto the shortest path from the source to the destination. Since VCs are available due to the requirement of the honeycomb routing algorithm, they can be also used in this case and a more complex algorithm is not necessary.

Extensive research has been done in the field of mesh topologies due to their convenient arrangement of nodes in a regular grid like in [28, 18, 26, 56] and many more. Hence it is not further discussed in this work.
4.6 Hexagonal Topology

The implemented routing algorithm for hexagonal topologies has been introduced in [62]. It provides the shortest path between a source and destination and does not depend on VCs by design. Again like in the mesh, the Fabric is a toroidal hexagonal network. To use the same algorithm, datelines have been introduced at the boundaries of the Fabric.

To ease the understanding of the mapping of application onto a Fabric, systems designers usually rely on 2D addressing schemes only. However since a new direction compared to mesh and honeycomb is added, this algorithm requires a three tupled address. Hence the 2D address that is given in the packet header, needs to be converted into a 3D one. As it can be observed in figure 4.12 multiple absolute addresses can be used to describe the location of the same point $P$ from origin $O$: $(-1, -2, 0)$, $(1, 0, 2)$, $(0, -1, 1)$, etc.

![Figure 4.12: Multiple possibilities are provided to describe the position of $P$](image)

Since flits should be routed on the shortest path to the destination, the first 2 addresses include 3 hops and hence are not used. The last address routes a packet within 2 hops to its destination. In [62] it has been shown that the relative address for the shortest path has always of the following format: At least one component of the 3D address is zero. If that is the case, one of the remaining elements is positive whereas the other one is negative.

The algorithm for converting a 2D address into the described format of a 3D address is given in listing 4.7.

```
void generateNewRelAddr(int &dx, int &dy, int &dz) {
    int temp = min(abs(dx), abs(dy));
    if(dx > 0 && dy > 0)
        temp *= -1;
    else if(dx >= 0 || dy >= 0)
        temp = 0;
    dx += temp;
    dy += temp;
    dz = temp;
}
```

Listing 4.7: The conversion algorithm to convert a two tupled address into a three tupled one
4.6 Hexagonal Topology

4.6.1 Proof

The algorithm converts a relative 2D address into a relative 3D address so that the flit traverses on the shortest path. In 2D addressing nine possibilities of how the tuple can look, exist like as there are: 

- \((+, +), (+, -), (-, +), (-, -), (0, +), (+, 0), (-, 0), \) and \((0, 0)\).

- In case the tuple is \((+, +)\) (line 4), \(dz(a)\) is not going to be 0. Instead it is the complement of the lower value of \(dx(a)\) or \(dy(a)\) and hence negative. Since \(dz(a)\) will either be equal to \(-dx(a)\) or \(-dy(a)\) one or both are nullified meeting the requirement of the shortest path routing.

- Similar if the tuple is in the format \((-,-)\) (bypassing the \(if\) and \(else\) branch). In that case, \(dz(a)\) is positive and similar to the reasoning above one or both components of the 2D address will be set to 0 in the addition sequence starting in line 9. If e.g. \(dy(a) < dx(a) < 0\), the result will be:

\[
dy_{\text{new}}(a) = dy_{\text{old}}(a) + dx_{\text{old}}(a),
\]

\[
dx_{\text{new}}(a) = dx_{\text{old}}(a) + |dx_{\text{old}}(a)|,
\]

\[
dz_{\text{new}}(a) = |dx_{\text{old}}(a)| > 0
\]

In both cases the route that a flit takes without conversion will be an L shape in figure 4.12 and it can be observed that by introducing a path parallel to the z axis, the way can be cut short. E.g. consider the route from \(P\) to \(O\). Instead of traveling the route \((1, 2)\) which comprises of 3 hops, the original relative address is \((0, 1, -1)\), resulting in 2 hops.

- All other cases are handled by the \(else if\) branch in line 6. If one of the 2D address elements is equal or greater than 0, the other one needs to be either equal to 0 \(((0, 0), (+, 0), (0, +))\) or is less than 0 \(((+, -), (-, +), (0, -), (-, 0))\), because the case of \((+, +)\) has already been handled by the earlier \(if\) condition. In this situation the flit is either traversing in parallel to one of the \(x\) or \(y\) axis (if one of the component of the 2D address is equal to 0) or the \(z\) component is counterproductive and does not help in finding a shorter path. Hence \(dz(a)\) is set to 0 meeting again the requirement of the shortest path routing.

To avoid deadlocks the routing algorithm in [62] can be described as a negative-first algorithm. If it is taken under consideration that there is at
maximum only one negative component in the 3D address tuple, since one of the remaining elements must be either positive, if the last one is 0, or both are equal to 0, a simplified priority based algorithm can be used checking first for the negative address tuples. Such an algorithm is given in listing 4.8.

```c
1 direction getRouteHexa_NegFirstnoVC(int dx, int dy, int dz) {
2     if(dx == 0 && dy == 0 && dz == 0)
3         return EJECTPORT;
4
5     if(dx < 0) return WEST;
6     if(dy < 0) return NORTHEAST;
7     if(dz < 0) return SOUTHEAST;
8     if(dx > 0) return EAST;
9     if(dy > 0) return SOUTHWEST;
10    return NORTHWEST; // for dz > 0
11 }
12
13 direction getRouteHexaDateLine_NegFirst2DVC(direction dir ← [3], int dx, int dy, int dz, int &vc) {
14     direction nextOutput = getRouteHexa_NegFirstnoVC(dx, dy, dz);
15     if(nextOutput == in_array(dir)) vc++;
16     return nextOutput;
17 }
18
Listing 4.8: The negative-first routing algorithm in hexagonal topologies.

4.7 Summary

In this chapter the routing algorithm for three topologies (honeycomb, mesh and hexagonal) has been described. In addition other topologies of potential interest such as the Flattened Butterfly, Spidergon and Stargon topologies have been introduced. By using a step-by-step explanation method, the system designer is able to develop new routing algorithms for other topologies as well. The algorithms are tightly connected to the chosen topology. Currently RECONNECT supports the three topologies mentioned earlier at the same time by superimposing the honeycomb onto a mesh and further the mesh onto the hexagonal topology. Again the system designer has the freedom to tailor RECONNECT to be integrated into one only or into a new topology such as a ring by disabling additional [IP][OP] pairs.
Chapter 5

Test Case: REDEFINE

The architecture in which RECONNECT is integrated is called REDEFINE, a dataflow oriented, reconfigurable multi-processor architecture (refer to [57]). It consists of supporting logic (refer to section 5.2) and a portion performing the computation called Fabric as described in section 5.3. The Fabric comprises multiple tiles and an interconnect realized by a Network on Chip (NoC). Each tile consists of an NoC router and a Compute Element (CE) which is equivalent to a computation core in modern personal computers. Depending on the requirement and which algorithm is executed on REDEFINE, CEs can be replaced by specialized ones with extra functionality provided by added or replaced modules in the Arithmetic Logical Unit (ALU). Hence the Fabric becomes heterogeneous, if requirements dictate.

A data flow processor is a processor lacking a program counter stating the next instruction to be executed. Instead instructions and their operands are marked as ready, if all operands have arrived. After readiness has been achieved, the instructions are scheduled to be executed. Hence the order of execution is not determined which results in a higher utilization of the ALU due to the avoidance of delays for completion of load and store instructions. In real cases however memory dependencies must be respected such as in the example listing 5.1.

```
1 a = b + c;
2 if (a > 10)
3     d = 2 * a;
4 else
5     d = d << a;
```

Listing 5.1: An example showing dependencies among instructions

As long as a is not computed, neither the multiplication nor the shifting can be executed. To ensure correctness, dependencies are introduced by predicking instructions. So in this example the shift and the multiplication instruction are predicated. After calculating a, both are ready to execute.
However the if condition will only generate one predicate and depending on that only one of the following instruction is executed, whereas the other one is squashed.

5.1 RETARGET

To run applications on REDEFINE the application written in the high-level language C, is compiled by RETARGET [1]. RETARGET generates a data flow graph (DFG) of the application in which the nodes represent Basic Blocks (BBs) and the edges their dependencies among each other. The DFG is then further processed to obtain Hyper Operations (HyperOp) [2,3] that are acyclic, disjunct, and directed subgraphs of the DFG. In addition it is required that between two HyperOps there is always a producer-consumer relationship (Convexity condition). Hence a HyperOp cannot produce data for another HyperOp which is also a producer for the first HyperOp at the same time (refer to figure 5.1 in which HyperOp 1 and 2 cannot be merged, since the merged HyperOp would produce to HyperOp 3, but also consume from the same).

Loops that occur in the source code of the application, cannot be handled natively by data-flow processors, since loops require instructions to control their execution. For instance, assume that two HyperOps are in a producer-consumer relationship. It needs to be ensured that a buffer overflow does not occur which can easily happen, if the loop inside the producer is executed at a faster rate. Hence control is required to slow down the producer. In addition it needs to be ascertained in which loop iteration a data token that is destined for the consumer, is generated. To accomplish this hardware support is integrated into the Support Logic (SL) of REDEFINE to be able to distinguish multiple data tokens coming from different loop iterations (but from the same instruction).

Further each HyperOp is divided into a single or multiple partial Hyper-Op (pHyperOp) which are launched on a CE each. Depending on the size of
the local memory available in each CE, the size of the pHyperOps also differ. For launching the application RETARGET is aware of the underlying topology within the Fabric. It tries to arrange the pHyperOps of each HyperOp on physical closely located CEs so that communication delays caused by the NoC are minimized.

5.2 Support Logic (SL)

The SL launches the HyperOps onto the Fabric, which in turn executes the instructions. Returning results might be used to launch other HyperOps which are ready to be launched, if all inputs become available. This procedure continues till the application terminates. There are multiple steps involved on how HyperOps are launched onto the Fabric. They are briefly introduced in this section.

1. The Scheduler is aware of HyperOps which are ready to be launched. The readiness depends on the availability of the inputs for each HyperOp. Obviously to start any application there is at least one or often multiple HyperOps which do not have any input values and hence can be launched shortly after the reset. The information of HyperOps ready for launch, is forwarded to the Resource Binder (RB).

2. The RB maintains a matrix representing the state of each CE in the Fabric. The resource requirements viz. the number of needed CEs and their distances among each other, are dictated by the compiler [1]. Hence the RB checks the matrix of available CEs, if there is a combination which matches the mapping requirement. After it found one, it informs the HyperOp Launcher (HL) which specific HyperOp needs to be launched and the coordinates of the location in the Fabric.

3. The HL downloads the HyperOp that consists of instructions and required constants, from its attached memory banks. Currently 5 banks allow a parallel download. It forms NoC compatible packets by generating relative addresses for all instructions in the pHyperOp. Depending on the location to where data has to be sent, the Access Routers (ARs) in the Fabric are contacted and the packet is handed over. The purpose of the ARs is to establish connections from the SL to the Fabric to inject and eject data as it can be observed in figure 5.2. Currently only row wise launching is supported meaning that if a destination CE is in the second row of the Fabric, the data is injected into the AR of the same row.

4. During the execution of pHyperOps new data can be generated which is either destined to another pHyperOp or to a HyperOp waiting in the
Scheduler. In the first case the data is immediately forwarded to the CE. If a HyperOp is waiting for the data, it is forwarded to the ARs at the boundary of the Fabric which in turn sent it to the Inter HyperOp Data Forwarder (IHDF).

Another mechanism to generate data is caused by an instruction initiating a request to retrieve from memory or to store a value into it. Load or store instructions are forwarded to the Load/Store Unit (LSU) attached to each AR.

Figure 5.2 shows the relationship of the described modules of the SL. It also displays the Intellectual Property Controller (IPC) which controls the SL and, among other responsibilities, loads data into the memory banks of the HL. In the Execution Fabric tiles marked with T comprises an NoC router and a CE. Tiles marked with A represent ARs providing connectivity from and to the Fabric.

![Diagram of REDEFINE and its major modules and their relationship among each other](image)

**Figure 5.2:** An overview of REDEFINE and its major modules and their relationship among each other

### 5.3 Fabric

Using an NoC for the Fabric of REDEFINE has practical but also historical reasons. Compared to a bus system an NoC is scalable and decentralized.
It does not require a global arbitration mechanism and all units can inject data at the same time into the NoC. The router will handle forwarding and congestion automatically.

In early stages of REDEFINE, the memory for each CE was a separate node in the NoC as shown in figure 5.3. It becomes obvious that if the memory supplied the operands and the computed result was again stored in a memory, the preferred topology was the honeycomb with a node degree of three (excluding the link to the sink/generator). Many operations require two operands and produce one result which means that a computation graph of an application could be directly mapped onto the Fabric.

However, the communication overhead was immense and hence it was decided to integrate the memory into the CE. Since then, the honeycomb topology became less important, so multiple different topologies can be selected and compared in terms of performance, area, and power consumption of the routers.

5.4 Minimum Flit Sizes in Multiflit Environments

If the router supports packets segmented into multiple flits (refer to section 3.2.3), it is important to know what the bus width of the connections among the routers should be. A lower width will ease the wiring efforts and complexity of muxes, demuxes, FIFO buffers, etc., but on the other hand require multiple clock cycles to transfer a packet. A wider bus raises the chances that a packet can be completely transmitted within a single flit. So, a compromise needs to be found to satisfy both, high performance and low complexity.
Since REDEFINE can be used for a variety of scenarios, the optimum flit size is not determinable at the time of writing of this thesis. However in this section some common algorithms are introduced that were executed on REDEFINE. The total time an algorithm requires till it terminates, depends on the occurrences of packet types (see table 5.1).

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Sizes [Bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Packet</td>
<td>85</td>
</tr>
<tr>
<td>Sticky Token</td>
<td>74</td>
</tr>
<tr>
<td>LSU Packet</td>
<td>82</td>
</tr>
<tr>
<td>IHDF Packet</td>
<td>82</td>
</tr>
<tr>
<td>Operand Packet</td>
<td>55</td>
</tr>
<tr>
<td>Operand Mask</td>
<td>116</td>
</tr>
<tr>
<td>Predicate</td>
<td>55</td>
</tr>
<tr>
<td>Predicate Mask</td>
<td>50</td>
</tr>
<tr>
<td>Custom Instruction</td>
<td>39</td>
</tr>
<tr>
<td>Topology Configuration</td>
<td>20</td>
</tr>
</tbody>
</table>

\[
\text{Size}_{\text{flit}} = \text{Size}_{\text{packet}}
\]

Table 5.1: The size of the different packet types used by REDEFINE in the Fabric. Displayed is the required bus width to transfer the packet as a whole, which apart from the payload itself also includes the address tuples (3 × 4 bits), VC number (2 bits) and template/union bits stating which packet type is valid (4 bits).

The minimum size of a flit is currently 16 (14 without VC) bits, since the address tuple (3 × 4 bits), the VC number (2 bits) and flit distinction information (2 bits) are expected to be stored in the header flit. If the constant MULTITOPOLOGY (refer to section 3.2.11) is defined, 6 bits need to be added to the minimum flit size for the payload of the Topology Configuration Packet.

Consider the scenario, in which the minimum flit size is less than 22 bits and hence the topology configuration packet is split into a header and a tail flit carrying partly or the whole payload. After the direction for the head flit has been calculated, the router configures its registers for the new topology setting and forwards the head flit. However it might happen that in the next clock cycle, when the new values of the registers become valid, some of the Input Ports (IPs) are powered down to reduce power consumption. In that case, the tail flit might get stuck in one of the just powered down IPs which leads to a situation in which there is a head flit without a tail flit. Since interleaving of flits is not allowed, the resources along the path of the head flit are blocked and cannot be reused. Hence it is imperative that the payload of the topology configuration packet (2 bits) fits into a single flit. To allow the router to determine what packet is currently passing through, the template bits (4 bits) are also required.

In case of multiflit environments the router needs to distinguish between
5.4 Minimum Flit Sizes in Multiflt Environments

a header flit, containing the address and VC number, payload flit, tail flit and flits that are so small that the intended packet payload fits into the payload field of the header flit. The latter flits are called head-tail flits. Since the information that is carried in the header flit, remains constant, the payload field in the header flit increases, if the flit size is increased. If the size is increased beyond the largest packet, the only flit type that is transmitted in the NoC are head-tail flits. To allow the router distinguishing between a header, payload, tail and head-tail additional 2 bits are required for each flit transmitted. The new packet sizes depend on the number of flits each packet is segregated into and it is calculated in equation 5.1.

\[
\text{Size}_{P, MF} = \text{Size}_{FP, MF} + \text{Size}_{HP, MF}
\]  

(5.1)

with

\[
\text{Size}_{FP, MF} = \left\lfloor \frac{\text{Size}_{P, SF}}{\text{Size}_F - 2} \right\rfloor \times \text{Size}_F \\
\text{Size}_{HP, MF} = \begin{cases} 
0 & \text{if } \text{Size}_F \times n = \text{Size}_{P, SF} \\
\text{Size}_{P, SF} \mod (\text{Size}_F - 2) + 2 & \text{otherwise}
\end{cases}
\]

\(\text{Size}_{P, MF}\) is the new packet (P) size in bits in multiflt (MF) environments, whereas \(\text{Size}_F\) is the flit size. \(\text{Size}_{FP, MF}\) (FP = FullPacket) is the size of flits whose payload fields are completely filled with the information of the original packet, but the payload field of the last flit (\(\text{Size}_{HP, MF}\)) might not be completely used (HP = HalfPacket). \(\text{Size}_{HP, MF}\) equals 0, if the whole packet fills out the payload fields of the flits completely and no space is unused. However if it is used, then 2 bits are added to the remaining payload of the packet due to the distinction bits.

Equation 5.1 displays the physical sizes of the new packets. Consider an example in which the IHDF packet of 82 bits in total is sent over a link of 24 bits width. The equation calculates the total amount of information that needs to be transmitted over that link. The result will be \(\left\lfloor 3.6 \right\rfloor \times 24 + (82 \mod 22 + 2) = 90\) bits. In other words: The IHDF packet is divided into 4 flits and each additional flit add 2 bits to determine the type of the flit.

However flit distinction bits are completely hidden from the developer and are automatically inserted by the Bluespec System Verilog (BSV) compiler. Hence in this section in which the application behavior is analyzed for various flit sizes, these distinction bits are not further considered.

Apart from the Topology Configuration Packet all other packets are destined for the CEs and SL and are hence left untouched by the routers. The Topology Configuration Packet is examined by the routers and it configures
Test Case: REDEFINE

itself to the desired topology accordingly. This packet is dropped by the Assembly Unit (AU) once it reaches its destination. After a reset is issued to REDEFINE, it takes around 100 cycles till the first HyperOp is launched onto the Fabric. In the meantime the Fabric is configured for the desired topology which does not result in an overhead regarding the application execution. Hence in the following examples, the timer to estimate the execution time in cycles for the various applications is started only after the first payload for a CE is injected into the Fabric and stopped when the last flit has left the Fabric. This time spent on the Fabric, is henceforth called Fabric Execution Time.

5.5 Fabric Execution Time

5.5.1 Cyclic Redundancy Check (CRC)

CRC is a very small application which is compiled into only three HyperOps. One HyperOp consists of two pHyperOps whereas the other two consist only of one pHyperOp each. The current mapping algorithm uses a naïve approach to map these pHyperOps onto the Fabric. It tries to maximize the available memory bandwidth by preferring a mapping in which the pHyperOps belonging to the same HyperOp are arranged in a vertical fashion to potentially maximize the amount of paths to all ARs.

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Occurrences</th>
<th>Sizes [Bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Packet</td>
<td>6413</td>
<td>85</td>
</tr>
<tr>
<td>Sticky Token</td>
<td>514</td>
<td>74</td>
</tr>
<tr>
<td>LSU Packet</td>
<td>772</td>
<td>82</td>
</tr>
<tr>
<td>IHDF Packet</td>
<td>774</td>
<td>82</td>
</tr>
<tr>
<td>Operand Packet</td>
<td>7443</td>
<td>55</td>
</tr>
<tr>
<td>Operand Mask</td>
<td>0</td>
<td>116</td>
</tr>
<tr>
<td>Predicate</td>
<td>1281</td>
<td>55</td>
</tr>
<tr>
<td>Predicate Mask</td>
<td>255</td>
<td>50</td>
</tr>
<tr>
<td>Custom Instruction</td>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>Topology Configuration</td>
<td>12</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 5.2: Occurrences of the various packet types during the execution of a CRC against the sizes of the packets.

Table 5.2 shows the amount of packet types that are sent through the NoC for the CRC application. Further it is investigated, how the Fabric Execution Time suffers for the CRC application, if the flit size is continuously reduced and different topologies are used. In figure 5.4 it can be observed that the number of flits does not always increase, if the flit size is reduced. As an
example a flit size of 116 bit or 87 bit does not show any impact. The reason is that the largest packet that carries the Operand Mask, does not occur during the execution at all. It would only be this packet type which is divided into two flits after a reduction to a 87 bit wide connection among the routers. In case of the CRC application using an NoC supporting flit sizes greater than 87 bits, cannot be justified, since the additional wire and buffer would not be utilized, but on the other hand would increase the power and area consumption of the routers.

At a flit size of 86 bits, the Instruction Packet is divided into two flits. Originally only 85 bits in size, the payload of this packet would easily fit into this flit size. However it needs to be kept in mind that RECONNECT is not aware of the packet types that finally need to traverse through the NoC. Hence provisions that even the largest packet (i.e. the Operand Mask Packet) is transmitted, are already made, when the flit size is reduced from 116 to 115 bits and the constant MULTIFLITSUPPORT (refer to section 3.2.3) is set. Hence the flit distinction bits allowing to distinguish among head, headtail, payload and tail flits, have already been added increasing the packet size of the Instruction Packet to 87 bits in total. Thus at 86 bits the Instruction Packet is divided in a header and a tail flit both carrying a part of the payload. Since it occurs often during the execution of the CRC application, it has a negative impact on the number of flits transmitted through the network.
At flit sizes of 116 bits a steep increase of unused bits occurs caused by freed bits which were formerly used to distinguish header, headTail, payload and tail flits. These freed bits are made available to carry payload. But since the largest packet does not occur as described above, these additional bits are still not utilized and left unused.

The interesting flit sizes and henceforth used for further investigation are at the steps at which the number of required flits increases, whereas the number of unused bits is minimal (refer to figure 5.4). Hence the behavior of REDEFINE for flit sizes of 116, 87, 84, 76, 57, 52, 43, 39, 30, 27, 24, 22, 21, and 20 bits is analyzed further.

![Figure 5.5](image)

*Figure 5.5: The mapping of the HyperOps of the CRC application onto the Fabric. CRC consists of 3 HyperOps: 2 of the are mapped onto one CE only (striped area around (0,0)) whereas the larger one occupies 2 CEs (grayed out area). The thick links exist for the honeycomb topology, the mesh topology consists of the thick links and the thin link, whereas hexagonal also includes the dotted link.*

The mapping of CRC is shown in figure 5.5 and it can be immediately concluded that there is not much difference in execution time expected between a mesh and honeycomb topology since the honeycomb topology provides almost the same subset of links for this particular position on the Fabric. The only situation in which the execution time can vary, is a congestion on the link between node (0,0) and the AR at (5,0) which is also shared by node (0,1) in the honeycomb topology. In the mesh topology node (0,1) uses the path through (5,1) to reach (5,0), since the algorithm follows a west-first policy. Thus dedicated resources for this path are available. For a congestion to occur, sufficient amount of data needs to be generated. If the congestion does not take place, the latency between (0,1) and (5,0) is the same for mesh and honeycomb topologies. As it can be observed in figure 5.6 it seems that the router at (0,0) does not need to delay any traffic due to an over-utilized link between (0,0) and (5,0) and hence a congestion which would lead to an advantage of the mesh topology, does not appear.

If the mesh topology is compared with the hexagonal one, only a small amount of reduction is expected. Especially if flits have to be sent from (5,0) to (0,1). These packets include LSU, IHDF and Instruction Packets only. In addition only one of the HyperOps benefits from this additional link: The one that consists of two pHyperOps of which only the pHyperOp that is launched onto (0,1) is using the new link offered by the hexagonal topology. The other
two HyperOps are launched onto node (0,0) sequentially only and do not benefit at all. A marginally gain in Fabric Execution Time that is promised by a higher radix, can barely be observed in figure 5.6. In fact it is negligible for all considered flit sizes.

For all topologies the execution of the application is fastest, if the flit size is equal to the maximum packet size. Serialization latency required to segment the packet into multiple flits, does not occur in that case. As expected the execution time increases, if more flits are required to send a packet. Depending on which type of packet is segmented and the number of occurrences of that packet during the execution, the execution time increases either significantly (observe the steep increase between a flit size of 30 and 27 bits) or only marginally (among the flit sizes of 57, 76 and 84 bits).

![Figure 5.6: The Fabric Execution Time against the maximum flit size for CRC in various topologies.](image)

5.5.2 Advanced Encryption Standard (AES) Decryption

Compared to the HyperOps of the CRC algorithm, the ones of the AES decryption (AES-D) algorithm do not have a high degree of dependency among each other and hence can be executed in parallel. Figure 5.7 shows a snapshot of mapped HyperOps onto the Fabric since not all HyperOps fit on the Fabric at the same time and need to be scheduled.

In figure 5.8 the execution time of the Fabric is plotted against the bit sizes of the flits. As it can be observed, the benefits of a higher radix depends
largely on the flit size that can be transmitted: Whereas a flit size of 43 or 52 result in a 5.5% reduction of execution time between honeycomb and mesh/hexagonal, it is only 4.8% or less for larger or smaller flit sizes. When mesh and hexagonal topologies are compared, a higher radix has only a marginal effect which suggests that the \( \text{pHyperOps} \) are not mapped to CE locations in which the additional links provided by the hexagonal tessellation can be used. In fact, since there is no inter-HyperOp communication between the launched ones, it can be derived from figure 5.7 that mostly only vertical links are used for inter-HyperOp communication.

The main difference between AES-D and CRC of the previous section is the lower dependency of the HyperOps between each other. This results in less packets whose sole purpose is to maintain the order of read and write requests from and to the memory located in the LSU. These dependencies are controlled by Predicate packets which need to be generated resulting in a higher amount of Instruction Packets. Table 5.3 shows the amount of packets that have been transmitted through the NoC in detail.

### 5.5.3 SOBEL Edge Detection

Interestingly and compared to the other applications executed on REDEFINE, the edge detection does not always result in a higher Fabric Execution Time, if the flit size is decreased. As it can be observed in figure 5.9, the execution time for honeycomb and mesh topologies is marginally reduced, if the flit size is also reduced from 76 to 57 bits. For the hexagonal topology the same can be observed, if the flit size is changed from 84 to 76 bits.

In addition at very small flit sizes, the hexagonal topology performs worse when compared with mesh. A thorough examination of the traffic pattern revealed that at a flit size lower than 21 bits, only the packets destined for
5.5 Fabric Execution Time

![Graph of Fabric Execution Time against Flit Sizes](image)

**Figure 5.8:** The Fabric Execution Time against the maximum flit size for AES decryption in various topologies.

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Occurrences</th>
<th>Sizes [Bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Packet</td>
<td>3531</td>
<td>85</td>
</tr>
<tr>
<td>Sticky Token</td>
<td>248</td>
<td>74</td>
</tr>
<tr>
<td>LSU Packet</td>
<td>64</td>
<td>82</td>
</tr>
<tr>
<td>IHDF Packet</td>
<td>351</td>
<td>82</td>
</tr>
<tr>
<td>Operand Packet</td>
<td>3803</td>
<td>55</td>
</tr>
<tr>
<td>Operand Mask</td>
<td>0</td>
<td>116</td>
</tr>
<tr>
<td>Predicate</td>
<td>46</td>
<td>55</td>
</tr>
<tr>
<td>Predicate Mask</td>
<td>8</td>
<td>50</td>
</tr>
<tr>
<td>Custom Instruction</td>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>Topology Configuration</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>8063</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.3:** Occurrences of the various packet types during the execution of an AES decryption against the sizes of the packets.
Figure 5.9: The Fabric Execution Time against the maximum flit size for SOBEL edge detection in various topologies.

the LSU and IHDF require one more flit to accommodate their payload. Hence compared to the configuration in which the flit size is 21 bits, more flits are required to execute the same application with flit sizes of only 20 bits. The IHDF packet affects the time at which successive HyperOps are launched onto the Fabric, and since more flits are required for this packet type, all successive HyperOps are delayed. Therefore the CEs newly assigned to following HyperOps, are also expected to produce results at a later point in time. In other words: The production of results is temporally shifted which would logically lead to the conclusion that the Fabric Execution Time is higher at flit sizes of 20 bits compared to configurations with 21 bits.

However due to the temporal shift of HyperOp launches in configurations in which the flit size is 21 bits, sudden congestions of a short duration occur at the nodes neighboring the AR. These congestions which can not be observed, if the flit size is reduced to 20 bits, delay the launch of successive HyperOps. The advantage of larger flit sizes of 21 bits and thus less number of flits traversing through the NoC are nullified and cause a marginal increase of 1.4% of total execution time when compared to flit sizes consisting of 20 bits.

5.5.4 Further Application Examples

AES Encryption, Elliptic Curve Point (ECP) Addition (ECPA) and Elliptic Curve Point (ECP) Doubling (ECPD) algorithms, Givens rotation, matrix mul-
5.5 Fabric Execution Time

Multiplication, and LU decomposition are also executed on REDEFINE. However these algorithms do not give any further findings, since their behavior is similar and comparable with the already discussed ones. Summarized it can be said that in large applications the richest topology i.e. the hexagonal one, always performs best, whereas the honeycomb topology shows the highest execution time. In small applications such as the LU decomposition, the differences among the topologies are negligible. In general the execution time increases, if the flit size is reduced. But sometimes and on rare occasions the same behavior that was observed while executing the Sobel edge detection algorithm (a smaller flit size leads to a lower execution time), is detected. The graphs about Fabric Execution Timings for the mentioned applications are attached to appendix A.

It needs to be kept in mind that the results regarding the Fabric Execution Time are specific for REDEFINE and its compiler. The conclusions drawn from these results that were obtained by compiling and executing the applications, do not necessarily be true for all existing or future applications. Due it is impossible to comment on the performance of the NoC for all relevant applications, it is thus necessary to carry out the performance study with synthetic traffic patterns that are independent of the architecture and the compiler, as in Chapter 6.

5.5.5 Flattened Butterfly

To estimate the benefits of the topology of a Flattened Butterfly [34], the traffic pattern of the executed applications needs to by analyzed further. Especially the hop count is of importance, since the Flattened Butterfly drastically shorten it: From any source to any destination, the hop count is reduced to a maximum of two hops only (refer to figure 4.3 on page 45). The distances for the various applications is tabulated in table 5.4. Since the Flattened Butterfly shortens the distance in either x-direction or y-direction but not both at the same time, the distances given in the table, are shown direction-wise. For instance in a mesh topology the hop count equals two, if a flit has to travel to the relative address (2,0). Since all nodes in a row are fully connected in the Flattened Butterfly topology, it would require only one hop, reducing the latency of the flit. However, if the flit has to travel to the relative address of (-1,1) the hop count in both topologies is equal namely two. If the dimensions were not considered, the topology of the Flattened Butterfly would reduce the hop count to one also in this case giving the Flattened Butterfly an undeserved performance advantage.

As it can be observed the average hop count for the flits is approximately 1 and 0.37 hops in average in x and y direction respectively. The SL of RE-DEFINE is quite successful in mapping pHyperOps with dependencies close to each other although the compiler is not yet aware of the used topology. Once the compiler is extended to include also the topology in its mapping
Table 5.4: Distances the flits have to travel during the execution of various applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Direction</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESD</td>
<td>x</td>
<td>516</td>
<td>4585</td>
<td>1957</td>
<td>1253</td>
<td>0</td>
<td>1.47</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>7402</td>
<td>420</td>
<td>170</td>
<td>162</td>
<td>157</td>
<td>0.23</td>
</tr>
<tr>
<td>AESE</td>
<td>x</td>
<td>674</td>
<td>4678</td>
<td>2199</td>
<td>1235</td>
<td>0</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>7697</td>
<td>478</td>
<td>255</td>
<td>242</td>
<td>114</td>
<td>0.25</td>
</tr>
<tr>
<td>CRC</td>
<td>x</td>
<td>2303</td>
<td>15663</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>14907</td>
<td>3071</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.17</td>
</tr>
<tr>
<td>ECPA</td>
<td>x</td>
<td>614</td>
<td>4809</td>
<td>1617</td>
<td>876</td>
<td>0</td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>7167</td>
<td>389</td>
<td>170</td>
<td>146</td>
<td>44</td>
<td>0.17</td>
</tr>
<tr>
<td>ECPD</td>
<td>x</td>
<td>364</td>
<td>2793</td>
<td>1002</td>
<td>505</td>
<td>0</td>
<td>1.35</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>4164</td>
<td>248</td>
<td>118</td>
<td>100</td>
<td>34</td>
<td>0.20</td>
</tr>
<tr>
<td>GIVENS</td>
<td>x</td>
<td>1330</td>
<td>4108</td>
<td>2968</td>
<td>2503</td>
<td>0</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>8829</td>
<td>650</td>
<td>1240</td>
<td>70</td>
<td>120</td>
<td>0.35</td>
</tr>
<tr>
<td>LU</td>
<td>x</td>
<td>222</td>
<td>3286</td>
<td>1277</td>
<td>6</td>
<td>0</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>4363</td>
<td>156</td>
<td>272</td>
<td>0</td>
<td>0</td>
<td>0.17</td>
</tr>
<tr>
<td>MATMUL</td>
<td>x</td>
<td>1320</td>
<td>11789</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>11147</td>
<td>1122</td>
<td>852</td>
<td>0</td>
<td>0</td>
<td>0.22</td>
</tr>
<tr>
<td>MRIQ</td>
<td>x</td>
<td>630</td>
<td>5053</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>4425</td>
<td>270</td>
<td>500</td>
<td>470</td>
<td>30</td>
<td>0.49</td>
</tr>
<tr>
<td>SHA1</td>
<td>x</td>
<td>5134</td>
<td>25721</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>24516</td>
<td>3445</td>
<td>2505</td>
<td>386</td>
<td>15</td>
<td>0.31</td>
</tr>
<tr>
<td>SOBEL</td>
<td>x</td>
<td>8820</td>
<td>32471</td>
<td>7468</td>
<td>6</td>
<td>0</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>35815</td>
<td>3920</td>
<td>3136</td>
<td>2744</td>
<td>3150</td>
<td>0.64</td>
</tr>
<tr>
<td>Σ</td>
<td>x</td>
<td>21927</td>
<td>114956</td>
<td>18512</td>
<td>6408</td>
<td>0</td>
<td>1.06</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>130432</td>
<td>14169</td>
<td>9218</td>
<td>4320</td>
<td>3664</td>
<td>0.37</td>
</tr>
</tbody>
</table>
calculations, the averages will even be lower. As a side note it is interesting to see that in a $6 \times 6$ toroidal Fabric, relative addresses with $y > 4$ occur although the maximum distance from any source to any destination cannot be more than three hops (refer to figure 5.10). The SL is not yet aware of the vertical toroidal links and is not utilizing them. Although the average hop count is already very low, it is expected to decrease further, once the SL supports all toroidal links. The additional links of Flattened Butterfly bypassing immediate neighbors will not be utilized very often. As described in chapter 7 every Input Port (IP)/Output Port (OP) pair contributes significantly to the area and power consumption. Taking the router with a radix of four as a basis to perform the power and area calculations, each IP/OP pair adds approximately 25% and 50% to the area and power consumption respectively. According to these numbers, a router with a low radix would be preferable. Thus the implementation of this kind of topology cannot be justified.

![Figure 5.10: The row and column wise distances of nodes as seen from the black one. In a toroidal $6 \times 6$ Fabric the distances for each dimension cannot exceed three hops.](image)

Although not contributing significantly to the average, for many applications it can be observed that there are exactly 6 flits be routed in x-direction with a hop count $\geq 2$ such as in LU or MATMUL. These are the configuration packets determining the topology that is desired. The packets do not affect the overall execution time as mentioned in section 5.4 on page 73. Since these packets are injected into the NoC well before any application can be launched, it is irrelevant, how many hops they travel and therefore how long it takes to forward them to their destinations.

### 5.5.6 Spidergon and Stargon Topology

Unfortunately neither the Spidergon nor the Stargon topology described in section 4.2.2 are suitable for REDEFINE in which specifications state that HyperOps can occupy a $5 \times 5$ section of the Fabric at maximum. Spidergon is too small with $4 \times n$ with $n > 0$ nodes and is hence not a solution.
Regarding the Stargon topology as shown in figure 4.4d on page 46, it can be observed that the NoC is divided into two layers of the same size (shaded in the figure) connected by a few extremely long links. The upper half can communicate with the lower half only through these links. While the largest HyperOp can be supported by this topology, a heavy penalty is expected to be paid, if the HyperOp is spread over the two halves. As an alternative, the NoC can be resized to such an extent that each half can accommodate the largest HyperOp. But this makes it necessary to create a $10 \times n$ with $n > 5$ topology. Any HyperOp that does not have any input values or whose input values coming from executing HyperOps have arrived, are ready to be launched. In case of an application in which the HyperOps are very small or have a high dependency among them and hence have to be launched sequentially, a larger NoC results in many unused resources. Hence a Stargon topology seems also to be unpractical for REDEFINE.

The main problem is that the logical structure of both topologies with the nodes at the boundaries of the network, does not match the two dimensional arrangement of nodes expected by the SL of REDEFINE. Hence these topologies are not further investigated.

5.6 Summary

To obtain numbers about the performance of RECONNECT, it has been integrated into a target architecture called REDEFINE which is briefly described. In this case the performance measurement was the Fabric Execution Time which stated how long an execution took in various configurations. One take away from this exercise is the finding that the topology of an NoC cannot be considered completely independent of the target architecture. The SL of REDEFINE is not aware of the used topology. Hence the expectations of providing a richer topology such as the hexagonal one, could not be met: The execution time of applications did not decrease significantly.

By analyzing the traffic pattern it can be concluded that the Flattened Butterfly as well as Spidergon and Stargon topologies are not an option for REDEFINE at the moment. The additional links of the Flattened Butterfly will be hardly utilized. In case of Spidergon and Stargon the support logic and the way applications are mapped onto the Fabric, make any potential benefits offered by these topologies, negligible.
Chapter 6

Artificial Traffic Generators

To obtain essential characteristics such as latency and throughput of RECONNECT, the Processing Elements (PES) have been replaced by artificial traffic generators inducing different kind of traffic patterns into the Network on Chip (NoC). In this chapter the test environments and used generators are described in details. Beside a traffic generator injecting traffic uniformly, also self-similar traffic traces have been generated. Apart from these different traffic patterns, the destination address of the flit plays a significant role.

Each traffic generator is divided into four categories:

1. **Normal**: The address range is only limited by the size of the NoC. In a $6 \times 6$ toroidal NoC such as the one in REDEFINE, a tuple of the relative address ranges from $-2$ to $3$ in each direction to reach any destination from any source.

2. **Close Neighbor**: All packets are destined to the immediate neighbors. This setting restricts the relative address to range from $-1$ to $1$ in each direction.

3. **Bit Complement**: The absolute address of the destination is calculated by bit complementing each coordinate of the source address. This result in a star like pattern in which the flits are send diagonally across the network.

4. **Tornado**: In a $6 \times 6$ Fabric the relative destination tuples always equals $(2,2)$. Hence toroidal links are as any other link heavily included in this configuration. Formally each tuple of the address is calculated by

$$addr_{dest} = addr_{src} + ([n/2] - 1) \mod n \quad (6.1)$$

in which $n$ equals the number of nodes in a direction \([17]\).
6.1 Test Environment

The traffic generator determines at what time a flit with what address needs to be injected into the network. If the router does not accept the flit at the specified time, the flit needs to be queued. This queue needs to be empty and all flits must have arrived at their destinations, before the test can be concluded. While the flit waits in the queue for its turn, its latency count increases with every clock cycle.

Each test run has a warm-up phase in which flits are injected into the network, but do not contribute to the measurement of the overall performance. In the conducted tests a warm-up phase of 1000 clock cycles was used. After the test, which was executed for 1000000 cycles, a cool down phase started, in which the traffic generators were deactivated to allow the built up queues to be flushed into the NoC.

6.2 Uniform Traffic Pattern

A artificial generator for uniform traffic can randomly create a flit at any point in time. It depends on only a single threshold value stating how often and at what time a flit is injected. With that value, the offered load can be controlled. Figure 6.1 and figure 6.2 display the latency and throughput in term of maximum capacity of the network for various offered loads respectively.

The latency results obtained are comparable with the numbers presented in [34] and justify this approach and test environment.

As it can be observed, the throughput for close neighbor communication is significantly higher, if compared with the normal address generation method. This is expected, since the flits are ejected from the NoC at a very early stage. Interestingly the throughput does not always increase, if the offered load is intensified (refer to figures 6.2a and 6.2d). This is due to the so called weak fairness of the arbiters. Since they do not consider the age of the flit due to the reasons elaborated in 3.3.2.1, flits competing for common resources might stay an unequal amount of time in the Virtual Channels (VC). Comparatively old flits are not assigned a higher priority than freshly arriving flits. Consider the following example: A flit $A$ that already waits for a long time in a VC, but could not be forwarded due to unavailability of the receiving VC of the next router. A new flit $B$ arrives in another VC for the same destination like flit $A$ and at the same time the receiving VC becomes available. In case that the VC in which flit $A$ resides has been served last by the VC arbiter, it will grant the resources to flit $B$ instead, resulting that flit $A$ waits even longer.

By using the crossbar comprised of muxes and demuxes only, RECONNECT routers can be configured to require only one single clock cycle to forward flits. Without raising the level of complexity of the arbiters located in the critical path of that single cycled router, this problem of the weak fairness
cannot be addressed. Since the load on the network is very low for the applications mentioned earlier, so that congestions hardly occur, the decrease of throughput for loads beyond 30% to 40% is acceptable. In addition, since during the execution of the applications mostly very close neighborhood communication is encountered, the problem of unfair arbiters does not occur.
Figure 6.1: Latency for uniform traffic patterns and various address generation methods
Figure 6.1: Latency for uniform traffic patterns and various address generation methods
Figure 6.2: Throughput for uniform traffic patterns and various address generation methods
Figure 6.2: Throughput for uniform traffic patterns and various address generation methods
6.3 Self Similar Traffic Pattern

In real life situations it can be observed that the traffic is not uniformly randomly distributed. Instead it has a bursty character in which in a very short time period a comparatively large amount of data is transmitted, followed by duration in which no flit is generated at all [47]. Hence the traffic generator has been extended to accommodate these characteristics. The generator has two states: One in which the generator is deactivated and in off-state. In on-state the generator injects flits according to the configured injection rate. The generator remains in on-state and off-state for $t_{on}$ and $t_{off}$ cycles respectively as given below:

$$t_{on} = (1 - r)^{-\frac{1}{\alpha_{on}}}$$

$$t_{off} = (1 - r)^{-\frac{1}{\alpha_{off}}}$$

$r$ is a random variable uniformly distributed between 0 and 1. $\alpha_{on}$ and $\alpha_{off}$ have been fixed to 1.9 and 1.25 respectively [7].

The latency and throughput results of self similar traffic patterns are depicted in the figures 6.3 and 6.4 respectively. These results are not comparable with the ones given in section 6.2. While in uniform traffic pattern the traffic generator stays always in an active state and hence continuously injects flits according to the injection rate, it is not the case in self similar traffic generators. There are periods in which the generator is deactivated, hence by the end of the simulation the number of injected flits is lower. Another testing method would be to adapt the injection rate. The longer the generator is deactivated, the higher must be the injection rate while it is active.

It would be useful to validate this approach against results in published literature such as [27]. However the exact testing environments are not described, hence making it difficult to compare.
Figure 6.3: Latency for self similar traffic patterns and various address generation methods
Figure 6.3: Latency for self similar traffic patterns and various address generation methods
Figure 6.4: Throughput for self similar traffic patterns and various address generation methods
Figure 6.4: Throughput for self similar traffic patterns and various address generation methods
6.4 Summary

In this chapter the traffic generators and the testing environment have been described in detail. It could be observed that the weak fairness of the arbitration stages causes a drop in throughput after reaching a peak at an offered load of 30%. As expected the throughput increases, the richer the network is. Thus the hexagonal topology shows the highest performance followed by the mesh topology. The honeycomb topology gives the worst results.

The obtained results are comparable with the ones found in literature such as [34]. However in some publications the tabulated performances are much better. Unfortunately in many cases the test environments are not described detailed enough to allow comparisons or to draw any conclusions. For instance another method apart from the one described in this chapter, would be that the flit is dropped, if the router does not accept it at the specified time due to congestions. Hence the average latency is not increased by waiting flits. In the end the number of flits that are injected into the network, is lesser compared to the number of flits injected by using a queue, although the injection rate is the same from the point of view of a generator.

The take away from the exercise in this chapter is that, if a paper is published promising to improve the performance, the ideas need to be implemented and run against the own test traffic patterns before conclusions can be drawn.
Chapter 7

Synthesis Results

A router including its assembly unit has been synthesized by using Design Vision and FSD0A_A 90nm standard cell library for UMC’s 90nm logic SP-RVT (Low-K) for typical cases for different flit sizes of interest. As mentioned before only sizes which cause packets to be segmented into different flits and have the minimum of unused bit are analyzed (refer to figure 5.4) to obtain power and area details.

For NoC routers realistic power numbers are difficult to acquire. One reason is the availability of various methods of measurement as described below. Another reason is that from a global point of view the routers of the network are in different states and busy levels. If the CRC application (section 5.5.1) is considered as an example, it can be observed that in a Fabric of 6 × 6 nodes, only 3, at maximum 4 nodes depending on the configured topology, are used. The other nodes and their routers remain idle, not consuming any switching power. In this section a single router has been analyzed while the activity factor has been fixed to 0.5 which reflects a switching of every clock cycle and is therefore the worst power consumption.

Mainly two different measurement methods exist, if power and area numbers need to be acquired:

1. The router including the Assembly Unit (AU) is synthesized, but the ports are left unconnected. The synthesis tool can induce the activity factor only on those ports which are exported through interfaces. However for estimations about the area consumption, these ports are optimized to the extent that they become non-existent.

To prevent this undesired optimization the Input Ports (IPs) and Output Ports (OPs) can be connected to buffers. However before a flit can be stored in the VC, the previously calculated VC is read from the head flit and after which the flit is stored in the appropriate VC. If the ports are left unconnected, these logical blocks do not count for the critical path calculation which is the longest path between two buffers and hence determines at which clock frequencies a circuit is able to operate.
Neglecting this issue results in wrong estimates about the maximum frequency.

2. To avoid the removal of the unconnected IP and OP, the OPs are connected to the IP in the second measurement method. However, since the IP and OP interfaces are no longer exposed, the tool cannot induce an activity factor into these ports. The only interface that is left unconnected is the one towards the sink/generator. While this gives certainly lower power numbers compared to the first method described above, the area numbers and the estimate about the maximum clock frequency are more realistic, since the complete length of the path a flit has to travel from its buffer to the receiving buffer is considered.

7.1 Synthesis Tool Parameters

The following numbers have been obtained by the Synopsis tool “Design Vision” and are an estimation. Accurate power and area numbers can only obtained after post synthesis which is not done in this work, since REDEFINE itself needs to mature, before this step can be taken.

The following parameters are set, if not mentioned otherwise:

- A clock frequency of 450MHz corresponding to the master clock given to REDEFINE.

- Each IP consists of four VC and each VC is a first-in-first-out (FIFO) buffer capable of holding four flits in total.

- The AU is included in the router.

- Relative addressing scheme so that the OP needs to update the header flit every hop.

- Activity factor of 50%.

- It is unlikely that the support for multiple topologies provided by the router, will be utilized once RECONNECT is integrated into a target architecture. Thus for each hexagonal, mesh and honeycomb topology a router configuration has been compiled into Verilog. Further several configurations with various flit sizes have also been created and compiled for each one of the topologies.
7.2 Area and Power Consumption of the Single Cycle Router

As can be observed in figure 7.1, the rate of power as well as the area consumption between the smallest (20 bit) and the largest flit (116 bit), is almost linear with the flit size for all topologies. At 115 bit the highest power consumption is observed, even more than at 116 bit of flit size. The reason is that at a flit size of 116 bit, the AU becomes obsolete, since packets do not have to be segmented into multiple flits any more, which in turn leads to a lower overall power consumption of the router.

In mesh topology the router has an additional port activated when compared to honeycomb. From figure 7.1a an estimate can be derived of how much power each added and active port consumes. Depending on the flit size a router integrated in a mesh topology consumes approximately 27% to 30% more power when compared with a router in a honeycomb topology. Hexagonal routers need around 48% more power than routers in mesh. This number is doubled, if the hexagonal router is compared with one that is integrated in a honeycomb topology. As a rule of thumb, it can be said, that each IP/OP pair added to the honeycomb router, increases the power consumption by approximately 25%.

It is obvious that with additional ports, also the area required by the router increases (refer to figure 7.1b). Like the power consumption, the area increases linearly with the flit size. A router integrated in a mesh topology requires between 26% and 30% more area than a router in a honeycomb topology depending on the flit size, whereas a router in hexagonal topologies approximately 50% and 91% more than a router in a mesh and honeycomb respectively.

The linear increase of power and area with the increase of the flit size suggests that the router does hardly have control logic which depends on the flit size. Roughly speaking the control logic comprises of:

- Routing algorithm,
- VC arbitration within the IP,
- OP arbitration for competing IPs,
- and relative address update mechanism.

The main purpose of a router is to transfer data from one port to another one. Hence it mainly consists of buffers and buses which hugely depend on the size of a flit. For instance the VC have registers which allow to store several flits. If the flit size is changed, so are the registers resulting in a removal or addition of registers. The same applies to all buses that a flit encounters while traversing through the modules of the router. Hence the
Figure 7.1: Power and area consumption of a single router for various flit sizes. The drop at 116 bit is caused by the removal of the AU whose functionality of segmenting packets into multiple flits is not required at these sizes anymore.
gate count is approximately linear with the flit size and since the gate count is proportional to the area consumption, the area is also expected to change linearly.

7.3 VC Depth

During the execution of applications described in section 5.5 it was observed that the VCs hardly reach a state in which they do not accept any data anymore. Even congestions and competitions of several IPs for a single OP as the ones described in section 5.5.3 are resolved quickly. Hence it can be assumed that the VCs can be reduced to a depth in which they can hold the largest packet completely comprising of multiple flits which is:

$$VC_{\text{Depth}} = \left\lceil \frac{L}{B} \right\rceil$$ (7.1)

with $B$ being the flit size and $L$ representing the maximum packet size, which is currently 116 bits in case of REDEFINE. Figure 7.2 depicts the power and area consumption with the newly adapted VC depths. As it can be observed, a reduction of flit size does not necessarily result in a lower area and power consumption. If, for instance, the flit size is reduced from 76 to 57 bits, the largest packet needs 3 flits in total to accommodate its payload. Hence the depth of the VC is increased by one flit. In total it is possible to store $3 \times 57 = 171$ bits in total for each VC whereas it was only $2 \times 76 = 152$ bits in the previous case. At flit sizes of 116 bits all VC depths are reduced to one since the flit size corresponds to the packet size exactly.

Compared to the power and area ratings discussed in section 7 the increase is moderate, when the flit size is enlarged. In fact a router supporting a flit size of 76 bits has nearly the same power and area consumption when compared with a flit size of 43 bits. With this finding and in view of the Fabric Execution Time (refer to section 5.5 for various applications), it seems unreasonable to use routers supporting only the smaller flit sizes causing a drastic increases of the execution time. Regarding the chosen flit sizes: As mentioned in section 5.5.1 only those flit sizes are investigated, at which the number of flits for a given set of packets differs and the number of unused flits is minimal (refer to figure 5.4).
Figure 7.2: Power and area consumption, if the VC has to hold one packet eventually comprising of multiple flits.
7.4 Area and Power Consumption of the Pipelined Routers

As mentioned in section 3.6 the currently provided throughput is not sufficient for various application such as in H.264 decoding. At a clock speed of 450MHz and a flit size of 116 bits, the gross throughput per link peaks at 6.07GBps including information for control such as addressing and VC calculations. Since the wiring complexity is high, a more realistic bus width between the routers will be 32 or 64 bit after deduction of the control field to match the available payload field the word length of the attached processing cores. In addition the link throughput does not consider congestions, if e.g. multiple modules of the H.264 access the same shared memory provided by the Load/Store Unit (LSU). Based on the results of the artificial traffic generators (refer to section 6) and under the assumption that a mesh topology is used for implementing a H.264 decoder, the network has to provide four times (see below) the capacity to forward the flits in time which is limited by the frame rate of the video stream.

If a flit carries 4 Bytes payload and if a bandwidth between memory and processing core needs to be 637MBps, the router needs to forward 166,985,728 flits per second. Due to the saturation point of the mesh topology at approximately 25% of network capacity (refer to figures 6.1a and 6.2a), the router should be able to process 166,985,728 × 4 = 667,942,912 flits. Even if the router operates at the maximum frequency of 550MHz, it can forward only 550,000,000 flits on a single link per second at maximum. Since this peak performance is a theoretical value and it is most unlikely to be reached in reality, it is obvious that this throughput cannot be accomplished with the current implementation, justifying engineering efforts for a pipelined version.

As mentioned earlier in section 3.6.1 there are two pipelined versions available:

1. One pipeline stage after IP/OP arbitration (version 1).

2. In version 2 the crossbar establishing the connectivity between each IP to all OPs of a router, is replaced by a butterfly crossbar providing three pipeline stages in total (refer to figure 3.8 on page 35). This implementation does not require an IP/OP arbitration.

Whereas in the first version the area consumption and power dissipation is almost equivalent with the results that are presented for the single cycle router, since only 24 (honeycomb topology) to 49 (hexagonal topology) flip flops are needed additionally to retain the states of the arbiters, a tremendous increase in both area and power can be observed in the router using the pipelined butterfly crossbar. This is due to the drastic increase of required
storage capacity in the pipeline stages. Figures 7.3 and 7.3b depict the area and power consumption of the router respectively. To allow a fair comparison, all settings such as activity factor and frequency, remain the same.
Figure 7.3: Power and area consumption of a single pipelined router using a butterfly as a crossbar for various flit sizes. The drop at 116 bit is caused by the removal of the AU whose functionality of segmenting packets into multiple flits is not required at these sizes anymore.
7.5 Maximum Clock Frequency

The maximum frequency is calculated from the maximum path length between two buffers including their hold and setup time. The impact of the flit size on this frequency is negligible, since the control logic such as arbitration and path calculation, is not affected by the flit size. For each bit that is added to the flit size, another layer transmitting this additional bit, is necessary in the crossbar and at the interfaces of the router IP and OP. Since this additional layer is in parallel with the already existing ones, the critical path does not change.

However, a change in the router radix has an impact on the complexity of the muxes and demuxes inside the crossbar and the routing algorithm. But again, the arbitration steps, routing algorithm etc. remain the same. The impact of the radix is listed in table 7.1.

The pipelined router featuring the butterfly crossbar (version 2) could be clocked fastest. The mesh and hexagonal topology require the butterfly replacing the original crossbar comprising of muxes and demuxes, to consists of 3 stages in total. Although the honeycomb topology which has a radix of four, requires only a two staged butterfly, this optimization has not been implemented yet. A change in the number of supported IP/OP pairs will only effect the latency of the flit, but not the frequency. Together with the finding that a change of flit sizes will not result in an elongated critical path, only one configuration needs to be synthesized. The maximum frequency for this router with a flit size of 116 bit is 1.4GHz.

Theoretically at these speeds the bandwidth requirement of the H.264 decoder can be matched. However in reality it should be further investigated, once REDEFINE is equipped with relatively large processing elements.
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Single Cycle</th>
<th>Pipelined (ver. 1)</th>
<th>Pipelined (ver. 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>30</td>
<td>87</td>
</tr>
<tr>
<td>Honeycomb</td>
<td>600</td>
<td>575</td>
<td>575</td>
</tr>
<tr>
<td>Mesh</td>
<td>550</td>
<td>575</td>
<td>525</td>
</tr>
<tr>
<td>Hexagonal</td>
<td>475</td>
<td>475</td>
<td>425</td>
</tr>
</tbody>
</table>

Table 7.1: Maximum clock frequency of the router reported by the design compiler in pre-synthesis phase.
7.6 Summary

In this chapter the area and power numbers for various router, topology and flit settings have been tabulated. With these numbers the system designer is able to estimate the cost of important feature configurations. Since the area and power consumption increases almost linearly, even area and power estimates can be derived for flit sizes beyond the ones that are mentioned above. In addition to the single cycle routers the pipelined routers have been synthesized leaving the settings of the synthesis tool untouched to allow a comparison. As expected the tremendous increase of buffer requirements for each pipeline stage is reflected by the comparatively higher power and area figures. At these costs the expected maximum clock frequency is almost tripled from 550MHz that a single cycle router runs at, to 1.4GHz.

In this chapter a theoretical approach has been described justifying the implementation of the pipelined routers. They are necessary to meet the bandwidth requirements of high level and data intense applications found in the H.264 domain as an example.
Chapter 8

Conclusion

In this work a highly configurable router implementation called RECONNECT has been presented. It has been realized in Bluespec System Verilog (BSV) and it provides configuration and module flexibility. Configuration flexibility is realized by the declaration of several defines which alter the functionality or behavior of the routers such as the existence of Virtual Channels (VCs) or their depth. Module flexibility is provided by multiple ready made modules which can replace the integrated ones. An example are the Input Ports (IP) providing either VCs or Assembly Units (AU) or the option to replace the routing algorithms.

Usually the traffic pattern and throughput requirements are well known by analyzing the traffic traces of a particular application before the implementation of a Network on Chip (NoC) is considered. In a bottom-up approach the NoC is then tailored towards these requirements to achieve the maximum performance by including required functionality into the routers such as the number of IP/Output Port (OP) pairs to build a specific topology. In case of RECONNECT, a top-down approach has been taken. The domain in which RECONNECT is suppose to serve, is unknown. Hence the router architecture is application agnostic and kept as generic and modularized as possible. Similar to a module construction system the system designer is able to help himself out of a pool of already implemented modules to plug in the functionality that is required. This work also gives a short introduction into BSV to support developers to implement new functionality into the router. To do so some caveats and lessons learned during the RECONNECT realization, has been pointed out such as the arbitration stages and their fairness.

With all that in mind, RECONNECT becomes a powerful toolbox resulting in an NoC that can be easily tailored towards future requirements by giving the system designer the freedom of choice and a fast and easy way to synthesize an NoC that best suits the needs.

Another contribution of this work is the development of a routing algorithm for the honeycomb topology. In literature the combs of this topology
are usually complete which makes it easier to develop a routing algorithm. However if the honeycomb topology is imposed on a mesh, combs appear that are half in size (refer to figure 4.10 on page 53). Then the honeycomb topology becomes similar to the one of a mesh with each alternative vertical link missing and existing algorithms found in literature could not be used. Its availability ensures that the system designer is able to switch from mesh to a honeycomb topology without much engineering efforts i.e. adaptation of the Fabric and the supporting logic of the target architecture to the new topology. Extensively it has been described, how the routing algorithm for the honeycomb topology has been developed. In case of the necessity to integrate RECONNECT in a topology that has not been considered in this work, the designer has an instruction manual at hand to develop routing algorithms for these topologies, if they have not been discussed earlier in published work. Some of these algorithms use 3D tupled addresses that might not be supported by the target architecture. Hence an address conversion unit able to convert 2D addresses into 3D ones, has been developed. Since the support logic of the target architecture might not be possible to adapt to the 3D addressing scheme of the Fabric, this unit makes any changes in other modules of the target architecture superfluous.

To estimate the costs of some of the important configuration parameters of RECONNECT such as flit size, VC depth and number of IP/OP pairs, a router has been compiled from BSV into Verilog and analyzed by the synthesis tool Design Vision for 90nm technology. It has been discovered that area and power numbers are increasing linearly with the flit size. Due to the buffers located in each IP, adding an IP/OP pair to the router increases its area and power consumption by 25% (with a router of radix of 4 as basis). With these statistics the system designer is able to estimate the power and area characteristics of an NoC even before implementing it saving valuable development time.

8.1 Conclusion regarding REDEFINE

As a proof of concept this implementation has been integrated into REDEFINE to establish connections among the Compute Elements (CEs). Due to the router configurability, it could be investigated, what NoC properties such as topology or flit sizes, are most suitable for certain applications running on REDEFINE and how these NoC characteristics affect the execution time. Hence the first question that needs to be answered is: Based on the results in the previous chapter, what topology is the best for REDEFINE?

As discussed in section 5.5 it is obvious that a topology with the lowest degree with the smallest flit size is favorable in terms of power and area consumption. However it needs to be kept in mind that REDEFINE is a generic data flow architecture, hence an assumption that the domain of
operation is known, cannot be made. In future it is inevitable that larger Hyper Operations (HyperOps) are executed on the Fabric whose nodes will probably continue to be arranged in a two dimensional grid. In that case the problem with a honeycomb topology are the missing alternating vertical links. If a partial HyperOp (pHyperOp) needs to communicate with a node that is located two hops into the northern or southern direction, the hop count is actually four hops in honeycomb, which is double the amount compared to mesh.

In the comparatively small applications such as CRC or LU (refer to section 5.5.1 and A respectively) that are available for testing during this work, it is evident that the topology choice impacts the Fabric Execution Time only marginally due to the small HyperOp size. On the other hand application with large HyperOps such as AES and SOBEL (section 5.5.2 and 5.5.3 respectively) benefit from a richer network.

Another disadvantage that speaks against the use of the honeycomb topology is the dependency on four VCs which are implemented as first-in-first-out (FIFO) buffers that are expensive in terms of area and power. In addition due to the arrangement of the nodes in a 2D grid that is not the “natural” environment for a honeycomb topology (see figure 4.1a on page 43), the developed routing algorithm depends on being integrated into a toroidal topology with always an even number of rows and columns. Rearranging the nodes in favor of honeycomb causes huge changes in the REDEFINE Support Logic (SL) and its compiler, since especially the Resource Binder (RB) depends highly on a 2D grid.

On the other hand in mesh topologies the usage of VCs can be avoided completely, if the topology is not toroidal. The advantages of the short paths that is given in toroidal topologies can be covered by rearranging the Access Router (AR) into the center column and by adapting the Fabric size to be able to hold two largest HyperOps on either side of this center column. In addition although the degree of the node in mesh is higher, it is expected to consume less area and power when compared with the VC dependent honeycomb topology.

The second topology that needs to be compared with mesh is the hexagonal one. As in honeycomb, the hexagonal topology is not integrated into its natural environment making it dependent on toroidal links and hence on VCs. However since the degree is higher than in mesh and honeycomb, the impact of two additional IP/OP pairs becomes evident, if power and area consumption is inspected. With such a high price to be paid, an application developer expects the same amount of benefit in terms of latency and throughput. Unfortunately REDEFINE is not able to utilize these additional links as desired. Only during launching of HyperOps and Load/Store Unit (LSU) and Inter HyperOp Data Forwarder (IHDF) communication, these links are used. Also here it becomes mandatory to make the RB aware of the topology in the Fabric. In addition the degree of inter HyperOp communication needs to be
raised, before any improvements in hexagonal topologies can be expected.

In summary: Since the SL of REDEFINE is already fixed to a two dimensional topology, a mesh topology provides the same arrangement and, as elaborated above, will be the better option.

The second question that needs to be answered is: What shall be the flit size? From the application point of few, more wires results in a higher bandwidth, but on the other hand an increasing flit size raises the wiring complexity. Currently REDEFINE is not in a stage in which it is run through the steps of post synthesis, layout and routing. In addition the domain of REDEFINE is generic making it difficult to estimate the bandwidth requirements of the NoC. At present the applications do not impose any restrictions. For instance in case of AES Decryption which takes up to 26.395 clock cycles to finish the execution. Since it is not a stream, the amount of time in which the decryption needs to be completed, is not specified. Due to the lack of enforcement of this specific requirement, the system designer has a free choice: In power critical systems, a very small flit size can be chosen causing an application to take more time in the order of several magnitudes. If the applications are to run faster, the designer can simply increase the flit size till specifications are met. Once compiler, CE and SL support for streaming applications is integrated in REDEFINE, the requirements will become clearer and then this question needs to be reexamined.

8.2 Future Work

The existing RECONNECT router can be further enhanced by allowing heterogeneous topologies meaning a topology in which e.g. the bandwidth among the routers is not a constant. In case of large functional blocks the bandwidth requirements are often different. Providing this large bandwidths throughout the NoC result in a high area and power consumption. One way to tackle this problem is to allow the system designer to specifically configure the bandwidth requirement for each link by e.g. a parameter given to the router while instantiating it.

The possibility of virtual circuits should also be investigated. Especially in streaming applications the traffic pattern does not change. Instead of the necessity that every flit needs to negotiate its way through the NoC towards its destination, dedicated paths similar to telephone lines, so called virtual circuits, are established, maintained and terminated in the end. However a rudimentary virtual circuit or also called guaranteed bandwidth can be maintained by injecting a header flit followed by payload flits. As long as no tail flit is injected, the path from a specific source to a defined destination exists. However the problem are the shared paths between the router. For instance if the destination node in the NoC requires a flit coming from a different source, this flit cannot be delivered as long as the virtual circuit is
established. Thus it needs to be ensured that there are no overlapping paths while the application executes. Especially in applications processing data streams this cannot be always guaranteed requiring further research in this field.

The target architecture might run at different clock frequencies which are comparatively low. In that case the NoC performance can be improved, if synchronizers are placed at the injection ports into the routers. This has not yet been implemented.
Appendix A

Fabric Execution Time

This section contains the remaining figures depicting the Fabric Execution Time of various figures and the effects of flit sizes and topologies. The algorithms are described briefly in [57, 1].

Figure A.1: AES Encryption algorithm
Figure A.2: ECPA

Figure A.3: ECPD
Figure A.4: GIVENS algorithm

Figure A.5: Lower upper matrix factorization
Figure A.6: Matrix multiplication

Figure A.7: MRIQ algorithm
Fabric Execution Time [10^5 Cycles]

Flit Sizes [Bits]

Hexagonal
Mesh
Honeycomb

Figure A.8: Secure Hash Algorithm (SHA) version 1
Appendix B

List of Files

The functionality of the router has been implemented in several files. In chapter 3 in which the architecture of the router is described, the file names of several modules are mentioned in the title of the sections. To shorten the names, regular expressions [22] have been used. This appendix tabulates the filenames along with a short description for each file.

The following files provide the functionality for the routers and are located in the directory ./fabric/NoCRouterSingleCycle/ and ./fabric/NoCRouterXBar/:

Arbiter([0-9]+).bsv (section 3.3.2.1): These files contain the Matrix Arbiters. The number of required request lines is reflected by the number in the filename.

TbArbiter([0-9]+).bsv: Test benches for the arbiters mentioned above.

Generator.bsv, Sink.bsv: These files contain a simple generator and sink to test the functionality of the routers.

GeneratorTestAssemblyUnit.bsv: A generator to test the connectivity between a router and the Assembly Unit (AU).

InputPort(XBar)?.bsv (section 3.3): This file contains the functionality of the Input Port (IP). If the butterfly crossbar is used to implement pipeline routers, then InputPortXBar.bsv is used.

TbInputPort.bsv: Test benches for the IP.

NoCRouter.bsv (section 3): This file contains the top level module instantiating and connecting other modules such as IPs, Output Ports (OPs), and arbiters.

TbNoCRouter1.bsv: This file instantiates one router including generators and sinks according to the number of IPs and OPs. It allows to test the
functionality of a single router. As an alternative the test bench can be altered to establish connections from the [IP] that are usually connected to the neighboring routers, to the [OP] of the same router.

**OutputPort.bsv (section 3.5):** The [OP] of the router containing the address update logic. It is the last module a flit passes through before it is forwarded to a neighboring router.

**RoutingCommon.bsv, RoutingConvert.bsv:** These files are used to provide functions that are the same for all routing algorithms. Currently it contains only one function: It converts the representation of the dateline or the [OP] into a format that is understood by the routing algorithm.

**Routing(.*).bsv (section 4.2):** These files contain the routing algorithm for various topologies. For instance in **RoutingMesh.bsv** the algorithms for the mesh topology are implemented.

**(Tb)?XBar.bsv (section 3.6.1):** These files contain the routing algorithm for various topologies. For instance in **RoutingMesh.bsv** the algorithms for the mesh topology are implemented.

Connections among the routers are established by files located in `./fabric/Fabric/`:

**generators/ (section 6):** This directory contains the artificial traffic generators written in C++. There are two types of generators:

1. **generator(.*).(c|h)pp:** These generators create new traffic pattern during run time. Every time the simulation is executes, the traffic patterns differ.

2. **offlineGenerators/:** In this directory traffic generators are located which are executed before the simulation runs. Text files are created stating at which point in time a source has to send a packet to a destination. The advantage of using offline generators is the reconfigurability of traffic patterns without recompiling the whole Fabric and its test bench.

Since C++ is not a hardware description language, a compilation of the Fabric with artificial traffic generators into Verilog is not possible. Only an executable for i386 and x86_64 PCs can be generated.

**TbFabricGenerators.bsv:** Test benches for the generators described above.

**Fabric(Generators)?_IFC.bsv:** These files provide the interfaces of the Fabric to the modules of the supporting logic of the target architecture. These interfaces are connected to the [IP] of the Access Routers [AR].
Fabric.bsv: This file establishes the connectivity among the routers and exposes the interfaces of the AR. This file does not exist originally, but it needs to be generated by a Python script called multiTopo.py. The output of this script is a Bluespec System Verilog (BSV) source file providing the said functionality statically. Unfortunately the compilation of the Fabric takes much more time, if the Fabric is generated dynamically by a BSV file only.

GeneratorFlitCProgram.bsv: To execute C++ programs, an interface between the BSV and the C++ code is required. Its task is to translate the variable types from C such as char and int, into a type that is understood by BSV. It is clock aware and ensures that the functions of the C program are executed accordingly. This could be a function that needs to run every clock cycle to check the file of the artificial traffic generators, if a new flit needs to be injected into the Fabric.

SinkFlitCProgram.bsv: This file collects arrived flits to allow a statistical analysis about the latency and throughput.

Files which serve a common purpose and which the supporting logic of the target architecture needs to have access to, are located in ./fabric/common/:

AssemblyUnit.bsv (section 3.3.1): The AU is encapsulated in this file.

TbAssemblyUnit.bsv: A test bench for the AU.

TbAssemblyUnitToObtain MAX_PACKETS.bsv: To get an overview about the segmentation of packets into multiple flits, this test bench can be used. It runs through all available packet types and neatly prints out the sizes of the flits and how many flits are required for all existing packets.

FlitStructure.bsv: All packet types that are sent through the Network on Chip (NoC), including their bit field definitions, are summarized in this file.

FlitStructureDefaultValue.bsv: In some cases it is desired that BSV defines standard values for the packets. These values are stored in this file.

FlitStructureFShow.bsv: In BSV the $display command prints a message. However in case of a union or structure, it would be better, if the values of each entry/bit field are printed separately to ease debugging and for a better overview. This file informs the BSV compiler, how structures and unions are displayed.

FlitStructurePackUnpack.bsv: This file defines, how the unions and structures of the packet types are converted into a bit field and vice versa.
**FlitStructureSizes.bsv:** To segment the packets into multiple flits, it needs to be calculated, how many flits are required for each packet.

**LIFO.bsv:** The LIFO module provides a generic module for a buffer following Last-In-First-Out semantics. Currently it is only used by the AU.

**TopologyConfiguration.bsv:** Since the router can be configured to support multiple topologies at run time, it needs to know, what these topologies are. All topologies mentioned in this file, are supported.
References


REFERENCES


REFERENCES


REFERENCES


