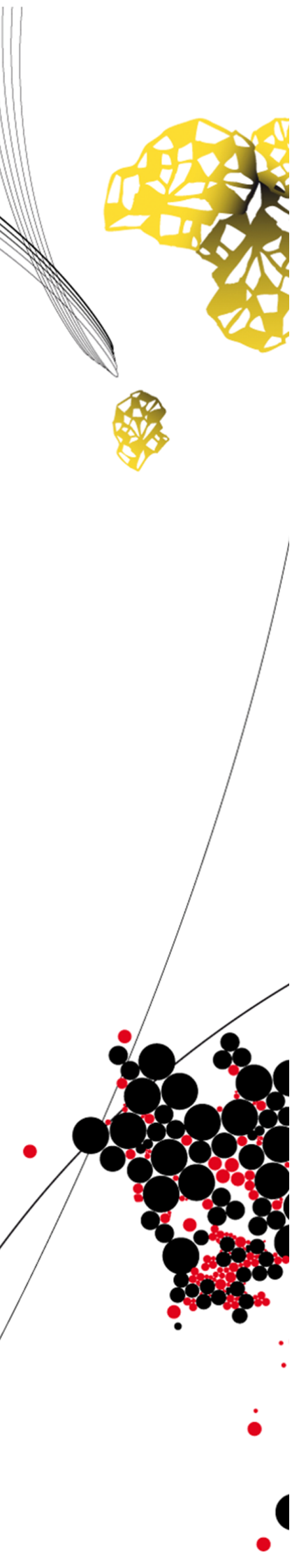


# UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,**
**Mathematics & Computer Science**

**Master's Thesis**

# Mapping dataflow over multiple FPGAs in Clash

**Sander (D.J.) Bremmer**
**November 2020**

**Supervisors:**
Prof.Dr.Ing. D.M. Ziener
Ir. H.H. Folmer
Ir. J. Scholten
Dr.Ir. J. Kuper

CAES Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

# Preface

Right now, you are reading my master's thesis, in which I tell you how I map dataflow graphs on multiple FPGAs. This thesis is written for all those interested and familiar with the designing FPGAs in a Clash. And for those looking for a structure in which FPGAs communicate with each other in a deterministic way.
This master's thesis was written for the Embedded System study programme at the University of Twente and was carried out at the CAES-group. The reason for me to choose the CAES-group were the courses Embedded computer Architectures 1 and 2.
After some personal setbacks, which delayed the completion of the thesis, I would like to thank my parents, committee members and fellow students for all the help and time they have given me to advise and guide me to complete the thesis. Firstly I would like to thank my parents, who gave me the time and space to study. Secondly, I would like to thank Hendrik Folmer, who was my daily supervisor for his support, motivation and critical feedback. The same goes for Jan Kuper, my weekly supervisor, with whom I had a weekly meeting together with Hendrik Folmer, Oguz Meteer, and other students. I would also like to thank Daniel Ziener for being my overall supervisor and offering a graduation place. Last but not least, I would like to thank all my fellow students of the CAES-group for their constructive and moral support.

Sander (D.J.) Bremmer
Vriezenveen, 28 October 2020

# Summary

Modern cars have many parts that work, partially, electronically. Miscommunication between the different parts can result in accidents. Therefore, the communication time between the various electronic components is critical. Those electronic components in a car we represent as a Field Programmable Gate Array (FPGA). An FPGA is a device with flexible hardware, on which computationally demanding applications, are more and more implemented these days. FPGA designs also grow and no longer fit on one FPGA. Spreading tasks over multiple FPGAs can be beneficial for implementations. The distribution of tasks across multiple FPGAs makes that FPGAs need to interact to solve a problem. While working, they communicate, this communication time is critical and is complex. To model this interaction and critical communication time, we use dataflow graphs. Dataflow is a suitable and well-known communication model to model time and data dependency.

The goal of this thesis is to map a dataflow graph over multiple FPGAs, where each node of the dataflow graph is assigned his own FPGA. These FPGAs must then be interconnected. We can use the same structure for the connection as the dataflow graph. Still, we want to support different dataflow graphs. So, we started looking for a suitable communication structure, called the hardware topology. After selecting a topology, we make a design that fits the topology. We implement this in CAES Language for Synchronous Hardware (Clash). The implementation results in communication time between the different FPGAs. We, therefore, want to know how we can display and calculate this communication time.

As a starting point, we choose the ring topology, with the Nebula ring interconnect. The nebula-ring interconnect, is an all to all interconnect. All FPGAs in the ring can send data to each other via the ring network. In this ring, there are slots, where every FPGA has its own slot. Those slots shift around. This shifting means that every FPGA sees its own slot every once in a while. An FPGA can use its own slot to inject data into the ring. The FPGA for which the data is intended extracts that data from the ring. Which FPGA is a source for the data and which is the destination, is modelled by the dataflow graph. After choosing the ring hardware topology,

we know how the FPGAs are set up and how they communicate, we design and implement this in Clash. In the ring topology, every FPGA has the same structure. Because of this one structure. We design a model for one FPGA, which we can then apply to the other FPGAs. On one FPGA, there will be several elements that we connect. An element is an actor representing an actor of the dataflow graph. The actor is connected to an output memory buffer, which serves as a waiting place for the messages that enter the ring. The actor is also connected to an input memory buffer, which is a waiting place for the messages coming from the ring so that messages from different edges can be consumed at the same time. Both memory buffers are connected to a router. The router chooses, when the own slot arrives, which message from the output buffer is injected into the ring. It also routes the messages coming from the ring into the input memory buffer. If a message is not destined for the FPGA, the router sends them further over the ring. The routers, and nebula slots, of different FPGAs, are interconnected, in the ring topology. This ring structure we simulated in Clash.

The implemented hardware architecture we model as a resulting dataflow graph, of which we've charted the communication path. The communication path is the path a message travels from source to destination over the ring. These communication paths are added to the initial dataflow graph as identity actors. An identity actor is added to each edge of the initial dataflow graph. On the resulting dataflow graph, the user can perform a post-analysis. We can guarantee deterministic behaviour if we calculate the Worst-Case Execution Time (WCET) as firing time for the identity actors. For this purpose, we made two equations. With the first calculation, we are entirely dependent on the maximum number of messages in the output buffer. With the second calculation, we are dependent on the maximum number of messages on one output edge and the number of output edges. After a simulation in Clash, we see that the simulation results are the same as the calculations.

The conclusion is that we have chosen for a ring topology with the Nebula ring interconnect. Where each FPGA represents an actor of the initial dataflow graph. The user can then give an initial dataflow graph to our Clash implementation. This implementation is modelled as a resulting dataflow graph, in which additional identity actors are added. These actors represent the network communication time between two actors of the initial dataflow graph. For these actors, we can calculate the firing time. The designer can then analyse this model. We also compared the calculated results with the Clash simulation and found that they are the same.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| **ABS** | Anti-lock Braking System |
| **CAES** | Computer Architecture for Embedded Systems |
| **Clash** | CAES Language for Synchronous Hardware |
| **CLB** | Configurable Logic Block |
| **CSDF** | Cyclo-Static DataFlow |
| **DSE** | Design Space Exploration |
| **EDSL** | Embedded Domain Specific Language |
| **FIFO** | First In First Out |
| **FPGA** | Field Programmable Gate Array |
| **HDL** | Hardware Description Language |
| **HPC** | High-Performance Computing |
| **HSDF** | Homogeneous Synchronous DataFlow |
| **LCM** | Least Common Multiple |
| **NI** | Network Interface |
| **PCB** | Printed Circuit Board |
| **SDF** | Synchronous DataFlow |
| **VHDL** | VHSIC-HDL, Very High-Speed Integrated Circuit Hardware Description Language |
| **WCET** | Worst-Case Execution Time |

# Part I

# Introduction, Background and Related work

# Chapter 1

# Introduction

## 1.1 Context

Modern cars have many parts that work, partially, electronically, such as the airbag, Anti-lock Braking System (ABS), speed sensor, electronic brakes and clutch system. Self-driving vehicles have even more sensors and computing devices. Those sensors and computing devices communicate with one and other. Miscommunication between the different parts can result in accidents and must therefore not happen. For example, an airbag must deploy within a specified time, an automatic brake system must break before an accident occurs and the ABS must react to reduce the brake distance. Therefore, the communication time between the different electronic components is critical.
Cars are just one example of critical communication time between the electronic components in a vehicle, still, there are more time-critical systems, such as medical implants, e.g. heart-implants and peacemakers, electronic aeroplane control systems or other industrial process controllers.

An FPGA, see Figure 1.2 is a device with flexible hardware. This flexibility ensures that parts of the system can be changed without buying all-new processors. Implementations on FPGAs are more flexible and often work faster than on a CPU, because of parallel computation. Therefore, computationally demanding applications, such as neural networks, learning algorithms, High-Performance Computing (HPC) and real-time graphics processing are more and more implemented on FPGAs these days [3]–[7]. FPGA designs Also grow and, therefore, no longer fit on one FPGA. Because the programmable area of an FPGA is not unlimited, spreading tasks over multiple FPGAs can be beneficial for implementations. Such as the car example earlier, where different electronic part are located at different positions in a car. The distribution of tasks across multiple FPGAs makes that FPGAs need to work together to solve a problem. While working, they communicate. This commu-

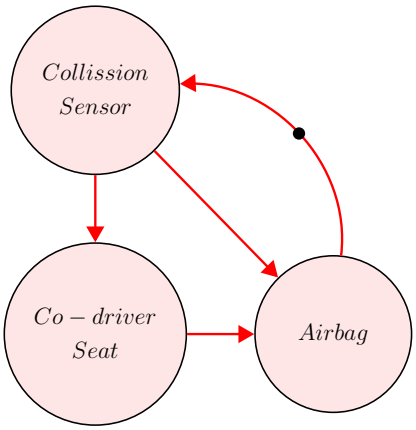



**Figure 1.1:** Airbag dataflow example.

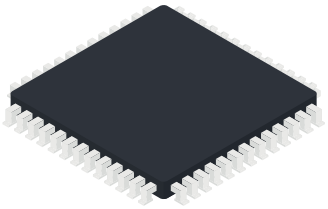


**Figure 1.2:** FPGA [1]

nication takes time and can become complex.

To model the critical communication time between FPGAs, we use dataflow graphs. Dataflow is a suitable and well-known communication model to model time and data dependency.

Dataflow graphs consist of three parts, namely actors, which perform tasks, represented as circles, edges between actors, represented as arrows, which describe the data dependency between different actors and tokens, which indicate the availability of data, represented as dots. Before an actor can start his task, there must be enough tokens on all incoming edges. The actor consumes these tokens and after a predetermined time produces those tokens on the outgoing edges.

In the model of Figure 1.1, we use an airbag as illustrative example[1]. In the dataflow graph, the airbag actor has issued a token to show that it is enabled. If there is a collision, the collision sensor produces tokens for the airbag actor and the co-driver seat actor. If the co-driver seat sensor detects that the seat is occupied, it produces a token for the airbag. The airbag will then see tokens on both edges after which it will deploy.

[1]The example is a fabricated example and not based on reality, but is used to indicate the importance of a dataflow graph.

## 1.2 Goal

The goal of this thesis is to map dataflow graphs on multiple FPGAs. Each actor of the dataflow graph is assigned his own FPGA. These FPGAs must somehow communicate with each other. For that, we need to know the hardware communication structure. This structure is the hardware topology we need to find first. After finding the architecture, we make a design and implement this in CAES Language for Synchronous Hardware (Clash)[2]. The spreading over multiple FPGAs causes that there is communication time between them. We, therefore, want to analyse this. As a result, we make a new model to calculate and simulate the communication time.

So we are looking for an interface where a user provides an initial dataflow graph, after which it is mapped over multiple FPGAs. The user gets a resulting dataflow graph, with the modelled communication time. The user can do a post-analysis on this new model. The post-analysis allows the user/designer to see whether the resulting dataflowgraph model still meets the timing requirements. This dataflow mapping is interesting because it enables designs that do not fit on one FPGA to be spread over multiple FPGAs, still, creating a dataflow graph model that can be analysed.

In this thesis, we are not looking for which actor should be placed on which FPGA. It should work through random assignment, even though this may not be the optimal setup. Nor is it up to us to provide a dataflow graph and determine the functions of the actors. Also, we don't physically link hardware and take into account the propagation of clock signals, but we do want to find out what happens on one FPGA and what would happen if multiple FPGAs are linked.

[2] Clash is a functional hardware description language, well-known at the UT-CAES group. In Clash, we do simulations and transform High-level descriptions to low-level synthesisable VHSIC-HDL, Very High-Speed Integrated Circuit Hardware Description Language (VHDL), Verilog or SystemVerilog.

## 1.3 Research Questions

The following main -and- sub -questions answered to solve the problems.

*How do we design and analyse FPGA to FPGA communication in a defined topology, using dataflow graphs?*

*Which hardware communication infrastructure is suitable?*

*Given the topology, how do we map a dataflow graph onto multiple FPGAs?*

*Are there any dataflow graph constraints, if so, which ones?*

*How can we model the temporal behaviour of the design, analyse the communication and guarantee deterministic behaviour?*

*How do simulation results correspond to analysis results concerning timing?*

## 1.4 Approach and Outline

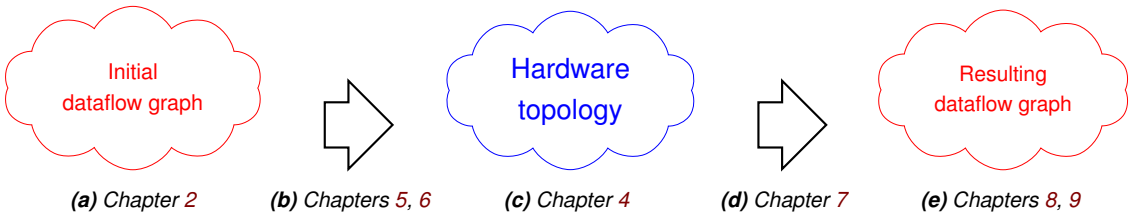



**Figure 1.3:** Designflow: Chapters 1, 2, 3 and 10

In Figure 1.3, we see the design approach of this project. The captions in the figures refer to the different parts to which it relates. Chapters 1, 2, 3 and 10 discusses the general project, and the captions in the subfigures refer to different chapters.

Part I In Chapter 2, we present the background information on various aspects used in this report. In the related work, chapter 3, we compare this project with the projects of others.

Part II In Chapter 4 we look for a hardware topology, we need this hardware topology because we want to know how FPGAs are connected, where the number of FPGAs is determined by the number of actors of the initial dataflow graph.
Now that we know how the FPGAs are connected we can, in Chapter 5 map an initial dataflow graph to the hardware topology, where tokens/data are sent through the communication channels of the topology, but where the model of the original dataflow graph is preserved. In Chapter 6, we will implement this in Clash. Still, we will make some implementation choices.

Part III The implemented communication architecture takes time, and we want to model this by adding actors to the initial dataflow graph. Therefore, we reconvert the implemented dataflow graph to a resulting dataflow graph in Chapter 7. Then, in Chapter 8, we look at what equations we can find to determine the firing time of the new actors. So that, in combination with the initial dataflow graph, we again have a complete dataflow to analyse. In Chapter 9, simulation results, we simulate the Clash implementation to see what time the new actors have so that we can compare this with the calculation of the equations.

Part IV In Chapter 10, conclusions are given. Finally, in Chapter 11, Future work, we discuss undiscussed and unimplemented subjects.

# Chapter 2

# Background

This chapter shows some background information of different aspects used in this thesis.

## 2.1 FPGA

FPGA is short for Field Programmable Gate Array and is a circuit of integrated programmable logic components, such as AND, OR, XOR, etc.

Two major FPGA manufacturers describe FPGAs as follows:

Intel [8] *"It is a semiconductor IC where a large majority of the electrical functionality inside the device can be changed; changed by the design engineer, changed during the Printed Circuit Board (PCB) assembly process, or even changed after the equipment has been shipped to customers out in the 'field'."*

Xilinx [9] *"FPGAs are semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLBs) connected via programmable interconnects."*

Integrated functions range from simple logic function to complex mathematical applications. Examples of these applications can be found in aerospace, automotive, medical applications, video processing, wired communication, etc. To design these circuits a Hardware Description Language (HDL) such as VHDL or Verilog is usually used.

## 2.2 Haskell and Clash

In Haskell evaluation of functions are similar to the calculation of mathematical functions. Haskell is a pure Functional programming language. This pure means that when a function is invoked, the result is the same every time, without side effects. Haskell is also lazy, that means that a function is only calculated when needed.
Clash is Functional HDL that borrows its syntax from Haskell and can best be described by Clash websites [10] or [11]: *"Clash is a functional hardware description language that borrows both its syntax and semantics from the functional programming language Haskell. It provides a common structural design approach to both combinational and synchronous sequential circuits. The Clash compiler transforms these high-level descriptions to low-level synthesisable VHDL, Verilog, or SystemVerilog."* more information, installation instructions or support can be found on their websites.
Next, an explanation of some used types, for an extensive description of the different aspects of Haskell and Clash see [12], [13], and [10].

### 2.2.1 Higher-Order Functions

A function that has a function as a parameter is a higher-order function. Here are some examples of higher-order functions used in this report explained using figures.

#### map

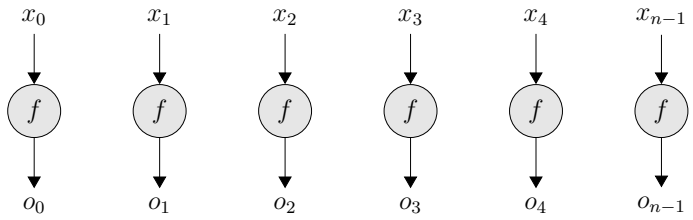



**Figure 2.1:** Higher order function: map

`map`, see Figure 2.1, is a higher-order function that applies a function `f` to every element of a list `xs`, to produce a list of outputs `os`. This is written as `os = map f xs` or by using the more abstract `fmap` function as `os = fmap f xs`, this can also be written as follows `os = f <$> xs`.

### zipwith

`zipWith`, see Figure 2.2, is like `map` a higher-order function. This function applies a function `f` to two arguments `xs` and `ys` to produce an output `os`. This is written as follows `os = zipWith f xs ys`.

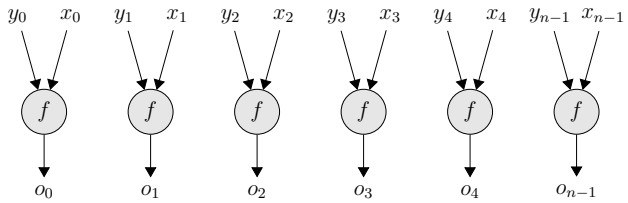



**Figure 2.2:** Higher order function: zipWith

### imap

The `imap` function, see Figure 2.3, is a higher-order function similar to the `zipWith` and `map` functions and is written as follows `os = imap f xs`. The difference with the zipWith function is that with the imap function, one of the arguments is filled in with a list of numbers representing the index; hence the 'i' in `imap`. This leaves only one argument `xs` that must be given to the imap. In that respect, it looks like the `map` function.

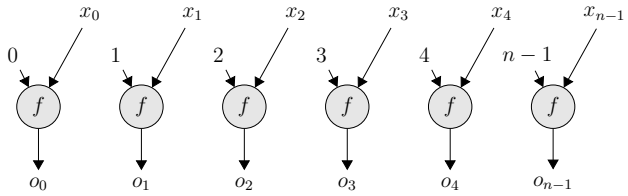



**Figure 2.3:** Higher order function: imap

### mapAccumR

The `mapAccumR`, see Figure 2.4 is a higher-order function that applies a function `f` to an argument `a` and a list `xs`. This, finally, results in a tuple, consisting of an argument `w` and a list `os`. The example from Figure 2.4 can be written as follows `mapAccumR f a xs = (w, os)`.

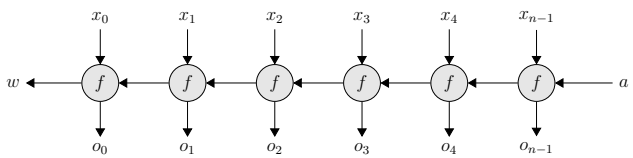



**Figure 2.4:** Higher order function: mapAccumR

### 2.2.2 Data Types

A data type is a specific type of data, such as integers and booleans. Each variable or expression is associated with a datatype. This datatype determines which values the variable or expression can assume.

#### Custom DataTypes

It is possible to create a custom data type. By using an Embedded Domain Specific Language (EDSL) within Haskell, we give value constructors a recognisable name. On this name, we can then pattern match. In Listing 1, we see an example of a data type. `Connect`, on line 1, is the constructor type. `a` and `b` are the type of parameters. `|`, on line 3, is a separation between value constructors, in this case `To`, on line 2 and `From` on line 3. `To` has two fields, with the variable type constructors `a` and `b`. `B` has as fields the type `String` and a variable type constructor `a`. The type of constructors `a` and `b` can be chosen when the type is used.

```
1 data Connect a b =
2 To   a b
3 | From String a
```

**Listing 1:** Data type example

```
1 data Connect a b =
2     To   { signal1 :: a
3          , signal2 :: b
4          }
5   | From { signal3 :: String
6          , signal1 :: a
7          }
```

**Listing 2:** Record syntax

#### Record Syntax {..}

Listing 2 shows a record syntax version of the Data type example of Listing 1. The record syntax, the part between { } has accessors. The accessors are the functions `signal1`, `signal2` and `signal3` on lines 2-3 and 5-6 respectively, which allow us to read individual values from the constructor. Accessors with the same name, in different value constructors, but within the same data type, are linked to each other. This is useful to connect different elements. An example is `signal1` on lines 2 and 6.

#### Maybe Type

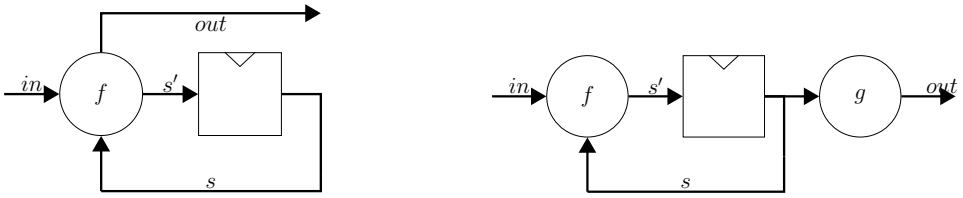
```
1 data Maybe a = Nothing | Just a
```

***Listing 3:*** *Maybe type*

The `Maybe a` data type, see Listing 3, is an existing data type, consisting of two value constructors, namely `Just a` and `Nothing`. The type contains either a value, `Just a`, or is empty and is displayed as `Nothing`. This is useful because, during type matching, we can easily see if something contains data or not.

### 2.2.3 Moore and Mealy

To transfer data from one clock cycle to the next, we can place a register between the out-and input of a function. We do this by using Mealy or Moore functions.




***(a)*** *Mealy machine* ***(b)*** *Moore machine*

**Figure 2.5:** Finite state machines

Figures 2.5a and 2.5b show a Mealy and Moore machine, as implemented in Clash, respectively.
Mealy functions are functions whose output `out` and new state `s'` can depend on the input `inp` and the previous state `s`.
The output `out` of a Moore function depends only on the previous state(s) (and possibly applied to second function `g`) but is in any case independent of the input `inp`. The new state `s'` may be dependent on the state `s` and input `inp`.

```
1 f s inp  = (s' ,out)
2   where
3     out = s + inp
4     s'  = inp
```

***Listing 4:*** *Mealy example*

An example of a function that can be used in a Mealy machine is implemented in Listing 4 where the new state `s'` is the input `inp` and where the output `out` is equal to the state `s` plus the input `inp`. In the example, it is clear that output depends on input `inp` and state `s`.

## 2.3 Dataflow

Dataflow is a suitable and well-known communication model to model time and data dependency. Only the relevant parts of the dataflow are explained in this thesis. The book of RTS2 [14] gives more comprehensive coverage of dataflow graphs.

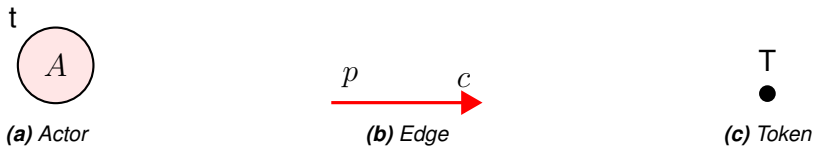



**Figure 2.6:** Dataflow parts

### 2.3.1 SDF

An SDF graph is a directed graph consisting of actors, edges, and tokens, see Figures 2.6a, 2.6b and 2.6c respectively, where the actors are visualised as nodes, the edges as red arrows and the tokens as dots. The edges represent First In First Out (FIFO) queues with unlimited storage space. In the FIFO, tokens are stored. At time 0 the number of tokens on an edge equals the initial tokens. Multiple tokens on edges are represented with dots or indicated by a number $T$ close to a token. A number at the end or beginning of the edge represents the number of tokens the actor consumes or produces. We indicate the consuming rate with a c and the producing rate with a p, where $c \geq 1$ and $p \geq 1$. The actors have a firing rule that they may not do anything before the tokens on the edge are equal to the consumption rates. After firing the actor produces tokens equal to the production rates. The firing duration $t$ is the time between consuming and producing.
An SDF graph, where every actor only consumes and produces one token per edge, is called a Homogeneous Synchronous DataFlow (HSDF) graph.

### 2.3.2 Self-Timed Schedule

If an actor fires as soon as possible, then the schedule is self-timed. Because an (H)SDF has a monotonic property [15], a shorter firing time of one actor cannot result in a later start time of another actor.

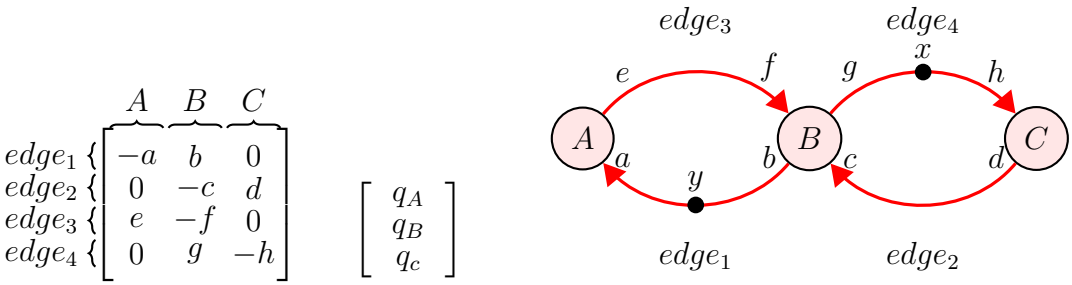
### 2.3.3 Strongly Connected

If there is a path, from every node in the graph to every other node, the graph is strongly connected.

### 2.3.4 Backpressure

When the producing actor is faster than the consuming actor, the producing actor experiences resistance, backpressure models this behaviour. Some strategies to solve backpressure are, dropping tokens, controlling the producing rate by a feedback edge or buffering, where produced tokens are stored in some memory unit until there is no more production but only consumption from another actor.

### 2.3.5 Topology Matrix




*(a)* Topology matrix *(b)* Repetition vector *(c)* Dataflow graph

**Figure 2.7:** Topology matrix example

A topology matrix shows the edges of a dataflow graph. Where the rows show the edges and the columns show the actors. A positive number in the matrix represents a producing edge, and a negative number represents a consuming edge. When the number is 0, it means the edge is not connected to the actor expressed in the column. An example can be seen in Figure 2.7.

### 2.3.6 Repetition Vector

With the topology matrix,see 2.7a, there is a way to find the firing rate, which is the number of times an actor has to fire before it is back in its initial state. This number of times is the repetition vector, see 2.7b. Finding this can be done by solving the following formula $T\ \vec{q}=\vec{0}$ where $T$ is the topology matrix, and $\vec{q}$ is the repetition vector. The values of $\vec{q}$ are both positive integers, and the only factor that divides both of them is one.
If the topology matrix has rank $n-1$, then the repetition vector exists. To find the rank of a matrix, transform the matrix to its row echelon form and count the number of non-zero rows. The repetition vector helps indicate the buffer sizes of the FIFO used between the nodes.

## 2.4 Network Topology

Different kind of network topologies exist, there are logical-topologies and physical topologies. The logical topology indicates how it seems that the devices are connected, but the physical topology is the structure of how different devices are connected via wires. Both topologies do not have to be the same. We use the physical topology as the connection of different FPGAs, and we replace the logical topology with a dataflow graph.

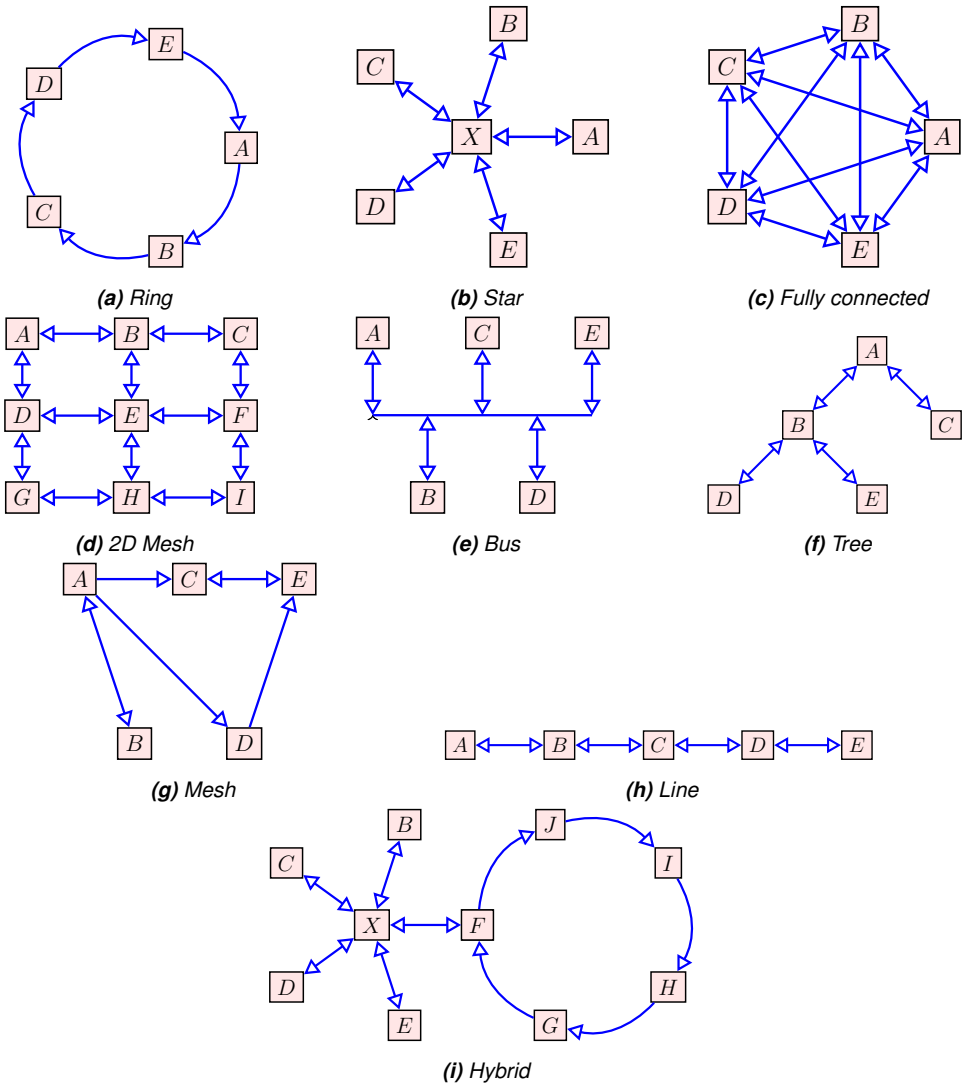



**Figure 2.8:** Topologies

From now on, when we use the word topology or hardware topology, we mean physical topology because we replace the logical topology with a dataflow graph. There are different topologies to implement, such as Ring, Star, Line, Mesh, Bus, Fully Connected, Tree and Hybrid, see the topologies of Figure 2.8. A Blue and open arrow displays the connection between topology nodes.

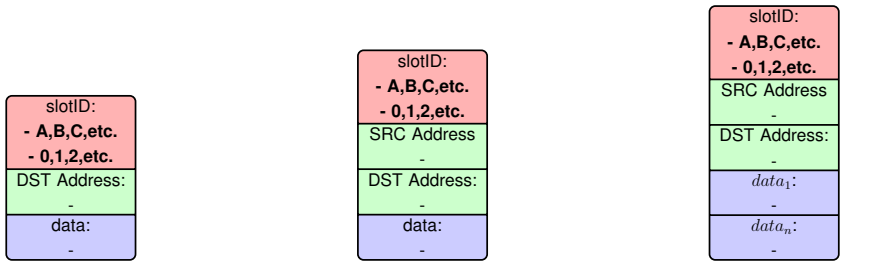
## 2.5 Nebula Ring Interconnect

The nebula ring-interconnect is an all to all interconnect. The ring is unidirectional, and data travels one-hop every cycle until it reaches its destination.
Much of the Nebula ring is eventually used in this project. Therefore, an explanation in this report, for more information, and the proof, about the nebula-ring, see [16]–[22].

### 2.5.1 Ringslotting

A quote of Dekens [18] and a rule that defines ringslotting:

**Rule:** *"If a slot identifier matches the identifier of the Network Interface (NI) it currently resides in, it is "owned" by that NI. Thus, NIs can always use their "own" slot to inject data onto the ring."*




***(a)*** *Old slot* ***(b)*** *New slot* ***(c)*** *New extended slot*

**Figure 2.9:** Nebula slots

The functioning of the ring is explained through an example.
In the subfigures of Figure 2.10, we see three nodes A, B, C. In front of that are three slots, each slot, see Figure 2.9a consists of a slotID, a destination address and a location for the data. Each Clock cycle the slots shift. If the slotID is equal to the node ID, the node can put data in the slot. In the example the clock = 0, see Figure 2.10a the slot with slotID A is offered to node A. Because slotID A is equal to the node Id A, A is allowed to place data in the slot, see Figure 2.10b. In this case, the destination is node C, and the data is the word *Hello*. At the beginning of the next clock cycle, slot A including its contents, is offered to node B. Because node B is not the destination, the data travels further on the ring. At the beginning of the next clock cycle, slot A is offered to node C, see Figure 2.10c. The destination is node C, so node C retrieves the data and clears or overwrites the data from the ring. In the example, it is cleared, see Figure 2.10d. In practice, the other nodes can of course also place data on the ring if the slot ID is equal to the node ID, but to keep the example simple, this has not been added. A receiving node always accepts incoming data.

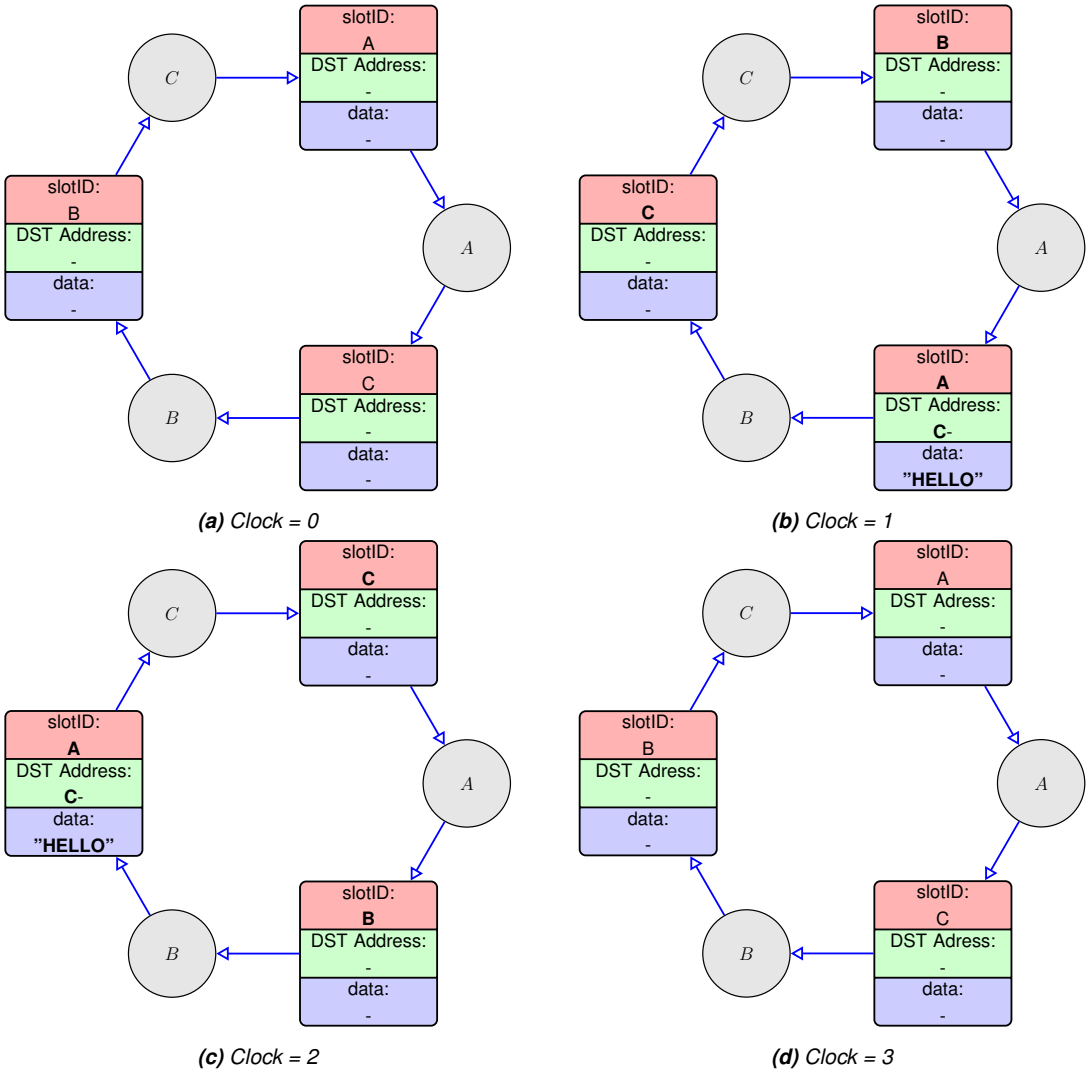



**Figure 2.10:** Nebula ring example

### 2.5.2 Hijacking

A node can now only send data once in the 'n' number of slots. Slots = 3 in the example of Figure 2.10. To lower the latency, it is, sometimes, possible to hijack slots of other nodes. This is best explained by a quote and rule of Dekens [18]:

**Rule:** *"If a NI is ready to send data, the current slot is empty, and the owner of the slot is not reached before the destination NI is reached, data can be injected into that slot."*

# Chapter 3

# Related Work

This chapter shows and compares the work of others, that relates to our thesis. We look at the differences between Nebula ring interconnect and our implementation. We discuss the different, physical and logical, topologies used in different Multi-FPGA systems. We are also looking at FPGA dataflow implementations, whether or not made in Clash.

## 3.1 Nebula Ring Differences

Much of the nebula ring is taken from [16]–[22], but some parts are different. Therefore, this is related work. The differences are:

- The implementation in this project is not connectionless but connection-oriented, this means that we use dedicated buffers for each connection, instead of shared memory.
- We implemented the flow control in hardware and not in software.
- We do not use an external memory location that is sent along with the data, see Figure 2.9a, but we do send a source address so that the receiver can determine where the data is stored, see Figure 2.9b.
- We can make a model with multiple slots for every node, this decreases the latency of the new actor see Figure 2.9b
- The width of the ring is also adjustable, making it possible to place multiple tokens on the ring at the same time. So only one SlotID, source and destination is needed for multiple tokens, see Figure 2.9c
- For the implementation, we give up the point of low hardware cost, but for that, the design is spread over multiple FPGAs.

## 3.2 FPGA to FPGA Communication

In a PhD thesis of Khalid [23], they try to find the best routing architecture topology concerning cost, speed and routability, using an experimental approach to evaluate and compare different architectures. The architectures they use are different mesh structures and crossbar implementations. This is in contrast to our project, where we also look for a topology and eventually choose a ring topology.

In an Article of Ramezani [24], CPA [1] is presented to schedule task graphs on multi-FPGA systems. This means that they are looking for the order in which specific tasks are executed and on which FPGA. They take into account the communication time between different FPGAs. They can also place multiple tasks on one FPGA. In their example, in Chapter 3.5, they schedule five functions on two FPGAs.
Although finding the order in which the tasks are executed is not essential in this project, it is useful to know which actor should be modelled on which FPGA. They use a separate central controller for managing and scheduling different FPGA tasks. We use a dataflow graph for the scheduling, where each actor is placed on one FPGA and, therefore, there is no separate controller.

In a Paper of Owaida and Alonso [25], they use a ring network topology for distributing data. They have a single master node where all data is stored and distributes data to slave FPGAs. Eventually, the partial results are propagated and aggregated over the ring until it reaches the master node. A difference with our implementation is that we don not have a master node where the result ends, but every actor of the dataflow graph acts as a master node, from where data starts and ends. Another difference is we do not merge the results, but depending on the slot, send the results over the ring to the next FPGA until it arrives at the right actor.

In the paper of Mencer et al. [26], they present CUBE, where they have connected 512 FPGAs as a systolic chain with identical interfaces between them. Their complete system is mostly similar to what we have in mind. Except that they use a chain where we use a ring and that they want to work, in the future, on deterministic communication[2]. We do deterministic communication through dataflow graphs and use of the Nebula ring Interconnect over the whole hardware topology.

[1] Critical Path-Aware
[2] We cannot find their future work on a deterministic communication for the cube.

## 3.3 Dataflow on Hardware

In Chapter 5 of the Master thesis of van Raalte [2], the generation of hardware architecture from dataflow is proposed. A similarity with this project is that it also uses Clash. However, there are also differences, and those differences are:

- He generates a hardware architecture to implement on one FPGA. Contrary to this thesis, where a dataflow graph is spread over multiple FPGAs.
- He makes uses of a data dependency graph and a separate dataflow graph for scheduling, see Figure 3.1. Whereas we use the dataflow graph as the data dependency graph.
- He makes use of a general scheduler connected to all nodes of the dataflow graph.
- He uses a crossbar to get data from one actor to the other, whereas we use a ring hardware topology to connect the different FPGAs.
- The firing of the actor and the crossbar is controlled by a separate controller, whereas we use all FPGAs al partial controllers.
- His implementation only supports a strictly periodic schedule and has to comply with this schedule, always. Our project uses self-scheduling. Thus, it is allowed to go faster than the WCET, because of the monotonic properties of an SDF graph.
- His actors have a firing time of at least one clock cycle, where our actors could have a firing time of zero clock cycles, but we do add extra nodes to the dataflow graph that take time.
- They have no support for SDF graphs yet. Where we can use SDF graphs.

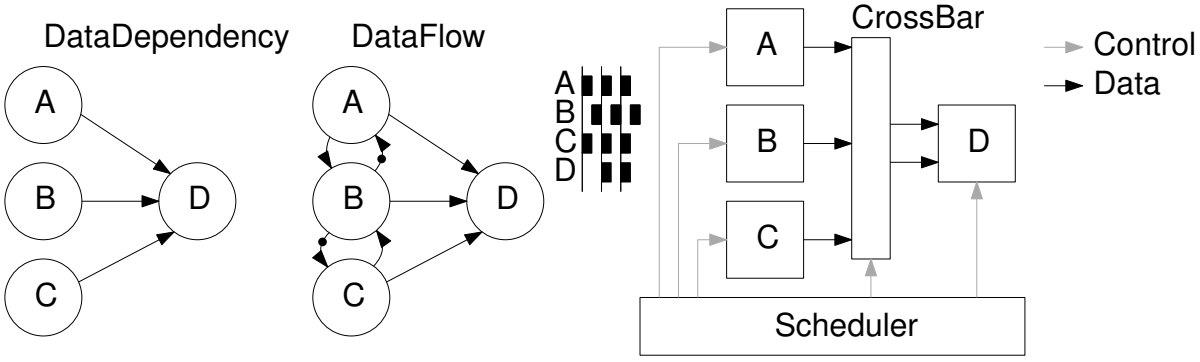



**Figure 3.1:** Hardware architecture [2]

In a paper by Liu et al. [27], they map an initial dataflow graph across multiple processors. They also place new actors between existing actors to model the time between different processors. They call them network actors; we call them identity actors. They assume the network actors have a constant delay, something we calculate depending on the buffer sizes. Another project, which has many similarities with our project is the PhD thesis of Ali [28]. Where they map dataflow graphs on a 2d-mesh topology, differences are that we use a ring topology, instead of the 2d-Mesh, so our routing is more straightforward, and we use FPGA where they rely on CPUs.

# Part II

# Design Space Exploration (DSE)

# Chapter 4

# Topology Choices

This chapter depicts the design choices of the hardware topology. The topology is the physical connection of the different FPGAs. This chapter also shows the pros and cons of different topologies and why a particular topology is chosen. That is why we are answering the following question:

*Which hardware communication infrastructure is suitable?*

The topology is needed to give us a general structure on which we can map a dataflow graph, see the centre cloud of Figure 4.1.


**Figure 4.1:** Designflow: Hardware topology

## 4.1 Connecting FPGAs

### 4.1.1 Topologies

Table 4.1 shows on which topology a dataflow graph can be mapped most successfully. The top row of the table shows an overview of the considered implementation factors for the different topologies. The scale on the second row shows the importance of each element, where a higher number is more significant than a lower number.
A plus sign means to add one point and a minus sign means to subtract a point, this is multiplied with the scale. The total column indicates the value of the various

**Table 4.1:** DSE Topologies

| | Modularity | Physical setup | Input\output pairs | T hroughput | Total |
|---|---|---|---|---|---|
| **Scale** | 2 | 2 | 1 | 1 | |
| **Bus** | +++ | +-- | +++ | --- | **4** |
| **Fully Connected** | +++ | --- | --- | +++ | **0** |
| **Hybrid** | +- | +- | +- | +- | **0** |
| **Line** | ++- | ++- | ++- | +- | **5** |
| **Mesh** | +- | +-- | +-- | +-- | **-4** |
| **Ring** | +++ | ++- | +++ | +- | **11** |
| **Ring-Intermediate** | +++ | +++ | ++- | ++- | **14** |
| **Star** | ++- | +-- | ++- | ++- | **2** |
| **Tree** | +- | ++- | +-- | +-- | **0** |

topologies. A higher value is more positive and therefore, has a better chance of success.

### Modularity

Because of the intention to spread the dataflow over multiple FPGAs, we need an algorithm that controls the communication. There are three possibilities to do this, namely:

1. One controller FPGA.
2. A controller on some of all the FPGAs.
3. Splitting the controller (equally) overall FPGAs.

For one FPGA that controls other FPGAs theoretically, all topologies are possible, but due to physical constraints, it is not possible to implement all topologies. We could divide the controlling over a part of the FPGAs, but this does not give a uniform structure for every FPGA. A drawback of splitting the controlling technique overall FPGAs is that every FPGA needs controlling. Nevertheless, we prefer this option because every FPGA can be in a uniform structure. We think the uniform structure

is essential because this ensures modularity and helps with the implementation, that is why we set the scale to 2.

#### Physical Setup

The physical configuration of the FPGAs is an essential factor to consider because this is how the FPGAs are drawn up. If we assume, we will place the FPGAs in the layout of the topology, and we connect the FPGAs with wires, then we need for some topologies, many wire crossings to connect them, this is inconvenient. Also, adding a board must be not too cumbersome. Therefore, the scale is set to 2.

#### I/O Pairs

The combination of an input and an output is called an in/output-pair or link. The number of connections is also a factor to consider because we do not want a topology with many links. After all, there is not one FPGA with unlimited in-output pairs. So we want a setup with the least amount of links. Although it is essential, it doesn't matter whether it is one or two links, this is still comprehensible and, therefore, the scale is set to 1.

#### Throughput

The throughput is not the most important. Therefore, the scale is set to 1, because we want the maximum communication time, a.k.a WCET, to be deterministic. However, it is still a factor to consider because slow performance can mean that no user wants to use the system.

### 4.1.2 Choosing Topology

From Table 4.1 we can see that the total score of four of them come above five points, so for these topologies, are next some pros and cons explained.

*Bus*
- PROS:
  - Only one in/output pair per FPGA.
  - Every FPGA has the same structure if connected to one bus.
- CONS:
  - The output must be protected from incoming messages from other devices.

* There should be a protocol in place to decide when an FPGA is allowed to speak on the bus. This protocol influences the throughput.

*Ring*

- PROS:
  - Only two connections per FPGA.
  - Every FPGA has a uniform structure.
  - A deterministic solution exists (Nebula ring interconnect).
- CONS:
  - The ring communicates in one direction, so communication with a previous FPGA can be slow.
  - One incorrect or broken wire brakes the whole system.
  - The first and last FPGA must be connected. This connection can cause long wires.

*Line*

- PROS:
  - Communication in two directions, so a possible faster response than a ring topology.
  - Physical placement, the FPGAs can be next to each other.
- CONS:
  - The ends of the structure are not uniform with the inner FPGAs.

*Ring Intermediate*

A ring whereby the feedback loop runs through the previous node, see Figure 4.2.

- PROS:
  - Communication in two directions, so a possible faster response than a ring topology.
- CONS:
  - The ends of the topology do have a self-loop.
  - It is an improvement from the ring topology. Therefore, the ring topology should be implemented first.

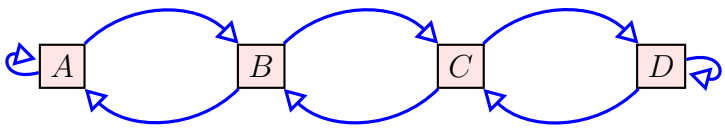



**Figure 4.2:** Ring-intermediate topology

## 4.2 Conclusion Topology

Even though the ring does not have the highest score in Table 4.1, as a starting point, taking into account the developing time, we choose the ring topology, because a deterministic solution exists. Namely, the Nebula ring interconnect, described in the Background Chapter 2.5. It has a deterministic implementation and is familiar at the CEAS-group of the University of Twente. However, for a more accelerated communication and higher throughput, it would be, in the future, an idea to implement ring-intermediate, whereby the rules of the nebula ring could be adapted to be still deterministic.

# Chapter 5

# Realisation and Structural Choices

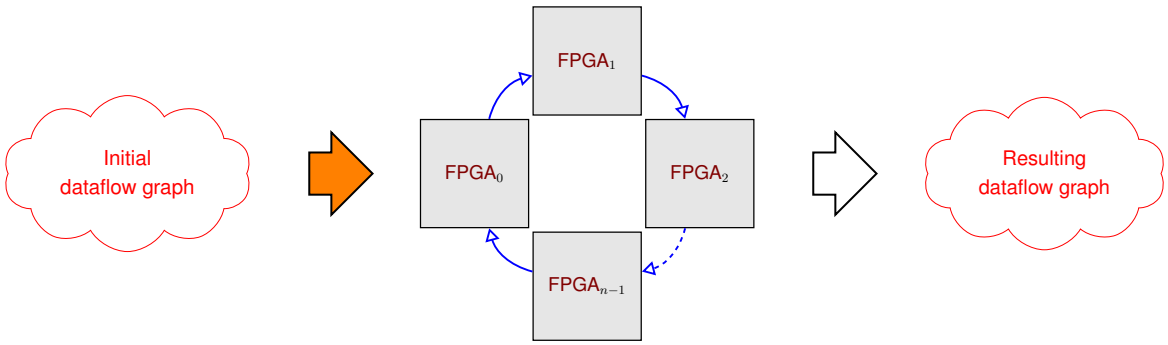



**Figure 5.1:** Designflow: Initial dataflow graph to ring topology

After the choice for a ring topology, with the Nebula ring interconnect. We have a structure on how the FPGAs are placed and communicate. We answer the following question.

*Given the topology, how do we map a dataflow graph onto multiple FPGAs?*

The question represents how to get from the initial dataflow graph to the ring hardware topology, in which every task modelled in the initial dataflow graph is executed on multiple FPGAs, see Figure 5.1. This chapter also explains how an actor of the initial dataflow graph maps to the FPGAs in the ring. We start with a dataflow graph and convert it to the ring hardware topology. It also shows what parts of the dataflow graph correspond with the ring.

To give an impression of the realised design, Figure 5.2 shows a ring containing three FPGAs. It also shows the different realised elements on an FPGA.

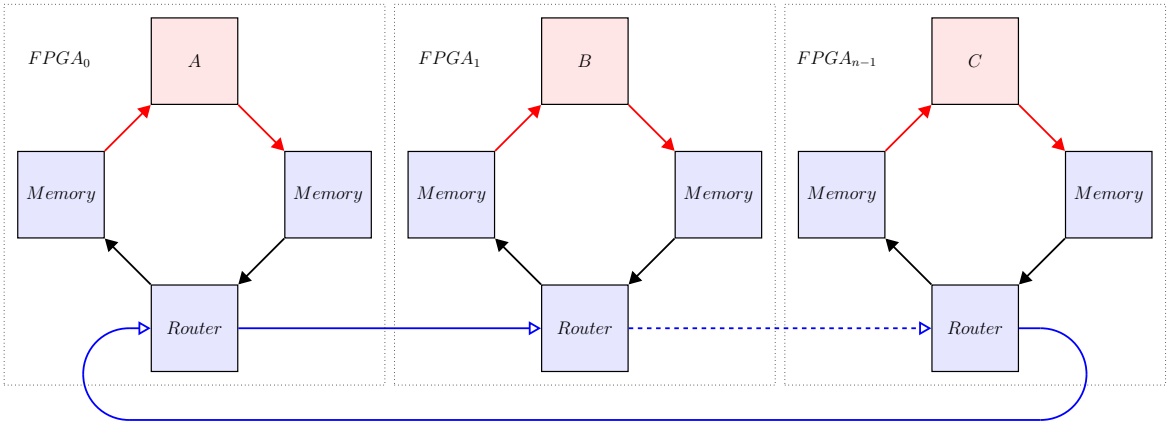



**Figure 5.2:** Brief hardware implementation preview

However, before that, we answer the following question:

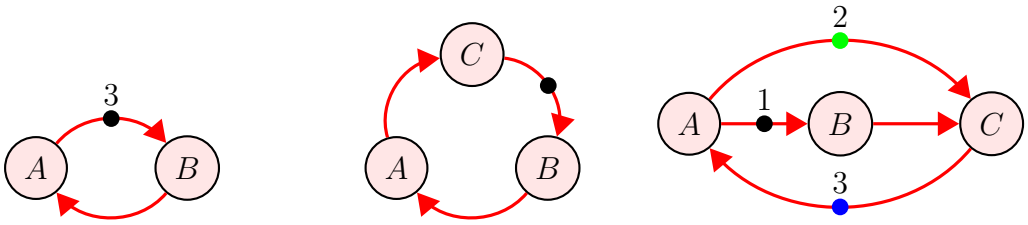
*Are there any dataflow graph constraints, if so, which ones?*

## 5.1 Dataflow Constraints

To execute a dataflow graph on the ring topology, we need to come up with a dataflow graph that we will turn into hardware.
We start with a simple dataflow graph of Figure 5.3. In hardware, we would like to assign this to a ring topology with two FPGAs.


**Figure 5.3:** Simple dataflow graph

In dataflow, an actor can place an unlimited amount of tokens on an edge. In hardware, this is not possible because edges represent memory, and unlimited memory does not exist. Therefore, we need a feedback link that limits the production of an actor. This limitation can come from the receiving actor, see Figure 5.4a, but in case of another dataflow graph can also come from another Actor, see Figures 5.4b or 5.4c. So the first constraint was that every edge needed a controlling backpressure edge, but in our case, this constraint is too strict, and therefore if the graph is strongly connected, it is also fine.
Another constraint is that a dataflow graph should not have multiple edges from one actor to another, because we then need an indicator to distinguish between different edges.[1].




***(a)** Example 1* ***(b)** Example 2* ***(c)** Example 3*

**Figure 5.4:** Dataflow graph examples

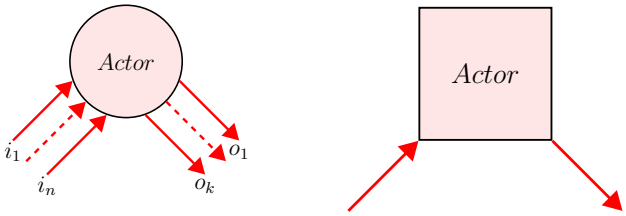
## 5.2 General FPGA Realisation Information

Every task, represented by a node in the dataflow graph, is executed on a different FPGA, where a ring topology physically connects FPGAs. Due to this ring topology, the in-and outputs of every FPGA are the same. Therefore, the model for every FPGA is the same. The input and one output of every FPGA consists of parallel wires connecting them.

[1] This is because of the implementation

## 5.3 FPGA Elements

### 5.3.1 The Actor




***(a)*** *Dataflow actor* ***(b)*** *Hardware actor*

**Figure 5.5:** Actor models

In this thesis, dataflow actors are modelled and shown by circular nodes. Hardware elements are modelled and shown by a square box. To create a model that describes the dataflow actors but also the hardware function, we use a red square node. Not all actors in a dataflow graph have the same incoming and outgoing edges, see Figure 5.5a. Therefore, in hardware, the edges are bundled and displayed as a single red arrow, see Figure 5.5b. So the arrows only represent data dependency, not the actual data size.

### 5.3.2 Memory

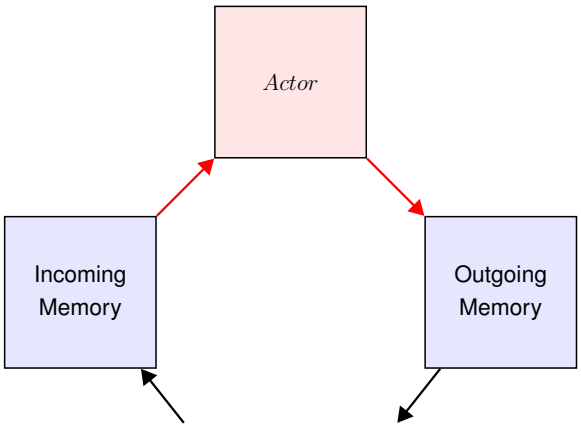



**Figure 5.6:** Actor and memories

As mentioned earlier, not all actors in a dataflow graph have the same number of edges. To model these edges, every FPGA has two memory elements. One to model the input edges and one for output edges.

The outgoing edges of the actor are the incoming edges of another actor on another FPGA. It seems contradictory to implement two memory element, but this is done because, in the end, a ring interface is placed between the in and outgoing memory of two FPGAs. We need the input memory because an actor processes tokens and is maybe still busy with a previous firing or has to wait for tokens from other edges. So it can not accept new tokens directly.
We need output memory because not all data can be on the ring at the same time. So, the memory elements serve as synchronisation.

Figure 5.6 shows how the node of Figure 5.5b connects to the two memory elements. In the memory elements, the edges that run to and from the actor are described as FIFOs.
These FIFOs can have two different forms, namely bundled or separate. In the bundled version we combine all edges, so the data goes through one FIFO. In the separate form, we give every source/destination an own FIFO buffer. Because we have two memory elements, we have to choose for both a bundled or separate implementation. Therefore, we have listed the pros and cons of bundled and separate memory elements.

#### Bundled

A memory element consists of one FIFO in which all tokens/messages go through.

PROS
- Uses most likely less memory

CONS
- We need a control algorithm that prevents messages from getting mixed up.
- For the incoming memory. It is difficult for the actor to see if there are enough tokens available.
- For the outgoing memory, messages that are not in the first place of the FIFO cannot be put on the ring.

#### Separate

A memory element consists of parallel FIFOs, where each FIFO represents a dataflow edge.

PROS
- No mix up of tokens/messages.
- With the help of higher-order functions in Clash, we can create one FIFO buffer, and map this to get the separate FIFO buffer.

- For the incoming memory, the actor knows of every edge if there are enough tokens available.
- For the outgoing memory, there is a row of ”first” messages in the FIFOs. So we can choose which message, to what destination, to insert on the ring.

CONS
- More memory is needed [2]
- The design loses some flexibility because the input memory can only receive from the initially designed nodes of the dataflow graph.

### Incoming Memory

For the incoming memory, we choose the separate implementation because there is no mix up of message, and an actor can for every edge see that there are enough tokens/messages available. From now on, we call the separate incoming FIFOs, incoming buffer or input buffer.

### Outgoing Memory

For the outgoing memory, we also chose for the separate implementation, because we then have a row of first messages, to choose from, and to put on the ring. From now on, we call the outgoing separate FIFOs, outgoing buffer or output buffer.

[2] Due to the Clash implementation, we store all data in separate FIFOs and cannot share the memory

### 5.3.3 The Router

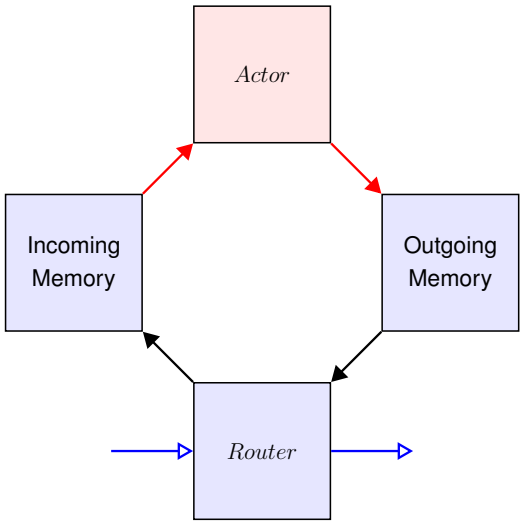



**Figure 5.7:** Basic hardware implementation, actor, memories and router

The router is the element between the memory elements and the ring, see Figure 5.7. The router makes two decisions. The first is to decide whether a message coming from the ring is addressed to the FPGA or not. If it belongs to the FPGA, it can store it, depending on the source, in one of the FIFOs of the incoming buffer. If a message is not addressed to the FPGA, the message will continue on the ring. The second decision is to select from one of the FIFOs of the outgoing buffer, which message is allowed on the ring.

### 5.3.4 Ring Hop

The ring hop displays the time a message spends on the ring between two consecutive FPGAs. i.e. the ring Hop it the same as the slot of the nebula ring, see Figure 2.9b.

### 5.3.5 Controlling

As explained in Chapter 2.3, the properties of an SDF graph are that they consume and produce a fixed amount of tokens. The actor must be realised by the designer that implements the function of the dataflow actor. A controller is placed between the actor and the buffers, to take some pressure off for the designer. The controller checks whether the amount of tokens coming and going to the buffers is equal to the consumption and production rates of the dataflow graph. If there are a sufficient amount of tokens stored in the input buffer, the controller will signal the actor that it can fire. If the actor produces a sufficient amount of tokens, only then the controller will place them in the output buffer.

### 5.3.6 Complete FPGA

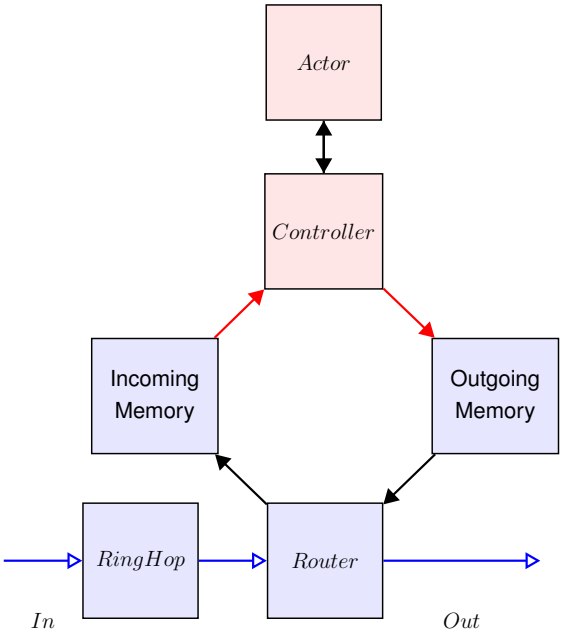



**Figure 5.8:** FPGA implementation

The implementation of an FPGA is formed by connecting the different elements.[3], see Figure 5.8.

## 5.4 The Ring

When the different FPGAs are connected, it looks like Figure 5.9, where every FPGA has its own, on the ring increasing, ID. Tokens from the dataflow graph travel along the topology ring.

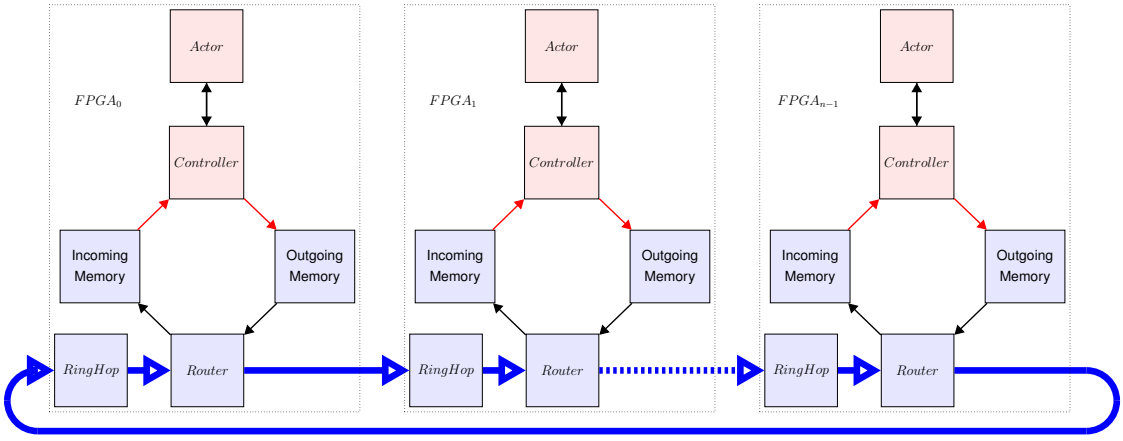



**Figure 5.9:** Hardware ring implementation example

[3] Actor, controller, incoming buffer, outgoing buffer , router and ringhop

## 5.5 Summary by Example

Looking at the initial dataflow graph example of Figure 5.10, in Figure 5.11, we divide the initial tokens between the output buffer of one actor and the input buffer of another actor. This distribution is to show that an incoming FIFO belongs to an outgoing FIFO.
If the slotID is equal to the ID of the FPGA, then each router may decide from which output FIFO a message is placed on the ring [4]. When the tokens arrive at their destination FPGA, the router places it in the correct incoming FIFO. Eventually, the controller will see that there are enough tokens in the input buffer and presents them to the function of the FPGA. After the firing time, the function will produce messages again, which will be placed in the output buffer.

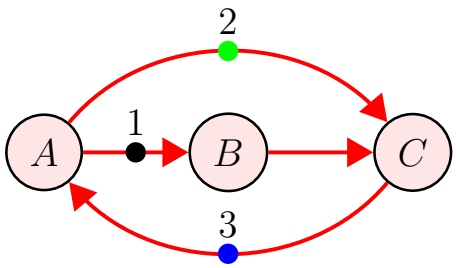



**Figure 5.10:** Three node, dataflow graph example

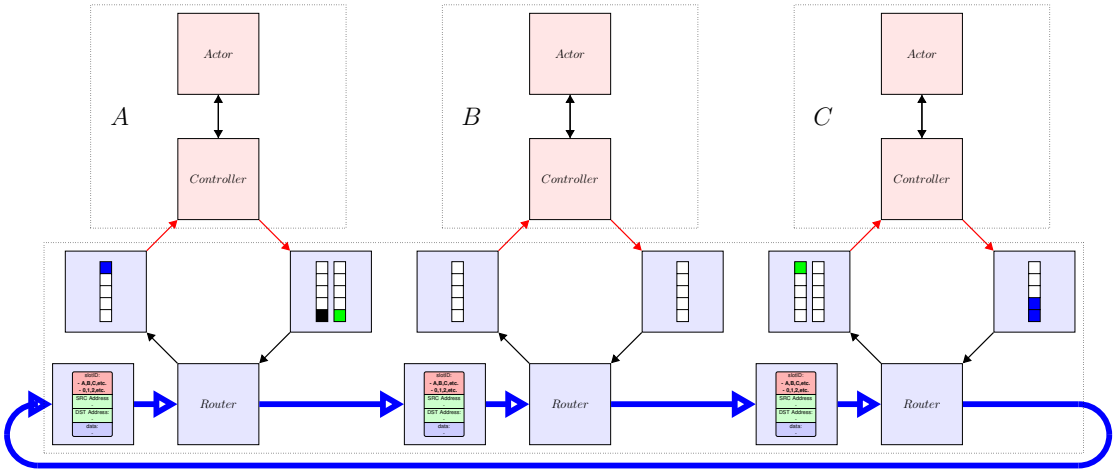



**Figure 5.11:** Hardware implementation: Three node, dataflow graph example

[4] B's router has nothing to put on the ring yet, and C's router has only one FIFO to choose from

## 5.6 Conclusion Realisation

To conclude this chapter, we have seen that the dataflow graph must be strongly connected. To implement the ring, we place various connected elements on one FPGA, which we connect to realise a ring. We use a router to receive or forward data from the previous FPGA. We also use separate FIFOs buffers for synchronisation between the FPGAs. The controller is used to make it easier for the user of the system. The user/designer remains responsible for the realisation of the actor, and the ringhop is used to represent the slots of the nebula ring.

# Chapter 6

# Clash Implementation Choices

This chapter shows the implementation in Clash. It first describes the setup of an FPGA. Secondly, it shows the type of ring. Then how to connect the different elements, such as the router, memory, controller and function to each other. Connecting is a cumbersome task, and with datatypes, we tried to simplify this. Then it explains in detail how various elements, and the corresponding choices, are implemented on one FPGA.

## 6.1 FPGA Setup

### Dataflow Setup

On every FPGA we want one actor, an actor has consumption and production rates, so we have to tell every FPGA what they are. Also, each FPGA has its own identification, so the Nebula ring interconnect, knows if a received message belongs to the FPGA or not. Because the Nebularing permits hijacking and can also be used as a credit-ring, we use an Embedded Domain Specific Language (EDSL) to indicate the different modes.

```
1 data Setup id wd rd s r =
2     Setup  { myId      :: id
3            , sIds      :: Vec s id
4            , amountS   :: Vec s (index (wd + 1))
5            , rIds      :: Vec r id
6            , amountR   :: Vec r (index (rd + 1))
7            , modus     :: RoutingMode
8
```

***Listing 5:*** *Default dataflow Data type*

In Listing 5, we see the data type `Setup`. This data type is used to define the three things, the name/identification of the FPGA, the producing and consumption rates of the actor, and the mode of the router.

1. `myId`, on line 2, is the identification of the FPGA. Each FPGA has a different number, letter, name, etc.
2. On every FPGA, one actor is placed, where each actor has input and output edges, with respectively consuming and producing rates.

   `sIds` , line 3, represents to which other actors the actor has edges.

   `amountsS` , line 4, represents the corresponding production rates.

   `rIds` , line 5, represents from which other actors the actor has edges.

   `amountR` , line 6, represents the corresponding consumption rates.
3. The modus, on line 7, indicates the direction of the ring and whether hijacking is used or not. There are four possible options, constructed in a data type `RoutingMode`, an overview:

`IncreasingWithoutHijack` Regular ring (increasing order) without hijacking.

`IncreasingWithHijack` Regular ring (increasing order) with hijacking.

`DecreasingWithoutHijack` Credit-ring (decreasing order) without hijacking.

`DecreasingWithHijack` Credit-ring (decreasing order) with hijacking.

### Element States

The different elements have states. By combining them, they are not all over the place, and it is more convenient for the user/designer to define them. By merging, we can use the value constructor as one Mealy machine, we do this in Chapter 6.3.1. This way, we are no longer entirely dependent on the functions `bundle` and `unbundle`.

```
1 data ElementStates id h sd r s d f a   =
2     ElementStates { obState     :: Vec s (Vec d (Maybe a))
3                   , ibState     :: Vec r (Vec f (Maybe a))
4                   , rState      :: Index s
5                   , rhState     :: Vec h (RingContent id (Vec sd (Maybe a)))
6                   }
```

***Listing 6:*** *type of Element States.*

Listing 6 shows the type `ElementStates` and is used to display the states of all the different elements on one FPGA. The accessors/states of this data type are:

`obState` , line 2, This is the state of the outgoing buffer. Thus, the states of the output FIFOs.

`ibState` , line 3, This is the state of the incoming buffer. Thus, the states of the input FIFOs.

`rState` , line 4, This is the state of the pointer in the router. This indicates from which output FIFO a message can be placed on the ring.

`rhState` , line 5, This is the state of the ring hop, so the state of the Nebula slot. It is a vector, so it is possible to put multiple slots in a row.

## 6.2 The Ring Content Type

Messages stored in the ring hop slots circulate in the ring. The width of the ring is determined by the size of the slot.

```
1 data RingContent  id c =
2           Invalid
3         | EmptySlot   {slotId :: id}
4         | ContentSlot {slotId :: id, source :: id, destination :: id, content:: c  }
5
```

***Listing 7:*** *Ring Content type*

We made a new data type, `RingContent`, see line 1 of Listing 7. This `RingContent` type has three value constructors [1].
One of the value constructors is the `Invalid` constructor on line 2. This constructor can be used in case the transportation time between two FPGA takes more than one clock cycle.
Another value constructor is the `EmptySlot` constructor, on line 3, this means, as the name says, the message on the ring is a valid slot, with slot ID, but is empty. The last value constructor `ContentSlot`, on line 4, means that it is a valid slot with content. The content consists of four accessors. Namely, the `slotId`, the `source` that is the source of the message, the `destination` where the content heads towards and the actual message, called `content`. Type `c` is a vector containing one or more content messages for the ring. The type `id` can be of any kind as long as it is a numeric type, such as Unsigned 'n' or Char. Because the ring content is in one type, we can easily see if a message contains content by pattern matching.

[1] A value constructor is an EDSL.

## 6.3 Connecting FPGA Elements

### 6.3.1 Clash Names

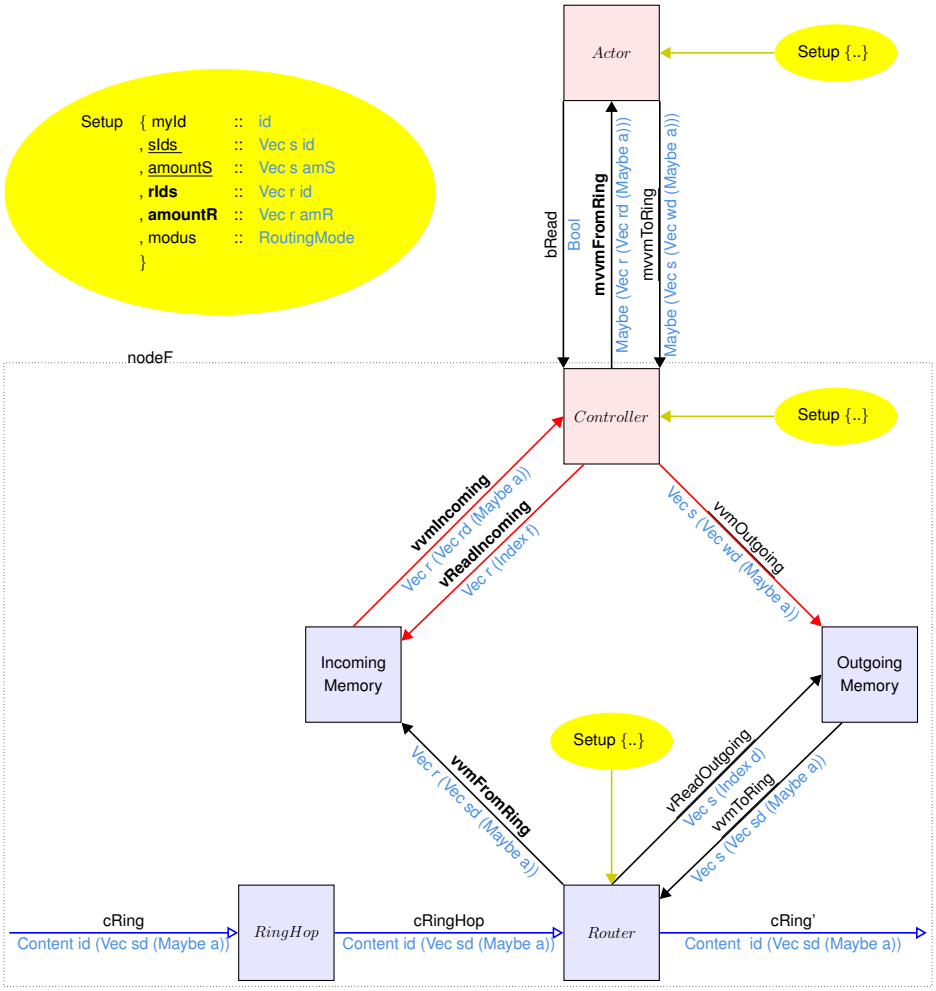



**Figure 6.1:** Clash implementations schematic

The names used, to link the different elements, which are functions in Clash, start with specific letters that correspond to the types. So when a name begins with:

**c** Ring Content type.

**b** Bool type.

**v** Vector type.

**vvm** Vector of Vectors with Maybe type.

**mvvm** Maybe a Vector of Vectors with Maybe type.

This naming is done to make it easier to understand what type goes from one element to the other and vice versa. To see how the functions and their type are connected in Clash, see Figure 6.1 or Appendix A.

### Connecting Data Type

Value constructors within the same data type can have accessors with the same name and type. Because Haskell/Clash is a pure language, they should always give the same result within a function. Because of this, accessors with the same name are linked to each other. We make use of this to connect different elements.

```
1  data ElementConnect id d f rd sd wd r s a  cr ff =
2      ...
3      | FromFuncCtrl       { mvvmFromRing      :: Maybe (Vec r (Vec rd (Maybe  a)))
4                           , vvmOutgoing       :: Vec s (Vec wd (Maybe a))
5                           , vReadIncoming     :: Vec r (index f)
6                           , vvmNewCredits     :: Vec r (Vec cr (Maybe (index 1)))
7      }
8      ...
9      | ToOutgoingBuffer   { vvmOutgoing       :: Vec s (Vec wd (Maybe a))
10                          , vReadOutgoing     :: Vec s (index d)
11     }
12     ...
```

***Listing 8:*** *Part of Connection data type*

For connecting the different elements, we made the data type, `ElementConnect`. Part of this data type is shown in Listing 8. For the rest of the data, see Appendix D.1.1. On line 1, the name of the data type is `ElementConnect` is given. The type parameters of the data type are `id`, `d`, `f`, `rd`, `sd`, `wd`, `r`, `s`, `a`, `cr` and `ff` will be explained in Chapter 6.3.2. For every different element, we made one input constructor and one output constructor. For easy recognition, all input constructors start with `To` and output constructors with `From`. `ToOutgoingBuffer`, on line 9, and `FromFuncCtrl`, on line 3 are examples of an input constructor and output constructor respectively. They share the common accessor `vvmOutgoing`, see lines 4 and 9, this is the link between the controller and the outgoing memory, see Figure 6.1.

### Connecting Elements

We use record wildcards to connect the accessors within a function. Using record wildcards is a trick, where we only have to write `{..}` after the value constructor, that makes the values of the accessors within the record syntax available without writing it all down.

The elements from Chapter 6.3.1 are connected in this chapter. Value constructors of the same type are given to the relevant function/element

The function we created to connect the different elements is nodeF, see Listing 9.

Lines 8-12 show the functions of the different elements we want to connect. Those functions are `funcCtr`, `inComingBuffer`, `outgoingBuffer`, `router` and the `ringHop`.

```
1  nodeF   Setup{..} ElementStates{..} ToNodeF{..}   =  (newStates, FromNodeF{..})
2    where
3      newStates = ElementStates { obState      = obState'
4                                , ibState      = ibState'
5                                , rState       = rState'
6                                , rhState  = rhState'
7                                }
8     FromFuncCtrl{..}                   = funcCtrl Setup{..}         ToFuncCtrl{..}
9     (ibState' , FromIncomingBuffer{..})= inComingBuffer     ibState ToIncomingBuffer{..}
10    (obState' , FromOutgoingBuffer{..})= outGoingBuffer     obState ToOutgoingBuffer{..}
11    (rState'  , FromRouter{..}        )= router   Setup{..} rState  ToRouter{..}
12    (rhState' , FromRingHop{..}       )= ringHop            rhState ToRingHop{..}
```

***Listing 9:*** *Connecting the Elements*

The controller and the router need the data type `Setup`, why they need it is made clear in Chapter 6.1. Other inputs of the functions are the (initial) states of the functions that are `ibState`, `obState`, `rState` and `rhState`. The values are the accessors of `ElementStates\{..\}` of line 1.
The last inputs of the functions are the input value Constructors that are `ToFuncCtrl`, `ToIncomingBuffer`, `ToOugoingBuffer`, `ToRouter` and `ToRingHop`. A part as said before, is shown in Listing 8 and the rest in Appendix D.1.1.

The functions return the output value constructors on lines 8-12 and the new states on lines 9-12. From the output value constructors, that are: `FromFuncCtrl`, `FromIncomingBuffer`, `FromOutgoingBuffer`, `FromRouter'` and `FromRingHop`. We use record wildcards again and thus ensure that the output of the functions is the input of other functions without having written them down. An advantage of this system is that it is easy to make a new connection between the elements by adding both accessors to two value constructors of the data type `ElementConnect`, see Listing 8. This connecting was especially useful during the design of the system. The other outputs of the functions, on lines 9-12, are the new variables of the states that are packed in a new data constructor, and also, value constructor, `ElementStates` on lines 3-7.
The output of the function `nodeF` on line 1, is the previously mentioned `newStates`, but also, the value Constructor `FromNodeF` which has as accessors the connections to the other FPGAs, but also to the user-defined ”real” function, that is the function of an actor in the dataflow graph.

### 6.3.2 Type Parameters

After the elements are connected, we set up the type of parameters

```
1 node_0_M ::HiddenClockResetEnable System =>
2 --                  ElementConnect id   d  f  rd sd wd r s a              cr ff
3      Signal System (ElementConnect Char 30 30 2  1  2  1 1 (Unsigned 100) 1  20)
4   -> Signal System (ElementConnect Char 30 30 2  1  2  1 1 (Unsigned 100) 1  20)
5 node_0_M = Mealy  (nodeF  def_0) init_0
```

***Listing 10:*** *Mealy node*

In Listing 10 we created on line 5 a Mealy machine, `Node_0_M`, from the function `nodeF`. We first give the `Setup` type, `def_0`, which is a function consisting of the data type `Setup`, as described in Chapter 6.1. As initial state, `init_0`, we provide a function consisting of the `ElementStates` described in Chapter 6.1. In this function, we define the different type parameters, as indicated in Chapter 6.3.1. Lines 3 and 4 show an example of a node with filled-in type parameters for the Mealy machine.
Next, the description of the type parameters:

- `id` The type of the identifier of the FPGA, such as Char or (Unsigned 10).
- `a` The type of the transferred data. It is defined by the user/designer and can be any type.
- `cr` The number of credits to receive from to credit-ring (not used).
- `ff` Depth of the incoming credit buffer (not used).
- `r` Indicates the number of input edges of an actor.
- `s` Indicates the number of output edges of an actor.
- `wd` The maximum producing rate of all the producing edges of an actor. i.e. if an actor has three producing edges(s= 3) with respectively 5,3,4 as production rate then $wd = max(5, 3, 4) = 5$.
- `rd` The maximum consumption rate of all the consuming edges of an actor. i.e. if an actor has five consuming edges(r = 5) with respectively 3,1,2,7,1 as consumption rate then $rd = max(3, 1, 2, 7, 1) = 7$.
- `sd` The length of the amount of message/tokens that can be placed on the ring at the same time. It is also, the number of messages/tokens coming from the ring, that is transferred to one of the FIFOs of the incoming buffer. Hence, the same name `sd` in Figures 6.6a and 6.6b.
- `d` The length of the FIFOs in the outgoing buffer. The length of d is determined by the maximum messages/token on an edge, so, i.e. if an actor has three producing edges with the maximum amount of tokens on the edge of 10,11,8, then the length of the FIFOs are $d = max(10, 11, 8) = 11$.
- `f` The length of the FIFOs in the incoming buffer. The same principle as `d` is applied to the consuming edges with type parameter/length `f`.

### Connecting an Actor/Function

The function to be performed by the actor is not yet connected to the connected elements `nodeF`.
To connect them, we need a function for the actor and connect this to the rest. This results in the implementation of one FPGA. The function was not connected, so, we can easily swap it.

```
1 f0  (state) xs =((state'), (output , read))
2  where
3           ...
4           ...
```

***Listing 11:*** *Function*

Listing 11 gives an example of the first line of a function `f0`, which is built as a Mealy machine.

```
1         f0M = Mealy  f0 (Nothing)
```

***Listing 12:*** *Mealy Function*

An example of a used Mealy function is shown in Listing 12, where the initial state is Nothing.

```
1  actor0 input = bundle ((cRing' <$> fromNode_)
2                       , (vReadCredits <$> fromNode_) -- credits read ( not used)
3                       , (vvmNewCredits <$> fromNode_) -- new Credits ( not used)
4                       )
5    where
6      (cRing_,  vvmCredits_)  = unbundle input
7      (toRing_, read_)        = unbundle $ f0M  (mvvmFromRing <$> fromNode_)
8      fromNode_               = node_0_M (ToNodeF <$> cRing_
9                                                  <*> toRing_
10                                                 <*> read_
11                                                 <*> vvmCredits_ --  Not used)
12                                        )
```

***Listing 13:*** *Function Connect*

In Listing 13, we connect the Mealy function, `f0M`, of Listing 12 and the Mealy function of the connected elements, `node_0_M` of Listing 10.
On line 6, we unbundle the input, in content from the ring and the credit-ring [2]. On lines 8-11, we connect the inputs of `node_0_M`; this results in the outputs of the node. On line 7, we extract the message for the function, which we then connect to the function. The results are the outputs of the function, which were already connected to `node_0_M` on lines 9 and 10. On lines 1 to 3, we take the results, for the ring ( and the credit part) from the node. For an illustration of the example, see Figure 6.2.

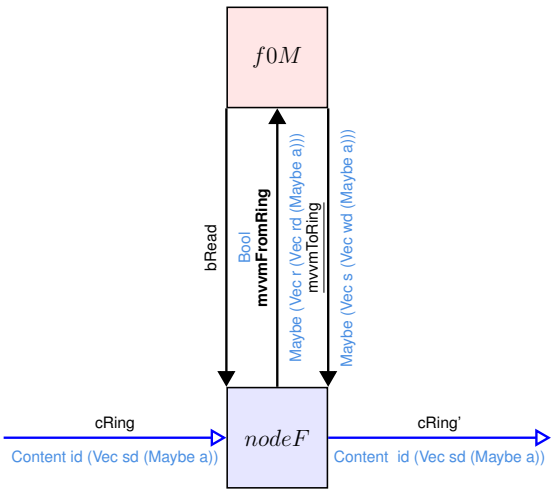



**Figure 6.2:** Connecting a hardware actor

[2]The credit-ring is not used or explained and is future work.

## 6.4 Elements in Detail

This section explains in more detail the different elements used on one FPGA. This chapter also explains the different type parameters mentioned in Chapter 6.3

### 6.4.1 Buffer

The buffers consist of multiple First In First Out (FIFO) buffers that are placed side by side, where every FIFO is an edge of the dataflow graph. The incoming FIFO buffer is a delayed copy of a FIFO in an outgoing buffer of another FPGA.

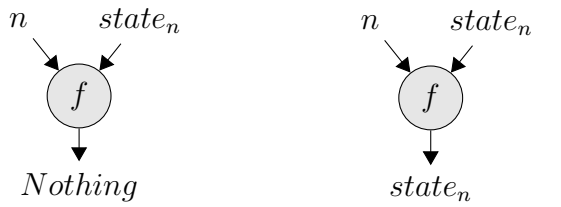
#### FIFO

The buffers consist of parallel FIFOs, and, therefore, we first explain one FIFO. The primary purpose of each FIFO is to receive and deliver multiple messages at the same time because we want to model SDF graphs.

```
1  fifoNN6 :: Vec ls (Maybe a)                          -- state
2       -> (Vec wd (Maybe a), index ls)                 -- xs
3       -> (Vec ls (Maybe a), Vec out (Maybe a))        -- (state'' , out )
4  fifoNN6 state xs                 = (state'' , out)
5    where
6      (inp, didRead)               = xs
7      ls                           = lengthS state
8      state'                       = imap f state
9        where
10         f idx s | idx < didRead   = Nothing
11                 | otherwise       = s
12     state''                      = take ls $ snd $ mapAccumRL g Nothing (state' ++ inp)
13       where
14         g acc x                  = case  x of
15                                      (Just _)  -> (x, acc)
16                                      _         -> (acc, x)
17     out                          = takeI state
```

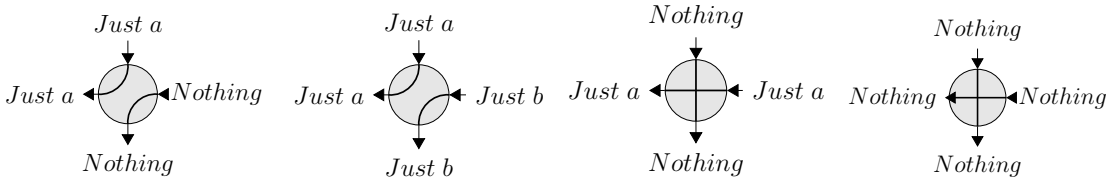
***Listing 14:*** *FIFO implementation*

We explain the implementation of this FIFO using Listing 14. For the visual representation we use Figures 6.3, 6.4 and 6.5.
The (initial) `state` on line 4, is the state of the FIFO at the beginning of a clock cycle. The `state` is a vector, because its length, `ls` on line 1, it is the length of the FIFO. This vector is a Maybe type, so it has `Just~a` data when occupied or is `Nothing` when empty. The other input is `xs`. `xs` is an input, consisting of a tuple, see line 6, where the first variable `inp` is the input of tokens/messages. It is a vector of length `wd`, see line 2, where `wd` is equal to the producing rate of the actor. The second variable in the tuple, `didRead` on line 6, we use to remove messages from the FIFO.




**(a)** $f : n < didRead$ **(b)** $f : n \geq didRead$

**Figure 6.3:** 'f' Executions




**(a)** option 1 **(b)** [option 2 **(c)** option 3 **(d)** option 4

**Figure 6.4:** 'g' Executions

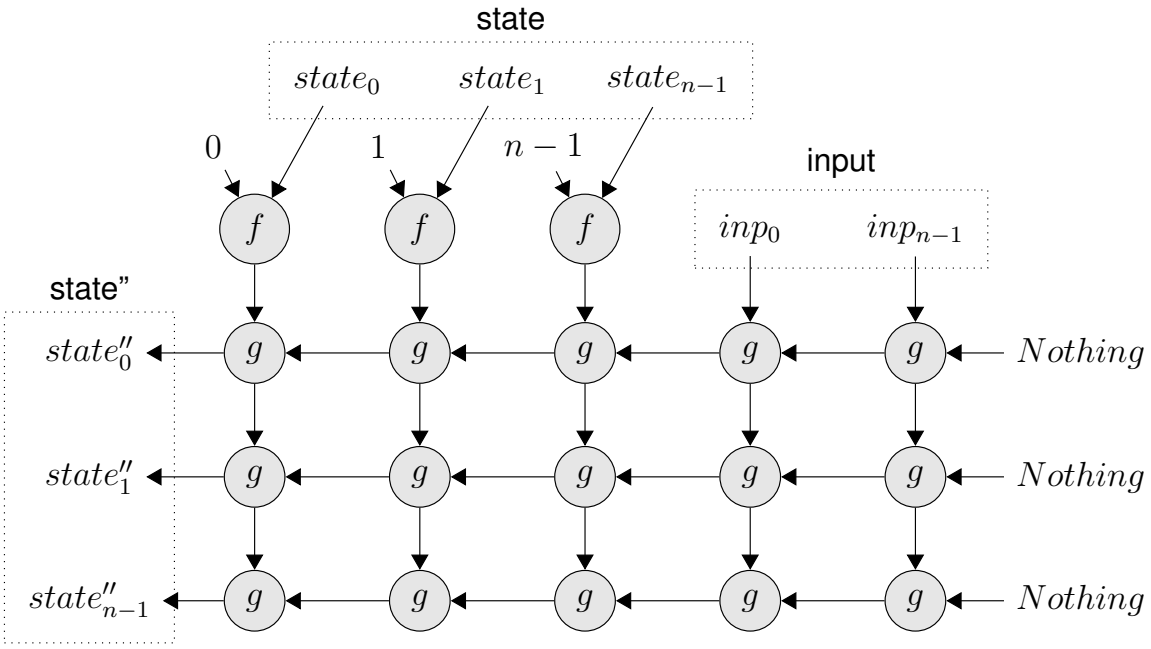



**Figure 6.5:** FIFO implementation

*So, how does the FIFO work?*

For this, we use Figure 6.5 as a reference. First, we delete messages from the FIFO. To do this, we use the `imap` function of line 8, where the variables of `state` are mapped over de function `f`. For all states where the index is smaller than `didRead` value, on line 10, the state is set to Nothing , if the state is greater than or equal to `didRead`, the state remains the same, see line 11. The two options of function `f` are shown in Figure 6.3.

The output of imap function `state'` is concatenated to the input `inp` on line 12. The result is transferred to the function `mapAccumRL` on line 12 within it propagates. The function `g` used in `mapAccumRL` function has four options, with two outcomes, these options are shown on lines 14-16 and in Figure 6.4 and show the propagation route of the messages.

Eventually, this results in a new state `state''`, because the length of the `state''` is the length of `state` + `inp`[3], we need to cut this off to the length of the FIFO, we do this with the `take` function on line 12.

The output of the FIFO is the new state `state''` on line 12 and the output `out` of line 17, which is a part of the (initial) `state`. The output length is the number of messages read from the FIFO at the same time. It is defined by the length of `out` on line 3. The number of messages read at the same time is equal to the consumption rate [4] of the dataflow graph or the number of messages on the ring at the same time.

[3] The figure does not show this.

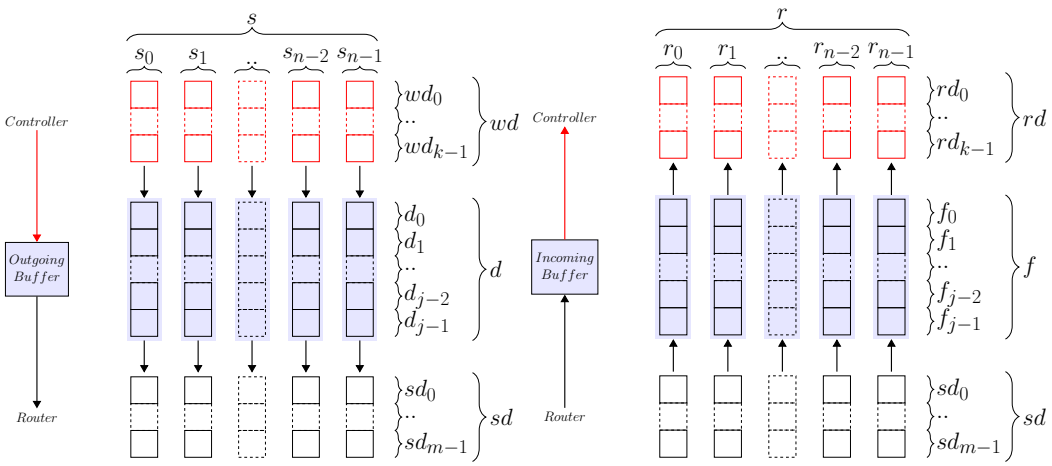
[4] Actually , the maximum consumption rate of the whole buffer.

### In/output Buffer

The in/output buffer consists of parallel FIFOs, where each FIFO represents part of a dataflow edge.

```
1 buffer states inps didRead= (states', o)
2   where
3     (states' , o)  = unzip $ zipWith fifoNN6 states (zip inps didRead)
```

***Listing 15:*** *Buffer*




***(a)*** *Outgoing* ***(b)*** *Incoming*

**Figure 6.6:** Buffer structure

We explain the in/output buffer using Figure 6.6 and Listing 15. Because the buffer function is created by mapping the FIFO in Clash, the FIFOs must have the same length. In Figures 6.6a and 6.6b, this length is `d` for the outgoing buffer and `f` for the incoming buffer respectively. The `states` on line 1 indicate the (initial) states of the FIFOs because one FIFO is a vector, `states` is a vector of vectors. The `inps` on line 1 are the inputs for the FIFOs, and depending on if it is the outgoing buffer or the incoming buffer, this is `wd` or `sd`, respectively, as shown in Figure 6.6. The `didRead` of line 1 is a vector with the number of messages read from the FIFOs, as explained before. On line 3 the buffer is mapped using the zipWith function as a function for the `zipWith` the FIFO `fifoNN6` is used and has variables `states`, `inps` and `didread`. They are zipped to get the right type for the FIFO. Eventually after unzipping this results in the new FIFO state `states'` and buffer output `o`, on lines 1 and 3.

### In/output Buffer Wrapping

```
1 outGoingBuffer states ToOutgoingBuffer{..}= (states', FromOutgoingBuffer{..})
2   where
3     (states' , vvmToRing)                = buffer states vvmOutgoing  vReadOutgoing
```

***Listing 16:*** *Incoming Buffer wrapper*

```
1 inComingBuffer states ToIncomingBuffer{..} = (states', FromIncomingBuffer{..})
2   where
3     (states', vvmIncoming)                =  buffer states vvmFromRing vReadIncoming
```

***Listing 17:*** *Outgoing buffer wrapper*

Listing 16 and 17 are wrapper functions that wrap the buffer so that they become the elements, incoming buffer and outgoing buffer, as explained in Chapter 6.3.1. The functions are implemented as a Mealy function. So, on line 1 there is an (initial) `state` and a resulting `states'` on line 3. They also make use of record wildcards to define the accessors. For the outgoing buffer, `vvmOutgoing` and `vReadOutgoing` of line 3 are accessors of `ToOutgoingBuffer{..}` on line 1 whereas `vvmToRing` of line 3 is an accessor of `FromOutgoingBuffer{..}` of line 1.
For the incoming buffer, `vvmFromRing` and `vReadIncoming` on line 3 are accessors of `ToIncomingBuffer{..}` of line 1, while `vvmIncoming` of line 3 is an accessor of `FromIncomingBuffer{..}` of line 1.

### 6.4.2 The Controller

As explained in 2.3, the properties of an SDF graph are that they consume and produce a fixed amount of tokens. The controller checks this amount of tokens so that it takes the pressure off the function.

```
1  funcCtrl Setup{..} ToFuncCtrl{..} = FromFuncCtrl{..}
2    where
3      checkS                          = case mvvmToRing of
4                                          Nothing  -> False
5                                          (Just v) -> validCheck'' (resize <$> amountS) v
6      checkR                          = validCheck'' (resize <$> amountR) vvmIncoming
7      mvvmFromRing  | checkR          = Just (selector (resize <$> amountR) vvmIncoming)
8                    | otherwise       = Nothing
9      vvmOutgoing   | checkS          = case mvvmToRing of
10                                         Just v -> v
11                                         _      -> repeat (repeat Nothing)
12                   | otherwise       = repeat (repeat Nothing)
13     vReadIncoming | checkR && bRead = resize <$> amountR
14                   | otherwise       = repeat 0
```

***Listing 18:*** *Clash Controller implementation*

#### Production Control

Listing 18 shows the implementation of the controller. On lines 3-5 we check the number of tokens/messages produced by the actor. There are two possibilities, the function offers Nothing, see line 4, so there are not enough messages. Alternatively, on line 5, the function offers a `Just ...`, we then check if there are enough messages produced. If enough messages are provided, lines 9-11 will take the messages out of the `Maybe` type, so we deliver the right type to the output buffer. If we do not have enough messages, we deliver `Nothing`s to the output buffer, see Line 12.

#### Consumption Control

On line 6, we check if the incoming buffer has enough tokens/messages. After checking if there are enough messages in the incoming buffer, we still have to be sure that we do not provide the actor too many messages at once. That is why we only select, on line 7, the number of messages that the actor consumes. If there are not enough messages, we send Nothing, see line 8.
On Lines 13 and 14 the controller gives a signal to the buffer of how many messages it has to remove from the different input FIFOs. Messages are only deleted if there are enough messages in the input buffer and if the actor gives permission, through `bRead` of line 13. Otherwise, see line 14, zero messages will be deleted. `bRead` is an accessor of `ToFuncCtrl{..}`

### 6.4.3 The router

The router has two main tasks, namely to check if the messages coming from the ring belongs to the FPGA, and it must determine which message to put on the ring. The router can be set up to hijacking slots, use a credit-ring or both.

#### Message for the Ring

First, we are going to figure out from which output buffers we can place a message on the ring. The outgoing buffer gives us the outputs of the different FIFOs, with a width of `s` and length of `sd`, see Figure 6.6a. The router has to choose from which FIFOs it can put messages on the ring.

```
1 a = checkDest <$> sIds
2 b = validForRingCheck vvmToRing
3 c = validForRingCheck vvmCredits
4 v = zipWith3 (\ x y z -> x && y && z) a b c
```

***Listing 19:*** *Routing conditions*

In Listing 19, we check for three conditions.

1. If there is data in de FIFO, at least the ring size `sd`. That is the number of tokens that are allowed on the ring at once, see line 2.
2. Check if there are enough credit tokens. When there is no credit-ring used, this should be True, always, see line 3.
3. Which destinations are allowed on the ring? See line 1.

The result, on line 4, is a list of Booleans `v`, that tells us to which actors we can send a message.

```
1  checkDest destinationId =
2    case (modus, cRingHop) of
3      (IncreasingWithHijack, EmptySlot _) ->  (a && b) || (b && c) || (c && a)
4                                                where
5                                                  a = destinationId  <= slotId'
6                                                  b = slotId'        <= myId
7                                                  c = myId           <  destinationId
8      (DecreasingWithHijack, EmptySlot _) ->  (a && b) || (b && c) || (c && a)
9                                                where
10                                                 a = destinationId  <  myId
11                                                 b = slotId'        <= destinationId
12                                                 c = myId           <= slotId'
13     (_, Invalid)                        -> False
14     _                                   -> slotId'  ==  myId
```

***Listing 20:*** *Ring (Hijacking) conditions*

This third point is implemented in Listing 20 and has four modes, as explained in Chapter 6.1.
The listing displays a function, `checkDest` on line 1, that is located in the router. To this function, we give a destination ID. From this ID we want to know if it is a valid destination concerning the incoming slot. The function returns a Boolean value True or False. The check is done for all possible destinations. The destinations are the actors to which the edges of the dataflow graph lead.
There is a possibility, shown on line 13, that the message to the router is `Invalid`. Invalidity is possible if the ring hop needs more time than one clock cycle, for example, if it needs time to (de)serialise messages.
When the modus, on line 2, is `IncreasingWithoutHijack`, this means that it is a ring where hijacking is not allowed and, therefore, the slotID `slotIdRingHop` coming from the ring hop must be equal to the own id ( `myID` of line 14. The same goes for mode `DecreasingWithoutHijack`, that is a credit-ring where hijacking is not allowed.

*Hijacking*

Another possibility, on line 3, is that the mode is `IncreasingWithHijack` . This is a routing mode where, under certain conditions, an FPGA can hijack the slot ID of another FPGA. This hijacking does not benefit the worst-case execution time but can decrease the latency. A requirement is that the message, `cRinghop` of line 2, coming from the ring, is `Empty`, it must also meet one of the following conditions:

- 
- $myID < Destination \leq slotID$
- $slotID \leq myID < Destination$

The previous rules are implemented in Listing 20, on lines 3-7.

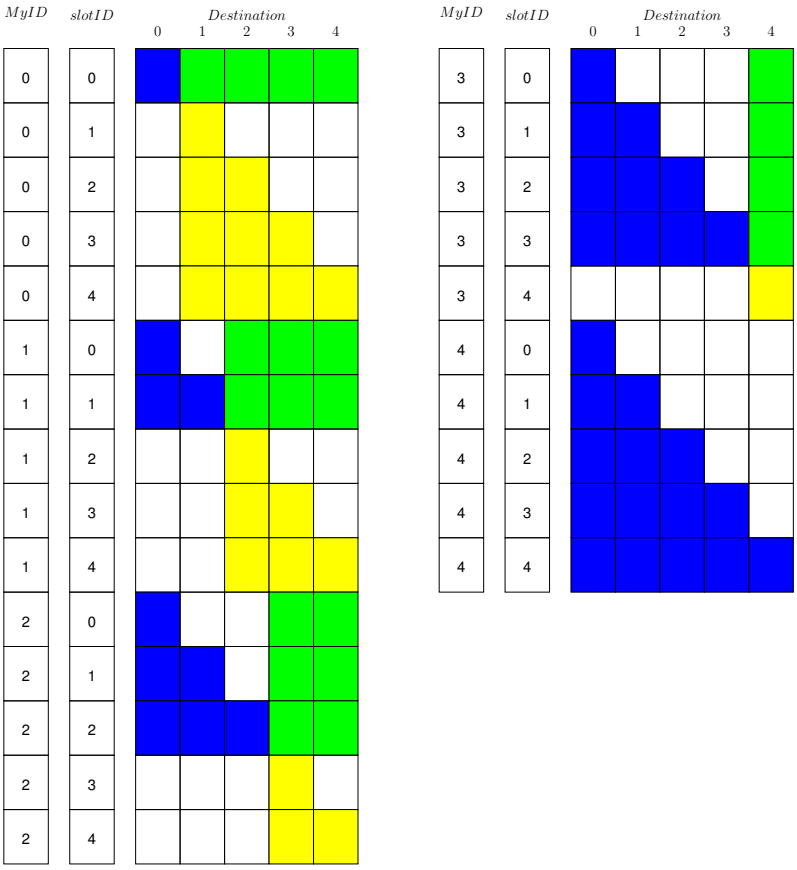



**Figure 6.7:** Hijacking

Figure 6.7 shows the setup rules, with modus `IncreasingWithHijack`, with five destination FPGA (0 - 4), the column myID gives the id of the FPGA, the slot id shows the ID of the slot, coming from the ring. Depending on the destination, we can decide if a message is allowed on the ring. If a message is allowed, that box is coloured. The different coloured boxes correspond with the rules shown before.

The last modus is modus DecreasingWithHijack. This modus is the same as modus `IncreasingWithHijack` except that it is used for the credit-ring. It is a ring where the messages are not sent to a succeeding FPGA ID but a previous decreasing FPGA ID. The conditions for hijacking, for the credit-ring, and an example can be found in Appendix B, the same conditions can be found in Listing 20, on lines 8-12.

*Round-Robin*

We want to distribute the message of the different FIFOs of the outgoing buffer equally on the ring, where we use every slot available. Therefore, we have to search the FIFOs for a message to place on the ring.

```
1  rr4  pointer validList     = (pointer' , out)
2    where
3      a                      = imap       (>=) (replicate (lengthS validList) pointer)
4      b                      = a          ++   (not <$> a)
5      c                      = zipWith   (&&)  b (validList ++ validList)
6      idx                    = elemindex True  c
7      out                    = resize    <$>  (mod <$> idx <*> Just (snatToNum (lengthS validList)))
8      pointer' = case out of
9                   (Just x) -> if x >= maxBound  then  minBound  else x + 1
10                  Nothing  -> pointer
```

***Listing 21:*** *Round-Robin implementation*

To do this, we implemented Round-robin, see Listing 21. The function `rr4` on line 1, needs a Boolean list, with all valid destinations ( `v` of Listing 19 or `validList` on line 1) and an (initial) `pointer` of line 1. The pointer is used to indicate the starting position in the list. Initially, the pointer starts at 0. The function returns an index that refers to a FIFO of the outgoing buffer. It also returns the position from where the pointer should begin at the next clock cycle.

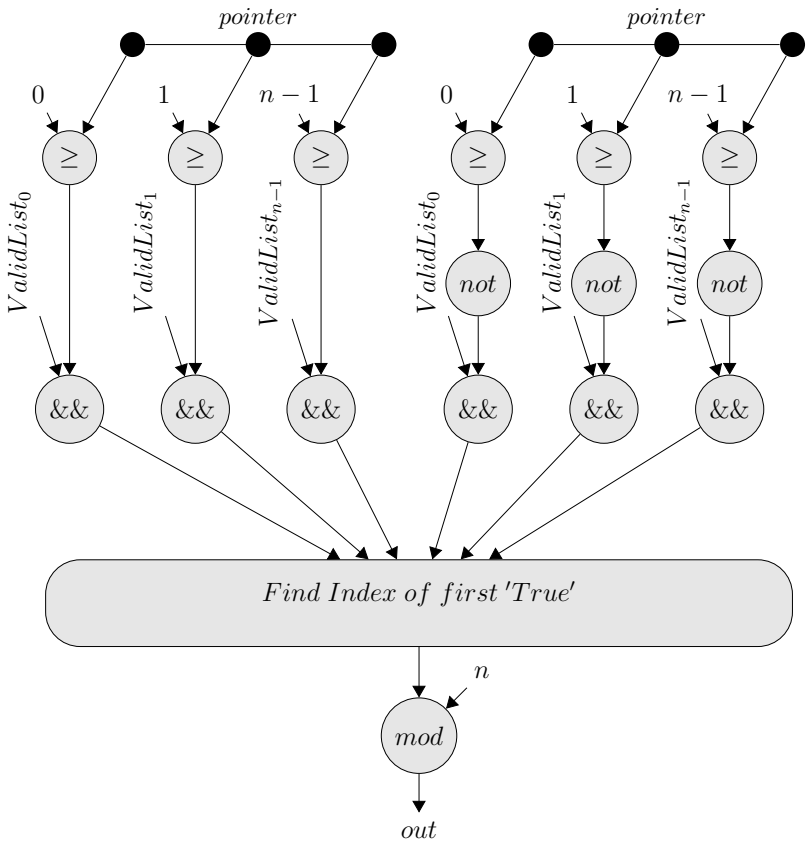
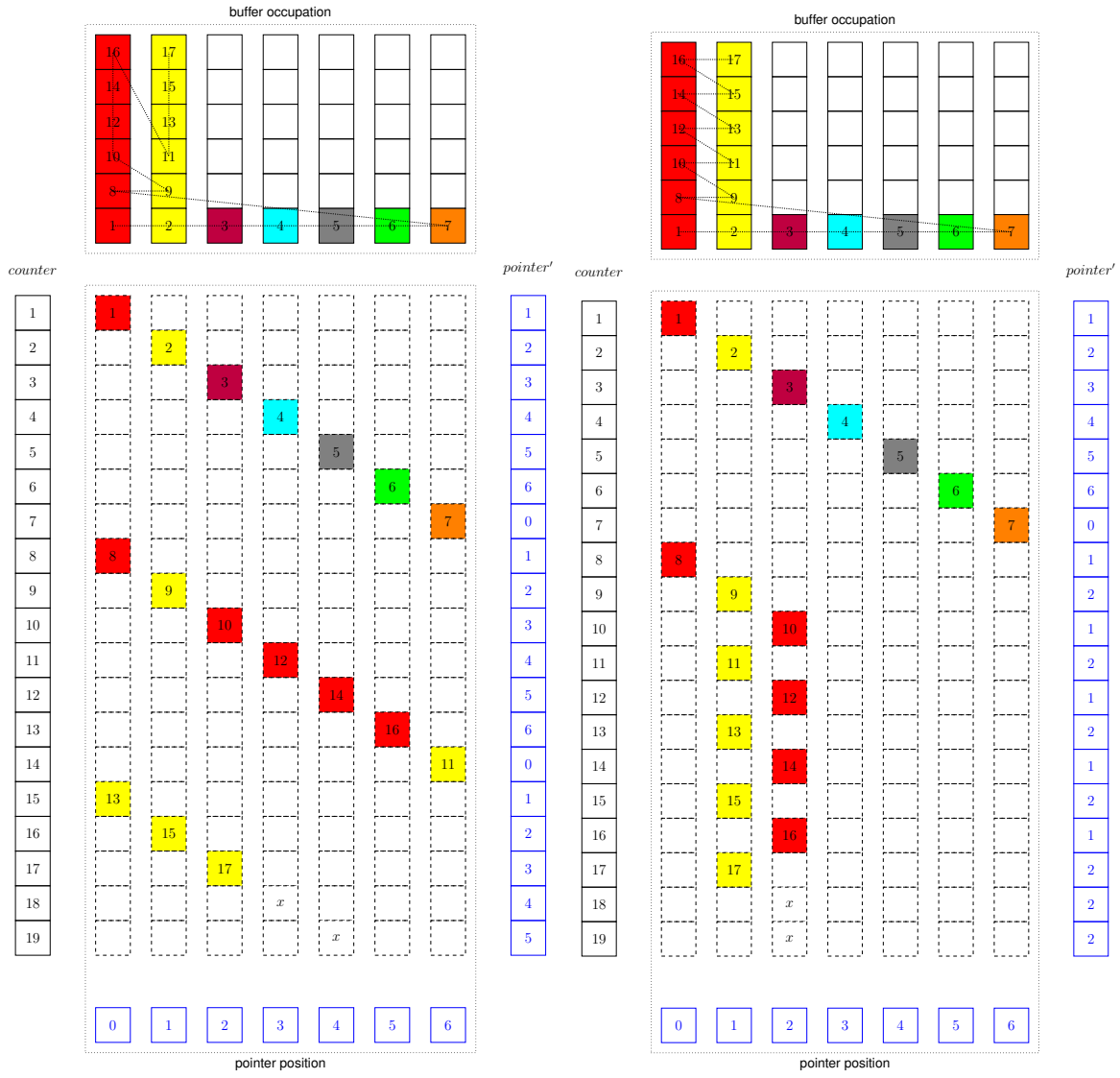



**Figure 6.8:** Round-Robin index selector

This round-robin function, see Listing 21 and Figure 6.8, is implemented as follows, the pointer is replicated and compared to a list of increasing integer numbers, equal to the length of the `validList` on line 3. When the integer number is greater or equal to the pointer, the result is True, otherwise False, see line 3. We make this pointer comparison twice, whereby the second part, on line 4, is inverted and added to the first part. The resulting list that is twice the size of the initial list, we zip with twice the `validList` on line 5.
The inversion and adding are done, so we have a list with the first True value starting at the index of the pointer. On line 6 we, search for the first True value. This result results in a Maybe index value, but due to the second added list, this index can lay outside the scope of the possible index. To get the output, that is of type Maybe index. On line 7, we take the modulus of the found index and the length of the list. It results in output `out` of type `Maybe index`. The index refers to the index of a FIFO in the outgoing buffer.

*Selecting Pointer*

The only thing to do now is to determine what the next pointer will be, see `pointer'` on line 8. For this next pointer selection, we considered two options.
We could add 1 to the pointer. Thus, `pointer' = pointer + 1` and loop around in case the max bound is reached. Alternatively, we could make it depended on the found index, and if there is not a message available to send on the ring, we let the pointer as it is.




***(a)*** *Possibility 1* ***(b)*** *Possibility 2*

**Figure 6.9:** Round-Robin, pointer update examples

Next, we will explain the two different possible pointer updates with the help of the two images of Figure 6.9.
The upper part of the images shows de buffer occupation of the outgoing buffer. The

columns are the FIFOs, and thus the buffer has a width of seven, this would mean that the dataflow graph has seven output edges.
The marked boxes(yellow) say that there is a message in that FIFO. The line running through it indicates the order in which messages are sent over the ring, where we send one message at the time over the ring.
The (blue) part at the bottom of the images shows the indexes of the pointer (0- 6). The column on the left side, named counter, counts the amount of time a valid slot is found.
In the middle part, when the counter is 1, we see that the pointer is set to 0, and that message one is offered to the ring. The new pointer is 1. This new pointer is indicated in the right column `pointer'`(blue). Now the pointer is 1, and message two is offered to the ring. The pointer is updated to '2'. Message three is provided, etc.
After message ten is offered, something interesting happens. In Figure 6.9a we see that pointer is moved to the next position and starts again at the front. By sending this sequence of messages, the left FIFO has precedence over the other FIFOs, in this case, the second column.
If instead, of just adding one we make the pointer dependent on the index that contains the message we want to send, we get to see the pointer updates as in Figure 6.9b, we do this by adding one to the found index.
In the upper part of Figure 6.9b, we see that the messages are sent horizontally, without preference for one of the FIFOs. Therefore, this order is chosen and implemented in Listing 21, line 9.

### Routing of the Router

We know now which message is being sent to the ring. We also know which message is coming from the ring, so we are going to route the messages in the right direction.

```
1  (cRing'  , toBuffer ,update ) = case (cRingHop , rrOutC ) of
2    (EmptySlot _, EmptySlot _        )-> (EmptySlot slotId', (N  , repeat N), N  )
3    (EmptySlot _, ContentSlot _ _ d v)-> (rrOutC           , (N  , repeat N), J d)
4    (ContentSlot _ s b c , EmptySlot _   )
5                         | b == myId   -> (EmptySlot slotId', (J s, c       ), N  )
6                         | otherwise   -> (cRingHop         , (N  , repeat N), N  )
7    (ContentSlot _ s b c,  ContentSlot _ _ d v)
8                         | b == myId   -> (rrOutC           , (J s, c       ), J d)
9                         | otherwise   -> (cRingHop         , (N  , repeat N), N  )
10   _                                   -> (Invalid          , (N  , repeat N), N  )
```

***Listing 22:*** *Routing Conditions*

The implementation of the routing in the shown in Listing 22. The `N` and `J` are

short for Nothing and Just and are written, so it fits on a page in this report.
There are two content input signals, one from the function/node, `rrOutC` on line 1, this is the message found with the Round-robin of Chapter 6.4.3 and one from the ring, `cRingHop` on line 1. Both have the option to be `Empty`, contain `Content` or be `Invalid`. Together they form five possibilities. Namely, both are empty, see line 2, both contain content, see line 7, one contains content, and the other is empty, see lines 3 and 4, or the message from the ring is `Invalid`. If it is an Invalid then there is no valid slot. Thus, sending to the ring is also not allowed.
If the message from the ring contains content, we have to check if the message belongs to this FPGA. We do this by comparing the destination address in the slot with its "own" address, `myId` on line 5 or 8. If it is not equal to the own ID, one of the other options is invoked on lines 6 or 9 respectively. Depending on these states we obtain, on line 1, the new message from the ring `cRing'`, the new message to the incoming buffer `toBuffer`. If we did read from the output buffer, we also tell from what FIFO we did read, this `update` on line 7.
The cRing' is the new message that will be placed on the ring, due to checks as shown in Listing 19 and explained in Chapter 6.4.3, we know that the message is allowed in the slot.

### 6.4.4 The Ringhop

The ring hop is the slot of the nebula Ring. Or slots if we want to use multiple consecutive slots.

```
1 ringHop  rhState  ToRingHop{..} =   (rhState', FromRingHop{..})
2   where
3     cRingHop =  last rhState
4     rhState' =  cRing +>> rhState
```

***Listing 23:*** *Ringhop implementation*

We implemented the ringhop as a Mealy machine in Clash, see Listing 23.
The `rhState`, on line 1, is the (initial) state. This is a vector of slots. `ToRingHop{..}` is the input and has the accessor `cRing`, see line 1.
The `cRing` is, on line 4, added to the head of the vector of slots. The last element is shifted out. This creates the new state `rhState'`. The last element of the 'old' vector `rhState`, we use as output, see line 3. `cRingHop` on line 3, is an accessor of `FromRingHop{..}`.

## 6.5 Conclusion Implementation

There are some important aspects of the Clash implementation. The controller checks if there are enough messages in the input buffer. If it is equal to the consumption rate of the dataflow graph, passes them on to the function. It also checks if the production of messages is equal to the production rate of the dataflow graph.
The FIFOs are implemented in such a way that they accept and provide multiple messages at the same time. Due to the fixed hardware implementation, these accept and provide rates are always the same. For the implementation of the buffers, the FIFOs are placed side by side. It implies that due to higher-order functions, the separate FIFOs must be equal to each other. By using the `Maybe` type, we can still vary the acceptance of messages.
We have implemented the router in such a way that it can also be used as a credit-ring and allows hijacking of slots. The order in which messages are injected into the ring has no preference for a particular FIFO. It is also possible to put multiple messages/tokens on the ring.
We connected the buffers, router, ringhop and controller using custom data types and record syntax. It ensured that the different elements are ”simply” connected.
For connecting FPGAs, we also made a custom data type, `RingContent`.

# Part III

# Analysis and Simulation Results

# Chapter 7

# Reconversion

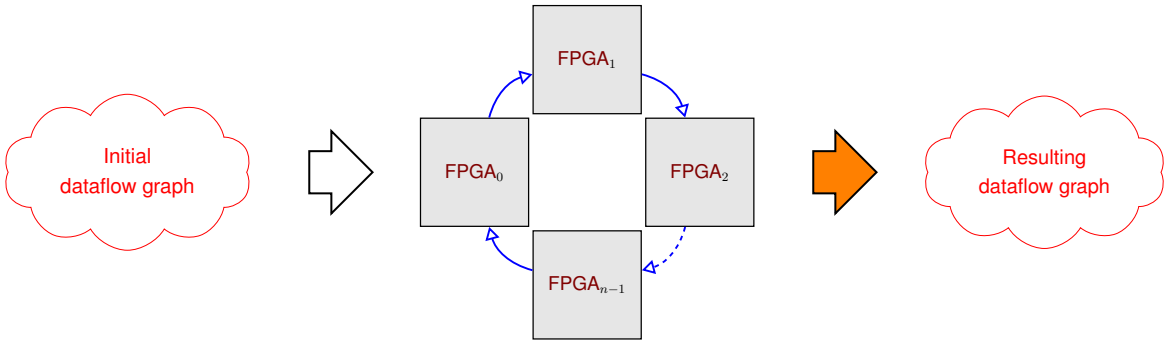



**Figure 7.1:** Designflow: Ring topology to resulting dataflow graph

This chapter shows how to model timing behaviour of the hardware, and get a modified initial dataflow graph, see Figure 7.1 For this purpose, the communication paths are added as identity actors to the initial dataflow graph. With this new model, we answer the following question in this and the next chapter:

*How can we model the temporal behaviour of the design, analyse the communication and guarantee deterministic behaviour?*

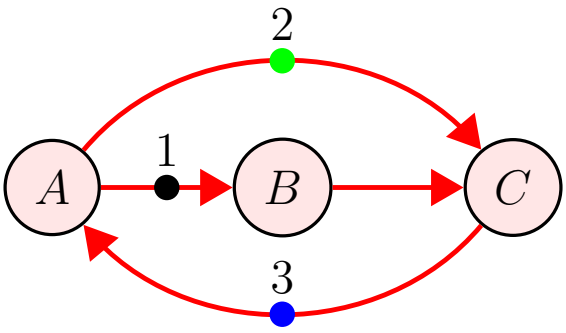



**Figure 7.2:** Three Node, dataflow graph example

## 7.1 Communication Path

In dataflow, an edge between two actors is represented by a FIFO and would not have any firing time. The edge from A to B of the example in Figure 7.2, would then look like Figure 7.3, where the edge is a FIFO.
In our hardware design, an edge does not just consist of a FIFO buffer but consists of various elements that form a communication path from one actor to another. Those elements are a *controller* of the sending actor, an *outgoing buffer*, a *router*, a slot in the *ringhop*, possibly several more *ringhops* and *routers*, then an *incoming buffer* and finally, a *controller* of the receiver. Figure 7.4 shows the communication path containing the various elements of the different edges of Figure 7.2.

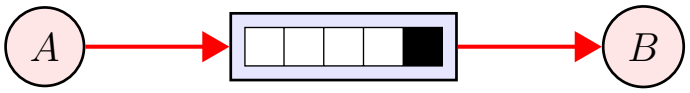



**Figure 7.3:** Edge representation

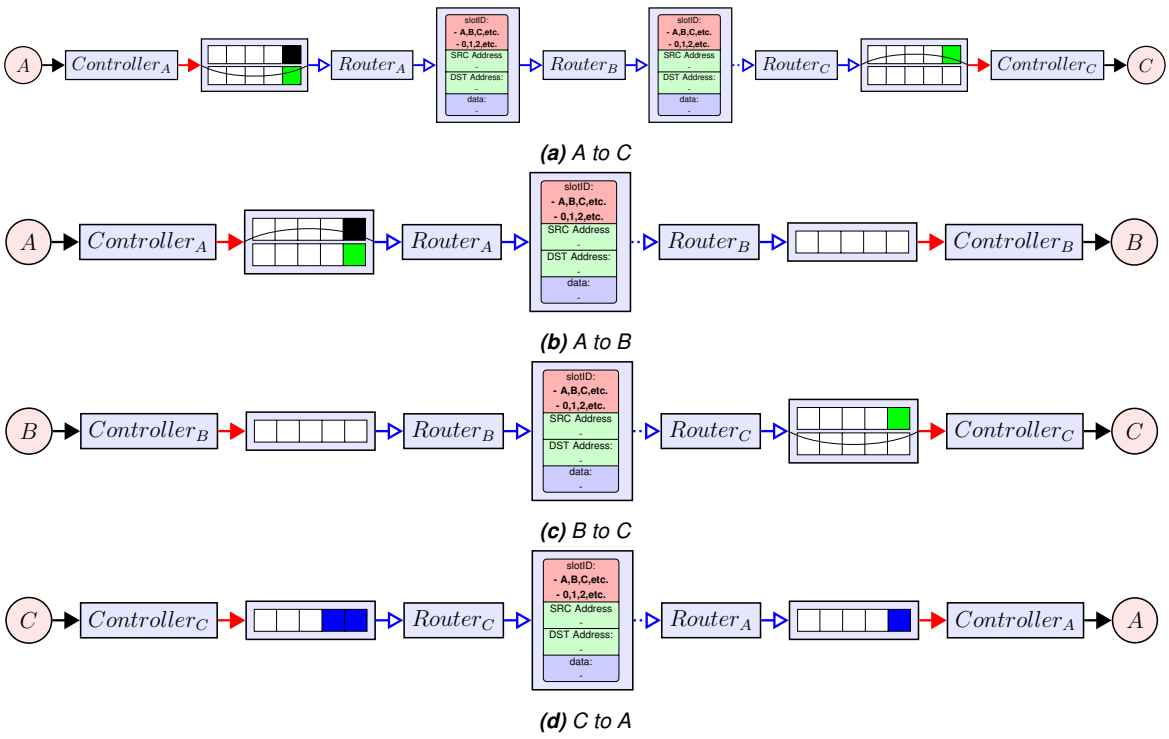



**Figure 7.4:** New edge representation

## 7.2 Identity Actors

In our design, the communication path takes time. We model this as a new actor. The new actor, with one input and output edge, replaces every edge of the original dataflow graph. This new actor does not change anything to the tokens/messages, it only has a firing time, and an equal consumption and production rate, equal to the amount of message allowed in one slot. That is why we call this the identity actor. We consider the in/output buffers, in the identity actors, as an element that takes time.

The new edges represent FIFO buffers that do not take time. On these new edges, we place the initial tokens. The example of Figure 7.2, including the identity actors and tokens, is shown in Figure 7.5. All tokens on the input edge are consumed are produced concurrently.

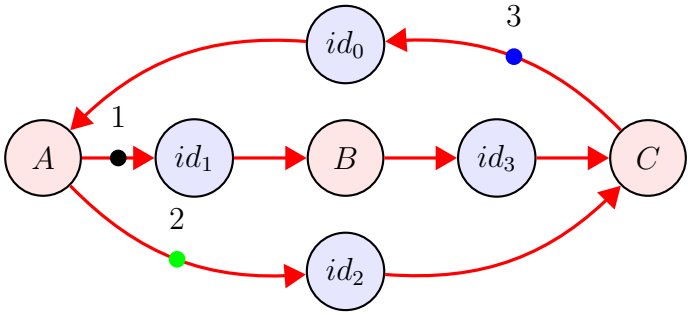



**Figure 7.5:** Resulting dataflow graph: Three node, dataflow graph example

## 7.3 Conclusion Reconversion

We modelled the communication path from one actor to another as a new identity actor, where an actor is added to each edge of the initial dataflow graph. This results in a dataflow graph on which the user can perform a post-analysis for the communication time between FPGAs.

# Chapter 8

# Timing Analysis

In this chapter, we show the formulas to calculate the WCET as firing time of the identity actors added to the initial dataflow graph, and we answer the question from Chapter 7, namely,

> *How can we model the temporal behaviour of the design, analyse the communication and guarantee deterministic behaviour?*

## 8.1 Calculation Introduction

For calculating the communication time, we take the time from entering the output buffer from one node up to and including the outcome of the outcoming buffer of the receiving node, this means we take the WCET as firing time of the identity actor.
For calculating the WCET between the different actors, we made two calculations. We made two calculations because, for the first calculation, we calculate the time until all messages/tokens in de buffer are injected into the ring and received at their destination. For the second calculation, we wait until the tokens/messages of a single FIFO in the outgoing buffer are sent and received at their destination.

For calculating the WCET, we need to know the different argument used in de equations;

- $B_i$ Time in the incoming buffer.
- $B_o$ Time in the output buffer.
- $E$ The number of outgoing edges or the amount of FIFOs in the outgoing buffer.
- $F$ The maximum amount of tokens in the FIFO. This not the total amount in the output buffer.
- $H$ The number of hops/slots from the source to the destination.
- $M$ The maximum amount of tokens in the output buffer.
- $N$ The number of hops/slots on the ring.
- $sd$ The number of messages send over the ring at the same time/ the number of tokens consumed or produced concurrently, see Figure 8.1.
- $T$ The time it takes for the token to hop from one FPGA to the other.
- $W$ Worst-Case Execution Time.

A condition for the equations is:

$$\forall \frac{F}{sd} \in \mathbb{N}$$

This means that for all FIFOs the maximum amount of tokens in one FIFO must be dividable by the number of messages on the ring at the same time. This is to prevent the new identity actors, who have sd as consumption and production rate, from deadlocking. The result is a natural number. If sd is one, there would not be an issue.

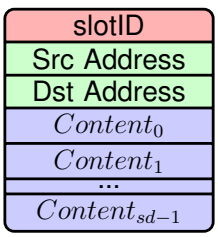



**Figure 8.1:** New extended slot, with sd content places

### 8.1.1 Calculation 1

This calculation assumes there is a maximum of tokens in the output buffer of an FPGA. The time it takes before the last message in the output buffer is received at its destination node, we can interpret as the firing time, of the identity actor. Something we need to know is the maximum amount of tokens that could be on every edge of the initial dataflow graph. With the help of the SDF3 tool from [29], we can calculate this maximum[1].

For this calculation, we use Equation 8.1

$$W_1 = B_o + NT - 1 + \frac{NT(M - sd)}{sd} + HT + B_i \tag{8.1}$$

Equation 8.1 is build up as follows. First, we have to wait before a message that has entered the output buffer is available for the router to send on the ring, this takes $B_o$ clock cycles. If we are unlucky, we just missed our own slot and have to wait $NT - 1$ clocks cycles for the first valid slot is available before we can send the first $sd$ messages over the ring. Now we have to send the rest of the messages over the ring. We can do this every time there is a slot available, that is ever $NT$ clocks cycles. We have to do this a total of $M - sd$ times because that is the number of messages left in the buffer to send[2]. We can divide this by $sd$ because that is the number of messages that can be sent over the ring at the same time. After this we need the hops, $H$, a message must take to get to its destination. We have to multiply this by $T$. This $T$ is at least one because we need a position for the slot [3]. Eventually, we have to add the time the message is in the incoming buffer, $B_i$.

After some algebra Equation 8.1 results in Equation 8.2.

$$W_1 = B_o - 1 + \frac{NTM + HTsd}{sd} + B_i \tag{8.2}$$

In our implementation the time of the buffers $B_o$ and $B_i$ are both one, this results in Equation 8.3.

$$W_1 = \frac{NTM + HTsd}{sd} + 1 \tag{8.3}$$

[1] Or at least get an indication.

[2] We already send the first $sd$ messages after $NT - 1$ clocks.

[3] $T$ can be greater than one, e.g. when the messages must be serialized between the nodes.

### 8.1.2 Calculation 2

With this calculation, we only want to send the message from one FIFO the destination belonging to that FIFO. For this calculation, we use Equation 8.4.

$$W_2 = B_o + NT - 1 + NT(E-1) + \frac{ENT(F-sd)}{sd} + HT + B_i \tag{8.4}$$

The equation is build up as follows. $B_o$, $NT-1$ and $HT+B_i$ of Equation 8.4 are equal to those parts of Equation 8.1. The difference lies in $NT(E-1)$ and $\frac{NTE(F-sd)}{sd}$. Where $NT(E-1)$ is the time between the first message is allowed on the ring[4] and the first message of the desired destination. After that, we have to wait $NT$ clocks for the next available slot. Because of the distribution, see Chapter 6.4.3 we have to multiply this by the amount of outgoing edges $E$ of the source node. In turn, we want to send the remaining $F-sd$ Messages over the ring, so we multiple this with $F-sd$. We can divide this by $sd$ because that is the number of messages we can send over the ring at the same time

After some algebra Equation 8.4 results in Equation 8.5.

$$W_2 = B_o - 1 + \frac{NTEF + HTsd}{sd} + B_i \tag{8.5}$$

As said before the time of $B_o$ and $B_i$ are both one, and this results in Equation 8.6.

$$W_2 = \frac{NTEF + HTsd}{sd} + 1 \tag{8.6}$$

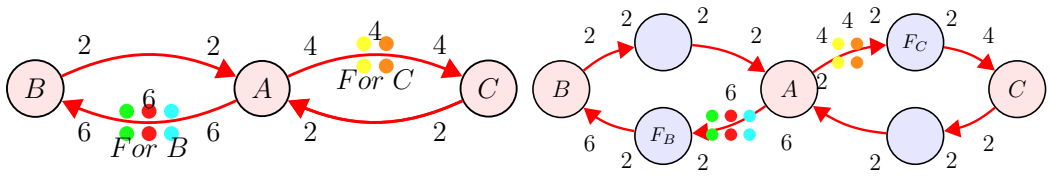
[4] after $B_o + NT - 1$ clock cycles.

### 8.1.3 Example calculation 1 and 2

In this section, we explain the Equations 8.1 and 8.4 using an example. We start with a general explanation and then for the equations specifically.

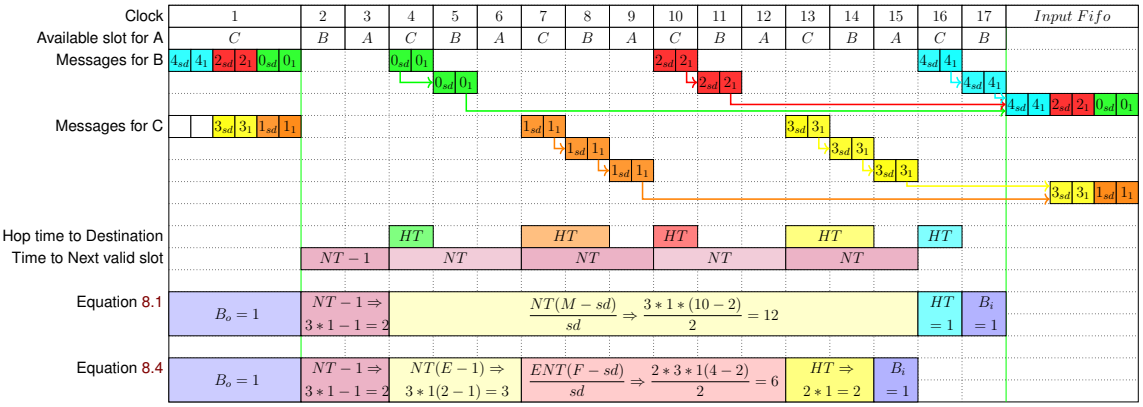
#### In General

In Figure 8.2 we see that actor A has two output edges and sends a total of ten messages to actors B and C. Because in the example *sd* is two, we put two messages on the ring at the same time. That is why the messages are divided into five parts. The time the messages from A are in the output buffer, $B_o$, is one clock cycle. We see that after one clock cycle the available slot is C. Therefore, we have to wait for $NT - 1$ clock cycles, before the available slot is A, and we can put first messages, $0_{sd}$ and $0_1$ on the ring. After that we can inject every $NT$ clock cycles messages on the ring, that is every three clock cycles. It then takes $HT$ time before the messages reach their destination and are put in the input FIFO of the recipient, depending on the number of hops this is one or two clock cycles, in the example.




***(a)*** *Initial dataflow graph*

***(b)*** *Resulting dataflow graph*




***(c)*** *From A to B and C*

**Figure 8.2:** Timing example

### Calculation 1

The time from the first message until the last message is $\frac{NT(M-sd)}{sd}$, where the number of Slots $N$ is three. This is equal to the number of FPGA in the system, namely A, B and C. The hop time $T$ is one clock cycle because there is nothing to (de)serialized. The number of messages $M$ is ten and $sd$ is two, which is the number of messages placed on the ring at the same time. After 15 clock cycles, we can put the last messages $4_{sd}$ and $4_1$ on the ring, after one hop it arrives at B, after which the last message has arrived. It takes another clock cycle in the Input FIFO of B before the messages are offered to actor B, so the total is 17 clock cycles.

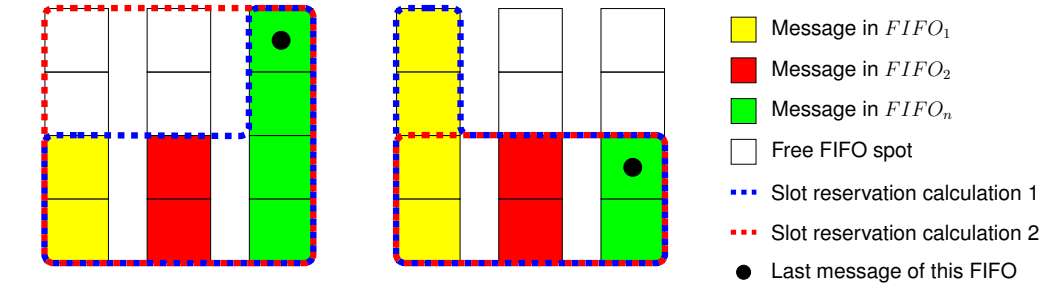
### Calculation 2

For calculation 2 and thus Equation 8.4, we only look at the number of messages in a FIFO. If we then look at the messages intended for C, we see that we want to send four messages. We still have to wait for $NT-1$ clock cycles before we can put the first messages, $0_{sd}$ and $0_1$, on the ring, but these messages are for B. Therefore, we have to wait for another $NT(E-1)$ clock cycles before we can put the first messages for C on the ring. After six clock cycles, we can put the first message $1_{sd}$ and $1_1$ on the ring. After that ,in case of a valid slot, we inject, once every E edges, a message for C on the ring. The total time is represented by $\frac{ENT(F-sd)}{sd}$. Where $F-sd$ is two, and the number of messages is what is still in the FIFO. After 12 clock cycles, the last message for C is put on the ring. After that, the message has to take another two hops to arrive at C. It then takes another clock cycle in the Input FIFO of C before the messages are offered to actor C. The total is 15 clock cycles.

If we would perform the same calculation on the messages from FIFO B, (the figure is then no longer representative). The result is 20 clock cycles, which is more than the earlier 17 clock cycles and thus indicates that both have advantages and disadvantages.

### 8.1.4 Final WCET

Calculation 1 and calculation 2 both have there pros and cons. With both formulas, we are looking for the amount of slots needed to inject (all) messages of the output buffer into the ring. With calculation 1, Equation 8.3, the number of slots reserved is equal to the maximum number of messages in the output buffer. We neglect the empty spots in the buffer where there are no messages.
With calculation 2, Equation 8.6, it is as if the calculation draws a rectangle around/through the buffer and then counts all FIFO spots within this rectangle as slots reserved. The width of the rectangle is equal to the number of FIFO edges in the buffer. The length of the rectangle is equal to the maximum number of messages in that FIFO for which we want to know how long it will take for the last message of that FIFO to be injected into the ring. With calculation 2 it is possible to make an excessive calculation if there are fewer messages in all other FIFOs. Therefore taking the minimum of the two formulas is sufficient.
In Figure 8.3 we see two examples of possible buffer occupations. Both buffers have a total of eight messages in three FIFOs. In Figure 8.3a, we see that if we calculate the slots with calculation 1, we reserve eight slots for the eight messages in the buffer. For calculation 2, we reserve 12 slots, before the last message is injected into the ring. Calculation 1 is, therefore, the better calculation in this case, because calculation 2 reserves also four slots for the free FIFO spots. In Figure 8.3b, we see that if we calculate the slots with calculation 1, we reserve eight slots for the eight message. For calculation 2, we reserve, before the last message injected into the ring, six slots. Calculation 2 is, therefore, the better calculation in this case, because calculation 1 also reserves two FIFO spots [5], that are used by later slots.




***(a)*** *Calculation 1 = 8 slots, Calculation 2 = 12 slots*

***(b)*** *Calculation 1 = 8 slots, Calculation 2 = 6 slots*

***(c)*** *Legend*

**Figure 8.3:** Buffer occupation, with reserved slots for both calculations

In the final equation, see Equation 8.7, we take the minimum of both equations.

$$W_m = Min(\frac{NTM + HTsd}{sd} + 1, \frac{NTEF + HTsd}{sd} + 1) \tag{8.7}$$

where,

$$\forall \frac{F}{sd} \in \mathbb{N}$$

[5] The yellow messages outside the red line.

## 8.2 Conclusion Timing Analysis

Calculating the WCET as firing time for the identity actors, added to the resulting dataflow graph, we can guarantee deterministic behaviour and analyse this. For this purpose, we made two equations. For the first calculation, we are entirely dependent on the maximum number of messages in the output buffer. With the second calculation, we only depend on the messages in one FIFO, to a certain depth. This depth is multiplied by the number of edges, to get the amount of own slots. By taking the minimum of both calculations, we get actual firing time. This firing time per identity actor is represented as an SDF graph and, therefore, is equal to the WCET. Some messages arrive earlier. The identity actor can therefore also be expressed as Cyclo-Static DataFlow (CSDF) actor, but this is something for future work.

# Chapter 9

# Simulation Results

This chapter shows the results of implemented dataflow graphs and compares it with the calculated time of Chapter 8. We want to see if the firing time of identity actors correspond with te WCET formulas found in Chapter 8. Therefore, we answer the following question:

> *How do simulation results correspond to analysis results concerning timing?*

We start with an explanation of the simulation setup. Then we calculate the WCET of some examples. Next, we discuss the Clash simulation results. After that, we compare the results and conclude the chapter.

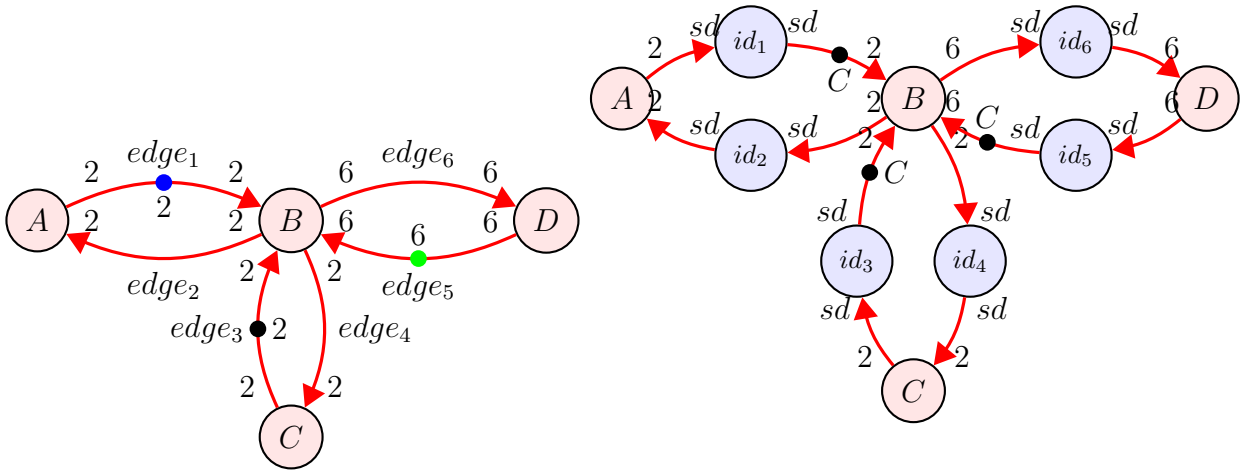
## 9.1 Simulation Setup

$$
\begin{array}{c}
\begin{array}{cc} & \overbrace{\phantom{00}}^{A}\ \overbrace{\phantom{00}}^{B}\ \overbrace{\phantom{00}}^{C}\ \overbrace{\phantom{00}}^{D} \end{array} \\
\begin{array}{l} edge_1 \{ \\ edge_2 \{ \\ edge_3 \{ \\ edge_4 \{ \\ edge_5 \{ \\ edge_6 \{ \end{array}
\begin{bmatrix} 2 & -2 & 0 & 0 \\ -2 & 2 & 0 & 0 \\ 0 & -2 & 2 & 0 \\ 0 & 2 & -2 & 0 \\ 0 & -6 & 0 & 6 \\ 0 & 6 & 0 & -6 \end{bmatrix}
\end{array}
\quad
\begin{bmatrix} 2 & -2 & 0 & 0 \\ -2 & 2 & 0 & 0 \\ 0 & -6 & 6 & 0 \\ 0 & 6 & -6 & 0 \\ 0 & -2 & 0 & 2 \\ 0 & 2 & 0 & -2 \end{bmatrix}
\quad
\begin{bmatrix} 6 & -6 & 0 & 0 \\ -6 & 6 & 0 & 0 \\ 0 & -2 & 2 & 0 \\ 0 & 2 & -2 & 0 \\ 0 & -2 & 0 & 2 \\ 0 & 2 & 0 & -2 \end{bmatrix}
\quad
\begin{bmatrix} 2 & -2 & 0 & 0 \\ -2 & 2 & 0 & 0 \\ 0 & -6 & 6 & 0 \\ 0 & 6 & -6 & 0 \\ 0 & -2 & 0 & 2 \\ 0 & 4 & 0 & -4 \end{bmatrix}
$$

***(a)*** *Option 1* ***(b)*** *Option 2* ***(c)*** *Option 3* ***(d)*** *Option 4*

**Figure 9.1:** Topology matrices for different implementations

To simulate in Clash, we designed one dataflow graph, in which we adjusted the consumption and production rates, to do different tests. The different topology matrices, of the dataflow graphs, we called *options* and are shown in Figure 9.1.




*(a) Initial dataflow graph*

*(b) Resulting dataflow graph with identity actors*

**Figure 9.2:** Dataflow graphs of option 1

In Figure 9.2 are of option 1, the original and resulting dataflow graphs shown. The Clash implementation of option 1 actor B can be seen in Listing 24, as already explained in Chapter 6.1.

For the test, we test $edge_6$, because actor D is the last actor in the list "A, C, D". We choose the last actor, because that is where the pointer, in the router[1], points to last. In the meantime tokens/messages are sent to other actors over the ring. The different rates in the options ensure that, e.g. for option 1, six messages go to actor D, two to actor A and two to actor C. This means that there are more messages sent to D (6) than the total amount of messages sent to A and C (4). This corresponds with calculation 1, Equation 8.3.

With option 2, only two messages go to actor D and to actors A and C go two and six messages respectively. In this case, it is not necessary to take into account all messages. Because now we do not have to wait until all six messages for actor C have been sent. Waiting until two messages from actor A and C have been sent is enough. This corresponds with calculation 2, Equation 8.6.

We also have some simulations in which the ring size, sd, is set to 2. This is to show what happens when multiple messages are sent over the ring at the same time. We did a test where we enabled hijacking, and we also increased the hoptime to see what happens when the ringhop costs more than one clock cycle.

[1] see Chapter6.4.3

### 9.1.1 Clash Setup

In Listing 25, we see the initial states of the FPGA with id B. On lines 1-3, we see the tokens/messages in the incoming buffer. We see that there are multiple `Just 0`s offered to B. The amount of `Just 0`s are exactly equal to the consumption rate of actor B. So actor B can start firing immediately.
The firing time must not be a factor and, therefore, it is set to 0.

On line 5, we see there is nothing in the initial input of the output buffer. On line 6, we see the initial pointer of the router. This pointer is 0 and means that it points to A of the list of " A, C, D", this is a list to where actor B has an edge. On line 7, we chose the first slot id B. This id equals the identifier of actor B. We do this because after one clock cycle the slot, with id B, is shifted one place. Which is exactly equal to the minimum time the tokens are in the output buffer. This makes sure that we have exactly missed our 'own' slot, which allows us to compare the WCET with the calculations from Chapter 8.

```
1  def_1 = D { myId    = 'B'                          --
2            , sIds    = 'A' :> 'C' :> 'D' :> Nil   -- edge 2 , edge 4 , edge 6
3            , amountS =  2  :>  2  :>  6  :> Nil   --
4            , rIds    = 'A' :> 'C' :> 'D' :> Nil   -- edge 1 , edge 3 , edge 5
5            , amountR =  2  :>  2  :>  6  :> Nil   --
6            , modus   =  IncreasingWithoutHijack   --
7            }
```

***Listing 24:*** *Option 1 actor B in Clash*

```
1  init_1 = NodeStates   { ibState     = ((replicate d2 (Just 0) ++ repeat Nothing) -- From A
2                                       :> (replicate d2 (Just 0) ++ repeat Nothing) -- From C
3                                       :> (replicate d6 (Just 0) ++ repeat Nothing) -- From D
4                                       :> Nil)
5                        , obState     = repeat (repeat Nothing)
6                        , rState      = 0
7                        , rhState     = EmptySlot 'B' :> Nil
8                        }
```

***Listing 25:*** *option 1 actor B Initial states*

### 9.1.2 Calculation Results

Of all the edges and options we have calculated in Table 9.1, for Equations 8.4 and 8.1, the WCET. For all options, the ring size, sd, is one, and the hoptime is one clock cycle. Except for option 4 where the Ringhop time is seven clock cycles. The last column shows the minimum value as calculated with Equation 8.7. Because we are interested in $edge_6$, see Figure 9.2, we highlighted it. Table 9.2 is equal to Table 9.1, except weve now set the ring size sd to 2. We can see that the WCET is almost halved.

**Table 9.1:** Calculation results with ring size(sd) = 1

***(a)*** *option 1*

| Edge | Formula | | Min |
|---|---|---|---|
| | 8.6 | 8.3 | |
| 1 | 10 | 10 | 10 |
| 2 | 28 | 44 | 28 |
| 3 | 12 | 12 | 12 |
| 4 | 26 | 42 | 26 |
| 5 | 27 | 27 | 27 |
| 6 | 75 | 43 | 43 |

***(b)*** *option 2*

| Edge | Formula | | Min |
|---|---|---|---|
| | 8.6 | 8.3 | |
| 1 | 10 | 10 | 10 |
| 2 | 28 | 44 | 28 |
| 3 | 28 | 28 | 28 |
| 4 | 74 | 42 | 42 |
| 5 | 11 | 11 | 11 |
| 6 | 27 | 43 | 27 |

***(c)*** *option 3*

| Edge | Formula | | Min |
|---|---|---|---|
| | 8.6 | 8.3 | |
| 1 | 26 | 26 | 26 |
| 2 | 76 | 44 | 44 |
| 3 | 12 | 12 | 12 |
| 4 | 26 | 42 | 26 |
| 5 | 11 | 11 | 11 |
| 6 | 27 | 43 | 27 |

***(d)*** *option 4, hoptime:7*

| Edge | Formula | | Min |
|---|---|---|---|
| | 8.6 | 8.3 | |
| 1 | 36 | 36 | 36 |
| 2 | 190 | 358 | 190 |
| 3 | 190 | 190 | 190 |
| 4 | 512 | 344 | 344 |
| 5 | 71 | 71 | 71 |
| 6 | 351 | 351 | 351 |

**Table 9.2:** Calculation results with ring size (sd) = 2

***(a)*** *option 1*

| Edge | Formula | | Min |
|---|---|---|---|
| | 8.6 | 8.3 | |
| 1 | 6 | 6 | 6 |
| 2 | 16 | 24 | 16 |
| 3 | 8 | 8 | 8 |
| 4 | 14 | 22 | 14 |
| 5 | 15 | 15 | 15 |
| 6 | 39 | 23 | 23 |

***(b)*** *option 2*

| Edge | Formula | | Min |
|---|---|---|---|
| | 8.6 | 8.3 | |
| 1 | 6 | 6 | 6 |
| 2 | 16 | 24 | 16 |
| 3 | 16 | 16 | 16 |
| 4 | 38 | 22 | 22 |
| 5 | 7 | 7 | 7 |
| 6 | 15 | 23 | 15 |

***(c)*** *option 3*

| Edge | Formula | | Min |
|---|---|---|---|
| | 8.6 | 8.3 | |
| 1 | 14 | 14 | 14 |
| 2 | 40 | 24 | 24 |
| 3 | 8 | 8 | 8 |
| 4 | 14 | 22 | 14 |
| 5 | 7 | 7 | 7 |
| 6 | 15 | 23 | 15 |

***(d)*** *option 4: hoptime:7*

| Edge | Formula | | Min |
|---|---|---|---|
| | 8.6 | 8.3 | |
| 1 | 36 | 36 | 36 |
| 2 | 106 | 190 | 106 |
| 3 | 106 | 106 | 106 |
| 4 | 260 | 176 | 176 |
| 5 | 43 | 43 | 43 |
| 6 | 183 | 183 | 183 |

### 9.1.3 Clash Simulation Results

In Tables 9.4[2], 9.5[3] and 9.6[4], we see three results from different simulations. For other and elaborate results, see Appendix C. The results in Tables 9.4, 9.5 and 9.6 are explained by describing each column and row.

#### Column Description

The first and ninth columns show the clock cycles. The second column shows what is provided to actor B, so this is what is available in the various FIFOs of the incoming buffer. In the third column, we see what the function of actor B produces for the different receiving actors A, C and D. The fourth column indicates if the function of actor B has consumed tokens/messages. The fifth column indicates to what actor the ring sends a message. ”1,0,0” is to actor A, ”0,1,0” is to actor C and ”0,0,1” is to actor D. The sixth, seventh and eighth columns indicate what the output buffer offers the router to send over the ring. The tenth column shows which tokens/messages are offered to actor D. The eleventh column shows what the function of actor D produces. Column 12 shows whether actor D has read the tokens of the incoming buffer. Column 13 shows what message comes in from the ring and is put in the incoming buffer. The last column relates to the equations from Chapter 8.
For Table 9.6, this calculation is not available because there is no formula for hijacking. We also set, for Table 9.6, the ring size to 2, as shown in columns 6 7,8 and 13.

#### Row Description

For the row description, we describe in Table 9.3, each clock cycle.

**Table 9.3:** Clock cycle explanations

| Clock cycle | | | |
|---|---|---|---|
| Option 1, Table 9.4 | Option 2, Table 9.5 | Option 2, Table 9.6 | Meaning |
| 1 | 1 | 1 | Tokens are offered to actor B, who claims to have consumed and produced new tokens after a firing time of 0 clock cycles. |
| 2 | 2 | 2 | Messages Just 10 for actor A, Just 12 for actor C and Just 13 for actor D are offered to the router, from which it can choose. |
| 5,17 | 5, 17 | 2 | Messages for A sent. |
| 9, 21 | 9, 21 | 3, 4, 6 | Messages for C sent. |
| 13, 25, 29, 33, 37, 41 | 13, 25 | 5 | Messages for D sent. |
| 15, 27, 31, 35, 39, 43 | 15, 27 | 7 | The previous messages to D, received 2 clock cycles later. |
| 44 | 28 | 8 | Received messages offered to actor D, who consumes them and produces immediately. |
| 43 | 27 | 7 | WCET/firing time of the identity actor. |

[2] see Figures 9.1a and 9.2
[3] see Figures 9.1b and 9.2
[4] see Figures 9.1b and 9.2

**Table 9.4:** Result, $edge_6$, option 1, Ringsize(sd)=1, without hijacking

| C 1 | Column 2 | Column 3 | Column 4 | Column 5 | Column 6 | Column 7 | Column 8 | C 9 | Column 10 | Column 11 | Column 12 | Column 13 | Column 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0 | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>,Just 12<br>Just 12<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | TRUE | 0,0,0 | N | N | N | 1 | N | N | F | N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 2 | N | N | F | N | $NT-1$ |
| 5 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 5 | N | N | F | N | |
| 9 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 9 | N | N | F | N | |
| 13 | N | N | F | 0,0,1 | Just 10 | Just 12 | Just 13 | 13 | N | N | F | N | |
| 15 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 15 | N | N | F | Just 13 | |
| 17 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 17 | N | N | F | N | |
| 21 | N | N | F | 0,1,0 | N | Just 12 | Just 13 | 21 | N | N | F | N | |
| 25 | N | N | F | 0,0,1 | N | N | Just 13 | 25 | N | N | F | N | $\frac{NT(M-sd)}{sd}$ |
| 27 | N | N | F | 0,0,0 | N | N | Just 13 | 27 | N | N | F | Just 13 | |
| 29 | N | N | F | 0,0,1 | N | N | Just 13 | 29 | N | N | F | N | |
| 31 | N | N | F | 0,0,0 | N | N | Just 13 | 31 | N | N | F | Just 13 | |
| 33 | N | N | F | 0,0,1 | N | N | Just 13 | 33 | N | N | F | N | |
| 35 | N | N | F | 0,0,0 | N | N | Just 13 | 35 | N | N | F | Just 13 | |
| 37 | N | N | F | 0,0,1 | N | N | Just 13 | 37 | N | N | F | N | |
| 39 | N | N | F | 0,0,0 | N | N | Just 13 | 39 | N | N | F | Just 13 | |
| 41 | N | N | F | 0,0,1 | N | N | Just 13 | 41 | N | N | F | N | $HT$ |
| 43 | N | N | F | 0,0,0 | N | N | N | 43 | N | N | F | Just 13 | $B_i$ |
| 44 | N | N | F | 0,0,0 | N | N | N | 44 | Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31 | TRUE | N | |

**Table 9.5:** Result, $edge_6$, option 2, ringsize(sd) = 1, without hijacking

| C 1 | Column 2 | Column 3 | Column 4 | Column 5 | Column 6 | Column 7 | Column 8 | C 9 | Column 10 | Column 11 | Column 12 | Column 13 | Column 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 13<br>Just 13<br>Nothing<br>Nothing<br>Nothing<br>Nothing | TRUE | 0,0,0 | N | N | N | 1 | N | N | F | N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 2 | N | N | F | N | $NT-1$ |
| 5 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 5 | N | N | F | N | $NT(E-1)$ |
| 9 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 9 | N | N | F | N | |
| 13 | N | N | F | 0,0,1 | Just 10 | Just 12 | Just 13 | 13 | N | N | F | N | |
| 15 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 15 | N | N | F | Just 13 | $\frac{ENT(F-sd)}{sd}$ |
| 17 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 17 | N | N | F | N | |
| 21 | N | N | F | 0,1,0 | N | Just 12 | Just 13 | 21 | N | N | F | N | |
| 25 | N | N | F | 0,0,1 | N | Just 12 | Just 13 | 25 | N | N | F | N | $HT$ |
| 26 | N | N | F | 0,0,0 | N | Just 12 | N | 26 | N | N | F | N | |
| 27 | N | N | F | 0,0,0 | N | Just 12 | N | 27 | N | N | F | Just 13 | $B_i$ |
| 28 | N | N | F | 0,0,0 | N | Just 12 | N | 28 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N | |

**Table 9.6:** Result, $edge_6$, option 2, ringsize(sd) = 2, with hijacking

| C 1 | Column 2 | Column 3 | Column 4 | Column 5 | Column 6 | Column 7 | Column 8 | C 9 | Column 10 | Column 11 | Column 12 | Column 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer |
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 13<br>Just 13<br>Nothing<br>Nothing<br>Nothing<br>Nothing | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N |
| 2 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N |
| 3 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 3 | N | N | F | N,N |
| 4 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 4 | N | N | F | N,N |
| 5 | N | N | F | 0,0,1 | N,N | Just 12,Just 12 | Just 13,Just 13 | 5 | N | N | F | N,N |
| 6 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | N,N | 6 | N | N | F | N,N |
| 7 | N | N | F | 0,0,0 | N,N | N,N | N,N | 7 | N | N | F | Just 13,Just 13 |
| 8 | N | N | F | 0,0,0 | N,N | N,N | N,N | 8 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N,N |

## 9.2 Corresponding Results

In Table 9.7, we see the results of the equations and Clash simulation. We tested different options, varied the ring size is, adjusted the Hoptime, and allowed hijacking. For a system where hijacking is allowed, we cannot do a calculation, so this is not shown. Table 9.7 also contains references to the results of the previous simulation and tables, in which the same answer is given. For detailed simulation results, there is also an Appendix reference.
We see that the minimum result of the equations is equal to the simulation results.

**Table 9.7:** $Edge_6$ result comparison

| Edge | Option | Ring Size | Hijacking | Hop Time | Result equation: 8.6 | Result equation: 8.3 | Equation Table | Simulation Result | SimulationTable | Appendix |
|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 1 | 1 | No | 1 | 75 | 43 | 9.1a | 43 | 9.4 | C.1.2 |
| | | 1 | Yes | 1 | - | - | - | 15 | - | C.1.1 |
| | | 2 | No | 1 | 39 | 23 | 9.2a | 23 | - | C.1.3 |
| | | 2 | Yes | 1 | - | - | - | 9 | - | C.1.4 |
| | | 2 | No | 2 | 77 | 45 | - | 45 | - | C.1.5 |
| | | 2 | No | 3 | 115 | 67 | - | 67 | - | C.1.6 |
| | | 2 | No | 7 | 267 | 155 | - | 155 | - | C.1.7 |
| | 2 | 1 | No | 1 | 27 | 43 | 9.1b | 27 | 9.5 | C.2.1 |
| | | 1 | Yes | 1 | - | - | - | 11 | - | C.2.2 |
| | | 2 | No | 1 | 15 | 23 | 9.2b | 15 | - | C.2.3 |
| | | 2 | Yes | 1 | - | - | - | 7 | 9.6 | C.2.4 |
| | | 2 | No | 7 | 99 | 155 | - | 99 | - | C.2.5 |
| | 3 | 1 | No | 1 | 27 | 43 | 9.1c | 27 | - | C.3.1 |
| | | 1 | Yes | 1 | - | - | - | 9 | - | C.3.2 |
| | | 2 | No | 1 | 15 | 23 | 9.2c | 15 | - | C.3.3 |
| | | 2 | Yes | 1 | - | - | - | 7 | - | C.3.4 |
| | 4 | 2 | No | 7 | 183 | 183 | - | 183 | - | C.4.1 |

## 9.3 Conclusion Simulation

From this chapter, it can be concluded that the Clash simulation results are equal to the calculation of Chapter 8.

# Part IV

# Conclusions and Future Work

# Chapter 10

# Conclusions

In this chapter, we conclude this report and answer the main question:

*How do we design and analyse FPGA to FPGA communication in a defined topology, using dataflow graphs?*

We do this by answering the sub-questions that we asked in the Introduction.

*Which hardware communication infrastructure is suitable?*

As hardware topology, we have chosen the ring topology because it gives a uniform structure for each FPGA, which helps with the modularity and ensures that each FPGA only needs one Input and output port. There was also a deterministic implementation available, namely the Nebula ring interconnect.

*Are there any dataflow graph constraints, if so, which ones?*

The main restrictions for the dataflow graph are that they must be strongly connected and that is not allowed to have multiple edges from one FPGA to another.

*Given the topology, how do we map a dataflow graph onto multiple FPGAs?*

On each FPGA in a ring structure, we place an actor of the dataflow graph. To realise the ring, we placed a router, ringhop, controller and buffers on one FPGA. In Clash, we connected those elements using custom data types and record syntax. To

achieve a ring, we connect those FPGAs. For synchronisation between the FPGAs, we use separate First In First Outs (FIFOs) buffers. In the Clash implementation, the mapping of FIFOs implies that all FIFOs must be equal to each other, defining unused FIFO spots. This also means that FIFOs, which accept and provide multiple messages, must always receive and deliver a fixed number of messages. To bypass this, we used the Maybe Type. We used a router to receive or forward data from the previous FPGA. the router also choose which message to inject into te ring. There is no preference for a particular FIFO for the order in which messages are injected into the ring. We made it is also possible to put multiple messages/tokens on the ring. Although this is for future work, we have implemented the router in such a way that it can also be used as a credit-ring and allows hijacking of slots. The controller is used to make it easier for the user of the system to know if there are enough tokens to consume. The user/designer remains responsible for the realisation of the actor and knowing when there are enough tokens produced. The controller also protects against a too low production.

*How can we model the temporal behaviour of the design, analyse the communication and guarantee deterministic behaviour?*

By modelling the network communication time between two actors as a new actor, an identity actor is added to each edge of the initial dataflow graph. This creates a resulting dataflow graph model on which the user can perform an analysis for the communication time between FPGAs.

By calculating the WCET as firing time for the identity actors, we can guarantee deterministic behaviour.
We made two equations to calculate this WCET. With the first option, we are entirely dependent on the maximum number of messages in the output buffer. With the second option, we only depend on the messages in one FIFO, to a certain depth, multiplied by the number of edges. By taking the minimum of both options we get actual WCET, as firing time for the identity actor.

*How do simulation results correspond to analysis results concerning timing?*

We have seen that the Clash simulation results are equal to de calculation of Chapter 8.

**With the answers of the sub-questions, we answer the main question:**

We have chosen for a ring topology with the Nebula ring interconnect. Where each FPGA represents an actor of the initial dataflow graph. The user can then give an initial dataflow graph to our Clash implementation. This is modelled by a resulting dataflow graph, in which additional identity actors are added. These actors represent the network communication time between two actors of the initial dataflow graph. For these actors, we can calculate the WCET as firing time. The designer can then analyse this new model. We also compared the calculated results with the Clash simulation and found that they are the same, for the tests we did.

# Chapter 11

# Future Work

This chapter presents future work on subjects that have not yet been discussed or implemented in this thesis. These topics are mainly direction in which the project can be expanded or optimised.

## 11.1 Maximum Buffer Occupation

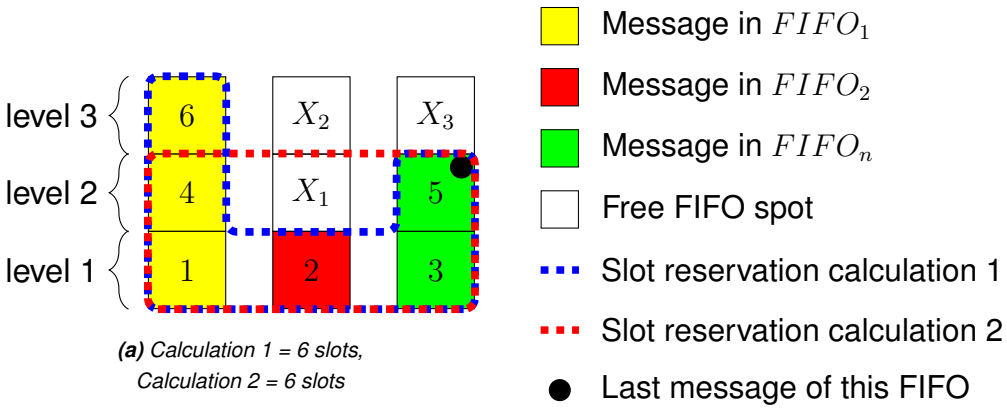
In our calculations, we assume that we know how many messages are maximally stored in the output buffer. With the help of the SDF3 tool from [29], we can calculate this maximum or at least get an indication. However, further research is still required to find this maximum.

## 11.2 Actor Location

It also has to be determined which actor has to be placed on which FPGA. Closely linked FPGAs can reduce communication time. It may also be possible to put multiple actors on the same FPGA. An article of Ramezani [24] is a good start.

## 11.3 Calculation Improvement

### 11.3.1 Adaption of Existing Calculation




***(a)*** *Calculation 1 = 6 slots, Calculation 2 = 6 slots*

***(b)*** *Legend*

**Figure 11.1:** Buffer occupation example

It is possible to make, option 2, calculation 8.4 from Chapter 8.1.2 even more strict. e.g. We assume that Figure 11.1 is the maximum output buffer occupation, where each column represents a FIFO from the output buffer and a $X_i$ is an empty spot in the FIFO. If we want to send the messages of the green right column or Figure 11.1 then, without hijacking, we have to send up to level 2. This means we need three edges times tow levels = six own slots until the last green message, but in practice, we only need five own slots to send until the last green message. So we took into account time to send message $X_1$, while this is an unfilled slot. So the calculation can be stricter.

### 11.3.2 Additional Calculations

To calculate the firing time of the identity actor, we calculate how long it takes to receive the last message. This means that moving all tokens from one side of the identity actor to the other takes the same time. However, in practice, some tokens have reached their destination earlier. Therefore, we can model this with a CSDF graph, instead of an SDF graph.

## 11.4 Credit Ring

Currently, each edge of the initial dataflow graph represents an output FIFO, some network communication, and an input FIFO. The size of the input FIFO must be equal to the maximum number of messages in the output FIFO of the other FPGA, to ensure that the messages can be received. Suppose an actor of the dataflow graph has multiple input edges. In that case, the input buffer also has multiple FIFOs, so each FIFO must be equal to the maximum number of messages of its equivalent output FIFO on the other FPGA. Because of the higher-order functions in Clash, the FIFOs in the buffers must also be equal to each other. This allocating memory may cause us to reserve memory we don not use.

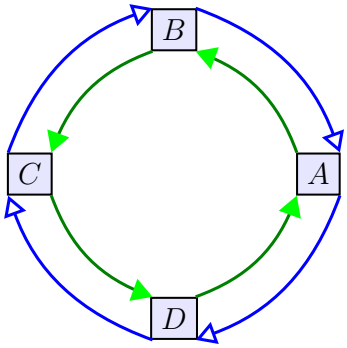



**Figure 11.2:** Credit-ring topology

By adding a credit-ring that goes in the opposite direction, see Figure 11.2, we can make the FIFOs smaller. Thus, saving memory. In this way, we can indicate when there is a place in the input FIFO available. The credits require memory, but the credits do not necessarily have the same size as the messages of the original ring. The implementation of the credit-ring is at the expense of the firing time of the identity actors. Therefore, the formulas from Chapter 8 are no longer valid and have to be redefined.

### 11.4.1 Credit-ring in Clash

In Clash, we have already started implementing the credit-ring. We use the same implementation as the regular ring, but change the mode to `DecreasingWithoutHijack` or `DecreasingWithHijack`, depending on whether or not the credit-ring is allowed to hijack. The corresponding rules for the credit-ring are available in appendix 11.3. To connect the regular ring to the router, we have already modified the router and controller and added some inputs and outputs. See appendix A.2 how we connect the credit-ring to the controller. The controller reads from the input FIFO and indicates that places are available again by adding credits to the output buffer of the

credit-ring. The regular router, in turn, sees that credits are available, only in those cases can it send a message. If it has sent a message, a credit is removed from the input buffer of the credit-ring. Simulating the credit-ring has not been done and is something for future work.

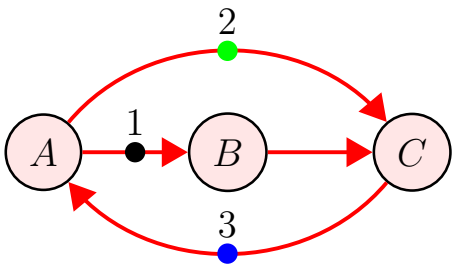



**Figure 11.3:** Three Node, dataflow graph example

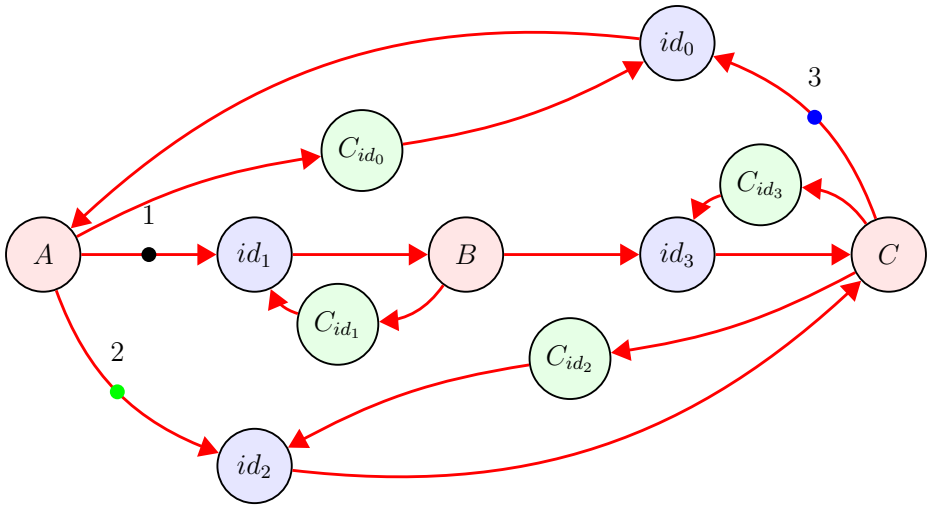



**Figure 11.4:** Resulting Dataflow graph: Three node, dataflow graph example with credit-ringif we summarise the previous slides.

If we take the initial dataflow graph from the dataflow graph of Figure 11.3, then the resulting dataflow graph looks like in Figure 11.4. This transformation shows that a simple initial dataflow graph quickly becomes complex when a credit-ring is used as well.

## 11.5 Additional Slots

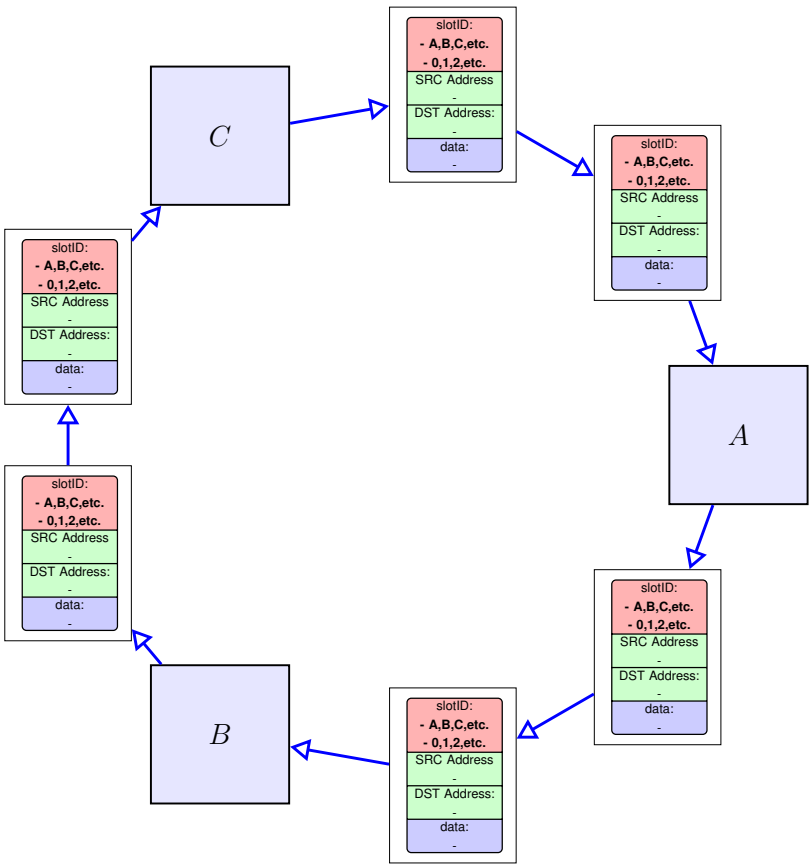



**Figure 11.5:** Multiple slots in Nebula ring

It is possible to place multiple slots one after the other, see Figure 11.5 for an example. These multiple slots allow us to provide multiple slots for each FPGA. If the slots are located one after the other, e.g. A, A, B, B, C, C, ..., then the waiting time until an own slot is available is longer. This waiting makes the WCET/firing time larger, but we can also choose to give a certain FPGA multiple 'own' slots at the expense of another FPGA. This allocation, in turn, ensures that some FPGAs have a larger bandwidth and, therefore, faster get all messages to their destination. The calculations take this partially into account, but it still needs to be redefined and tested in Clash. In Clash, we did use these multiple slots to simulate hopTime, but then we set the slots to `Invalid`.

## 11.6 Ring-Intermediate Topology

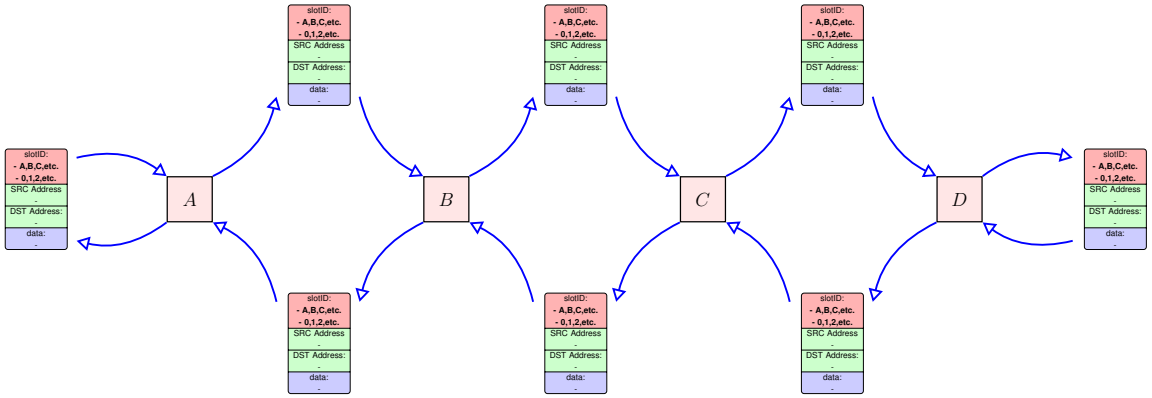



**Figure 11.6:** Ring-intermediate example

As announced in Chapter 4, we could use the ring-intermediate topology. For this, we need to change the router and split it, whereby a part of the messages will go to the nodes with a higher numbers/characters and a part to the lower numbers/characters. This routing ensures that the decision in the router remains trivial. In the ring-intermediate, there are at least twice as many slots as in the regular ring.
See Figure 11.6 for a topology with four FPGAs, where we usually have four slots, now we have eight. The path from the last FPGA back to the first one now runs through all other FPGAs. This makes the communication path longer. However, for some actors, the communication path has become shorter. Because of this the communication time changes and the expectation is that the ring-intermediate for the inner FPGAs has on average a faster firing time [1] and that for the outer FPGAs the WCET has not increased. e.g. For a Ring with three FPGAs A, B, C there are two hops from C to B. Further for the ring-intermediate, there are two hops from C to A.
Again it is possible to replace a slot id with another slot id, so a slot is capable of sending more data, while another is reduced. The implementation and calculations of this design are future work.

[1] This depends on which actor is placed on which FPGA

## 11.7 CSDF Graphs

In short, Cyclo-Static DataFlow (CSDF) graphs are dataflow graphs that cycle through different consuming and production rates, and firing times. For now, it is not yet possible to use CSDF graphs. Because we now have a fixed consumption and production rate. If we made the production and consumption rate a vector, we are in Clash obligated to make each vector equal. This can be solved by using the Least Common Multiple (LCM). e.g. if an actor has two outgoing edges of which one edge produces [2,3,5] and the other edge [4,2] then this would be implemented as [2,3,5,2,3,5] and [4,2,4,2,4,2]. The same principle applies to the incoming edges. The router, controller and function still need to be adapted to take this cyclic implementation into account.

## 11.8 Multi-Edged Dataflow Graphs

It is not yet possible to create a dataflow graph with multiple edges from one FPGA to another, see 11.7 for an example. This is because the elements on the FPGA do not know that there are multiple edges. Every message that now enters the router will be put in the same input FIFO.

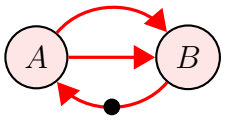



**Figure 11.7:** Multi-edged dataflow graph example

```
1  data RingContent  id a  b    =
2            Invalid
3          | EmptySlot
4          | ContentSlot { slotId      :: id
5                        , source      :: id
6                        , destination :: id
7                        , edgeID      :: Index b
8                        , content     :: a
9                        }
```

***Listing 26:*** *Ring Content type*

There are several ways to indicate that there are multiple edges; one example is that we can add an edge Id to the `Content`-type, see Listing 26. So we will indicate to which edge the message belongs. The router, controller and function must then be adjusted so that they can handle this.

## 11.9 (De)serialising

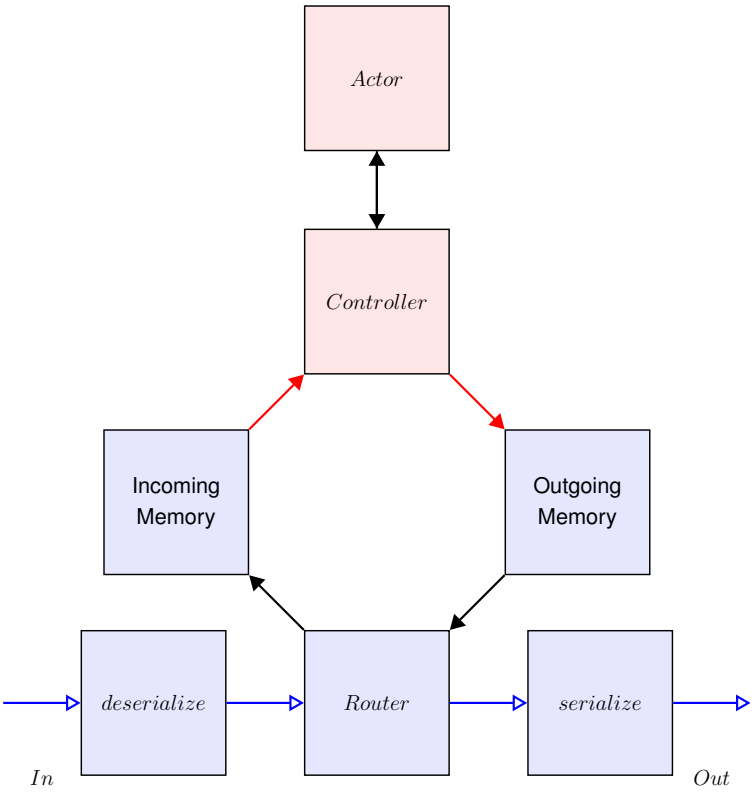



**Figure 11.8:** FPGA with serialiser and deserialiser

The width of the content of the ring is the number of wires between FPGAs. This is equal to the number of bits the slot has, see Figure 2.9b. To reduce this number. The ringhop can be adjusted so that it has a protocol that deserialises the data. A serialiser still needs to be realised and implemented. An FPGA implementation then looks similar to Figure 11.8. During the time of deserialisation, we signal the router with `Invalid`. We assume that the time of serialising is equal to the deserialisation. If we, for example, have 160 pins to utilise for every FPGA, we can use eighty for the input and eighty for the output. If then the `slotID` and Content-type; thus, the `source`, `destination` and `content` are bigger than eighty bits or ten bytes. So, serialising and de serialising is inevitable.

## 11.10 Physical Implementation

```
--Actor A ------------------------------------------------------------------------------------------
actor0 input = bundle ( (cRing'         <$> fromNode_)
                      , (slotId'        <$> fromNode_)
                      , (vReadCredits   <$> fromNode_)
                      , (vvmNewCredits <$> fromNode_)
                      )
    where
         (cRing_, slotId_, vvmCredits_) = unbundle input
         (toRing_, read_)               = unbundle $ f0M  (mvvmFromRing <$> fromNode_)
         fromNode_                      = node_0_M (ToNodeF <$> cRing_
                                                            <*> slotId_
                                                            <*> toRing_
                                                            <*> read_
                                                            <*> vvmCredits_
                                                    )
```

***Listing 27:*** *Connecting Function to Controller*

```
        topEntity =   actor0
```

***Listing 28:*** *making the topEntity*

Actual testing on physical hardware is also something for future work. We can do this by making a topEntity of an actor. In Listing 27, we see how we connect the function `f0M` to our system, creating a complete FPGA implementation. In Listing 28 we create a topentity. From this topEntity we can generate a Verilog or VHDL design. This generated design can then be programmed on an FPGA, with for example Quartus or Vivado. This generation and programming have to be done for all actors. Then they have to be physically connected. Also, clock synchronisation has to be added between the FPGAs.

# Bibliography


[1] fullvector / Freepik, Web, Jul. 2020, designed by fullvector / Freepik. [Online]. Available: https://www.freepik.com/free-photos-vectors/icon

[2] E. Raalte, "Automating system generation in clash," Master's thesis, University of Twente, 2019.

[3] H. Zodpe and A. Sapkal, "Fpga-based high-performance computing platform for cryptanalysis of aes algorithm," in *Computing in Engineering and Technology*. Springer, 2020, pp. 637–646.

[4] J. H. Oh, Y. Hyun Yoon, J. K. Kim, H. Bin Ihm, S. H. Jeon, T. Heon Kim, and S. E. Lee, "An FPGA-based Electronic Control Unit for Automotive Systems," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*, Jan. 2019, pp. 1–2, iSSN: 2158-4001.

[5] A. Ling and J. Anderson, "The Role of FPGAs in Deep Learning," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 2017, p. 3. [Online]. Available: https://doi.org/10.1145/3020078.3030013

[6] G. A. Constantinides, "FPGAs in the Cloud," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 2017, p. 167. [Online]. Available: https://doi.org/10.1145/3020078.3030014

[7] "Intel Completes Acquisition of Altera." [Online]. Available: https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[8] "What is an FPGA? Programming and FPGA Basics - INTEL® FPGAS." [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/new-to-fpgas/resource-center/overview.html

[9] "What is an FPGA? Field Programmable Gate Array." [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[10] “Clash.” [Online]. Available: https://clash-lang.org/

[11] “Qbaylogic.” [Online]. Available: https://qbaylogic.com/

[12] “Haskell,” Online, Jun. 2020. [Online]. Available: https://www.haskell.org/

[13] M. Lipovaca, *Learn you a haskell for great good!: a beginner’s guide*. no starch press, 2011.

[14] M. Bekooij, “Datafow analysis for real-time multiprocessor systems,” May 2017, lecture Notes Real-Time Systems 2 Course.

[15] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2018.

[16] B. H. Dekens, P. Wilmanns, M. J. Bekooij, and G. J. Smit, “Low-cost guaranteed-throughput communication ring for real-time streaming mpsocs,” in *2013 Conference on Design and Architectures for Signal and Image Processing*. IEEE, 2013, pp. 239–246.

[17] B. H. Dekens, P. S. Wilmanns, G. J. Smit, and M. J. Bekooij, “Low-cost guaranteed-throughput dual-ring communication infrastructure for heterogeneous mpsocs,” in *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*. IEEE, 2014, pp. 1–8.

[18] B. H. J. Dekens, *Low-Cost Heterogeneous Embedded Multiprocessor Architecture for Real-Time Stream Processing Applications*. University of Twente, 2015.

[19] B. H. Dekens, M. J. Bekooij, and G. J. Smit, “Real-time multiprocessor architecture for sharing stream processing accelerators,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 81–89.

[20] G. G. Wevers, “Hardware accelerator sharing within an mpsoc with a connectionless noc,” September 2014. [Online]. Available: http://essay.utwente.nl/66088/

[21] D. Veer, “Design of a gmsk receiver prototype on a heterogeneous real-time multiprocessor platform,” Master’s thesis, University of Twente, 2016.

[22] G. Kuiper, “Guaranteed-throughput improvement techniques for connectionless ring networks,” Master’s thesis, University of Twente, 2013.

[23] M. A. Khalid, *Routing architecture and layout synthesis for multi-FPGA systems*. Ph. D. dissertation, Dept. of ECE, Univ. Toronto, 1999.

[24] R. Ramezani, "Dynamic scheduling of task graphs in multi-fpga systems using critical path," *The Journal of Supercomputing*, pp. 1–22, 2020.

[25] M. Owaida and G. Alonso, "Application partitioning on fpga clusters: Inference over decision tree ensembles," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 295–2955.

[26] O. Mencer, K. H. Tsoi, S. Craimer, T. Todman, W. Luk, M. Y. Wong, and P. H. W. Leong, "Cube: A 512-fpga cluster," in *2009 5th Southern Conference on Programmable Logic (SPL)*. IEEE, 2009, pp. 51–57.

[27] W. Liu, M. Yuan, X. He, Z. Gu, and X. Liu, "Efficient sat-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization." IEEE, 2008, pp. 492–504.

[28] H. Ali, "Integrating dataflow and non-dataflow real-time application models on multi-core platforms," Ph.D. dissertation, Faculdade de Engenharia da Universidade do Porto, 2017.

[29] S. Stuijk, M. Geilen, and T. Basten, "SDF$^3$: SDF For Free," in *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006, pp. 276–278. [Online]. Available: http://www.es.ele.tue.nl/sdf3

# Part V

# Appendices

# Appendix A

# Clash Schematics

## A.1 Regular Ring

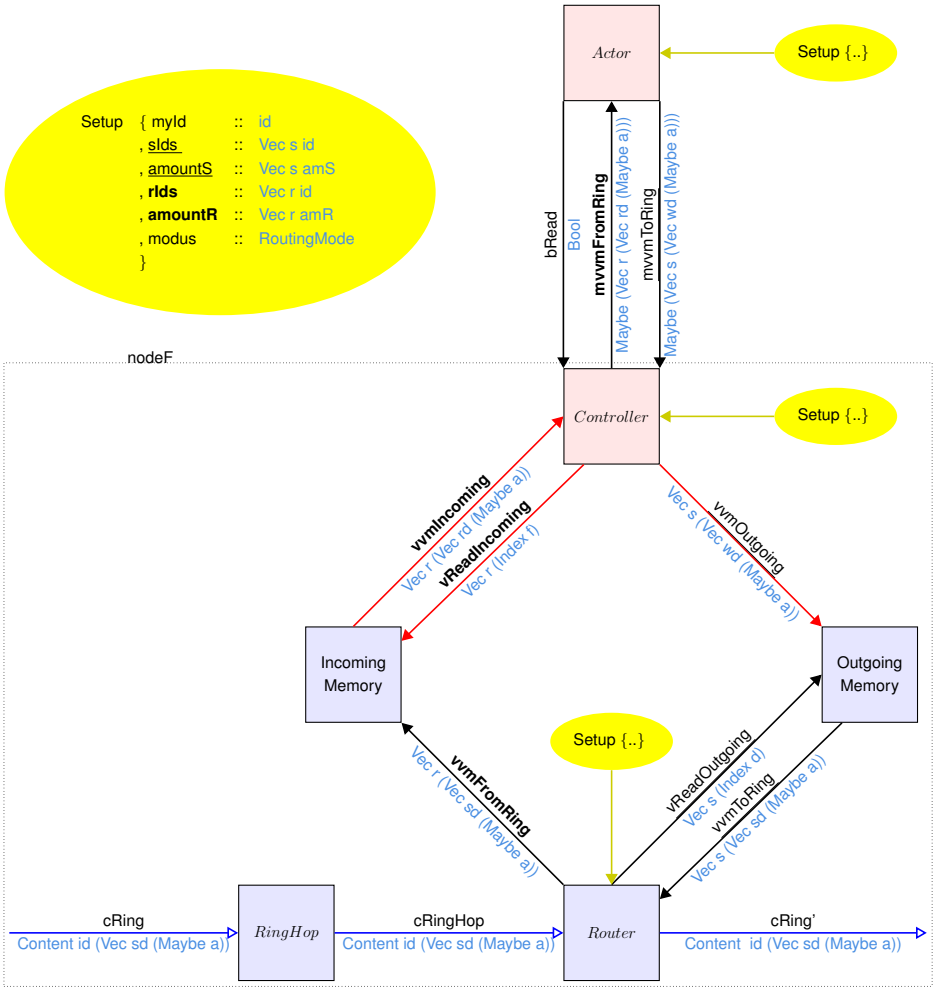



**Figure A.1:** Clash implementations schematic

## A.2 Credit Ring

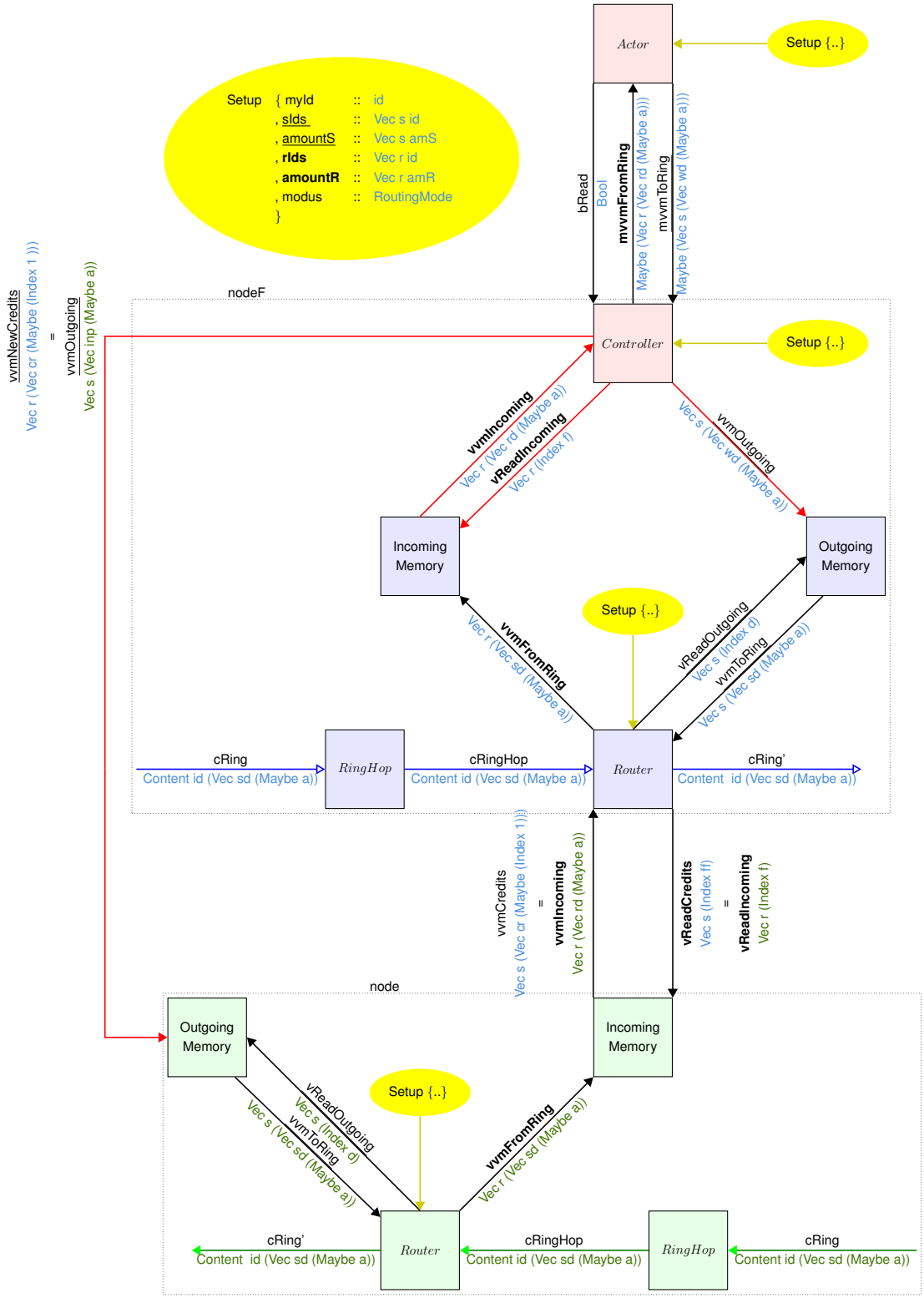
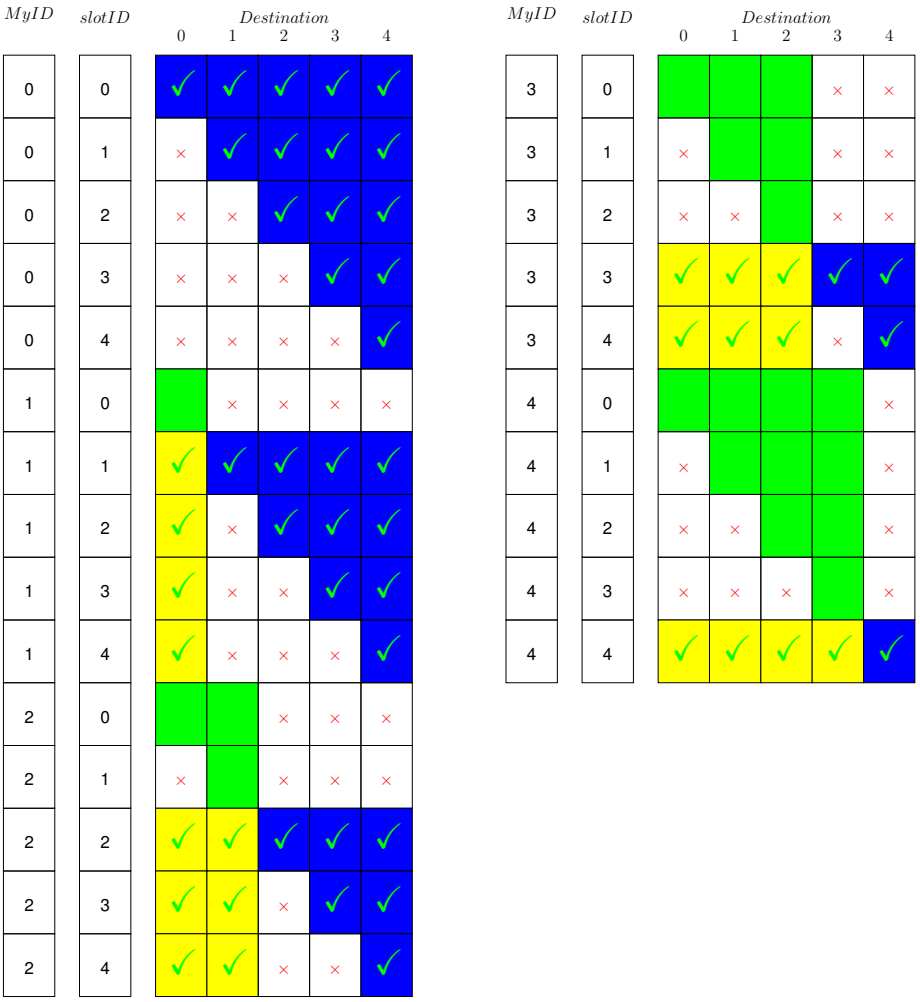



**Figure A.2:** Clash implementations schematic, with credit-ring

# Appendix B

# Rules Credit Ring Hijacking

- 
- $Destination < myID \leq slotID$
- $slotID \leq Destination < myID$



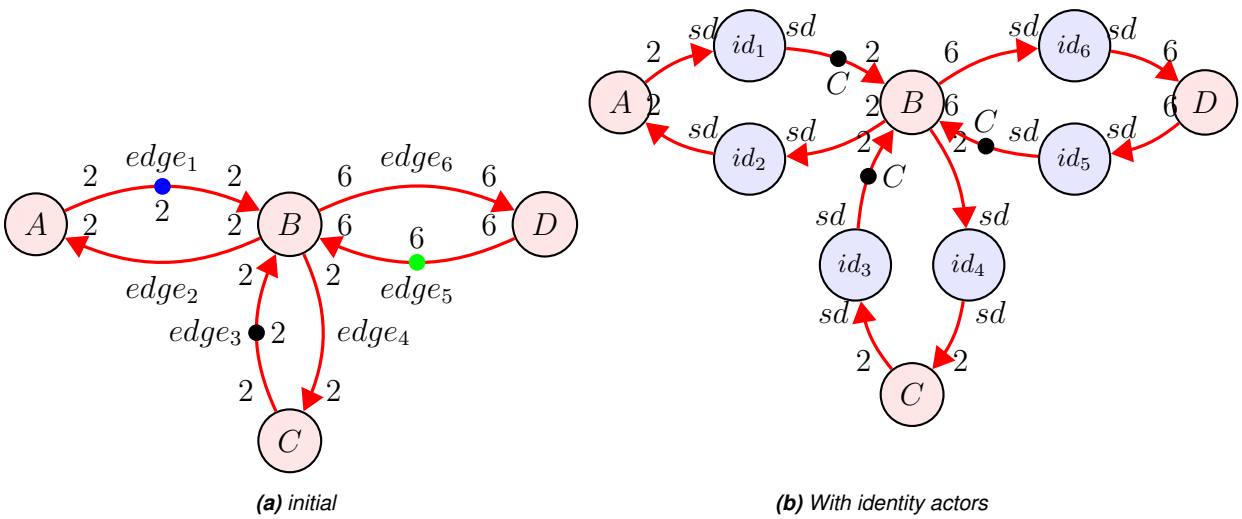
# Appendix C

# Simulation Results

$$\begin{array}{c} \\ edge_1 \{ \\ edge_2 \{ \\ edge_3 \{ \\ edge_4 \{ \\ edge_5 \{ \\ edge_6 \{ \end{array} \overbrace{\begin{array}{cccc} A & B & C & D \\ 2 & -2 & 0 & 0 \\ -2 & 2 & 0 & 0 \\ 0 & -2 & 2 & 0 \\ 0 & 2 & -2 & 0 \\ 0 & -6 & 0 & 6 \\ 0 & 6 & 0 & -6 \end{array}}^{} \quad \begin{bmatrix} 2 & -2 & 0 & 0 \\ -2 & 2 & 0 & 0 \\ 0 & -6 & 6 & 0 \\ 0 & 6 & -6 & 0 \\ 0 & -2 & 0 & 2 \\ 0 & 2 & 0 & -2 \end{bmatrix} \quad \begin{bmatrix} 6 & -6 & 0 & 0 \\ -6 & 6 & 0 & 0 \\ 0 & -2 & 2 & 0 \\ 0 & 2 & -2 & 0 \\ 0 & -2 & 0 & 2 \\ 0 & 2 & 0 & -2 \end{bmatrix} \quad \begin{bmatrix} 2 & -2 & 0 & 0 \\ -2 & 2 & 0 & 0 \\ 0 & -6 & 6 & 0 \\ 0 & 6 & -6 & 0 \\ 0 & -2 & 0 & 2 \\ 0 & 4 & 0 & -4 \end{bmatrix}$$

***(a)*** *Option 1* ***(b)*** *Option 2* ***(c)*** *Option 3* ***(d)*** *Option 4*

**Figure C.1:** Topology matrices for different implementations




***(a)*** *initial* ***(b)*** *With identity actors*

**Figure C.2:** Dataflow graphs: Option 1

# C.1 Option 1

## C.1.1 Ringsize(sd)=1, With Hijacking, HopTime(T)=1

**Table C.1:** Result $edge_6$, Option 1, Ringsize(sd)=1, With Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0 | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 12<br>Just 12<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | TRUE | 0,0,0 | N | N | N | 1 | N | N | F | N | |
| 2 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 2 | N | N | F | N | |
| 3 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 3 | N | N | F | N | |
| 4 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 4 | N | N | F | N | |
| 5 | N | N | F | 0,0,1 | Just 10 | N | Just 13 | 5 | N | N | F | N | |
| 6 | N | N | F | 1,0,0 | Just 10 | N | Just 13 | 6 | N | N | F | N | |
| 7 | N | N | F | 0,0,1 | N | N | Just 13 | 7 | N | N | F | Just 13 | |
| 8 | N | N | F | 0,0,0 | N | N | Just 13 | 8 | N | N | F | N | |
| 9 | N | N | F | 0,0,1 | N | N | Just 13 | 9 | N | N | F | Just 13 | |
| 10 | N | N | F | 0,0,1 | N | N | Just 13 | 10 | N | N | F | N | |
| 11 | N | N | F | 0,0,1 | N | N | Just 13 | 11 | N | N | F | Just 13 | N/A |
| 12 | N | N | F | 0,0,0 | N | N | Just 13 | 12 | N | N | F | Just 13 | |
| 13 | N | N | F | 0,0,1 | N | N | Just 13 | 13 | N | N | F | Just 13 | |
| 14 | N | N | F | 0,0,0 | N | N | N | 14 | N | N | F | N | |
| 15 | N | N | F | 0,0,0 | N | N | N | 15 | N | N | F | Just 13 | |
| 16 | N | N | F | 0,0,0 | N | N | N | 16 | Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31 | TRUE | N | |

## C.1.2 Ringsize(sd)=1, Without Hijacking, HopTime(T)=1

**Table C.2:** Result $edge_6$, Option 1, Ringsize(sd)=1, Without Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0 | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 12<br>Just 12<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | TRUE | 0,0,0 | N | N | N | 1 | N | N | F | N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 2 | N | N | F | N | |
| 3 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 3 | N | N | F | N | $NT-1$ |
| 4 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 4 | N | N | F | N | |
| 5 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 5 | N | N | F | N | |
| 6 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 6 | N | N | F | N | |
| 7 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 7 | N | N | F | N | |
| 8 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 8 | N | N | F | N | |
| 9 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 9 | N | N | F | N | |
| 10 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 10 | N | N | F | N | |
| 11 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 11 | N | N | F | N | |
| 12 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 12 | N | N | F | N | |
| 13 | N | N | F | 0,0,1 | Just 10 | Just 12 | Just 13 | 13 | N | N | F | N | |
| 14 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 14 | N | N | F | N | |
| 15 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 15 | N | N | F | Just 13 | |
| 16 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 16 | N | N | F | N | |
| 17 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 17 | N | N | F | N | |
| 18 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 18 | N | N | F | N | |
| 19 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 19 | N | N | F | N | |
| 20 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 20 | N | N | F | N | |
| 21 | N | N | F | 0,1,0 | N | Just 12 | Just 13 | 21 | N | N | F | N | |
| 22 | N | N | F | 0,0,0 | N | N | Just 13 | 22 | N | N | F | N | $\frac{NT(M-sd)}{sd}$ |
| 23 | N | N | F | 0,0,0 | N | N | Just 13 | 23 | N | N | F | N | |
| 24 | N | N | F | 0,0,0 | N | N | Just 13 | 24 | N | N | F | N | |
| 25 | N | N | F | 0,0,1 | N | N | Just 13 | 25 | N | N | F | N | |
| 26 | N | N | F | 0,0,0 | N | N | Just 13 | 26 | N | N | F | N | |
| 27 | N | N | F | 0,0,0 | N | N | Just 13 | 27 | N | N | F | Just 13 | |
| 28 | N | N | F | 0,0,0 | N | N | Just 13 | 28 | N | N | F | N | |
| 29 | N | N | F | 0,0,1 | N | N | Just 13 | 29 | N | N | F | N | |
| 30 | N | N | F | 0,0,0 | N | N | Just 13 | 30 | N | N | F | N | |
| 31 | N | N | F | 0,0,0 | N | N | Just 13 | 31 | N | N | F | Just 13 | |
| 32 | N | N | F | 0,0,0 | N | N | Just 13 | 32 | N | N | F | N | |
| 33 | N | N | F | 0,0,1 | N | N | Just 13 | 33 | N | N | F | N | |
| 34 | N | N | F | 0,0,0 | N | N | Just 13 | 34 | N | N | F | N | |
| 35 | N | N | F | 0,0,0 | N | N | Just 13 | 35 | N | N | F | Just 13 | |
| 36 | N | N | F | 0,0,0 | N | N | Just 13 | 36 | N | N | F | N | |
| 37 | N | N | F | 0,0,1 | N | N | Just 13 | 37 | N | N | F | N | |
| 38 | N | N | F | 0,0,0 | N | N | Just 13 | 38 | N | N | F | N | |
| 39 | N | N | F | 0,0,0 | N | N | Just 13 | 39 | N | N | F | Just 13 | |
| 40 | N | N | F | 0,0,0 | N | N | Just 13 | 40 | N | N | F | N | |
| 41 | N | N | F | 0,0,1 | N | N | Just 13 | 41 | N | N | F | N | $HT$ |
| 42 | N | N | F | 0,0,0 | N | N | N | 42 | N | N | F | N | |
| 43 | N | N | F | 0,0,0 | N | N | N | 43 | N | N | F | Just 13 | $B_i$ |
| 44 | N | N | F | 0,0,0 | N | N | N | 44 | Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31 | TRUE | N | |

## C.1.3 Ringsize(sd)=2, Without Hijacking HopTime(T)=1

**Table C.3:** Result $edge_6$, Option 1, Ringsize(sd)=2, Without Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing, | Nothing | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing, | Nothing, | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| 1 | Just 0 | Just 13 | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | $B_o$ |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| 2 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| 3 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 3 | N | N | F | N,N | $NT-1$ |
| 4 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 4 | N | N | F | N,N | |
| 5 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 5 | N | N | F | N,N | |
| 6 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 6 | N | N | F | N,N | |
| 7 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 7 | N | N | F | N,N | |
| 8 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 8 | N | N | F | N,N | |
| 9 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 9 | N | N | F | N,N | |
| 10 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 10 | N | N | F | N,N | |
| 11 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 11 | N | N | F | N,N | |
| 12 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 12 | N | N | F | N,N | $\frac{NT(M-sd)}{sd}$ |
| 13 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 13 | N | N | F | N,N | |
| 14 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 14 | N | N | F | N,N | |
| 15 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 15 | N | N | F | Just 13,Just 13 | |
| 16 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 16 | N | N | F | N,N | |
| 17 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 17 | N | N | F | N,N | |
| 18 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 18 | N | N | F | N,N | |
| 19 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 19 | N | N | F | Just 13,Just 13 | |
| 20 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 20 | N | N | F | N,N | |
| 21 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 21 | N | N | F | N,N | |
| 22 | N | N | F | 0,0,0 | N,N | N,N | N,N | 22 | N | N | F | N,N | $HT$ |
| 23 | N | N | F | 0,0,0 | N,N | N,N | N,N | 23 | N | N | F | Just 13,Just 13 | $B_i$ |
| | | | | | | | | | Just 13 | Just 31 | | | |
| | | | | | | | | | Just 13 | Just 31 | | | |
| | | | | | | | | | Just 13 | Just 31 | | | |
| 24 | N | N | F | 0,0,0 | N,N | N,N | N,N | 24 | Just 13 | Just 31 | TRUE | N,N | |
| | | | | | | | | | Just 13 | Just 31 | | | |
| | | | | | | | | | Just 13 | Just 31 | | | |

## C.1.4 Ringsize(sd)=2, With Hijacking, HopTime(T)=1

**Table C.4:** Result $edge_6$, Option 1, Ringsize(sd)=2, With Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>,Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0 | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 12<br>Just 12<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>,Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | |
| 2 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| 3 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 3 | N | N | F | N,N | |
| 4 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 4 | N | N | F | N,N | |
| 5 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 5 | N | N | F | N,N | |
| 6 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 6 | N | N | F | N,N | |
| 7 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 7 | N | N | F | Just 13,Just 13 | |
| 8 | N | N | F | 0,0,0 | N,N | N,N | N,N | 8 | N | N | F | Just 13,Just 13 | N/A |
| 9 | N | N | F | 0,0,0 | N,N | N,N | N,N | 9 | N | N | F | Just 13,Just 13 | |
| 10 | N | N | F | 0,0,0 | N,N | N,N | N,N | 10 | Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31 | TRUE | N,N | |

## C.1.5 Ringsize(sd)=2, Without Hijacking, HopTime(T)=2

**Table C.5:** Result $edge_6$, Option 1, Ringsize(sd)=2, Without Hijacking, HopTime(T)=2

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| 1 | Just 0 | Just 13 | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | | | | | |
| 8 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 8 | N | N | F | N,N | $NT-1$ |
| 9 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 9 | N | N | F | N,N | |
| 10 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 10 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | ⋮ | | | | | |
| 16 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 16 | N | N | F | N,N | |
| 17 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 17 | N | N | F | N,N | |
| 18 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 18 | N | N | F | N,N | |
| | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | | | | | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 24 | N | N | F | N,N | |
| 25 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 25 | N | N | F | N,N | |
| 26 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 26 | N | N | F | N,N | |
| 27 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 27 | N | N | F | N,N | |
| 28 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 28 | N | N | F | N,N | $\frac{NT(M-sd)}{sd}$ |
| 29 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 29 | N | N | F | Just 13,Just 13 | |
| 30 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 30 | N | N | F | N,N | |
| 31 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 31 | N | N | F | N,N | |
| 32 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 32 | N | N | F | N,N | |
| 33 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 33 | N | N | F | N,N | |
| 34 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 34 | N | N | F | N,N | |
| 35 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 35 | N | N | F | N,N | |
| 36 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 36 | N | N | F | N,N | |
| 37 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 37 | N | N | F | Just 13,Just 13 | |
| 38 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 38 | N | N | F | N,N | |
| 39 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 39 | N | N | F | N,N | |
| 40 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 40 | N | N | F | N,N | |
| 41 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 41 | N | N | F | N,N | |
| 42 | N | N | F | 0,0,0 | N,N | N,N | N,N | 42 | N | N | F | N,N | $HT$ |
| 43 | N | N | F | 0,0,0 | N,N | N,N | N,N | 43 | N | N | F | N,N | |
| 44 | N | N | F | 0,0,0 | N,N | N,N | N,N | 44 | N | N | F | N,N | |
| 45 | N | N | F | 0,0,0 | N,N | N,N | N,N | 45 | N | N | F | Just 13,Just 13 | $B_i$ |
| | | | | | | | | | Just 13 | Just 31 | | | |
| | | | | | | | | | Just 13 | Just 31 | | | |
| | | | | | | | | | Just 13 | Just 31 | | | |
| | | | | | | | | | Just 13 | Just 31 | | | |
| | | | | | | | | | Just 13 | Just 31 | | | |
| 46 | N | N | F | 0,0,0 | N,N | N,N | N,N | 46 | Just 13 | Just 31 | TRUE | N,N | |

## C.1.6 Ringsize(sd)=2, Without Hijacking, HopTime(T)=3

**Table C.6:** Result $edge_6$, Option 1, Ringsize(sd)=2, Without Hijacking, HopTime(T)=3

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Just 0<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 0<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing,<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0 | Just 10<br>Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 12<br>Just 12<br>Just 12<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | $NT-1$ |
| 12 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 12 | N | N | F | N,N | |
| 13 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 13 | N | N | F | N,N | |
| 14 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 14 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 24 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 24 | N | N | F | N,N | |
| 25 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 25 | N | N | F | N,N | |
| 26 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 26 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 36 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 36 | N | N | F | N,N | |
| 37 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 37 | N | N | F | N,N | |
| 38 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 38 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 42 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 42 | N | N | F | N,N | |
| 43 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 43 | N | N | F | Just 13,Just 13 | $\frac{NT(M-sd)}{sd}$ |
| 44 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 44 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 48 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 48 | N | N | F | N,N | |
| 49 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 49 | N | N | F | N,N | |
| 50 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 50 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | | | | | |
| 54 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 54 | N | N | F | N,N | |
| 55 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 55 | N | N | F | Just 13,Just 13 | |
| 56 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 56 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 60 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 60 | N | N | F | N,N | |
| 61 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 61 | N | N | F | N,N | |
| 62 | N | N | F | 0,0,0 | N,N | N,N | N,N | 62 | N | N | F | N,N | $HT$ |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | N,N | ⋮ | N | N | F | N,N | |
| 66 | N | N | F | 0,0,0 | N,N | N,N | N,N | 66 | N | N | F | N,N | |
| 67 | N | N | F | 0,0,0 | N,N | N,N | N,N | 67 | N | N | F | Just 13,Just 13 | $B_i$ |
| 68 | N | N | F | 0,0,0 | N,N | N,N | N,N | 68 | Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31 | TRUE | N,N | |

### C.1.7 Ringsize(sd)=2, Without Hijacking, HopTime(T =7

**Table C.7:** Result $edge_6$, Option 1, Ringsize(sd)=2, Without Hijacking, HopTime(T)=7

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0 | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 12<br>Just 12<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | $NT-1$ |
| 28 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 28 | N | N | F | N,N | |
| 29 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 29 | N | N | F | N,N | |
| 30 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 30 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 56 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 56 | N | N | F | N,N | |
| 57 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 57 | N | N | F | N,N | |
| 58 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 58 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 84 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 84 | N | N | F | N,N | |
| 85 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 85 | N | N | F | N,N | |
| 86 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 86 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | N | N | F | N,N | $\frac{NT(M-sd)}{sd}$ |
| 112 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 112 | N | N | F | N,N | |
| 113 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 113 | N | N | F | N,N | |
| 114 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 114 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 126 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 126 | N | N | F | N,N | |
| 127 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 127 | N | N | F | Just 13,Just 13 | |
| 128 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 128 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 140 | N | N | F | 0,0,0 | N,N | N,N | Just 13,Just 13 | 140 | N | N | F | N,N | |
| 141 | N | N | F | 0,0,1 | N,N | N,N | Just 13,Just 13 | 141 | N | N | F | N,N | |
| 142 | N | N | F | 0,0,0 | N,N | N,N | N,N | 142 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | N,N | N,N | ⋮ | N | N | F | N,N | $HT$ |
| 154 | N | N | F | 0,0,0 | N,N | N,N | N,N | 154 | N | N | F | N,N | |
| 155 | N | N | F | 0,0,0 | N,N | N,N | N,N | 155 | N | N | F | Just 13,Just 13 | $B_i$ |
| 156 | N | N | F | 0,0,0 | N,N | N,N | N,N | 156 | Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Just 13 | Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31<br>Just 31 | TRUE | N,N | |

## C.2 Option 2

### C.2.1 Ringsize(sd)=1, Without Hijacking, HopTime(T)=1

**Table C.8:** Result $edge_6$, Option 2, Ringsize(sd)=1, Without Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 13<br>Just 13<br>Nothing<br>Nothing<br>Nothing<br>Nothing | TRUE | 0,0,0 | N | N | N | 1 | N | N | F | N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 2 | N | N | F | N | |
| 3 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 3 | N | N | F | N | $NT-1$ |
| 4 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 4 | N | N | F | N | |
| 5 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 5 | N | N | F | N | |
| 6 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 6 | N | N | F | N | |
| 7 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 7 | N | N | F | N | |
| 8 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 8 | N | N | F | N | $NT(E-1)$ |
| 9 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 9 | N | N | F | N | |
| 10 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 10 | N | N | F | N | |
| 11 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 11 | N | N | F | N | |
| 12 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 12 | N | N | F | N | |
| 13 | N | N | F | 0,0,1 | Just 10 | Just 12 | Just 13 | 13 | N | N | F | N | |
| 14 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 14 | N | N | F | N | |
| 15 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 15 | N | N | F | Just 13 | |
| 16 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 16 | N | N | F | N | |
| 17 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 17 | N | N | F | N | |
| 18 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 18 | N | N | F | N | $\frac{ENT(F-sd)}{sd}$ |
| 19 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 19 | N | N | F | N | |
| 20 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 20 | N | N | F | N | |
| 21 | N | N | F | 0,1,0 | N | Just 12 | Just 13 | 21 | N | N | F | N | |
| 22 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 22 | N | N | F | N | |
| 23 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 23 | N | N | F | N | |
| 24 | N | N | F | 0,0,0 | N | Just 12 | Just 13 | 24 | N | N | F | N | |
| 25 | N | N | F | 0,0,1 | N | Just 12 | Just 13 | 25 | N | N | F | N | $HT$ |
| 26 | N | N | F | 0,0,0 | N | Just 12 | N | 26 | N | N | F | N | |
| 27 | N | N | F | 0,0,0 | N | Just 12 | N | 27 | N | N | F | Just 13 | $B_i$ |
| 28 | N | N | F | 0,0,0 | N | Just 12 | N | 28 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N | |

### C.2.2 Ringsize(sd)=1, With Hijacking, HopTime(T)=1

**Table C.9:** Result $edge_6$, Option 2, Ringsize(sd)=1, With Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing | Just 10<br>Just 10<br>Nothing<br>Nothing<br>Nothing<br>Nothing | | | | | | | | | | | |
| | Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0 | Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12 | | | | | | | | | | | |
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing | Just 13<br>Just 13<br>Nothing<br>Nothing<br>Nothing<br>Nothing | TRUE | 0,0,0 | N | N | N | 1 | N | N | F | N | |
| 2 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 2 | N | N | F | N | |
| 3 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 3 | N | N | F | N | |
| 4 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 4 | N | N | F | N | |
| 5 | N | N | F | 0,0,1 | Just 10 | Just 12 | Just 13 | 5 | N | N | F | N | |
| 6 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 6 | N | N | F | N | |
| 7 | N | N | F | 0,1,0 | N | Just 12 | Just 13 | 7 | N | N | F | Just 13 | N/A |
| 8 | N | N | F | 0,1,0 | N | Just 12 | Just 13 | 8 | N | N | F | N | |
| 9 | N | N | F | 0,0,1 | N | Just 12 | Just 13 | 9 | N | N | F | N | |
| 10 | N | N | F | 0,1,0 | N | Just 12 | N | 10 | N | N | F | N | |
| 11 | N | N | F | 0,1,0 | N | Just 12 | N | 11 | N | N | F | Just 13 | |
| 12 | N | N | F | 0,0,0 | N | N | N | 12 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N | |

### C.2.3 Ringsize(sd)=2, Without Hijacking, HopTime(T)=1

**Table C.10:** Result $edge_6$, Option 2, Ringsize(sd)=2, Without Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| 1 | | | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| 3 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 3 | N | N | F | N,N | $NT-1$ |
| 4 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 4 | N | N | F | N,N | |
| 5 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 5 | N | N | F | N,N | |
| 6 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 6 | N | N | F | N,N | |
| 7 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 7 | N | N | F | N,N | |
| 8 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 8 | N | N | F | N,N | $NT(E-1)$ |
| 9 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 9 | N | N | F | N,N | |
| 10 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 10 | N | N | F | N,N | |
| 11 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 11 | N | N | F | N,N | |
| 12 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 12 | N | N | F | N,N | |
| 13 | N | N | F | 0,0,1 | N,N | Just 12,Just 12 | Just 13,Just 13 | 13 | N | N | F | N,N | $HT$ |
| 14 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 14 | N | N | F | N,N | |
| 15 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 15 | N | N | F | Just 13,Just 13 | $B_i$ |
| 16 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 16 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N,N | |

### C.2.4 Ringsize(sd)=2, With ijacking, opTime(T)=

**Table C.11:** Result $edge_6$, Option 2, Ringsize(sd)=2, With Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| 1 | Nothing | Nothing | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | |
| 2 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| 3 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 3 | N | N | F | N,N | |
| 4 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 4 | N | N | F | N,N | |
| 5 | N | N | F | 0,0,1 | N,N | Just 12,Just 12 | Just 13,Just 13 | 5 | N | N | F | N,N | N/A |
| 6 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | N,N | 6 | N | N | F | N,N | |
| 7 | N | N | F | 0,0,0 | N,N | N,N | N,N | 7 | N | N | F | Just 13,Just 13 | |
| 8 | N | N | F | 0,0,0 | N,N | N,N | N,N | 8 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N,N | |

## C.2.5 Ringsize(sd)=2, Without Hijacking, HopTime(T)=2

**Table C.12:** Result $edge_6$, Option 2, Ringsize(sd)=2, Without Hijacking, HopTime(T)=7

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| 1 | Nothing | Nothing | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | $NT-1$ |
| 28 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 28 | N | N | F | N,N | |
| 29 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 29 | N | N | F | N,N | |
| 30 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 30 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 56 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 56 | N | N | F | N,N | |
| 57 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 57 | N | N | F | N,N | $NT(E-1)$ |
| 58 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 58 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 84 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 84 | N | N | F | N,N | |
| 85 | N | N | F | 0,0,1 | N,N | Just 12,Just 12 | Just 13,Just 13 | 85 | N | N | F | N,N | |
| 86 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 86 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | ⋮ | N | N | F | N,N | $HT$ |
| 98 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 98 | N | N | F | N,N | |
| 99 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 99 | N | N | F | Just 13,Just 13 | $B_i$ |
| 100 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 100 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N,N | |

## C.3 Option 3

### C.3.1 Ringsize(sd)=1, Without Hijacking, HopTime(T)=1

**Table C.13:** Result $edge_6$, Option 3, Ringsize(sd)=1, Without Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing | Just 10<br>Just 10<br>Just 10<br>Just 10<br>Just 10<br>Just 10<br>Just 12<br>Just 12<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 13<br>Just 13<br>Nothing<br>Nothing<br>Nothing<br>Nothing | TRUE | 0,0,0 | N | N | N | 1 | N | N | F | N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 2 | N | N | F | N | |
| 3 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 3 | N | N | F | N | $NT-1$ |
| 4 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 4 | N | N | F | N | |
| 5 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 5 | N | N | F | N | |
| 6 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 6 | N | N | F | N | |
| 7 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 7 | N | N | F | N | |
| 8 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 8 | N | N | F | N | $NT(E-1)$ |
| 9 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 9 | N | N | F | N | |
| 10 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 10 | N | N | F | N | |
| 11 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 11 | N | N | F | N | |
| 12 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 12 | N | N | F | N | |
| 13 | N | N | F | 0,0,1 | Just 10 | Just 12 | Just 13 | 13 | N | N | F | N | |
| 14 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 14 | N | N | F | N | |
| 15 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 15 | N | N | F | Just 13 | |
| 16 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 16 | N | N | F | N | |
| 17 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 17 | N | N | F | N | |
| 18 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 18 | N | N | F | N | $\frac{ENT(F-sd)}{sd}$ |
| 19 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 19 | N | N | F | N | |
| 20 | N | N | F | 0,0,0 | Just 10 | Just 12 | Just 13 | 20 | N | N | F | N | |
| 21 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 21 | N | N | F | N | |
| 22 | N | N | F | 0,0,0 | Just 10 | N | Just 13 | 22 | N | N | F | N | |
| 23 | N | N | F | 0,0,0 | Just 10 | N | Just 13 | 23 | N | N | F | N | |
| 24 | N | N | F | 0,0,0 | Just 10 | N | Just 13 | 24 | N | N | F | N | |
| 25 | N | N | F | 0,0,1 | Just 10 | N | Just 13 | 25 | N | N | F | N | $HT$ |
| 26 | N | N | F | 0,0,0 | Just 10 | N | N | 26 | N | N | F | N | |
| 27 | N | N | F | 0,0,0 | Just 10 | N | N | 27 | N | N | F | Just 13 | $B_i$ |
| 28 | N | N | F | 0,0,0 | Just 10 | N | N | 28 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N | |

### C.3.2 Ringsize(sd)=1, With Hijacking, HopTime(T)=1

**Table C.14:** Result $edge_6$, Option 3, Ringsize(sd)=1, With Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| 1 | Nothing | Nothing | TRUE | 0,0,0 | N | N | N | 1 | N | N | F | N | |
| 2 | N | N | F | 1,0,0 | Just 10 | Just 12 | Just 13 | 2 | N | N | F | N | |
| 3 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 3 | N | N | F | N | |
| 4 | N | N | F | 0,1,0 | Just 10 | Just 12 | Just 13 | 4 | N | N | F | N | |
| 5 | N | N | F | 0,0,1 | Just 10 | N | Just 13 | 5 | N | N | F | N | |
| 6 | N | N | F | 1,0,0 | Just 10 | N | Just 13 | 6 | N | N | F | N | N/A |
| 7 | N | N | F | 0,0,1 | Just 10 | N | Just 13 | 7 | N | N | F | Just 13 | |
| 8 | N | N | F | 0,0,0 | Just 10 | N | N | 8 | N | N | F | N | |
| 9 | N | N | F | 1,0,0 | Just 10 | N | N | 9 | N | N | F | Just 13 | |
| 10 | N | N | F | 1,0,0 | Just 10 | N | N | 10 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N | |

### C.3.3 Ringsize(sd)=2, Without Hijacking, HopTime(T)=1

**Table C.15:** Result $edge_6$, Option 3, Ringsize(sd)=2, Without Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| 1 | Nothing | Nothing | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | $B_o$ |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| 2 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| 3 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 3 | N | N | F | N,N | $NT-1$ |
| 4 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 4 | N | N | F | N,N | |
| 5 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 5 | N | N | F | N,N | |
| 6 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 6 | N | N | F | N,N | |
| 7 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 7 | N | N | F | N,N | |
| 8 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 8 | N | N | F | N,N | |
| 9 | N | N | F | 0,1,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 9 | N | N | F | N,N | $NT(E-1)$ |
| 10 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | Just 13,Just 13 | 10 | N | N | F | N,N | |
| 11 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | Just 13,Just 13 | 11 | N | N | F | N,N | |
| 12 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | Just 13,Just 13 | 12 | N | N | F | N,N | |
| 13 | N | N | F | 0,0,1 | Just 10,Just 10 | N,N | Just 13,Just 13 | 13 | N | N | F | N,N | $HT$ |
| 14 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | N,N | 14 | N | N | F | N,N | |
| 15 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | N,N | 15 | N | N | F | Just 13,Just 13 | $B_i$ |
| 16 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | N,N | 16 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N,N | |

## C.3.4 Ringsize(sd)=2, With Hijacking, HopTime(T)=1

**Table C.16:** Result $edge_6$, Option 3, Ringsize(sd)=2, With Hijacking, HopTime(T)=1

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 10 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Just 0 | Just 12 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Just 0 | Just 13 | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| | Nothing | Nothing | | | | | | | | | | | |
| 1 | | | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | |
| 2 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| 3 | N | N | F | 0,1,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 3 | N | N | F | N,N | |
| 4 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | Just 13,Just 13 | 4 | N | N | F | N,N | |
| 5 | N | N | F | 0,0,1 | Just 10,Just 10 | N,N | Just 13,Just 13 | 5 | N | N | F | N,N | N/A |
| 6 | N | N | F | 1,0,0 | Just 10,Just 10 | N,N | N,N | 6 | N | N | F | N,N | |
| 7 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | N,N | 7 | N | N | F | Just 13,Just 13 | |
| 8 | N | N | F | 0,0,0 | Just 10,Just 10 | N,N | N,N | 8 | Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N,N | |

## C.4 Option 4

### C.4.1 Ringsize(sd)=2, Without Hijacking, HopTime(T)=7

**Table C.17:** Result $edge_6$, Option 4, Ringsize(sd)=2, Without Hijacking, HopTime(T)=7

| # | Actor B Consuming edge(s) | Actor B Producing edge(s) | Actor B Read | B Writes To | To Router B from Outgoing Buffer | | | # | Actor D Consuming edge(s) | Actor D Producing edge(s) | Actor D Read | From Router D to Incoming Buffer | Formula |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Just 0<br>Nothing<br>Nothing<br>Nothing<br>Nothing | Just 10<br>Just 10<br>Just 10<br>Just 10<br>Nothing<br>Nothing<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 12<br>Just 13<br>Just 13<br>Just 13<br>Just 13<br>Nothing<br>Nothing | TRUE | 0,0,0 | N,N | N,N | N,N | 1 | N | N | F | N,N | $B_o$ |
| 2 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 2 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | $NT-1$ |
| 28 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 28 | N | N | F | N,N | |
| 29 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 29 | N | N | F | N,N | |
| 30 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 30 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 56 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 56 | N | N | F | N,N | |
| 57 | N | N | F | 0,1,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 57 | N | N | F | N,N | $NT(E-1)$ |
| 58 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 58 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 84 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 84 | N | N | F | N,N | |
| 85 | N | N | F | 0,0,1 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 85 | N | N | F | N,N | |
| 86 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 86 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 98 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 98 | N | N | F | N,N | |
| 99 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 99 | N | N | F | Just 13,Just 13 | |
| 100 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 100 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 112 | N | N | F | 0,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 112 | N | N | F | N,N | |
| 113 | N | N | F | 1,0,0 | Just 10,Just 10 | Just 12,Just 12 | Just 13,Just 13 | 113 | N | N | F | N,N | $\frac{ENT(F-sd)}{sd}$ |
| 114 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 114 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 140 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 140 | N | N | F | N,N | |
| 141 | N | N | F | 0,1,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 141 | N | N | F | N,N | |
| 142 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 142 | N | N | F | N,N | |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | ⋮ | N | N | F | N,N | |
| 168 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | Just 13,Just 13 | 168 | N | N | F | N,N | |
| 169 | N | N | F | 0,0,1 | N,N | Just 12,Just 12 | Just 13,Just 13 | 169 | N | N | F | N,N | |
| 170 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 170 | N | N | F | N,N | $HT$ |
| ⋮ | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | ⋮ | N | N | F | N,N | |
| 182 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 182 | N | N | F | N,N | |
| 183 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 183 | N | N | F | Just 13,Just 13 | $B_i$ |
| 184 | N | N | F | 0,0,0 | N,N | Just 12,Just 12 | N,N | 184 | Just 13<br>Just 13<br>Just 13<br>Just 13 | Just 31<br>Just 31 | TRUE | N,N | |

# Appendix D

# Clash Code

## D.1 Connecting Elements

### D.1.1 DataTypes

```
-- ================================================================================================================
--  Project           : Mapping Dataflow graphs on multiple FPGAs
--  Faculty           : University of Twente, CAES-group
--  Program name      : DataTypes.hs
--  Author            : Sander Bremmer
--  Date created      : 31-07-2020
--  Purpose           : creating datatypes, for connecting , FPGA State and ring Content
-- ================================================================================================================
-- r   = receive from 'r' Nodes
-- s   = send to 's' Nodes
-- sd  = send 'sd'  messages on the ring at the same Time
-- id  = the type the nodes have to identify them
-- d   = depth of outgoing buffer, so fifo size
-- f   = depth 'f' of incoming buffer, so fifo size
-- rd  = receive depth of incoming buffer, from the edge
-- wd  = send depth out outgoing buffer , to the edge,
-- a   = type of data  the nodes send or recieve
-- cr  = amount of credits to  receive from to credit ring
-- ff  = depth of the incoming credit buffer.
-- ================================================================================================================

{-# LANGUAGE StandaloneDeriving #-}
module DataflowMultiFPGA.DataTypes where

import Clash.Prelude
-- ================================================================================================================
---------- DATA ON RING ---------- DATA ON RING ---------- DATA ON RING ---------- DATA ON RING ----------
-- ================================================================================================================
data RingContent  id a =  Invalid
                        | EmptySlot        {slotId::id}
                        | ContentSlot      {slotId :: id, source :: id, destination :: id, content:: (Vec sd (Maybe a))
                           deriving   (Show , Generic, NFDataX )
-- ================================================================================================================
---------- NODE STATES ---------- NODE STATES ---------- NODE STATES ---------- NODE STATES ----------
-- ================================================================================================================
data ElementStates id h sd r s d f a   =
    ElementStates      { obState :: Vec s (Vec d (Maybe a)) -- Out going Buffer state
                       , ibState :: Vec r (Vec f (Maybe a)) -- Incoing Buffer state
                       , rState  :: Index s                 -- Router state = pointer
                       , rhState :: Vec h ( RingContent id (Vec sd (Maybe a)))
                       } deriving (Show, Generic)

deriving instance
    ( NFDataX a, NFDataX id, KnownNat h, KnownNat sd, KnownNat d,KnownNat s, KnownNat f, KnownNat r)
```

```
45      =>  NFDataX (ElementStates id h sd r s d f a  )
46
47  -- ================================================================================================================
48  ------- ROUTING MODES ----------- ROUTING MODES ----------- ROUTING MODES ----------- ROUTING MODES ------
49  -- ================================================================================================================
50  data RoutingMode =      IncreasingWithoutHijack     --       regular ring without hijacking
51                     |    IncreasingWithHijack        --       regular ring with hijacking
52                     |    DecreasingWithoutHijack     --       credit ring without hijacking
53                     |    DecreasingWithHijack        --       creditring, with hijacking
54                          deriving Show
55
56  -- ================================================================================================================
57  ----------- FIXED VALUE ----------- FIXED VALUE ----------- FIXED VALUE ----------- FIXED VALUE ----------
58  -- ================================================================================================================
59  data Setup id wd rd s r =
60          Setup   { myId      :: id                       -- Id of Node
61                  , sIds      :: Vec s id                 -- send to these id's
62                  , amountS   :: Vec s (Index (wd + 1))   -- corresponding production rates sIds
63                  , rIds      :: Vec r id                 -- receive from these id's
64                  , amountR   :: Vec r (Index (rd + 1))   -- corresponding consumtion rates rIds
65                  , modus     :: RoutingMode              --represents the 4 possibilities of routing
66                  } deriving (Show)
67
68  -- ================================================================================================================
69  ----------------- COMMUNICATION INSIDE NODE ----------------- COMMUNICATION INSIDE NODE -----------------
70  -- ================================================================================================================
71  data ElementConnect id d f rd sd wd r s a  cr ff =
72  ---------- FUNCTION --------- FUNCTION --------- FUNCTION --------- FUNCTION --------- FUNCTION ----------
73      ToFuncCtrl              { mvvmToRing        :: Maybe (  Vec s (Vec wd (Maybe a)))
74                              , vvmIncoming       :: Vec      r (Vec rd (Maybe a))
75                              , bRead             :: Bool
76                              }
77  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
78      | FromFuncCtrl          { mvvmFromRing      :: Maybe (  Vec r (Vec rd (Maybe  a)))
79                              , vvmOutgoing       :: Vec      s   (Vec wd (Maybe a))
80                              , vReadIncoming     :: Vec      r   (Index f)
81                              , vvmNewCredits     :: Vec      r   (Vec cr (Maybe (Index 1)))
82                              }
83  ---------- RING HOP --------- RING HOP --------- RING HOP --------- RING HOP --------- RING HOP ----------
84      | ToRingHop             { cRing             :: RingContent  id  (Vec sd (Maybe a)) --
85                              }
86      | FromRingHop           { cRingHop          :: RingContent  id  (Vec sd (Maybe a)) --
87                              }
88  --------- ROUTER -------- ROUTER -------- ROUTER -------- ROUTER -------- ROUTER -------- ROUTER ---------
89      | ToRouter              { cRingHop          :: RingContent  id  (Vec sd (Maybe a)) --
90                              , vvmToRing         :: Vec      s   (Vec sd (Maybe a))
91                              , vvmCredits        :: Vec      s   (Vec cr (Maybe (Index 1) ))
92                              }
93  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
94      | FromRouter            { cRing'            :: RingContent  id  (Vec sd (Maybe a)) --
95                              , vvmFromRing       :: Vec      r   (Vec sd (Maybe a))
96                              , vReadOutgoing     :: Vec      s   (Index d) --
97                              , vReadCredits      :: Vec      s   (Index ff)
98                              }
99  --------- MESSAGE BUFFER -------- MESSAGE BUFFER -------- MESSAGE BUFFER -------- MESSAGE BUFFER ---------
100     | ToIncomingBuffer      { vvmFromRing       :: Vec      r (Vec sd (Maybe a))
101                             , vReadIncoming     :: Vec      r (Index f)
102                             }
103 --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
104     | FromIncomingBuffer    { vvmIncoming       :: Vec      r (Vec rd (Maybe a))
105                             }
106 -- ================================================================================================================
107     | ToOutgoingBuffer      { vvmOutgoing       :: Vec      s (Vec wd (Maybe a))
108                             , vReadOutgoing     :: Vec      s (Index d) --
109                             }
110 --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
111     | FromOutgoingBuffer    { vvmToRing         :: Vec      s (Vec sd (Maybe a))
112                             }
113 -------- NODE -------- NODE -------- NODE -------- NODE -------- NODE -------- NODE -------- NODE --------
114     | ToNode                { cRing             :: RingContent  id  a
115                             , vReadIncoming     :: Vec      r   (Index f)
```

```
                          , vvmOutgoing       :: Vec      s   (Vec wd (Maybe a))
                          , vvmCredits        :: Vec      s   ( Vec cr (Maybe (Index 1) ))
                          }
--  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
    | FromNode            { cRing'            :: RingContent  id  a
                          , vvmIncoming       :: Vec      r   (Vec rd (Maybe a))
                          , vReadCredits      :: Vec      s   (Index ff)
                          }
-- ============================================================================================================
    | ToNodeF             { cRing             :: RingContent  id  a
                          , mvvmToRing        :: Maybe (  Vec s (Vec wd (Maybe a)))
                          , bRead             :: Bool
                          , vvmCredits        :: Vec      s   (Vec cr (Maybe (Index 1 )))
                          }
--  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
    | FromNodeF           { cRing'            :: RingContent id   a
                          , mvvmFromRing      :: Maybe (  Vec r (Vec rd (Maybe  a)))
                          , vReadCredits      :: Vec      s   (Index ff)
                          , vvmNewCredits     :: Vec      r   (Vec cr (Maybe (Index 1)))
                          -- for Debug
                          , vvmFromRing       :: Vec      r   (Vec sd (Maybe a))
                          , vvmToRing         :: Vec      s   (Vec sd (Maybe a))
                          } deriving (Show, Generic, NFDataX)
-- ============================================================================================================
```

## D.1.2 NodeConnect

```
 -- ============================================================================================================
--  Project           : Mapping Dataflow graphs on multiple FPGAs
--  Faculty           : University of Twente, CAES-group
--  Program name      : NodeConnect.hs
--  Author            : Sander Bremmer
--  Date created      : 31-07-2020
--  Purpose           : Connecting the  elements/components
-- ============================================================================================================
 {-# LANGUAGE RecordWildCards , ExistentialQuantification            #-}
module DataflowMultiFPGA.NodeConnect where

import Clash.Prelude
import DataflowMultiFPGA.DataTypes
import DataflowMultiFPGA.Buffer
import DataflowMultiFPGA.Router
import DataflowMultiFPGA.Controller
import DataflowMultiFPGA.RoundRobin
import DataflowMultiFPGA.RingHop

-------------------------------------------------------------------------------------------------------------
-- ---------- NODE WITHOUT FUCNTION --------------- NODE WITHOUT FUCNTION -----------
-------------------------------------------------------------------------------------------------------------

node :: ( Ord id
        , Show id
        , Num id
        , KnownNat d
        , KnownNat rd
        , KnownNat sd
        , KnownNat something0
        , KnownNat something1
        , KnownNat wd
        , KnownNat s
        , KnownNat r
        , KnownNat f
        , KnownNat cr
        , KnownNat ff
        , KnownNat n1
        , (1 <= cr)
        , 1 <= (s * wd)
        , 1 <= (r * rd)
        , 1 <= (rd + sd)
```

```
       , (((rd + sd) - 1) + something1) ~ f
       , 1 <= (sd + wd)
       , (((sd + wd) - 1) + something0) ~ d
       , Div (s + s) 2 ~ s
       , (n20 + 1) ~ sd
       , (n1 + 1) ~ h)
       =>  Setup id (wd + 1) (rd + 1) s r
         -> ElementStates id  h sd  r s d f a
         -> ElementConnect id  d f rd sd wd r s a  cr ff
         -> ( ElementStates id  h sd r s d f a,  ElementConnect id d f rd sd wd r s a  cr ff)

node   Setup{..} ElementStates{..} ToNode{..}=  (newStates, FromNode{..} )
    where
        newStates =  ElementStates   {
                                    obState  = obState'
                                    , ibState  = ibState'
                                    , rState   = rState'
                                    , rhState = rhState'
                                  }
        (ibState'    , FromIncomingBuffer{..}  ) = inComingBuffer      ibState      ToIncomingBuffer{..}
        (obState'    , FromOutgoingBuffer{..}  ) = outGoingBuffer      obState      ToOutgoingBuffer{..}
        (rState'     , FromRouter{..}   )        = router        Setup{..} rState       ToRouter{..}
        (rhState', FromRingHop{..})          = ringHop             rhState  ToRingHop{..}

-- ------------------------------------------------------------------------------------------------------------
-- ----- NODE WITH FUCNTION ------ NODE WITH FUCNTION ------ NODE WITH FUCNTION -----
-- ------------------------------------------------------------------------------------------------------------
nodeF :: ( Ord id
       , Show id
       , KnownNat d
       , KnownNat rd
       , KnownNat sd
       , KnownNat something0
       , KnownNat something1
       , KnownNat wd
       , KnownNat s
       , KnownNat r
       , KnownNat f
       , KnownNat cr
       , KnownNat ff
       , KnownNat n1
       , (1 <=? cr) ~ 'True
       , 1 <= (s * wd)
       , 1 <= (r * rd)
       , 1 <= (rd + sd)
       , (((rd + sd) - 1) + something1) ~ f
       , 1 <= (sd + wd)
       , (((sd + wd) - 1) + something0) ~ d
       , Div (s + s) 2 ~ s
       , (n20 + 1) ~ sd
       ,  (n1 + 1) ~ h)
       =>  Setup id (wd + 1) (rd + 1) s r
         -> ElementStates id  h sd r s d f a
         -> ElementConnect id d f rd sd wd r s a  cr ff
         -> ( ElementStates id h sd r s d f a,  ElementConnect id d f rd sd wd r s a  cr ff)

nodeF   Setup{..} ElementStates{..} ToNodeF{..}=  (newStates, FromNodeF{..} )
    where
        newStates =  ElementStates   {
                                    obState  = obState'
                                    , ibState  = ibState'
                                    , rState   = rState'
                                    , rhState = rhState'
                                  }
        FromFuncCtrl{..}                    = funcCtrl       Setup{..}          ToFuncCtrl{..}
        (ibState', FromIncomingBuffer{..}  ) = inComingBuffer           ibState  ToIncomingBuffer{..}
        (obState', FromOutgoingBuffer{..}  ) = outGoingBuffer           obState  ToOutgoingBuffer{..}
        (rState' , FromRouter{..}   )        = router         Setup{..} rState   ToRouter{..}
        (rhState', FromRingHop{..})          = ringHop                 rhState  ToRingHop{..}
-- ============================================================================================================
```

## D.2 Simulation Results

## D.3 Elements in detail

### D.3.1 Controller

```haskell
-- ============================================================================================================
--  Project           : Mapping Dataflow graphs on multiple FPGAs
--  Faculty           : University of Twente, CAES-group
--  Program name      : Controller.hs
--  Author            : Sander Bremmer
--  Date created      : 31-07-2020
--  Purpose           : Controller between  functional actor and buffers
-- ============================================================================================================
{-# LANGUAGE RecordWildCards #-}
module DataflowMultiFPGA.Controller where

import Clash.Prelude
import DataflowMultiFPGA.DataTypes
import DataflowMultiFPGA.HelperFunctions

funcCtrl Setup{..} ToFuncCtrl{..} = FromFuncCtrl{..}
  where
    checkS        = case mvvmToRing of
                      Nothing   -> False
                      (Just v ) -> validCheck'' (resize <$> amountS) v
 -- checkR = returns boolean if there are enough packages received on all edges.
    checkR        = validCheck'' (resize <$> amountR) vvmIncoming

    mvvmFromRing  | checkR          = Just (selector (resize <$> amountR) vvmIncoming)
                  | otherwise       = Nothing

    vvmOutgoing   | checkS = case mvvmToRing of
                             Just v -> v
                             _      -> repeat ( repeat Nothing) -- should never be called
                  | otherwise       = repeat ( repeat Nothing)

    vReadIncoming | checkR && bRead = resize <$> amountR
                  | otherwise       = repeat 0

    vvmNewCredits | checkR && bRead = repeat (repeat (Just 0 ))
                  | otherwise       = repeat (repeat Nothing )
-- ============================================================================================================
```

### D.3.2 Router

```haskell
-- ============================================================================================================
--  Project           : Mapping Dataflow graphs on multiple FPGAs
--  Faculty           : University of Twente, CAES-group
--  Program name      : Router.hs
--  Author            : Sander Bremmer
--  Date created      : 31-07-2020 31-07-2020
--  Purpose           : Router of the ring
-- ============================================================================================================
{-# LANGUAGE RecordWildCards              #-}
module DataflowMultiFPGA.Router where

import Clash.Prelude
import DataflowMultiFPGA.DataTypes
import DataflowMultiFPGA.RoundRobin
import DataflowMultiFPGA.HelperFunctions

router Setup{..} pointer  ToRouter{..} = ( pointer'   ,FromRouter{..})
    where
-- function to check if content with a specified destination id can be placed on the available slot
```

```
        slotId' = slotId cRingHop
        checkDest destinationId =
                case (modus, cRingHop) of
                        ( IncreasingWithHijack, EmptySlot _) ->  (a && b) || (b && c) || (c && a)
                                                                where
                                                                        a = destinationId  <=  slotId'
                                                                        b = slotId'          <=  myId
                                                                        c = myId           <   destinationId
                        ( DecreasingWithHijack, EmptySlot _) ->  (a && b) || (b && c) || (c && a)
                                                                where
                                                                        a = destinationId  <  myId
                                                                        b = slotId'  <= destinationId
                                                                        c = myId           <= slotId'
                        (_, Invalid)                        -> False
                        _                                   -> slotId'  ==  myId

        a = checkDest <$> sIds
        b =  validForRingCheck vvmToRing
        c =  validForRingCheck vvmCredits
        v= zipWith3 (\ x y z -> x && y && z) a b c

        (pointer' , idx ) = rr4 pointer v

        rrOutC = case idx of
          ( Just  i ) ->  ContentSlot   slotId' myId (sIds !! i)  (vvmToRing !! i)
          _           -> EmptySlot slotId'
-- rout the content
        (cRing'  , toBuffer ,update ) = case (cRingHop , rrOutC ) of
         (EmptySlot   _ , EmptySlot _         )-> (EmptySlot slotId', (Nothing , repeat Nothing), Nothing)
         (EmptySlot   _ , ContentSlot _ _ d v )-> (rrOutC           , (Nothing , repeat Nothing), Just d )
         (ContentSlot _ s b c , EmptySlot _   )
                                | b == myId    -> (EmptySlot slotId', (Just s  , c             ), Nothing)
                                | otherwise    -> (cRingHop         , (Nothing , repeat Nothing), Nothing)
         (ContentSlot _ s b c,  ContentSlot _ _ d v)
                                | b == myId    -> (rrOutC           , (Just s  , c             ), Just  d)
                                | otherwise    -> (cRingHop         , (Nothing , repeat Nothing), Nothing)
         _                                     -> (Invalid          , (Nothing , repeat Nothing), Nothing)

-- amount read from output Message Buffer
        vReadOutgoing  =  zipWith3 fg  sIds (replicate (lengthS sIds)   update)
                                            (repeat (snatToNum (lengthS (fl vvmToRing))))
        vReadCredits  =   zipWith3 fg  sIds (replicate (lengthS sIds)   update)
                                            (repeat (snatToNum (lengthS (fl vvmCredits))))

        fg sid x  lngt = case x of
                Just i | i == sid  ->  lngt
                       | otherwise -> 0
                Nothing -> 0
        fl ::KnownNat m => Vec n ( Vec m a) -> Vec m Bool
        fl x = repeat True

-- new Input to Input Message Buffer
        vvmFromRing =   zipWith f (Just <$> rIds) ( repeat toBuffer )
                        where
                        f a b | a == fst b = snd b
                              | otherwise  = repeat Nothing
-- ============================================================================================
```

## D.3.3 Round-Robin

```
-- ============================================================================================
--  Project           : Mapping Dataflow graphs on multiple FPGAs
--  Faculty           : University of Twente, CAES-group
--  Program name      : RoundRobin.hs
--  Author            : Sander Bremmer
--  Date created      : 31-07-2020
--  Purpose           : Round robin implementation
-- ============================================================================================
module DataflowMultiFPGA.RoundRobin where
```

```
10
11  import Clash.Prelude
12
13  rr4  pointer validList      = (pointer' , out)
14          where
15              a = imap (>=) (replicate (lengthS validList) pointer)
16              b = a ++ (not <$> a)
17
18              c = zipWith (&&) b (validList ++ validList)
19              idx = elemIndex True c
20
21              -- out =     case idx of
22              --                 Just x -> Just £  resize £ mod x (snatToNum (lengthS validList))
23              --                 _       -> Nothing
24
25              out = resize <$> (mod <$> idx <*> Just (snatToNum (lengthS validList)))
26
27
28              pointer' =  case out of
29                                  (Just x) -> if x >= maxBound  then  minBound  else x + 1
30                                  Nothing -> pointer
31  -- ============================================================================================================
```

## D.3.4 Buffer

```
1   -- ============================================================================================================
2   --  Project           : Mapping Dataflow graphs on multiple FPGAs
3   --  Faculty           : University of Twente, CAES-group
4   --  Program name      : Buffer.hs
5   --  Author            : Sander Bremmer
6   --  Date created      : 31-07-2020
7   --  Purpose           : Creating Input and output Buffers
8   -- ============================================================================================================
9
10  {-# LANGUAGE RecordWildCards              #-}
11  module DataflowMultiFPGA.Buffer where
12
13  import Clash.Prelude
14  import DataflowMultiFPGA.DataTypes
15  import DataflowMultiFPGA.Fifo
16
17  buffer states inps didRead= (states', o)
18      where
19          (states' , o )  = unzip $ zipWith fifoNN6 states (zip inps didRead)
20
21  outGoingBuffer  states  ToOutgoingBuffer{..}  =  (states'  ,FromOutgoingBuffer{..} )
22      where
23          (states'  , vvmToRing)   =   buffer     states  vvmOutgoing   vReadOutgoing
24
25  inComingBuffer   states ToIncomingBuffer{..} =  (states' ,FromIncomingBuffer{..} )
26       where
27              (states'  , vvmIncoming)=  buffer states vvmFromRing vReadIncoming
28  -- ============================================================================================================
```

## D.3.5 FIFO

```
1   -- ============================================================================================================
2   --  Project           : Mapping Dataflow graphs on multiple FPGAs
3   --  Faculty           : University of Twente, CAES-group
4   --  Program name      : Fifo.hs
5   --  Author            : Sander Bremmer
6   --  Date created      : 31-07-2020
7   --  Purpose           : Fifo bor the buffers
8   -- ============================================================================================================
9   {-# LANGUAGE RecordWildCards              #-}
10  module DataflowMultiFPGA.Fifo where
```

```haskell

import Clash.Prelude
import DataflowMultiFPGA.HelperFunctions

fifoNN6:: (  KnownNat out,
      KnownNat something,
       KnownNat ls,                -- length state
       KnownNat wd,
      (out + wd - 1 + something ) ~  ls
      ) =>
      Vec ls (Maybe a)             -- state
     -> ( Vec wd (Maybe a),        -- wd
          Index ls)                -- didread
     -> (
          Vec ls (Maybe a),        -- state''
          Vec out (Maybe a)        -- out
        )

fifoNN6  state xs  = (state''   , out )
  where
    (inp, didRead ) = xs
    ls =  lengthS state
    state'  = imap f state
              where
                f idx s | idx < didRead  = Nothing
                        | otherwise = s
    state'' = take ls  $ snd $ mapAccumRL   g Nothing (state' ++ inp)
                where
                     g acc x = case  x of
                      (Just _ )  -> (x, acc)
                      _          -> (acc, x)
    out = takeI state
-- ============================================================================================================
```

## D.3.6 Ring Hop

```haskell
-- ============================================================================================================
--  Project           : Mapping Dataflow graphs on multiple FPGAs
--  Faculty           : University of Twente, CAES-group
--  Program name      : RingHop.hs
--  Author            : Sander Bremmer
--  Date created      : 31-07-2020
--  Purpose           : Slot of nebula ing slot
-- ============================================================================================================
{-# LANGUAGE RecordWildCards , ExistentialQuantification            #-}
module DataflowMultiFPGA.RingHop where

import Clash.Prelude
import DataflowMultiFPGA.DataTypes

ringHop  rhState  ToRingHop{..} =   (rhState', FromRingHop{..})
      where
        cRingHop =  last rhState
        rhState' =  cRing +>> rhState
-- ============================================================================================================
```

## D.3.7 Helper Function

```haskell
-- ============================================================================================================
--  Project           : Mapping Dataflow graphs on multiple FPGAs
--  Faculty           : University of Twente, CAES-group
--  Program name      : HelperFunctions.hs
--  Author            : Sander Bremmer
--  Date created      : 31-07-2020
--  Purpose           : Functions that help other functions
-- ============================================================================================================
```

```haskell
module DataflowMultiFPGA.HelperFunctions where

import Clash.Prelude

-------------------------------------------------------------------------------------------------------------
fromJust def a = case a of
    (Just x) -> x
    _        -> def
-------------------------------------------------------------------------------------------------------------
maybeToBool  x = case x of
        (Just _) -> True
        _        -> False
-------------------------------------------------------------------------------------------------------------
validForRingCheck  inpss = and <$>  (maybeToBool <$> ) <$>  inpss
-------------------------------------------------------------------------------------------------------------

validCheck'' amounts inpss = and (concat o)
    where
      inpssB = (maybeToBool <$> ) <$>  inpss
      inpssBA =   zip <$> inpssB <*> (repeat  <$> amounts)
      o = imap f <$> inpssBA
        where
          f idx (inpsB,amount) | idx' idx >= amount  =  True
                               | otherwise     =  inpsB
            where
              idx'::KnownNat n => Index n -> Index ( n + 1)
              idx' = resize
-------------------------------------------------------------------------------------------------------------
mapAccumRL :: KnownNat n => (y    -> a -> (y  , a))   -> y    -> Vec n a -> (Vec n a, Vec n y)
mapAccumRL f acc e =  mapAccumL i e (replicate ( lengthS  e ) acc)
     where
      i a b =  (b' ,a' )
        where  (a' ,b' ) = mapAccumR f b a

-- mapAccumRL :: KnownNat n => (y    -> a -> (y  , a))   -> y    -> Vec n a -> (Vec n a, Vec n y)
mapAccumRL' :: (y -> x -> (y, x))-> Vec n1 y -> Vec n2 x -> (Vec n2 x, Vec n1 y)
mapAccumRL'  f acc e =  mapAccumL i e acc -- (replicate ( lengthS  e ) acc)
    where
        i a b =  (b' ,a' )
            where  (a' ,b' ) = mapAccumR f b a

f acc x = case fst x of
        (Just _ )  -> (x, acc)
        _          -> (acc, x)

g acc x = case (fst x) of
        True  -> (x, acc)
        _     -> (acc, x)


selector  amounts  vss = zipWith ggg amounts vss

ggg amount vs = init $ imap fff (vs ++ singleton Nothing)
      where
        fff idx v  | idx < amount = v
                   | otherwise    = Nothing
-- ============================================================================================================
```

# D.4 Simulation Example

## D.4.1 Option 1, Ring 1, Modes 0, time 1

```haskell
-- ============================================================================================================
--  Project            : Mapping Dataflow graphs on multiple FPGAs
--  Faculty            : University of Twente, CAES-group
--  Program name       : TestConnect_4_Opt_1_Ring_1_mod_0_char.hs
```

```
5  --  Author              : Sander Bremmer
6  --  Date created        : 31-07-2020 31-07-2020 31-07-2020
7  --  Purpose             : Testing option 1, with ring size 1 , without hijacking , hoptime = 1
8  -- ================================================================================================
9  {-# LANGUAGE RecordWildCards , ExistentialQuantification             #-}
10
11 module TestConnect_4_Opt_1_Ring_1_mod_0_char where
12
13 import Clash.Prelude
14
15 import DataflowMultiFPGA.NodeConnect
16 import DataflowMultiFPGA.DataTypes
17 import DataflowMultiFPGA.RoundRobin
18 import DataflowMultiFPGA.Router
19 import Debug.Trace
20 import qualified Data.List as L
21 import Text.Printf
22 import System.Directory
23 -- ================================================================================================
24 ------- MAKE NODES -------- MAKE NODES -------- MAKE NODES -------- MAKE NODES -------- MAKE NODES -------
25 -- ================================================================================================
26 ------------------------------------------------------------------------------------------------------
27 ---------- INITIAL NODE STATES ----------- INITIAL NODE STATES ------------ INITIAL NODE STATES ----------
28 ------------------------------------------------------------------------------------------------------
29 init_0 = ElementStates   {
30                       ibState      = repeat (repeat Nothing)
31                     , obState      =  repeat (repeat Nothing)
32                     , rState       = 0  -- where pointer start for round robin
33                     , rhState  = EmptySlot 'A' :>Nil
34                     }
35
36 init_1 = ElementStates   {
37                       ibState      = (    (replicate d2 (Just 0) ++ repeat Nothing)
38                                       :> (replicate d2 (Just 0) ++ repeat Nothing)
39                                       :> (replicate d6 (Just 0) ++ repeat Nothing)  :> Nil )
40                     , obState      =   repeat (repeat Nothing)
41                     , rState       = 0
42                     , rhState  = EmptySlot 'B' :>Nil
43                     }
44
45 init_2 = ElementStates   {
46                       ibState      = repeat (repeat Nothing)
47                     , obState      = repeat (repeat Nothing)
48                     , rState       = 0
49                     , rhState  = EmptySlot 'C' :>Nil
50                     }
51 init_3 = ElementStates   {
52                       ibState      = repeat (repeat Nothing)
53                     , obState      = repeat (repeat Nothing)
54                     , rState       = 0
55                     , rhState  =  EmptySlot 'D':>Nil
56                     }
57
58 ------------------------------------------------------------------------------------------------------
59 ---------- FIXED VALUE ----------- FIXED VALUE ----------- FIXED VALUE ----------- FIXED VALUE -----------
60 --------- DATAFLOW INPUT -------- DATAFLOW INPUT -------- DATAFLOW INPUT -------- DATAFLOW INPUT ---------
61 ------------------------------------------------------------------------------------------------------
62
63 --  ::  Setup id   (wd + 1 ) (rd + 1) s r
64 def_0 = Setup { myId    = 'A'              -- :: id
65               , sIds    = 'B' :> Nil       -- :: Vec s id
66               , amountS =  2  :> Nil       -- :: Vec s amount
67               , rIds    = 'B' :> Nil       -- :: Vec r id
68               , amountR =  2  :> Nil       -- :: Vec r amount
69               , modus   =  IncreasingWithoutHijack              -- ::    RoutingMode
70               }
71
72 def_1 = Setup { myId    = 'B'                       -- :: id
73               , sIds    = 'A' :> 'C' :> 'D' :> Nil  -- :: Vec s id
74               , amountS =  2  :>  2  :>  6  :> Nil  -- :: Vec s amount
75               , rIds    = 'A' :> 'C' :> 'D' :> Nil  -- :: Vec r id
```

```
               , amountR =  2  :>  2  :>  6  :> Nil  -- :: Vec r amount
               , modus   =  IncreasingWithoutHijack               -- ::    RoutingMode
               }

def_2 = Setup { myId    = 'C'              -- :: id
               , sIds    = 'B' :> Nil       -- :: Vec s id
               , amountS =  2  :> Nil       -- :: Vec s amount
               , rIds    = 'B' :> Nil       -- :: Vec r id
               , amountR =  2  :> Nil       -- :: Vec r amount
               , modus   =  IncreasingWithoutHijack               -- ::    RoutingMode
               }

def_3 = Setup { myId    = 'D'              -- :: id
               , sIds    = 'B' :> Nil       -- :: Vec s id
               , amountS =  6  :> Nil       -- :: Vec s amount
               , rIds    = 'B' :> Nil       -- :: Vec r id
               , amountR =  6  :> Nil       -- :: Vec r amount
               , modus   =  IncreasingWithoutHijack               -- ::    RoutingMode
               }

-------------------------------------------------------------------------------------------------------------
--------- MEALY OF NODES -------- MEALY OF NODES -------- MEALY OF NODES -------- MEALY OF NODES ---------
-------------------------------------------------------------------------------------------------------------
-- ElementConnect id d f rd sd wd r s a
node_0_M ::HiddenClockResetEnable System =>
     Signal
       System
       --ElementConnect   id    d   f   rd  sd  wd r   s   a                cr  ff
       ( ElementConnect   Char  30  30   2   1   2   1   1   (Unsigned 100)   1  20  )
     -> Signal
          System
          (ElementConnect Char  30  30   2   1   2   1   1   (Unsigned 100)   1   20 )
node_0_M =   mealy  (nodeF  def_0) init_0

node_1_M ::HiddenClockResetEnable System =>
     Signal
       System
       --ElementConnect   id    d   f   rd  sd  wd r   s   a                cr  ff
       ( ElementConnect   Char  30  30   6   1   6   3   3   (Unsigned 100)    1   20 )
     -> Signal
          System
          (ElementConnect Char  30  30   6   1   6   3   3   (Unsigned 100)    1   20 )
node_1_M = mealy (nodeF  def_1) init_1

node_2_M ::HiddenClockResetEnable System =>
     Signal
       System
       --ElementConnect   id    d   f   rd  sd  wd r   s   a                cr  ff
       ( ElementConnect   Char  30  30   2   1   2   1   1   (Unsigned 100)    1   20 )
     -> Signal
          System
          (ElementConnect Char  30  30   2   1   2   1   1   (Unsigned 100)    1   20 )
node_2_M = mealy (nodeF  def_2) init_2

node_3_M ::HiddenClockResetEnable System =>
     Signal
       System
       --ElementConnect   id    d   f   rd  sd  wd r   s   a                cr  ff
       ( ElementConnect   Char  30  30   6   1   6   1   1   (Unsigned 100)    1   20 )
     -> Signal
          System
          (ElementConnect Char  30  30   6   1   6   1   1   (Unsigned 100)    1   20 )
node_3_M = mealy (nodeF  def_3) init_3

-------------------------------------------------------------------------------------------------------------
---------- MAKE Functions ------- MAKE Functions ------- MAKE  Functions ------- MAKE Functions ----------
-------------------------------------------------------------------------------------------------------------

f0  (state) wd =((state), ( out , read))
  where
    (read ,out )  = case wd  of
```

```
                    (Nothing  ) -> (False , state)
                    (Just x   ) -> (True  , inject)
    inject = Just  ( (replicate d2 (Just 01)  )  :> Nil )

f1  (state) wd =((state), ( out , read))
  where
    (read ,out )  = case wd  of
                    (Nothing  ) -> (False , state)
                    (Just x   ) -> (True  , inject)
    inject = Just  ( (replicate d2 (Just 10) ++ repeat Nothing)
                  :> (replicate d2 (Just 12) ++ repeat Nothing)
                  :> (replicate d6 (Just 13)   )  :> Nil )

f2  (state) wd =((state), ( out , read))
  where
    (read ,out )  = case wd  of
                    (Nothing  ) -> (False , state)
                    (Just x   ) -> (True  , inject)
    inject = Just  ( (replicate d2 (Just 21)  )  :> Nil )

f3  (state) wd =((state), ( out , read))
  where
    (read ,out )  = case wd  of
                    (Nothing  ) -> (False , state)
                    (Just x   ) -> (True  , inject)
    inject = Just  ( (replicate d6 (Just 31) )  :> Nil )

f0M = mealy  f0 (Nothing )
f1M = mealy  f1 (Nothing )
f2M = mealy  f2 (Nothing )
f3M = mealy  f3 (Nothing )

-------------------------------------------------------------------------------------------------------------
-- ------- Add Actor --------- Add Actor -------- Add Actor -------- Add Actor -------- Add Actor --------
-------------------------------------------------------------------------------------------------------------
--Actor A ------------------------------------------------------------------------------------------------
actor0 input = bundle ( (cRing' <$> fromNode_)
                      , (vReadCredits <$> fromNode_)
                      , (vvmNewCredits <$> fromNode_)
                      )
  where
    -- (cRing_,  vvmCredits_)  = unbundle input
    (cRing_,  vvmCredits_)  = unbundle input

    (toRing_, read_)                 = unbundle $ f0M  (mvvmFromRing <$> fromNode_)
    fromNode_                      = node_0_M (ToNodeF <$> cRing_
                                               <*> toRing_
                                               <*> read_
                                               <*> vvmCredits_
                                              )
-- Actor B ------------------------------------------------------------------------------------------------
actor1 input = bundle ( (cRing' <$> fromNode_)
                      , (vReadCredits <$> fromNode_)
                      , (vvmNewCredits <$> fromNode_)
                      )
  where
    (cRing_,  vvmCredits_)  = unbundle input
    (toRing_, read_)                = unbundle $ f1M  (mvvmFromRing <$> fromNode_) -- changed to 1
    fromNode_                      = node_1_M (ToNodeF <$> cRing_                  -- changed to 1
                                               <*> toRing_
                                               <*> read_
                                               <*> vvmCredits_
                                              )
-- Actor C ------------------------------------------------------------------------------------------------
actor2 input = bundle ( (cRing' <$> fromNode_)
                      , (vReadCredits <$> fromNode_)
                      , (vvmNewCredits <$> fromNode_)
                      )
  where
    (cRing_,  vvmCredits_)  = unbundle input
    (toRing_, read_)                = unbundle $ f2M  (mvvmFromRing <$> fromNode_) -- changed to 2
```

```
218       fromNode_                       = node_2_M (ToNodeF <$> cRing_                   -- changed to 2
219                                                           <*> toRing_
220                                                           <*> read_
221                                                           <*> vvmCredits_
222                                                   )
223   -- Actor D ---------------------------------------------------------------------------------------------
224   actor3 input = bundle ( (cRing' <$> fromNode_)
225                         , (vReadCredits <$> fromNode_)
226                         , (vvmNewCredits <$> fromNode_)
227                         )
228     where
229       (cRing_,  vvmCredits_)  = unbundle input
230       (toRing_, read_)                = unbundle $ f3M  (mvvmFromRing <$> fromNode_) -- changed to 3
231       fromNode_                       = node_3_M (ToNodeF <$> cRing_                   -- changed to 3
232                                                           <*> toRing_
233                                                           <*> read_
234                                                           <*> vvmCredits_
235                                                   )
236   ---------------------------------------------------------------------------------------------------------
237   -- ------- TopEntity --------- TopEntity -------- TopEntity -------- TopEntity -------- TopEntity --------
238   ---------------------------------------------------------------------------------------------------------
239   topEntity =   actor0  -- FPGA Actor A
240   -- topEntity =   actor1  -- FPGA Actor B
241   -- topEntity =   actor2  -- FPGA Actor C
242   -- topEntity =   actor3  -- FPGA Actor D
243
244
245
246   -- =====================================================================================================
247   ------- SIMULATION -------- SIMULATION -------- SIMULATION -------- SIMULATION -------- SIMULATION -------
248   -- =====================================================================================================
249   ---------------------------------------------------------------------------------------------------------
250   -- --------- CONNECTION NODES ------------ CONNECTION NODES ------------ CONNECTION NODES --------------
251   ---------------------------------------------------------------------------------------------------------
252
253   createRing wd =     bundle  ( mvvmFromRing    <$> fromNode_0
254                               , vReadCredits    <$> fromNode_0
255                               , vvmNewCredits   <$> fromNode_0
256                               , mvvmFromRing    <$> fromNode_1
257                               , vReadCredits    <$> fromNode_1
258                               , vvmNewCredits   <$> fromNode_1
259                               , mvvmFromRing    <$> fromNode_2
260                               , vReadCredits    <$> fromNode_2
261                               , vvmNewCredits   <$> fromNode_2
262                               , mvvmFromRing    <$> fromNode_3
263                               , vReadCredits    <$> fromNode_3
264                               , vvmNewCredits   <$> fromNode_3
265                               --for debug
266                               , vvmFromRing     <$> fromNode_0
267                               , vvmFromRing     <$> fromNode_1
268                               , vvmFromRing     <$> fromNode_2
269                               , vvmFromRing     <$> fromNode_3
270                               , vvmToRing       <$> fromNode_0
271                               , vvmToRing       <$> fromNode_1
272                               , vvmToRing       <$> fromNode_2
273                               , vvmToRing       <$> fromNode_3
274                               )
275         where
276           ( wd_0, read_0, wd_1, read_1, wd_2, read_2, wd_3, read_3) = unbundle wd
277
278           fromNode_0  = node_0_M (ToNodeF <$> (cRing'   <$>  fromNode_3)
279                                           <*> wd_0
280                                           <*> read_0
281                                           <*> pure ( repeat (repeat (Just 0))))
282
283           fromNode_1  = node_1_M (ToNodeF <$> (cRing'   <$>  fromNode_0)
284                                           <*> wd_1
285                                           <*> read_1
286                                           <*> pure ( repeat (repeat (Just 0))))
287
288           fromNode_2  = node_2_M (ToNodeF <$> (cRing'   <$>  fromNode_1)
```

```
                                      <*> wd_2
                                      <*> read_2
                                      <*> pure ( repeat (repeat (Just 0))))

        fromNode_3  = node_3_M (ToNodeF <$> (cRing'   <$>  fromNode_2)
                                      <*> wd_3
                                      <*> read_3
                                      <*> pure ( repeat (repeat (Just 0))))
-------------------------------------------------------------------------------------------------------------
--------- Add Functions --------- Add Functions --------- Add  Functions --------- Add Functions ---------
-------------------------------------------------------------------------------------------------------------
wrappFunction  n = bundle (f0_in, f0_out, f0_read , readCr_0, newCredits0,
                           f1_in, f1_out, f1_read , readCr_1, newCredits1,
                           f2_in, f2_out, f2_read , readCr_2, newCredits2,
                           f3_in, f3_out, f3_read , readCr_3, newCredits3,
                           --for debugging
                           fr0  , fr1   , fr2     , fr3,
                           tr0  , tr1   , tr2     , tr3)
  where
   (  f0_in , readCr_0, newCredits0,
      f1_in , readCr_1, newCredits1,
      f2_in , readCr_2, newCredits2,
      f3_in , readCr_3, newCredits3,
      -- for debugging
      fr0   , fr1     , fr2   , fr3,
      tr0   , tr1     , tr2   , tr3
      ) = unbundle $ createRing $ bundle (f0_out, f0_read,
                                          f1_out, f1_read,
                                          f2_out, f2_read,
                                          f3_out, f3_read
                                          )
   (f0_out,f0_read) = unbundle $ f0M  f0_in
   (f1_out,f1_read) = unbundle $ f1M  f1_in
   (f2_out,f2_read) = unbundle $ f2M  f2_in
   (f3_out,f3_read) = unbundle $ f3M  f3_in


-------------------------------------------------------------------------------------------------------------
----------------- CLEAR DISPLAY IN TERMINAL ------------------ CLEAR DISPLAY IN TERMINAL -----------------
-------------------------------------------------------------------------------------------------------------

wrapperFuncOutput  = simulate_lazy @System wrappFunction [1..]

result clocks = unlines [
                            header
                          -- , L.intercalate "" (L.replicate (L.length (header L.++ "")) "-") L.++ ""
                          , unlines $ ( formatLine <$> (L.zipWith (,) counter wrapperFuncOutput ))
                        ]
      where
          counter = [1..clocks]
          -- format for screen display
          -- format       = "%-3s|%-80s|%-80s|%-10s|%-20s|%-20s|%-20s|%-20s|" L.++
          --                  "   %-3s|%-80s|%-80s|%-10s|%-20s|%-20s|%-20s|%-20s|" L.++
          --                  "   %-3s|%-80s|%-80s|%-10s|%-20s|%-20s|%-20s|%-20s|"

          -- format without debug for text file
          -- format       = "%s|%s|%s|%s|%s|%s|         %s|%s|%s|%s|%s|%s|" L.++
          --                  "   %s|%s|%s|%s|%s|%s|          %s|%s|%s|%s|%s|%s|"
          -- format with debug
          format       = "%s|%s|%s|%s|%s|%s|%s|%s|   %s|%s|%s|%s|%s|%s|%s|%s|" L.++
                           "   %s|%s|%s|%s|%s|%s|%s|%s|%s|   %s|%s|%s|%s|%s|%s|%s|%s|%s|"
          header       = printf format
              "#"
              "Actor A Consuming edge(s)"
              "Actor A Producing edge(s) "
              "Actor A Read"
              "A Writes To"
              "A Produces new Credits"
              "From Router A to Incoming Buffer" -- debug
              "To Router A from Outgoing Buffer" -- debug
```

```
            "#"
            "Actor B Consuming edge(s)"
            "Actor B Producing edge(s) "
            "Actor B Read"
            "B Writes To"
            "B Produces new Credits"
            "From Router B to Incoming Buffer" -- debug
            "To Router B from Outgoing Buffer" -- debug

            "#"
            "Actor C Consuming edge(s)"
            "Actor C Producing edge(s) "
            "Actor C Read"
            "C Writes To"
            "C Produces new Credits"
            "From Router C to Incoming Buffer" -- debug
            "To Router c from Outgoing Buffer" -- debug

            "#"
            "Actor D Consuming edge(s)"
            "Actor D Producing edge(s) "
            "Actor D Read"
            "D Writes To"
            "D Produces new Credits"
            "From Router D to Incoming Buffer" -- debug
            "To Router D from Outgoing Buffer" -- debug


        formatLine  (cnt, (f0_in, f0_out, f0_read, readCr_0, newCredits0,
            f1_in, f1_out, f1_read, readCr_1, newCredits1,
            f2_in, f2_out, f2_read, readCr_2, newCredits2,
            f3_in, f3_out, f3_read, readCr_3, newCredits3,
            fr0,fr1,fr2, fr3,
            tr0, tr1, tr2,tr3))   = printf format
              (show cnt      ) -- clk c
              (g f0_in    ) -- to fucntion from ring
              (g f0_out   ) -- from function to ring
              (b f0_read  ) -- did function read
              (show readCr_0) -- amount of credits read
              (show ((map g) <$>  newCredits0)) -- new credits after sending something out
              (show ((map g) <$>  fr0))  -- debug  -- from router to incoming buffer
              (show ((map g) <$>  tr0)) -- debug

              (show cnt      )
              (g f1_in    )
              (g f1_out   )
              (b f1_read  )
              (show readCr_1)
              (show ((map g) <$>  newCredits1))
              (show ((map g) <$>  fr1))-- debug
              (show ((map g) <$>  tr1))-- debug

              (show cnt      )
              (g f2_in    )
              (g f2_out   )
              (b f2_read  )
              (show readCr_2)
              (show ((map g) <$>  newCredits2))
              (show ((map g) <$>  fr2))-- debug
              (show ((map g) <$>  tr2))-- debug

              (show cnt      )
              (g f3_in    )
              (g f3_out   )
              (b f3_read  )
              (show readCr_3)
              (show ((map g) <$>  newCredits3))
              (show ((map g) <$>  fr3))-- debug
              (show ((map g) <$>  tr3))-- debug
          where
            g a = case a of
```

```haskell
                    (Just x) -> show a
                    Nothing  -> "N"
              b a = case a of
                    True -> show a
                    _    -> "F"
ggg a = case a of
          (Just x) -> show a
          Nothing  -> "N"

-------------------------------------------------------------------------------------------------------------
------ Test ----- Test ----- Test ----- Test ----- Test ---- Test ----- Test ----- Test ----- Test -------
-------------------------------------------------------------------------------------------------------------
-- test to display output on screen
testWrappFunc clocks = do putStrLn $ result clocks

-- test to display output to file

fileName = "\\Results\\TestConnect_4_opt_1_Ring_1_mod_0_char3333.txt"

writeToFile  fileName clocks= do
    dir <- getCurrentDirectory
    appendFile (dir L.++ fileName)  ( [x | x <- (result clocks) ])
wrt =  writeToFile fileName 250   -- write with predefined filename  and  clocks
-- ============================================================================================
```