BUILDING A FRAMEWORK IN CLASH TO CREATE DETERMINISTIC SENSOR AND ACTUATOR INTERFACES FOR FPGA

Master Thesis

Abstract

This master Thesis is part of a bigger research project at CAES. The goal of this project is to build a framework in Clash to create deterministic systems with sensors and actuators. This Thesis focusses on the communication aspect with the sensors and actuators. Research will be done on how to design and analyze a framework that creates deterministic interfaces for the connected sensors and actuators.

Supervisors

Dr.Ir. A.B.J. Kokkeler

H.H.Folmer Msc

Dr.Ir. R. Langerak

Bart Wijlens

b.wijlens-1@student.utwente.nl

CONTENTS

1 Introduction 4
2 Background 5
2.1 Explaination terms 5
2.1.1 Determinism 5
2.1.2 Framework 5
2.2 Clash 5
2.3 Dataflow 6
2.3.1 What is dataflow? 6
2.3.2 Backpressure 7
2.3.3 Self edges 7
2.3.4 periodic behaviour 8
2.3.5 Assumptions and info 9
2.4 Framework within main project 9
2.5 Goal 10
2.6 Design process 10
3 Designing framework 11
3.1 Sensor interface in Detail 11
3.2 Use of Dataflow in Framework 11
3.2.1 Hardware perspective 12
3.2.2 Theoretical perspective 12
3.3 Dataflow and Hardware 12
3.3.1 Structural dataflow 12
3.3.2 Functional Dataflow 13
3.4 Final Solution 13
3.5 Improve performance Framework 15
4 Implement framework in hardware 17
4.1 framework in hardware 17
4.1.1 Backpressure 17
4.1.2 Nodes 19
4.2 Synchronization 20
4.2.1 Mean time between failures for synchronizer 20
4.2.2 Synchronizer Solutions 22

4.3 Framework contents 24
4.3.1 Communication nodes 25
4.3.2 Controllers 26
4.3.3 Others 28
4.4 Framework Implementations for Sensor interface 29
4.4.1 Communication interface 29
4.4.2 Sensor driver 29
4.4.3 Synchronization 29
4.4.4 IO sampler 33
5 Results 34
5.1 Test process 34
5.1.1 Test board 34
5.1.2 Test process individual components 34
5.1.3 Test process implementation 34
5.2 Testing framework 41
5.2.1 Testing individual components 41
5.2.2 Testing implementation 42
6 Conclusion 46
7 Discussion and Future work 47
8 Appendix 48
A.1 Communication protocols 48
A.1.1 SPI 48
A.1.2 GPIO 48
A.1.3 I2C 48
A.1.4 UART 48
A.2 MTBF for synchronizer in case frequencies aren’t multiples 49
A.3 Deriving dataflow from clash 52
A.3.1 Dataflow Abstraction 52
A.3.2 Derivation process 53
A.3.3 Other Nodes 57
A.4 Adding Controllers 63
A.5 MTBF calculation for synchronizer sensor 65
A.6 Results simulation 67

A.6.1 Frequencies are equal 67
A.6.2 Frequencies are multiple 68
A.6.3 Frequencies are not a multiple 69
9 References 70

1 INTRODUCTION

Now a days cars are equipped with hundreds of sensors and actuators, all in place to improve the reliability and safety. While progress has been made with respect to smart usage of these sensor and actuators, the real time aspect is still hard to guarantee. When a sensor sends out a signal, the data often needs to go through a CPU before the corresponding actuator can be controlled accordingly. This single CPU is used by multiple sensors meaning there is chance of unforeseen delays, other processes or sensors could still be using the processor. There exist solutions which can overcome these unforeseen delays, strict scheduling for example. However, the complexity and unpredictability of most systems makes it hard to implement these techniques.

A CPU schedules its processes to guarantee that the deadline for each process is met. This scheduling is only possible if all processes are known beforehand, when an unforeseen process arrives like pressing the breaks in a car it can result in scheduling problems. At that moment the CPU has two options, delay the current process or delay the new process. This trade off leads to unforeseen delays which can cause processes to miss their deadline. Missing deadlines can lead to system failures or slow responses. Most systems are tested thoroughly meaning that the chance of this happening is extremely small but no hard guarantees can be given.

The CAES group at the University of Twente is working on a solution to make systems with sensors and actuators better analysable. This analysability can be used to give better guarantees with respect to timing. One way to achieve this is by replacing the CPU for an FPGA. FPGAs contain in contrast to CPUs reconfigurable hardware, the hardware can be configured according to the purpose of the FPGA. Since a FPGA can be configured it shares less resources than a CPU allowing it to run multiple processes separately from each other. Separating processes removes most of the scheduling problems present in a CPU. The goal of the project is to build a framework that can be used to create deterministic hardware designs for system with sensors and actuators. The development of the framework is done in Clash, a Haskell based tool used for hardware description.

This thesis focusses on the sensor communication aspect and answers the question: “How to design a framework in Clash to create deterministic sensor interfaces for FPGAs?”. To solve this question there started with searching for a method on how to analyse and prove determinism. This method will be used to design the framework and to guarantee determinism. In the end the framework will be used to create a test setup to show how it can be used to create deterministic sensor interfaces.

2 BACKGROUND

2.1 EXPLAINATION TERMS

2.1.1 DETERMINISM

A deterministic system is a system that behaves in a predictable way so there is no randomness involved. When an input is given to the system it should be known beforehand when the corresponding output is generated. As an example a CPU will be used, a CPU is deterministic when all its processes are known beforehand. These processes can be scheduled resulting in no random behaviour. In most cases however there is an OS in the way that also generates processes. These processes are not known beforehand but have to be scheduled as well. This will results in unpredictable execution times for the different processes which makes the system nondeterministic.

2.1.2 FRAMEWORK

A framework is a set of tools and functions which can be used to create system with functionality the framework was make for. For example a framework for audio processing could consist of a set of different processing functions that can be combined to create an audio processing pipeline. There are a number of reasons why frameworks are useful:

- ➔ It makes it easier for users to implement the functionalities the framework is made for.
- ➔ Frameworks can give guarantees when used according to its specifications.
- ➔ A framework is already tested meaning it is guaranteed to work.
- ➔ Allows reusing of code
- ➔ Easily expandable

2.2 CLASH

Clash is a tool developed within the research group CAES at the University of Twente (Uchevler, Svarstad, Kuper, & Baaij, 2013) and is now part of the company Qbaylogic. The tool has been developed as a new way of designing hardware. Instead of using traditional hardware description languages (HDLs) like VHDL and Verilog, this tool uses the functional language Haskell. The Clash-language is a subset of Haskell which means all code written in the Clash-language can run within the Haskell environment, the other way around is not guaranteed. The code written is converted by the Clash-compiler to the traditional HDLs (VHDL, Verilog).
The tool is still in development meaning there can be bugs and parts of the syntax can change over time. The version used is a development version of 0.99.3.
Working with Clash is different compared to traditional HDLs, Clash approaches problems from a functional standpoint while HDLs take the hardware approach. It is still possible to mimic HDLs however but this is not recommended. When Clash is

used correctly it can improve the development speed and reliability of FPGA/hardware projects. There are a number of ways in which Clash surpasses HDLs in terms of functionality:

- In most cases the code written in Clash is smaller than in other languages, a functional description is more compact.
- The time to simulation is extremely short, code written can be simulated within seconds. This is a big improvement compared to programs like ModelSim and Vivado which can take tenths of seconds.
- Clash makes use of Haskells automatic type derivation which determines types at compile time making designs less type bound. When used correctly Clash only needs one location where the variable types are defined meaning that sizes and types of variables can be easily changed without having to go through all IPs separately.
- The possibility of dynamic hardware generation, this is a functionality of Haskell which can be used in Clash. When parts of a function are known beforehand the function is simplified by already filling in the known parts. The hardware generated from this is based on the simplified function making it possible to generate hardware depending setting variables, variable ranges or variable types.

2.3 DATAFLOW

Dataflow is used throughout the report for analysis since it is deterministic by definition. Dataflow doesn't contain random behaviour or choice.

2.3.1 WHAT IS DATAFLOW?

Dataflow graphs are diagrams consisting of nodes (circles), edges (arrows) and tokens (black circles) (Figure 1) (Edward A. Lee T. M., May 1995) (Edward A. Lee D. G., October 1987) (Grootte, 2016). Edges connect nodes to indicating there is communication. The "data" that is communicated between the different nodes is represented by tokens. Depending on the type of dataflow used the tokens are stored on a node or an edge, in this thesis there is chosen to have edges store the tokens. Edges can store more than one token, the number of tokens stored is indicated by a number near the token.
When a node has at least one token at all of its input edges it is able to "fire", this means it consumes a token from all of its input edges and creates a token at all of its output edges. A delay can be added between consuming and firing this is indicated inside the node.

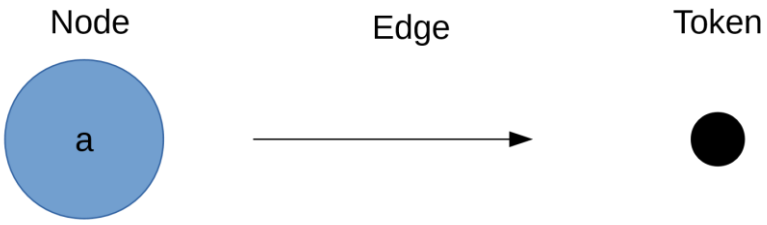


Figure 1: Dataflow components

2.3.2 BACKPRESSURE

Backpressure in dataflow means two nodes are connected with edges in both directions (Figure 2). This is useful to limit the number of tokens that can be stored on an edge. The tokens on both edges combined is the maximum number of tokens that can be stored.

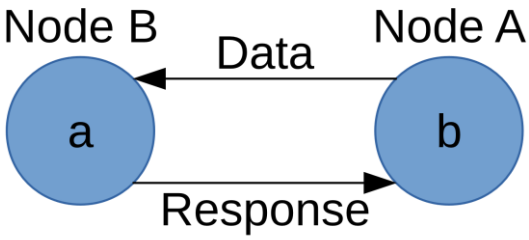


Figure 2: Dataflow backpressure

2.3.3 SELF EDGES

A self-edge in dataflow is an edge from a node that is connected to the same node as is shown in Figure 3.

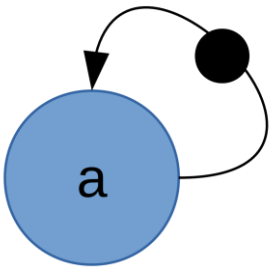


Figure 3: dataflow self-edge

Self-edges are used to limit the number of times a node can execute concurrently, the limit is set by the number of tokens on the self-edge. Without the self-edge the node can run in parallel until all the input tokens are consumed.

2.3.4 PERIODIC BEHAVIOUR

Under some condition dataflow is periodic, this is important since it determines how well the graph performs. A dataflow graph is periodic when it is strongly connected, this means that each node can be reached from every node in the graph (Bekooij, 2017). The period in this case can be calculated with the mean cycle ratio (MCR). The MCR is equal to the slowest period of all loops in the dataflow graph. A loop is the path taken by a token when it leaves a node and returns without passing the same node twice. The period of a loop is calculated by dividing the time it takes a token to make the loop by the number of tokens in the loop. A simplified formula is given below:

$$MCR = \max(all\ p)$$

$$p = \frac{delay\ loop}{tokens\ in\ loop}$$

To show how the MCR works an example is given.

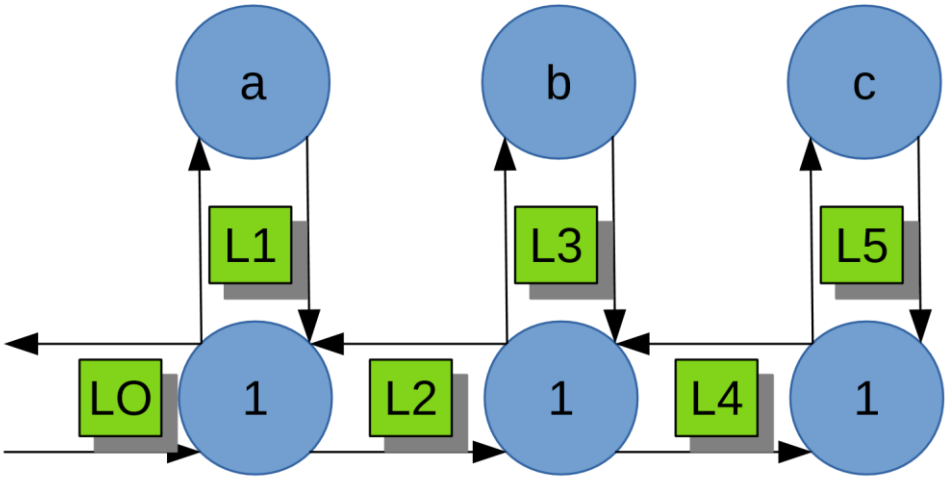


Figure 4: Dataflow with loop periods

The dataflow in Figure 4 has names for each loop (L1 till L5), the output period is given by LO. To calculate the output period of LO the period of each loop needs to be determined:

$$L1 = a + 1$$

$$L2 = 1 + 1 = 2$$

$$L3 = b + 1$$

$$L4 = 1 + 1 = 2$$

$$L5 = c + 1$$

It is assumed the number of tokens on each loop equals 1 so the output period can be determined by taking the maximum of all loops.

$$LO = \max(L1, L2, L3, L4, L5)$$

It there would be self-edges than these should be included in the MCR calculation as well.

2.3.5 ASSUMPTIONS AND INFO

There are a few assumptions made in this report that are important to know. To start the delays for the nodes are always multiples of 1 clock cycle. There is chosen to have multiples of one clock cycle to keep the dataflow in sync with the clock. A second assumption are the implicit self-edge, all pictures in the report assume the nodes have self-edges.

2.4 FRAMEWORK WITHIN MAIN PROJECT

As mentioned in the introduction the framework is going to be part of a bigger project, this means it has to be fitted in somehow. It is important to know where the framework will be located within this project because it will influence certain design choice made later on. The layout used for the main project is shown in Figure 5, the sensor interface is the focus of this thesis.

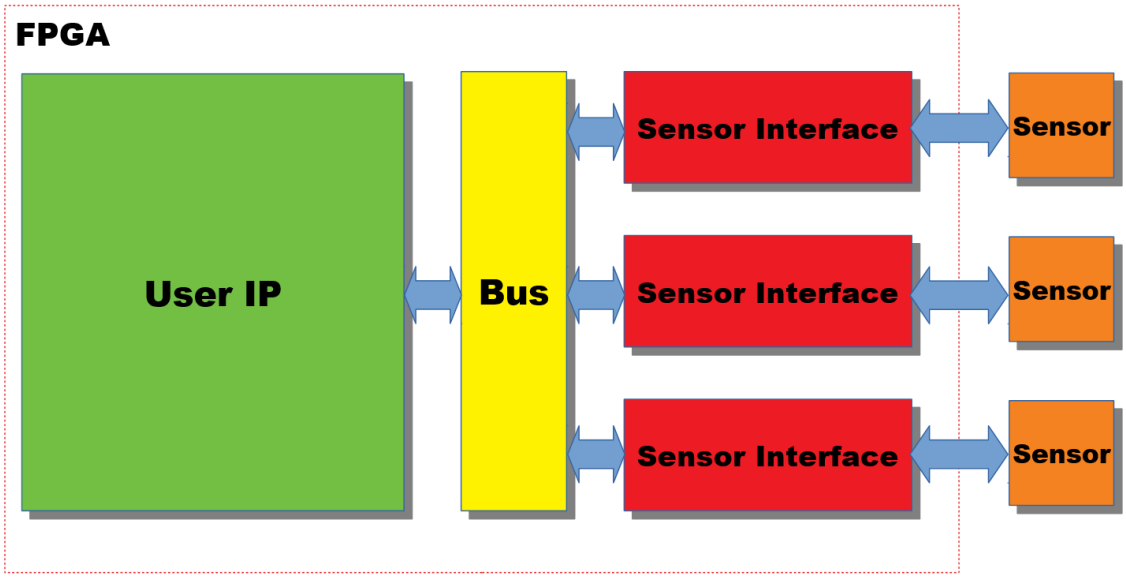


Figure 5: General layout

The layout consists of four parts:

Sensor
Represents the sensors and actuators connected to framework of the main project.

Sensor interface
The sensor interface is responsible for the communication between the sensors and actuator and the bus. It converts data from the bus to something the sensor can use and the other way around. The sensor interface is going to be implemented as a framework and is the focus of this thesis.

Bus
Guides the data from the User IP to the correct sensor interface

User IP
The user IP is responsible for the functional behaviour. It is responsible for reading the sensor data and controlling the corresponding actuator, its behaviour should be fully deterministic.

2.5 GOAL

The goal of the project is to create framework which consists of a set of building blocks which can be combined to create sensor/actuator interfaces for FPGAs. The designs made with these blocks should be easily analysable and must guarantee deterministic behaviour at all times. To describe the blocks in hardware the Clash-language is used. The design should be as small as possible in terms of hardware since they are going to be implemented on a FPGA.

2.6 DESIGN PROCESS

The framework will be designed in a number of steps. There is started with exploring the required functionality of the framework in more detail. Next there will be looked at how to design a framework using dataflow. The designed framework will be tested and used to create a test setup to show it can create deterministic sensor interfaces.

3 DESIGNING FRAMEWORK

3.1 SENSOR INTERFACE IN DETAIL

The sensor interface is explored in more detail to get a better idea on what the sensor interface does and how to build a framework for it. As said before the sensor interface is responsible for translating the communication between the sensor and user IP. This process is split up into four different blocks to make analysis easier but also to make it better suitable for a framework (Figure 6).

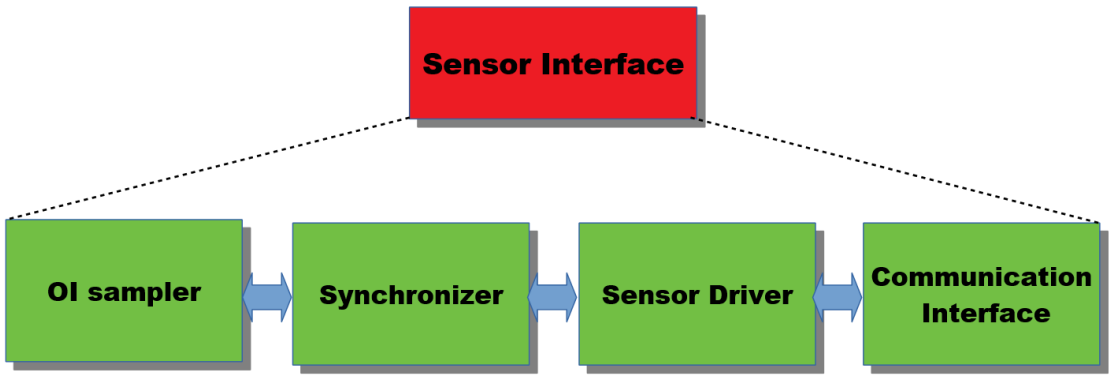


Figure 6: Sensor Interface

As can be seen from the figure the sensor interface will be implemented as a pipeline where each block will have its own functionality as described below:

The **Communication Interface** is responsible for translating the sensor/actuator protocol (SPI, I2C, UART, …) to something that can be used by the sensor driver.

The **Sensor Driver** is responsible for communicating with the sensor/actuator and initializing the sensor if necessary. The sensor driver handles commands from the user IP and sends sensor data back.

The **Synchronisation** is responsible for handling the communication between different clock domains. Most of the times the clock frequency for a sensor is different from that of the users IP meaning some sort of synchronization is necessary.

The **OI sampler** acts as the front end of the sensor, it can receive input data for the sensor and send output data to the user. This block compensates the variable delay of the synchronizer as will be explained later.

3.2 USE OF DATAFLOW IN FRAMEWORK

As explained earlier, dataflow is going to be used for analysing and proving the deterministic behaviour of the framework. However it is not yet defined how dataflow is going to be used in the framework. There is looked at two different approaches. When taking the hardware approach a hardware design is made that is represented

in dataflow. The dataflow representation is used to show the hardware design is deterministic. If the theoretical approach is taken a dataflow design is made which is then represented in hardware. In this case the hardware is deterministic because it is derived from dataflow.

3.2.1 HARDWARE PERSPECTIVE

The framework will consist of a set hardware blocks that can be combined to create different sensor interfaces. Each blocks will perform one of the functions shown in Figure 6 (Communication interface, driver, synchronizer or IO sampler). Different blocks can be made for each function to create different sensor interfaces. The blocks will be designed using the Clash-language meaning they are directly implemented in hardware. Taking this approach is useful since it gives a lot of flexibility with respect to hardware design choices. Being close to the hardware allows designing with optimization in mind which results in smaller and faster designs. The downside is that hardware is not deterministic by definition.
Proving the determinism is done by deriving a dataflow equivalent of each block. If the conversion is possible determinism is guaranteed.
From a user perspective this approach works as follows. The user connects different hardware blocks to create a sensor interface. The hardware blocks all have a dataflow equivalent, these can be used by the user to do analysis on the design.

3.2.2 THEORETICAL PERSPECTIVE

The theoretical perspective starts with dataflow. The framework will consist of a set of dataflow nodes that can be combined to create a sensor interface. The nodes have a hardware equivalent that will be used to implement the design. This approach guarantees determinism since it is based on dataflow. From a user perspective this solution will consist of a number of nodes, each with its own functionality. The user can combine these node to create a sensor interface. The hardware is a functional equivalent of the dataflow graph meaning the properties of the dataflow graph do also hold for the hardware.

3.3 DATAFLOW AND HARDWARE

To use dataflow in combination with hardware a link between dataflow and hardware needs to be defined. This link is mainly based on how tokens are interpreted, dataflow components get a different meaning based on the interpretation of tokens. Two different definitions are explained here called structural dataflow and functional dataflow.

3.3.1 STRUCTURAL DATAFLOW

For structural dataflow a token represents a signal in hardware. A signal can contain data or nothing. Signals will change every clock cycle meaning a new token needs to be produced each cycle. The nodes in the graph represent hardware components

like multiplexers and OR gates. The edges represent the wires between these components. A wire in hardware does not have memory meaning an edge can only store one token at a time. All nodes have a delay of 1 clock cycle to guarantee new tokens each clock cycle.
The definition of data is lost with this approach, it is not known when a token contains data or nothing. For example, a node can produce data every 5 clock cycles, in dataflow this would mean it produces 4 tokens that contain “nothing” and one that contains the data. The contents of these 5 tokens cannot be observed meaning each one them could all be a “nothing” or “data”.
This representation can be used to prove that hardware is deterministic since it guarantees a new signal every clock cycle. The graph doesn’t say anything about the deterministic behavior of the data itself. Data could still be outputted at random.

3.3.2 FUNCTIONAL DATAFLOW

When using the functional dataflow representation a token represent data in contrast to the structural dataflow where a token could be data as well as nothing. The nodes represent functions that use the input tokens to create an output. The time a node takes to do this must be multiple of 1 clock cycle to keep all nodes in sync with the clock. The edges can be seen as FIFOs in hardware, the size of the FIFO is limited meaning it can only hold a limited amount of data. This limitation is controlled by implementing backpressure, the number of tokens in the backpressure loop indicate the size. Adding backpressure makes the dataflow graph automatically strongly connected which means the dataflow graph is periodic. The delay of each of the nodes depend on their functional description in hardware, it should be at least one when it has internal storage. As a rule of thumb can be used: if a node has more than one input and output edge it has internal storage. The tokens only represent data meaning this representation can be used to prove the deterministic behavior of data.

3.4 FINAL SOLUTION

A decision needs to be made on how to design the framework. There are two choices that need to be made, is the structural or functional representation of hardware going to be used and is the framework going to be designed according to the theoretical or the hardware perspective.

Deciding whether to use the functional or structural representation of hardware is easy. The dataflow graph should proof the deterministic behaviour data resulting in the only possible representation, the functional one. This means nodes will represent functions and tokens represent data.

For deciding on how to design the framework itself the hardware approach seemed to be the best solution at first. A set of deterministic hardware blocks that is connected one after the other to create a pipeline looked like a simple and elegant solution. The fact that it would use less hardware than its theoretical counterpart made it even better. The blocks in the pipeline would share the same interface making it easy to

interchange them for a block with different functionality.
It turned out that this design approach results into problems. The freedom given while designing the hardware blocks resulted in nondeterministic designs. This made it necessary to formulate a design rule to prevent these design mistakes: “The output period or delay should not be influenced by nondeterministic inputs”. A second problem became clear later in the process, the timing. If one of the processes in the pipeline is slower than the blocks before it, the system would break since data can be lost. At first glance this was a problem that could be overcome by controlling the delay of the different blocks. The constraint limited the flexibility of the framework since certain combinations of blocks could result into timing problems. Besides the constraints for the framework itself there was also a problem with the processes after the IO sampler. It the processes after the sensor interface would be slower than then the output period of the sensor interface it would also result in data loss. To solve the delay problem a feedback system was added, each receiving block should indicate to the sending block if it is ready. The sending block is halted until the receiving block is able to process the data. Implementing these halt signals needed a redesign for each block.
The added complexity in terms of feedback reduced the hardware gain and prevented the relative ease with which blocks could be designed. Converting the hardware designs to dataflow often resulted in complex dataflow graph or no dataflow graph at all. This made is hard to prove determinism and do analysis on the designs.

The first approach turned out to be more complex than initially thought, this was the reason to try the theoretical approach. Approaching the problem from dataflow perspective directly gave much better results. Analysis could be done beforehand since the dataflow calculation methods could be used. The timing problem was covered by the backpressure required by the functional hardware representation. The added backpressure made the whole system become self-timed. The designing of nodes turned out to be easier than expected, the fact that dataflow rules should apply gave a good guidance during the design process.
In the end there is chosen to use the theoretical approach because it turned out to be better analysable and easier to implement.

The final solution for the framework uses a theoretical perspective, this means designs will be made in dataflow which are later converted to hardware. The dataflow will use the functional representation of hardware meaning each node represents a function instead of a hardware components. The design from Figure 6 is converted accordingly as is shown in Figure 7.

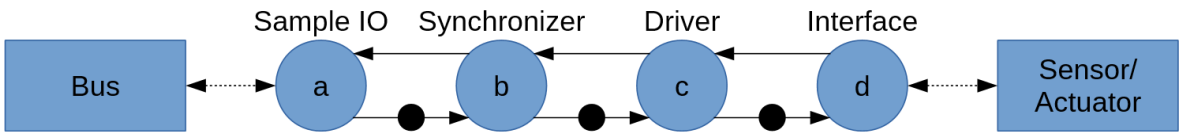


Figure 7: Dataflow sensor interface

The blocks from Figure 6 are converted to dataflow nodes. Each node is connected with backpressure to limit the storage on the edges.

3.5 IMPROVE PERFORMANCE FRAMEWORK

The design from Figure 7 is not very efficient. With this design the nodes can only fire when both the previous and next node are finished. This structure limits parallel processing which can be shown by calculating the MCR:

$$period = \max(a + b, b + c, c + d, a, b, c, d)$$

The output period equals the period of two nodes added.
By separating the communication aspect from the controller/functional part the parallelism can be improved (Figure 8). The ctrl node represent the functional part which is called controller from now on. The controller is responsible for processing incoming data and controlling the connected communication node. The communication node is responsible for distributing incoming data to the next communication node and the controller. This node has a delay of 1 clock cycle because it has more than one input and output edge. The red arrows show how the data from the tokens goes through the system. As can be seen from the picture, data only goes in one direction. This choice is made to prevent forcing the input and output data to go through the same processing pipeline. Most of the times input data will require different processing than output data.

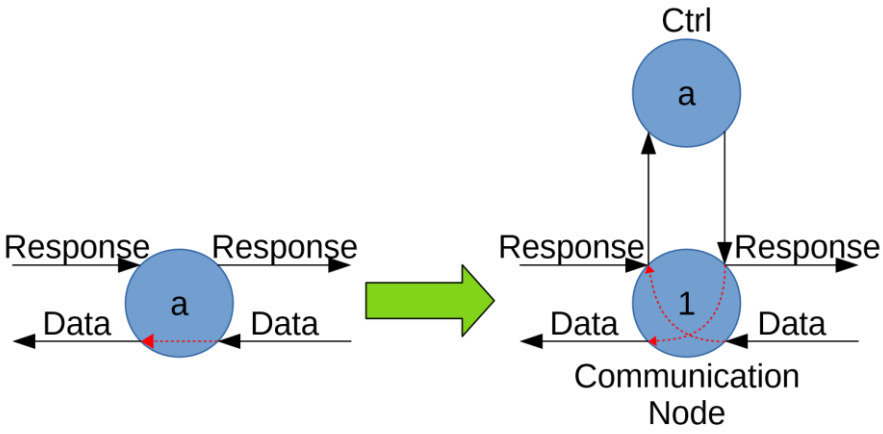


Figure 8: General layout framework

When this new design is put in Figure 7 it results in a design with communication nodes at the bottom and controllers at the top (Figure 9).

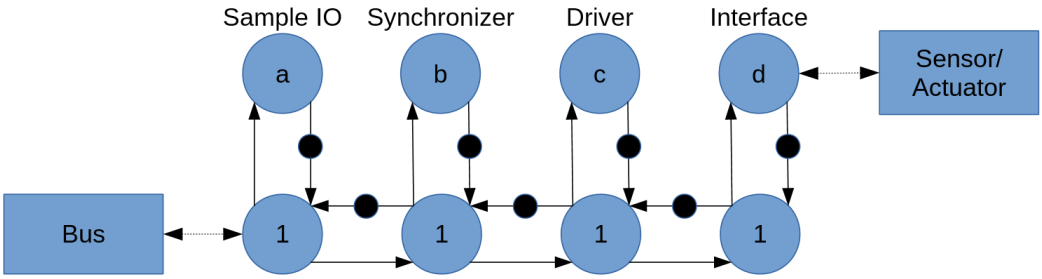


Figure 9: Dataflow general layout improved parallelism

The MCR is calculated for the new design:

$$period = \max(a+1, b+1, c+1, d+1, 2, a, b, c, d, 1)$$

The output period now equals the delay of one of the controller plus the delay of a communication node. It is assumed the controllers will have a delay bigger than the 1 clock cycle from the communication node. Based on this assumption it can be concluded that the period will be smaller than for the previous solution (Figure 7).

4 IMPLEMENT FRAMEWORK IN HARDWARE

4.1 FRAMEWORK IN HARDWARE

For the conversion from dataflow to hardware a standard procedure is used. There are two parts that need to be discussed, the backpressure and the nodes. Here the implementation of these two dataflow elements is explained.

4.1.1 BACKPRESSURE

Nodes/Edges/Backpressure.hs

Designing backpressure in hardware is a complex operation. The backpressure needs to store tokens (data) but cannot have a delay. To solve this problem there is started with a definition of tokens and edges in hardware. The choice is made to represent an edge as two lines, a data line and a control line. The control line is used for indicating tokens on the edge. When the control line is high it indicates there is a token present, when the line is low there is no token. The data inside the token is handled by the separate data line which sends the data in parallel. The values for the data and control line are stored at the output buffer of the node sending the token. When a node sends out a new token its output buffer is updated, the control signal is set to high to indicate the new token while the data line gets the new data. The backpressure is implemented with only one token in the loop, this means that new data can only be send when the previous data has been read.

When the backpressure edges are implemented in hardware as described above it would result in a working backpressure design (Figure 10). Keep in mind the nodes in the figure represent the hardware equivalent of a node as will be explained later.

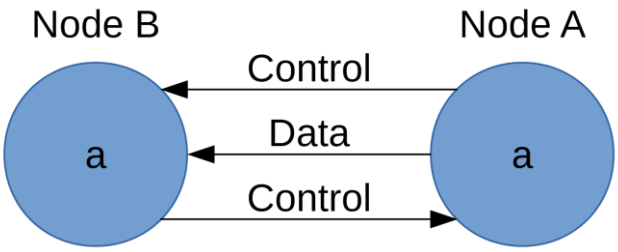


Figure 10: Backpressure in hardware

The problem with this design is the following. When “Node A” sends a token because its input control line is high (high indicates a token), it has to wait at least one clock cycle before the input control signal is updated by “Node B”. This delay is the result of “Node B” having to update its output buffer. This means that “Node A” cannot read its inputs for one clock cycle after sending because they are not updated yet. This is not according to dataflow specifications and makes node design unnecessary complex. The nodes now need to have logic for their own behavior as well as well as logic for controlling the backpressure.
A solution needs to be found on how to get rid of the delay and preventing nodes

from having to worry about backpressure logic. This problem is solved by implementing the backpressure as a state machine who switches between two circuit layouts, the state is controlled by the controller (Figure 11). The design shown here represents the backpressure between node A and node B (Figure 12). The state is switched when there is a signal edge on one of the input signals. The signal edges are used because it makes synchronization easier later on. The switching of state takes one clock cycle, to prevent strange behavior the input and output node should have at least a delay of one clock cycle. The inverters which connect the input to the output of a node act as a direct response to the input. It looks like the token on the incoming edge is consumed at the moment a token is put on the outgoing edge. This is not the exact behavior of dataflow since dataflow consumes tokens at the start of its execution. However, consuming the token at “firing” time doesn’t influence the dataflow behavior.
There are “signal edge” symbols shown next to the input lines, these indicate the signal is a signal edge instead of the signal itself. This line is high when the input signal changes and low when it equals its previous value. The switching of state is done by the controller according to the following rules:

- ➔ If State is X and there is an edge on output A the state goes to Y
- ➔ If State is Y and there is an edge on output B the state goes to X

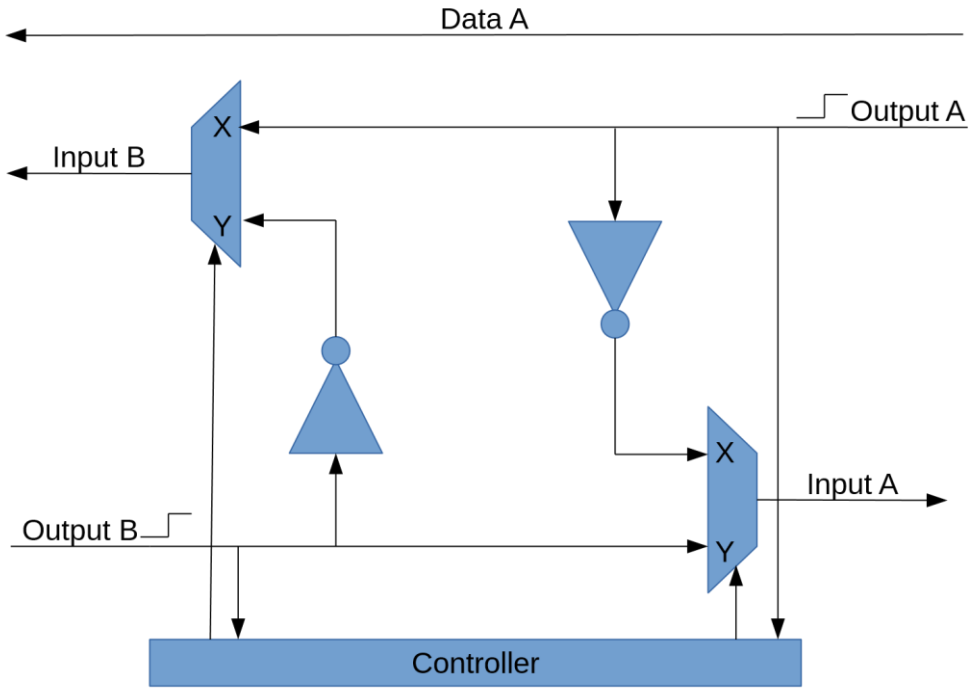


Figure 11: Hardware design backpressure

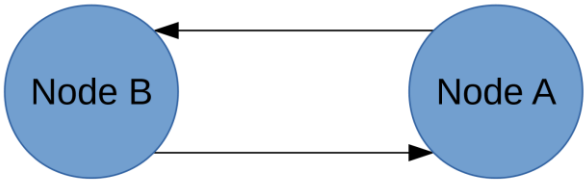


Figure 12: Backpressure between node A and node B

The backpressure only communicate data in one direction as is shown by only one data line. An example shown in Figure 13 for dataflow graph of Figure 14.

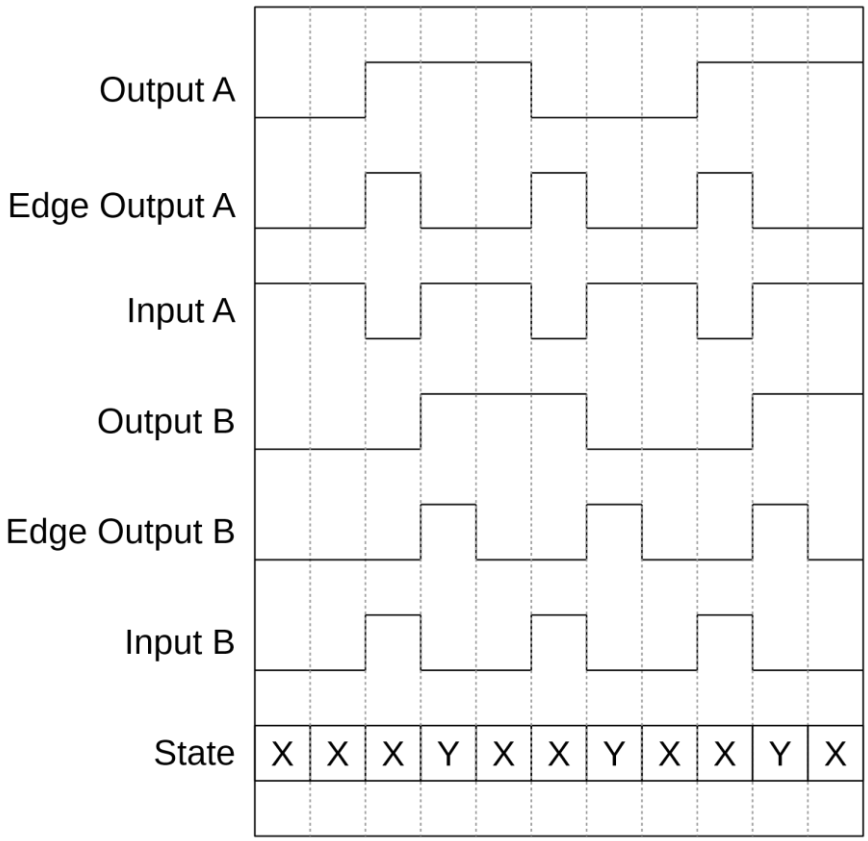


Figure 13: Plot working backpressure

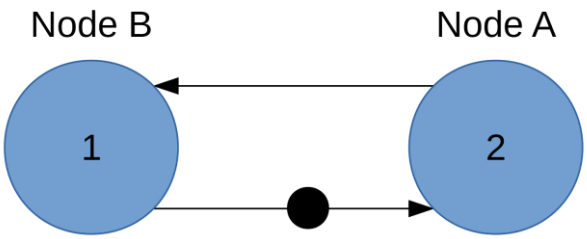


Figure 14: Example dataflow layout

4.1.2 NODES

All nodes will be implemented as state machines. Each node will have at least two states called “Fire” and “Wait”. During the “Fire” state the node produces new tokens,

while the “Wait” state is used for waiting on new input tokens. When in the “Fire” state the next state depends on the available input tokens. When there are tokens available at all input edges (inputs are high) the next state is again the “Fire” state, if not the next state is “Wait”. When the node is in the “Wait” state and tokens are available at the inputs (inputs are high) the node goes to the “Fire” state. Every state execution takes one clock cycle meaning that every node has at least a delay of 1 clock cycle.
The state machine explained here is for a basic node. When the input tokens need to be processed states can be added before the “Fire” state. This means when tokens are available at all inputs the nodes goes through all the processing states finishing with the “Fire” state.

4.2 SYNCHRONIZATION

The framework needs to support synchronization between multiple clock domains. Some information is given with respect to calculating the Mean Time Between Failures for synchronizers and which designs are already available.

4.2.1 MEAN TIME BETWEEN FAILURES FOR SYNCHRONIZER

When communicating between different clock domains there is a change that data is corrupted due to the metastability of flip flops. This happens when the receiver reads the input data just before or just after it has been changed. How much time depends on the setup and hold times of the flip flops used. It cannot be guaranteed that metastability will never happen since the 2 clock domains are asynchronous. The only way to say something about the reliability of this communication is by calculating the chance an error will happen. There are different methods found on the internet, this one is chosen because of it is well documented (Wellheuser, 2018) (Patharkar, 2015) (blendics, 2018).
The chance a flip flop is metastable after a period t_R is given by:

$$P_F = P_E * P_S$$

Where

$$P_E = Chance\ of\ entering\ metastability$$

$$P_S = Chance\ of\ being\ metastable\ after\ t_R$$

P_E is determined by calculating the chance that the edge of the output clock is inside the danger zone of the flip flop (T_0). This is the period where the propagation delay of the flipflop is higher than the clock. The probability is calculated by dividing T_0 by the clock period t_C.

$$P_E = \frac{T_0}{t_c} = f_c T_0$$

$$T_0 = t_{su} - t_h$$

$$f_c = output\ frequency\ (Hz)$$

$$t_c = clock\ period\ (s)$$

$$t_{su} = setup\ time\ (s)$$

$$t_h = hold\ time\ (s)$$

The metastability of a flip flop after t_R seconds is given by:

$$P_S = e^{-\frac{t_R}{\tau}}$$

Where

$$t_R = resolution\ time$$

$$\tau = a\ time\ constant, flip\ flop\ specific$$

The resolution time is the time available for the signal to become stable. This means it is a full clock cycle minus constant delays.

$$t_R = \frac{1}{f_c} - t_{prop} - t_{su}$$

When both probabilities are combined it results in the following formula:

$$P_F = P_E P_S = f_c * (t_{su} - t_h) * e^{-\frac{\frac{1}{f_c} - t_{prop} - t_{su}}{\tau}}$$

To get the failure rate the changing frequency of the input needs to be known. Assume the input changes with a rate f_d, then the failure rate becomes:

$$\lambda = f_d P_F = f_d f_c * (t_{su} - t_h) * e^{-\frac{\frac{1}{f_c} - t_{prop} - t_{su}}{\tau}}$$

The failure rate can be used to calculate the mean time between failures (MTBF) as follows

$$MTBF = \frac{1}{\lambda} = \frac{e^{\frac{\frac{1}{f_c} - t_{prop} - t_{su}}{\tau}}}{f_d f_c * (t_{su} - t_h)} = \frac{e^{\frac{t_R}{\tau}}}{f_d f_c * T_0}$$

The value for the propagation delay t_{prop}, setup time t_{su}, hold time t_h and the flip flop constant τ all depend on the FPGA. These values are not documented, that is why there is decided to base the setup and hold times on the calculation example from ti (Wellheuser, 2018). This example uses values from a FIFO chip which is comparable to what is presented here. Keep in mind that these values are only an indication and could be completely off depending on the FPGA.

$$t_{prop} + t_{su} = 1.3\ ns$$

$$T_0 = t_{su} - t_h = 2.05\ ns$$

$$\tau = 0.4\ ns$$

To improve the MTBF it is possible add an extra flip flops in series, by doing so the performance is improved since the resolution time is doubled.

$$P_F = P_E * P_{S1} * P_{S2}$$

$$MTBF = \frac{e^{\frac{t_R}{\tau}}}{f_d f_c T_0} * e^{\frac{t_R}{\tau}} = \frac{e^{\frac{2t_R}{\tau}}}{f_d f_c T_0}$$

Multiple flipflops can be added to improve the MTBF even more, this is at the cost of communication speed since more cycles are needed to transport one message from one to the other clock domain. The MTBF for “x” number of flip flops is:

$$MTBF == \frac{e^{x\frac{t_R}{\tau}}}{f_d f_c T_0}$$

As both the formulas show the MTBF will increase when the input or output frequency decreases, this can be explained by the fact that the flipflops have more time to become stable.

4.2.2 SYNCHRONIZER SOLUTIONS

There are a number of different designs available for synchronizing data between 2 clock domains. The different layouts mentioned here vary in complexity (Sachin Hatture, 2015) (Tejas, Amit, & Divyanshu, 2018). These designs are mainly used when the two frequencies that need to be synchronized aren’t a multiple of each other. The synchronizer designs are implemented to improve the Mean Time Between Failures (MTBF). Since there is chance data is read by the receiving domain when it is not yet stable.

The first design is simply 2 flipflops in series (Figure 15).

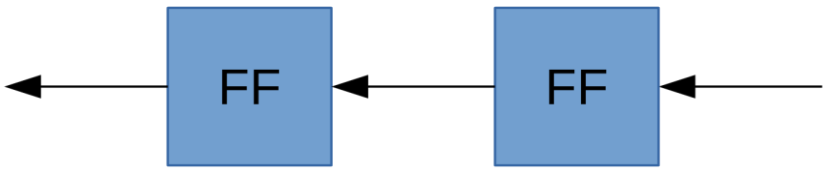


Figure 15: Dual flip synchronizer

This circuit useful since it uses minimal hardware to make multidomain communication possible. The downside of this solution is that there is no knowing

when a signal has arrived in the other domain. The design is also not useful for synchronizing multiple bits since it will decrease the MTBF drastically. Correct operation of this design requires the input signal to be constant during synchronization.

To improve the synchronization of multiple bits the following solution is used (Figure 16).

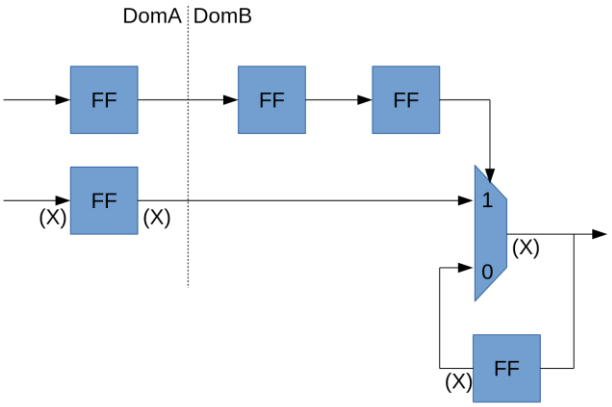


Figure 16: Dual flip flop data synchronizer

This solution is based on the 2 flip flop synchronizer but has a separate communication line for data. The data line has twice as long to become stable compared the synchronization line. To make this solution work the data line should remain constant until the data has been read on the receiver side. Since there is no feedback this is hard to determine. This problem can be circumvented by making the clock frequency of the receiver domain at least twice as high as the clock of the sender domain.

The next design improves the feedback part (Figure 17).

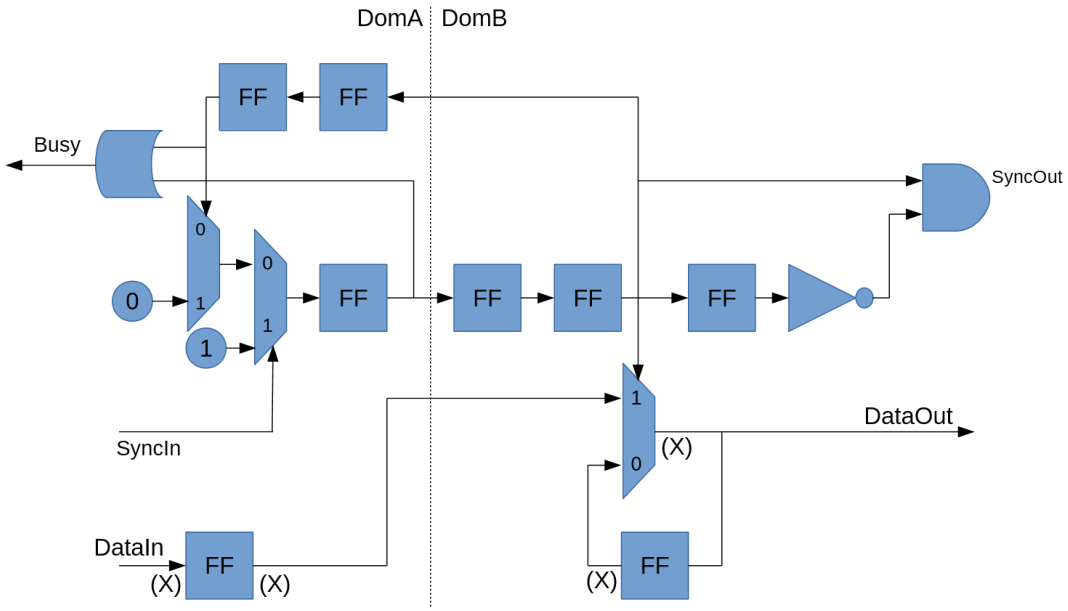


Figure 17: Synchronizer with feedback

This design send a signal to the receiver, this signal is then returned to the sender. During this period a busy signal is kept high at the sender side to indicate no new signal can be send. This solution solves the feedback problem at the cost of more hardware. It does not allow to send data since it only has one communication line.

Two other designs found improve on the data communication aspect. The first solution adds data lines to a sort of handshake design, the last solution uses a FIFO buffer as communication. A FIFO is the most reliable way of communication but does require a lot of hardware. The best trade-off is the handshake protocol with data communication abilities.

4.3 FRAMEWORK CONTENTS

The framework is going to consist of a number of dataflow nodes that have a hardware equivalent. Multiple nodes have been designed to create a first basis for the framework. Each node is designed in a number of steps:

1. Design a dataflow node of the function that is going to be added

2. Convert the designed node to hardware design (explanation shown in appendix A.4)

3. Test hardware design and convert back it back to dataflow to prove they are equal. (explanation shown in appendix A.3)

4.3.1 COMMUNICATION NODES

As explained before, these nodes are responsible for sending the data through the system. They will be used in combination with a controller. A number of different nodes are designed as is shown here:

4.3.1.1 INPUT NODE

Nodes/Basic/NodeInput.hs

The input node reads data from its input and outputs it to the controller (Figure 18).

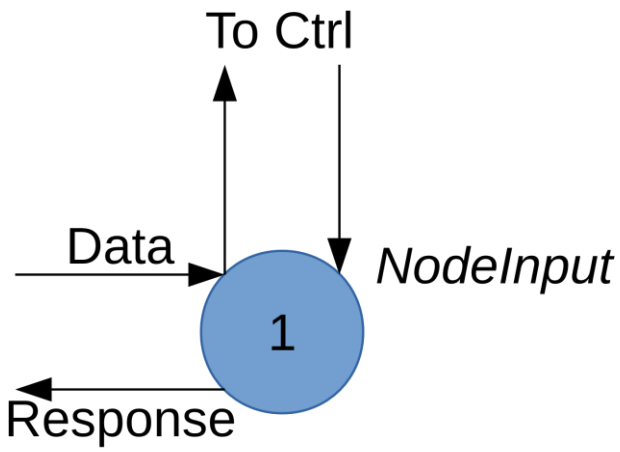


Figure 18: NodeInput

4.3.1.2 OUTPUT NODE

Nodes/Basic/NodeOutput.hs

The output node read data from the controller and outputs it to the rest of the system (Figure 19).

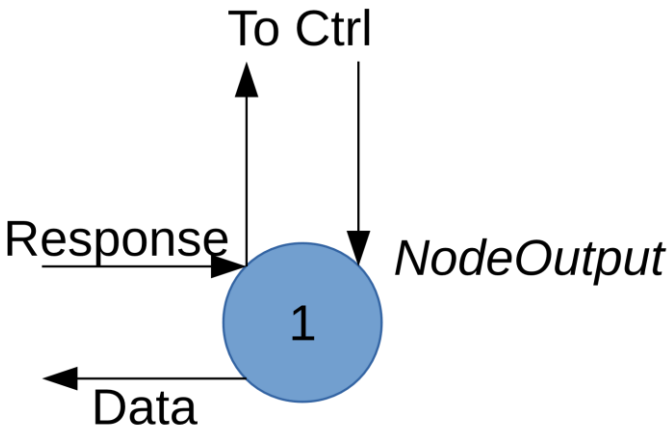


Figure 19: NodeOutput

4.3.1.3 NORMAL NODE

Nodes/Basic/NodeNormal.hs

The normal node read input data and sends it to the connected controller, at the same time it reads the previous controller value and sends it to the output (Figure 20).

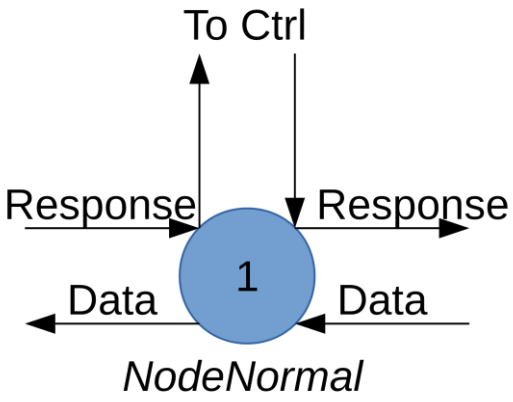


Figure 20: NodeNormal

4.3.1.4 BLOCKING NODE

Nodes/Basic/NodeBlock.hs

The blocking node blocks its input after an output, it is enabled again when the controller returns a signal (Figure 21). The controller should be of the type delay.

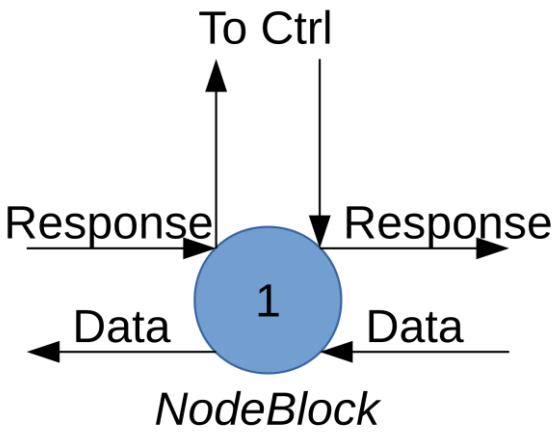


Figure 21: NodeBlock

4.3.2 CONTROLLERS

Controller manipulate the input data and then output them again. A controller should be used in conjunction with a communication node. A number of pre-made controllers is shown here. A structured way of creating custom controllers is explained in appendix A.4.

4.3.2.1 SPI

Nodes/Controllers/SPI/Stream.hs
Nodes/Controllers/SPI/StreamR.hs

The SPI controllers are used to communicate with a SPI interface. There are two design:

Stream: Does not pull CS high after each message
StreamR: Pulls CS high after every message

4.3.2.1.1 DATAFLOW DESIGN

The dataflow model can be found in Figure 22.

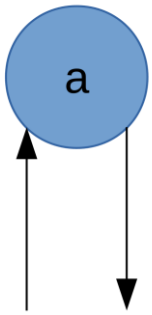


Figure 22: Controller SPI

The initial dataflow design has an unknown execution time a, the value for a is determined by the implementation in Clash. The Clash implementation has a delay equal to x+1 where x represents the message size.

4.3.2.1.2 HARDWARE DESIGN

To understand where the x+1 delay for the node is coming from the underlying hardware needs to be known. The controller reads data from its input and puts it in an internal buffer, the first bit is directly written to the sensor since it is unnecessary to store. The next x (depending on message size) clock cycles data is read from the sensor and put in the buffer while data from the buffer is written to the sensor. When the last bit is written the full buffer is outputted together with the last bit from the sensor making the output complete (Figure 23).

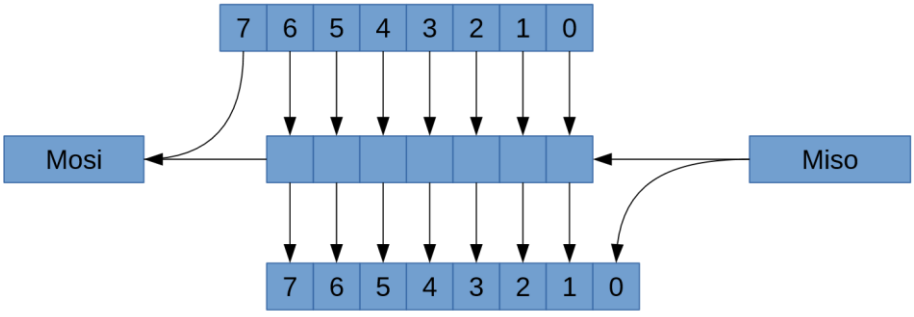


Figure 23: Working SPI controller

When no data is available at the input of the node the controller is halted and the sensor clock is turned off until new data is available.

4.3.2.2 NODE DELAY

Nodes/Delay/Delay.hs

The delay node is a controller that adds a delay between its input and output. It can be used in combination with the nodeBlock to block the input for a number of clock cycles. The dataflow looks like Figure 24.

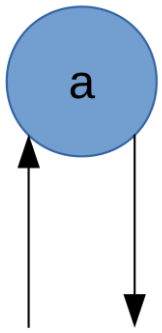


Figure 24: Delay

4.3.3 OTHERS

This node could be categorized as a controller or a communication node since it doesn't store tokens. It converts an incoming token outputs it directly hence the 0 delay.

4.3.3.1 SLICE TOKEN

Designs/SliceToken.hs

The slice token node can be used to make the length of a token in number of bits smaller, this can be useful for getting rid of useless data. The node can be implemented with zero delay since cutting of bits in hardware is done by not connecting wires.

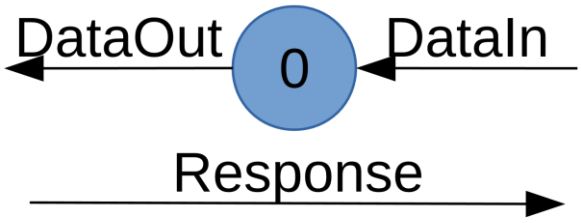


Figure 25: Dataflow SliceToken

The node has no state so proving its correct behavior is not necessary.

4.4 FRAMEWORK IMPLEMENTATIONS FOR SENSOR INTERFACE

These are implementations made with framework nodes explained above. These designs are made to create a sensor interface. In some cases a custom parts are added, the choice to include these parts will be explained when this is the case.

4.4.1 COMMUNICATION INTERFACE

4.4.1.1 SPI INTERFACE

Designs/SPIinterface.hs

This design is made to make working with SPI easier, the design needs as inputs an SPI controller type (Stream, StreamR) and the type of communication node. The design then automatically generates the correct layout of nodes.

4.4.2 SENSOR DRIVER

No specific implementations have been designed as sensor driver.

4.4.3 SYNCHRONIZATION

There are cases where a design needs multiple clock domains, for example when communicating with a sensor. Sensors do often run at lower clock speed then the logic using the sensor data. There is the option to use oversampling for this problem to get rid of the second clock but this needs hardware to convert the sample data. The higher clock speed used for oversampling will also limit the flexibility in placing the hardware on the FPGA, for lower clock speeds wires can be longer giving the FPGA more freedom in placing the different hardware components. To communicate between two clock domains synchronization is needed. Two designs are shown, one in case the two frequencies are multiples and one in the case they are not. Keep in mind that the execution times are worst case since they depend on the distance between the edges of the two clocks. This means that the output period of a synchronizer is variable and not deterministic. The IO sampler that will be explained later can be used to compensate the variability.

4.4.3.1 SYNCHRONISATION: FREQUENCIES OF BOTH DOMAINS IS MULTIPLE

Designs/SyncMult.hs

In case the two clocks are a multiple of each other the edges of the clocks are in sync once every x clock cycles (depending on frequencies). The ratio between the two frequencies is given by the frequency ratio or the period ratio:

$$R_f = \frac{f_{in}}{f_{out}} = \frac{\frac{1}{d_{in}}}{\frac{1}{d_{out}}} = \frac{d_{out}}{d_{in}}$$

$$d_{out} = period\ of\ clock\ out\ (s)$$

$$d_{in} = period\ of\ clock\ in\ (s)$$

$$R_f = frequency\ ratio$$

There are two problem that need to be overcome. First is the synchronizing from the fast to the slow clock domain, the sending node needs to halt until the receiving domain has processed the data. The second problem arises when synchronizing from a slow to a fast clock domain, the node in the fast clock domain can read the same input signal multiple times since the output node updates its output signal too slow.
The synchronization from the fast to the slow domain has already been covered by the backpressure since it halts the sending node. The other way around is a bit more difficult, it is solved as follows. The synchronization uses a custom node which toggles its output instead setting it high or low, this in combination with the backpressure design explained earlier prevents the system from reading the same input multiple times. The layout is shown in Figure 26. No synchronization flip flops are needed since there is no metastability.

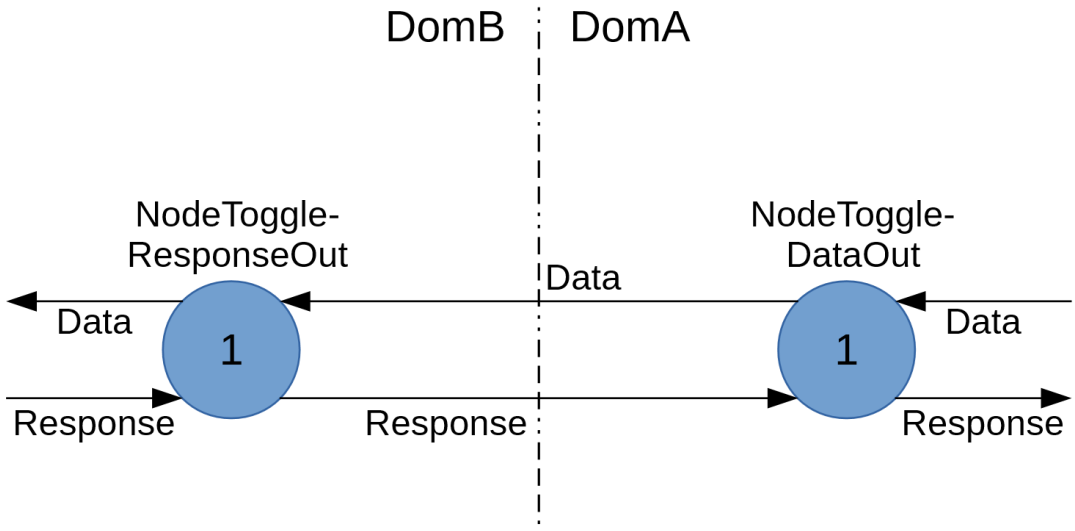


Figure 26: Synchronization multiple

The backpressure design should run at the highest frequency of the two synchronizer frequencies to assure correct edge detection.

The worst case throughput of the synchronizer is determined by maximum period of the internal loop.

$$Throughput_{syncMult} = d_{in} + d_{out}$$

$$d_{out} = period\ of\ clock\ out\ (s)$$

$$d_{in} = period\ of\ clock\ in\ (s)$$

The maximum delay from input to output is:

$$Delay_{syncMult} = d_{in} + d_{out}$$

$$d_{out} = period\ of\ clock\ out\ (s)$$

$$d_{in} = period\ of\ clock\ in\ (s)$$

4.4.3.2 SYNCHRONISATION: FREQUENCIES OF BOTH DOMAINS IS NO MULTIPLE

Designs/SyncNoMult.hs

When the clocks are not a multiple of each other synchronization becomes more difficult, it can never be guaranteed that the clock edges are in sync.

The already existing designs explained earlier are hard to combine with the dataflow format used. There has been decided to use the same design as in Figure 26. The communication between the two domains is extended with a dual flip synchronizer (Figure 27). The flip flops are indicated by the circles with c and d in them. The value for c and d represent the number of flip flops in the synchronizer.

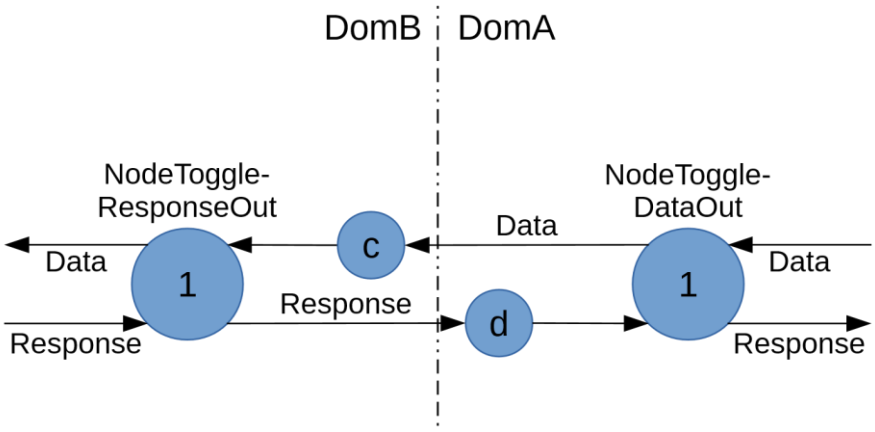


Figure 27: Synchronizer frequencies aren't multiples

For the synchronization in both directions the normal flip flop design is used (Figure 15). The data itself is synchronized separately based on (Figure 16), the only difference is that the multiplexer is included in the receiver node. One requirement for synchronization is that the input signals will be kept constant during synchronization giving the flip flops time to become stable, the toggle nodes are perfect suited for this since they keep their output constant for at least the time it takes for the feedback to

respond which is always larger than time it needs to become stable. The backpressure is a custom designed since it had to be split up to make room for the synchronizer flip flops Figure 28.

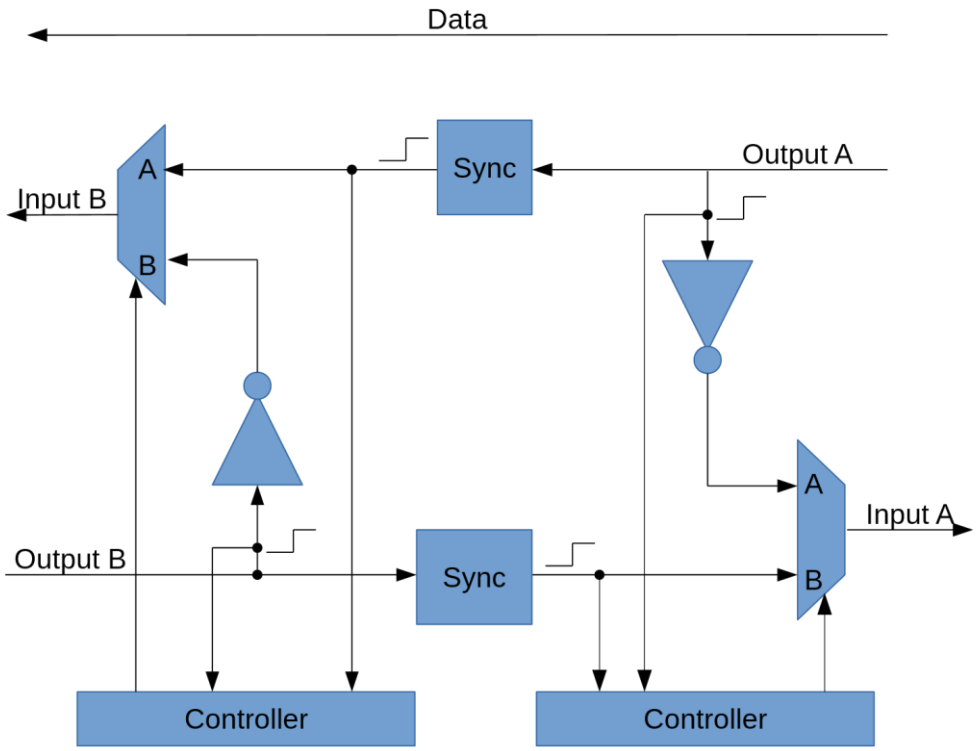


Figure 28: Back pressure design for synchronization

The minimum throughput of this synchronizer can be calculated by determining the maximum period of the design:

$$ThroughputMin_{syncNoMult} = (d + 1) * d_{in} + (c + 1) * d_{out}$$

The maximum delay of the design is

$$Delay_{syncNoMult} = d_{in} + (c + 1) * d_{out}$$

The MTBF is calculated for this design and can be found in appendix A.2.

4.4.3.3 SMART SYNC

Designs/SmartSync.hs

The smart sync design automatically determines which synchronization solution is best based on the given input frequencies. It can choose between no synchronization, when fin and fout are the same, synchronization for multiples and synchronization for no multiples.

4.4.4 IO SAMPLER

4.4.4.1 FLOW GATE

Designs/FlowGate.hs

This design is based on the NodeBlock communication node, it can be combined with different delay controllers and by doing so influence its behavior. At this moment there is only one option which is called “Delay”, this option adds a Delay controller to the communication node so the input blocked for a number of clock cycles after each input. The design compensates for the extra delay added by the communication node. (Figure 29).

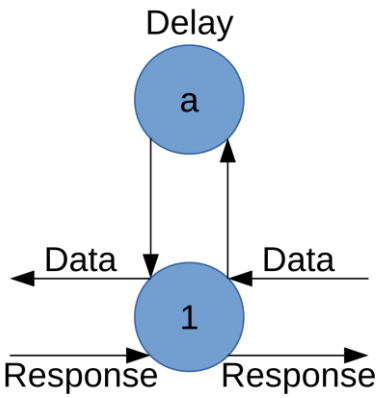


Figure 29: FlowGate

5 RESULTS

The framework will be tested in two steps:

- ➔ First all components of the framework will be tested separately. This includes both the nodes and the designs made with these nodes, like the synchronizer.
- ➔ Next the components of the framework are used to build a interface for a sensor. This is done to show that the framework can be used to build a deterministic interface.

5.1 TEST PROCESS

5.1.1 TEST BOARD

The FPGA board that is going to be used for testing is the Zybo-Z20 from Digilent. This board is equipped with a Zynq-7010 FPGA from Xilinx. All Digilent boards make use of a standard interface called PMOD, this interface is mainly used for connecting peripherals like sensors and actuators. The PMOD interface is called a standard but still uses 3 different pin layouts (1 x 6, 2 x 6 and 2 x4). Digilent provides a collection of devices that can be connected to the PMOD interface, these vary from displays to light sensors. Most of the devices are provided with an example code, sometimes this is only in C, other times also VHDL or Verilog. The official PMOD interface supports four protocols: SPI, GPIO, I^2C and UART. The four communication protocols supported by PMOD are very common.

5.1.2 TEST PROCESS INDIVIDUAL COMPONENTS

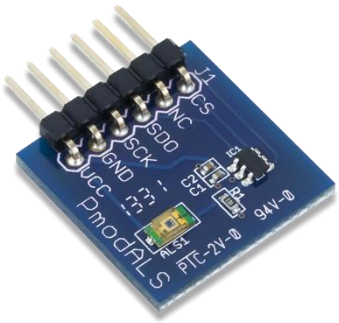
The individual components of the framework will be tested separately to verify that they are working correctly. These tests will be done by comparing the output of each of the individual components to a unique test vector. These test vectors are designed in such a way that most input to output combinations of a component are covered. The first tests will be done in the simulation environment of Clash. When the simulations in Clash are successful the code will be converted to Verilog. The FPGA programming software Vivado is then used to simulate the Verilog code, the results are compared to the results from Clash. When this works the implementation for each component will be generated to get an idea of the hardware footprints, these footprints are given in LUTs and flipflops.

5.1.3 TEST PROCESS IMPLEMENTATION

To test if the framework is useful for building a sensor interface a test setup is made using the layout shown in Figure 6. The sensor that is going to be used for testing is a SPI ambient light sensor. The setup will be tested with an infinite supply of tokens at its input which is the side that will be connected to the bus. This is done to get a periodic output, asking constantly for new data is the most stress that can be put on the system. The designs will first be tested in Clash by using the internal simulation

tool. If the simulations are successful the code is converted to VHDL and tested in Vivado. The output period of these simulations is determined and compared to their theoretical value. Next the design in implemented on the FPGA, it will be added as an AXI peripheral so the output can be read out by a CPU and displayed. The output signal indicating new data is connected to an output pin so the output period can be compared to the theory. The last step is checking the size of the design in number of flip flops and LUTs

5.1.3.1 SENSOR



The sensor used is a Digilent PMOD light-sensor. This is a sensor with an SPI interface that sends out 2 bytes of data every time the chip select goes from high to low. The 2 bytes consist of 3 zeros, then 8 bits of data followed by 4 zeros (Figure 30). The 15 bits are collected by the driver, the driver cuts off the zeros and then puts the data on the output. The sensor has an operating frequency between 1 MHz and 4 MHz. There is chosen to use an SPI sensor instead of an I2C, UART or GPIO because the other interfaces had implementation problems. The implementation of I2C needs tri-state inputs and outputs for communication which cannot be implemented in Clash. The UART interface needs a clock which must be synchronised which is not possible is Clash unfortunately. A solution would be to do oversampling, if SPI was not easier to implement this interface would have been chosen. The last interface that could be used is GPIO, this is not used since it is too specific for each sensor which is not in line with a framework design.

0	0	0	x	x	x	x	x	x	x	x	0	0	0	0

Figure 30: Data layout light sensor

5.1.3.2 LAYOUT

The layout of the test setup will differ depending on the input and output frequencies. The Clash design automatically changes the layout depending on these frequencies. To guarantee the design fully works the three possible layouts will be tested separately. These layouts are:

- ➔ Frequency of sensor and receiver are equal
- ➔ Frequency of sensor is multiple of the receiver frequency
- ➔ Frequency of sensor and receiver are no multiple.

5.1.3.2.1 DESIGN 1: FREQUENCIES ARE EQUAL

When the input and output frequency are equal, the synchronization step is not necessary, the dataflow is given in Figure 31.

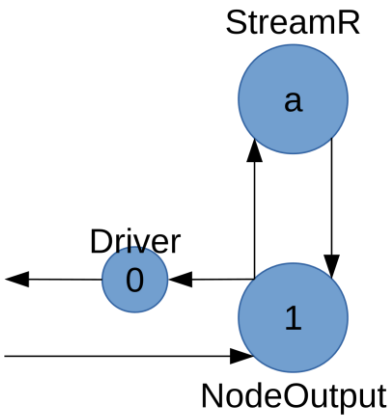


Figure 31: Dataflow Lightsensor without synchronization

The IO sampler is left out since there is no synchronization so no variable delay. The SPI interface uses the streamR design, this design pulls CS high after every message. This is necessary to have the sensor work correctly. The sensor itself only outputs data, this is why the communication node below the streamR SPI interface uses the NodeOutput design. The delay for the streamR controller equals the message size plus 1 which results in a delay of 16 cycles.
The output period can be determined by calculating the MCR, there are only three loops in this design to it is relatively easy to do.

$$MCR = \max(16 + 1, 16, 1) = 17\ cycles$$

5.1.3.2.2 DESIGN 2: FREQUENCIES ARE MULTIPLES

The next test uses two frequencies that are multiples. The sensor frequency is set to 2 Mhz, the frequency of the user IP is set to 100 Mhz. The design that results from these frequencies is shown in Figure 32.

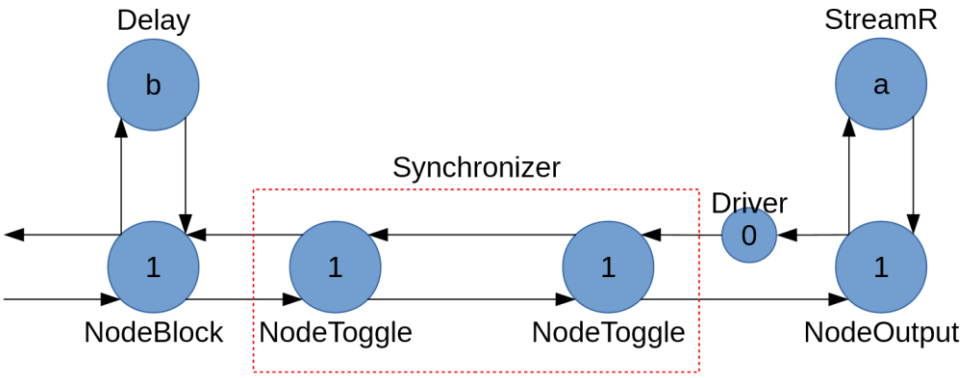


Figure 32: Dataflow light sensor when frequencies are multiples

The design consists of three components: The SPI interface, the synchronization and the IO sampler. The delay for the streamR controller is the same as for equal frequencies so 16 cycles.
The synchronization uses the design for frequencies that are a multiple of each other.
The last component is the IO sampler. The IO sampler is responsible for correcting the variable delay of the synchronizer so the output of the dataflow is deterministic. The period of the IO sampler should be equal or higher than the MCR of the rest of the sensor interface.
The delays for the nodes are given in number of clock cycles for their specific domain. The delays are corrected by dividing them by the frequency ratio given below so all delays are given in number of output clock cycles.

$$Rf = \frac{f_{sensor}}{f_{userIP}} = \frac{2\ MHz}{100\ MHz} = \frac{1}{50}$$

The corrected dataflow graph is shown in Figure 33. The different loops in the dataflow graph are marked.

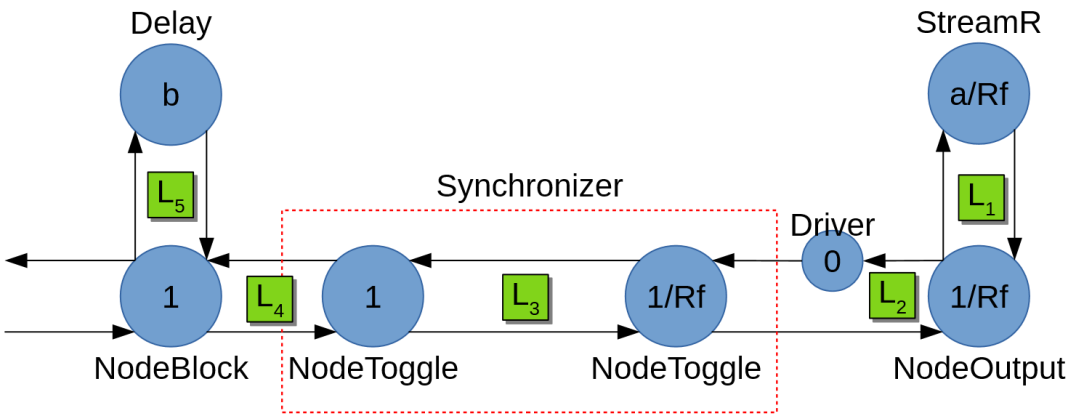


Figure 33: Dataflow light sensor when frequencies are multiples, corrected

The value for b is still unknown. It can be calculated by using the property that the output period (L_5) should equal the MCR of the rest of the dataflow:

$$MCR_{other} = L_5 = b + 1 = max(l_1, l_2, l_3, l_4)$$

The periods of the designs are:

$$l_1 = \frac{16 + 1}{Rf} = 850$$

$$l_2 = \frac{1 + 1}{Rf} = 100$$

$$l_3 = 1 + \frac{1}{Rf} = 51$$

$$l_4 = 1 + 1 = 2$$

When these values are filled in it results in a value for b:

$$b = max(850,100,51,2) - 1 = 849$$

As can be seen from the results the SPI interface is the slowest part of the design. The design will generate an output with a period of 850 clock cycles.

5.1.3.2.3 DESIGN 3: FREQUENCIES ARE NO MULTIPLE

The last case is if the frequencies used are no multiple. The frequency for the userIP is set to 101 MHz, the sensor still used 2 MHz. The new design will look like Figure 34.

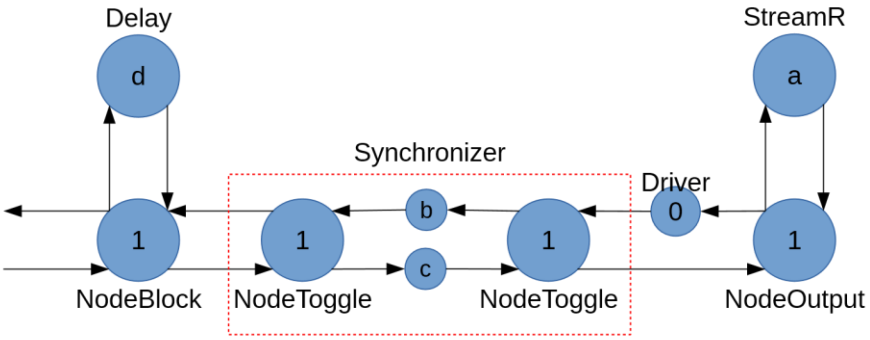


Figure 34: Dataflow Light sensor, frequencies are no multiples

As can be seen, in this case the synchronizer for frequencies that aren't a multiple is used. The variables b and c represent the number of flip flops between the two nodes, both b and c are set to two since there is no option in the code yet to set number of flip flops. There is started with the calculation of a.
The value for "a" equals the 16 clock cycles from before since the message size did not change.
The clock cycles delay in the sensor domain are corrected in the same way as was done for the previous solution. The frequency ratio is used to convert the delays in the sensor domain to the output domain.

$$Rf = \frac{f_{sensor}}{f_{userIP}} = \frac{2\ MHz}{101\ MHz} = \frac{2}{101}$$

The corrected dataflow graph is shown in Figure 35. The different loops in the dataflow graph are again marked.

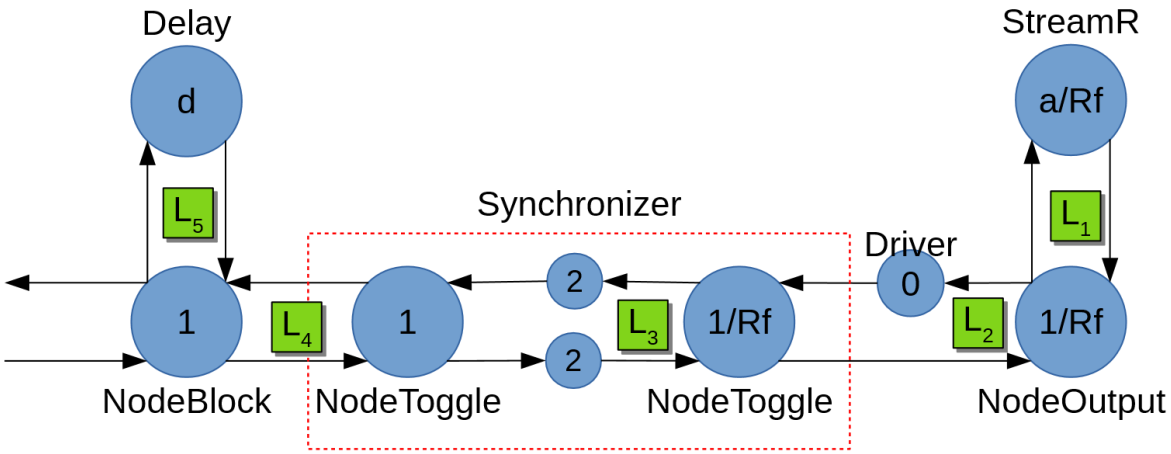


Figure 35: Dataflow light sensor when frequencies aren't multiples, corrected

The value for d is still unknown but can be calculated by using the property that the output period (L_5) should equal the MCR of the rest of the dataflow:

$$MCR_{other} = L_5 = d + 1 = max(l_1, l_2, l_3, l_4)$$

The periods of the designs are:

$$l_1 = \frac{16 + 1}{Rf} = 858.5$$

$$l_2 = \frac{1 + 1}{Rf} = 101$$

$$l_3 = 1 + 2 + \frac{2 + 1}{Rf} = 154.5$$

$$l_4 = 1 + 1 = 2$$

When these values are filled in it results in a value for d:

$$d = max(858.5, 101, 154.5, 2) - 1 = 857.5$$

The ceil value for d is used since half clock cycles do not exist.

$$ceil(d) = 858$$

The slowest loop is again the SPI interface. The output period equals in this case 859 clock cycles.

The MTBF is calculated to check if the synchronizer is reliable enough. A good target value for MTBF could not be found so it is set to 1000 years. The calculation for this setup resulted in a MTBF of:

$$\text{MTBF} = 5357753{,}12\ years$$

This is much higher than the required 1000 years, the full calculation can be found in appendix A.5.

5.1.3.3 TEST ON FPGA

The last step is to test the design on the FPGA, this will be done by connecting it to the AXI bus and communicate over UART. The bit indicating new data can be analyzed with the build in logic analyzer of Vivado, the period of this signal should equal (depending on design).

$$Frequencies\ are\ multiple: 850\ samples$$

$$Frequencies\ are\ not\ multiple: 859\ samples$$

5.2 TESTING FRAMEWORK

5.2.1 TESTING INDIVIDUAL COMPONENTS

All the components inside the framework are tested in Clash as well as in Vivado, the results are given in Table 1.

Table 1: Results testing components

NODE NAME	SETTINGS	CLASH SIM	VIVADO SIM	SIZE
BACKPRESSURE	Constant	Success	Success	1 LUT 3 REGISTER
NODEINPUT	Input: 8 Bit Output: 8 Bit	Success	Success	2 LUT 10 REGISTERS
NODEOUTPUT	Input: 8 Bit Output: 8 Bit	Success	Success	2 LUT 11 REGISTERS
NODENORMAL	Input: 8 Bit Output: 9 Bit	Success	Success	2 LUT 20 REGISTERS
NODEBLOCK	Input: 8 Bit Output: 8 Bit	Success	Success	2 LUT 11 REGISTERS
DELAY	Delay: 3	Success	Success	2 LUT 5 REGISTERS
SYNCMULT	Input: 8 Bit Output: 8 Bit	Success	Success	7 LUT 23 REGISTERS
SYNCNOMULT	Input: 8 Bit Output: 8 Bit	Success	Success	8 LUT 30 REGISTERS
FLOWGATE	Input: 8 Bit Output: 8 Bit Delay: 4	Success	Success	5 LUT 18 REGISTERS
STREAM	Input: 8 Bit Output: 8 Bit	Success	Success	14 LUT 25 REGISTERS
STREAMR	Input: 8 Bit Output: 8 Bit	Success	Success	14 LUT 24 REGISTERS

5.2.2 TESTING IMPLEMENTATION

The three different layouts for the light sensor interface are tested separately.

5.2.2.1 FREQUENCIES ARE EQUAL

5.2.2.1.1 SIMULATION

The simulation in clash for the interface with equal frequencies is ran successfully. The clash-language is converted to VHDL and tested in Vivado. The time stamps where new data arrives are 80,003 ms, 165,003 ms and 250,003 ms. The time between these results is 85 ms. The output frequency is 2 MHz meaning a period of 500 ns, this means the number of cycles between outputs is:

$$\frac{85000}{0{,}5} = 170000\ cycles$$

The period should be 17 cycles which means the value from the simulation is 10000 times to high, when there is looked at the results in more detail it can be seen that a clock period takes 5 ms (smallest period without change), this is 10000 times too high. The difference is probably caused by the simulation clock which means the error is caused by the Clash compiler.

5.2.2.1.2 FPGA

The design is implemented on the FPGA, there is tested if the sensor is read out correctly. The output values cover the full range of the sensor so it can be assumed sensor is read out correctly.
Next the output period is tested to see if the output is periodic, the results are shown in Figure 36. Every pulse indicates new data, for periodic behavior the distance between the pulses should be equal.

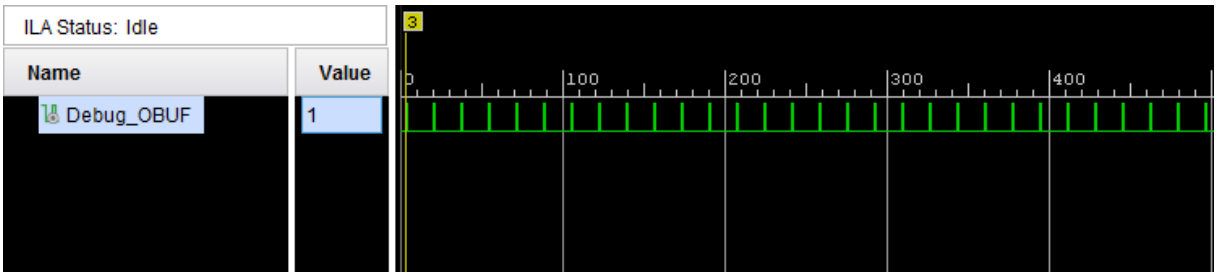


Figure 36: Output signal indicating new data for the setup where the frequencies for the sensor and userIP are equal. Each pulse indicates a new output, the period between the pulses is constant meaning the output is periodic.

The pulses are at sample times:

$$3, 20, 37, 54, 71, 88, 105$$

The times between the pulses equals the 17 determined in the method.

The size of the design when implemented on a FPGA is:

45 REGISTERS, 20 LUT

5.2.2.2 FREQUENCIES ARE MULTIPLE

5.2.2.2.1 SIMULATION

The next design that is going to be tested is the one where frequencies are multiples. The IO sampler added is filled in by hand since it is not yet possible to let Clash do this. The period is determined by running the function for calculating the MCR before compilation, the result in then filled in. The function returns a MCR of 850 cycles which equals the value calculated earlier.
The clash simulation is ran successfully so the code is converted to VHDL. The simulation in Vivado shows outputs generated at: 85.203 us, 170.203 us, 255.203 us. The difference between these outputs is exactly 85.000 us. The output frequency equals 100 MHz which results in a period of 10 ns. With these values the output period can be calculated:

$$\frac{85000000}{10} = 8500000\ cycles$$

The calculated output frequency is again a factor 10.000 too high. The graphs show that the clock period during simulation equals 100 us, this is about 10.000 times higher than the period calculated which explains the difference.

5.2.2.2.2 TEST ON FPGA

The simulations work correctly so the design can be implemented on a FPGA. The sensor is read out correctly meaning the sensor interface is working. Next the output is tested so see if it is periodic, the results are shown in Figure 37. Every pulse indicates new data, the distance between the pulses should be equal for periodic behavior.

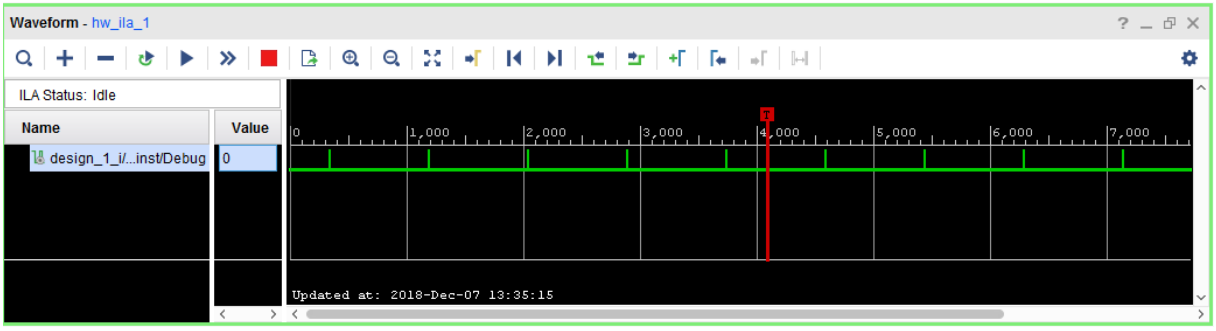


Figure 37: Output signal indicating new data for the setup where the frequencies for the sensor and userIP are a multiple of each other. Each pulse indicates a new output, the period between the pulses is constant meaning the output is periodic.

The pulses are at sample times:

$$339{,}1189, 2039, 2889, 3739, 4589, 5439, 6289, 7139, 7989$$

The time between the samples are 850 samples meaning the output period equals the 850 cycles determined in the method.

The size of the final design is

100 REGISTERS, 40 LUT

5.2.2.3 FREQUENCIES ARE NO MULTIPLE

5.2.2.3.1 SIMULATION

The design where the 2 frequencies are not a multiple of each other is the most complex version. Again the output period is filled in by hand but should in this case equal 859 cycles. After the Clash simulation is ran successfully the code is converted to VHDL. The periodic is checked in Vivado, outputs are generated around: 85.349,62 ns, 170.399,21 ns, 255.448,8 ns. The difference between these times is 85.049,59 ns which means the output is periodic. The output frequency is 101 MHz meaning it has a period of 9,901 ns, this means the number of cycles between outputs is:

$$\frac{85.049.000{,}59}{9{,}901} = 8590000\ cycles$$

This is again a factor 10.000 too high, the reason is most likely the same as for the previous result.

5.2.2.3.2 TEST ON FPGA

With the simulation being successful the design can be implemented on the FPGA. The sensor data seems to be correct so there can be continued with testing the output interval. The output interval is tested in the same way as for the other designs, the results are shown in Figure 38.

Figure 38: Output signal indicating new data for the setup where the frequencies for the sensor and userIP are not a multiple of each other. Each pulse indicates a new output, the period between the pulses is constant meaning the output is periodic.

The pulses are at sample times:

$$830, 1689, 2548, 3407, 4266, 5125, 5984, 6843, 7702$$

The times between the pulses equals the 859 determined in the method.

The size of the full layout is:

107 REGISTERS, 44 LUT

6 CONCLUSION

Designing a framework that has to deliver deterministic and real time designs is not an easy task. The first step taken in the design process was to use dataflow for analysis. Dataflow is useful because it is deterministic by definition and has a lot of analysis techniques already available. For the framework itself there where two options on how to design it. The first option was to design hardware blocks which could be combined to create a sensor interface. These designs would all have a dataflow equivalent to prove their deterministic behavior. The reason for choosing a hardware approach was the flexibility and ease of design. It turned out that this solution was much more complex than it initially seemed. There were problems with timing, dataflow conversion and doing analysis as a whole. This resulted in trying another approach. The second approach starts with dataflow. The framework is designed in dataflow and later converted to hardware. The framework itself consists of a set of dataflow nodes which can be combined by the user to create sensor interfaces. Each node has a hardware equivalent which is used to generate the design for a FPGA. The nodes are connected with backpressure to limit the number of tokens that can be stored on an edge and to make the whole system self-timed. This solution is much better analyzable at the cost of some flexibility. The designs made with the framework are automatically self-timed due to the backpressure, this makes designing simple since the timing problems are handled automatically.
Each node in the framework has been tested to check its functionality and to assure it behaves according to the dataflow rules. To test the deterministic behavior of the framework an interface for a SPI light sensor created. Its output signal is tested to see if it is deterministic and if it outputs realistic sensor readings.
The test done gave positive results, the output period was deterministic and the sensor was read out correctly. The amount of area used by the components is low, the test setup only used 118 FF and 58 LUTs. The small hardware footprint makes it realistic for implementation on a FPGA which has multiple thousands flip flops and LUTs available.

To summarize, building a framework based on dataflow is a good solution. The framework is deterministic and can be analyzed by using dataflow techniques. In terms of flexibility it is a bit limited due to the limitations that come with dataflow. The use of backpressure makes the design self-timed, this gets rid of all kind of timing problems. By expanding the framework in the future the functionality can be improved.

7 DISCUSSION AND FUTURE WORK

There is a difference between the simulation done in Vivado and what is specified in Clash. The clock in Vivado is 10.000 times slower than the one implemented in Clash. This will not cause any problems since the relative speed difference between the clocks stays the same. However, it something that should be kept in mind when doing analysis on the simulation results.

The framework is at this moment limited in its functionality since there is only a small number of nodes available. The number of nodes should be increased in the future to add functionality to the framework.
The way nodes are connected at this moment is not as easy as it should be, all edges have to be connected by hand. In the future this should be simplified, an example of what the syntax could look like is:

$$output = node1 \ <--> \ node2 <--> \ (node3 \ input)$$

The synchronization step for synchronizing when frequencies are not a multiple of each other could result into problems since it hasn't been fully verified. If an edge is missed due to metastability the synchronizer is locked meaning the whole system could get stuck. This is something that should be tested in the future.

8 APPENDIX

A.1 COMMUNICATION PROTOCOLS

The PMOD interface uses four types of protocols. Each of these protocols is explained here.

A.1.1 SPI

SPI is a three+ wire communication protocol consisting of one master and 1+ slaves. Each slave adds an extra wire for communication in the form of a chip select. When the chip select is pulled low the slave is selected and the communication can start. The communication part of the interface consists of a clock, a mosi (master out, slave in) and a miso (master in, slave out). The word size and the operating frequency are not defined. The send and received message do not even have to be the same size. Most SPI slaves read on the falling edge to prevent problems with instable data.

A.1.2 GPIO

GPIO stands for general purpose IO, it does not have a standard since it fully custom. This means nothing can be said about the communication protocol.

A.1.3 I2C

The I2C communication interface only uses 2 wires, one clock and a data line. The protocol supports multiple masters and slaves. The maximum clock frequency is 100 kHz for normal speed and 400 kHz for full speed. In the default state both the clock and data line are kept high. To start communication first the data line is pulled low and then the clock. Next the clock is started and the address is send, this address can be 7 or 10 bits with an extra bit for reading or writing. In case of 7 bits one byte is send consisting of the address and the read/write bit. If the 10 bit address is used it will be send using two bytes. The byte starts with the code 11110 followed by 2 of the address bits and then the read/write bit. The next byte consists of the remaining 8 bits from the address. After each byte the receiver (slave in this case) will pull the data line low to indicate the message was received. All the message send with I2C have to be 8 bit.

A.1.4 UART

UART uses 2 wires for communication the same as for I2C. The purpose of these wires is different however, one is meant for sending and the other for receiving. The 2 devices connected share a clock speed which is defined beforehand. A message is send by pulling the tx line from high to low, next the 8 bit long message is send followed by a parity bit (optional). The parity bit is added for error checking, it counts the number of ones in the message and indicates if it is even or odd. The last 1 or 2 bits send are stop bits, these indicate the end of a message.

A.2 MTBF FOR SYNCHRONIZER IN CASE FREQUENCIES AREN'T MULTIPLES

The MTBF formula for synchronizers that have to synchronize between two frequencies that aren't a multiple is determined here. MTBF calculations are only necessary for these types of synchronizers, for the other versions the metastability is covert by the compiler. The number of flip flop used for synchronization is represented by the variables c and d which are shown in Figure 27, the number of flip flops determines the MTBF. The relation between c and d and MTBF is calculated. There are two MTBFs that need to be calculated.

- ➔ Synchronization of response bit ($MTBF_{Response}$)
- ➔ Synchronisation of message ($MTBF_{Message}$)

When synchronizing data first the message is send covert by the $MTBF_{Message}$, next the receiver domain has to send back a response with a MTBF of $MTBF_{Response}$. The MTBFs can be seen as if in series so they can be added as follows:

$$MTBF_{total} = \frac{1}{\frac{1}{MTBF_{Response}} + \frac{1}{MTBF_{Message}}}$$

There is started with the $MTBF_{Response}$, this can be calculated with the formula determined earlier:

$$MTBF_{response} = \frac{e^{d*\frac{\frac{1}{f_e} - 1.3\ ns}{0.4\ ns}}}{f_d f_e * 2.05\ ns}$$

$$f_d = frequency\ of\ input\ data$$

$$f_e = input\ frequency$$

$$d = number\ of\ flip\ flops\ in\ synchronizer$$

The $MTBF_{Message}$, is going to be a bit more difficult since the synchronization works differently compared to the response bit. The message that has to be synchronized has 2 clock cycles to become stable, but it needs to be guaranteed that all bits are stable to prevent data incoherency. Besides the message the synchronization line can also become instable, this results in the following MTBF formula:

$$MTBF_{message} = \frac{1}{\frac{bits}{MTBF_{MessageBit}} + \frac{1}{MTBF_{Sync}}}$$

$$bits = size\ message\ is\ bits$$

$$MTBF_{MessageBit} = MTBF\ for\ 1\ bit\ of\ the\ message$$

$$MTBF_{Sync} = MTBF\ for\ synchronization\ bit$$

There is started with the $MTBF_{Sync}$ this used the same formula as for $MTBF_{Response}$ only the direction is the other way around:

$$MTBF_{sync} = \frac{e^{c*\frac{\frac{1}{f_c}-1.3\ ns}{0.4\ ns}}}{f_a f_c * 2.05\ ns}$$

$$f_d = frequency\ of\ input\ data$$

$$f_c = output\ frequency$$

$$c = number\ of\ flip\ flops\ in\ synchronizer$$

Next the $MTBF_{MessageBit}$ is determined, each bit has twice as long to become stable, this can be covered by multiplying the output frequency with 1 divided by the number of flipflops used for the synchronization bit.

$$MTBF_{messageBit} = \frac{e^{\frac{\frac{c}{f_c}-1.3\ ns}{0.4\ ns}}}{\frac{1}{c} f_a f_c * 2.05\ ns}$$

$$f_d = frequency\ of\ input\ data$$

$$f_c = output\ frequency$$

$$c = number\ of\ flip\ flops\ in\ synchronizer$$

To get a better insight in how MTBF reacts to frequency, number of bits and number of flip flops, plots are made. All plots are made for a small frequency range since values can become very big making it impossible to read the graph. The first plot shows how the number of bits influences the MTBF. The number of synchronizer flip flops is kept at two in this case (Figure 39).

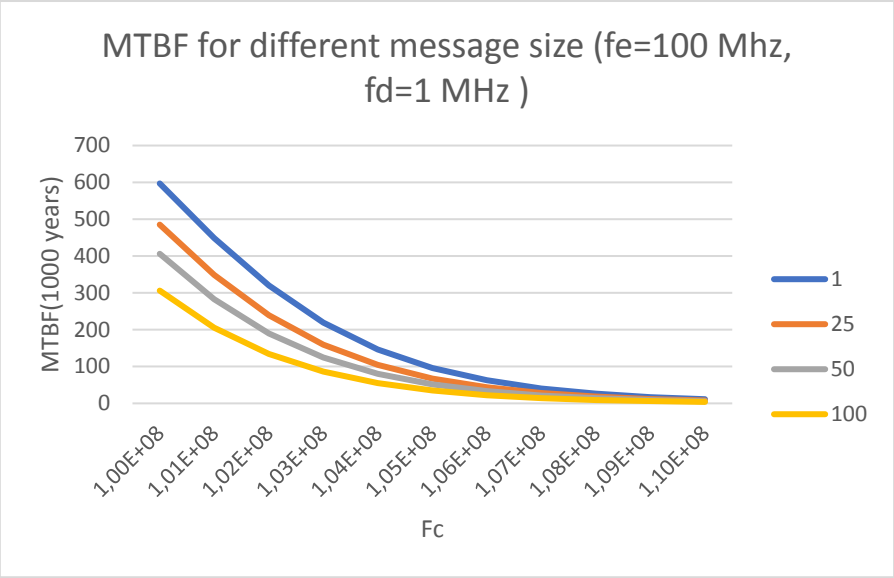


Figure 39: MTBF for different message sizes

As can be seen from the graph the MTBF is halved for 100 times the message size. The second plot shows how the MTBF depends on the in and output frequency. In this case the number of synchronization flip flops is kept at 2, the message size is kept constant at 10 bits (Figure 40).

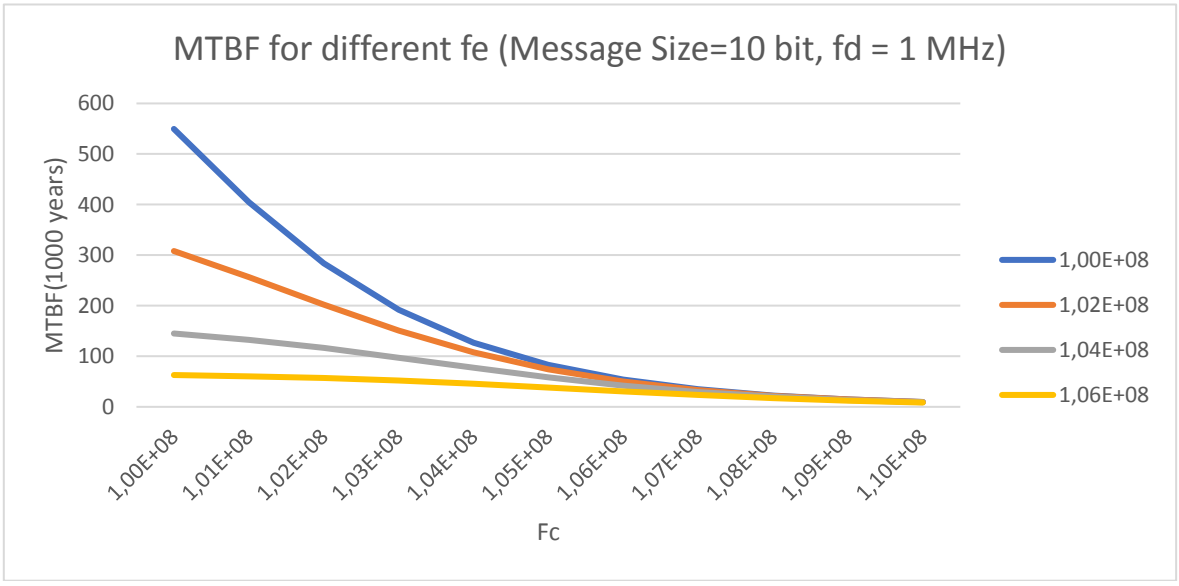


Figure 40:MTBF for different input frequencies

As can be seen from the graphs the influence is substantial, for increasing the input frequency by 6 Mhz the MTBF is divided by 10.

The last graph shows the MTBF for data input speeds (Figure 41).

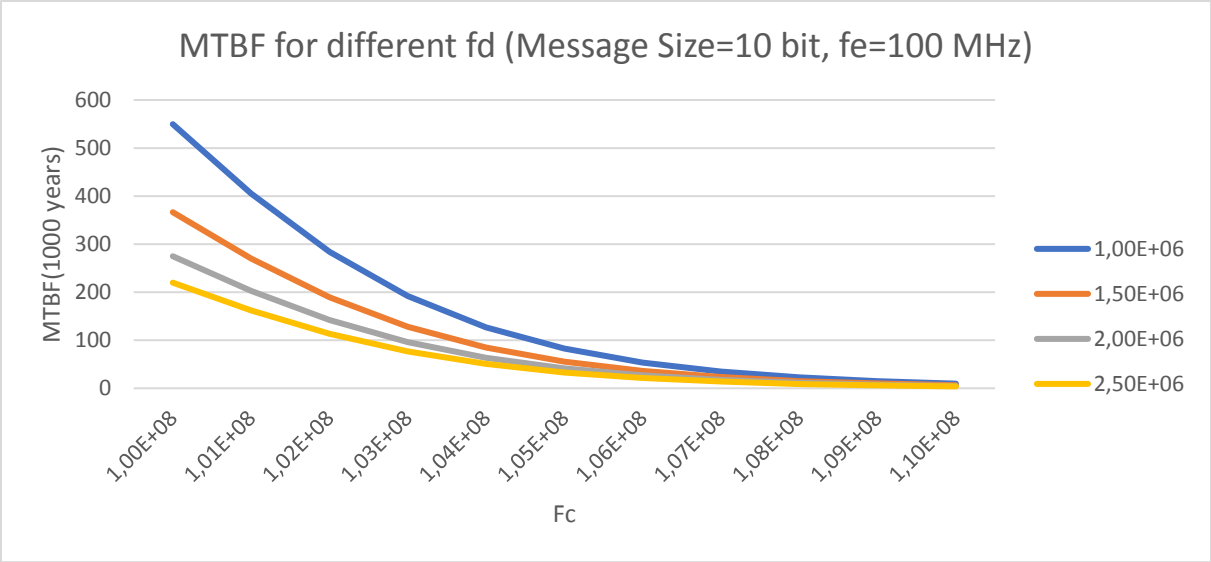


Figure 41: MTBF for different output frequencies

This graph shows that the MTBF is halved when the data input speed is doubled, is should be kept in mind that the influence of the data speed is substantial.

A.3 DERIVING DATAFLOW FROM CLASH

Deriving dataflow from the clash-language is done according to a standard procedure. The procedure is the same for each node, there is started with the state space, the state space is converted to dataflow and then simplified.

A.3.1 DATAFLOW ABSTRACTION

The dataflow graphs can in some cases be simplified, there are simplification methods used throughout the report.

A.3.1.1 MERGE NODES

Nodes can be merged when they happen after each other (data dependency), on the condition that the first node only has inputs and the last node only has outputs (except the edge in between). When merged the delay of both nodes can be added, the input and output edges are also combined into one node (Figure 42).

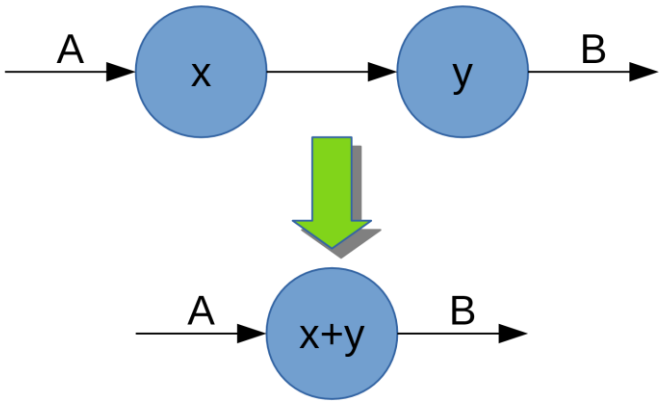


Figure 42: merging two nodes

A.3.1.2 MERGE TOKENS

It is also possible to merge a number of tokens. Imagine three nodes A, B and C, A produces x tokens, C consumes X tokens and B is positioned in between processing these tokens (Figure 43). In this case the delay of node B can be multiplied by x when the tokens produced and consumed is reduced to 1. What happens is that there is assumed that all tokens are consumed at once. Keep in mind that node B should not have any inputs or outputs except from A and C.

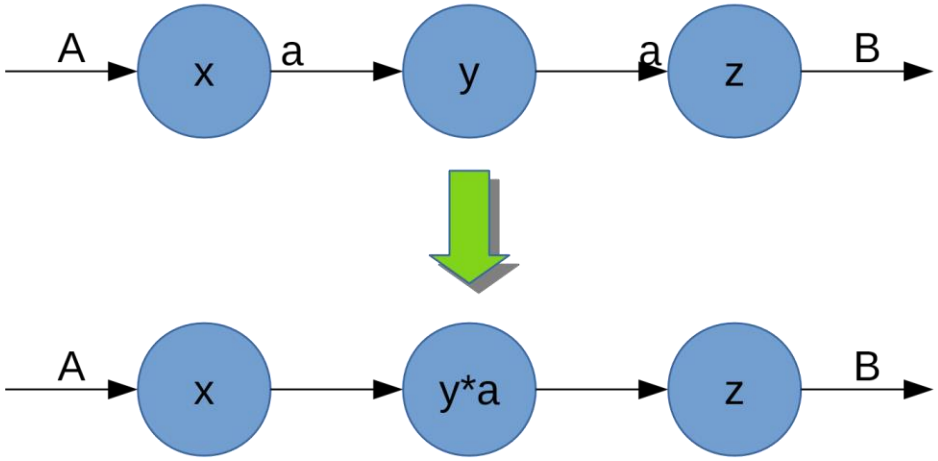


Figure 43: Merge tokens

A.3.2 DERIVATION PROCESS

Deriving a dataflow graph from the Clash design is a standard procedure. Each derivation will be done according to the procedure explained here, as an example the input node is going to be used.

Nodes/Basic/NodeInput.hs

First a state space diagram is derived from the Clash code. This is a diagram showing how the code goes through different states.

```
---------Predefined functions-----------------------------------------
------
    predef1 = case (edgeDataIn == high && edgeCtrlOut == high) of
                True  -> Fire
                False -> Wait

---------Next state---------------------------------------------------
-----
    stateNext = case stateI of
      Wait   -> predef1
      Fire -> predef1
```

As can be seen from the code, for both states (Fire and Wait) the next state is determined by the same condition. The values checked by this condition are the input signals from the controller and input data. The code is used to create a state space diagram (Figure 44):

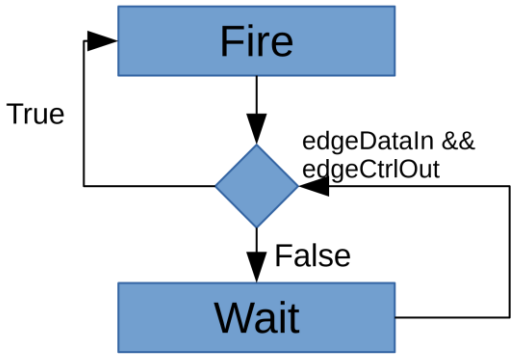


Figure 44: Statespace NodeInput

The state space diagram shows that from both states the same condition is checked as was done by the code. The next step is converting the state space diagram to dataflow. Doing this conversion brings some complexity with it since conditions cannot be represented. To make representing possible some assumptions are done, the input and output signals will be represented by edges where high represents a token and low means there is not. The states will be represented by nodes which have edges for input and output signals and edges for determining the execution order. To determine which inputs and output each state has the code has to be used again, this time the content of the state is examined.

```
    state' = case stateNext of
--            ( edgeResponseOut, ( ctrlIn,  edgeCtrlIn),    stateI)
      Wait -> (             low, ( ctrlIn,         low), stateNext)
      Fire -> (            high, ( dataIn,        high), stateNext)
```

The signals edgeResponseOut and edgeCtrlIn represent output signals and they are only high (produce tokens) for the Fire state. With this information the nodes for the states and their corresponding output edges can be derived (Figure 45).

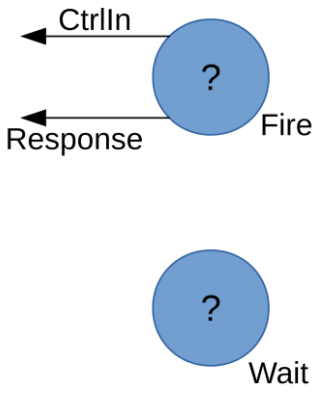


Figure 45: Node Input states in dataflow

Edges connected to external processes like sensors or actuators can be removed since they are assumed to be infinite consumers/producers of tokens. The next step is determining the dependencies and delays of all of the nodes. There is started with the condition, as can be seen from the state space the design stays in the wait state until the condition is satisfied. The condition can also be represented by adding the input edge to the Wait state (Figure 46).

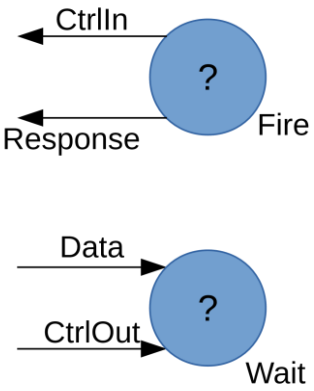


Figure 46: Node Input states of dataflow with inputs

Now only the delays and the dependencies between the nodes have to be added. There is an edge from the Wait to the Fire state since after Wait is finished Fire has to start. When Fire is finished the next state could be Fire or Wait. There has been decided to incorporate the conditions inside the Wait state meaning from the Fire

state it has to go to the Wait state to check the conditions. The delay of each state is normally one clock cycle, however since the Wait state includes a condition it is reduced to 0. The Wait states delay is controlled by its inputs, meaning when an input is detected it should fire immediately. The delay of the node will always be a multiple of 1 clock cycle since it will be connected to nodes which satisfy this property (Figure 47).

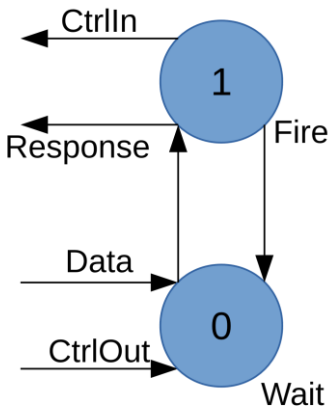


Figure 47: Dataflow NodeInput from statespace (1/2)

The dataflow can be simplified by combining the two states, this results in Figure 48.

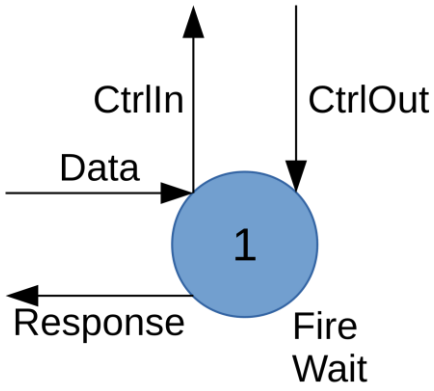


Figure 48: Dataflow NodeInput from statespace (2/2)

A.3.3 OTHER NODES

A.3.3.1 OUTPUT NODE

Nodes/Basic/NodeOutput.hs

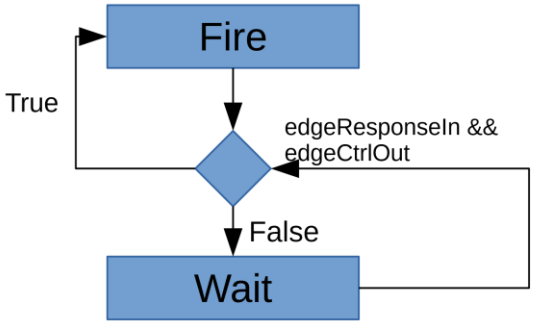


Figure 49: Statespace nodeOutput

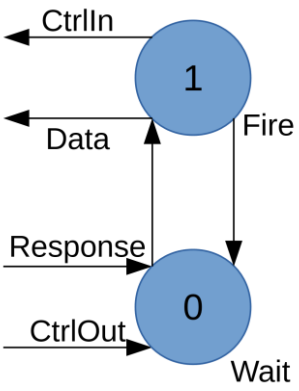


Figure 50: Dataflow NodeOutput from statespace (1/2)

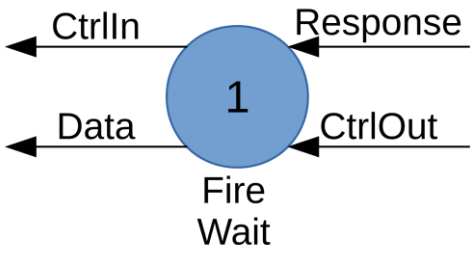


Figure 51: Dataflow NodeOutput from statespace (2/2)

A.3.3.2 BLOCKING NODE

Nodes/Basic/NodeBlock.hs

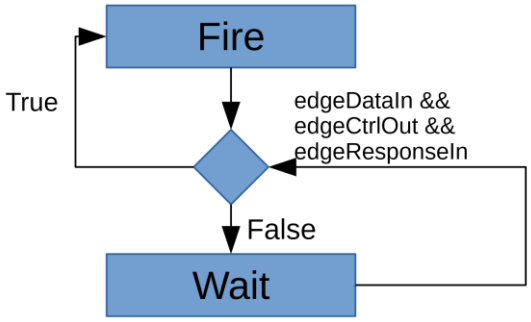


Figure 52: Statespace nodeBlock

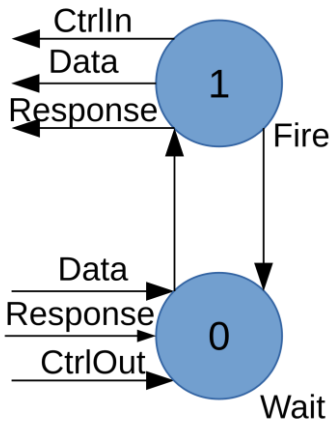


Figure 53: Dataflow NodeBlock from statespace (1/2)

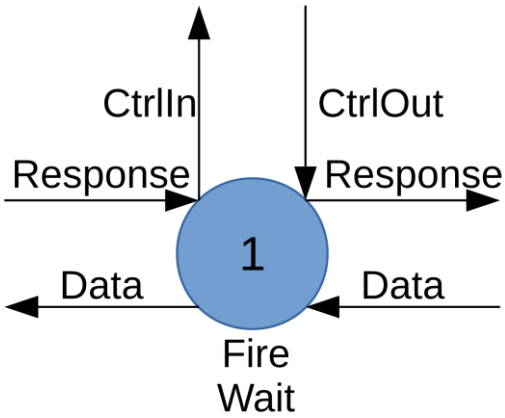


Figure 54: Dataflow NodeBlock from statespace (2/2)

A.3.3.3 NORMAL NODE

Nodes/Basic/NodeNormal.hs

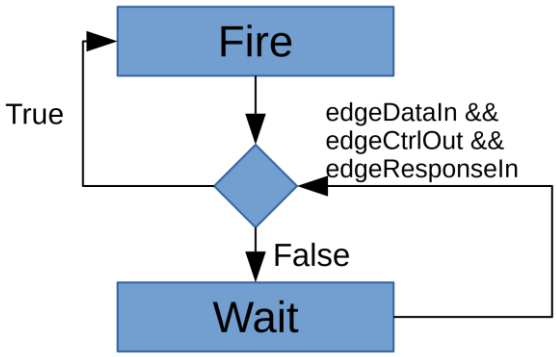


Figure 55: Statespace nodeNormal

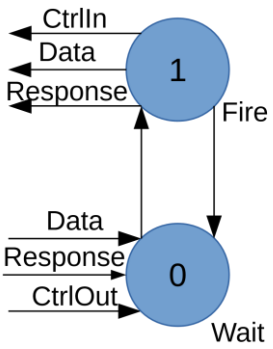


Figure 56: Dataflow NodeNormal from statespace (1/2)

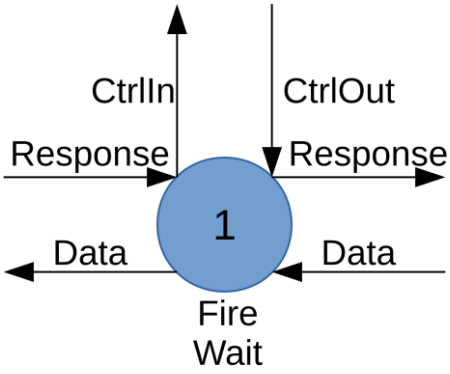


Figure 57: Dataflow NodeNormal from statespace (2/2)

A.3.3.4 SPI INTERFACES

Nodes/Controllers/SPI/Stream.hs and ../StreamR.hs

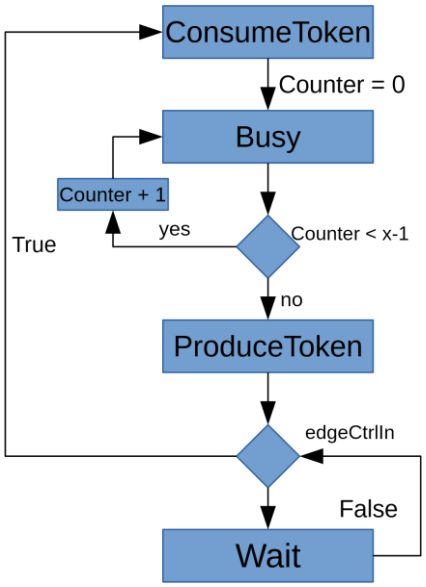


Figure 58: Statespace stream and streamR

The state space is converted to dataflow, the counter condition is implemented as multiple tokens produced on an edge since it force to perform the state busy a x-1 number of times.

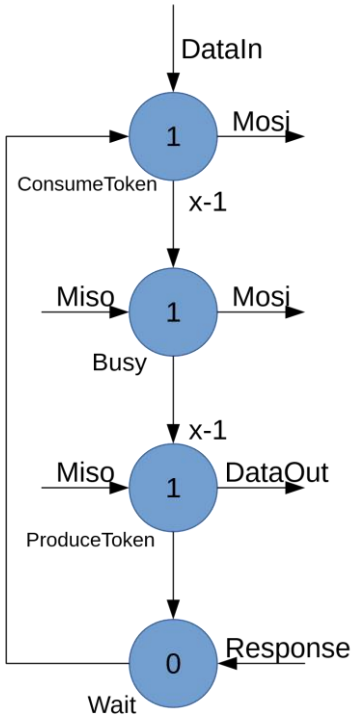


Figure 59: Dataflow Stream and StreamR from statespace (1/3)

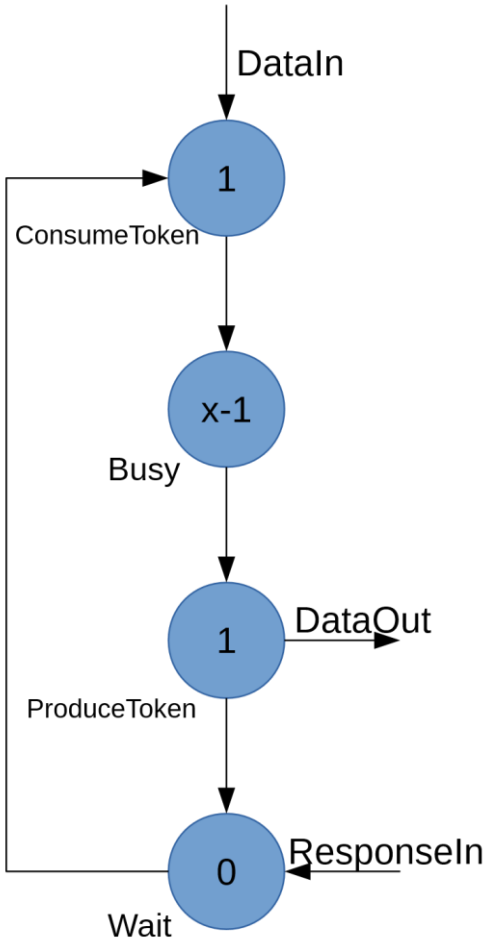


Figure 60: Dataflow Stream and StreamR from statespace (2/3)

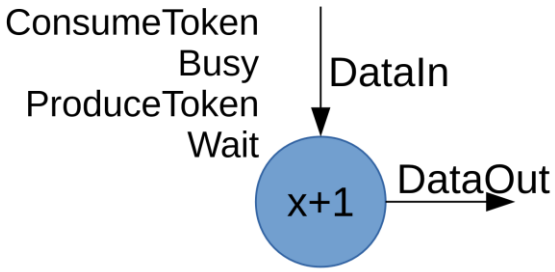


Figure 61: Dataflow Stream and StreamR from statespace (3/3)

A.3.3.5 DELAY CONTROLLER

Nodes/Delay/Delay.hs

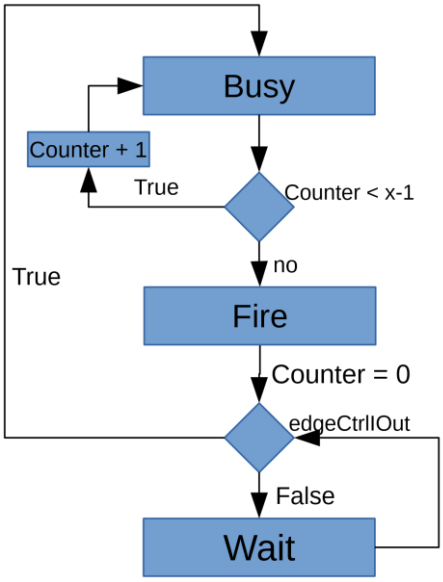


Figure 62: Statespace Delay

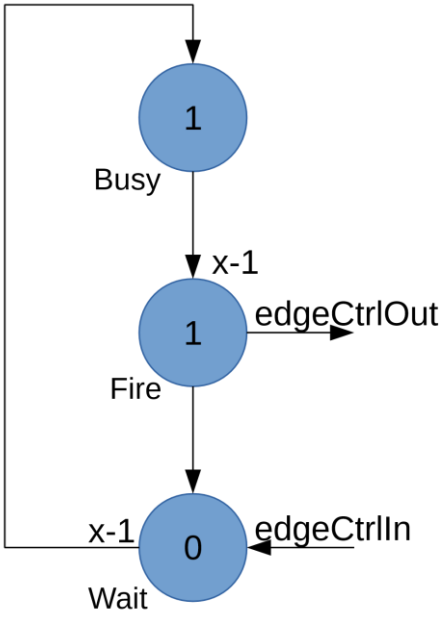


Figure 63: Dataflow Delay from statespace (1/3)

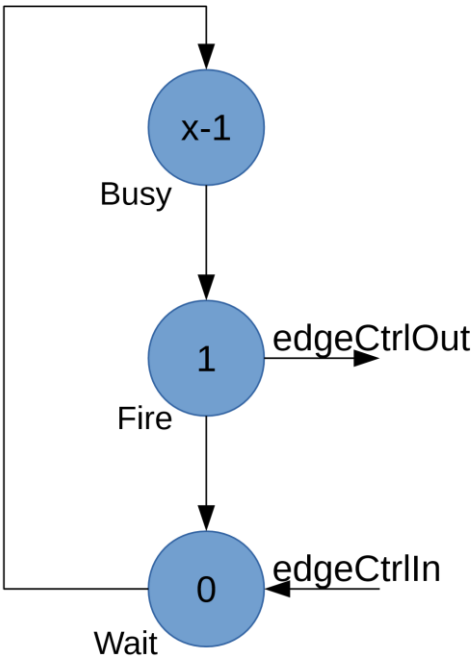


Figure 64: Dataflow Delay from statespace (2/3)

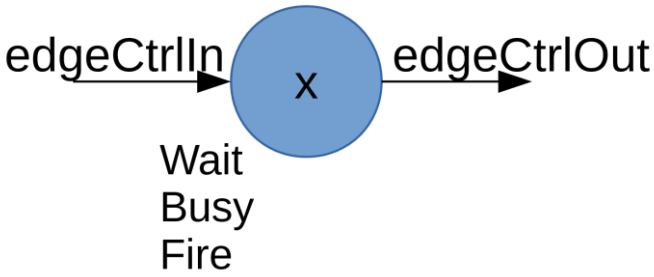


Figure 65: Dataflow Delay from statespace (3/3)

A.4 ADDING CONTROLLERS

Adding controllers is really easy, every node consists of a moore machine which uses predefined types as input and output.

The first step is load the correct modules:

```
import Clash.Explicit.Prelude
import Types.Types
```

With the modules included the states and the state variables are created, the controllers are designed as a moore machine so the output variables are also present in the state:

```
data States = <add states here> | Wait | Fire deriving
(Undefined,Generic)
type State a = (<add own vars here>, EdgeCtrl a, States)
```

The next step is making the control part of the node. A layout is shown which is used throughout the framework, a custom layout can be used as long as the communication part with other nodes is satisfied.

```
--------------------------------------------------------------------
----------
-- Moore functionality
--------------------------------------------------------------------
----------
node :: (KnownNat a)
  => State a
  -> (EdgeCtrl a)
  -> State a
node state input = state'
  where
---------Inputs-------------------------------------------------
----
    (dataIn, edgeDataIn) = input
---------Load State---------------------------------------------
-----
    (<own vars here>, (ctrlOut, edgeCtrlOut), stateI) = state


---------Predefined functions-----------------------------------
------
    predef1 = case (edgeDataIn == high) of
                True  -> <start state which consumes tokens>
                False -> Wait

---------Next state---------------------------------------------
-----
    stateNext = case stateI of
      Wait   -> predef1
      Fire -> predef1 <end state which fires tokens>
     <Custom states are added here>

---------Fill in state------------------------------------------
----
    state' = case stateNext of
--            ( ..             , ( ctrlOut,  edgeCtrlOut),
stateI)
      Wait -> ( <own vars here>, ( ctrlOut,          low),
stateNext)
      Fire -> ( <own vars here>, (  dataIn,         high),
stateNext)
```

```
      <Other states> -> …
```

The next step is adding the output function, this is just copying state variables to the output:

```
-------------------------------------------------------------------------
----------
-- Moore Output
-------------------------------------------------------------------------
----------
nodeOut :: (KnownNat a)
  => State a
  -> EdgeCtrl a
nodeOut state = output
  where
    (<own vars here>, edgeCtrlOut, stateL) = state
    output = edgeCtrlOut
```

The last step is to combine the two functions in a moore machine

```
-------------------------------------------------------------------------
----------
-- Main function
-------------------------------------------------------------------------
----------
main
  :: (KnownNat a)
  => Clock domA Source
  -> Reset domA Asynchronous
  -> Signal domA (EdgeCtrl a)
  -> Signal domA (EdgeCtrl a)
main clk rst = moore clk rst node nodeOut (<own vars>, (0, high),
Wait)
```

A.5 MTBF CALCULATION FOR SYNCHRONIZER SENSOR

To get a feeling on how reliable the system is the MTBF is calculated

$$MTBF_{total} = \frac{1}{\frac{1}{MTBF_{Response}} + \frac{1}{MTBF_{Message}}}$$

To for calculating the MTBF for the response first the “Data speed” needs to be determined, this is the speed with which data is outputted/produced.

$$f_d = \frac{1}{d * d_{out}} = \frac{1}{859 * d_{out}} = \frac{1}{859 * \frac{1}{f_c}} = \frac{f_c}{859} = \frac{101\ MHz}{859} \approx 117579\ Hz = 117{,}579\ kHz$$

$$MTBF_{response} = \frac{e^{d*\frac{\frac{1}{f_e}-1.3\ ns}{0.4\ ns}}}{f_d f_e * 2.05\ ns} = \frac{e^{2*\frac{\frac{1}{2\ MHz}-1.3\ ns}{0.4\ ns}}}{117{,}579\ kHz * 2\ MHz * 2.05\ ns} = many\ years$$

$$MTBF_{message} = \frac{1}{\frac{bits}{MTBF_{MessageBit}} + \frac{1}{MTBF_{Sync}}}$$

$$MTBF_{sync} = \frac{e^{c*\frac{\frac{1}{f_c}-1.3\ ns}{0.4\ ns}}}{f_d f_c * 2.05\ ns} = \frac{e^{2*\frac{\frac{1}{101\ MHz}-1.3\ ns}{0.4\ ns}}}{\frac{1}{2} * 2\ MHz * 101\ MHz * 2.05\ ns} = 6188724 \text{ years}$$

$$MTBF_{messageBit} = \frac{e^{\frac{\frac{c}{f_c}-1.3\ ns}{0.4\ ns}}}{\frac{1}{c} * f_d * f_c * 2.05\ ns} = \frac{e^{\frac{\frac{2}{101\ MHz}-1.3\ ns}{0.4\ ns}}}{\frac{1}{2} * \frac{1}{155} * 2\ MHz * 101\ MHz * 2.05\ ns}$$
$$= 319218573 \text{ years}$$

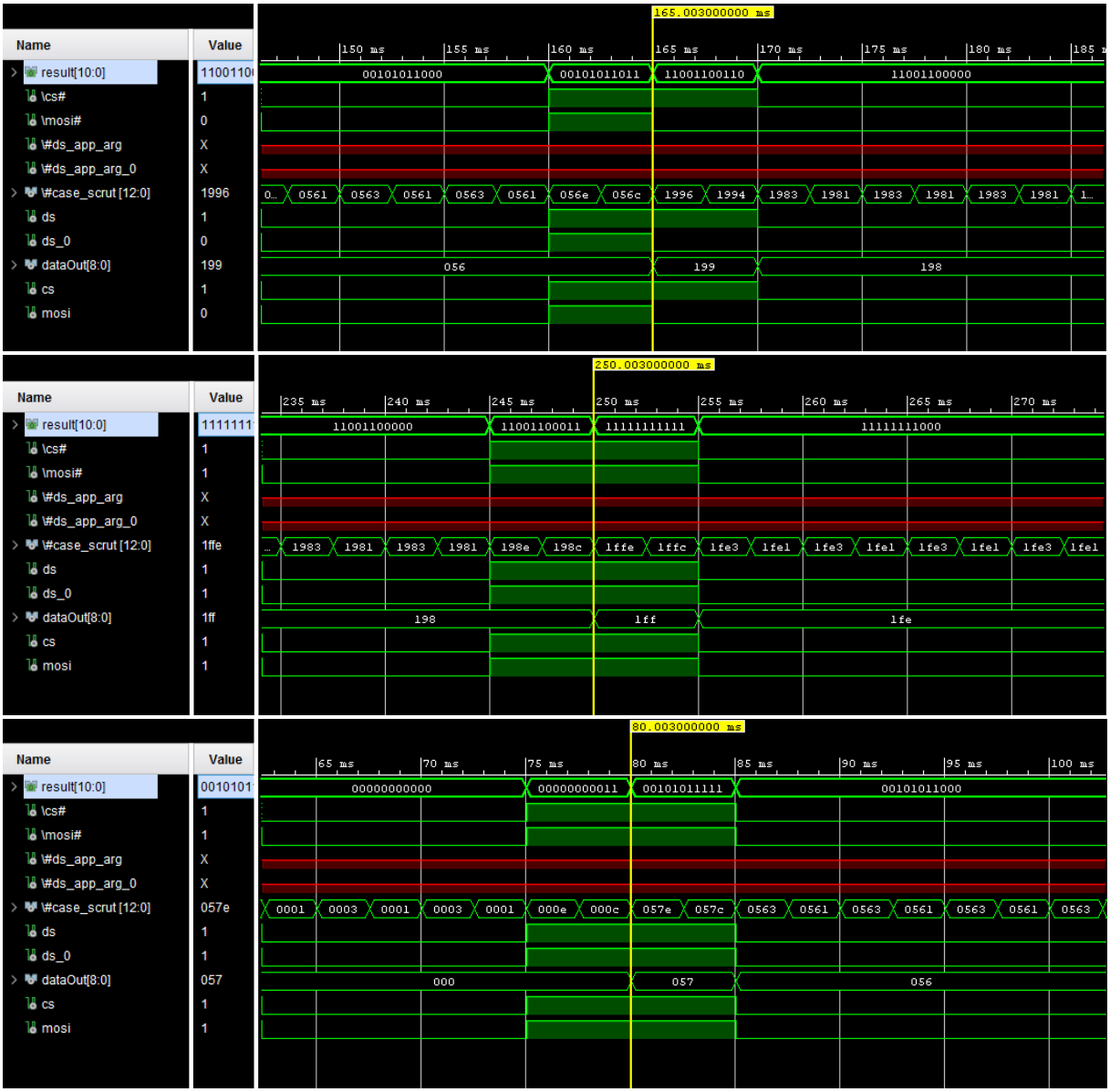
$$MTBF_{message} = \frac{1}{\frac{8}{319218573} + \frac{1}{6188724}} = 5357753{,}12 \text{ years}$$

$$MTBF_{total} = \frac{1}{\frac{1}{very\ big} + \frac{1}{5357753{,}12}} \approx \frac{1}{\frac{1}{5357753{,}12}} = 5357753{,}12\ years$$

A.6 RESULTS SIMULATION

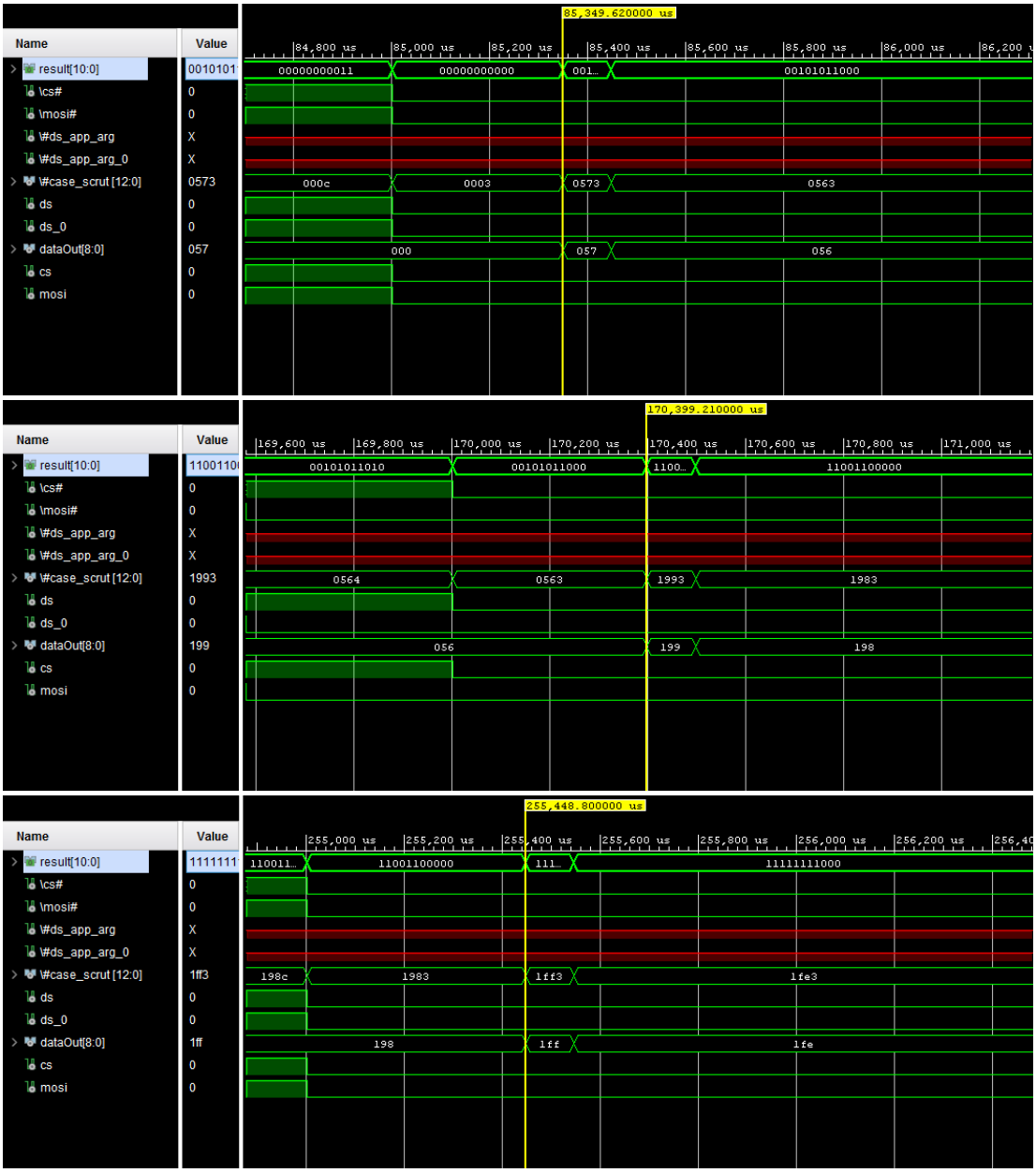
The plots used for determining the output interval are shown here.

A.6.1 FREQUENCIES ARE EQUAL



A.6.2 FREQUENCIES ARE MULTIPLE

A.6.3 FREQUENCIES ARE NOT A MULTIPLE



9 REFERENCES

(2018, June 4). Retrieved from ti.com: http://www.ti.com/lit/ds/scas520h/scas520h.pdf

Altera. (2018, June 4). *Altera.* Retrieved from Altera: http://arantxa.ii.uam.es/~die/%5BLectura%20Timing%5D%20Metaestabilidad%20AN042%20-%20Altera.pdf

Bahukhandi, A. (2018, April 23). *http://www-classes.usc.edu.* Retrieved from http://www-classes.usc.edu: http://www-classes.usc.edu/engr/ee-s/552/coursematerials/ee552-G1.pdf

Bekooij, M. (2017). *Dataflow Analysis for Real-Time Multiprocessor Systems.* Empel: Springer.

blendics. (2018, April 18). Retrieved from blendics: http://blendics.com/wp-content/uploads/2016/08/golson_snug14.pdf

digilent. (2018, September 3). *zybo z7 product page*. Retrieved from Website of digilent: https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/

Edward A. Lee, D. G. (October 1987). Synchronous data flow. *Proceedings of the IEEE*, 1235-1245.

Edward A. Lee, T. M. (May 1995). Dataflow Process Networks. *Proceedings of the IEEE*, 773-801.

Grootte, R. d. (2016). *On the analysis of synchronous dataflow graphs: a system-theoretic perspective.* Enschede: 978-90-365-4041-4.

interfacebus. (2018, June 4). *Design Metastable*. Retrieved from interfacebus.com: http://www.interfacebus.com/Design_MetaStable.html

Patharkar, A. S. (2015). Performance Analysis of Synchronizer and Measurement of Metastability. *International Conference on Computing Communication Control and Automation.* Pune, India: IEEE.

Sachin Hatture, S. D. (2015). Multi-clock domain synchronizers. *International Conference on Computation of Power, Energy, Information and Communication (ICCPEIC).* Chennai, India: IEEE.

Tejas, D., Amit, J., & Divyanshu, J. (2018, April 18). *Synchronizer techniques for multi-clock domain SoCs & FPGAs*. Retrieved from edn.com: https://www.edn.com/electronics-blogs/day-in-the-life-of-a-chip-designer/4435339/Synchronizer-techniques-for-multi-clock-domain-SoCs

Uchevler, B., Svarstad, K., Kuper, J., & Baaij, C. (2013). System-level modelling of dynamic reconfigurable designs using functional programming abstractions. Santa Clara, CA, USA: IEEE.

Wellheuser, C. (2018, June 4). *ti.* Retrieved from ti.com: http://www.ti.com/lit/an/scza004a/scza004a.pdf

Xilinx. (2018, June 4). *Xilinx.* Retrieved from Xilinx: http://userweb.eng.gla.ac.uk/scott.roy/DCD3/technotes.pdf