UNIVERSITY OF TWENTE.

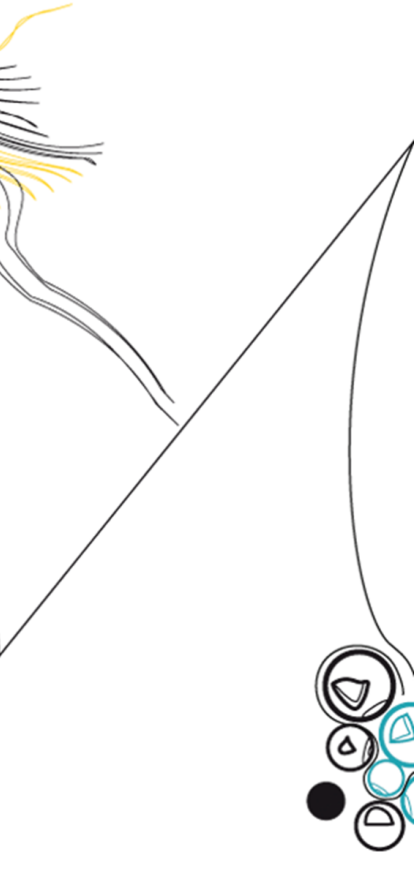
Faculty of Electrical Engineering, Mathematics & Computer Science

Automating System Generation in Cλash

Erik van Raalte [s1879383]

Master thesis

March 12, 2019



Exam committee:
dr. ir. A.B.J. Kokkeler
ir. H.H. Folmer
dr. ir. E. de Groote
dr. ir. A. Hartmanns

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

Summary

There is an increased interest in graduation assignments that involve the usage of the Cλash-language. To prevent that obtained knowledge in these projects is neglected, the CAES chair started a project where multiple students will work on a framework in the Cλash-language that can be utilized for real-time systems. This thesis is written in the early stages of the project. The ability to enable the communication of the (to be created) functional blocks is considered important and should be defined in an early stage of the project. The research question was therefore defined as: How can the unique features of the Cλash language be utilized to define a deterministic System on Chip interconnection standard? This Master thesis proposes a methodology and presents a tool that generates hardware from data flow.

In the background, several alternatives to derive a communication structure on a chip are discussed or excogitated. A conventional SoC communication standard is considered as a useful asset, as it enables to quickly derive a working system in a relatively short amount of time. It is also possible to create a deterministic system, if a simple bus architecture with a static schedule is used. However, implementing a conventional SoC standard does not utilize the properties that the Cλash-language offers. The quick realisation of a prototype is also not set as a requirement. After this conclusion, the Cλash-language was extensively applied a variety of different approaches. Along the way, several limitations are found. In the end, the proposed approaches are compared in a Design Space Exploration (DSE). From the DSE is concluded that a method that enables generation of hardware from a data flow graph and a so called dependency graph is the most interesting option to work out.

In the realisation phase, the data flow to hardware methodology is carefully defined. Thereafter, the methodology is implemented in Haskell and the Cλash-language. In the implementation phase, various limitations of the Cλash-language are found, which resulted in certain design choices for the presented data flow to hardware tool. The supported data flow model is HSDF, but it should not be difficult to support a subset of SDF. In the end, the tool is put to test with two design examples.

From the results was obtained that is possible to generate the components of a hardware architecture, given a set of functions and a schedule that they must comply to. The functional behaviour of the tested examples matched the behaviour that was depicted by the data flow graph. A resource-saving schedule did not result in a proportionally more efficient architecture

for the tested examples, because the tool mainly concerned the scheduling of the functional blocks and not the generation. Furthermore, the definition of a system on a higher level likely introduced overhead. Moreover, synthesis involves compiling the source to a supported (undecipherable) HDL with the Cλash-compiler, and synthesis from the generated HDL to the hardware. In one of the examples, the used synthesis tool could not work with the generated Verilog, while VHDL did not result in any problems. Therefore, bugs and compatibility of the Cλash-compiler can be a liability for the continuation of the project. Another major limitation was that Cλash is very strict with data types. It was opted to automate the nesting of generated components, but no intuitive method was found. Therefore the nesting is for now considered as a manual process in the current implementation.

List of acronyms

IC	Integrated Circuit
HDL	Hardware Description Language
VHDL	VHSIC Hardware Description Language
FPGA	Field Programmable Gate Array
CAES	Computer Architecture for Embedded Systems
LUT	Look Up Table
RTL	Register Transfer Level
P2P	Peer to Peer
HDL	Hardware Description Language
SoC	System on Chip
CPU	Central Processing Unit
GPU	Graphics Processing Unit
ASIC	Application Specific Integrated Circuit
IP	Intellectual Property
AXI	Advanced eXtensible Interface
AMBA	Advanced Microcontroller Bus Architecture
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
TCP/IP	Transmission Control Protocol / Internet Protocol
DSE	Design Space Exploration
RTS2	Real-Time Systems 2
HSDF	Homogeneous Synchronous Data Flow
(MR)SDF	(Multi-Rate) Synchronous Data Flow
CSDF	Cyclo-Static Data Flow
FIFO	First-In First-Out
LFSR	Linear Feedback Shift Register

Contents

Chapter 1

Introduction

A System on Chip (SoC) combines several electronic building blocks to a single chip as an (complex) electronic system. As opposed to conventional systems where components are connected with external wires, the components in a SoC are all internally connected as an Integrated Circuit (IC). This allows the designer to create physically smaller circuits that use less power and achieve a higher operating speed. A Central Processing Unit (CPU) or Graphics Processing Unit (GPU) is a SoC that is designed as an Application Specific Integrated Circuit (ASIC). These integrated circuits are fast, energy efficient and relatively cheap in large quantities. A system is described in an Hardware Description Language (HDL) such as VHSIC Hardware Description Language (VHDL) or Verilog. A cell libary with the available gates and corresponding characteristics is required to synthesize the description to an integrated circuit. Unfortunately, prototyping a system on an ASIC is relatively expensive, since the functionality of the chip cannot be adjusted after the manufacturing process. This is where the Field Programmable Gate Array (FPGA) is advantageous over the ASIC.

An FPGA is technically an ASIC that consists of many reconfigurable logic blocks with functionality that can be set by the designer after the chip has been manufactured. A block is often called a Look Up Table (LUT). These LUTs can be set to several combinational functions or used as memory. The logic blocks can also be inter-wired, providing design freedom. The functionality of an FPGA is described using methods similar to ASIC design. The main difference is that the description is translated to a bitstream using the software that is supplied by the manufacturer, and then programmed on to the FPGA.

During the last decades, the number of LUTs in FPGA's has grown significantly, and this subsequently resulted in an increase in the market size. Still, even though there is more hardware that can be described in the collection of logic cells, the languages did not follow the same trend of growth. Compared to all high level programming languages, Verilog and especially VHDL can feel cumbersome. Starting in 2009, the Computer Architecture for Embedded Systems (CAES) chair within the University of Twente has developed the Cλash-language, which uses a functional approach to describe hardware. The Cλash-language is based on the strongly typed language Haskell. The accompanying Cλash-compiler compiles

Cλash-language code to VHDL or Verilog, that can by synthesized to an FPGA or ASIC. Nowadays Cλash is developed further by the spin-off company Qbaylogic. However, the language is still taught on the university as part of the Embedded Computer Architectures 2 course. It is also possible to do a graduation project that involves the usage of Cλash.

Unfortunately, the CAES chair noticed that a lot of the results of existing master projects were often neglected in new projects. In March '18 the chair started a project in which multiple students participate and can apply their Cλash knowledge in order to design and refine a robot platform that has a Xilinx Zybo Z7 backbone. One of the aspects in which the CAES chair wants to distinguish itself from the Robotics and Mechatronics chair on the University, is that it wants the robot to operate real-time and be deterministic. The Cλash-language is a functional and therefore pure language. It's properties could be useful in designing such a system.

Real-time systems is one of the main research fields within the CAES chair. The concept "real-time" is used in many contexts. In the context of this thesis, a system is real-time if the exact behaviour can be characterized for all possible inputs and states that might occur. In other words, it should be deterministic.

1.1 Problem description

When several students work on their own sub-system or Intellectual Property (IP) blocks within the robot platform, it is of great significance that there is a communication standard in the early stages of the project. Especially if these components rely on information from each other. This Master project aims to resolve this necessity. The research topic during this project is defined as: *How can the unique features of the Cλash language be utilized to define a deterministic System on Chip interconnection standard?*

1.2 Outline of this thesis

This section describes the path that was taken throughout this project. In Part I, The background of the research is presented. The background consists of four chapters. In Chapter 2, the fundamentals of the Cλash-language and compiler are discussed. The remaining chapters discusses several approaches to realise communication on FPGAs.

The first approach is discussed in Chapter 3. In this chapter, the characteristics of existing protocols that are used to communicate on FPGA systems are presented and compared, it concludes with the rudimental demonstration of the Wishbone standard in Cλash. The second approach is listed in Chapter 4. It describes attempts to optimize conventional FPGA communication by using properties of the Cλash-language. The third approach (Chapter 5) is entirely different, and proposes generation of a hardware architecture from data flow. The

chapter discusses the fundamentals of data flow models and proposes how the model can relate to hardware designs. Then, the proposed approaches are compared to each other in the Design Space Exploration (DSE) that is listed in Chapter 6. The approach from Chapter 5 was considered the most the most valuable.

Part II consists of four chapters. Chapter 7 elaborates of the data flow to hardware methodology that was chosen in the DSE. The elaborated method was also implemented during this Master project. The implementation results of the method, along with a conclusion and discussion are presented in Chapters 8, 9 and 10, respectively.

Part I

Background

Chapter 2

CλaSH

The purpose of this chapter is to provide the reader with some of the fundamentals of The CλaSH-language and compiler. Although some parts are quite comphrehensible for those new to functional programming, other parts are more advanced. For this reason, all examples are accompanied by a diagram that depicts what the code actually does on a hardware level. The CλaSH website explains clearly why one could use the CλaSH-language for hardware description [1]:

The CλaSH-language is a functional hardware description language that borrows both its syntax and semantics from the functional programming language Haskell. It provides a familiar structural design approach to both combinational and synchronous sequential circuits. The CλaSH-compiler transforms these high-level descriptions to low-level synthesizable VHDL, Verilog, or SystemVerilog.

2.1 Fundamentals

The CλaSH-language implements a subset of the Haskell functions, and adds additional functionality for digital circuit design. Generating a synchronous (updated on the rising edge of the clock) function from a pure (without internal state) combinatorial function can be accomplished with a `mealy`, `moore` or `register` function. The simplified data type declaration of a mealy machine is as following:

```
mealy :: (s->i->(s,o)) -> s -> Signal i -> Signal o
```

The first argument of the `mealy` function, is a function with data types that must comply to a specific form, namely: `(s->i->(s,o))`. It has the current state of data type `s` and the input of data type `i` as input, and it produces a tuple consisting of the next state of data type `s` and an output of data type `o`. Furthermore, the internal state, which should be of the data type `s`, has an initial value that is given as the second argument of the `mealy` function.

The `mealy` function converts the combinatorial function, to a block that can capture state. It replaces the current state by the new state on the rising edge of the clock. However, Haskell

is a pure language without internal state. Therefore, variables in the discrete time domain are packed in the `Signal` data type, which is represented as a infinitely long stream of states of that variable. The variables are ”updated” on the rising edge of the clock that is defined in the system. This is the reason the last argument (the input), and the return value of the mealy machine (the output) are both of the `Signal` data type.

A basic example is shown in Listing 1 and depicted as hardware in Figure 2.1. The function `f` (line 1-4) adds the input `i` and current state `s` , and assigns as the new state `s'` (line 3). The current state is assigned to the output (line 4). The mealy machine is constructed in the `topEntity` function (line 6-7). Function `f`, which has the required form, is given as first argument. The initial state (the value `0`) is assigned as the initial register value in the second argument. The `topEntity` is the function that the CλAsh-compiler compiles to an HDL. Dependencies of the `topEntity` function are compiled as well. The data type declaration of the top entity must be monomorphic (exist in only one form), in order to be compiled to an HDL. In this example a `Signed 6` data type is used.

```
1        f s i = (s',o)
2          where
3            s' = s + i
4            o = s
5
6        topEntity :: Signal (Signed 6) -> Signal (Signed 6)
7        topEntity = mealy f 0
```

Listing 1: Mealy machine example

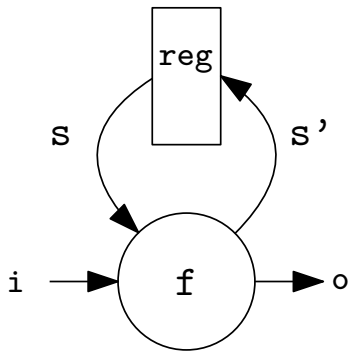


Figure 2.1: The mealy machine

2.2 Higher Order Functions

The aforementioned example shows how combinatorial logic like an adder can be used in a synchronous manner within the mealy machine. Another aspect that makes Haskell and the CλAsh-language powerful are the Higher Order Functions (HOFs). HOFs are functions within functions. For example the `foldl` function:

```
foldl :: (b -> a -> b) -> b -> Vec n a -> b
```

The `foldl` first takes a function (e.g. an arithmetic operation) that takes two data types (`b` and `a`) and produces a result of `b`. The next argument is the initial value of data type `b` and the last argument is a vector of data type `a`. The result is a value of data type `b`. The structure of a `foldl` is depicted in Figure 2.2 and Listing 2. The initial value `a` is applied with the first entry of the vector `xs` to the function `f`. The result of this operation is passed as argument for the second instance of `f`, along with the second record of the vector `xs`. This continues until the end of the vector is reached and the answer is computed. Note that this HoF is still purely combinatorial.

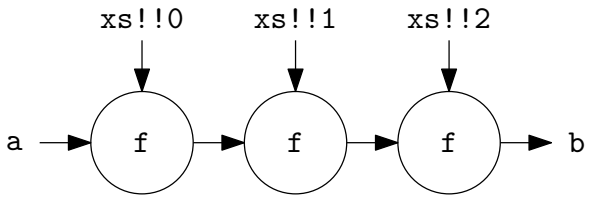


Figure 2.2: Visualization of the `foldl` function

```
1  b :: (b -> a -> b) -> b -> Vec n a -> b
2  b f a xs = foldl f a xs
```

Listing 2: `foldl` declaration

If the vector `xs` is fairly large, then the function `f` is replicated more often. Despite that the functional description is still correct, it might result in trouble when the algorithm is mapped on an FPGA, due to the larger area and longer critical path. The functional correct expression can be rewritten to be advantagous in terms of speed or area. Figure 2.3 and Listing 3 depict a version of the `foldl` with registers between the instantiations of the function `f`. As a result, this circuit has a shorter critical path. The function is no longer purely combinatorial, and requires a description that involves the mealy machine that was discussed in Section 2.1. First a function called `bT` is described. It has a form such that it can be used as an argument of the mealy function.

Line 1-7 lists the constraints that the function `bT` must comply to. It has the structure where inputs are shifted through a set of registers, where the function `f` is applied to the intermediate results. The fields `rs` and `rs'` hold the current and next state of the registers (line 8). Line 11 is the most interesting for this example. First, the input `a` is shifted into the vector of registers that hold the current state (`a +>> rs`). The next state of the vector (`rs'`), is described as function `f` executed over both (`a +>> rs`) and `xs`. This is accomplished by the `zipWith` function. Line 14-18 describes the data type definitions of the mealy machine and line 19 describes the initialization of the mealy machine. The first argument is the function `(bT (1:>2:>3:>Nil) (+))`, which complies with input requirement of the mealy machine, see Section 2.1. The second argument is the initial state of the registers, which are all set to `0`.

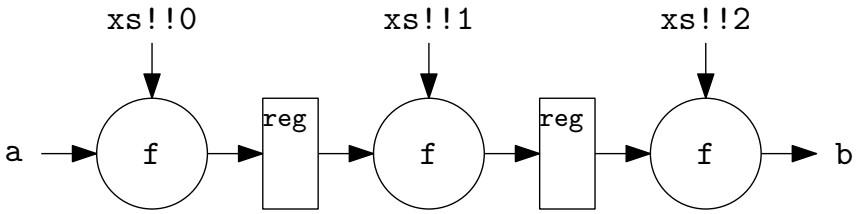


Figure 2.3: Visualization of a pipelined instance of the `foldl` function

```
1  bT
2    :: KnownNat n
3    => Vec (n+1) a
4    -> (a->a->a)
5    -> Vec (n+1) a
6    -> a
7    -> (Vec (n+1) a,a)
8  bT xs f rs a = (rs',o)
9    where
10     o = last rs'
11     rs' = zipWith f (a +>> rs) xs
12
13 -- Initialisation:
14 b
15   :: HiddenClockReset domain gated synchronous
16   => Num a
17   => Signal domain a
18   -> Signal domain a
19 b = mealy (bT (1:>2:>3:>Nil) (+)) (repeat 0)
```

Listing 3: Foldl optimized for throughput by means of pipelining

Figure 2.4 and Listing 4 show an instance of the foldl that is optimized for size. As opposed to the pipelined version, where the functions are chained after each other, this version used a self loop. This approach can be more efficient in area usage, because it uses only two variables to hold the current state, and only one instantiation of the function `f`. However, it will not produce a valid output on every clock edge. It therefore has a lower throughput than the pipelined instance. Again `bT` (line 1-12) is in a form, such that it can be supplied as argument to the instantiation of the mealy machine (line 15-20). Instead of applying the function `f` on the intermediate values that are contained in the vector of registers, there is only one function and one register that contains the intermediate result. On line 11, the new value of the intermediate result is assigned. It only fetches the input when a state counter has reached the maximum value, otherwise it processes the intermediate result. An additional register that holds the value of the counter, that is used to monitor the state of the intermediate value (line 10). The output only holds the result when the counter has reached the maximum, otherwise it holds `0` (line 12). This implementation is flawed, because there is no distinct signalling of the validity of the output signal. A solution is addressed in Section 2.4. In conclusion, the Cλash-language allows the designer to first focus on the algorithm, and then on the mapping on hardware.

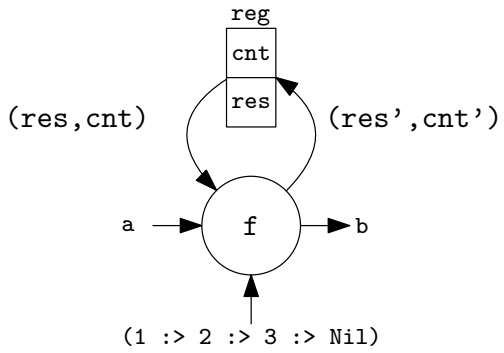


Figure 2.4: Visualization of a small area instance of the `foldl` function

```
1  bT
2    :: (KnownNat n, Num a)
3    => Vec n a
4    -> (a->a->a)
5    -> (Int,a)
6    -> a
7    -> ((Int,a),a)
8  bT xs f (cnt,res) a = ((cnt',res'),o)
9    where
10     cnt' = if cnt == length xs - 1 then 1 else cnt + 1
11     res' = if cnt == length xs - 1 then f a (xs !! 0)  else f res (xs !! cnt)
12     o    = if cnt == length xs - 1 then res' else 0
13
14 -- Initialisation:
15 b
16   :: HiddenClockReset domain gated synchronous
17   => Num a
18   => Signal domain a
19   -> Signal domain a
20 b = mealy (bT (1:>2:>3:>Nil) (+)) (0,0)
```

Listing 4: Foldl optimized for area consumption, without status signals

2.3 REPL and simulation

Another useful aspect of the CλAsh-language, is the ability to simulate every expression in an interactive environment called `clashi`. This makes it tangible to evaluate a system consisting of multiple expressions seperately. After the whole system is simulated in `clashi`, it can be compiled to a synthesizable HDL, a testbench for Vivado or Modelsim can automatically be generated as well. The resulting simulation should have the same behaviour as the `clashi` simulation. A common practice is to first define the system in the CλAsh-language and simulate it in the `clashi` environment and then generate the testbench to simulate it with the tools provided by the FPGA supplier.

2.4 Algebraic data types

The last aspect of the CλAsh-language that is discussed in this chapter are the algebraic data types. Instances of data types can be passed as arguments of functions and are also

results of functions. A basic data type is the `Bool` data type, which is monomorphic (can only exist in one form), because it's representation is unambiguous:

```
1 data Bool = False | True
```

An instance of `Bool` can be either `True` or `False`. There are also data types that take a parameter and therefore exist in more than one form, for example the polymorphic (which means that it can exist in multiple forms) `Maybe` data type:

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

Within this data type, `a` can be any data type (e.g. `Int` or `String`), but `Nothing` will always be `Nothing`. It is useful in functions that do not produce a result at all times. It can for instance be applied to improve the example that was depicted in Figure 2.4. The improved code is shown in Listing 5. The example is more complex and only the crucial differences are discussed. Instead of just inputs and outputs of the data type `a`, this example has wrapped `a` in the `Maybe` data type. For the input holds, as long as it is `Nothing`, no new value is fetched. A value of `Just a` fetches the value `a` and resets the counter. The output in this example is `Nothing`, as long as it is not valid, otherwise it is `Just <result>`. The initialization of the mealy machine in line 20-25 is changed, as the input and output are of the `Signal domain (Maybe a)` data type.

```
1  bT
2    :: (KnownNat n, Num a)
3    => Vec n a
4    -> (a -> a -> a)
5    -> (Int,a)
6    -> Maybe a
7    -> ((Int,a),Maybe a)
8  bT xs f (cnt,res) a = ((cnt',res'),o)
9    where
10     (cnt',res',o) =
11       case a of
12         Just r -> (1,f r (xs !! 0), Nothing)
13         Nothing ->
14           if cnt == length xs - 1 then
15             (0,f res (xs !! cnt),Just res')
16           else
17             (cnt+1,f res (xs !! cnt),Nothing)
18
19 -- Initialisation:
20 b
21   :: HiddenClockReset domain gated synchronous
22   => Num a
23   => Signal domain (Maybe a)
24   -> Signal domain (Maybe a)
25 b = mealy (bT (1:>2:>3:>Nil) (+)) (0,0)
```

Listing 5: Foldl optimized for area consumption with `Maybe` data type

Data types can also be created by the designer. By pattern matching on the data type, an elementary arithmetic logic unit can be created in a few lines of code, see Listing 6. The executed function depends on the first argument, which is of the `Operator` data type that is defined on line 1. The function then takes two additional arguments of data type `a`, that

belong the the `Num` class. It produces a result that is of data type `a` as well (line 4). E.g. `alu Add 1 2` executes the function on line 5, and `alu CmpGt 1 2` executes the function on line 9.

```
1  data Operator = Add | Sub | Incr | Imm | CmpGt
2    deriving (Eq,Show)
3
4  alu :: (Num a) => Operator -> a -> a -> a
5  alu Add   x y = x + y
6  alu Sub   x y = x - y
7  alu Incr  x _ = x + 1
8  alu Imm   x _ = x
9  alu CmpGt x y =, if x > y then 1 else 0
```

Listing 6: Arithmetic Logic Unit in the CλAsh-language

In conclusion, there are many additional options that the CλAsh-language provides in contrast to conventional languages like VHDL and Verilog. The next chapters explain the process on how to utilize the CλAsh-language and compiler on conventional SoC communication standards, as well as new approaches to enable SoC communication. In the remainder of this thesis, both the CλAsh-language and CλAsh-compiler are designated under the name CλAsh, except when a specific property is addressed.

Chapter 3

Comparison between existing SoC communication standards

The aim of this chapter is to understand the basic concepts of commonly used communication protocols in the industry. The full study on existing SoC communication standards is presented in Appendix A.

There exist several interfacing protocols for Systems on Chips. The provided IP blocks for Xilinx FPGA's are for example connected through the Advanced eXtensible Interface (AXI) [2] from ARM. The standard Intel Altera FPGA peripherals communicate over the Avalon bus [3]. Other common protocols are the Advanced Microcontroller Bus Architecture (AMBA) [4] from ARM and the Wishbone bus [5] from Opencores. This chapter concerns a comparison and overview of common features and methodologies used in existing SoC protocols. More elaboration on the individual protocols can be found in Appendix A.

The protocols that are described in Appendix A feature some common practices. The only communication standard that is entirely different is the network on a chip methodology, which is only briefly discussed in this chapter and not in the appendix.

3.1 Characteristics of existing communication standards

All of the standards introduce a generic interface which makes the task of adding new IP blocks to an existing SoC design a less cumbersome task for the hardware designer. This procedure makes hardware design more lean, as components or even whole busses can be re-used for other designs. A choice for a bus itself becomes clear when many components are connected together. Wires in a chip cost area, and increase the power consumption of a chip. A bus is a more efficient method of connecting various components to each other. It often features an arbiter that selects which components can communicate with others. This concept is elaborated in more detail in the next paragraphs. The usage of a bus standard is almost a neccesity in the design of a chip. However, there are different bus standards which all offer different properties.

It is common that a bus consists of master and slave interfaces. An IP that is connected to a master interface, may initiate transfers to an IP that is connected to a slave interface. A slave interface does in general not initiate a transfer, and just responds to a master. A bus consist of a group of masters and slaves that can communicate by means of an interconnect, see Figure 3.1. The masters M_0 and M_1 may communicate with slaves S_0, S_1 and S_2. If every master had a connection to every slave, that would results in many wires. An interconnect is a device which directs communication between masters and slaves, by routing them together, and thus saving wires and energy.

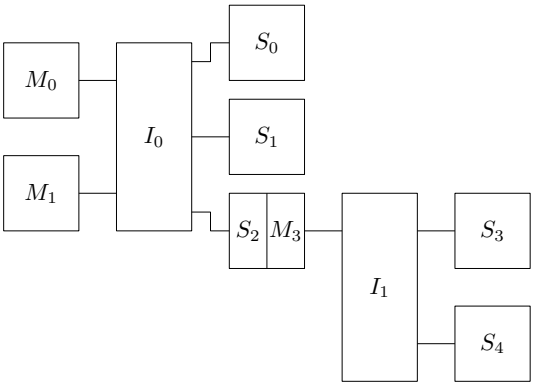
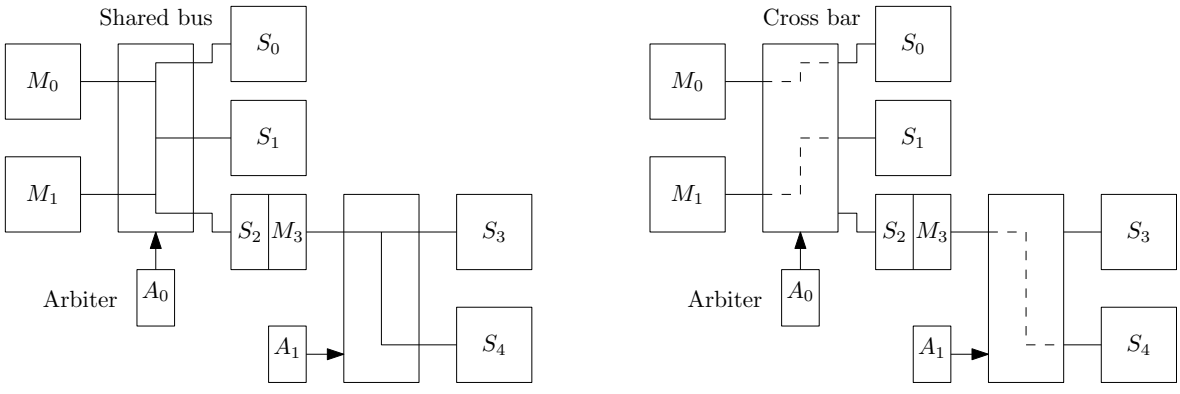


Figure 3.1: An example bus topology

There exist several types of interconnects. One interconnect type is a shared bus, as is used in the AMBA protocol. In a shared bus, all masters and slaves are connected together, see Figure 3.2a. Since only one instance may communicate over the interconnect at the time, an arbiter that directs the communication is required to make this work. Another approach is a so called crossbar interconnect, which creates a physical connection between a master and a slave, so that multiple masters can communicate with a different slave, at the same time, see Figure 3.2b. The latter has the advantage that it can provide a higher throughput since masters can simultaneously communicate with slaves. However, it has a larger hardware utilization.



(a) Shared bus interconnect

(b) Crossbar interconnect

Figure 3.2: Different variations of interconnect

There are also types of communication that do not involve an interconnect. One of them is so called Peer to Peer (P2P) communication where a master interface is directly connected to a slave interface. A chain of P2P interfaces (an IP block can feature a slave and a master interface at the same time) is sometimes also referred to as a pipeline interface. These kind of interfaces are used when high datarates are required (for example image processing). Commonly used communication standards differ in the support of Master to Slave connection methods.

Another distinction is the protocol that is used to communicate. Every protocol uses a different set of signals to send and receive commands and data. The AMBA standard even offers two seperate interfaces that offer a different kind of communication (Advanced High-performance Bus (AHB) and Advanced Peripheral Bus (APB)). On a high level each communication protocol performs the same function namely, transfer data or request data, but the different implementations lead to different characteristics on both area and performance. Some protocols allow so called burst transfers, where a master does a single request and the slave responds in multiple words. Most protocols do also support embedded status messages about the transfers. This can be used to optimize the scheduling in the interconnect. For example, a slave is given a burst request and it can only give a part of the response. Then it can notify the master that it did not send the entire message yet. These are called split transfers. Many protocols also support pipelining, where a master can queue several requests at a slave, it therefore does not have to wait for a response to initiate a new request.

3.2 Network on a chip

An entirely different and fairly recent approach is a Network on a Chip. As the name implies, it borrows some of the semantics of the Transmission Control Protocol / Internet Protocol (TCP/IP) stack in terms of communication between components. The concept is, to place IP blocks in a (two dimensional) grid and attach a switch to every IP, see Figure 3.3. A switch that receives a package can either pass it through the IP block it is connected to, or pass it to another switch that is closer to the destination IP that the package is headed to. Some studies show that an architecture like this can reduce the power consumption significantly when there is much traffic between components.

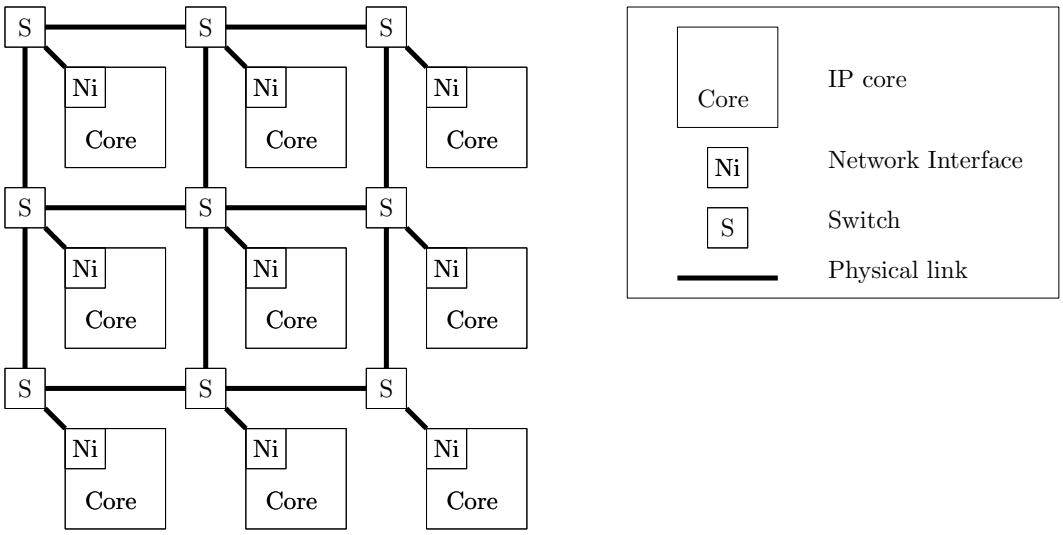


Figure 3.3: A Network on a Chip architecture from [6]

3.3 Comparison

This study clarified the basic concepts of some SoC communication standards, along with a recent approach called Network on a Chip. The communication standards that are discussed with more comprehension in Appendix A, share some of the characteristics that were discussed in Section 3.1. However, some standards are more complex then others. The FPGA architecture that is used is relevant in the choice of the communication standard as well. The Avalon standard is only used by Altera IP blocks. Therefore it would not make sense to use this standard in a Xilinx FPGA, as there are simply no IP blocks availible that comply to this standard. The Wishbone bus comes with a variety of open source packages that work on both Xilinx and Intel Altera FPGA's [7]. Xilinx IP's use the AXI4 bus, but the standard is very comprehensive to code by hand. Xilinx offers interface generation tools, but they do lack the support of code generation to Cλash. Finally, the Network on a Chip methodology is considered overly complex for the application that is aimed to be build.

The Wishbone is likely the best choice if an existing standard is chosen. The protocol is simple and yet versatile as it supports a majority of the options that more complex protocols offer. A rudimentarily Wishbone master and slave are described in Appendix B of this thesis. Although the description is relatively simple, it does not utilize the unique properties of Cλash, like higher order functions. Therefore, the description is similar to a conventional HDL description. The usage of a conventional communication standard is compared to other alternatives in the DSE in Chapter 6.

Chapter 4

Existing Communication optimized with Cλash

This chapter describes a part of the research that was done as after the implementation of the Wishbone master and slave (see Appendix B). Although the Wishbone master and slave worked as expected, it still required a large amount of manual labor to connect a set of slaves and masters to an interconnect. In an environment like Quartus or Vivado this is relatively easy to connect interfaces together, due to the availability of a block diagram editor and extensive support for conventional hardware description languages.

The aim of this chapter is, to find a method that inherits properties of existing SoC standards, while utilizing the properties of Cλash to connect components in a more elegant manner. Moreover, it also investigates the possibilities to use algebraic datatypes to define a more intuitive communication protocol, as Cλash is less readable on bit level, opposed to Verilog or VHDL due to the strict type system.

4.1 Assignment of slaves to an address space

A master can communicate to slaves through an address space. An interconnect can either communicate the address the master requests to all slaves, where the slaves have to do the decoding, or use an interconnect that decodes the addresses. This is explained best through by means of an example:

- One master with 4 bit address space
- One slave that has a 3 bit address space
- Two Slaves with 2 bit address space

The 4 bit address space can address all the slaves, as shown in the right part of Figure 4.1. An hardware architecture that could be used to do the decoding to the corresponding slaves

is shown in the left part of the figure. The most significant bit functions as a control bit for the subsequent multiplexer.

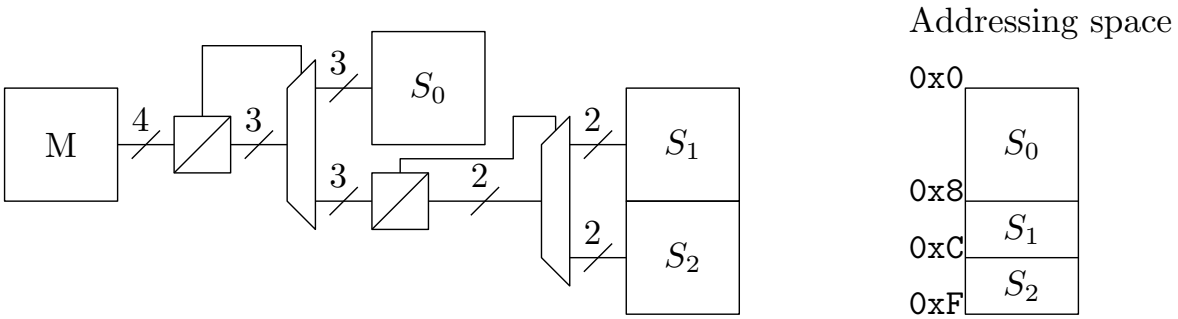


Figure 4.1: Hardware and address space allocation for the presented example

Ideally, one would specify a master with an accompanying addressing space, along with a list of slaves and their adressing space. The aim was to use CλaSh to automatically derive a interconnect architecture that would allocate the slaves within the addressing space. Unfortunately, after many attempts this turned out to be unfeasible mainly because it would require multiple slaves with varying address space as argument. This is not possible due to the semantics of the CλaSh-language. The only solution that worked was to generate an individual addressing filter for each slave. However, this creates a large amount of redundant hardware. For example, S_1 and S_2 in Figure 4.1 share the first multiplexer, this would be replicated with the aforementioned filtering method. An alternative approach, that has not been worked out, is to outsource the address decoding to the slave itself and transfer the whole message from the master to all the slaves.

4.2 Communication with algebraic data types

Another approach that is attempted, is a communication standard that relies entirely on the algebraic datatypes. This method was neglected early when it turned out that it could not possibly result in an efficient Register Transfer Level (RTL) implementation. However, it is included to clarify the steps that have been taken within this master project.

The concept is to create slaves that utilize an unique instruction set. The master can then run the instructions that the slave supports directly. A rudimental example that controls the state of 4 LEDs is shown in Listing 7. The `OpCode` and `RspCode` (line 5-6) data types show the instructions and responses that the slave supports. The `State` data type (line 4) represents the data that is registered in the slave. The behaviour of the slave itself is defined in `slaveT` (line 8-14). Note that the type declaration is conform with a function that can be supplied to a mealy machine. The `slaveT` has an internal state of type `State` as first argument. The second argument (the input), is a value of the type `OpCode`. The output of the function is the the response to the master (`RspCode`) as well as the updated state of type `State`. The listed slave supports three commands: Write a pattern to the LEDs

(`Write State`), read the current pattern in the LEDs (`Read`) or do nothing (`OpIdle`). In the example that is considered in the next paragraph, another slave that controls an RGB LED is considered as well. The internals of this slave are similar to the earlier explained LED, and therefore not shown.

The master can communicate with both the slaves as shown in Listing 8. The data type `Slaves` (line 1) contains all possible operation codes, as it is either `Led Led.OpCode` or `Rgb Rgb.OpCode`. The advantage of this method is that it does not require address decoding, as the instruction set represents the addressed slave. A master could execute commands to both slaves as shown in line 3-8.

The problem with this method is that it gets inefficient on RTL level, as the `Slaves` datatype can get relatively large. Moreover, this design has a structure of a single core processor. The master is destined to act as a director within slaves, which will likely cause bottlenecks in larger use cases. Moreover, multiple instances of a slave require yet another operation code in the `Slaves` type. In conclusion, it is not scalable.

```
1  module Led where
2  import Clash.Prelude
3
4  type State = BitVector 4
5  data OpCode = Write State | Read | OpIdle
6  data RspCode = RspWriteOK | RspRead State | RspIdle
7
8  slaveT :: State -> OpCode -> (RspCode,State)
9  slaveT s i = (rsp,s')
10     where
11     (rsp,s') = case i of
12         (Write x)   -> (RspWriteOK,x)
13         (Read)      -> (RspRead s,s)
14         (OpIdle)      -> (RspIdle,s)
```

Listing 7: A algebraic slave that controls 4 LED's

```
1  data Slaves = Led Led.OpCode | Rgb Rgb.OpCode
2
3  prog :: [Slaves]
4  prog = [
5      Led (Led.Write 0b1000),
6      Led (Rgb.Write 0b010000),
7      Led Led.Read
8      ]
```

Listing 8: Master code that can communicate with the slaves by using algebraic commands

Chapter 5

Hardware and data flow

This chapter presents a method to derive an hardware architecture from a subset of synchronous data flow. The intention is to present the data flow fundamentals and explain why data flow can be useful in real-time applications. Subsequently, one of the studies of this thesis is presented. It describes how hardware could be derived from data flow.

5.1 Dataflow

The book that is used as part of the Real-Time Systems 2 (RTS2) course [8] on the University of Twente explains the fundamentals of Dataflow with full comprehension. This section only explains the part of the theory that is required to understand the material in this thesis.

Multi-core programming is often described as a difficult programming challenge. It is hard to reason about the utilization of parallelism while designing correct functional behaviour. The RTS2 course presents a model for multi-core systems and the tools to analyze these systems. Multi-core does not only refer to multi-core CPU's, as it can also point to multi-accelerator systems, as often used in FPGA systems. The analysis model is called data flow and is explained in the next paragraph.

Dataflow is a set of models that can be used to describe real-time behaviour. In this thesis Homogeneous Synchronous Data Flow (HSDF)and Multi-rate Synchronous Data Flow (SDF) are considered. HSDF consists of the elements that are shown in Figure 5.1. A snippet from [8] gives the following definition of HSDF: *A Homogeneous Synchronous Dataflow (HSDF) is defined as a directed graph $G(V, E)$ that consists of actors $v \subseteq V$ and directed edges with $e \subseteq E$ with $e = (v_i, v_j)$. The edges represent First-in-first-out queues that have unbounded storage capacity. In the queues indivisible tokens can be stored.* Homogeneous Synchronous data flow was originally introduced in [9].

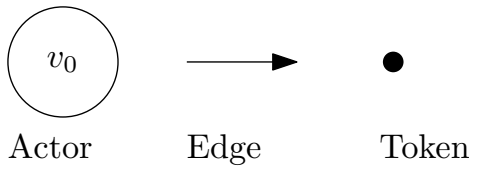


Figure 5.1: Elements of data flow

An actor is a node without state that can represent a task, such like a mathematical function. A task that is represented by an actor is executed when the actor fires. An actor within a HSDF graph must comply to a so called firing rule: one token must be present on each incoming edge of an actor. In multi-rate graphs, this can be more than one token as well. After an actor has fired, it produces a token (or multiple tokens in multi-rate) on all it's outgoing edges. The time between the consumption of tokens on the incoming edges, and the production of tokens on the outgoing edges is called the firing duration. A token is a indivisable element, which means it can not be party consumed or produced. Due to the abstraction of the model, a token can present data, space or synchronization events. Tokens are distributed among actors by means of queue that is represented by an edge. Queues can hold an unbounded amount of tokens. Furthermore, tokens are consumed from a queue in the order that they are produced.

A graph consisting of the elements described above is called a HSDF graph. Because of the semantics, these graphs can by analyzed on throughput and latency. Another propery of data flow is that it is deterministic, which makes it useful for real-time applications. The designer must find a data flow model for the application that is designed. A correct model allows to be examined thorougly on utilization, throughput and latency. The analysis techniques themselves are not part of the scope of this thesis. Some examples of HSDF graphs are shown in Figure 5.2. An edge from an actor to itself is called a self edge. A self-edge forces an actor to fire non-concurrent. The graph on the left in Figure 5.2 does not feature a self edge at actor A, and the result is that it consumes and fires tokens concurently as shown in the schedule below the graph. The centered graph features self-edges on both edges, which prevents the concurrent firing. The graph on the right cannot fire concurrently due to insufficient tokens. In this thesis the self edges are implicitly added to the actors.

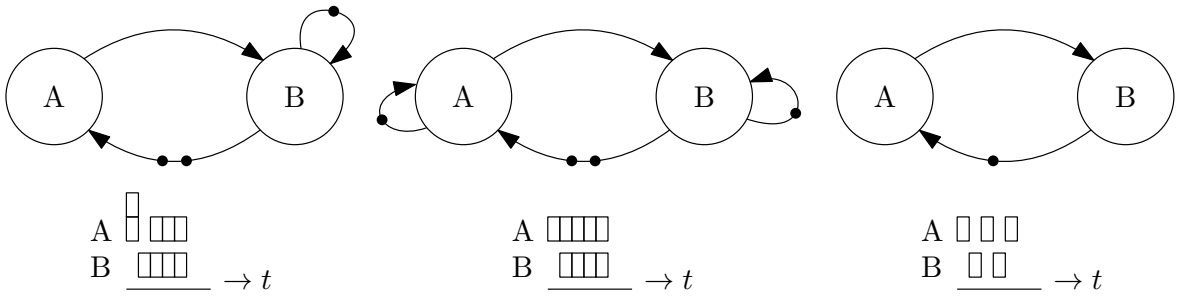


Figure 5.2: HSDF examples, all actors have an execution time of one time unit

Another variation of synchronous data flow is (Multi-Rate) Synchronous Data Flow ((MR)SDF) [10] and is often called SDF. An rudimentary SDF graph is depicted in Figure 5.3. As the name suggests, it support multiple consumptions or production of tokens in comparison with HSDF. The cited literature gives a more comprehensive definition, this explanation serves only as a quick overview.

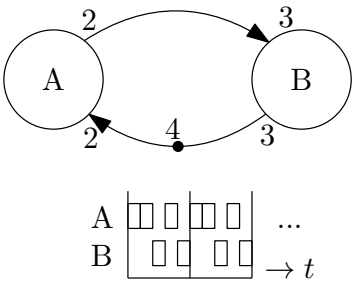


Figure 5.3: SDF example, every actor has an execution time of one time unit and implicit self edges.

One other aspect that is concerned is the consistency of graphs and the repetition vector. The full theory on repetition vectors has been elaborated in Chapter 5 of [8] and proven in [10]. An SDF graph is called consistent if the production and consumption of tokens over time is balanced. An inconsistent graph causes the amount of tokens to increase towards infinity (unlimited buffers, not feasible), or decrease to zero (deadlock, no actor can fire). The consistency can be determined with the topology matrix that is shown in (5.1). The columns in the matrix represent in and outcoming edges the actors in a graph and the rows represent the edges that are present in the graph.. A consumption of an edge e_k is labeled by γ_k and a production is labeled by π_k, see Figure 5.4.

$$\Psi = \begin{bmatrix} \pi_0 & -\gamma_0 \\ -\gamma_1 & \pi_1 \end{bmatrix} \tag{5.1}$$

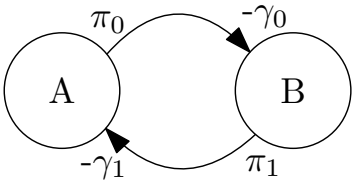


Figure 5.4: Graph that is represented by (5.1)

The topology matrix of the graph in Figure 5.3 is represented in (5.2). The repetition vector is the smallest integer vector z such that $\Psi\vec{z} = \vec{0}$. If the graph is connected and consistent and after each actor v_i has fired z_i times, then the repetition vector indicates the number of firings required of each actor, that result in the initial token state on the edges. For the matrix shown in (5.2) the repetition vector is $\vec{z} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$. This implies that A fires 3 times and B fires 2 times in

order to return to the initial token distribution.

$$\Psi = \begin{bmatrix} 2 & -3 \\ -2 & 3 \end{bmatrix} \tag{5.2}$$

A data flow graph can fire either self-timed (fire as soon as the firing rules are obeyed) or according to a schedule (the firing rules are still obeyed, but the enabling time of an actor can vary). A graph where the actors eventually fire according to a reoccuring pattern is called periodic. Actors can also fire according to a strictly periodic schedule. A strictly periodic schedule has as additional constraint that each of the actors in a graph fire in a periodic schedule themselves. This can be clarified by the example that is shown in Figure 5.5. The lower schedule is periodic. The firing pattern repeats every three time units ($A \rightarrow A \rightarrow B$). Therefore, the scheduled is called periodic. A schedule is strictly periodic if every actor in the schedule fires periodic with respect to itself or $s(k) = s_i(0) + kp_j$. In the periodic schedule, the time between firings with actor A varies between 1 and 2 time units. Therefore the actor does not meet the requirements for a periodic actor. The time between firing for actor B is always 2 time units, and therefore this actor is periodic.

A schedule where all individual actors fire periodic, and thus the schedule of the graph is strictly periodic, is shown in the upper part of Figure 5.5. It can be observed that the utilization of the actors is lower in the strictly periodic schedule.

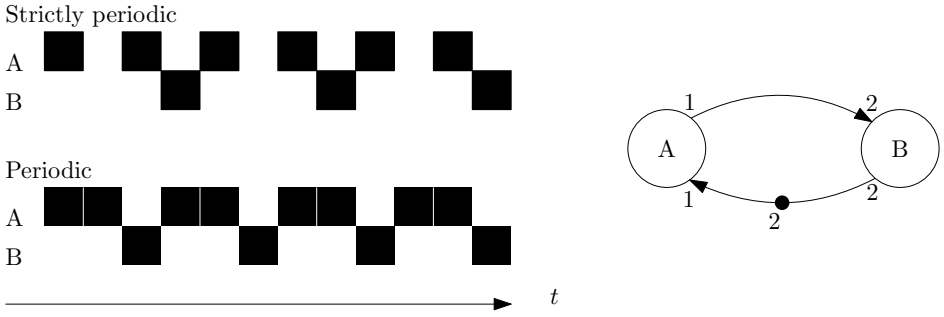


Figure 5.5: Difference between periodic and strictly periodic, implicit self edges and an execution time of one time unit for both actors are assumed

A method to efficiently derive the strictly periodic schedule of a given valid data flow graph is described in [11] and implemented in Haskell in [12].

5.2 Relation between data flow and hardware

Throughout the last decades there has been an increasing demand for high performance and low power systems. As Moore's Law seems at its end, optimalizations through hardware design seem like the solution to this problem. However, systems get more complicated while the development time shortens. [13]

As discussed in the introduction, defining hardware in functional languages can shorten the process from algorithm to hardware, as the definition of the algorithm and mapping the algorithm on hardware can be separated. The mapping flexibility that Cλash features can be utilized to optimize an architecture for throughput, latency or power consumption in a later stage of the design process.

In Section 5.1 it was discussed that although the semantics of data flow are limited, it is a powerful tool to analyze systems on latency and throughput, and could therefore be suitable for real-time hardware design. One of the major limitations of data flow is that there is no choice element. Many existent hardware designs use decision making. It therefore is difficult to derive data flow from hardware that was designed in a conventional manner. An approach is to use abstractions in data flow. E.g. If an actor has an execution time of either 1 or 5 time-units, the actor can be modeled with an execution time of 5. This is pessimistic, but it assures that it can still be analyzed with data flow tools. However, this is just a trivial example where one actor is considered. if an existing hardware architecture results in a fairly large network of actors with variable execution time, abstraction gets more difficult and results in a less accurate representation of the system that is analyzed. A type of data flow graphs that covers varying execution times is called Cyclo-Static Data Flow (CSDF) [14].

The problem can also be approached the other way arround. In other words, hardware may also be derived from data flow instead. A basic data flow graph of the HSDF type is depicted in Figure 5.6.

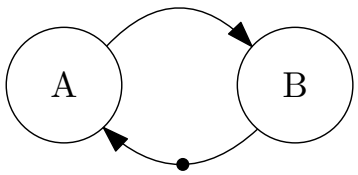


Figure 5.6: A basic HSDF graph with two actors, self edges are implicit and the execution time of both A and B is one time unit

Both actors A and B have implicit self edges and a firing duration of one time unit. From the graph, it can be deduced that A and B fire mutually exclusive, because there is only one token that is passed between both actors. This is only an abstract interpretation that requires a definition about what tokens and actors represent, before it can be expressed in hardware. Actors are defined as blocks that compute an arbitrary function. A function is a box that consumes one or multiple inputs and produces one or multiple outputs. A function can (among other functions) be part of a function in a higher hierarchy. It is therefore a neccesity that functions can communicate their outputs to the inputs of other functions. This holds also for function within hardware. The characterization of tokens is less apparent than that of actors. Assume Figure 5.6 holds for a particular system where actor B depends on the data from actor A. How can this property be extracted from the figure? If tokens would represent data, then it it implies that A depends on B and vice versa. Therefore tokens can

not represent a dependency of B on A. Another way to look at tokens is as an synchronization mechanism on the schedule where A and B comply to. E.g. If Figure 5.6 has an additional token on the edge from A to B, then both actors could fire simultaneously and the throughput would be different. However, the graph still holds no information about the dependencies on data between both actors. To resolve this problem, an additional graph as shown in Figure 5.7 can be introduced.

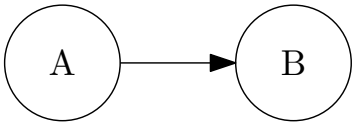


Figure 5.7: A graph that depicts that B has a dependency on A

The graph is called a data dependency graph and it consists of the actors that are present in an accompanying data flow graph, along with edges that only indicate data dependency between the actors. This concludes the system in which A and B are present: Both the scheduling and data relation are now derived.

From the information of both graphs a hardware architecture can be derived. But first the definition of tokens should be elaborated more. Another data flow schedule for which the data dependency in Figure 5.7 still holds, is shown in Figure 5.8. According to the graph, actor A fires N times as often as actor B. Therefore, the amount of tokens on the edge B to A represents a First-In First-Out (FIFO) data buffer with size N, between A and B. The definition of data flow states that edges represent an unbounded token capacity, but this is not feasible in a real design. From the distribution of tokens on the edges, it can be derived what data buffer sizes are required.

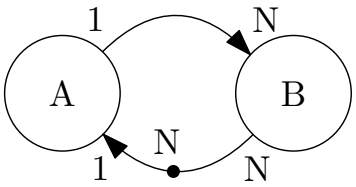


Figure 5.8: Another SDF graph. self edges are implicit and the execution time of both A and B is one time unit

In the aforementioned graph, A fires N times, then B fires once, after which the schedule repeats. The designer could improve the throughput of the graph by increasing the buffer from N to $N + 1$. This adjustment causes that A does not have to wait for B to complete in order to fire again, hence improves the throughput. The designer can also use SDF analysis tools to detect deadlock occurence or other undesired behaviour, if the schedule is known.

Another advantage of the data flow with data dependency methodology is that a system that is derived from a feasable data flow graph does not require back pressure between the actors, as long the actors are modeled correctly. Back pressure is the process where a node

that receives data and confirms to the producer that the data is indeed received. Since the firing times in a graph are known, as well as the relation of firings between actors, there is no neccesity to communicate this kind of status messages. In the Design Space Exploration that is presented in Chapter 6 the data flow and data dependency design methodology is compared to other alternatives.

Chapter 6

Design Space Exploration

The DSE is a systematic method to characterize alternatives to criteria. In this chapter the alternatives that are proposed in the background and characterized to the criteria that are proposed in the introduction. In summary, the communication standards must comply to the following requirements:

- The development language is fixed to the Cλash-language;
- The resulting structure should be deterministic.

The next paragraphs evaluate the three approaches that could be elaborated further during this research. The chapter concludes with a comparison of the presented approaches, and concludes with the approach that is realized.

A conventional SoC standard
One of the major advantages of using an existing SoC standard is that the implementation is just a matter of following the specification. Another advantage is that there are many IP's that use these standards. Hence one can use existing IP's and reduce the time until there is a functional platform. However, it is not important to create the robot platform as soon as possible, since it has no commercial use. Another aspect is that all existing protocols operate on bit-level, simply because this is convenient in conventional languages. The Cλash-language supports higher order functions, which may enable a fundamentally different design approach. Another downside of conventional standards is that they are designed for systems that are not neccesarily set up to be deterministic. Dataflow requires a static schedule in order to be fully deterministic. A conventional communication standard can also be deterministic if a static schedule is used, but does not offer advantages over the strict data flow methodology.

Communication using algebraic data types
Both Cλash and Haskell use a (customizable) data type mechanism. The proposed idea discussed in Chapter 4 is to design IP's with its own instruction set. Each IP can be controlled according to their own instruction set, instead of utilizing the conventional memory mapped approach to communicate. A downside of this approach is that it results in a non-standardized way to communicate with IPs. It looks clear in the code as instructions could

look like `RotateMotor 1 200` (Rotate motor one 200 degrees). However, tests with this methodology result in inefficient structures on RTL level since there can not be a bus if every IP can use a different datatype to transfer data or instructions. It is not scalable to larger designs as well.

Data dependency & data flow approach
The approach presented in Chapter 5 with data dependency and data flow has the advantage that the hardware can systematically be derived from both the data flow (scheduling) and the data dependency (data). It also enables the hardware architect, to adjust the schedule at compile time (e.g. pre-evaluation in Haskell) in order to improve performance in different areas. It will therefore also use the properties of Cλash and Haskell. Moreover, it is a fundamental different approach which is interesting for the CAES group. The downside is that the standard is non-existing and must be carefully defined. It will therefore likely take longer to create a functioning prototype.

6.1 Comparison and conclusion

The three proposed methods are evaluated on the constraints that were elaborated in the introduction of this chapter, and rewarded accordingly. The rewards are ranked on an ordinal scale: $- < 0 < +$. The weights on each constraint are equal. Therefore the total score is the sum of all the constraints. The highest scoring approach is elaborated as part of this thesis.

The first constraint is the utilization of Cλash. Conventional SoC protocols are already being utilized within conventional HDLs. During this project a Wishbone interface for a master and slave were designed in Cλash as a test case. Besides the functional aspect of Cλash, there is no challenge and research value in designing conventional standards. Moreover, existing standards are not designed to be deterministic. Algebraic methods already utilize Cλash more, since custom instructions are just different datatypes. Still, communication is similar to conventional methods. The data dependency method also fits the profile of Cλash. Especially the pre-evaluation in Haskell is interesting because it can generate the appropiate hardware at compile time.

As for analysis possibilities, performance can be evaluated on conventional standards. As stated earlier, the freedom that these standards offer on the field of configurability and flexibility, can result in a non-deterministic system. The same conclusion can be applied to the Algebraic type communication proposition. The data dependency and data flow approach uses data flow to derive a guaranteed schedule, that is fully deterministic. It can also be evaluated with data flow analysis tools. It offers reconfigurability as well. The throughput and latency as well as the hardware utilization can be adjusted by changing the schedule.

Finally, the project value of the proposals are compared. The project value is rated according to the estimated time that it will take to create a first prototype. A conventional SoC standard has the advantage that is has already been used for larger designs, and therefore it is more likely to result in a functioning platform. The algebraic type communication will likely result in many different instruction sets, and therefore different data types. This will likely result in many point to point connections, which will increase power consumption and decrease the overview of the design. The data dependency and data flow approach has potential, since the project is still in early and adaptable state. It introduces an entirely new approach to design digital systems. However, it requires a more implementation, as no existing standard is used and it is not assured to be feasible for larger systems.

The results are summarized in Table 6.1. From the Table it can be concluded that the data dependency and data flow method is the most interesting to elaborate for this Master thesis.

Table 6.1: Results of the Design Space Exploration

	Utilizing Cλash	**Analysis possbilities**	**Project value**	**Total score**
Conventional SoC standard	-	0	+	0
Algebraic type communication	+	0	-	0
Data dependency and data flow	+	+	-	+

Part II

Realisation

Chapter 7

Dataflow to Hardware

This chapter presents a new approach to derive a hardware architecture from data flow. It builds further on the concepts that were introduced in Chapter 5.

The definition of data flow is explained in [8] and in some detail in Chapter 5 of this thesis. A data flow graph is defined as: *a directed graph $G(V, E)$ that consists of actors $v \subseteq V$ and directed edges with $e \subseteq E$ with $e = (v_i, v_j)$. The edges represent First-in-first-out queues that have unbounded storage capacity.*

First, this chapter elaborates the definition of data dependency graphs that was presented in Chapter 5. A data dependency graph is a directed graph $DG(V, E)$ with actors $v \subseteq V$ and directed edges $de \subseteq DE$ where $de = (v_i, v_j)$. An edge $de = (v_i, v_j)$ implies that v_j depends on data from v_i. The set of actors that is present in the data dependency graph, equals the set of actors that is present in the data flow graph.

The remainder of this chapter describes how a hardware architecture can be derived from a data flow graph and data dependency graph. The aim is to explain the methodology in this chapter, and to provide overview and clarity to the reader. Chapter 8 concerns the process of solvings problems that arise with the implementation of the chosen methodology.

The aim is to generate a hardware architecture from a given set of functions, a data dependency graph and a data flow graph. The set of functions is represented as actors. The behaviour of the generated hardware complies with the scheduling within the data flow graph. The following intermediate goals are deduced:

- The data flow schedule graph must be evaluated to acquire information on whether functions (actors) should be executed (fired). The tokens in a graph function as synchronization mechanism;
- The syntax of a function (actor) in hardware must be specified;
- Functions that depend on data from other functions should be connected to to the functions that provide those dependencies. There should be a generic interface to

connect the functions to.

The immediate goals are elaborated in Section 7.2, 7.3 and 7.4. However, first a top-level overview of the proposed methodology is presented in Section 7.1. This Section serves to sketch the bigger picture of the presented approach.

7.1 Overview

This section presents the generic hardware architecture that can be used to construct a system for a set of functions, a given data flow graph and a data dependency graph. Figure 7.1 depicts an example system that uses a generic hardware architecture. The actors A, B, C and D in the data dependency and data flow graph represent the timing behaviour of the functions A, B, C and D in the hardware architecture on the right side of the Figure, respectively. The distribution of tokens in the data flow graph, serves as synchronization mechanism for the firing of the actors (and thus for the execution of the functions). Within data flow there is no clock domain, whereas in hardware there is. Within this thesis one time unit in data flow corresponds with one cycle of the clock, measured at the rising edge. The firing schedule to which the actors comply is saved in the Scheduler, that is shown in the hardware architecture. The Scheduler signals the functions A, B, C and D to fire. The derivation of the schedule is discussed in Section 7.2. The constraints on a function must are discussed in Section 7.3. The data dependency graph depicts the relation on data that functions have. In this example, function D relies on the data of actor A, B and C. A generic interface that enables the functions to pass data to each other is the CrossBar component in the hardware architecture. The motivation for a crossbar interface is explained and further elaborated in Section 7.4.

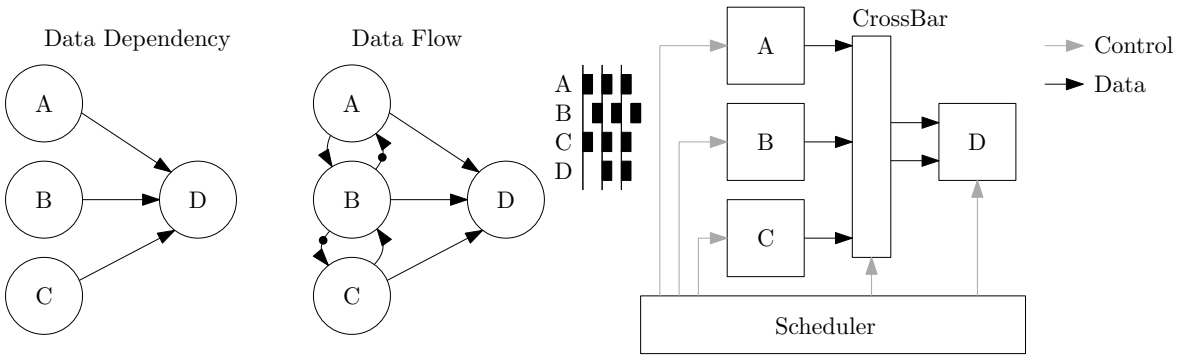


Figure 7.1: Example with interconnect

7.2 Derivation of a schedule

Chapter 5 discussed some techniques to derive a schedule, given a data flow graph. A Haskell tool to derive a strictly periodic schedule from data flow is being developed at the University of Twente [12]. Given a consistent data flow graph, the tool returns the list of actors

along with their period and their relative start time. The tool will only generate schedules for graphs that allow to be scheduled strictly periodic, which may limit the type of graphs that can be used. This was not considered as a problem, since it does not restrict the further process of the project and saved design time. In future work, this part of the thesis can be elaborated to increase the support.

7.3 Function block specification

The constraints concerning the execution of a function block can be partly derived from the scheduler, because the scheduler signals to the function to execute. Therefore, the implementation of the function must be able to work with the enable signals that the scheduler provides. Therefore, a function block should have at least an enable input. An actor (function block) with an execution time of one time unit (one clock cycle) should produce a valid output on the rising edge of the enable signal. The designer can also create a function block that may take more clock cycles to compute it's output after an asserted enable signal. In that case the fire duration of the corresponding actor has to be adjusted accordingly in the data flow graph.

7.4 Connecting data dependencies

Connecting data dependencies consists of connecting the output of all producing functions to the input all of consuming functions. However, doing this for all actors can result in a fairly large web of connection wires. According to Chapter 3, interconnects are often used in existing protocols to tackle these problems, see Figure 7.2. Interconnects exist in many forms but in essence they connect inputs to outputs as a result of the control signal it receives from an arbiter.

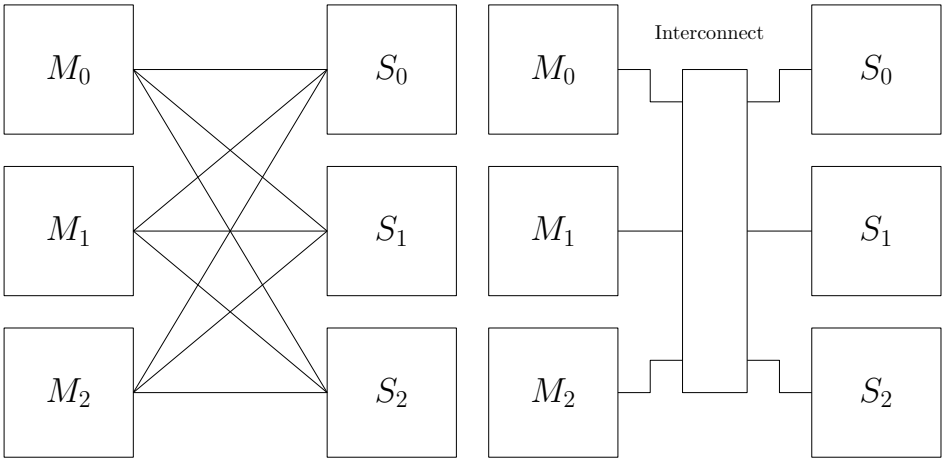


Figure 7.2: Direct connection vs an interconnect

A crossbar is a versatile interconnect. In contrast to a shared bus which was discussed in Chapter 3, all the inputs can be mapped to one or more of the outputs. It is therefore a very

generic interconnect that can be configured in many manners, and its function within the derivation of an hardware architecture can be explained by means of the earlier presented example in Figure 7.1. The center part depicts the scheduler of the actors. Note that the scheduler also controls the setting of the crossbar. The actors are funtional blocks with a firing duration of one time unit, or one clock cycle from hardware perspective. The schedule is shown in the upper right part of the Figure. D depends on A, B and C. The outputs of A, B and C are therefore connected to the input of the crossbar, whereas D is connected to the output of the crossbar. The schedule is such that A and C fire out of phase with B. Therefore D only requires two simultaneous inputs, while the function requires three inputs. Buffering is abstracted out of this example. The crossbar minimizes the amount of wires based on the schedule. It also offers a generic interface to which actors can be connected. The versatality of the crossbar can be elaborated further by means of another example that is shown in Figure 7.3.

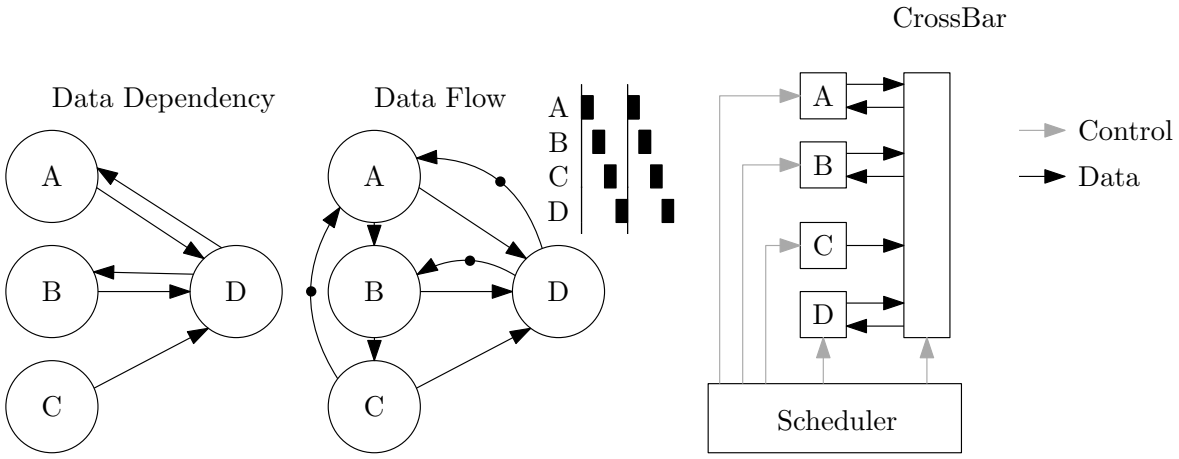


Figure 7.3: Second example with intertwined dependency graph

In this example there are more dependencies, which is why the structure is slighty different from the previous example. Note that a function can be a consumer and producer at the same time. Furthermore, function D has only one input, since all it's dependencies can be scheduled at different times. Again, if A, B and C are rescheduled such that they fire simultaneous, then the crossbar and the scheduler would look different.

Chapter 8

Results

This chapter concerns the results that followed from the elaboration of the proposed approach, that was described in Chapter 7. The result of this thesis is a tool that generates a hardware architecture based on the data flow and data dependency graph that are supplied by the designer. The chapter describes the main building blocks that are used to construct the tool. The aim is to describe the presented blocks on behavioral level, such that it is understandable for readers who are not familiar with CλaSH or Haskell. The accompanying CλaSH code of each block is attached in Appendix C and D. The blocks are represented as libraries that are static and do not require adjustments in order to be used in a design.

- Analysis libraries (Section 8.1)
  - Data dependency (Section 8.1.1, code in Appendix C.1)
  - Crossbar minimalization (Section 8.1.2, code in Appendix C.1)
- Hardware generation libraries (Section 8.2)
  - Crossbar (Section 8.2.1, code in Appendix D.2)
  - Scheduler (Section 8.2.2, code in Appendix D.3)
  - Actor buffering (Section 8.2.3, code in Appendix D.1)

The libaries that are presented are used to derive a system top entity. The configuration of such a system requires additional work and configuration, as well. The templates of the configurations are listed in Appendix E.1 and E.2. Two examples on the creation of a configuration are elaborated in Appendix F and referenced in 8.3. In these examples it was not possible to neglect the CλaSH-language entirely, since it serves as a step-by-step guide on how to connect the components together. The examples also involve an examination of the resulting RTL.

8.1 Analysis

This section explains the parts of the library that are pre-evaluated in Haskell, and where only the results of the expressions are compiled to hardware. The results are used to generate

the desired hardware structure in Section 8.2.

8.1.1 Data dependency

The data dependency graph is created by means of a custom data type within proposed library. A graph consists of actors and dependency edges. Actors have three fields: name, period and start time. An edge has two fields namely, the name of the producer and the name of the consumer. A data dependency graph *with scheduling information* is generated by the function `genDDgraph`. The function takes a valid data flow graph that is constructed in the data flow schedule generation tool [12], and a list that consists of the data dependency edges that are part of the data dependency graph.

8.1.2 Crossbar minimalisation

The crossbar minimalistion is added to bundle the data of actors that fire non-simultaneously, as it saves hardware. A performance evaluation is elaborated in the example that is presented in Section 8.3. Solving the minimization problem is attempted in two phases. First, it evaluates how the data of multiple producers can be bundled for one consumer. The next step is, to combine this process for all consumers within a graph.

Determining producer combinations for one consumer
For a data dependency graph with one consumer and one or more producers, it must be examined how the crossbar can be configured efficiently (largest reduction of wires to the consumer). The derivation depends on the schedule that is supplied in the data flow graph.

The algorithm to derive the most efficient structure is written in Haskell, but the result is lifted (pre-evaluated at compile time) so that the Cλash-compiler reads it as a constant when the hardware is derived. The function has the consuming actor and the corresponding data dependency graph as argument. What it does first is finding all producing actors for the consuming actor.

Then it simulates the timing behaviour of all the actors for a determined time frame. This time frame needs to be large enough, such that it can be ensured that the actors will never fire simultaneously. The current implementation of the library first takes the least common multiple (LCM) of the actor firing periods.[1] The LCM on itself is not sufficient, since actors also have start up times, which may cause that they do not run for a larger part of the simulated schedule. Therefore the schedule needs to be evaluated at least until there has been a period where all actors were booted at the start, see Figure 8.1.

[1] It could also be the repetition factor times the period of the actor when SDF graphs are supported. The finalization of SDF support is considered future work. Therefore, the repetition vector is not considered in the current implementation.

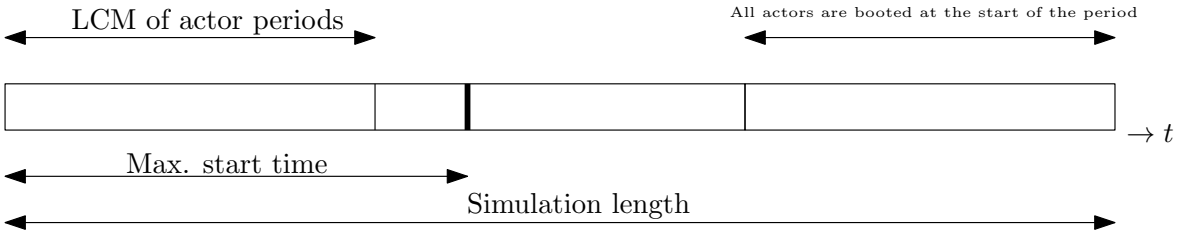


Figure 8.1: Indication of the minimum simulation period

From the lists of execution times, it can be determined which actors have overlapping execution times and which do not. From a list with n producing actors, m combinations of actors need to be found. m will represent the number of ports on the crossbar, and should therefore be minimized. The constraint in generating combinations, is that the actors that are combined do not have overlapping firing times. A method to minimize m is to find the largest combination possible by starting with a combination size p that is equal to the number of producing actors. If this does not result in a valid result, then do it for $p-1$ on all permutations of p. When a combination is found, the resulting combination is removed from the remaining list of actors to search in, and put in the list with results. Then the whole process is repeated until the remainder is empty. The length of the list with results, is the number of ports required to schedule the producers for the consumer. The algorithm described above is implemented in Haskell code, and the results are used in generating the scheduler for the crossbar.

Determining producer combinations for all consumers
Often a data dependency graph consists of multiple consumers. Thus executing the algorithm above for every actor that is consuming, and merging the resulting scheduling control vectors results in one crossbar, that has ports for the inputs and outputs of the functions in the data dependency and data flow graph.

8.2 Hardware generation

This description elaborates the design decisions that were performed on the Cλash implementation with respect to the generation of hardware.

8.2.1 CrossBar

This Section describes the generation of the crossbar on hardware level. An architecture of an M rows by N columns crossBar is shown in Figure 8.2 for $M = N = 3$. The main property is that every one of the inputs $n \subseteq N$ can be directed to $m \subseteq M$. This can be accomplished by using $M * (N-1)$ multiplexers. Thus for every m, there are $N-1$ control bits required.

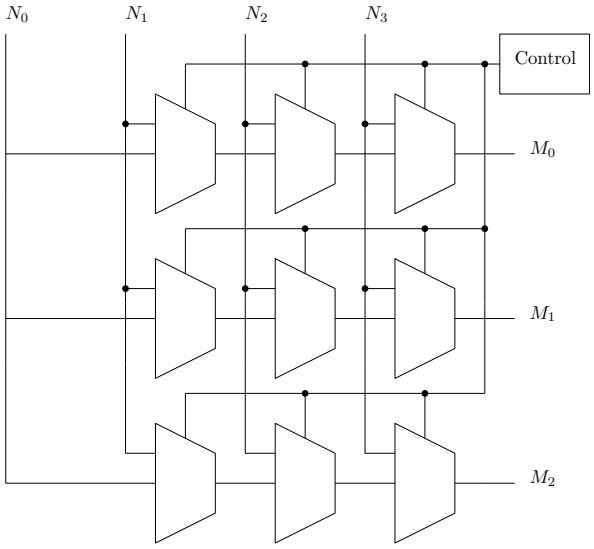


Figure 8.2: M x N crossbar

However, some trivial states can be eliminated. E.g. if N_3 is directed to M_0, then there is no explicit state required for the muxes at the other inputs of N that can direct to M_0. The control vector can therefore be expressed more efficiently. The desired behaviour is shown in Table 8.1 and can be modeled in hardware by Figure 8.3. It converts a control vector of length $\lceil log_2 n \rceil$ bits to $n-1$ bits, that control the state of the muxes.

Table 8.1: Truth table for control logic to set 3 multiplexers connected to one output of M

input	control output	actor output
00	111	N_0
01	011	N_1
10	001	N_2
11	000	N_3

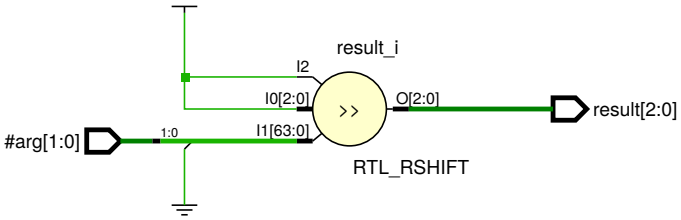


Figure 8.3: Multiplexer control logic that is equivalent the truth table in 8.1

8.2.2 Scheduler

The data flow graph contain all the required information about the scheduling of the actors. The scheduling data is evaluated in haskell, to generate a vector that features enable signals for all individual actors over time. This vector is included in a Asynchronous Read-Only

Memory (`AsyncROM`) in CλaSh, along with a program pointer that is increased over time to point to the correct control vector. Once the end of the schedule is reached, the program pointer needs to be reset to the point in the schedule. Figure 8.4 depicts how this point is derived. The point is set to the start of the first period in the schedule where all the actors are started. From here, the program counter is again increased to the end of the schedule, after it is reset again.

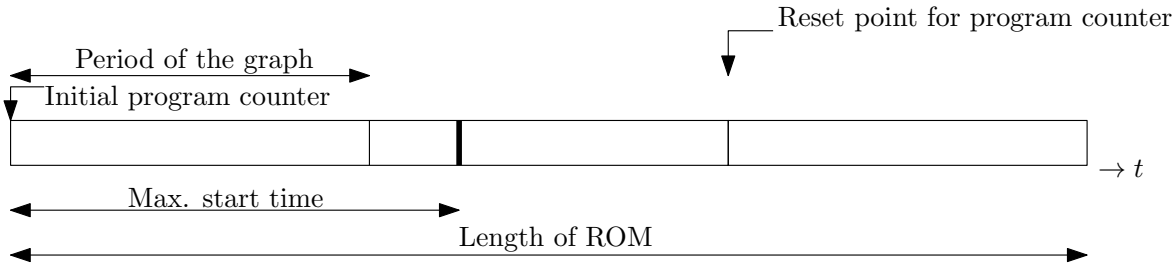


Figure 8.4: Illustration about the derivation of the Program counter

8.2.3 Actor buffering

In the previous examples the buffering was left out. However, when a function receives input arguments non-simultaneously, a buffering system must be characterized. It is a possibility to use a simplified circular FIFO buffer as shown in Figure 8.5. The FIFO is simplified because it does not feature a full or empty signal. These indicators are not required because the buffer can be modeled into the data flow graph. The FIFO depicted in the Figure has a read pointer and a write pointer, and a size that is defined as N data blocks. dIn and $dOut$ are ports where data words of predefined widths can be written or read, respectively. At every instance of writing or reading, the corresponding pointers are increased and wrapped in the range that the buffer can hold.

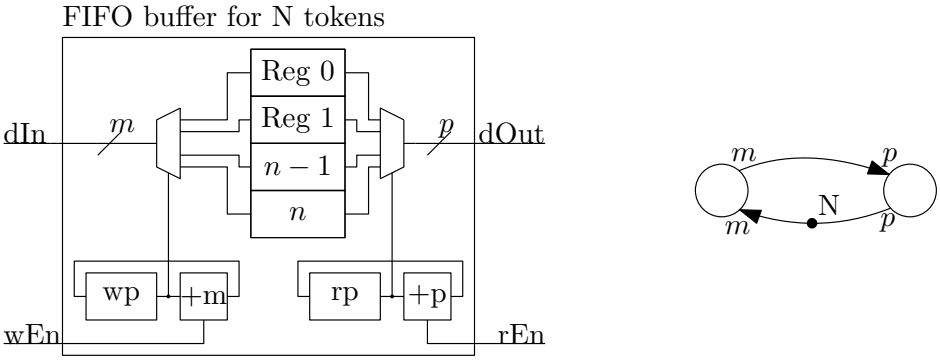


Figure 8.5: A FIFO structure

The aforementioned buffer methodology holds only if a consuming actor only depends on data from one producing actor. However, if it requires data from more actors, then besides more buffers there is more logic required. In Figure 8.6 an example that depicts such a problem. D depends on A,B and C, A and C fire simultaneously and B fires out of phase. The

crossbar is optimized to two wires, since the presented schedule allows this. However, D is still a function that requires three arguments. The outgoing edges on the crossbar show what data can be held during execution: one holds either A or B and the other holds C or nothing. Actor D requires 3 buffers, for A, B and C. The data on the outgoing edges of the crossbar can therefore be multiplexed to the appropiate buffer, by means of a demultiplexer. There is a relation between the state of the crossbar, and the direction in which data should be directed within the buffer, and therefore the same control signals can be used.

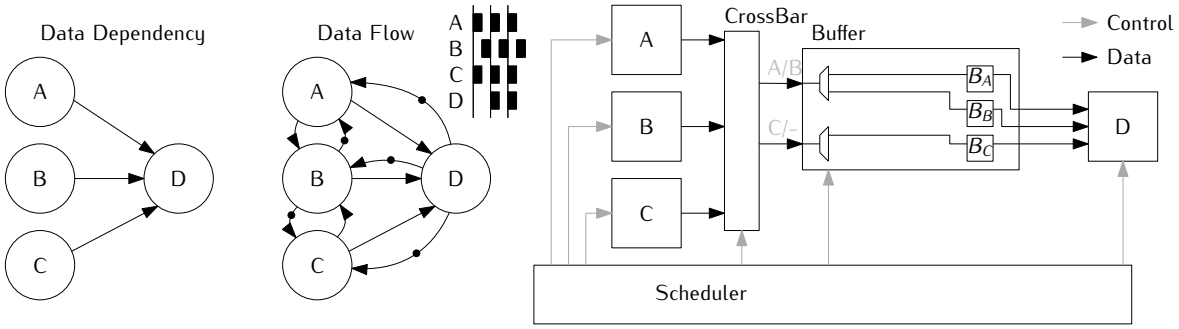


Figure 8.6: A FIFO structure

The edges δ_{da}, δ_{db} and δ_{dc} that are present in the data flow graph in Figure 8.6 all hold one token. The tokens indicate the buffer size that is utilized between D and A, B and C, respectively. In fact, there must always exist a simple cycle (with a token indicating the buffersize) between the actors that are involved in a data dependency edge.

The designer should also be aware that not all the data flow principles can be applied on a data flow graph. Figure 8.7 depicts an example with two data flow graphs that have the same sumber of tokens, but with a different initial distribution. The data flow graph on the left side is correct, as it cleary indicates that there is a buffer with the size of two datablocks between A and B. On the other hand, the graph on the right indicates that A and B may fire at the same time, and both have a buffer between them with the size of one. Although it is a valid data flow graph, it contradicts with the dependency graph. B depends on A, but it can immediately fire, without receiving data from A. It is therefore unsupported to generate hardware from the graph on the right.

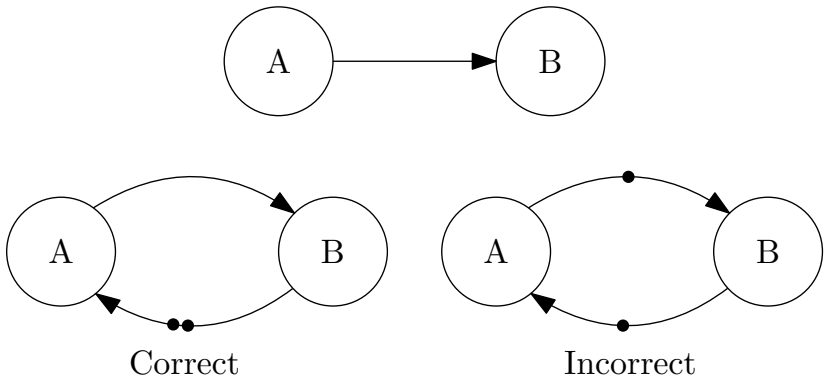


Figure 8.7: Token distribution

The process of initiating a buffer to an actor is described in the derivation of the example system in Section 8.3.

8.3 System derivation

Ideally, one would generate the hardware that complies to the data flow graph and data dependency graph automatically. Automatic generation was also stated as one of the requirements within this project. However, it is complicated to fully generate it, due to the semantics of the Cλash-language. The blocks that are discussed in the previous sections are functions within Cλash. Rudimentarily speaking, functions take inputs and produce outputs. However, the generation of a system as discussed in the methodology is a combination of intertwined functions, and not simply a composition.

In the future, code generation might be the best solution to tackle this problem. One can generate the design blocks, and then run a utility that connects these to each other. For now, the only method is to connect the blocks by hand. The tool suggests the connections that the user has to make in order to make it work. This thesis provides two step by step tutorials on how to create a system. The first example is elaborated in Section 8.3.1 and the second example is elaborated in Appendix F. The results are summarized in the conclusion of this thesis.

8.3.1 Example: Sorter of Random numbers

This section describes the configuration of an example system. First the design is worked out step by step for a specified token configuration. Then the example describes two alternations in the amount of tokens to change the temporal behaviour of the system.

Figure 8.8 depicts an example where there are six pseudo random number generators (A to F) that feed data through two blocks: G (sort the numbers from large to small) and H (represent the numbers unsorted).

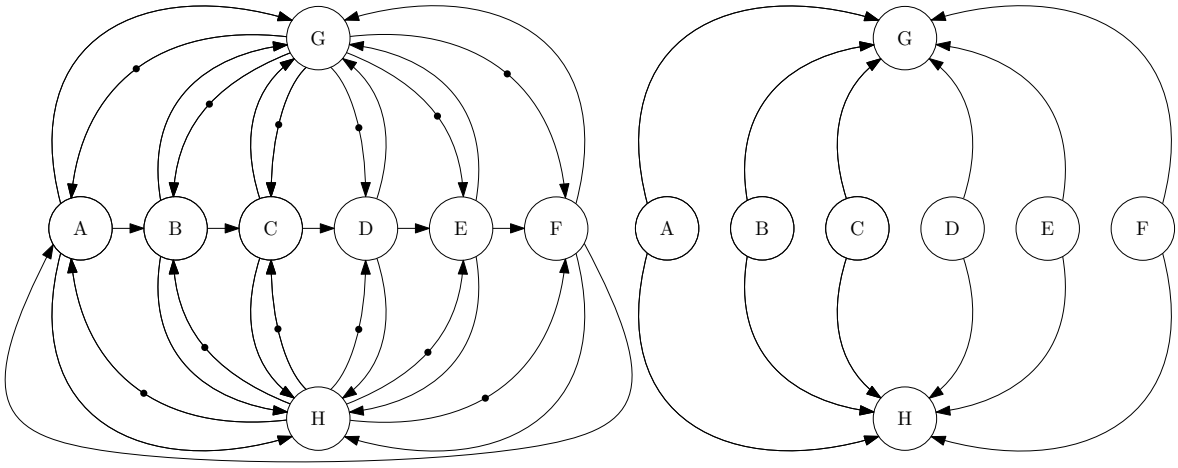


Figure 8.8: Sorter of random numbers, left the dataflow graph and right the data dependency graph

First the function that the actors represent must be represented in CλAsh. The random number generator can be described by Listing 9 and the sort function can be described by Listing 10. The sort function will be attached to a buffer in a later stage. Therefore it does not contain the enable signal that activates the function yet. The random number generator is based on a Linear Feedback Shift Register (LFSR) and does not require any buffers because it is only a producer. Therefore, it already contains the enable signal.

```
1  lfsrT
2    :: forall n .  KnownNat n
3    =>  BitVector (n+1) -> Unsigned (n+1)
4    -> Bool
5    -> (Unsigned (n+1), Unsigned (n+1))
6  lfsrT seed s en = (s',o)
7    where
8      mask = map (.&. (lsb s)) (bv2v seed)
9      o = s
10     s' = if en then
11             unpack . v2bv $ zipWith xor mask (0:>(init . bv2v . pack  $ s))
12           else
13             s
14
15
16 lfsr = mealy (lfsrT (0b1100000 :: BitVector 7))
```

Listing 9: Pseudo random nummer generator with 7 bits output and a polynomial of $x^7 + x^6 + 1$ that results in 127 different output states

Next, in order to derive a schedule a valid dataflow graph and list of data dependency edges must be specified in the `SysLift` Module, see Listing 11. Line 2-6 represent the definitions of the actors. An actor has a name and a firing duration. Lines 13-30 represent the data flow edges and token definitions within the graph. Finally, line 33-35 describe the data dependency edges.

The amount of initial tokens can be varied, and will influence the behaviour as will be shown later. The tokens shown in Figure 8.8 are always present. In addition tokens can be

```
mm :: forall a . (Bounded a, Ord a) => a -> a -> (a,a)
mm x y  | x > y     = (x, y)
        | otherwise = (y, x)

swap (x,y) = (y,x)

bubble :: forall n a . (KnownNat n, Bounded a, Ord a) => Vec n a -> a -> (Vec n a, a)
bubble xs a = swap $ mapAccumR mm a xs

bsort :: forall n a . (KnownNat n, Bounded a, Ord a) => Vec n a -> Vec n a
bsort xs = snd $ mapAccumL bubble xs (repeat mInf)
  where
    mInf = minBound :: a
```

Listing 10: Sort algorithm from ECA2 course in 2017

placed on the edges δ_{AB}, δ_{BC}, δ_{CD}, δ_{DE}, δ_{EF} and δ_{FA}. In the first demonstration, one token is placed on all the aforementioned edges.

```
dataflowGraph = Graph (M.fromList
  [hsdfNode 'a' 1 -- random generator
  ,hsdfNode 'b' 1 -- random generator
  ,hsdfNode 'c' 1 -- random generator
  ,hsdfNode 'd' 1 -- random generator
  ,hsdfNode 'e' 1 -- random generator
  ,hsdfNode 'f' 1 -- random generator
  ,hsdfNode 'g' 1 -- Sorter
  ,hsdfNode 'h' 1 -- Id
  ])
  (
   [
     SDFEdge 'a' 'b' 1 1 1,SDFEdge 'b' 'c' 1 1 1,SDFEdge 'c' 'd' 1 1 1
    ,SDFEdge 'd' 'e' 1 1 1,SDFEdge 'e' 'f' 1 1 1,SDFEdge 'f' 'a' 1 1 1

    ,SDFEdge 'a' 'g' 0 1 1,SDFEdge 'g' 'a' 1 1 1,SDFEdge 'b' 'g' 0 1 1
    ,SDFEdge 'g' 'b' 1 1 1,SDFEdge 'c' 'g' 0 1 1,SDFEdge 'g' 'c' 1 1 1
    ,SDFEdge 'd' 'g' 0 1 1,SDFEdge 'g' 'd' 1 1 1,SDFEdge 'e' 'g' 0 1 1
    ,SDFEdge 'g' 'e' 1 1 1,SDFEdge 'f' 'g' 0 1 1,SDFEdge 'g' 'f' 1 1 1


    ,SDFEdge 'a' 'h' 0 1 1,SDFEdge 'h' 'a' 1 1 1,SDFEdge 'b' 'h' 0 1 1
    ,SDFEdge 'h' 'b' 1 1 1,SDFEdge 'c' 'h' 0 1 1,SDFEdge 'h' 'c' 1 1 1
    ,SDFEdge 'd' 'h' 0 1 1,SDFEdge 'h' 'd' 1 1 1,SDFEdge 'e' 'h' 0 1 1
    ,SDFEdge 'h' 'e' 1 1 1,SDFEdge 'f' 'h' 0 1 1,SDFEdge 'h' 'f' 1 1 1


    ,SDFEdge 'a' 'a' 1 1 1,SDFEdge 'b' 'b' 1 1 1,SDFEdge 'c' 'c' 1 1 1
    ,SDFEdge 'd' 'd' 1 1 1,SDFEdge 'e' 'e' 1 1 1,SDFEdge 'f' 'f' 1 1 1
    ,SDFEdge 'g' 'g' 1 1 1,SDFEdge 'h' 'h' 1 1 1
  ])

ddEdges = [DDEdge 'a' 'g', DDEdge 'b' 'g', DDEdge 'c' 'g',DDEdge 'd' 'g',
           DDEdge 'e' 'g', DDEdge 'f' 'g', DDEdge 'a' 'h',DDEdge 'b' 'h',
           DDEdge 'c' 'h',DDEdge 'd' 'h', DDEdge 'e' 'h', DDEdge 'f' 'h']
```

Listing 11: Dataflow graph and dependency edges

A valid strictly periodic schedule can be derived from the aforementioned dataflow graph. The schedule along with the dependency edges can be verified by running the function `ddGraph`. this will return the output that is shown in Listing 12. It depicts the actors along with their period and start time (line 3-11). Furthermore, it lists all the data dependency edges (line 12-25).

The resulting hardware architecture should be connected as depicted in Figure 8.9. This

```
*SysConfig> ddGraph
Actors:
[DDActor:'a', period: 2, Start time: 0;
,DDActor:'b', period: 2, Start time: 0;
,DDActor:'c', period: 2, Start time: 0;
,DDActor:'d', period: 2, Start time: 0;
,DDActor:'e', period: 2, Start time: 0;
,DDActor:'f', period: 2, Start time: 0;
,DDActor:'g', period: 2, Start time: 1;
,DDActor:'h', period: 2, Start time: 1;
]
Edges:
['a'-->'g';
,'b'-->'g';
,'c'-->'g';
,'d'-->'g';
,'e'-->'g';
,'f'-->'g';
,'a'-->'h';
,'b'-->'h';
,'c'-->'h';
,'d'-->'h';
,'e'-->'h';
,'f'-->'h';
]
```

Listing 12: Output of the `ddGraph` function

requires manual nesting of the functions. This paragraph will guide the reader through the progress of establishing the desired hardware architecture. The difficult part in connecting the components is, that they have intertwined dependencies. The (automatically generated) firing schedule of the actors and the schedule of the crossbar can be accessed by running `fireScheduleV` or `cBSchedule`, respectively. The `Signal pCnt` is the program counter that selects the desired record from the schedules.

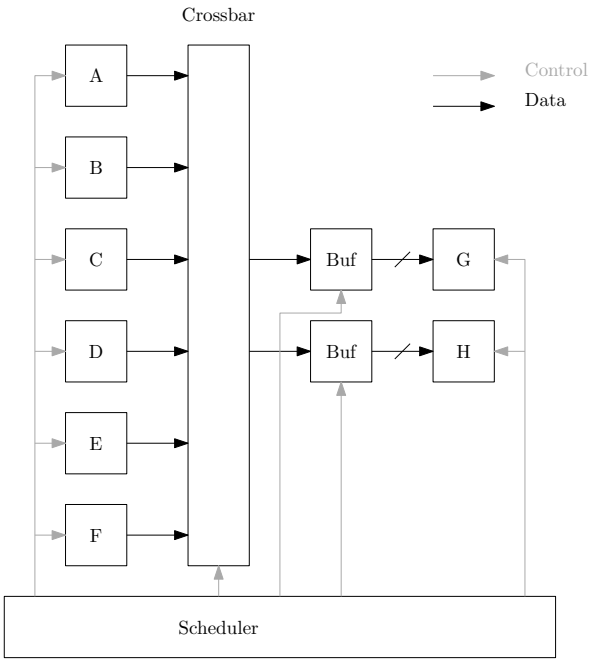


Figure 8.9: Connection diagram of the functional blocks

```
fireScheduleV = $(listToVecTH fireSchedule)

pCnt = pCntWrap fireScheduleV

actorEn = asyncRom fireScheduleV <$> pCnt
crossBarCtrl = asyncRom cBSchedule <$> pCnt
```

The `Signal actorEn` represents, which actors can fire over time by means of a vector of booleans. The `Signal crossBarCtrl` holds the configuration of the crossbar over time. Next, the enable signals for the individual actors must be split from `actorEn`. The output of the crossbar has 12 ports, that have to be connected to the actors G and H. The order in which the actors are placed depend on the order in which the data dependencies were defined. The order can be observed by running `crossBarInfo`, which will return an output as shown in Listing 13. The result is a list of tuples with three fields. The function of each field in a tuple is as follows: The first field of a tuple represents an actor that is consuming, which is in this example actor G (line 2) or actor H (line 9). The second field lists the actors that produce data for the actor in the first field of the tuple (line 3-8 and line 10-15 for actor G and H, respectively). The second field is a list of lists. Every list within the list represents the actors that share a port. In this example there are six lists of 1 actor within a list. This indicates that there are six ports to the consuming actor, where each of the ports is only used for one actor. The number of ports to which the actor in the first field is connected to, is listed in the third field of the tuple (line 9 and line 16 for actors G and H, respectively). From the output can therefore be concluded that G is connected to the first 6 outputs of the crossbar, and H is connected to the last 6 outputs of the crossbar.

```
*SysConfig> crossBarInfo
[(DDActor:'g', period: 2, Start time: 1;
,[[DDActor:'f', period: 2, Start time: 0;
],[DDActor:'e', period: 2, Start time: 0;
],[DDActor:'d', period: 2, Start time: 0;
],[DDActor:'c', period: 2, Start time: 0;
],[DDActor:'b', period: 2, Start time: 0;
],[DDActor:'a', period: 2, Start time: 0;
]],6),(DDActor:'h', period: 2, Start time: 1;
,[[DDActor:'f', period: 2, Start time: 0;
],[DDActor:'e', period: 2, Start time: 0;
],[DDActor:'d', period: 2, Start time: 0;
],[DDActor:'c', period: 2, Start time: 0;
],[DDActor:'b', period: 2, Start time: 0;
],[DDActor:'a', period: 2, Start time: 0;
]],6)]
```

Listing 13: Output of the `ddGraph` function

Next, the vector of enable signals `actorEn` (the scheduler) is split to the actors. `actorEn` is a vector with booleans of the `Signal` type:

```
enableABCDEF = traverse (\ x -> (!! x) <$> actorEn) (iterate d6 (+1) 0 )
enableG = (!! 6) <$> actorEn
enableH = (!! 7) <$> actorEn
```

The functions that represent the actors F to A can now be initialized in the system. Since they

all represent the same randomize function (albeit with a different starting value), the following transformation can be applied:

```
actorFEDCBA
  = traverse (\ x -> lfsr (x+1) $ (!! x) <$> actorEn) (iterate d6 (\x -> x - 1) 5)
```

`actorFEDCBA` is a `Signal` that represents a vector with the values that are produced by the six instances of the `lfsr` function. Note that the enable signals from the scheduler (`actorEn`) are included as well.

The actors G and H require the instantiation of a buffer. This buffer needs to be defined in the `SysLift` module. The size of the buffers that are required for the buffers of both actor G and actor H are found with the `findBufferSizes` function. The functions looks at the tokens on all outgoing edges from G or H:

```
bufG = map fromInteger $ findBufferSizes ((!! 6) . getActors $ ddGraph)
bufH = map fromInteger $ findBufferSizes ((!! 7) . getActors $ ddGraph)
```

Then the actors G and H can be instantiated in the `SysConfig` module:

```
actorG = actSort buf enableG
  where
    t x = take d6 <$> x
    buf = bufWrap pRangeV initS $ bundle (t crossBarCtrl,t cbOut,enableABCDEF,enableG)
    pRangeV = $(listToVecTH (pRange  bufG ))
    initS = (ptrsV,ptrsV,bufV) where
      bufV = $(listToVecTH (L.replicate (L.sum bufG) (0 :: DataType)))
      ptrsV = $(listToVecTH (ptrs . pRange $ bufG))


actorH = actDev buf enableH
  where
    t x = drop d6 <$> x
    buf = bufWrap pRangeV initS $ bundle (t crossBarCtrl,t cbOut,enableABCDEF,enableH)
    pRangeV = $(listToVecTH (pRange bufH ))
    initS = (ptrsV,ptrsV,bufV) where
      bufV = $(listToVecTH (L.replicate (L.sum bufH ) (0 :: DataType)))
      ptrsV = $(listToVecTH (ptrs . pRange $ bufH ))
```

These functions are relatively hard to read, but they both consist of the same fields. The buffer needs to demultiplex the data that it receives. In order to do the correct buffer allocation, it requires information about which actors are enabled (`enableFEDCBA`) and about the current state and relevant outputs of the crossBar (`crossBarCtrl` and `cbOut`), for the port that the actor is connected to. The latter requires some filtering, as only the parts for actor G or actor H are relevant. The function `t` is used to select the relevant inputs for the initiated buffer. The other signals are more straight forward:

`actor[X]` is the coupling of a function to a buffer and the enable signal of that actor.

`t` filters the relevant information of the crossBar to pass it to the buffer. It is based on the information from `crossBarInfo`.

`buf` represents initiation of the actual buffer

`pRangeV` contains the range in which the actors that produce data for the consuming actor can write. (has dependency to `bufX` in `SysLift` module)

`initS` is the initial state of the buffer

`bufV` is the actual buffer (has dependency to `bufX` in `SysLift` module)

`ptrsV` represents the vector of pointers that is bounded to the range in `pRangeV`.

The output of the crossbar (`cbOut`) has not been defined yet. To generate a crossbar, one needs to give the number of inputs and outputs (`sysConfigConstraints`) to the `crossBar2d` function. Then the crossbar also has a control signal (`crossBarCtrl`) and it requires the inputs of all the actors that produce data. The `crossBar2d` function does not natively operate in the `Signal` domain, and needs to be lifted.

```
1  cbOut = liftA2 (crossBar2d sysConfigConstraints) actorFEDCBA crossBarCtrl
```

If the code in the `SysLift` and `SysConfig` module are adjusted according to the described steps, then the hardware architecture could be simulated. The whole graph functions as a topentity. Therefore the designer must decide which actor outputs must be available, e.g. G and H:

```
1  topEntity
2   :: Clock System Source
3   -> Reset System Asynchronous
4   -> Signal System ((Vec 6 DataType),(Vec 6 DataType))
5  topEntity = exposeClockReset $ bundle (actorG,actorH)
6  {-# NOINLINE topEntity #-}
7
8
9  testBench :: Signal System ((Vec 6 DataType),(Vec 6 DataType))
10 testBench = done
11   where
12     done         = topEntity clk rst -- testInput)
13     clk          = tbSystemClockGen (not <$> pure True)
14     rst          = systemResetGen
```

Evaluating the testbench will now give the outputs over time. Here the testBench is sampled 10 clock cycles. From the results can be noticed that results are computed on every other cycle.

```
*SysConfig> sampleN 10 testBench
[(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<6,5,4,3,2,1>,<6,5,4,3,2,1>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<98,97,96,3,2,1>,<3,98,2,97,1,96>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<97,96,80,49,48,1>,<97,49,1,80,96,48>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<120,96,80,48,40,24>,<80,120,96,40,48,24>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<60,48,40,24,20,12>,<40,60,48,20,24,12>)]
```

Improving the throughput of the system

The throughput of the graph can be improved, by increasing the buffersize between the random generators (A to F) and the consumers (G and H) in the dataflow graph. In the new scenario, both G and H can produce data at each clock cycle, by increasing all the buffers to a size of two:

```
*SysConfig> sampleN 10 testBench
[(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<6,5,4,3,2,1>,<6,5,4,3,2,1>)
,(<98,97,96,3,2,1>,<3,98,2,97,1,96>),(<97,96,80,49,48,1>,<97,49,1,80,96,48>)
,(<120,96,80,48,40,24>,<80,120,96,40,48,24>),(<60,48,40,24,20,12>,<40,60,48,20,24,12>)
,(<30,24,20,12,10,6>,<20,30,24,10,12,6>),(<15,12,10,6,5,3>,<10,15,12,5,6,3>)
,(<103,98,97,6,5,3>,<5,103,6,98,3,97>),(<98,97,83,80,49,3>,<98,83,3,49,97,80>)]
```

Another adjustment to the temperal behaviour of the graph

Another example of a different schedule is one where only one token is passed through A to F. Besides the schedule, this also requires an adjustment to the `t` function within the function declaration, because the number of crossbar outputs are adjusted to the generated schedule. In this example, both G and H only require one simultaneous input:

```
1 t x = take d1 <$> x -- first port of crossbar for actor G
2 t x = drop d1 <$> x -- second port of crossbar for actor H
```

The increased buffers of the previous example are set to their original size. A problem here, is that order of the actors has changed, according to the `crossBarInfo` function. Therefore the outputs of the crossbar must be adjusted accordingly.

```
1 --  former: actorFEDCBA
2 actorABCDEF= traverse (\ x -> lfsr (x+1) $ (!! x) <$> actorEn)  (iterate d6 (+1) 0 )
```

The result of the simulation is as expected. G and H can very every 7th clock cycle:

```
*SysConfig> sampleN 14 testBench
[(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<0,0,0,0,0,0>,<0,0,0,0,0,0>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<0,0,0,0,0,0>,<0,0,0,0,0,0>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<0,0,0,0,0,0>,<0,0,0,0,0,0>)
,(<6,5,4,3,2,1>,<1,2,3,4,5,6>),(<0,0,0,0,0,0>,<0,0,0,0,0,0>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<0,0,0,0,0,0>,<0,0,0,0,0,0>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<0,0,0,0,0,0>,<0,0,0,0,0,0>)
,(<0,0,0,0,0,0>,<0,0,0,0,0,0>),(<98,97,96,3,2,1>,<96,1,97,2,98,3>)]
```

In the current state of the tool, it is not always possible to derive hardware from a valid dataflow schedule. if the start times or period of the tokens is a fraction (which can be observed by running `ddgraph`), then it is not possible to generate a schedule. This is due to the constraint that one time unit equals one clock period, and this is an indivisible element.

RTL-level comparison for different schedules

In this section, the influences on the resulting hardware, due to a different schedule are examined in Vivado 2017.4 for the Zybo Z7-20 platform. It was aimed to generate Verilog code with the Cλash-Compiler 0.99.3. However, for unknown reasons, the resulting code could not be synthesized, as Vivado started to utilize all RAM and SWAP on the host computer. However, the design could be compiled to VHDL. The ”Run Implementation” process is ran (which maps the hardware on the FPGA), and then the results are examined. A clock of 100 MHz was used as timing constraint. The aim was to perform a power analysis based on the SAIF file that was obtained, by running the testbench that was generated by Cλash. However, the design was so small that the power consumption was neglected by the Idle usage of the FPGA. The following three schedules and results are evaluated:

- Buffer size of two between the consumer and the producers, all producers fire concurrently. (Period of schedule: 1) $\rightarrow$ 1342 LUTs and 329 Flip-Flops.
- Buffer size of one between the consumer and the producers, all producers fire concurrently. (Period of schedule: 2) $\rightarrow$ 543 LUTs and 190 Flip-Flops.
- Buffer size of one between the consumer and the producers, only one producer fire in a time instance. (Period of schedule: 7) $\rightarrow$ 485 LUTs and 212 Flip-Flops.

The utilization is a little higher than expected. It is difficult to base this claim on the number of LUTs. However, an estimate of the number of Flip-Flops can be deduced from the Cλash implementation. For the first example, with the larger buffer size, the expected amount of Flip-Flops can be deduced as follows:

- Datatype size is 7 bits and will occupy 7 Flip-Flops
- actors G and H hold a buffer of size 2, for each of the producing actors. $\rightarrow 2 \cdot 84$FF
- the total buffer size of the actors G and H is 12, therefore the read and write pointers are 4 bits wide. There are 6 write pointers and one readpointer $\rightarrow 2 \cdot 48$FF
- The schedule has a period of one, this requires only one register. $\rightarrow 1$FF
- The random number generators each have 7 registers to store the current state. $\rightarrow 42$FF

In total this should utilize around 307 Flip-Flops, opposed to 349. An elaboration on the pre-routing RTL-level clarifies that for both actor G and H, the whole dependency structure is generated on RTL. However, this would imply that the design would be larger than 349. It is therefore unclear where the extra registers exactly come from. As the thesis does not focus on area optimization, it is also of less importance. What is more interesting, is the hardware savings, as a result of a different schedule. Unfortunately, the hardware savings are not proportional to the throughput reduction that the less concurrent schedule offers. This is mainly due to the fact that all the hardware utilizing functions are still synthesized, despite the different enable rates.

Chapter 9

Conclusions

The research question within this thesis was stated as: *How can the unique features of the Cλash language be utilized to define a deterministic System on Chip interconnection standard?*. This chapter aims to answer this question.

During this study, a methodology to generate hardware from data flow was excogitated and implemented as a tool. The compliance with data flow ensures that the generated hardware is deterministic. One of the constraints within the project was to use the features that the Cλash-language offers. This has been utilized in a way that data flow graphs are pre-evaluated and translated in Haskell, to synthesizable expressions that the Cλash-compiler can compile to either VHDL or Verilog. The proposed methodology was tested for two HSDF examples, where a strictly periodic schedule was feasible. The resulting architectures followed the exact behaviour that was described in data flow.

The results showed that the tool indeed changes the behaviour of the synthesized hardware according to the data flow schedule. However, the changes in hardware were not proportional to the changes in the throughput, because the hardware of the functions are still generated. The only difference is that the function is less often addressed to execute. Components as the crossbar are smaller in hardware, due to wire optimizations. However, this is often trivial in comparison with the rest of the design. The resulting architecture is likely to be more energy efficient, as actors are less often activated. It has been attempted to deduce the energy consumption, but the tested designs were too small to give an accurate energy consumption.

Although the presented method has it's limitations, it is an entirely different view on connecting functional hardware blocks to each other, which might be interesting for the research topics within the CAES group. The shortcomings and potential subjects for future work are elaborated further in Chapter 10.

Chapter 10

Discussion and Future work

This chapter discusses the limitations of the obtained results. It also points to areas of interest for future work.

10.1 Nesting functions

One of the major limitations is that nesting the generated blocks is a cumbersome task. A function often receives its input from the (selected) output of multiple other functions and produces an output that is (partially) connected to other functions. Moreover, if the functions operate on the `Signal` domain, then all connection functions need to be lifted to this domain, which adds even more manual labour to connect the blocks to each other. A possible solution to the aforementioned problem is to use a visual tool that can help the designer to connect individual components to each other. It would be interesting if the tool could generate the resulting Cλash wrapper so that the whole system can still be simulated in the interactive environment. The University of Twente has attempted to visualize Haskell in [15]. This utility could be a good starting point. The downside of this tool is that it still requires the designer to connect the generated components, while the process of generating components already implies how they should be connected. This process would involve a parser that scans the files and the generation of code that wraps the components together.

10.2 Support for SDF

Another limitation is the lack of support for Multi-rate synchronous data flow (SDF) graphs when generating the hardware. Most of the blocks that were introduced support a subset of SDF, which can be added to the tool with relative ease. However, some of the properties do not just translate to hardware. This section elaborates in which way the presented methodology is limited in the usage of SDF. The presented method is not restricted on the field of synchronization, as long as a feasible strictly periodic schedule can be derived. However, the method is restricted on the generation of buffers in the current implementation, as an actor deduces its buffer size based on the tokens on its outgoing edges.

Figure 10.1 depicts an SDF graph along with its firing schedule and token distribution. In contrast to the examples that were shown in Chapter 8, it is difficult to make unambiguous statements about the buffer derivation. Actor B consumes three tokens (data blocks) and frees up three buffer slots as a result. It can be modelled with the earlier discussed tools. Actor A is on the other hand more difficult to model. It produces two data blocks in one firing, therefore it needs two data lines to transfer it to the buffer of actor B. However, this results in a contradiction when it produces only one data block for actor C. In conclusion, although the data flow graph shows a feasible schedule, the current derivation method for the buffer size is not sufficient.

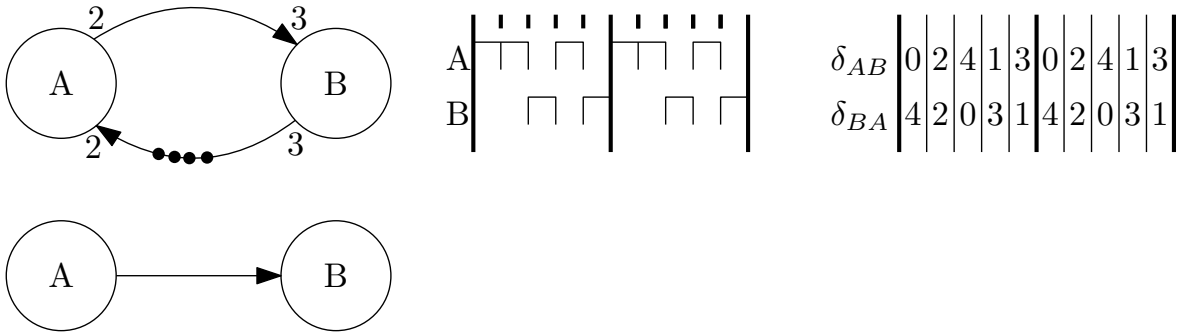


Figure 10.1: An example with implicit self-edges that causes the buffer generation to fail

Another case where the buffer generation results in faulty behaviour for SDF is shown in Figure 10.2. In this example, A produces 2 tokens for actor C, while producing only one token for actor B. Again, the data flow graph can be scheduled, but the buffer generation can not be correct because the representation of tokens is ambiguous (the result of A represents 1 and 2 tokens for B and C, respectively).

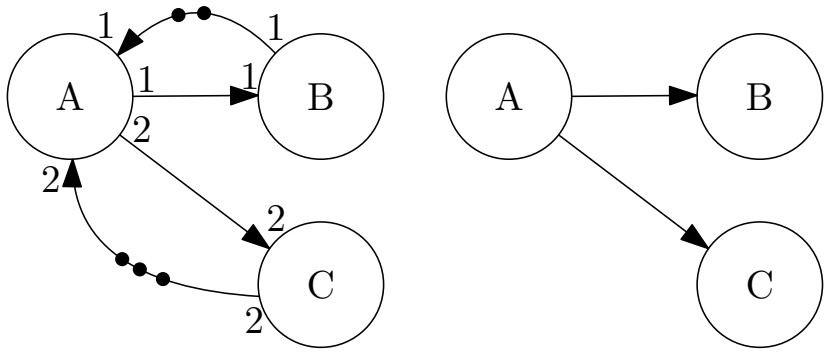


Figure 10.2: A second example with implicit self-edges that causes the buffer generation to fail

As long as all produced tokens have an unambiguous representation, it is possible to derive the correct buffer sizes. This is depicted by Figure 10.3. In this example, the token production from A to each consuming actor is the same, while the actors B and C consume a different number of tokens when they fire. From the way it is modelled, it can be observed that B and

C require 2 and 4 data blocks from A, respectively.

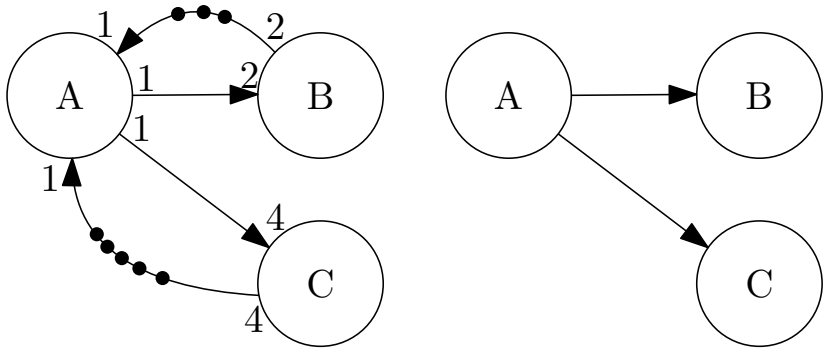


Figure 10.3: An example with implicit self-edges that can work with the current buffer generation

Another example is shown in Figure 10.4. This graph can operate correctly, because A does fire the same amount of tokens to all its consuming tokens. However, since it fires two data blocks, it could also be modelled as an actor that fires one larger block of data (it can be seen as an abstraction), which then results in the graph that is shown in Figure 10.5. It does not change the behaviour of a data flow graph, since the tokens serve as a synchronization mechanism. The graph depicts single token production, thus it could be generated to a hardware structure.

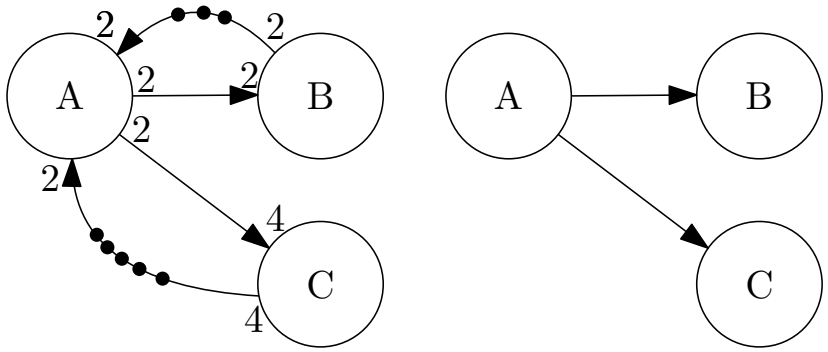


Figure 10.4: Another example with implicit self-edges that can work with the current buffer generation

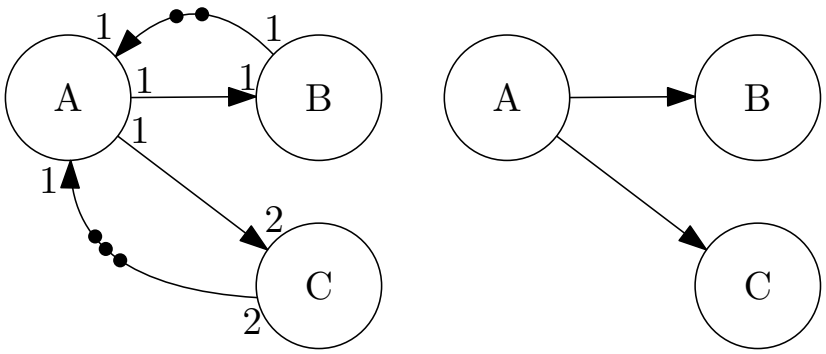


Figure 10.5: An abstraction of Figure 10.4

10.3 Bugs in the Cλash-compiler

While conducting the tests in Section 8.3 and Appendix F, one of the Cλash designs could only be synthesized when it was compiled to VHDL. The resulting VHDL design could be simulated in Vivado, and also simulated successfully in the interactive Cλash environment. This behaviour is undesirable, especially when students work with different versions of Cλash and different versions of Vivado that may cause these problems.

10.4 Interfacing between schedules

Systems that allow a feasible strictly periodic schedule have been considered in this thesis. However, often a larger system consists of multiple sub-systems, which could result in no feasible strictly periodic schedule. The time frame of this master project did not allow to finalize the principles that are discussed in this section. The presented principles are considered future work.

In Figure 10.6 two controllers which both have their own schedule are shown. The firing of actor C requires on a unpredictable event on the outside. Therefore, once it is enabled by the scheduler in scheduler 1, it must send an acknowledge back to notify that it has indeed fired. scheduler 2 is meanwhile blocking on the firing of actor D, since it must first be notified by scheduler 1 that C has fired.

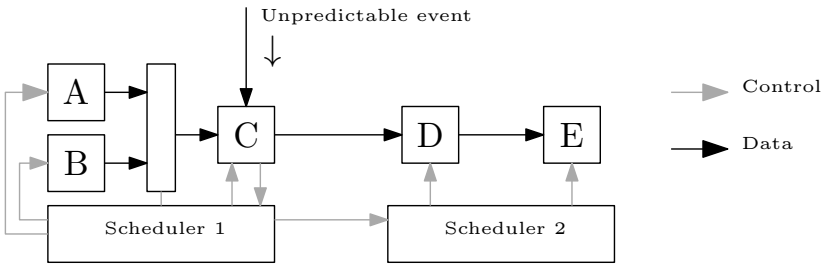


Figure 10.6: A possible configuration for multiple controllers

One method to realise the communication between two schedules is to use handshake techniques that are used in Globally Asynchronous Locally Synchronous systems [16]. This makes sure that every component obeys the firing rules by means of back presure. A redundant handshaking protocol is the 4-phase protocol [17]. It is often used in asynchronous systems, but can also be used for synchronous systems that operate in one or multiple clock domains, as this might be the case. The phases of the 4 phase protocol are as following:

1. The producer pushes the data along with an asserted request signal;

2. The consumer fetches the data and asserts the acknowledge signal;

3. The producer negates the request signal;

4. The consumer negates the acknowledge signal.

The protocol at first seems superfluous as every signal returns to zero. However, it is unwise to neglect the steps entirely, because that can result in false reads. There is however also a 2 phased protocol that acts solely on bit transitions.

Nevertheless, the latter requires additional logic to implement the transition handshaking mechanism. For systems that require buffering the structure in Figure 10.7 could be used. This method uses a FIFO that uses a read and write interface that communicate according to the four phase handshkaing protocol. It has a relatively high latency since it requires two handshakes to communicate data from one system to another, but the throughput can be relatively high, since the actors can (to some extend) act independently due to the FIFO buffer in between. For homogeneous Peer to Peer transfers the approach in Figure 10.8 could be used as well. It has a lower throughput (since one schedule can be stalled by another), but it requires only one handshake for a transfer, and thus has a lower latency.

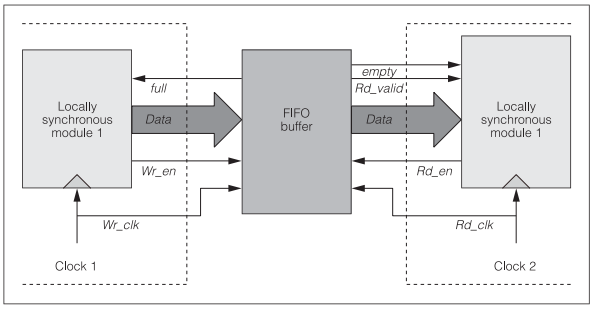


Figure 10.7: Typical FIFO-based GALS system [16]

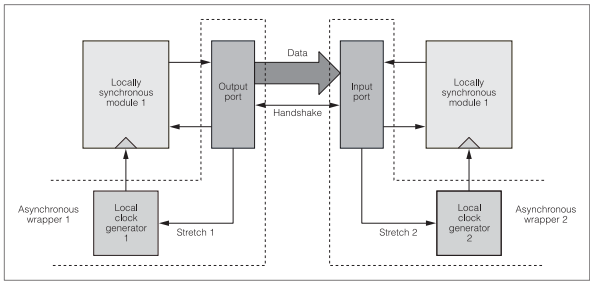


Figure 10.8: Gals system with pausable clocking [16]

The interfaces that enable four phase handshaking between schedules have been designed in Cλash (code available in Appendix D.4) and are briefly elaborated in this section. The designed interfaces are as following:

T_{IF} An interface that requests (transmits) data to one data receiving interface, and waits for an acknowledge back before it proceeds.

TV_{IF} An interface that requests data to one or multiple data receiving interfaces, and waits for an acknowledge back from all receivers before it proceeds.

R_{IF} An interface that receives data from a transmitter, and subsequently asserts its acknowledge to signal the transmitter that the data is received.

RV_{IF} An interface that receives data from multiple transmitters, and subsequently asserts it acknowledge when it has received data from all it's transmitters.

Three possible use cases of the interfaces are depicted in Figure 10.9. The handshaking interfaces are not yet part of the presented tool. Integration of these interfaces or another type of interface is considered as future work.

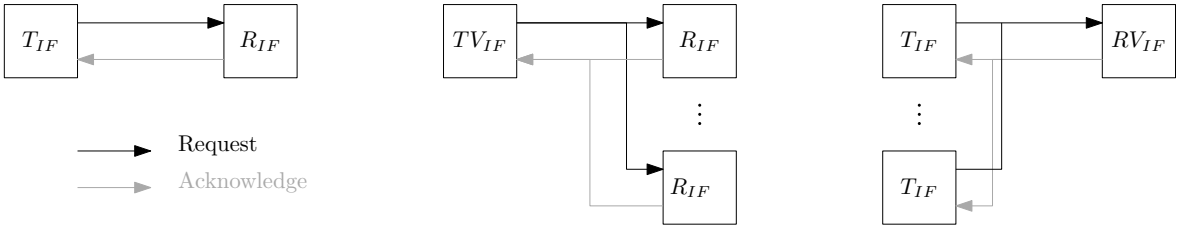


Figure 10.9: An example with the available interface types

10.5 Other optimizations

The presented method requires a data flow graph where all actors have a self-edge. It could be interesting to analyse what the largest actor firing concurrency is, and generate the amount of required actors in hardware automatically. The implementation of this behaviour extends the subset of data flow that the tool can operate in. Furthermore, it will influence the design on hardware utilization as well.

Another limitation of the generation to hardware, is that fractional execution or start times are not supported. One time unit was set to be equal to one clock cycle and is therefore indivisible. Therefore graphs that result in fractional schedules are not supported. This is not necessarily a problem, since a schedule can be multiplied by the least common multiple of the fraction denominators within the schedule.

Bibliography

[1] "From Haskell to Hardware," 2018, visited on 18-10-2018. [Online]. Available: http://www.clash-lang.org/

[2] ARM, "Amba axi and ace protocol specification axi3, axi 4, and axi4-lite ace and ace-lite," visited on 04-10-2018. [Online]. Available: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf

[3] Intel®, "Avalon® interface specifications," visited on 04-10-2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

[4] ARM, "Amba™ specification," visited on 04-10-2018. [Online]. Available: http://soc.eecs.yuntech.edu.tw/Course/SoC/doc/amba.pdf

[5] OpenCores.org, "Wishbone b4," visited on 04-10-2018. [Online]. Available: https://cdn.opencores.org/downloads/wbspec_b4.pdf

[6] A. Topirceanu, "Network on chips," visited on 13-01-2019. [Online]. Available: https://sites.google.com/site/alexandrutopirceanu/research/networks-on-chips

[7] OpenCores.org, "Opencores," visited on 21-01-2019. [Online]. Available: https://opencores.org/

[8] P. W. Maarten Wiggers, Joost Hausmans and M. Geilen, *Dataflow Analysis for Real-Time Multiprocessor Systems*. Springer, 2017, visited on 26-11-2018.

[9] R. Reiter, "Scheduling parallel computations," *J. ACM*, vol. 15, no. 4, pp. 590–599, Oct. 1968. [Online]. Available: http://doi.acm.org/10.1145/321479.321485

[10] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[11] R. de Groote, "On the analysis of synchronous dataflow graphs: a system-theoretic perspective," Ph.D. dissertation, University of Twente, 2 2016, visited on 25-01-2019.

[12] R. d. G. Hendrik Folmer, "Haskell dataflow fpga," visited on 10-01-2019. [Online]. Available: https://github.com/Hendrik-0/haskell-dataflow-fpga

[13] Y. Kim, S. Jadhav, and C. S. Gloster, “Dataflow to hardware synthesis framework on fpgas,” in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Oct 2016, pp. 91–96.

[14] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static data flow,” in *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, May 1995, pp. 3255–3258 vol.5.

[15] A. B. et al., “Visual programming meets haskell,” visited on 30-01-2019. [Online]. Available: https://github.com/viskell/viskell

[16] M. Krstic, E. Grass, F. K. Gürkaynak, and P. Vivet, “Globally asynchronous, locally synchronous circuits: Overview and outlook,” *IEEE Design Test of Computers*, vol. 24, no. 5, pp. 430–441, Sep. 2007.

[17] JensSparso, *Asynchronous Circuit Design: A Tutorial*, 2006, visited on 26-11-2018. [Online]. Available: http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/855/pdf/imm855.pdf

Part III

Appendix

Appendix A

Existing SoC communication

This Appendix gives a brief overview of a variety of commonly used protocols in the industry. A comparison is drawn in Chapter 3 of this thesis.

A.1 AMBA

The Advanced Microcontroller Bus Architecture specification defines three distinct busses [4]:

AHB or Advanced High-performance Bus is designed for high clock frequency, high performance system modules. It's aim is to provide a efficient connection of processors, on/off-chip memories.

ASB or Advanced System Bus is the predecessor of the AHB. The core differences are the lack of burst transfers and split transactions.

APB or Advanced Peripheral Bus is optimized for power consumption and reduced interface complexity. It can be used in conjunction with either ASB and AHB.

Table A.1 describes the differences between the distinct busses a little more concrete:

Table A.1: The differences between the distinct AMBA busses

AMBA AHB	AMBA ASB	AMBA APB
High performance	High Performance	Low power
Pipelined operation	Pipelined operation	Latched address and control
Multiple bus masters	Multiple bus masters	Simple interface
Burst transfers		Suitable for many peripherals
Split transactions		

The main aim of the AMBA specification is to facilitate in right-first-time development of embedded products with several CPUs or signal processors. Another important aspect is that the bus encourages modular system design, by using several busses. A typical AMBA architecture is shown in Figure A.1. It can be observed that the high-performance modules communicate over an AHB/ASB backbone, whereas the low power peripherals are connected

with an APB. The AHB/ASB backbone can access the peripherals on the APB by means of a bridge that functions as a slave.

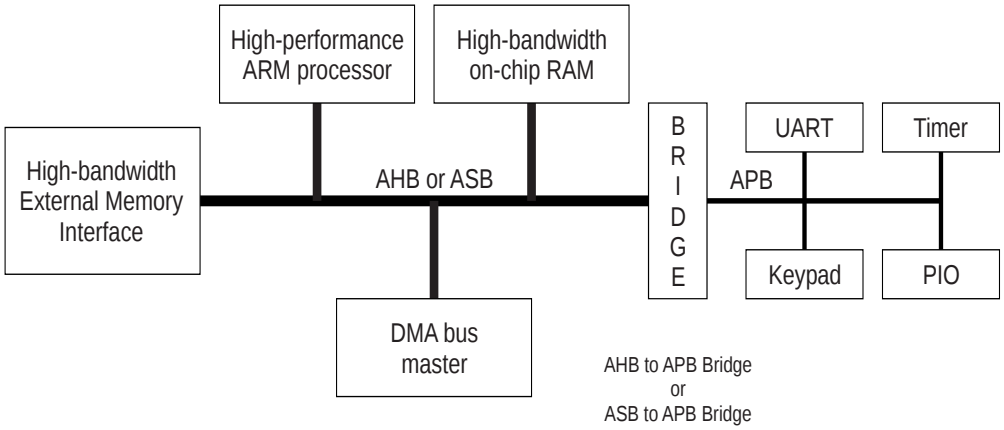


Figure A.1: A typical AMBA bus structure

The AHB/ASB backbone exchanges data with a different protocol as the APB. Since the AHB is an improved version of the ASB, only the former bus is elaborated further. With the AHB a master is routed through all the slaves, there is no partial address encoding, meaning each slave must decode the full address. If there are multiple masters on the bus, then the arbiter grants one master at the time to communicate with a slave. The response from the slave is fed to the master by the decoder. A transaction on the bus consists of two cycles. (1) An address and control cycle and (2) One or more cycles for the data. The control signals inform the slave about the direction, width and whether it is a burst write or not. The slave supports various response signals to inform the master about its current transfer. With AHB, the address and control phase of any transfer occurs during the (last) data phase of the previous transfer. This can result in a higher throughput, as shown in Figure A.2.

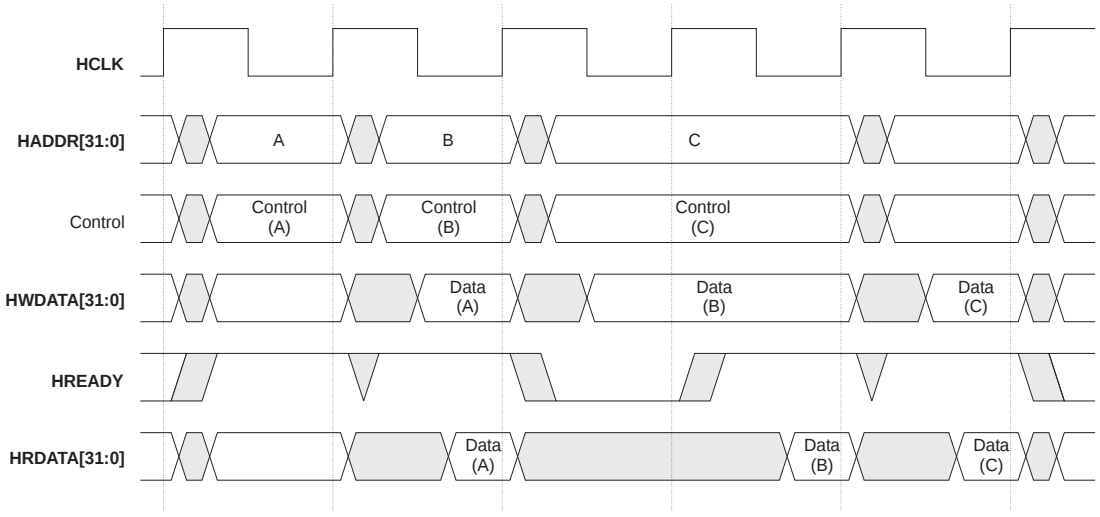
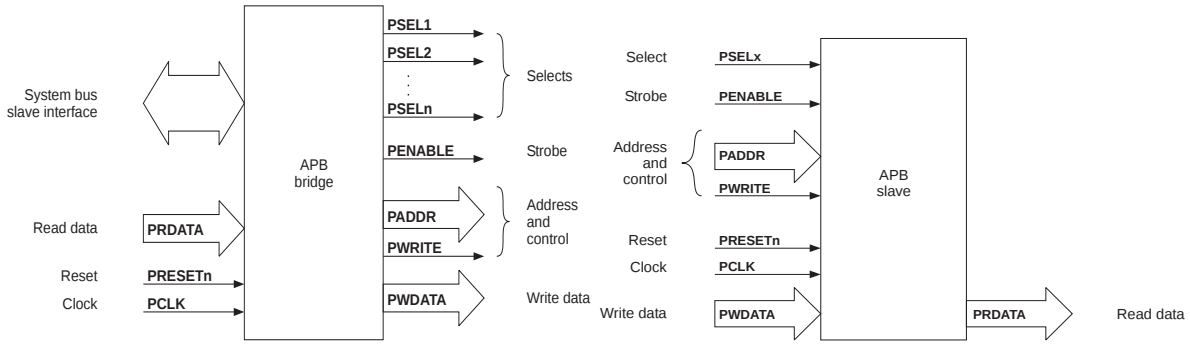


Figure A.2: Multiple transfers example

The APB complies to a different protocol, as it is optimized for minimal power consumption and reduced interface complexity. In contrast to the AHB, there is only one master present at the APB, namely the APB bridge. This bridge is then again a slave of the AHB bus. The bridge and a slave are depicted in Figure A.3a and Figure A.3b, respectively. The bridge decodes the address and asserts the corresponding PSELx that is inputs to a slave. All the slaves are connected on a shared bus, and this system ensures that only one slave can communicate at the time. The protocol over which the bridge and slave communicate is depicted in Figure A.4. On the start of a request, a transition from `IDLE` to `SETUP` occurs. In this stage, the address is decoded and the desired slave is selected. In the next cycle the machine is moved to the `ENABLE` state, in which `PENABLE` is asserted. The address, write and select signals must remain stable during this phase. From this stage the machine moves to either `IDLE` or `SETUP`. The former state is used if another transfer must follow.



(a) The bridge (master) *(b)* An APB Slave

Figure A.3: APB components

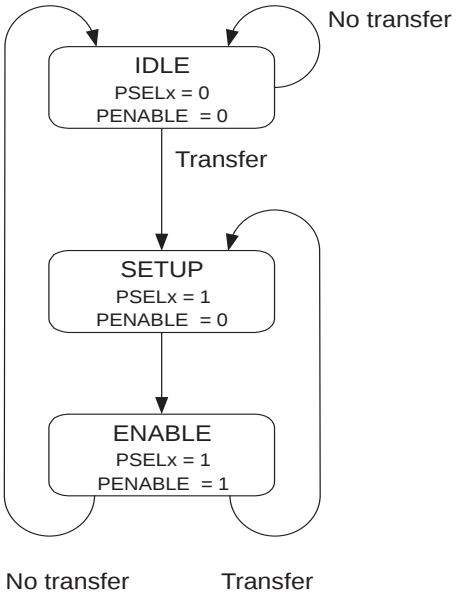


Figure A.4: Transfer protocol

A.2 AXI bus

The AMBA AXI protocol [2] is designed for systems with high performance and operating speed, but also offers subsets of the protocol for relatively simple pheripherals. It is also developed by ARM, and compaitible with AHB and APB interfaces. The AXI4 standard can be divided in several classes:

- Full AXI4: Supports burst data transfers, therefore suitable for high-speed peripherals like DDR3 memory.
- AXI4 Stream: Ideal for processing pipelines (e.g. FFT's, audio- and videoprocessing)
- AXI4 Lite: Subset of the AXI4 standard, no burst available. Suitable for simple peripherals with area constraints.

The AXI protocol consists of five independent transaction channels: Read/Write address, Read/Write data and write response. A channel can carry multiple words/bits that contain information. The AXI protocol allows master and slave interfaces to exchange information. The master is responsible for initiating either a read or write request. The channel communications is shown in Figure A.5 and A.6, respectively. The databus width varies from 8 to 1024 bits and the corresponding channel does also provide a response signal code.

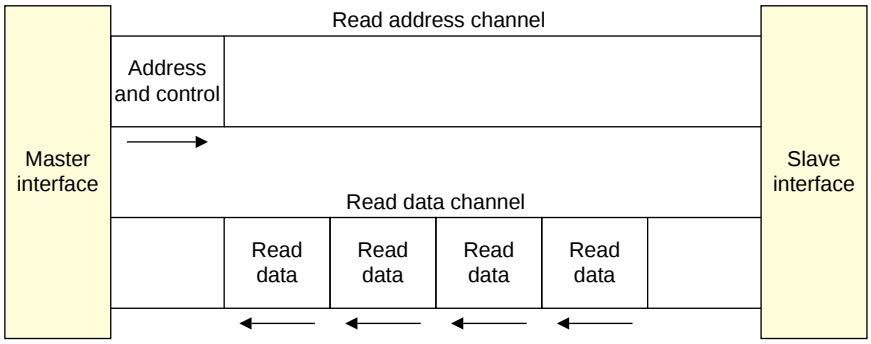


Figure A.5: Read transaction architecture as described in [2]

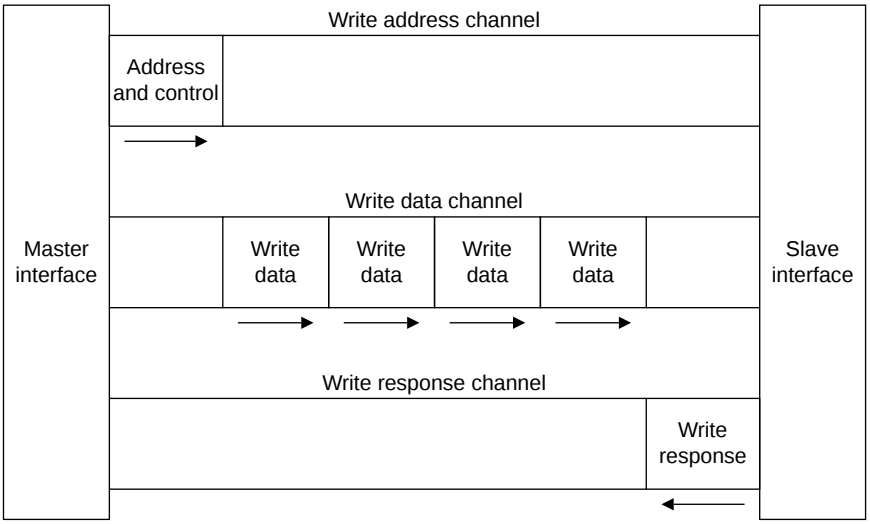


Figure A.6: Write transaction architecture as described in [2]

The structure of the interface and interconnect is shown in Figure A.7. The interconnect handles partial slave addressing, as opposed to AMBA, there is no shared bus. The interconnect can connect a master to a slave directly, which enables concurrent bus access if there are multiple masters.

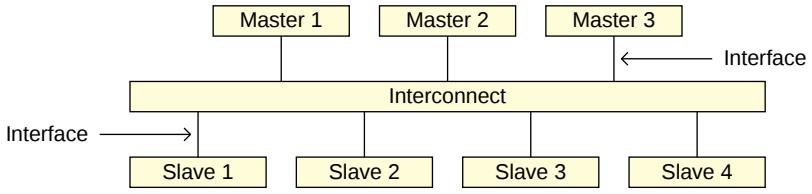


Figure A.7: Interface and interconnect as described in [2]

A.3 Avalon bus

The Avalon bus [3] is used in system design on Intel Altera FPGA's. The standard offers several interfaces for streaming high-speed data, reading and writing registers and memory and controlling off-chip devices. The specification defines the following interfaces:

- Avalon Streaming Interface (Avalon ST), for unidirectional dataflow.
- Avalon Memory Mapped Interface (Avalon-MM), address-based read/write interface.
- Avalon Conduit interface, accomodates signals that do not fit in another interface (e.g. outputs)
- Avalon Tri-State Conduit Interface (Avalon-TC), used to interface with off-chip pheripherals.
- Avalon Clock Interface, an interface that drives or receives clocks

- Avalon Reset Interface, an interface that provides reset connectivity
- Avalon Interrupt Interface, an interface that allows components to signal events to other components, without using another bus.

The interfaces that are considered in this comparison are the Avalon MM and Avalon ST interface. The Avalon MM interface specifies several different signal types, of which only a few are required for a functional design. It also contains Intel Altera specific signals for the NIOS II processor. The interface supports Burst transfers up to 1024 data transactions. The standard finally provides some configurable properties that allow a modular design flow.

A typical read and write transaction is shown in Figure A.8. (1) The master asserts the address, range (byteenable) and read signal after the rising edge. The slave then asserts the waitrequest. (2) the wait request is sampled on the rising edge and causes the master to hold the current signals. (3) After negation of waitrequest, the slave asserts the readdata and response code. (4) the master samples the completed transfer. (5) The master initiates a write transfer and the slave asserts the wait request again. (6) The slave negates the waitrequeest signal again, and (7) captures the write data in the next clockcycle.

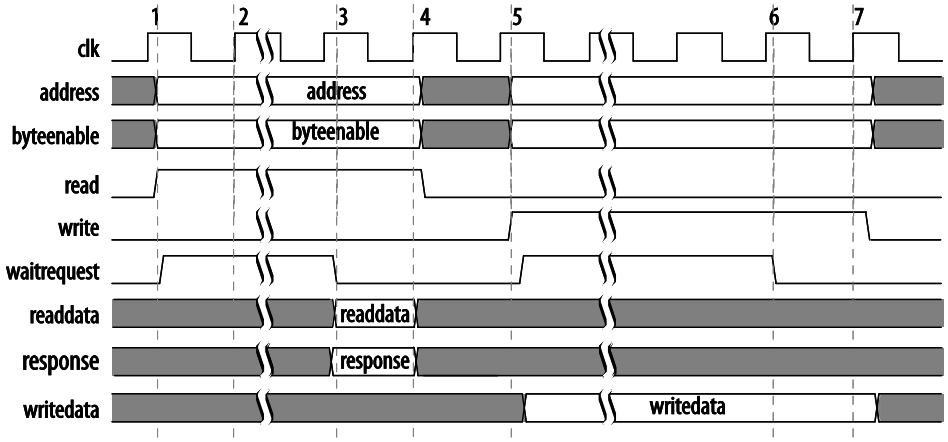


Figure A.8: Avalon MM read transfer

There are many more variations of the transfer that is described above, this can all be adjusted in the property configuration of the MM slave. Also, the interconnect structure is similair to AXI, there is partial address encoding that translates the byte address into an address space of a specific slave.

A.4 Wishbone

The wishbone B4 standard provides a simple protocol while remaining versatile and relatively simple to implement. The standard requires the designer to implement only a subset of the available options to create a working design. If desired, the designer can then expand it. The

designer has also the freedom to choose the interconnection type e.g.: Peer to Peer (P2P), a Shared bus (as used in AMBA) or a CrossBar (as used in AXI/Avalon), see Figure A.9.

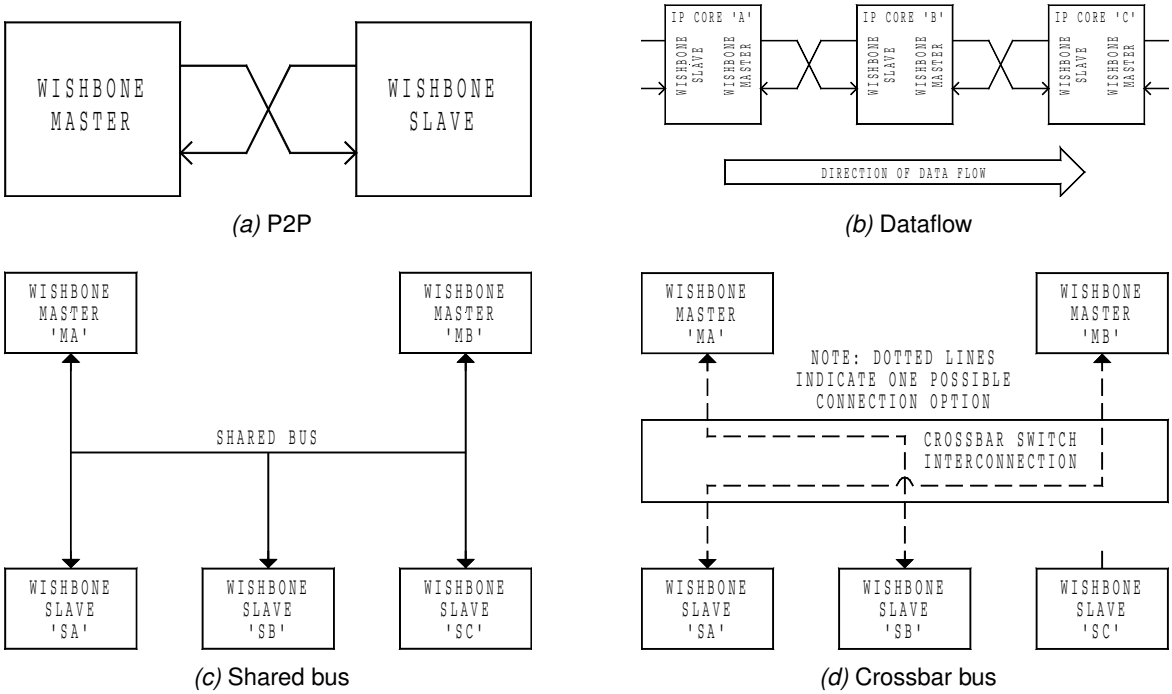


Figure A.9: Wishbone interconnects

Besides the mandatory and optional data signals, the designer can add so called TAGs that belong with either and addres, control or data word. This adds another layer of versatility, since extra information can be encoded in the transaction. A pipelined block write transfer is shown in Figure A.10. The tags are marked by `TGA`, `TGD` and `TGC` for Address, Data and Control, respectively. A transfer is initiated by the assertion of `CYC_O`. The master asserts `STB_O` as long as the data `DAT_O` is stable. In this so called pipelined mode, the master can theoretically send data on every clock cycle. In this example Data word `D1` is delayed because the slave asserts the `STALL_I` signal, which notifies the master that it cannot receive more data.

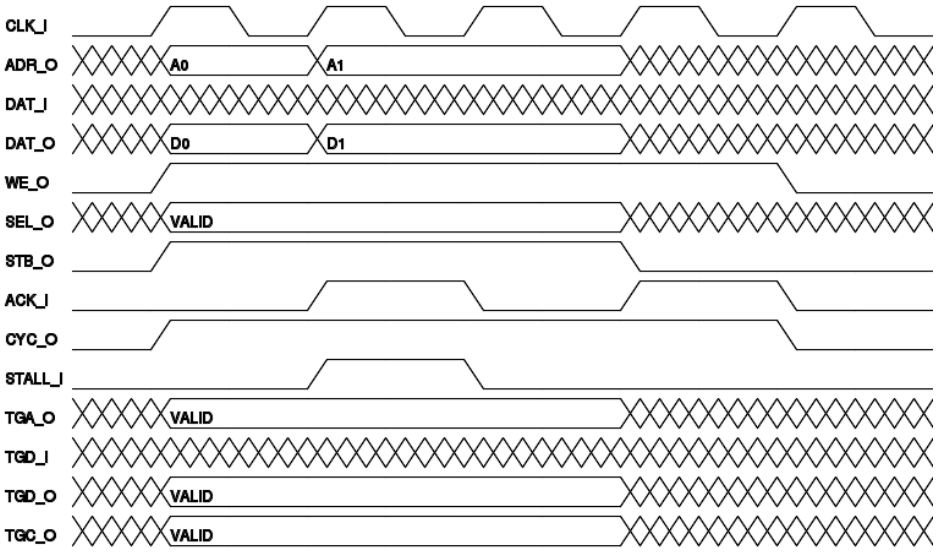


Figure A.10: Wishbone block write transfer

Appendix B

Wishbone implementation in Clash

B.1 Master

```
module WishboneMaster where

    import Clash.Prelude
    import Clash.Explicit.Testbench
    import Types
    import qualified Data.List as L



    masterT s (i_reset,i_cmd_stb,i_cmd_word,i_wb_err,i_wb_stall,i_wb_ack,i_wb_data) = (s',o)
      where
        -- Command words

        state = s

        (i_cmd_reqtype,i_cmd_data) = split i_cmd_word :: (BitVector 2, BitVector 32)
        (i_cmd_add,i_cmd_inc,i_cmd_addrData) =
          (bitToBool . msb . fst $ base, bitToBool . lsb . fst $ base, snd base)
            where
              base = (split  i_cmd_data :: (BitVector 2, BitVector 30))

        i_cmd_rd = i_cmd_stb && (i_cmd_reqtype == (00 :: BitVector 2)) -- Read request
        i_cmd_wr = i_cmd_stb && (i_cmd_reqtype == (01 :: BitVector 2)) -- write request
        i_cmd_bus = i_cmd_rd || i_cmd_wr -- signal that there is a read or write request

        i_cmd_addr = i_cmd_stb && (i_cmd_reqtype == (10 :: BitVector 2)) -- set Address
        i_cmd_special = i_cmd_stb && (i_cmd_reqtype == (11 :: BitVector 2)) -- Bus Reset (special)


        state' = nextState state

        nextState Reset = Idle
        nextState Idle =  if ((i_reset)||((i_wb_err)&&(o_wb_cyc))) then Reset
                          else if i_cmd_bus then BusRequest
                          else Idle
        nextState BusRequest =  if ((i_reset)||((i_wb_err)&&(o_wb_cyc))) then Reset
                                else if (not i_wb_stall) && i_wb_ack then Idle
                                else if (not i_wb_stall) && (not i_wb_ack) then BusWait
                                else BusRequest
        nextState BusWait = if ((i_reset)||((i_wb_err)&&(o_wb_cyc))) then Reset
                            else if i_wb_ack then Idle
                            else BusWait

        o_wb_cyc  | ((state'==Idle) || (state'==Reset) ) = False
                  | otherwise = True

        o_wb_stb  | (state' == BusRequest) = True
                  | otherwise = False

        o_cmd_busy = o_wb_cyc

        o_wb_we   | (state == Idle) = i_cmd_wr
                  | otherwise = o_wb_we

        o_wb_addr = if (state == Idle) then
                      if (not i_cmd_add) then  i_cmd_addrData
                      else o_wb_addr + i_cmd_addrData
                    else if ((o_wb_stb) && (not i_wb_stall) && i_cmd_inc ) then o_wb_addr + 1
                    else o_wb_addr

```

```
        newaddr = ((i_cmd_addr) && (not o_cmd_busy))

        o_wb_data | ((not o_wb_stb) || (not i_wb_stall)) = i_cmd_data
                  | otherwise  = o_wb_data


        s' = state'
        o = (o_cmd_busy,o_wb_cyc,o_wb_stb,o_wb_we,o_wb_data,o_wb_addr, o_rsp_stb,o_rsp_word,s,s')

        -- outputs to the user interface
        o_rsp_stb = if  (i_reset || i_wb_err) then True
                    else if o_wb_cyc then i_wb_ack
                    else newaddr

        o_rsp_word = if  (i_reset) then RspReset
                     else if (i_wb_err) then RspError
                     else if (o_wb_cyc && (not o_wb_we)) then RspData i_wb_data
                     else if (o_wb_cyc && (o_wb_we)) then RspAck
                     else RspAddr o_wb_addr False i_cmd_inc




    master = mealy masterT Reset


    topEntity
      :: Clock System Source
      -> Reset System Asynchronous
      -> Signal System (Bool,Bool,BitVector 34, Bool,Bool,Bool, BitVector 32)
      -> Signal System (Bool,Bool,Bool,Bool, BitVector 32,BitVector 30, Bool,Response,State,State)
    topEntity = exposeClockReset master
    {-# NOINLINE topEntity #-}


    inputs = L.zip i_reset i_cmd_stb
        where
          num_cycles = 10
          i_reset = L.replicate num_cycles False
          i_cmd_stb = L.replicate num_cycles False
```

B.2 Slave

```
module WishboneSlave where

import Clash.Prelude
import Clash.Explicit.Testbench


--memory
type WordType = (Signed 5)
type MemType = Vec 4 (WordType)

data State = Idle | Compute | Finish deriving(Eq)

-- Four registers

base_mem = (0:>0:>0:>0:>Nil)

slaveT s (i_wb_addr, i_wb_data, i_wb_stb, i_wb_we) = (s',o)
  where
    --Fetch from registers
    (memory,state,local_var)    = s

    -- State machine
    state'                        = changeState state

    changeState Compute
      | (local_var >= 9)         = Finish
      | otherwise                 = Compute
    changeState Finish                = Idle
```

```
29      changeState Idle
30          | (i_wb_stb && (not o_wb_stall) && i_wb_we)  = Compute
31          | (i_wb_stb && (not o_wb_stall) && (not i_wb_we))   = Finish
32          | otherwise                    = Idle
33
34      -- Local var
35      local_var'
36        | (state == Idle)        = i_wb_data
37        | (state == Compute)        = local_var+1 -- Random computation
38        | otherwise                    = local_var
39
40      -- Busy
41      busy
42        | (state == Compute)          = True
43        | otherwise                    = False
44
45      -- Ack
46      ack
47        | (state == Finish)         = True
48        | otherwise                    = False
49
50
51      -- Assignment to the Signals
52      o_wb_stall                     = busy
53      o_wb_ack                       = ack
54      o_wb_data
55        | (not i_wb_we)              = memory !! i_wb_addr
56        | (i_wb_we && state == Compute) = local_var
57        | (i_wb_we && state == Finish)  = 0
58        | otherwise                  = -1
59      o                              = (o_wb_stall, o_wb_ack, o_wb_data)
60
61
62      -- Next values in registers
63      memory'
64        | (i_wb_stb == True && i_wb_we == True && o_wb_stall == False && state == Finish)
65          = replace i_wb_addr local_var memory
66        | otherwise
67          = memory
68
69      s' =  (memory',state', local_var')
70
71  slave = mealy slaveT (base_mem,Idle,0)
72
73  topEntity
74    :: Clock System Source
75    -> Reset System Asynchronous
76    -> Signal System (WordType, WordType, Bool, Bool)
77    -> Signal System (Bool,Bool,WordType)
78  topEntity = exposeClockReset slave
79  {-# NOINLINE topEntity #-}
```

Appendix C

Clash code for Analysis

C.1 DataDependency module

```
{-# LANGUAGE DeriveAnyClass        #-}
{-# LANGUAGE DeriveGeneric         #-}


module Utils.DataDependency where
    import Clash.Prelude
    import GHC.Generics
    import Control.DeepSeq


    import qualified Data.List as L

    data DDActor n m = DDActor n m m deriving (Eq,Lift,Generic,NFData,ShowX)
    data DDEdge n    = DDEdge n n
    data DDGraph n m  = DDGraph [DDActor n m] [DDEdge n]

    instance (Show n,Show m) => Show (DDActor n m) where
      show (DDActor name period startTime) =
        "DDActor:" L.++ show name L.++
        ", period: " L.++ show period L.++
        ", Start time: " L.++ show startTime L.++
        ";\n"

    instance (Show n) => Show (DDEdge n) where
      show (DDEdge ns es) = show ns L.++ "-->" L.++ show es L.++ ";\n"

    instance (Show a, Show e) => Show (DDGraph a e) where
      show (DDGraph as es) = "Actors: \n" L.++ show as L.++ "\nEdges: \n" L.++ show es


    instance (Ord n, Ord m) => Ord (DDActor n m) where
      (<=)  a@(DDActor nA _ _) b@(DDActor nB _ _) =  nA <= nB
      (<)   a@(DDActor nA _ _) b@(DDActor nB _ _) =  nA < nB
      (>=)  a@(DDActor nA _ _) b@(DDActor nB _ _) =  nA >= nB
      (>)   a@(DDActor nA _ _) b@(DDActor nB _ _) =  nA > nB
      max   a@(DDActor nA _ _) b@(DDActor nB _ _) =  if nA > nB then a else b
      min   a@(DDActor nA _ _) b@(DDActor nB _ _) =  if nA < nB then a else b

    seqF :: Applicative f => [f a] -> f [a]
    seqF [x]    = (:[]) <$> x
    seqF (x:xs) = (L.++) <$> ((: []) <$> x) <*> seqF xs

    (>#) :: Applicative f => f a -> f (Vec n a) -> f (Vec (n + 1) a)
    (>#) = liftA2 (:>)


    lcmList :: Integral a => [a] -> a
    -- lcmList [] = error "empty list"
    lcmList [x] = x
    lcmList (x:y:ys) =  lcmList $ lcm x y : ys


    findInputsOutputsCrossbar g a = (i,o)
      where
        i = findProducers g a
        o = if not (null i) then bus g a else []


    buss g = bus g  (L.last . getActors $ g) -- build a bus for the last actor

```

```haskell
bus graph actor = findAllCombos ([],exTimes) -- Given a DDGraph and actors, find the ideal bundling to get data to an acto
  where
     exTimes = findExTimes graph (findProducers graph actor)
     -- Generate a list with execution times for the actors that produce data for given actor

findExTimes graph = L.map (`findActorFiringTimes` numSamples)
  where
    numSamples = leastCommonMultiple + maxBootTime
    leastCommonMultiple = findLCM graph
    maxBootTime = maximum $ L.map getBootTime (getActors graph)



findLCM (DDGraph actors _) = lcmList $ L.map getPeriod actors
  where
    getPeriod (DDActor _ p _) = p

getActors (DDGraph actors _) = actors

getEdges (DDGraph _ edges) = edges

getBootTime (DDActor _ _ st) = st



findActorFiringTimes a@(DDActor name p st) lcm' = (a,takeWhile (<= lcm') $ L.iterate (+ p) st)


findActorFiringTimesNoBoot a@(DDActor name p st) lcm' = (a,takeWhile (<= lcm') $ L.iterate (+ p) 0)


findProducers (DDGraph actors edges) actor =  L.map findProducingActor $ filter (\x -> getEdgeConsumer x == getActorName a
  where
    getEdgeConsumer (DDEdge p c) = c
    findProducingActor (DDEdge p c) = L.head $ filter (\x -> p == getActorName x) actors
    -- getActorName (DDActor n _ _) = n

findConsumers (DDGraph actors edges) actor = L.map (findActor . getEdgeConsumer) $ filter (\x -> getEdgeProducer x == getA
  where
    findActor a = L.head $ filter (\ x -> getActorName x == a) actors

getActorName (DDActor n _ _) = n
getEdgeProducer (DDEdge p c) = p
getEdgeConsumer (DDEdge p c) = c

findAllCombos (res,rem) = if null rem' then res'' else findAllCombos (res'',rem') -- Keep extracting largest combo until t
  where
    res'' = res':res -- append the results to eachother
    (res',rem') = extractLargestValidCombo rem --Find largest valid combo and split the result and the remaining exTimes


extractLargestValidCombo exTimes = (largestValidCombo,filter (not . (`elem` largestValidCombo)) exTimes) -- valid combo fr
  where
    largestValidCombo = L.head . findLargestValidCombos $ exTimes -- Take the first combo, for now it does not matter whic


findLargestValidCombos exTimes = findValidCombos (L.length exTimes) exTimes
  where
    findValidCombos 0 xs = error "Insert >0 execution times!"
    findValidCombos n xs = if and overlap then findValidCombos (n-1) xs else show -- If all combinations have overlap, the
      where
        uCombinations = findUComb n xs -- find unique combinations for the execution times (xs) with a width of n
        overlap = L.map (overlapN . L.map snd) uCombinations -- show which of the unique combinations have overlap
        show = L.map fst  $ filter (\(_,x) -> (not x)) $ L.zip uCombinations overlap -- Show the non-overlapping combinati


-- source: https://rosettacode.org/wiki/Combinations#Haskell
findUComb :: Int -> [a] -> [[a]]
findUComb m xs = combsBySize xs L.!! m
  where
    combsBySize = L.foldr f ([[]] : L.repeat [])
    f x next = L.zipWith (L.++) (L.map (L.map (x:)) ([]:next)) next


overlapN ::(Eq a, Integral a) => [[a]] -> Bool -- For a list of list, check if there are duplicate elements
overlapN []           = error "empty list"
overlapN [yss]        = False
overlapN (xs:ys:yss) = any (`elem` xs) ys || overlapN  ((xs L.++ ys) : yss)
```

Appendix D

Clash code for Hardware Generation

D.1 ActorBuffering module

```
module Utils.HWConfig.ActorBuffering where
    import Clash.Prelude
    import qualified Data.List as L
    import SysLift


    bufReplace
      :: KnownNat m
      => KnownNat n
      => Enum i
      => Vec m i
      -> Vec m a
      -> Vec n a
      -> Vec n a
    bufReplace addr vals  old = foldl (\z (x,y) -> replace x y z) old tup
      where
        tup = zipWith (\x y -> (x,y)) addr vals

    bufReplace'
      :: KnownNat m
      => KnownNat n
      => Enum i
      => Vec m i
      -> Vec m a
      -> Vec m Bool
      -> Vec n a
      -> Vec n a
    bufReplace' addr vals en old = foldl (\z (x,y,en) -> if en then replace x y z else z) old tup
      where
        tup = zipWith3 (\x y en -> (x,y,en)) addr vals en


    actorBufV
      :: forall n m . (KnownNat n, KnownNat m)
      => Vec m (Index n, Index n)
      -> BufferState n m
      -> ((Vec m Bool,Vec m DataType),Bool)
      -> (BufferState n m ,Vec m DataType)
    actorBufV pRangeV (wPtrsV,rPtrsV,buf) ((wEnV,wDataV),rEn) =  ((wPtrsV',rPtrsV',buf'),rDataV)
      where
        wPtrsV' = zipWith (\wEn ((min,max),wp) -> incr min max wp wEn) wEnV (zip pRangeV wPtrsV)
        rPtrsV' = zipWith (\rEn' ((min,max),rp) -> incr min max rp rEn') (repeat rEn) (zip pRangeV rPtrsV)

        incr min max p True = if p == max then min else p + 1
        incr min max p False = p
        buf' = bufReplace' wPtrsV wDataV wEnV buf
        rDataV = map (buf !!) rPtrsV


    funcWrapper
      :: KnownNat n
      => KnownNat m
      => KnownNat p
      => 1 <= n
      => 1 <= m
      => Vec n (BitVector (CLog 2 m))
      -> Vec n DataType
      -> Vec m Bool
      -> Bool
      -> Vec m (Index p,Index p)
```

```
      -> BufferState p m
      -> (BufferState p m ,Vec m DataType)
    funcWrapper cbControlV dataInV enableV rEn range  initS = out
      where
        vals = mergeV $ map demux (zip dataInV cbControlV)
        input = ((enableV, vals),rEn)
        out = actorBufV range initS input

    mergeV ::  (KnownNat n, KnownNat m, Ord a, 1 <= m, 1 <= n) => Vec m (Vec n a) -> Vec n a
    mergeV = (map maximum) . transpose


    demux :: forall m a . (Num a, KnownNat m, 1 <= m)  =>  (a,BitVector (CLog 2 m)) -> Vec (m) a
    demux (val,control) = replace control val $ repeat 0 :: Vec (m) a


    bufWrap pRangeV initS = mealy (\s  (x,y,z,a) -> funcWrapper x y z a pRangeV s ) initS
```

D.2 CrossBar module

```
module Utils.HWConfig.CrossBar where
    import Clash.Prelude
    import Clash.Sized.Internal.BitVector

    mux'
      :: a
      -> (Bit,a)
      -> a
    mux' y (b,x)  = if b == low then x else y


    crossBar2d
      :: KnownNat n
      => KnownNat m
      => 1 <= n
      => 1 <= m
      => (SNat n, SNat m)
      -> Vec n a
      -> Vec m (BitVector (CLog 2 n))
      -> Vec m a
    crossBar2d  constraints inputs control = map (crossBar1d inputs) control
      where
        crossBar1d inputs control1d  =
          foldl mux' (head inputs)  $ zipWith (\x y -> (x,y))  (bv2v (muxControl control1d)) (tail inputs)


    muxControl
      :: forall n . (KnownNat n)
      =>  1 <= n
      => BitVector (CLog 2 (n)) -> BitVector (n-1)
    muxControl x = shiftR (maxBound :: BitVector (n-1)) (fromIntegral . toInteger# $ x )
```

D.3 Scheduler module

```
module Utils.HWConfig.Scheduler where

import Utils.DataDependency
import Clash.Prelude
import Clash.Sized.BitVector
import SysLift

import qualified Data.List as L
import Data.Maybe

pCntWrap
  :: forall gated synchronous domain n a
   . HiddenClockReset domain gated synchronous
  => KnownNat n
  => Ord a
  => Vec n a
  -> Signal domain (Index n)
pCntWrap f = s
```

```
    where
      maxBootTime = $(lift $ maximum $ L.map getBootTime (getActors (ddGraph :: DDGraph Char Int)) )
      incWrp limit val = if val == limit then maxBootTime else val + 1
      len = resize (((fromInteger . toInteger . length $ f) :: Index (n+1)) - 1) :: Index n
      s = register (0 :: Index n) (incWrp len <$>  s )

-- Prevent Lift-ception -- generation of Enable schedule
fireSchedule = L.map (\x -> map (elem x . snd) meh ) $ L.take len $ L.iterate (+1) 0
  where
    len = maximum . L.concat $ L.map snd $ toList meh
    meh = $(listToVecTH $ L.map (`findActorFiringTimes` ((findLCM ddGraph :: Int) + maximum (L.map getBootTime (getActo


-- Returns the cb input control vector of the actor that is given as argument.
actorToBin a = pack . frJ $ elemIndex a uniqueProducersV
  where
    frJ (Just a) = a
    frJ Nothing  = 0 -- default control value for crossbar. Can be optimized to keep latest value to save energy.
    uniqueProducersV = $(listToVecTH (L.sort $ uniqueProducers :: [DDActor Char Int])) -- the ports that are required o


-- concat is required for compaitibility with the Clash
(cbBinary,cbLen,cbSchedule,filtered) = (L.concat cbSchedule,L.head cbSchedule,cbSchedule,filtered)
  where
    -- Converts the dependencies to a list of list with controlvectors.
    cbSchedule = L.map (L.foldl1 (L.++)) $  L.map (L.map (L.map actorToBin)) filtered
    -- Checks actor activation for all actors over the whole fireschedule, merges the dependencies to a list with depend
    filtered = L.map (\x -> mergeNlists $ filter (not . null )
        $ L.map producerActivation (L.zip (toList x) (getActors ddGraph))) fireSchedule

--Merges requirement vectors of all dependent vectors ,'-' is a don't care value, and can not be chosen as name for an a
mergeNlists = L.foldl1 merge2lists
merge2lists listA listB = L.map (L.map (uncurry repl) . zipped) [0..(lenA-1)]
  where
    repl (DDActor '-' 0 0) (DDActor '-' 0 0) = DDActor '-' 0 0
    repl (DDActor '-' 0 0) b = b
    repl a (DDActor '-' 0 0) = a
    lenA = L.length listA
    zipped x = L.zip (listA L.!! x) (listB L.!! x)

--Checks if an enabled actor is required by other actors, otherwise it is a don't care
-- Returns a lists of lists where the actor is required exactly.
producerActivation (e, a) = L.map (L.map ((\x ->if x && e then a else DDActor '-' 0 0) .  (a `elem`)) . snd ) cbInfo


sysConfigConstraints =
  (lengthS $(listToVecTH (uniqueProducers :: [DDActor Char Int]))
  ,lengthS $(listToVecTH $  L.replicate (L.sum $ L.map (L.length . snd) cbInfo) (0 :: Int)))
```

D.4 IF module

```
{-# LANGUAGE DeriveAnyClass        #-}
{-# LANGUAGE DeriveGeneric         #-}

module Utils.HWConfig.IF where
  import Clash.Prelude
  import GHC.Generics
  import Control.DeepSeq

  import qualified Data.List as L

  type Request a = Maybe a
  type Ack = Bool

  data State = Idle | Req | Ack  deriving (Eq,Show,Generic,NFData,ShowX)

  class NCL a where
    allHigh :: a -> Bool
    allLow :: a -> Bool

  instance NCL Bool where
    allHigh = id
    allLow = not

  instance NCL (Maybe a) where
    allHigh Nothing = False
    allHigh (Just _) = True

    allLow Nothing = True
    allLow (Just _) = False

  instance (KnownNat n, NCL a) => NCL (Vec n a) where
    allHigh = foldl (\ x y -> x && allHigh y) True
```

```
33        allLow = foldl (\ x y -> x && allLow y) True
34
35
36
37      -- Handshake module that sends data from one producer to n different consumers
38      handshakeTVIF :: (KnownNat n ) => (State,Request a) -> (Request a,Vec n Ack) -> (State,Request a)
39      handshakeTVIF (state,req) (dataIn,ackV) = (state',req')
40        where
41          req'   = case state of  Idle -> Nothing
42                                  Req  -> req
43                                  Ack  -> Nothing
44          state' = case state of  Idle -> case dataIn of
45                                    Nothing -> Idle
46                                    _       -> Req
47                                  Req -> if allHigh ackV then Ack else Req
48                                  Ack  -> if allLow ackV then Idle else Ack
49
50      -- Handshake module that sends data from one producer to one consumer
51      handshakeTIF :: State-> (Request a,Ack) -> (State,Request a)
52      handshakeTIF state (dataIn,ack) = (state',req)
53        where
54          req    = case state of  Idle -> Nothing
55                                  Req  -> dataIn
56                                  Ack  -> Nothing
57          state' = case state of  Idle -> case dataIn of
58                                    Nothing -> Idle
59                                    _       -> Req
60                                  Req  ->   if ack then Ack else Req
61                                  Ack  ->   if not ack then Idle else Ack
62
63
64      -- Handshake module that receives data from several producers to one consumer
65      handshakeRVIF :: (KnownNat n ) => (State,Ack,Vec n (Request a)) -> Vec n (Request a) -> (State,Ack,Vec n (Request a))
66      handshakeRVIF (state,ack,dataOutV) reqV = (state',ack',dataOutV')
67        where
68          (ack',dataOutV')   = case state' of Idle -> (False,dataOutV)
69                                              Req  -> (True,reqV)
70                                              Ack -> (True,dataOutV )
71          state' = case state of  Idle -> if allHigh reqV then Req else Idle
72                                  Req  -> Ack
73                                  Ack -> if allLow reqV then Idle else Ack
74
75
76      -- Handshake module that receives data from one producer to one consumer
77      handshakeRIF :: State-> Request a -> (State, (Ack, Request a))
78      handshakeRIF state req = (state',(ack',dataOut'))
79        where
80          (ack',dataOut')   = case state' of Idle -> (False,Nothing)
81                                             Req  -> (True,req)
82                                             Ack  -> (False,Nothing)
83          state' = case state of  Idle -> case req of
84                                    Nothing -> Idle
85                                    _       -> Req
86                                  Req  -> case req of
87                                    Nothing -> Ack
88                                    _       -> Req
89                                  Ack  ->   Idle
90
91      -- Instance of a TIF
92      hsT = mealy handshakeTIF Idle
93
94      -- Instance of a RIF
95      hsR = mealy handshakeRIF Idle
96
97      -- Instance of a TVIF
98
99
100     -- Instance of a RVIF
101     hsRV = moore handshakeRVIF id (Idle, False, replicate d2 Nothing) --(\(_,x,y)-> (x,y))
102
103     -- Test of TIF
104     hsTTest = simulate hsT [(Nothing,False),(Just 5, False),(Just 5, False),(Just 5,True),(Just 5, False)]
105
106     --  Test of RIF
107     hsRTest = simulate hsR [Nothing,Just 5,Just 5,Just 5, Just 5, Nothing, Nothing]
108
109     -- Test of TIF --> RIF
110     hsSeqTest = simulate hsSeq $ L.replicate 20 (Just 5)-- [Just 5,Just 5,Just 5, Just 5]
111       where
112         hsSeq i = o
113           where
114             (ack,o) = unbundle $ hsR t
115             t = hsT $ bundle (i,ack)
116
117     -- Test of RVIF
118     testHsRV x = L.take x $ simulate hsRV vs
119       where
120         vs = [ll,  hl,  hh,  lh,  ll]
121         ll = replicate d2 Nothing
```

```
122         hl = Just 5 :> Nothing :> Nil
123         lh = Nothing :> Just 5 :> Nil
124         hh = Just 5 :> Just 5 :> Nil
125
126
127     -- Test of 2 x TIF to a RVIF
128     hsVTest = L.take (L.length vs) $ simulate hsV vs
129       where
130         vs = [hl,  hh, lh, lh, ll,ll,ll]
131         ll = (Nothing,Nothing)
132         hl = (Just 5,Nothing)
133         lh = (Nothing,Just 5 )
134         hh = (Just 5,Just 5)
135         hsV i = bundle (co,s) -- connect A and B to C
136           where
137             (ai,bi) = unbundle i
138             ao = hsT $ bundle (ai,ack)
139             bo = hsT $ bundle (bi,ack)
140             -- ack = pure True
141             ci = liftA2 (:>) bo $ liftA2 (:>) ao (pure Nil)
142
143             (s,ack,co) = unbundle $ hsRV ci
144
145
146
147
148
149
150
```

Appendix E

User adjustable Clash code

E.1 SysLift module

```
module SysLift where
import Prelude

import qualified Clash.Prelude as CP

import qualified Data.Map as M
import Utils.HaskellDataflow.DataFlow
import Utils.HaskellDataflow.Graph
import Utils.HaskellDataflow.SVGWriter
import Data.Ratio
import Data.List
import Utils.DataDependency

hsdfNode l ex = (l,HSDFNode l ex)
csdfNode l ex = (l,CSDFNode l ex)


dataflowGraph = Graph (M.fromList
  [hsdfNode 'a' 1 -- adder
  ,hsdfNode 'b' 1 -- adder
  ,hsdfNode 'c' 1 -- adder
  ,hsdfNode 'd' 1 -- adder
  ])
  (
   [
     SDFEdge 'a' 'b' 1 1 1
    ,SDFEdge 'b' 'c' 0 1 1
    ,SDFEdge 'c' 'd' 0 1 1
    ,SDFEdge 'd' 'a' 0 1 1
  ])


dfSchedule = strictlyPeriodicSchedule dataflowGraph

printDFSchedule = printSchedule dfSchedule

ddEdges = [DDEdge 'a' 'b', DDEdge 'b' 'c', DDEdge 'c' 'd',DDEdge 'd' 'a']

genDDgraph edges Nothing = error "Empty Graph"
genDDgraph edges (Just mmap) = DDGraph actors' edges
  where
    actors' = if minTime < 0 then map f actors else actors
      where
       f (DDActor l p st) = DDActor l p (st - minTime)
       minTime = minimum $ map getBootTime actors
    actors = M.elems (M.mapWithKey p mmap) where
      p l (st,period) = DDActor l  period' st' where
        st'     | denominator st == 1     = fromInteger (numerator st)
                | otherwise               = 0 -- no support for fractions
        period' | denominator period == 1 = fromInteger (numerator period)
                | otherwise               = 0 -- no support for fractions


ddGraph = genDDgraph ddEdges dfSchedule

actorList = getActors ddGraph


-- this list constains all the info we need to derive a schedule.
cbInfo = filter (\(_,x) -> not . null $ x ) $ map res actorList where
```

```
61    res l = (l,f) where
62      func = findInputsOutputsCrossbar ddGraph l
63      notEmpty = not . null . fst $ func
64      f = if notEmpty then map (map fst) $ snd func else []
65      -- s = if notEmpty then map (map snd) $ snd func else []
66
67
68  uniqueProducers = nub $ foldl1 (++) $ map (foldl1 (++)) producers
69    where
70      producers = map snd cbInfo
71
72
73  -- Buffering
74  type DataType = CP.Unsigned 7
75  type BufferState n m  = (CP.Vec m (CP.Index n), CP.Vec m (CP.Index n),  CP.Vec n DataType)
76
77
78  findBufferSizes f = map getTokens $ filter
79    (\x -> getActorName f == getProducer x && elem (getConsumer x) producers)  $ edges dataflowGraph
80      where
81       producers =
82         map getEdgeProducer $ filter (\x -> getActorName f == getEdgeConsumer x ) ddEdges
83       getProducer (SDFEdge a b c d e)= a
84       getConsumer (SDFEdge a b c d e)= b
85       getTokens (SDFEdge a b c d e)= c
86
87  pRange :: [Int] -> [(Int,Int)]
88  pRange ls = snd $  mapAccumL (\max size -> (max+size,(max,max+size-1))) 0 ls
89  ptrs = map fst
90
91
92  crossBarInfo = zipWith ( \(x,y) z -> (x,y,z) ) cbInfo width
93    where
94     width = map (length . snd) cbInfo
```

E.2 SysConfig module

```
1   module SysConfig where
2
3   import Clash.Prelude
4   import Clash.Explicit.Testbench
5
6   import SysLift
7   import Utils.DataDependency
8   import Utils.HWConfig.CrossBar
9   import Utils.HWConfig.IF
10  import Utils.HWConfig.ActorBuffering
11  import Utils.HWConfig.Scheduler
12
13  import qualified Data.List as L
14  import Utils.HaskellDataflow.SVGWriter
15
16  import Random
17
18  topEntity
19   :: Clock System Source
20   -> Reset System Asynchronous
21   -> Signal System (DataType)
22  topEntity = exposeClockReset actorD
23  {-# NOINLINE topEntity #-}
24
25
26  testBench :: Signal System (DataType)
27  testBench = done
28    where
29      done         = topEntity clk rst
30      clk          = tbSystemClockGen (not <$> pure True)
31      rst          = systemResetGen
32
33
34
35  -- User defined functions
36  act
37    :: HiddenClockReset domain gated synchronous
38    => DataType
39    -> Signal domain Bool
40    -> Signal domain DataType
41  act i en = s
42    where
43      s = regEn (1) en (s+1)
44
45
46  adder
```

```
 :: HiddenClockReset domain gated synchronous
 => Signal domain (Bool,DataType)
 -> Signal domain DataType
adder = mealy f 0
  where
    f s (en,i)  = (s',o)
      where
        s' = if en then i + 1 else s
        o = s

actSort
 :: HiddenClockReset domain gated synchronous
 => KnownNat n
 => 1 <= n
 => Signal domain (Vec n DataType)
 -> Signal domain Bool
 -> Signal domain (Vec n DataType)
actSort i en = mux en (id <$> i) (pure (repeat 0))

actDev
 :: HiddenClockReset domain gated synchronous
 => KnownNat n
 => 1 <= n
 => Signal domain (Vec n DataType)
 -> Signal domain Bool
 -> Signal domain (Vec n DataType)
actDev i en = mux en (id <$> i) (pure (repeat 0))

-- CrossBar Schedule
cBSchedule = unconcat cBNewCLengthS cBNew
  where
    cBNew = $(listToVecTH cbBinary)
    cBNewLengthS = lengthS $(listToVecTH cbSchedule) -- number of scheduling samples (same as enable vector)
    cBNewCLengthS = lengthS $(listToVecTH cbLen) -- control vector length

cbOut = liftA2 (crossBar2d sysConfigConstraints) actorDABC crossBarCtrl -- inputs actors and control vector

-- Scheduler
fireScheduleV = $(listToVecTH fireSchedule)

pCnt = pCntWrap fireScheduleV

actorEn = asyncRom fireScheduleV <$> pCnt
crossBarCtrl = asyncRom cBSchedule <$> pCnt


actorDABC = sequenceA (actorD:>actorA:>actorB:>actorC:>Nil)

actorA = adder $ bundle  ((!! 0) <$> actorEn,(!! 1) <$> cbOut)
actorB = adder $ bundle  ((!! 1) <$> actorEn,(!! 2) <$> cbOut)
actorC = adder $ bundle  ((!! 2) <$> actorEn,(!! 3) <$> cbOut)
actorD = adder $ bundle  ((!! 3) <$> actorEn,(!! 0) <$> cbOut)
```

Appendix F

System derivation examples

F.1 Example: Processing pipeline

Figure F.1 depicts an example where four actors are chained after each other. For simplicity the actors all represent the increment function. The function can be described by the CλAsh description in Listing 14. The function has a enable signal and a port to receive data as input. Next, in order to derive a schedule a valid dataflow graph and list of data dependency edges must be specified in the `SysLift` Module, see Listing 15. The amount of initial tokens can be varied, and will influence the behaviour as will be shown later. For now, only one token between δ_{ab} is initialized. All other edges do not contain tokens.

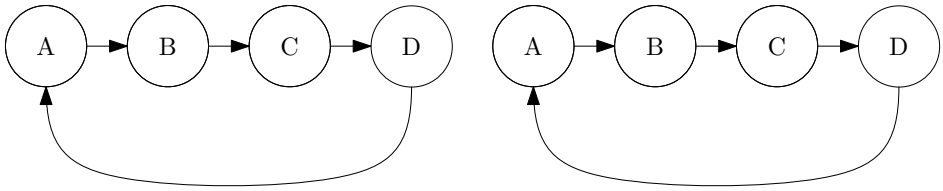


Figure F.1: Processing pipeline data flow graph (tokens not shown as they vary through this example), left the dataflow graph and right the data dependency graph

```
1  adder
2   :: HiddenClockReset domain gated synchronous
3   => Signal domain (Bool,DataType)
4   -> Signal domain DataType
5  adder = mealy f 0
6    where
7      f s (en,i)  = (s',o)
8        where
9          s' = if en then i + 1 else s
10         o = s
```

Listing 14: Increment function with internal state

```
dataflowGraph = Graph (M.fromList
  [hsdfNode 'a' 1 -- adder
  ,hsdfNode 'b' 1 -- adder
  ,hsdfNode 'c' 1 -- adder
  ,hsdfNode 'd' 1 -- adder
  ])
  (
   [
     SDFEdge 'a' 'b' 1 1 1
    ,SDFEdge 'b' 'c' 0 1 1
    ,SDFEdge 'c' 'd' 0 1 1
    ,SDFEdge 'd' 'a' 0 1 1
  ])

ddEdges = [DDEdge 'a' 'b', DDEdge 'b' 'c', DDEdge 'c' 'd',DDEdge 'd' 'a']
```

Listing 15: Dataflow graph and dependency edges

A valid strictly periodic schedule can be derived from the aforementioned dataflow graph. The schedule along with the dependency edges can be verified by running the function `ddGraph`, this will give the following output:

```
*SysConfig> ddGraph
Actors:
[DDActor:'a', period: 4, Start time: 3;
,DDActor:'b', period: 4, Start time: 0;
,DDActor:'c', period: 4, Start time: 1;
,DDActor:'d', period: 4, Start time: 2;
]
Edges:
['a'-->'b';
,'b'-->'c';
,'c'-->'d';
,'d'-->'a';
]
```

The resulting hardware architecture should look like Figure F.2. This requires manual nesting of the functions. This paragraph will guide the reader through the progress of establishing the desired hardware architecture. The difficult part in connecting the components is, that they have intertwined dependencies. The firing schedule of the actors and the schedule of the crossbar can be accessed by running `fireScheduleV` or `cBSchedule`, respectively. The `Signal pCnt` is the program counter that selects the desired record from the schedules.

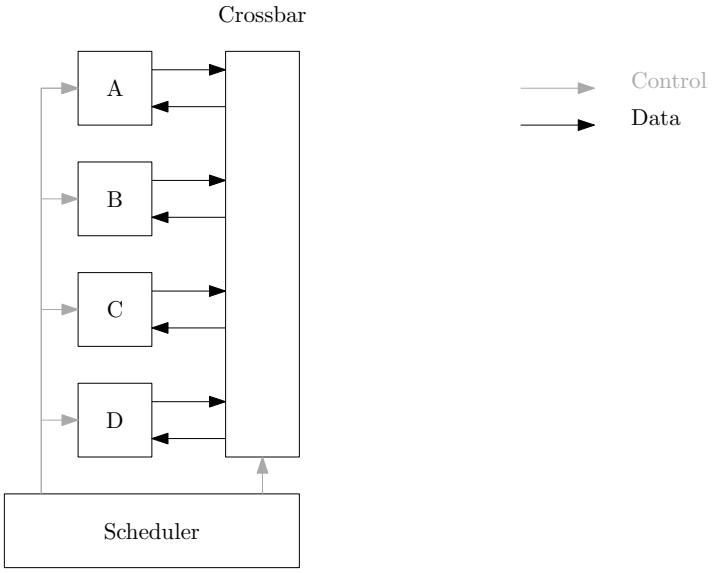


Figure F.2: Connection diagram of the functional blocks

```
1  fireScheduleV = $(listToVecTH fireSchedule)
2
3  pCnt = pCntWrap fireScheduleV
4
5  actorEn = asyncRom fireScheduleV <$> pCnt
6  crossBarCtrl = asyncRom cBSchedule <$> pCnt
```

The `Signal actorEn` now presents which actor can fire at a certain moment in time by means of a vector of booleans. The `Signal crossBarCtrl` holds the configuration of the crossbar of time. Next, the enable signals for the individual actor must be split from `actorEn`, and be connected to the `adder` function. The other input to the `adder` function comes from the output of the crossbar (`cbOut`). The output of the crossbar has four ports, that have to be connected to the according actors. The order in which the actors are placed depend on the order in which the data dependencies were defined. The order can be observed by running `crossBarInfo`:

```
*SysConfig> crossBarInfo
[(DDActor:'a', period: 4, Start time: 3;
,[[DDActor:'d', period: 4, Start time: 2;
]],1),(DDActor:'b', period: 4, Start time: 0;
,[[DDActor:'a', period: 4, Start time: 3;
]],1),(DDActor:'c', period: 4, Start time: 1;
,[[DDActor:'b', period: 4, Start time: 0;
]],1),(DDActor:'d', period: 4, Start time: 2;
,[[DDActor:'c', period: 4, Start time: 1;
]],1)]
```

The output is difficult to read, but it consists of a list that contains three field tuples. Within the tuple, the first record represents the consuming actor (D, A, B and C in this case). The second field is a list of lists. It specifies how many ports are connected to the consumer, and which values the ports can take over time. The last field can be derived from the second field, and indicates the with of a the port. This is useful in connecting the crossbar to consuming functional blocks. This is all the information that is required to fully connect the `adder` functions:

```
actorA = adder $ bundle  ((!! 0) <$> actorEn,(!! 1) <$> cbOut)
actorB = adder $ bundle  ((!! 1) <$> actorEn,(!! 2) <$> cbOut)
actorC = adder $ bundle  ((!! 2) <$> actorEn,(!! 3) <$> cbOut)
actorD = adder $ bundle  ((!! 3) <$> actorEn,(!! 0) <$> cbOut)
```

The output of the crossbar (`cbOut`) has not been defined yet. To generate a crossbar, one needs to give the number of inputs and outputs (`sysConfigConstraints`) to the `crossBar2d` function. Then the crossbar also has a control signal (`crossBarCtrl`) and it requires the inputs of all the actors that produce data. The order in which the inputs are given is the same as for the crossBar output (D, A, B and C). Furthermore, the `crossBar2d` function does not natively operate in the `Signal` domain, and needs to be lifted.

```
cbOut = liftA2 (crossBar2d sysConfigConstraints) actorDABC crossBarCtrl
actorDABC = sequenceA (actorD:>actorA:>actorB:>actorC:>Nil)
```

If the code in the `SysLift` and `SysConfig` module are adjusted according to the described steps, then the hardware architecture could be simulated. The whole graph functions as a topentity. Therefore the designer must decide which actors should be available, e.g. Actor D:

```
topEntity
 :: Clock System Source
 -> Reset System Asynchronous
 -> Signal System (DataType)
topEntity = exposeClockReset actorD
{-# NOINLINE topEntity #-}


testBench :: Signal System (DataType)
testBench = done
  where
    done         = topEntity clk rst
    clk          = tbSystemClockGen (not <$> pure True)
    rst          = systemResetGen
```

Evaluating the testbench will now give the outputs over time:

```
*SysConfig> sampleN 10 testBench
[0,0,0,3,3,3,3,7,7,7]
```

If the amount of initial tokens is changed, e.g. to $\delta_{AB} = 1$, $\delta_{BC} = 1$, $\delta_{CD} = 1$ and $\delta_{DA} = 1$, then the actors will fire differently, which then again results in different simulation results:

```
*SysConfig> sampleN 10 testBench
[0,1,2,3,4,5,6,7,8,9]
```

At the current implementation, it is not always possible to derive hardware from a valid dataflow schedule. if the start times or period of the tokens is a fraction (which can be observed by running `ddgraph`), then it is not possible to generate a schedule. This is due to the constraint that one time unit equals one clock period, and this is an indivisible element.

F.1.1 RTL-level comparison for different schedules

In this section, the influences on the resulting hardware, due to a different schedule are examined in Vivado 2017.4 for the Zybo Z7-20 platform. Verilog is generated by Cλash 0.99.3. The ”Run Implementation” process is ran (which maps the hardware on the FPGA), and then the results are examined. A clock of 100 MHz was used as timing constraint. The aim was to perform a power analysis based on the SAIF file that was obtained, by running the testbench that was generated by Cλash. However, the design was so small that the power consumption was neglected by the Idle usage of the FPGA. The results of the two schedules are as follows:

1. One initial token on the δ_{AB}: 37 Look-Up Tables, 33 Flip-flops
2. One initial token on all edges: 20 Look-Up Tables, 35 Flip-flops

From behavioural level, the presented design does not allow many differences is resource allocation, because there are no crossbar optimizations available. However, design 1 seems to take up almost twice the amount of LUTs, compared the design 2. It is likely that the latter is more simple because all the actors do fire at every cycle, whereas the first design fires according to a more difficult schedule. The exact source of the problem is not clear, since it could be either an optimization from the Cλash-compiler or from the synthesis tool. It is also impossible to re-allocate the individual blocks from the generated RTL-views, which makes it more difficult to find the cause of the optimization.