Chapter 1

Introduction

This introduction contains the project goals and some very short introductions on the FPGAs and how they are used, numerical methods for approximating ODEs, functional programming and CλaSH. The goal of this report is to be understandable and reproducible for any technical BSc student and therefore the short introductions are included for the sake of completeness and are assumed knowledge for the rest of the thesis.

1.1 Project goals

From the start, this project has had two main goals: firstly, obtaining information on the feasibility and the advantages and disadvantages of performing numerical mathematics (numerical approximations to ODEs) directly on (programmable) hardware, the FPGA. Secondly: figuring out whether higher-order functions are of much use for numerical mathematics. As per usual, having main goals spawns off several minor goals which support the main parts. Both supporting goals are about simplifying the process of configuring FPGAs: an easy way of setting up projects with complicated IO requirements and furthermore, developing a toolchain integration which turns the long process of compiling and deploying your FPGA project into the execution of a single command.

Alongside these concrete goals the underlying theme is to do as much work as possible in CλaSH, a library and compiler based on the functional programming language Haskell, developed by Christiaan Baaij at the CAES group at the University of Twente. Further elaboration on CλaSH can be found in section 1.5.

1.2 FPGAs

1.2.1 What is an FPGA?

An FPGA (Field Programmable Gate Array) is a chip in which you can specify the hardware yourself. In contrast to regular programming in which you generate a long list of instructions which are executed sequentially on a fixed chip configuration, the FPGA allows you to specify exactly which wire (signal) leads where and what operation should be applied to that signal.

This approach to programming can have several advantages. The first one arises from the large opportunities for parallelism. Every part of the FPGA can be executing a meaningful computation simultaneously, whereas processors are bound by the amount of physical cores they have in the amount of truly concurrent instruction executions possible. Secondly, a conventional processor only has a fixed instruction set. Using an FPGA you can define your own instructions (subcircuits), again providing a possible improvement in computational speed. According to [10], FPGAs were already capable of outperforming CPUs on very parallelizable numerical tasks on single and double precision floating point numbers in as early as 2004. Furthermore, as you are implementing your signal processing directly in hardware, there will be a fixed bound on the possible latency. This makes FPGAs ideal for purposes in high-throughput, low-latency signal processing, eg. real-time audio, video or data stream processing. Lastly, the reconfigurability of FPGAs whilst remaining close to the actual hardware allows for cost reductions in the verification of ASIC (Application-Specific Integrated Circuits) designs. It's cheaper to reprogram your FPGA than to have a new version of an ASIC manufactured.

However, the FPGA is a trade-off between implementing designs directly in hardware and being able to run multiple designs (after a reconfiguration). As a consequence of this, it still loses to ASICs with several orders of magnitude on performance [4] and CPUs still dominate in terms of versatility and on-the-fly reconfigurability.

1.2.2 How does it work?

An FPGA is built up from several distinct element types, depicted in figure 1.1

1. *Logic elements* Responsible for the actual signal processing. An FPGA may contain different types of logic elements, eg. memory, DSP and logic blocks. These blocks implement some signal processing capability which can be configured up to certain limits.
2. *Programmable interconnects* In order to be able to represent complex designs, the logic elements need to be connected in a certain way. This is what the programmable interconnects are for. Essentially, those are wires which can be turned on and off by the user as part of a design specification.
3. *IO blocks* Finally, the functionality implemented using the logic elements and programmable interconnects should be exposed to external signals in order to be useful. IO blocks can be used to control hardware pins, controlling a LED or reading a switch but also for more intricate IO facilities, eg. external memory controllers.

It might seem that programming an FPGA involves manually specifying the interconnects and exact configurations of the logic elements. However, specialized languages have been developed just for the purpose of describing the FPGA functionality at a higher level of abstraction and leave the specific routing and assignment of logic elements to the compiler. The two main advantages of these languages are that you do not have to worry about low-level problems like how the interconnects will be routed and secondly, your written specification will be portable across multiple FPGA vendors as long as the vendor supplies you with the proper compiler from your specification to a file which can be used to program the FPGA.

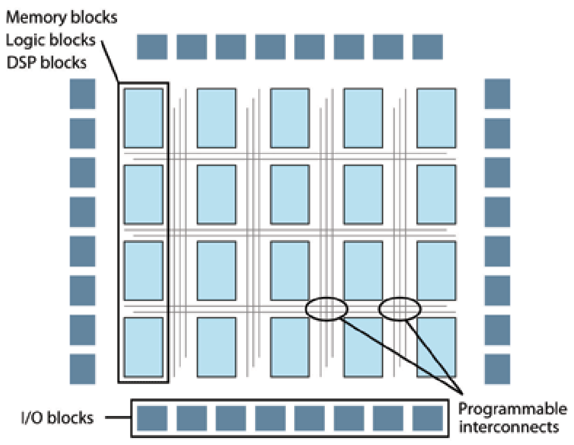

Fig. 1.1 FPGA fabric.

More information on FPGA functionality can be found in [11].

1.2.3 System-on-a-chip

FPGAs in itself can be useful, but especially for design with IO requirements that are more complex than just reading out hardware switches and controlling LEDs, more control is needed. This requirement has led to the rise of SoCs (System-on-Chip). These devices integrate an FPGA with additional hardware on a single chip. This extra hardware usually contains a CPU, which can be used to simplify the process of loading and extracting data from the FPGA. The SoC used for this thesis is the Terasic SoCKit, a development kit containing an Altera Cyclone V FPGA and a dual-core ARM A9 CPU on a single chip (Altera 5CSXFC6D6F31C6N), alongside a wide variety of IO possibilities. Further information on the SoC used is available at [5].

1.3 Numerical solvers for ODEs

The field of numerical solvers is a vast and active area of research. Furthermore, there is a lot of theory on which solver to pick for specific problems, related to stability, computational efficiency and other factors. However, the goal of this thesis is to get an impression of the feasibility of using numerical solvers for ODEs on FPGAs. Therefore, the selection of solvers will be restricted to some very basic schemes.

The solvers used have some common properties:

- *Fixed-step* The solvers compute a value for the ODE at a fixed step size. This means that in contrary to the continous 'mathematical' solution of an ODE, the output of the solver will be an approximate to the actual value of the solution at the discrete set of values of the independent variable, in which the difference between the values of the independent variables is defined by the fixed step size.

- *Single-step* The approximate value of the ODE x_k at t_k only depends on t_{k-1}, x_{k-1} and the ODE that is being approximated.

Euler

The easiest numerical solver, without doubt, is Euler's method (equation 1.1). For every discrete point in time, the value of the next point is approximately equal to the derivative at that point multiplied by the time step added to the current value. The simplicity of the scheme has a cost: it is not very accurate and the errors accumulate quickly. From [9], the maximum error of Euler's method can be shown to be linear in the time step and exponential in the interval length (eq 1.2), in which M and L are constants depending on the equation to be solved, $b-a$ is the solution interval length and h is the time step.

$$\begin{aligned} s_{k,1} &= f(t_k, x_k) \\ x_{k+1} &= x_k + h s_{k,1} \end{aligned} \tag{1.1}$$

$$\text{maximum error}_{\text{euler}} \leq \frac{Mh}{L}(e^{L(b-a)} - 1) \tag{1.2}$$

Runge-Kutta

The Runge-Kutta methods are a family of solvers, of which the 4th order version is the most well-known (RK4). The solver used here will be a second-order Runge-Kutta method, also known as the *improved Euler's method* [9]. This second order method requires the computation of two slopes (equation 1.3), in contrast to the single one required for Euler's method.

$$\begin{aligned} s_{k,1} &= f(t_k, x_k) \\ s_{k,2} &= f(t_k + h, x_k + h s_{k,1}) \\ x_{k+1} &= x_k + h\frac{s_{k,1} + s_{k,2}}{2} \end{aligned} \tag{1.3}$$

$$\text{maximum error}_{\text{RK2}} \leq \frac{Mh^2}{L}(e^{L(b-a)} - 1) \tag{1.4}$$

Note that the maximum error of RK2 is proportional to the square of the time step (equation 1.4). As the time step decreases by a factor of 2, the maximum error will decrease by a factor of 4, given that the equation and the range stay the same. However, the maximum error still depends exponentially on the interval length.

1.4 Functional programming

1.4.1 What is functional programming?

As the name suggests, the functional programming paradigm uses functions. These functions are used to build up the program and create structure. In contrast to the imperative programming paradigm, there is no assignment - there are only expressions. It is by evaluation of these expressions that you execute your program. These expressions consist of variables, constants and operations. However, the name variable may be ill-chosen, as assignment does not exist and therefore it is impossible for a variable to vary. Once you have bound a value to a certain variable, this value may not change and the value should be the same at every point where this variable is referenced (a concept called referential transparency).

The exclusion of assignment from functional languages has several consequences. It becomes impossible to program using loops. The alternative is the use of recursive (listing 1.1) and higher-order functions (functions that have other functions as parameters or output). Especially higher-order functions have the side effect that they add clarity about the way the program functions (listing 1.2). For instance, a **map** applies the same function to every element in a list, resulting in a new list. A fold would start at an initial value and element-by-element combine the list into a single value using a specified function, eg. addition. Both these operations would be implemented using a for-loop in an imperative language, but their goal is completely different. The availability of higher-order functions allows for more clarity in code by being able to specify exactly what kind of operation you want to perform. Another effect of the lack of assignment is the lack of side effects in functional programming. As there is no way to modify a variable, there is also no way of accidentally modifying a variable such that you enter an invalid state and other parts of the program stop functioning correctly.

Listing 1.1 Recursive functions

```haskell
fac  :: Num a => a -> a
fac 0 = 1
fac n = n * fac(n-1)
```

Listing 1.2 Higher order functions

```haskell
timesTwo :: Num a => [a] -> [a]
timesTwo xs = map (*2) xs

sum :: Num a => [a] -> a
sum xs = foldl (+) 0 xs

powers :: Num a => a -> a -> [a]
powers init power = iterate (\x -> x * power) init

powers100 = take 100 $ powers 1 2
```

The concept of higher-order functions also serves as an introduction to another very important concept in Haskell: the type system. Everything has a fixed type and often times, when only looking at the type definition you can already guess what the function is going to do. Consider the type signature of **map** in listing 1.3. It requires a (a->b), a function to turn something of type a into something of type b. Furthermore, it needs a list of a, [a] (indicated by the square brackets) and it returns something of type 'list of b' ([b]). The type signature of the **foldl** is slightly harder to understand, but it requires a function which needs an a and a b to produce a new b. Furthermore, an initial value (a) and a list of b to operate on ([b]) in order to return the final result, which has again type a. Function type signatures can be daunting to understand at first, but not all of them are as complicated as the **foldl**. For instance, a function which can be used to represent a differential equation, which needs a state of the system (the ODEState) in order to compute the derivative at that point (the D_ODEState). As a last example, the type signature of Solver: it requires some integration scheme, settings for the time (initial time and a time step), it requires an equation and an initial state of the system. All of this combined results in a list of states: the numerical approximation to the solution of the ODE. Note that in this example, the integration scheme Scheme itself is a function, for which understanding the type signature should pose no problem by now.

The last concept in functional programming which is important to understand is so-called lazy evaluation. In contrast to imperative programming, a value is only evaluated whenever it is needed. For instance, this allows for the concept of an infinite list, which is definitely not attainable in imperative programming as it would run out of memory. An example of this is the higher-order function **iterate**, shown in listing 1.2, as part of the powers function. This function generates an infinite list by repeatedly applying a function (in this case a multiplication by a constant) to its own output, starting at the initial value **init**. However, it would be impossible to actually display the entirety of said infinite list and therefore you apply another function, **take** n, which only consumes the first n elements of a list. As the total evaluation only requires the first n elements, these are the only ones that will actually be computed. [8]

Listing 1.3 Type signatures

```
map :: (a->b) -> [a] -> [b]
foldl :: (a->b->a) -> a -> [b] -> a

type Equation = ODEState -> D_ODEState
type Scheme   = TimeSettings -> Equation -> ODEState -> ODEState
type Solver   = Scheme -> TimeSettings -> Equation -> ODEState -> [ODEState]
```

1.4.2 Using FP for numerical mathematics

Functional languages have several properties which make them suitable for the purpose of solving problems in numerical mathematics. First and foremost, Haskell, being based on λ-calculus is very close to mathematics. The useful mathematical properties here are *referential transparency*, easy *partial function application* and being a *declarative language*. Referential transparency implies that a variable only has a constant value which is the same everywhere in the program. This prevents that changing a variable might have influence on another computation as a side effect and it corresponds to mathematical notation. For instance, in an imperative programming like C you could write i = i + 1, which is a mathematical impossibility and therefore not allowed in Haskell. Partial function application is another very useful concept. Often in numerical mathematics, you want to create or process a function. You need a function that has another function as return value. For instance, take a function which requires two arguments. After only applying a single argument, the object returned still needs the second argument in order to compute the final value. This is exactly according to the definition of a function: An object that still needs arguments before being able to return its final value. Being a *declarative language* means that you write code that specifies what you want to accomplish, not how to get there. This concept is again borrowed from mathematics. You put in a set of function definitions and Haskell will figure out how to actually compute the value you request according to those definitions. This property of declarativity also has the result that Haskell is a terse language whilst remaining easy to understand. Secondly, Haskell has a very strong type system. The type system has three main advantages. It becomes very easy to swap out and replace functions as long as you make sure that the types are the same. The Haskell compiler will start to assert errors immediately whenever you feed it something which does not make sense or could be ambiguous which is very useful when writing programs. By having a look at the types of a Haskell program it becomes very straightforward to see what the program does and how it works, which is very useful when attempting to understand your own or someone else's code. Lastly, a property which is often very important for numerical mathematics: Haskell is fast. According to the *Computer Language Benchmarks Game* [2], Haskell is almost on par with Java and Fortran but significantly faster than Python and Matlab (not shown), two languages which are often used for numerical mathematics nowadays. There is still a performance gap of around a factor 3 between Haskell and C (the reference), hence if speed is of the absolute highest concern C is still a valid option.

1.4.3 Example: Numerical solutions of ODEs in Haskell

As mentioned before, the types in Haskell reveal lots of information about the structure and functionality of the program. The three main types constituting the numerical solver for ordinary differential equations are listed in listing 1.4.

Listing 1.4 Main types for the ODE solver

```
type Equation = ODEState -> D_ODEState
type Scheme   = TimeSettings  -> Equation -> ODEState -> ODEState
type Solver   = Scheme -> TimeSettings -> Equation -> ODEState -> [ODEState]
```

Equation

In essence, a differential equation is a mapping (function) from a certain state of the system to the change of this system. This is also what the type signature of Equation signifies, a mapping from an ODEState to a D_ODEState, the change in state or the derivative. This generic set up allows the specification of any ODE for solving. The implementation in pure Haskell of a simple ODE is given in listing 1.5, which corresponds to the equation $x' = -x$. However, this representation is not very elegant and a lot of the code is performing unboxing of the types. Using property that this equation is linear, it is possible to use an utility method which takes as input a matrix and returns the Haskell differential equation function belonging to that matrix. The same can be done for heterogeneous linear systems using a different utility function, which does not only takes a matrix as input but also a list of functions representing the heterogeneous part of the equation.

Listing 1.5 Example equation for exponential decay

```
eq_exponential :: Equation
eq_exponential state      = [−x !! 0]
  where
    x = xs state
```

SolveMethod

The SolveMethod performs the actual computations on what the next value of the solution should be: the integration scheme. In order to obtain this next state, the scheme needs three input values: It needs information on the timing constraints of the solution, in this case it needs the time step. Furthermore, it needs the equation itself and it requires the state of the system at t_n in order to be able to determine the state of the system at $t_{n+1} = t_n + \Delta t$.

The most straightforward integration scheme is called forward Euler, given in equation 1.1. Listing 1.6 depicts the translation of the mathematical expression 1.1 to Haskell. Even though some list operations have been inserted (**zipWith** and **map**), the structure is still recognizable. It computes the change in state, multiplies this with the time step obtained in line 6 and adds the initial state in line 4. Lastly, the integration scheme returns the new state of the equation, consisting of a list of x-values and a corresponding time value. Implementations of different solvers (i.e. 4th order Runge-Kutta) can be found in appendix A.

Listing 1.6 Example code for the Euler integration scheme

```
1  euler :: Scheme
2  euler time equation initState      = ODEState newX newT
3    where
4      newX      = zipWith (+) (xs initState ) dX
5      dX        = map (timestep ∗) (equation initState )
6      newT      = (t initState ) + timestep
7      timestep  = dt time
```

Solver

The Solver function in listing 1.7 acts as the main interface to the program. You specify a SolveMethod, the TimeSettings (containing the time step and the time at which to stop solving), the equation itself and an initial condition. The Solver will then return a list of states of the system. This problem could be solved recursively, but a method featuring more clarity is a higher-order function, in this case **iterate**. This function keeps applying itself to its own output, starting with some specified initial value, generating an infinite list. However, we are not interested in the infinite list of solutions and therefore we only take the first N elements, in which N depends on the initial time, the final time and the time step used for the solution.

The solutions of a wide range of equations, both linear and non-linear, both homogeneous and heterogeneous and using the input matrix utility functions have been plot with suitable initial conditions to show their behavior in figure 1.2.

Listing 1.7 The main controlling function

```
solve :: Solver
solve solvemethod time equation initState = states
  where
    states = take steps $ iterate (solvemethod time equation) initState
    steps = ceiling $ (tMax time - t initState )/ dt time
```

Results

This example of approximating solutions to ODEs using Haskell shows the versatility the language and especially its type system. Without any trouble, it is possible to exchange solver schemes, equations, settings and initial conditions by merely changing a single identifier in the function call. A variety of equations with suitable initial conditions have been approximated: both linear and non-linear, of multiple orders, both homogeneous and heterogeneous. As an example, the equations of which the solutions (with suitable initial conditions) are depicted in figure 1.2 are listed in equations 1.5. The exact Haskell source generating the plot is shown in A, but depends on an external library for generating the plot [1].

$$
\begin{aligned}
&\text{Exponential} && x(t)' = -x(t) \\
&\text{Simple harmonic} && x(t)'' = -x(t) \\
&\text{Cosine hyperbolic} && x(t)' = \frac{\sqrt{x(t)^2 - a^2}}{a} \\
&\text{Simple harmonic} && \vec{\mathbf{x}}(t)' = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{\mathbf{x}}(t) \\
&\text{Simple forced harmonic} && \vec{\mathbf{x}}(t)' = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{\mathbf{x}}(t) + \begin{bmatrix} \sin(t) \\ e^{-t} \end{bmatrix}
\end{aligned}
\tag{1.5}
$$

1.5 CλaSH

Functional programming has several aspects in common with hardware design, which was the original reason for the development of CλaSH, a library and special compiler which is capable of compiling a subset of Haskell into HDL. In CλaSH, every (sufficiently complex) hardware design consists of two parts: a combinatorial part and a synchronous part. It is the combinatorial part which can be modeled with few problems in a functional way: there is no time dependency - the output is merely an evaluation of a certain function of the inputs and for every input the output will be the same. However, complex hardware designs do not merely consist of stateless combinatorial circuits, in order to perform work some statefulness

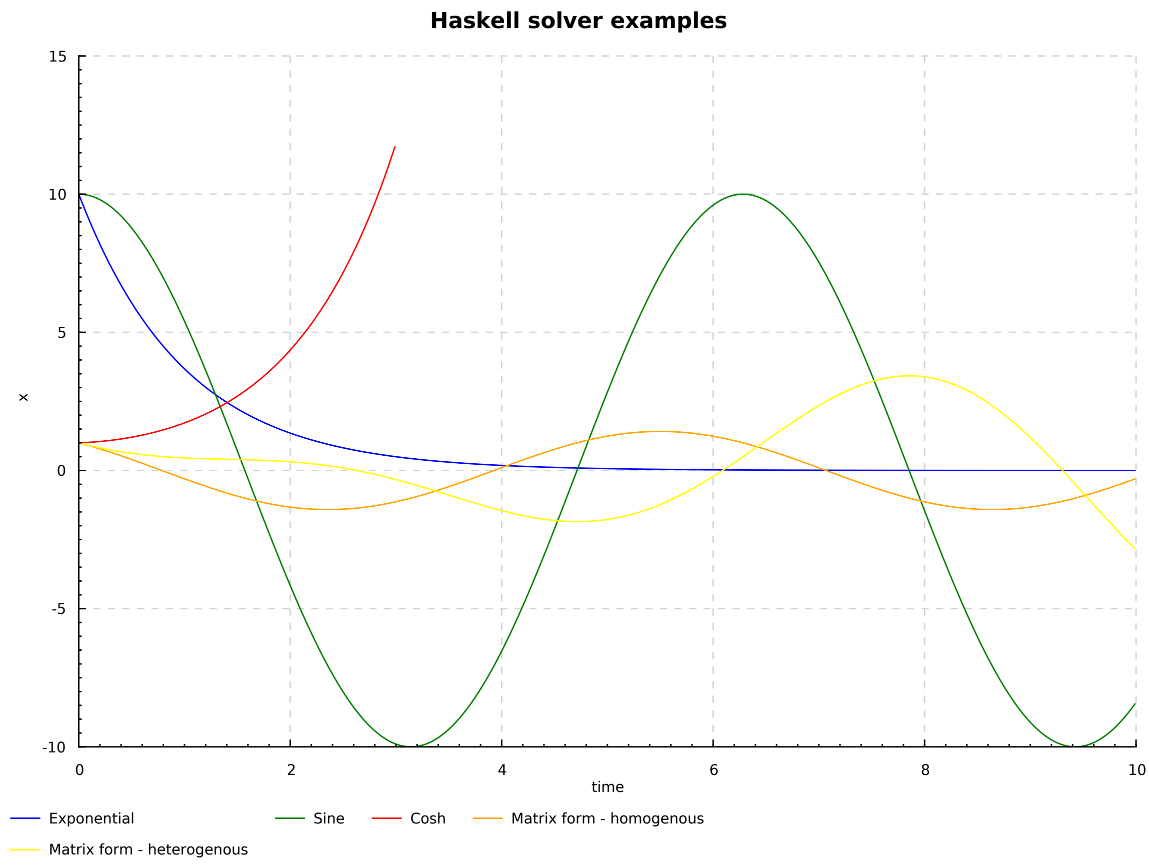

Fig. 1.2 Plots of the ODE solutions, simulated in Haskell.

must be included. This is the task of the synchronous part of the design, keeping track of the state. The state is often implement using latches or memory cells which can get updated with new values, computed by the combinatorial part, every clock cycle (hence the name synchronous part). Together, the combinatorial part and the synchronous part of hardware are called sequential logic: the output depends on both the current input and the past inputs.

1.5.1 Mealy machines

It is the synchronous (statefulness) part that cannot be modeled directly in Haskell, but CλaSH gets around this by use of the Mealy machine [6]. On every clock cycle, based on the input and the current state the combinational logic computes an output and a new state. The updated state gets stored in the memory elements, the output forwarded to the external ports of the hardware. When looking at some CλaSH-Haskell, this structure of generating an output and a new state based on an input and the current state is still clearly visible, for example in listing 1.8. After defining a function with the proper type signature for a Mealy machine you can pass this function to the CλaSH built-in function mealy, which handles the conversion to a valid topEntity. In this case the specified function is multiply-accumulate,

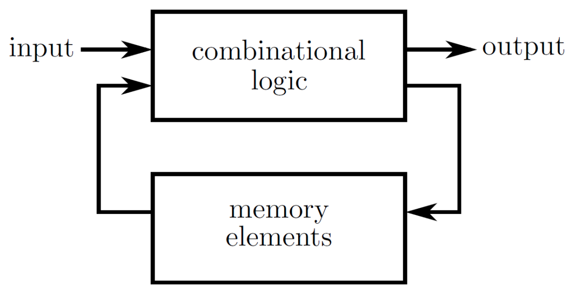

Fig. 1.3 A flowchart of the Mealy machine

which, for every clock cycle, multiplies its inputs together and adds it to an internal sum which gets forwarded to the output. [3]

Listing 1.8 A basic example a multiply-accumulate specification in CλaSH

```
topEntity :: Signal (Signed 9, Signed 9) -> Signal (Signed 9)
topEntity = mealy mac initial

mac state input = (state', output)
  where
    (x,y)  = input        -- unpack the two inputs
    state' = state + x*y  -- the new state
    output = state'       -- output the new state
```

1.5.2 Advantages of CλaSH

The power of CλaSH is manifold. Firstly, you are writing valid Haskell code, which allows you to simulate and verify your designs by relying on existing debugging techniques for Haskell. Secondly, Haskell enables more clarity in code, partially by use of higher-order functions, partially by use of it's record syntax, which allows you to easily group signals together in a meaningful way. This additional clarity is especially useful in complex designs in combination with the modularity of functional programs (as long as you adhere to the type signature), which allows you to easily swap out parts of design. Thirdly, notice that the only location at which the types of the signals has been specified is in the first line (`Signed 9`, a signed integer of length 9). This means that, in order to have our multiply-accumulate circuit work on, for instance, unsigned integers of length 32, the only place that needs modification is the `topEntity` function type signature. The other types are inferred from here.

After specifying your design in CλaSH, you can invoke the CλaSH-compiler to generate HDL. Both mainstream languages (VHDL and Verilog) are supported. It is then the HDL which can be compiled by a vendor-specific compiler into a binary file which can be flashed to the FPGA.

Chapter 2

Methods

This chapter contains a description of the methods used in describing the hardware on the FPGA side using CλaSH and VHDL. The description of the host-side (HPS) programming is included in appendix C, as even though this part of the project is crucial for obtaining results, it is merely tangential to the project goals. The methods are ordered chronologically in order to properly describe the process that lead to the FPGA side programming used in chapter **??**. This approach has as side effect that the explanation of the source code is evaluated lazily: only whenever necessary. Therefore the IO system is only covered after testing and synthesis, as it is not yet needed before.

2.1 Overall structure

In any reasonably complex project it pays off to keep a clear structure: it improves understandability and allows for easier debugging. For simple projects in CλaSH, the structure would be uniquely determined by the HDL generated by CλaSH, but this project also relies on other sources of HDL. Most noteworthy, the interface between the HPS and the FPGA; these signals are very specific to the type of the FPGA and the interconnects it has to the HPS. Luckily, the vendor (Altera) provides a way to generate HDL (the QSys system) in order to create a bridge from the host code running on the HPS to the programmable hardware, the FPGA. This process is described in appendix D. After configuring the bridge, VHDL can be generated and we are left with an instantiable VHDL component. However, even though CλaSH is capable of instantializing external IP components written in VHDL [7], the most sensible way of implementing such a structure (due to the easy extensibility) is by writing a specific connecting component in VHDL, which can instantiate the CλaSH-generated VHDL. The only responsibility of the connecting component is to distribute and forward signals: it should not perform any computation. It would be a straightforward process of generating such a connecting component based on the signal names in the top-level entities of the components it instantiates, however, due to the fact that it does not change for different designs and it is not very large, the connecting component has been written by hand in VHDL. Figure 2.1 depicts an overview schematic of the complete system, including both the FPGA and HPS side.

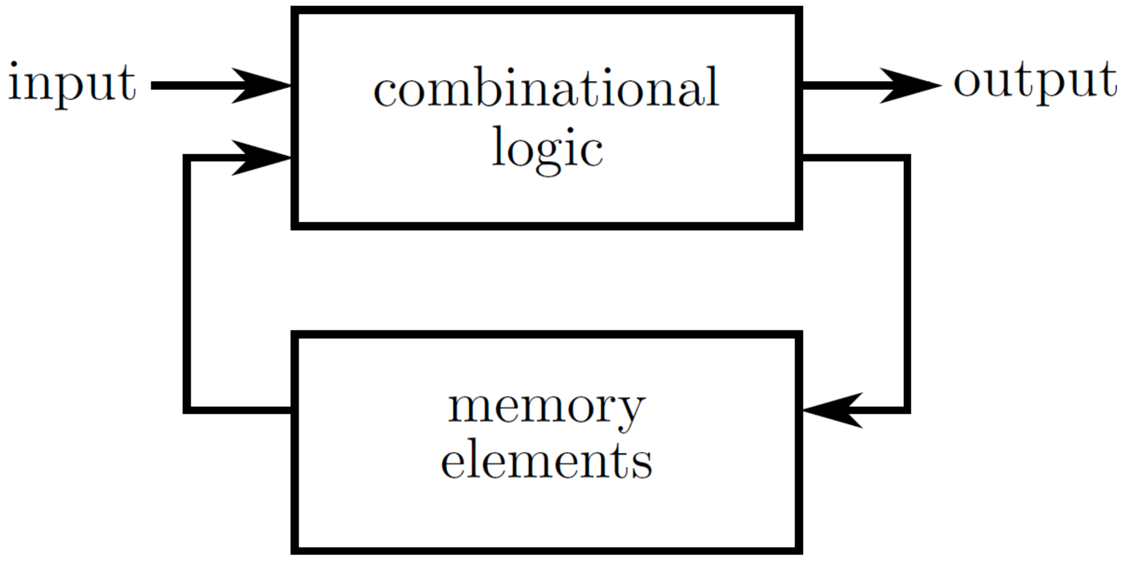

Fig. 2.1 TODO FIX ENTER GRAPHIC OF STRUCTURE WITH ILLUSTRATOR

2.2 External types

Usually, one of the first things to do when setting up a Haskell project is defining types, and using CλaSH forms no difference. It is especially useful to start by defining an input and an output signal. In a project with simple IO requirements the input and output signals would map straight to switches, keys and LEDs for testing purposes, but as the goal is to write data from the host system into the FPGA registers, the input and output signals will be a lot more complicated. The input and output signals from the CλaSH-generated VHDL top entity are shown in listing 2.1. The input consists of three separate channels: a bidirectional control channel, the actual input channel and some input needed for the output channel. It might appear strange that the output channel requires input, but this is necessary as the communications adhere to a strict master-slave pattern, in which the HPS is the master and the FPGA the slave. The HPS has to indicate whenever it wants to receive the FPGA output (by specifying an address (out_address) and setting the out_read-bit). As a consequence of the strict master-slave protocol, there are a lot less output signals than input signals: only the control and output channel can output data. Lastly, there are the keys, switches and LEDs as additional input and output signals. This concludes the type signatures of the input and output signals in CλaSH-Haskell. However, as the goal is to generate actual VHDL, CλaSH requires some additional information for the naming of the ports of the topEntity of the VHDL module. This is specified using the topEntity-annotation. In this case this annotation is not very interesting and it's merely a restatement of the input and output signal names, but when the CλaSH-generated VHDL should instantiate other VHDL-modules or use multiple clocks, the annotation can become more complex. It is shown in listing 2.2.

Listing 2.1 Input and output signals for CλaSH

```
data InputSignals = InputSignals { keys_input :: BitVector 4    -- keys_input
                                 , switches_input    :: BitVector 4  -- switches_input
                                 , control_write     :: Bit          -- control_write
                                 , control_writedata :: BitVector 32 -- control_writedata
                                 , control_address   :: BitVector 8  -- control_address
                                 , control_read      :: Bit          -- control_read
                                 , in_write          :: Bit          -- in_write
                                 , in_writedata      :: BitVector 32 -- in_writedata
                                 , in_address        :: BitVector 8  -- in_address
                                 , out_address       :: BitVector 8  -- out_address
                                 , out_read          :: Bit
                                 } deriving(Show)

data OutputSignals = OutputSignals { leds_status :: BitVector 4   -- leds_status
                                   , control_readdata :: BitVector 32  -- control_readdata
                                   , out_readdata     :: BitVector 32  -- out_readdata
                                   } deriving(Show)
```

Listing 2.2 Signal names used in the CλaSH-generated topEntity VHDL module

```
{-# ANN topEntity
  (defTop
    { t_name    = "compute_main"
    , t_inputs  = [ "keys_input"
                  , "switches_input"
                  , "control_write"
                  , "control_writedata"
                  , "control_address"
                  , "control_read"
                  , "in_write"
                  , "in_writedata"
                  , "in_address"
                  , "out_address"
                  , "out_read"
                  ]
    , t_outputs = [ "leds_status"
                  , "control_readdata"
                  , "out_readdata"
                  ]
    }
  )
  #-}
```

2.3 Internal types

So far the external types of the CλaSH-code have been covered, but the real work is done by the internal types: the types that keep track of the state of the system and allow it to do useful work. In order to allow for easy modifications to the types upon which the FPGA operates, they have been defined once and have been referenced everywhere else. This in combination with the property of the higher-order functions in CλaSH that they can operate on all vectors, regardless of length ensures that changing some internal types will not break the program.

2.3.1 Internal number representation

The solvers main data type is **type** Data = SFixed 8 24. The constructor SFixed 8 24 stands for a signed fixed-point number, with 8 integer bits and 24 fractional bits. It uses the 2's complement signed number representation, meaning that an 8-bit integer part is capable of representing the integers in the range $[-128..127]$. The 24 fractional bits give this number representation a smallest representable unit of 2^{-24}. This results in accuracy up to the 7th decimal place, which is similar to the IEEE 754 single precision floating point standard. The main reason for the use of fixed point integers is that CλaSH does not support floating point numbers yet, but additionally, fixed point representations are less demanding on FPGA area and result in a shorter critical path. Furthermore, the reason for choosing the total width of the number representation to be 32-bits is purely convenience: the input and output bridges are also 32 bits wide which allows you to send a single number per write or read. The UInt type acts as number representation in cases where only positive integer values are required, for instance in counters.

2.3.2 SystemConstants and SystemState

The SystemConstants keep track of the variables that do not change during the process of solving the ODE. It consists of a variety of constants that need values for the integration scheme (maxtime, timestep and maxstep). Furthermore, it may contain custom constants, which are passed to the equation to be approximated. This setup allows for (limited) changes to the equation by changing the constants at run-time without the time-consuming requirement of recompiling the entire FPGA side of the project.

Moving on to the data type which responsible for keeping track of the state of the system: the SystemState. This type has two fields: the ODEState, which is has the exact same function as the similarly named type in section 1.4.3: it keeps track of the values and the time in the numerical solver. It does this using a valueVector (of type Vec 4 Data) and a time variable (of type Data). The main difference between the type of the ValueVector in this implementation of ODEState and the one in section 1.4.3 is that this one uses a vector instead of a list. In Haskell, lists can have any length (including infinite) whereas the CλaSH-vectors have a fixed length. This property is very important when generating VHDL, as all vector lengths have to be immutable and known at compile-time in order to compile the higher-order Haskell functions (eg. map) to VHDL and afterwards to hardware.

The second field of the SystemState is a counter called step. Together with the maxstep field in SystemConstants, these govern the amount of output generated from the FPGA. A more elaborate explanation of the generation of output can be found in section 2.8.

Listing 2.3 Internal state variables for CλaSH

```haskell
type Data = SFixed 8 24
type UInt = Unsigned 32
type ValueVector = Vec 5 Data
type ConstantVector = Vec 20 Data

data ODEState = ODEState { valueVector :: ValueVector
                         , time :: Data
                         } deriving(Show)

data SystemState = SystemState { odestate :: ODEState
                      , step :: UInt
                      } deriving(Show)

data SystemConstants = SystemConstants { maxtime :: Data
                        , timestep :: Data
                        , maxstep :: UInt
                        , userconstants :: ConstantVector
                      } deriving (Show)

uIntMax = 4294967295 :: UInt

type Equation = (ODEState, ConstantVector) -> ValueVector
type Scheme = SystemConstants -> Equation -> ODEState -> ODEState
```

2.4 Implementation of equations and integration schemes

The CλaSH-implementations of the equations and integration schemes can be very similar to the implementations in plain Haskell, from section 1.4.3. However, CλaSH does not support any special operations yet (exponentiation, trigonometry or fractional powers) and therefore this imposes limitations on the type of equations which are representable, especially non-linear and heterogeneous equations which use special operations. In order to allow for easy specification without recompilation of a large range of equations, the default equation for testing will be a 4 by 4 matrix vector equation (2.1). This set up is capable of representing any linear and homogeneous fourth order equation with constant coefficients as well as lower-order equations with constant heterogeneous parts.

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}' = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 & c_7 \\ c_8 & c_9 & c_{10} & c_{11} \\ c_{12} & c_{13} & c_{14} & c_{15} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.1}$$

In order to implement this equation in Haskell (listing 2.4 shows the 2 by 2 version), it is certainly possible to use higher order functions, using `foldl (+) 0 $ zipWith (*) matrixRow Vector`. However, this would require the construction of additional vectors. Furthermore, as CλaSH is not yet completely optimizing, in order to obtain the highest possible performance for this hot-zone of the hardware, the decision was made to manually unroll the higher order functions into expressions containing only the unpacking of variables from the SystemConstants and simple arithmetic operators, $+$ and $*$.

Listing 2.4 Implementation of a second order equation with constant coefficients

```
matrix2d :: Equation
matrix2d (odestate, constants) = dxs
  where
    xs = valueVector odestate

    c1 = constants !! 4
    c2 = constants !! 5
    c3 = constants !! 6
    c4 = constants !! 7

    x0 = c1 * (xs !! 0) + c2 * (xs !! 1)
    x1 = c3 * (xs !! 0) + c4 * (xs !! 1)

    dxs = fst $ shiftInAt0 xs (x0 :> x1 :> Nil)
```

As for the integration schemes, these are incredibly similar to the implementations from section 1.4.3. The first part only contains unpacking of necessary variables from the records. For Euler's method, the actual work integration scheme is only a single line, the remaining lines check whether the time is not yet exceeding the maximum simulation time and generate an ODEState type, which gets used stored as part of the state in the main controlling logic, from listing 2.8. A reference to the controlling logic (the topEntity), the integration schemes and the equations can be found in appendix B

Listing 2.5 Euler's method in CλaSH

```
euler :: Scheme
euler constants equation state = state'
  where
    --Unpack the needed values
    c_user = userconstants constants
    c_maxtime = maxtime constants

    h = timestep constants
    t = time state
    xs = valueVector state

    --Apply Euler's integration scheme
    eulerxs = zipWith (+) xs $ map (*h) (equation (state, c_user))

    --Check the time constraints
    (xs', t') = if t < c_maxtime then (eulerxs, t + h)
                                 else (xs,t) -- already at maximum time

    state' = ODEState {valueVector = xs', time = t'}
```

2.5 Simulation

After writing the Haskell code which is going to be compiled by CλaSH, it is very easy to simulate your design: CλaSH includes a simulate function which is capable of generating user-specified signals. Especially in contrast with the process of generating test benches for VHDL, using Haskell to perform simulations saves a lot of time and typing. However, choosing to perform the tests in Haskell over a VHDL testbench does have the underlying assumption that CλaSH properly translates the Haskell specification into VHDL. CλaSH is also capable of generating VHDL testbenches, but this merely shifts the assumption of correctness from CλaSH' ability to generate VHDL to its ability to generate VHDL testbenches, therefore, this option of directly generating VHDL testbenches has not been explored further.

Listing 2.6 A simulation in CλaSH

```haskell
dis = defaultInputSignals

--write input
cis x y = defaultInputSignals { control_write = 1, in_address = x, in_writedata = y }
wis x y = defaultInputSignals { in_write = 1, in_address = x, in_writedata = y }

-- start computation and get output
scs = defaultInputSignals { control_write = 1, control_writedata = 1 }
fvs x = defaultInputSignals { out_address = x, out_read = 1 }

is  = [ cis 4 (-1)
      , cis 5 7
      , wis 1 10
      , wis 2 20
      , wis 3 30
      , wis 4 40
      , scs
      , dis
      , dis
      , fvs 1
      , fvs 2
      , fvs 3
      , fvs 4
      ]

test x = Data.List.take (Data.List.length x) $ Data.List.map out_readdata $ simulate topEntity x
```

The simulations in CλaSH consist of three parts:

1. *Signal definitions* - In order to keep the rest of the code concise, it is important to first define often-used signals. For versatility, these signals can even be functions: some values still have to be defined in order to return a signal.
2. *Signal listings* - This is where signals defined in step 1 are put together in order to create a list of signals: the input of the simulate function.
3. *Simulation* - The simulate function takes a topEntity and a list inputs and returns a list of output. Due to the infinite nature of the Signal in CλaSH, which is fed into the topEntity as input it is necessary to only take a certain amount of elements. However, as Haskell is evaluated lazily, the infinite lists are not a problem.

The output of the simulation is printed to screen, which was checked by eye for grave mistakes. If everything appeared to be correct, the design was ready for synthesis. The reason for such inexactness in verifying by simulation is that (at least for simple) Haskell programs, that they tend to have the property that they either work correctly or they do not work at all. Furthermore, the real trouble in debugging lies in the other parts of the FPGA side of the design: proper clock frequencies and the IO system.

2.6 Synthesis and deployment

After the simulation appears to be correct, CλaSH should generate HDL which can be compiled by the FPGA vendors tools into a binary file which can be used to program the FPGA. The process of synthesis and deployment consists of several steps, of which a short overview is shown below.

1. *Generating HDL* - The CλaSH compiler contains the optional flags *−−vhdl* and *−−verilog*. These can be used to generate HDL.
2. *Project creation* - Set up the external IO systems. These will not be written in CλaSH-generated HDL, but will be generated by another tool. In case of Altera this is the QSys system. Furthermore, add the manually written HDL files, for instance the connecting component and a frequency divider.
3. *Adding CλaSH-generated files to the project* - Add all CλaSH-generated files: if both the connecting component and the CλaSH-Haskell code were written properly then the connecting component should be able to instantiate the CλaSH-HDL as their ports match.
4. *Compiling* - After everything has been added and configured properly, start the compilation. For Altera FPGAs, the result will be a .sof (SDRAM Object File).
5. *Deploying* - Use the 'Programmer' feature in order to flash the .sof to the FPGA.

The process depicted above is rather high in manual workload as it uses the Quartus GUI. A shell script automatizing the process is described in appendix **??**.

2.7 Loading data into the FPGA

2.7.1 Constants

In order to understand the CλaSH source code, it's important to know what the variables mean. In order to keep the lines relatively short, the variable names are rather short, but they do follow a fixed pattern. All variables starting with i_ indicate input, s_ a state and c_ a constant value. As for the input variables, those consist further out of 1 or 2 characters. The first character indicates the source channel of the input (whether it is a real input : i, a control signal : c or a request for output o). The second optional character is either a, for address, or s, for 'set': the boolean indicating that the input is ready to be read. When this second character is missing, the data itself is meant.

Listing 2.7 Handling the input of the constants

```
systemConstants'  --Enter the constants into the ConstantVector
```

```
    | i_cs  == 1  && i_ca == 1           = systemConstants{  maxtime = i_c_d  }
    | i_cs  == 1  && i_ca == 2           = systemConstants{  timestep  = i_c_d  }
    | i_cs  == 1  && i_ca == 3           = systemConstants{  maxstep = i_c_u  }
    | i_cs  == 1  && i_ca >= 4           = systemConstants{  userconstants  = c_user'  }
    | otherwise                          = systemConstants
    where
      c_user'  = replace  i_ca  (unpack i_c  ::  Data)  c_user
      i_c_d    = unpack i_c  ::  Data
      i_c_u    = unpack i_c  ::  UInt
```

The first step of getting the system to work is loading constants into the FPGA. These constants govern the time step, the maximum time for simulation, how much output to generate and it's possible to specify custom constants which can be used in the equations you are solving. In order to keep the system simple, these constants are sent over the control channel as the input channel is reserved for initial values. However, before covering the specifics of handling the constants, it is important to understand the behaviour of the signals originating from the bridge between the HPS and the FPGA first. Whenever the HPS program writes the 32-bits value V to 8-bit address A, three things happen simultaneously, control_writedata takes on the value V, control_address gets set to address A and control_write gets set to true. This set up means that it is possible to differentiate the target of the control input signal based on the value of the address. Only whenever control_write is true the control input value can be considered valid. Lastly, as Haskell is a strongly typed language, you cannot simply insert a BitVector 32, originating from control_writedata into a Vec 4 (SFixed 8 24). You first have to cast or unpack the BitVector 32 into a SFixed 8 24, which luckily does not pose any problems as they both consist of 32 bits. The protocol for entering constants is depicted in table 2.1.

Table 2.1 The protocol for entering constant values into the FPGA, based on addresses

Address	**Function**	**Specifics**
0	Signaling flags	Writing 1 starts the computation
		Writing 2 performs a soft reset
1	Maximal computation time	
2	Time step	
3	Step limit for blocking	
4+	Custom constants	

2.7.2 Initial values

The initial values are loaded into the FPGA in the same way as the constants. The address designates the location at which the value should be stored. In order to understand how the data gets loaded into the FPGA registers it is important to If this is the case, the valueVector of ODEState in the SystemState gets updated: the value at in_address gets replaced with in_writedata, which gets unpack'ed into the main data type used by the application.

Listing 2.8 State machine responsible for controlling the solver

```
    (systemState', oul', block')
      --Handle the setup (reset the state, insert input values, start the computation)
      | i_c == 2 && i_cs == 1 && i_ca == 0  = ( initialSystemState , 0, 0)
      | i_is == 1                           = ( systemState{ odestate = s_odestate_in' }, 0, 1)
      | i_c == 1 && i_cs == 1 && i_ca == 0  = ( systemState{ step = 0 } , 0, 0)

      --Handle the computation and output:
      | block == 1 && i_os == 1             = ( systemState, pack (xs !! i_oa), block)
      | block == 0 && s_step < c_maxstep    = ( systemState_up' , 0, block)
      | block == 0 && s_step >= c_maxstep   = ( systemState{ step = uIntMax}, pack uIntMax, 1)

      --Default, do nothing
      | otherwise                           = ( systemState, oul, block)
      where
        s_odestate_in'  = s_odestate {valueVector = replace i_ia (unpack i_i :: Data) xs}

        s_odestate_up   = scheme systemConstants equation s_odestate
        valueVector_wt  = replace 4 (time s_odestate_up) (valueVector s_odestate_up)
        s_odestate_up'  = s_odestate_up {valueVector = valueVector_wt }
        s_step'         = s_step + 1

        systemState_up' = systemState{ odestate = s_odestate_up', step = s_step'}
```

2.8 Solving the system and extracting values

After sending the command to the FGPA to start solving the ODE over the control channel, on every clock cycle the FPGA will update the ODEState and the step variable. The ODEState gets updated by applying an integration scheme to the equation, which results in a new vector of values and a new value for the time. The step variable gets incremented to indicate that another step has passed. At some point, the value of step will exceed the value of maxstep, from SystemConstants. Whenever this happens, the FPGA stops updating the ODEState and writes all ones to the output port. This indicates that the FPGA is done processing and the results are ready to be collected by the HPS.

The step of SystemState in listing 2.3 field takes a bit more explanation. Every time the integration scheme gets applied, the step variable gets incremented. At some point, the value of step becomes larger or equal than the maxstep field from the data type SystemConstants. Whenever this happens, the system blocks until you order it to start again by setting the value of step to 0. During the time that the system does not progress, the values of the system can be read. This is done by requesting access to an address from the HPS. This request gets processed by the bridge and the hardware on the FPGA side into a high value for the out_read input signal, accompanied by a valid address from the out_address input signal. The FPGA is then responsible for actually writing the requested value to the output channel, in which the out_address is directly equal to the index in the ValueVector.

References

[1] Chart: A library for generating 2d charts and plots. https://hackage.haskell.org/package/Chart, 2014. [Accessed: June 2015].

[2] Computer language benchmarks game. http://benchmarksgame.alioth.debian.org/u32q/which-programs-are-fastest.html, 2015. [Accessed: June 2015].

[3] clash-prelude-0.8.1 - clash.tutorial. http://hackage.haskell.org/package/clash-prelude-0.8.1/docs/CLaSH-Tutorial.html, 2015. [Accessed: June 2015].

[4] Mining hardware comparison. https://en.bitcoin.it/wiki/Mining_hardware_comparison, 2015. [Accessed: June 2015].

[5] Sockit - the development kit for new soc device. http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=&No=816, 2015. [Accessed: June 2015].

[6] Christiaan Baaij. *CλasH: From Haskell To Hardware*. 2009. URL http://essay.utwente.nl/59482/.

[7] Christiaan Baaij. Cλash fpga starter. http://christiaanb.github.io/posts/clash-fpga-starter/, 2015. [Accessed: June 2015].

[8] J Kuper. *A Short Introduction to Functional Programming*. 2015.

[9] J Polking, A Boggess, and D Arnold. *Differential Equations with Boundary Value Problems*. Prentice Hall, Upper Saddle River, New Jersey, 2006.

[10] Keith Underwood. Fpgas vs. cpus: Trends in peak floating-point performance. *Association for Computing Machinery*, 2004. URL http://home.engineering.iastate.edu/~zambreno/classes/cpre583/2006/documents/Und04A.pdf.

[11] Wayne Wolf. *FPGA-Based System Design*. Prentice Hall, Upper Saddle River, New Jersey, 2004.