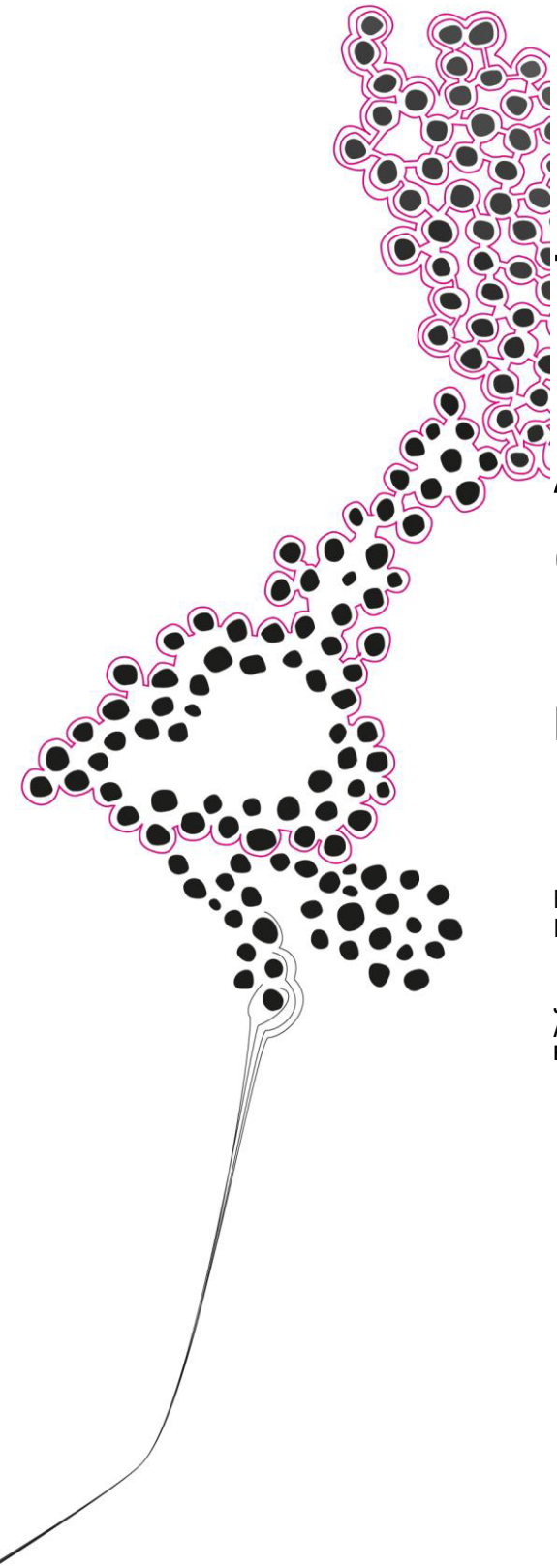
MASTER THESIS



TO A NEW HARDWARE DESIGN METHODOLOGY: A CASE STUDY OF THE COCHLEA MODEL

Floris van Nee

FACULTY EEMCS
INCAS³

JAN KUPER
JAN STEGENGA
ANDRE KOKKELER
RINSE WESTER

incas³

UNIVERSITY OF TWENTE.

Abstract

In this thesis a design methodology based on mathematical transformations and Haskell was investigated with a case study. The topic of the case study was the cochlea model, for which the goal was to design a working implementation of the cochlea model on an FPGA. The case study showed that the mathematical design methodology has advantages like faster development and less error-prone transformations, because the transformations are in mathematical form and can be verified. The CλaSH compiler which takes a subset of Haskell and compiles it into VHDL is very useful in this design methodology as it automates the last step from Haskell to VHDL. The cochlea model in its original form proved to be too large to fit on the selected FPGA. However, a less complex and less precise version of the model, with a reduced number of slices, led to an implementation which could fit on the FPGA.

Contents

1 Introduction

Hardware design has always been a complex task considering the tradeoffs that have to be made between speed, area and development time. Because the size of transistors has been decreasing for a long time, thereby making larger designs possible, it is becoming an even more complex task to design hardware correctly. Furthermore, the advent of FPGAs have made hardware design more widespread, because they provide a low-cost solution for many problems where a high-performance design is required. Therefore, it is vital to have a design process with support for these tradeoff factors efficiency and development time.

In the field of hardware design the design problem often consists of a specification which is written in a mathematical form. In order to end up with a synthesizable design the following methodology is often used: generally, a model of this specification is created in, for example, Matlab for validation, after which an implementation in a low-level language like C is created. As a last step, the C code is made suitable for implementation on hardware, for example by using it to write VHDL. This last step is usually done manually, but a lot of research currently focuses on automating this C to hardware conversion [1]. Also, SystemC is often used to translate the C code to as it provides slightly higher abstraction levels [2].

In order to see the possible advantages of a new design methodology, it is important to notice the flaws of the current design methodology. Most importantly, in the translation between the mathematical form and Matlab/C, a lot of important information is lost. Data dependencies, essential information for parallelisation, are clearly present in a mathematical form, but cannot be easily extracted from Matlab/C code [3]. Tools which automatically try to transform C code to VHDL are often restricted to a small subset of C for which it is possible for the tools to infer the data dependencies [1]. In other cases, the tool will generate either inefficient VHDL or it will not be able to process it at all. When engineers work on manually translating the C implementation in VHDL, they often have to infer these data dependencies from Matlab/C code themselves. This is a cumbersome and error-prone task which can easily lead to the introduction of bugs.

In this thesis we present a case study of a new design methodology which stays closer to the mathematical realm and therefore preserves data dependencies from the beginning. We discuss the complete design process starting from the mathematical description and ending at the implementation on an FPGA and note the advantages and disadvantages that were encountered. The fundamental idea behind the methodology is that the step to an imperative language like Matlab/C is removed or at least postponed as long as possible. By staying in the mathematical domain and doing transformations on mathematical formulas, it is possible to validate every transformation and therefore to preserve proven correctness of the design as long as possible. During the last step, the mathematical formulas are translated to Haskell, a functional programming language, and CλaSH, a library and compiler on top of Haskell to allow hardware descriptions and generation to VHDL [4]. Haskell has a large similarity to mathematics because it is a functional language. Therefore, the transformation between mathematics and Haskell/CλaSH is not very difficult, which further reduces the possibility of introducing unwanted bugs in the implementation.

The specific case study that is presented is about the implementation of a model of the cochlea. The cochlea is the auditory portion of the inner ear and a lot of research has previously been done on obtaining accurate mathematical models of the cochlea. Several models exist, some more complex than others, and in this case study we have chosen to focus on the one-dimensional cochlea model with linear damping distribution [5]. INCAS^3 is doing research on the cochlea model itself as well as on the implementation on several different platforms and on the use of the model for sound recognition. Investigating the implementation of the model on an FPGA was a natural next step as the model has a large potential for parallel execution and this is one of the areas in which FPGAs are strong. On top of that, because all processing is done locally on the FPGA instead of processed in a data centre, there are less privacy issues as not all data needs to be transmitted to a central server. Last, FPGAs are generally more energy efficient than software and they provide a first step to an ASIC design which are even more energy efficient. INCAS^3 generally does hardware design using the traditional approach with Matlab, C and VHDL, but is interested in the new approach as they have also observed problems with the traditional approach.

Therefore, the purpose of this research is two-fold. First, a new design methodology using CλaSH is tested with a case study to determine its strong and weak points. Second, by doing this case study valuable information about the implementation of the cochlea model on an FPGA is obtained. This allows us to formulate the reserach questions as follows:

- How well is the new design methodology and CλaSH suited for a large design project like the cochlea model?
- Is the FPGA a suitable platform for implementation of the linear cochlea model in terms of performance?

Factors that play a role in the first research question include the difficulties that arise when doing the mathematical transformations, how well the speed/area tradeoff can be considered in the early design stages with the mathematical formulas and if the current state of the CλaSH implementation is stable enough for a case study this size. For the second research question, the main factors are whether or not there is enough parallelism to exploit and whether the algorithm is not too complex to fit on an FPGA. As this is largely dependent on conditions like the type of FPGA and settings of the algorithm, we have formulated several specific implementation conditions which can be found in chapter 6.

This thesis is divided into several chapters. In the next chapter, an introduction and reasoning will be given to the mathematically based design methodology and hardware description language CλaSH. Chapter 3 gives an introduction to the cochlea and the linear model as described in literature. Chapter 4 is dedicated to one part of the algorithm where a tridiagonal system of equations needs to be solved. This chapter investigates different methods of solving the system. After a method to solve the system of equations has been chosen, it is possible to formulate the complete algorithm with recurrence relations and apply transformations to make it suitable for implementation on an FPGA. The transformations are described in chapter 5. Implementation specific issues are discussed in chapter 6 and results of simulations and synthesis are given in chapter 7. The thesis is concluded with a discussion of the results and a conclusion in chapter 8 and chapter 9.

2 CλaSH and functional hardware design

This chapter provides an introduction to the functional hardware design methodology using CλaSH. The first two sections focus on the design methodology and the theoretical advantages while the third section introduces CλaSH as a hardware description language.

2.1 Flaws in current design methodology

Traditionally, the hardware design methodology, especially in the signal processing domain, consists of several steps from specification to VHDL. Several improvements and abstraction layers have been suggested over the years, but the concept remains the same [2]. A graphical representation of this design methodology can be found in figure 2.1. The process always starts with a specification that needs to be implemented on hardware. The specification is often written down in a purely mathematical way. As a first step, this mathematical specification is converted to sequential code in a language like C. The main problem with this conversion is that the semantics of a C-like language do not match the semantics of the original mathematical specification. Because of this, a translation between the two is not straightforward. The largest difference between the two is called referential transparancy versus referential opaqueness [6]. In mathematics, expressions with the same parameters always yield the same result. Therefore, it is possible to replace an expression with its value. It is said to be referentially transparent. This is not the case in C, as expressions can have side-effects and they can yield different results each time they are evaluated. It is said to be referentially opaque. To validate correctness of the translation between mathematics and C, extensive verification is required. After the design has been verified, the second step is to translate the C-like language to RTL-style VHDL or Verilog code. Again, the semantics of VHDL and C-like code do not match, which leads to more time-consuming verification and the possible introduction of unwanted design faults.

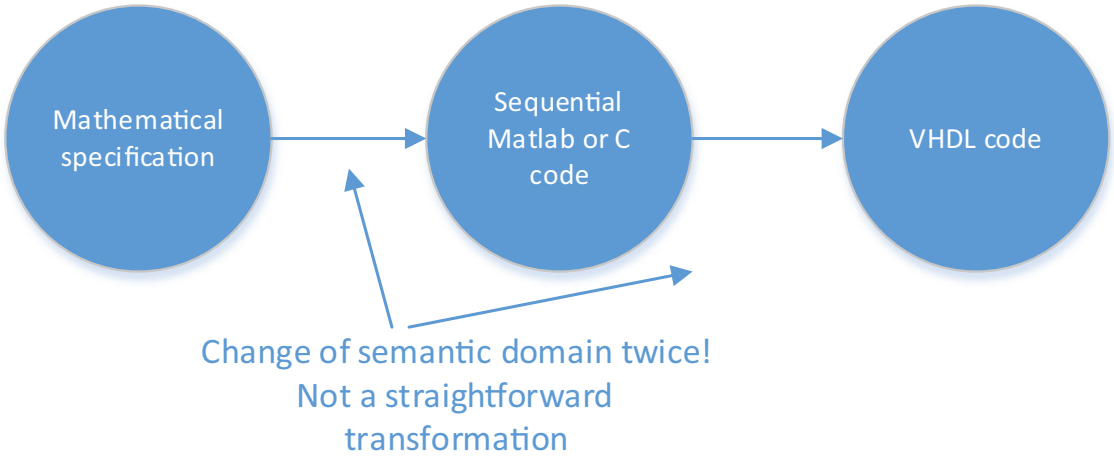


Figure 2.1: The traditional design methodology

Thus, the main flaw in the current design methodology is the need to change semantic domain twice: first from mathematics to sequential C-like code, then from C-like code to RTL-style VHDL or Verilog. Because the translation between these domains is not straightforward, extensive verification has to be performed. Also, the intermediate sequential C-like code is not strictly needed in the design process. This intermediate step mostly exists due to the fact that generally some implementation of the algorithm in software is required for verification, and that the most-used languages for software are sequential C-like languages. This generally makes it the first choice for implementation in software. However, these kind of languages are not suited for translation to hardware, because, as mentioned in the previous paragraph, there is a semantic mismatch between C-like languages and VHDL.

Therefore, we would like to have a methodology which includes the possibility to verify a software version of the algorithm, but not with a sequential C-like language. We would like to avoid the semantic mismatch, as discussed in the previous paragraphs, to reduce the effort required for verification and to reduce the possibility of introducing design faults.

2.2 Functional hardware design

The design methodology that is tested in this thesis is based on the functional programming language Haskell and the CλaSH compiler [7] [4]. A graphical representation of this methodology can be found in figure 2.2. It allows for software testing of the algorithm, but does not have the disadvantage of having to manually switch semantic domain twice. Instead, the mathematical specification can be rather straightforwardly converted to a Haskell program. The conversion between the mathematical specification and Haskell is less error-prone than the conversion between the specification and a C-like language, because the semantics of Haskell and mathematics largely match. Haskell has referential transparancy and functions in Haskell only specificy true data-dependencies, just like mathematical functions, which means that parallelism remains exposed. This is important, as in the transformation to hardware this parallelism needs to be exploited. In programs written in C, the designer needs to manually introduce parallelism again.

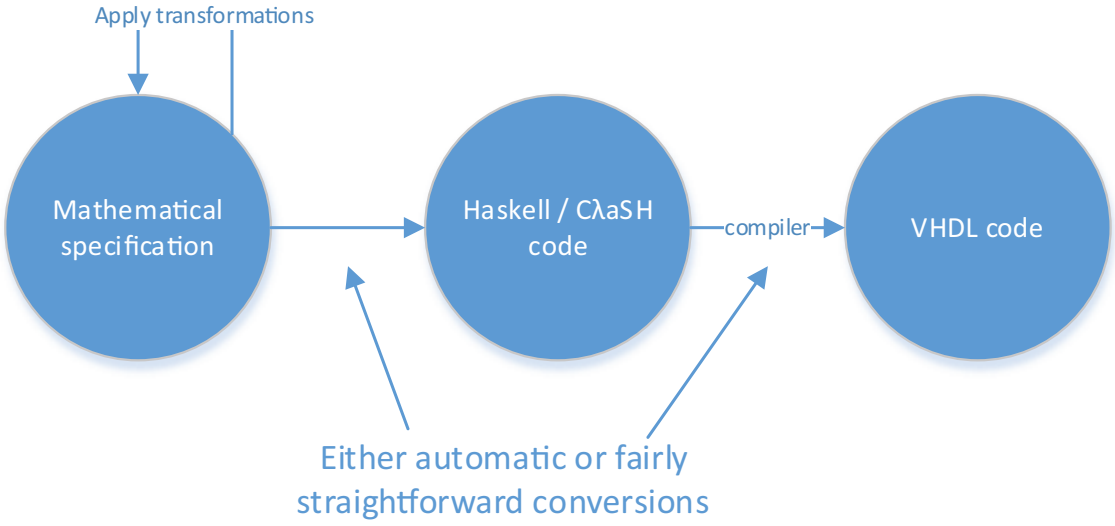


Figure 2.2: The functional design methodology

Of course, an evaluation model is needed in order to execute Haskell functions on a processor.

A processor is sequential in nature and the GHC compiler (Glasgow Haskell Compiler), the most widely used compiler for Haskell, can be used to compile Haskell programs to code which runs on a processor. This step is thus done automatically instead of manually in the case of translating the mathematical model to C.

When the mathematical specification has been translated to Haskell, the next step is to find transformations on the specification for optimizations. These transformations can be written down mathematically, allowing us to prove the correctness of the transformations at any point in the design process. Transformations can be as simple as operand interchangement of associative functions, but more complex transformations can also be done some of which are used and described in chapter 5.

The resulting specification in Haskell can then be given to the CλaSH compiler, which takes a subset of Haskell as input and produces VHDL-code of that design as output. Once the VHDL-code is obtained, it can be used with standard synthesis tooling. The choice for VHDL as output is solely made, because of the large number of existing synthesis tools.

As can be seen in figure 2.2, the big advantage of this methodology is that every step in the process is either performed by the compiler, mathematically provable or fairly straightforward. This is a huge advantage over the old methodology in figure 2.1, where both steps need to be done manually and are not straightforward at all.

2.3 CλaSH

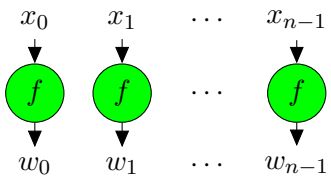
CλaSH is the compiler which takes a Haskell program and translates it to VHDL. Not every Haskell program can be translated to VHDL though. Only a subset of Haskell is supported in the current CλaSH version. Restrictions include fixed-point calculations only, no recursion and vectors are used instead of lists. This section aims to introduce CλaSH with a small example of a FIR-filter as also described in [3].

2.3.1 The functions map, zipWith and fold

In the functional programming language Haskell there are three functions which are used very often. These three functions operate on lists and all of them also take a function as one of their arguments, which means they are higher order functions. The functions are: `map`, `zipWith` and `fold`.

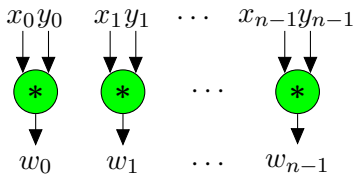
The function `map` takes a function `f` as input and a list of elements `xs` and applies the function `f` to every element in `xs`. The function `f` takes one input of type `a` and outputs a value of type `b` (possibly `a` equals `b`). Thus, the type of `map` is:

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



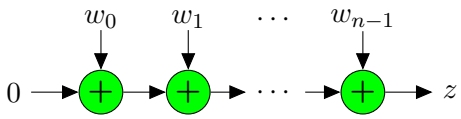
The function `zipWith` is similar to `map`, except that it takes a binary function `f` as input and two lists, `xs` and `ys`, instead of one. It then applies the function `f` pairwise to every element in `xs` and `ys`: it zips them together. The type of `zipWith` is:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```



The function `fold` reduces a list of elements to a single value. In order to that, it takes a list of elements `xs` to reduce, a starting value `z` and a function `f` which takes two inputs and produces one output. Reduction will start at on end of the list and at each step the result of the last element and the next element will be the input to the function `f`. The type of `fold` is:

```
fold :: (a -> b -> a) -> a -> [b] -> a
```



2.3.2 The filter

A FIR-filter is defined mathematically as follows:

$$y_t = \sum_{i=0}^{N-1} x_{t-i} \cdot h_i$$

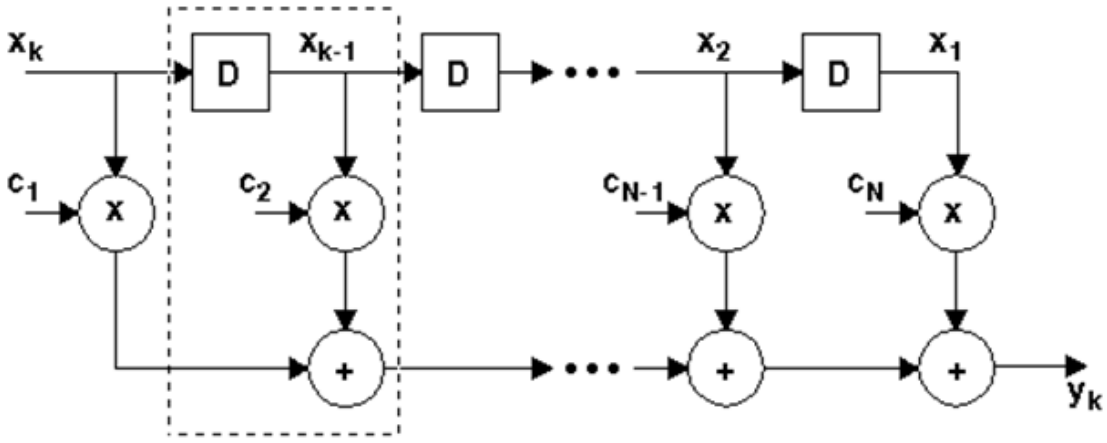


Figure 2.3: Structure of a FIR-filter

Essentially, the filter takes the dot-product of a coefficients vector h and a vector of the same size containing consecutive samples of the input stream x. We can use the `fold` and `zipWith` to define a function for the dot-product (note the `v` in front of the functions to indicate that they are CλaSH functions):

```
dotp xs hs = vfoldl (+) 0 (vzipWith (*) xs hs)
```

The FIR-filter can then be defined as:

```
fir coeffs xs = ys
  where
    ys = dotp coeffs wxs
    wxs  = window xs
```

The `window` function takes sufficient values from the input stream `xs` to match the size of `coeffs`. It then applies to dot-product function to produce the result of the filter.

3 The cochlea

3.1 The real cochlea

The word cochlea is the classical Latin word for inner ear. The human ear, or any mammalian ear for that matter, can be divided into three parts: the outer ear, the middle ear and the inner ear. A schematic of the complete human ear can be found in figure 3.1. Sound waves travel through air into the external auditory canal, which is part of the outer ear. The middle ear, with the malleus and incus acts as a conversion between the sound pressure in air in the outer ear and fluid displacement in the inner ear. The cochlea then converts this fluid displacement into signals that travel to the brain.

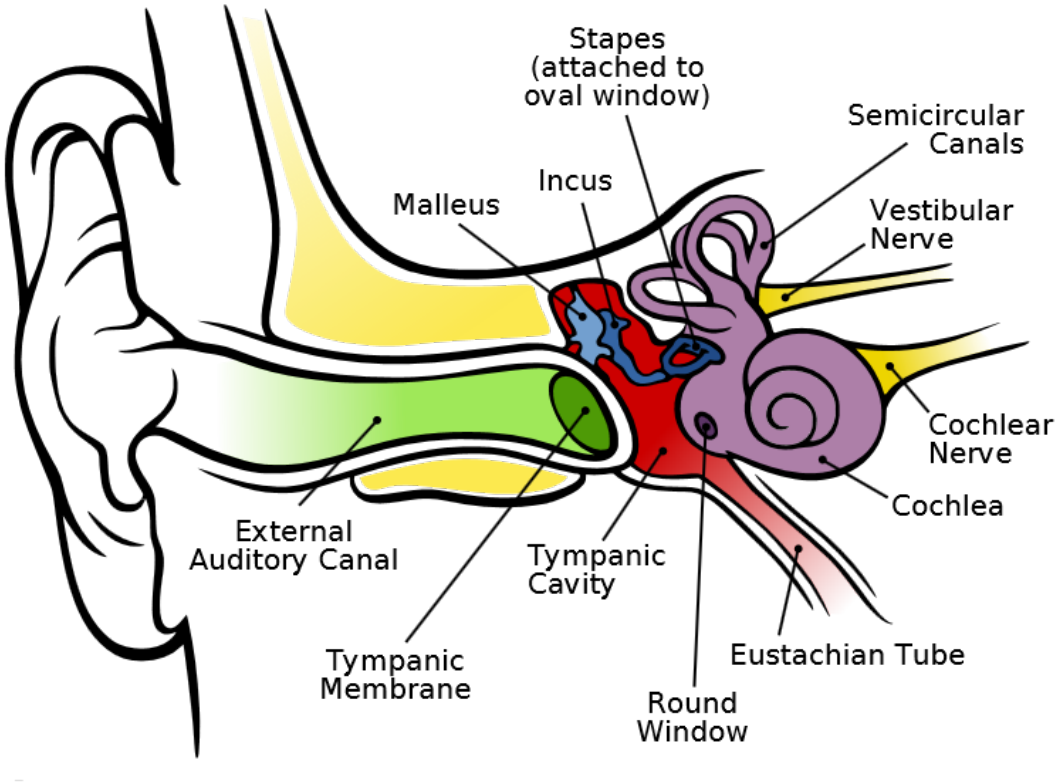


Figure 3.1: Schematic of the human ear [8]

3.1.1 Structure

The cochlea has the form of a spiral in which waves propagate from the base, located near the middle ear, to the apex at the center of the spiral. At the base there are two openings, of

which the upper one is oval and the lower one round. The main parts of the cochlea include (see figure 3.2):

- The scala vestibuli, the scala which lies superior to the cochlear duct. This has the oval window at the base.
- The scala tympani, the scala which lies inferior to the cochlear duct. This has the round window at the base.
- The scala media, also called cochlear duct, which lies in between the scala vestibuli and scala tympani. It has a high potassium ion concentration.
- The helicotrema, the location where the scala vestibuli and scala tympani merge at the apex.
- The basilar membrane, the membrane that seperates the scala media from the scala tympani.
- The Reissner's membrane, the membrane that seperates the scala media from the scala vestibuli.
- The Organ of Corti, the part taking care of the transduction mechanism. It is located on the basilar membrane. The mechanics of this are complex and not relevant for this thesis and will thus not be discussed further.

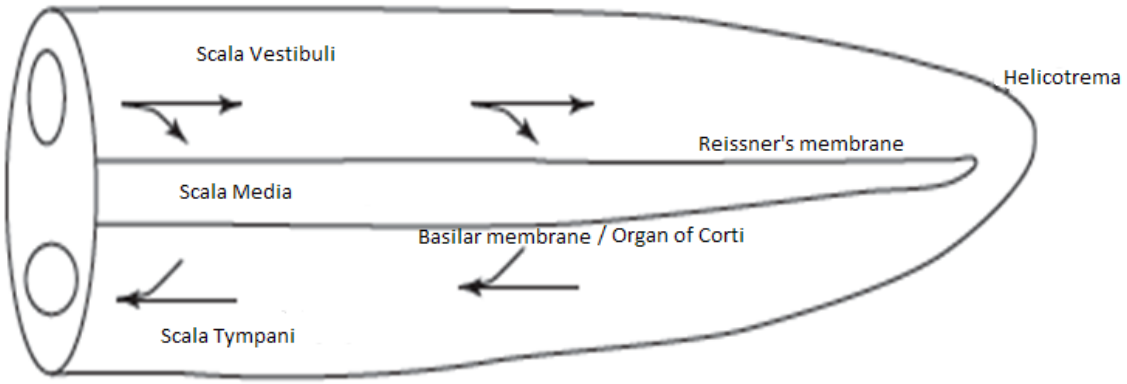


Figure 3.2: Structure of the cochlea [5]

For the model that is described in the next section, the important part of the cochlea is the basilar membrane together with the Organ of Corti. These two together are called the cochlear partition.

It is important to understand the general concept of the cochlea. Figure 3.3 illustrates that the basilar membrane is sensitive to different frequencies at different locations of the membrane. When a high frequency serves as input, the basilar membrane will react close to the base, while for low frequencies it will react close to the apex. Thus, the speed and displacement of the basilar membrane at a certain point can serve as an interpretation of the frequency of the input signal.

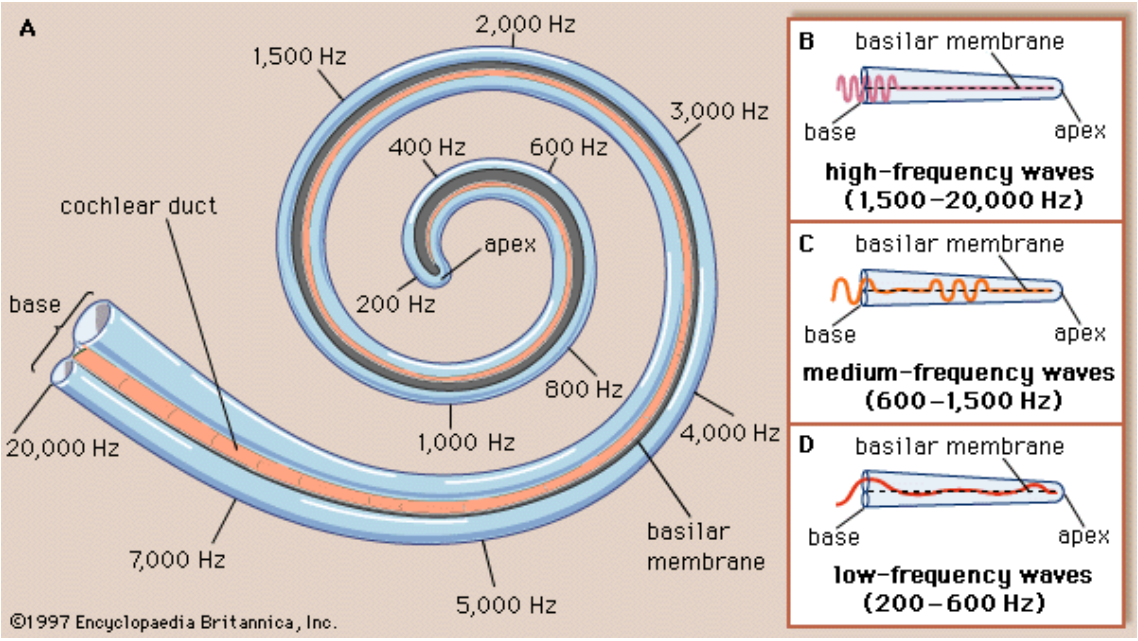


Figure 3.3: Structure and frequency response of the cochlea

3.2 The cochlea model

Several models of the cochlea exist [5] [9] [10]. In this section, a visual approach will be given for the models with linear and non-linear damping distribution as described by Duifhuis [9].

Both models are a drastical simplification of the real cochlea, because the real cochlea is a very complex organ. A drawing of the model can be found in figure 3.4. Simplifications of the real cochlea compared to the model include:

- The cochlea is rolled out. Instead of a spiral form, it is now stretched out in one dimension.
- The scala media and scala vestibuli are lumped together. The exact reason why this is possible is out of the scope of this thesis, but is based on the properties of the Reissner's membrane and the hydromechanical properties of the different fluids.
- The cochlea ducts have rigid walls. Communication can only happen through its oval and round windows.
- The cross-sectional areas of the ducts are assumed to be constant over length and are approximated by rectangles.
- The fluid is assumed to be inviscid and linear.
- The cochlear partition is assumed to be split up in small sections. Every section can be modelled as a mass-spring-damper system with different parameters. Thus, every section has a different resonance frequency.

Until here, the linear and non-linear version of the model are equivalent. The difference between the two arises with the damping distribution. The damping distribution can be modelled linearly or non-linearly. The linear and non-linear distribution can be found in figure 3.5. It specifies the velocity of the membrane on the x-axis versus the damping force on the y-axis. The model that is implemented in this thesis has linear damping distribution.

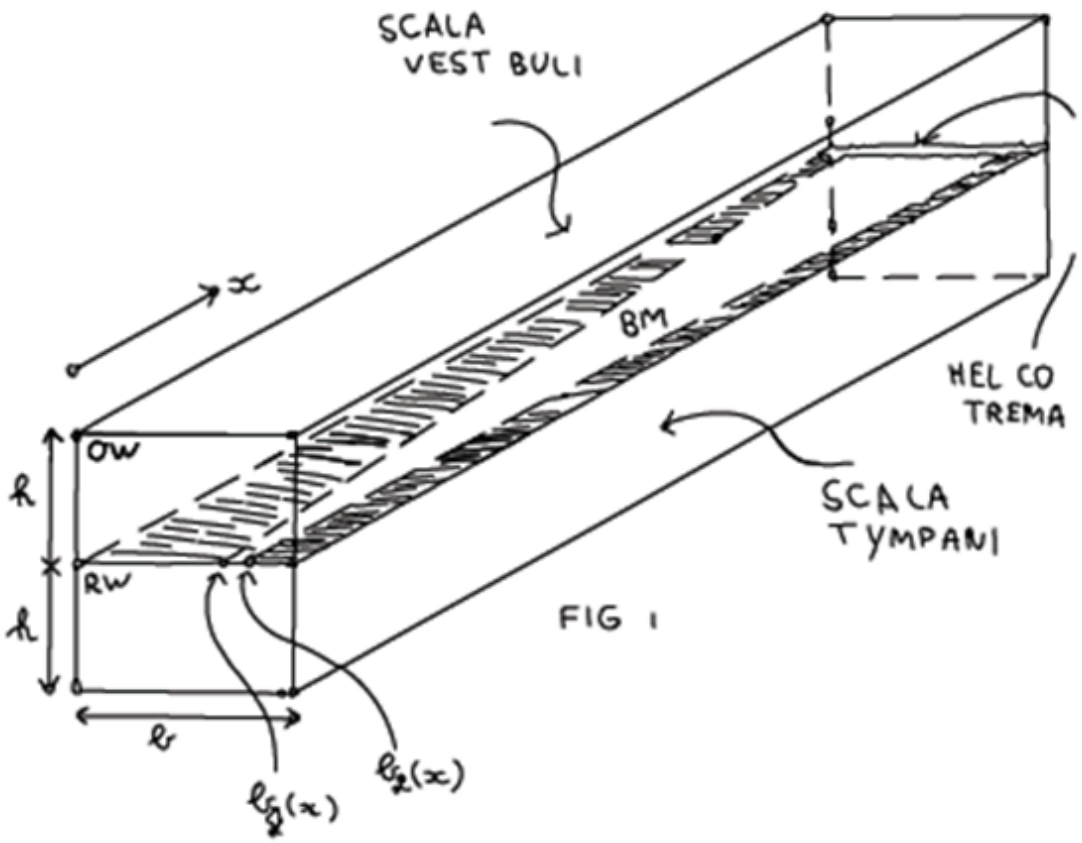


Figure 3.4: The simplified linear cochlea model [5]

One part that has been left unspecified yet is the input to the model. Sound waves first travel through the outer ear and middle ear before arriving at the cochlea. Therefore, even though the model only treats the cochlea, it is still important to have an idea of what happens to the sound waves before they reach the cochlea. The effect of the outer ear is assumed to be negligible and is thus ignored in the model. The middle ear has a simple mechanical transfer and is thus assumed to be linear in this model. Within INCAS^3, there is currently research on the effects of the middle ear on the input, because it can be modelled more precisely than a linear transfer. However, for simplicity the model described here uses the linear transfer function.

3.2.1 Mathematical approach to the model

This section describes the model with mathematical equations in terms of an electrical circuit. The previous section stated that the cochlea can be split in partitions where every partition can be modelled as a mass-spring-damper system. Mathematically, this is equivalent to an RLC-circuit: an electrical circuit containing of a resistor, inductor and a capacitor. As most of the audience of this thesis will be more familiar with electrical circuits, the model will be described in terms of an RLC-circuit.

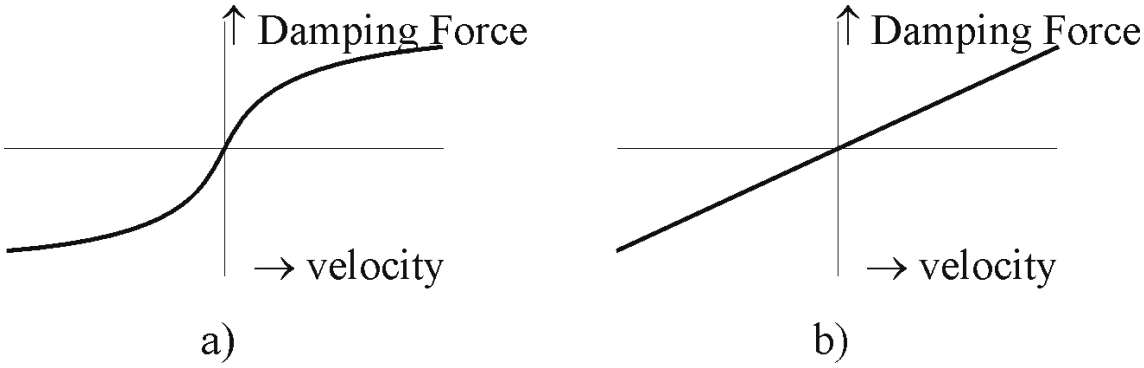


Figure 3.5: Example of damping distributions for both the non-linear model (a) and linear model (b) [10]

Examining this circuit leads to equations for the pressure at different points of the membrane. These equations are made dimensionless in order to bring all values within a small range to facilitate calculation in fixed point. Then, the equations for calculating the speed and distance of the membrane when the pressure is known are given. These are also made dimensionless. These equations still have a partial differential, thus they have to be discretized. The result of this discretization is a set of discretized recurrence relations for the speed and distance of the membrane.

Electrical equivalent

Van den Raadt describes the model as an RLC-circuit with $N+1$ oscillators [5]. Every oscillator corresponds to one of the small sections of the cochlear partition. An oscillator has a current through it. The current defines the speed of the membrane at that point. The charge (integral of the current/speed) defines the relative distance of one point of the membrane to position zero. We are interested in these two values for every oscillator in the membrane. The positions $n = 0$ (input signal) and $n = N$ (last oscillator) are handled somewhat differently, thus in this section we only focus on the oscillators between the first and last position, which means that $0 < n < N$. For the mathematical derivation for $n = 0$ and $n = N$ we refer to Van den Raadt [5].

One such oscillator can be described by the network in figure 3.6. With some circuit equations this can be rewritten to the simplified version in figure 3.7. While the figure suggests an electrical network, the names in the figure correspond to values in the actual model. Note that even though we use electric-domain analogy, we retain the mechanical-domain variable names. The following are used in the figures:

Note that the only thing that couples two oscillators is the fluid movement. Writing Kirchhoff Voltage Law (KVL) and Ohms law equations for point n in figure 3.7, using the definition of an inductor, gives us the voltage over two of the inductors:

$$p_{n-1} - p_n = 2m_c \frac{\partial U^+_{n-1}}{\partial t} \tag{3.1}$$

$$p_n - p_{n+1} = 2m_c \frac{\partial U^+_n}{\partial t} \tag{3.2}$$

Name	Electrical equivalent	Mechanical definition
U_n	current through point n downwards	speed of the membrane at
Y_n	charge at point n	position of the membrane at oscillator n
U^+_{n-1}, U^+_n	electrical current	fluid flowing to point n from the left and current flowing to point $n+1$ from the left respectively
p_{n-1}, p_n, p_{n+1}	voltage at said points	membrane pressure at said points
m_c	induction	acoustic mass of the cochlea fluid between two oscillators
m	induction	acoustic mass of oscillator n of the membrane
d_n	resistance	damping at oscillator n
s_n	capacitance	area of oscillator n

Table 3.1: Variable names and their electrical-domain equivalents

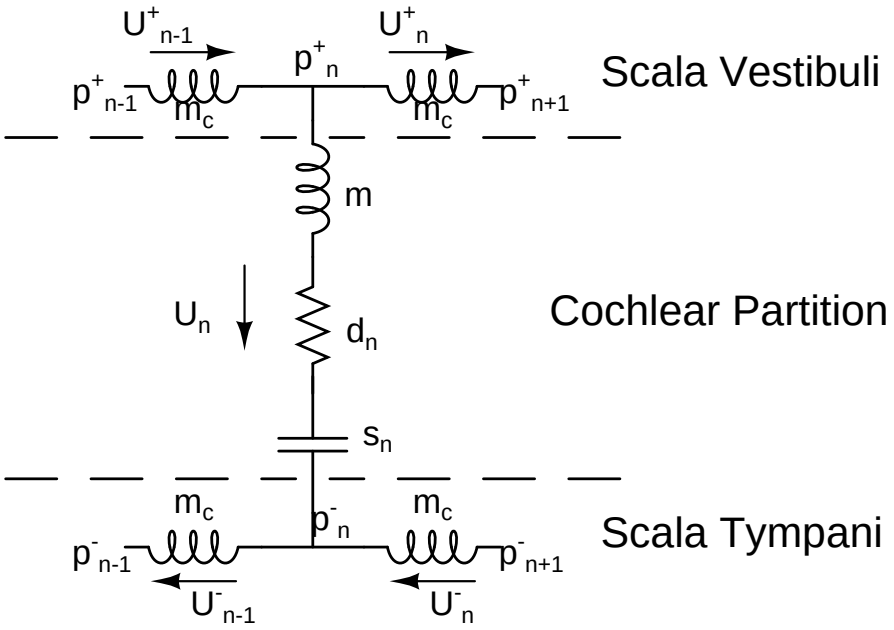


Figure 3.6: Oscillator for $0 < n < N$

Subtracting equation (3.2) from equation (3.1) and using Kirchoff Current Law, which states that the sum of the currents flowing from/to a node needs to be zero, leads to:

$$p_{n-1} - 2p_n + p_{n+1} = 2m_c \frac{\partial U_n}{\partial t} \tag{3.3}$$

This can be simplified further by first writing the voltage at point p_n as the sum of the voltage over the capacitor, resistor and inductor with inductance m:

$$p_n = m\frac{\partial U_n}{\partial t} + d_n U_n + s_n \int U_n \, dt \tag{3.4}$$

Here we define the last part of this equation as G_n, because it will make the final expression of the formulas more neat. Therefore, G_n becomes:

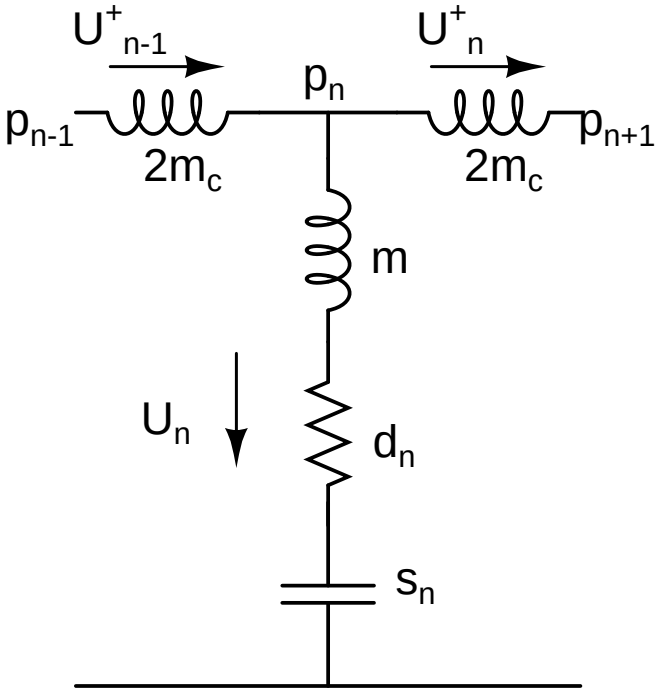


Figure 3.7: Simplified oscillator for $0 < n < N$

$$G_n = d_n U_n + s_n \int U_n \, dt \tag{3.5}$$

$$= d_n U_n + s_n Y_n \tag{3.6}$$

And G_n is substituted in equation (3.4):

$$p_n = m\frac{\partial U_n}{\partial t} + G_n \tag{3.7}$$

Slightly rewriting this with basic algebra leads to an equation for $\frac{\partial U_n}{\partial t}$:

$$\frac{\partial U_n}{\partial t} = \frac{p_n - G_n}{m} \tag{3.8}$$

Equation (3.8) can be substituted into our previous equation (3.3):

$$p_{n-1} - 2p_n + p_{n+1} = 2m_c\frac{p_n - G_n}{m} \tag{3.9}$$

$$p_{n-1} - 2p_n + p_{n+1} = 2\frac{m_c}{m}p_n - 2\frac{m_c}{m}G_n \tag{3.10}$$

$$p_{n-1} - 2(1 + \frac{m_c}{m})p_n + p_{n+1} = -2\frac{m_c}{m}G_n \tag{3.11}$$

Equation (3.11) is the main equation that needs to be solved in order to calculate p_n. There is one such equation for every oscillator in the model, thus for every $0 < n < N$. The structure

of the computation is already becoming clearer at this point. First, G_n can be calculated using state variables U_n and Y_n for every $0 < n < N$. Then, the right hand side can be calculated completely by multiping with $-2\frac{m_c}{m}$. Once this is known, we have to solve all these equations as a system of linear equations. The result of this computation yields all values p_n. As mentioned before, there is also an oscillator at $n = N$ and an input signal at $n = 0$. These are handled a little bit differently, but the principle is the same. It results in one large system of equations for $0 \leq n \leq N$ for a total of $n + 1$ equations.

Dimensionless equations

These equations contain values with a large range. Some, like the time step for example have a value of 10^{-3}, while others have much lower values. In order to perform the calculations more precise it is essential that the dynamic range of all values is as small as possible. This can be done by dividing the equations by some predefined constants with the same dimension. Afterwards, the equations are said to be dimensionless. Van den Raadt defines the following variables and chose a suitable value by using experimental data [5]. We use the tilde on top of a parameter to indicate that it is dimensionless. For example, $\tilde{\phi_n}$ is the dimensionless counterpart of ϕ_n. Parameters with a hat are used to indicate that these are used to make equations dimensionless.

$$\widehat{t_0} = 1 \cdot 10^{-3} s \tag{3.12}$$

$$\widehat{x_0} = 35 \cdot 10^{-3} m \tag{3.13}$$

$$\widehat{y_0} = 1 \cdot 10^{-9} m \tag{3.14}$$

The $\widehat{t_0}$ is used to make the quantity of time dimensionless. The $\widehat{x_0}$ is used to make the static quantities for the length of the basilar membrane dimensionless, like the ΔX which arises when discretizing the cochlea. To make the dynamic quantities dimensionless, like speed and pressure of the membrane, $\widehat{y_0}$ is used. For the full derivation, we refer to Van den Raadt [5]. In this document we present their main findings with a short derivation.

First, we define ϕ as:

$$\phi_n = \frac{p_n}{m_s} \tag{3.15}$$

ϕ_n has unit $\frac{m}{s^2}$, because m_c is a model parameter with unit m and p_n is in Pascal (see Van den Raadt for derivation), thus in order to make it dimensionless it needs to be multiplied with $\widehat{t_0^2}$ and divided by $\widehat{y_0}$. Defining $\tilde{\phi}$ as the dimensionless ϕ, we get:

$$\phi_n = \frac{\widehat{y_0}}{\widehat{t_0^2}} \tilde{\phi_n} \tag{3.16}$$

Now we expand our previous definition of G_n (equation (3.6)):

$$G_n = d_n U_n + s_n Y_n \tag{3.17}$$

$$= d_n \cdot b_{BM} \cdot \Delta X \cdot u_n + s_n \cdot b_{BM} \cdot \Delta X \cdot y_n \tag{3.18}$$

In this formula, b_{BM} is the width of the basilar membrane, ΔX the distance between two oscillators and u_n (unit $\frac{m}{s}$) and y_n (unit m) respectively the speed and distance per area of the basilar membrane.

We can make equation (3.18) dimensionless by making the following terms in the equation dimensionless:

$$y_n = \widehat{y}_0 \tilde{y}_n \tag{3.19}$$
$$\Delta X = \widehat{x}_0 \Delta \tilde{X} \tag{3.20}$$
$$t = \widehat{t}_0 \tilde{t} \tag{3.21}$$
$$\frac{d}{dt} = \frac{1}{\widehat{t}_0} \frac{d}{d\tilde{t}} \tag{3.22}$$

And using equation (3.19) and equation (3.22), u_n can also be made dimensionless:

$$\begin{aligned} u_n = \frac{dy_n}{dt} &= \frac{\widehat{y}_0}{\widehat{t}_0} \frac{d}{d\tilde{t}} \tilde{y}_n \\ &= \frac{\widehat{y}_0}{\widehat{t}_0} \tilde{u}_n \end{aligned} \tag{3.23}$$

These can be substituted into equation (3.18) to obtain a new equation for G_n:

$$G_n = b_{BM} \cdot \widehat{x}_0 \cdot \widehat{y}_0 \cdot \Delta \tilde{X} (\frac{d_n}{\widehat{t}_0} \tilde{u}_n + s_n \tilde{y}_n) \tag{3.24}$$

Using this new definition of G_n (unit $\frac{m^2}{s^2}$), we define the dimensionless g_n:

$$g_n = \frac{\widehat{t}_0^2}{\widehat{y}_0} \cdot \frac{G_n}{m_s} \tag{3.25}$$

As a last step before introducing our final equation for the normalized pressure $\tilde{\phi}_n$, we note that $\frac{2m_c}{m}$ can be normalized to:

$$\frac{2m_c}{m} = \alpha_{x_n}^2 \Delta \tilde{X}^2 \tag{3.26}$$

We refer to Van den Raadt voor de full derivation for this step. The important part is that α_{x_n} is a parameter which depends on several other parameters in the model, like membrane width, density of the cochlear liquid and the area of one oscillator.

It is now possible to substitute equations (3.15), (3.16), (3.25) and (3.26) into equation (3.11) to obtain the dimensionless equation for $\tilde{\phi}_n$ for $0 < n < N$:

$$\tilde{\phi}_{n-1} - (2 + \alpha_{x_n}^2 \Delta \tilde{X}^2) \tilde{\phi}_n + \tilde{\phi}_{n+1} = -\alpha_{x_n}^2 \Delta \tilde{X}^2 g_n \tag{3.27}$$

With g_n defined as:

$$g_n = \frac{b_{BM} \cdot \widehat{x}_0 \cdot \Delta \tilde{X}}{m_s} \cdot (d_n \hat{t}_0 \tilde{u}_n + s_n \hat{t}_0^2 \tilde{y}_n) \tag{3.28}$$

System of equations

Just like equation (3.11), equation (3.27) results in a system of $N+1$ equations (with $n=0$ and $n=N$ slightly different than shown here, but similar in principle) that can be solved by any method of solving systems of equations. Van den Raadt proved that the system is diagonally dominant. Furthermore, because every equation in the system only depends on its neighbours, it is a tridiagonal matrix system which can be solved efficiently. The resulting matrix-vector equation can be written as:

$$A \cdot \tilde{\boldsymbol{\phi}} = \boldsymbol{r} \tag{3.29}$$

Here, the right-hand side r_n and the main diagonal a_n of A (for $0 < n < N$) are defined as:

$$r_n = -\alpha_{x_n}^2 \Delta\tilde{X}^2 g_n \tag{3.30}$$

$$a_n = -(2 + \alpha_{x_n}^2 \Delta\tilde{X}^2) \tag{3.31}$$

The matrix A is defined as:

$$A = \begin{bmatrix} -(1+\alpha_0\Delta\tilde{X}_{01}) & 1 & 0 & 0 & 0 \\ 1 & a_1 & 1 & 0 & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & 1 & a_n & 1 & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & 0 & 1 & a_{N-1} & 1 \\ 0 & 0 & 0 & 1 & -(1+\alpha_{x_N}^2\Delta\tilde{X}^2 + \frac{4\hat{x}_0\Delta\tilde{X}}{\pi h + 4\hat{x}_0\Delta\tilde{X}}) \end{bmatrix} \tag{3.32}$$

The two diagonals next to the main diagonal of A are filled with 1, which shows the dependency on the neighbour oscillator, while the rest of the matrix is 0. Different methods to solve this system of equations are discussed in chapter 4.

Equations to integrate

The result of solving the system of equations will be the dimensionless pressure for every oscillator, $\tilde{\phi}_n$. Now, this result can be used to calculate the speed U_n and distance Y_n. Recall equation (3.7):

$$p_n = m\frac{\partial U_n}{\partial t} + G_n \tag{3.33}$$

This equation can be rewritten to:

$$\frac{\partial U_n}{\partial t} = \frac{p_n - G_n}{m} \tag{3.34}$$

$$\tag{3.35}$$

Also, as Y_n is defined as the integral of U_n, we have:

$$\frac{\partial Y_n}{\partial t} = U_n \tag{3.36}$$

Equations (3.34) and (3.36) need to be made dimensionless again. Again, for full details of this step see the document by Van den Raadt.

Equation (3.34) can be made dimensionless by substituting equations (3.16), (3.22), (3.23) and (3.25) and simplifying the resulting expression. This results in:

$$\frac{\partial \tilde{u}_n}{\partial \tilde{t}} = \tilde{\phi}_n - g_n \tag{3.37}$$

Equation (3.36) for Y_n can be made dimensionless by substituting equations (3.19), (3.22) and (3.23) and simplifying the resulting expression. This results in:

$$\frac{\partial \tilde{y}_n}{\partial \tilde{t}} = \tilde{u}_n \tag{3.38}$$

Thus, now we have the two dimensionless equations which need to be integrated over time. These equations are for oscillators at position $0 < n < N$. The equations for $n = 0$ and $n = N$ are slightly different, but have a similar form. For these equations we refer to Van den Raadt.

3.2.2 Discretized recurrence relations

The current model is already discretized for space: the basilar membrane is divided into sections where every section is modelled by an RLC-circuit as described in the previous section. However, the model is not discretized for time yet. Equations (3.37) and (3.38) contain integrals over time which need to be discretized in order to compute them.

To discretize these equations, we first recall the definition of the partial differential operator:

$$\frac{\partial f(t)}{\partial t} = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t} \tag{3.39}$$

A simple method of discretizing this partial differential operator is by taking a sufficiently small value for Δt to obtain an estimate of above equation without the limit. This method can be applied to equation (3.37) and (3.38) to eliminate the partial differential. As values like $\tilde{u}_n$ and $\tilde{y}_n$ also depend on the time, $\tilde{t}$, we write $\tilde{u}_{\tilde{t},n}$ and $\tilde{y}_{\tilde{t},n}$ from now on to make it explicit:

$$\begin{aligned} \frac{\partial \tilde{u}_{t,n}}{\partial \tilde{t}} = \tilde{\phi}_{\tilde{t},n} - g_{\tilde{t},n} \implies \\ \lim_{\Delta \tilde{t} \to 0} \frac{\tilde{u}_{\tilde{t}+\Delta\tilde{t},n} - \tilde{u}_{\tilde{t},n}}{\Delta \tilde{t}} = \tilde{\phi}_{\tilde{t},n} - g_{\tilde{t},n} \end{aligned} \tag{3.40}$$

$$\frac{\partial \tilde{y}_{t,n}}{\partial \tilde{t}} = \tilde{u}_{\tilde{t},n} \implies$$
$$\lim_{\Delta\tilde{t}\to 0} \frac{\tilde{y}_{\tilde{t}+\Delta\tilde{t},n} - \tilde{y}_{\tilde{t},n}}{\Delta\tilde{t}} = \tilde{u}_{\tilde{t},n} \tag{3.41}$$

By choosing $\Delta\tilde{t}$ small enough, we can obtain the discretization of time. This leads to a sequence of time steps. To simplify notation further, we now use the t as an integer indicating the time step. Thus, timestep t and $t+1$, to consecutive timesteps, differ by $\Delta\tilde{t}$ in actual dimensionless time. This leads to:

$$\frac{\tilde{u}_{t+1,n} - \tilde{u}_{t,n}}{\Delta\tilde{t}} = \tilde{\phi}_{t,n} - g_{t,n} \implies \tag{3.42}$$
$$\tilde{u}_{t+1,n} = \tilde{u}_{t,n} + \Delta\tilde{t} \cdot (\tilde{\phi}_{t,n} - g_{t,n})$$

$$\frac{\tilde{y}_{t+1,n} - \tilde{y}_{t,n}}{\Delta\tilde{t}} = \tilde{u}_{t,n} \implies \tag{3.43}$$
$$\tilde{y}_{t+1,n} = \tilde{y}_{t,n} + \Delta\tilde{t} \cdot \tilde{u}_{t,n}$$

These two equations are the discretization of equation (3.37) and (3.38), which means that we now have a fully discretized model for the cochlea. For completeness, let us recap all recurrence relations now. Note that all recurrence relations are now indexed both for time, t, and space, n. The boundaries for t and n are specified for all recurrence relations. Wherever the equations for $n = 0$ and $n = N$ are equal to those for $0 < n < N$, the boundaries reflect that. The boundary is set to $0 < n < N$ wherever the equations are not equal. In this case, we specify the relation for $n = 0$ and $n = N$, but do not give a formal derivation. We refer to Van den Raadt for the derivation of the equations for $n = 0$ and $n = N$.

Note that we have equations for $g_{t,n}$, $r_{t,n}$, $\phi\tilde{t}, n$, $u_{t,n}$ and $y_{t,n}$. The dependencies between these different equations is visualized in figure 3.8.

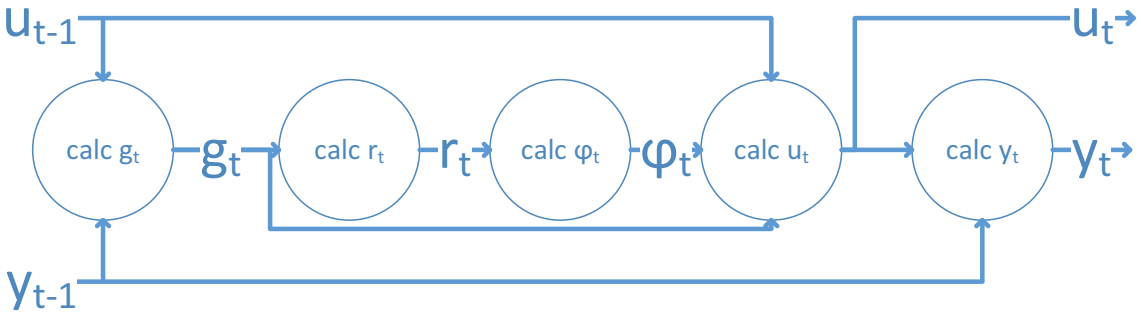


Figure 3.8: The data dependencies for different time steps between the equations

At timestep $t = 0$, all values are zero. For the electrical equivalent: current and charge are zero everywhere.

$$\tilde{u}_{0,n} = 0 \qquad 0 \leq n \leq N \tag{3.44}$$
$$\tilde{y}_{0,n} = 0 \qquad 0 \leq n \leq N \tag{3.45}$$

The values for the next step are then given by the following recurrence relations:

The relationship for $g_{t,n}$

Recall equation (3.28):

$$g_n = \frac{b_{BM} \cdot \widehat{x}_0 \cdot \Delta\tilde{X}}{m_s} \cdot (d_n \hat{t}_0 \tilde{u}_n + s_n \hat{t}_0^2 \tilde{y}_n) \tag{3.46}$$

In order to simplify the equations, we define:

$$\mathcal{U}_n = \frac{b_{BM} \cdot \widehat{x}_0 \cdot \Delta\tilde{X}}{m_s} \cdot d_n \hat{t}_0 \qquad 0 < n \leq N \tag{3.47}$$
$$\mathcal{Y}_n = \frac{b_{BM} \cdot \widehat{x}_0 \cdot \Delta\tilde{X}}{m_s} \cdot s_n \hat{t}_0^2 \qquad 0 < n \leq N \tag{3.48}$$

Both $\mathcal{U}_n$ and $\mathcal{Y}_n$ are only dependent on n and are constant over time, which is why they are only indexed with n. The equation for $n = 0$ is not shown, but is similar. Substituting this in (3.28) and adding the index over time for g, we get:

$$g_{t,n} = \mathcal{U}_n \cdot \tilde{u}_{t-1,n} + \mathcal{Y}_n \cdot \tilde{y}_{t-1,n} \qquad t \geq 1 \wedge 0 \leq n \leq N \tag{3.49}$$

The relationship for $r_{t,n}$

We define $r_{t,n}$ to be the right-hand side of the system of equations that needs to be solved at a given timestep. Recall equation (3.27):

$$\tilde{\phi}_{n-1} - (2 + \alpha_{x_n}^2 \Delta\tilde{X}^2)\tilde{\phi}_n + \tilde{\phi}_{n+1} = -\alpha_{x_n}^2 \Delta\tilde{X}^2 g_n \tag{3.50}$$

First, we define $\mathcal{R}_n$ for simplicity, just like $\mathcal{U}_n$ and $\mathcal{Y}_n$ were defined. See Van den Raadt for the calculation of $\mathcal{R}_n$ for $n = 0$ and $n = N$. For now, we assume that these values exist.

$$\mathcal{R}_n = -\alpha_{x_n}^2 \Delta\tilde{X}^2 \qquad 0 < n < N \tag{3.51}$$

The first oscillator $n = 0$ is handled differently than the other elements here, because it takes into account the current input i_t. Substituting $\mathcal{R}_n$ into (3.27):

$$r_{t,0} = \mathcal{R}_0 \cdot (i_t + g_{t,0}) \qquad t \geq 1 \tag{3.52}$$
$$r_{t,n} = \mathcal{R}_n \cdot g_{t,n} \qquad t \geq 1 \wedge 0 < n \leq N \tag{3.53}$$

Solving the equation

With the relation for the right-hand side given, it is now necessary to solve the system of linear equations. Several methods exist which are discussed and compared in chapter 4. For now, it is important to understand that the following equation needs to be solved for $\tilde{\phi}_t$ (see equation (3.29)), where A is a tridiagonal matrix, $\boldsymbol{r}_t$ a vector for all values of $r_{t,n}$ at a certain timestep and $\tilde{\boldsymbol{\phi}}_t$ a vector for all values of $\tilde{\phi}_{t,n}$ which need to be calculated.

$$A \cdot \tilde{\boldsymbol{\phi}}_t = \boldsymbol{r}_t \qquad t \geq 1 \tag{3.54}$$

The relationship for $u_{t,n}$

The discretized equations for $u_{t,n}$ is already given in equation (3.43). Recall that $u_{t,n}$ represents the speed of the membrane at point n. One minor adjustement is made to the equation for $u_{t,n}$. $\mathcal{V}_n$ is defined (m_s and m_{sm} are parameters of the model):

$$\mathcal{V}_0 = \frac{m_s}{m_{sm}} \cdot \Delta\tilde{t} \tag{3.55}$$

$$\mathcal{V}_n = \Delta\tilde{t} \qquad 0 < n \leq N \tag{3.56}$$

This is substituted into equation (3.43). Note that the the case for $n = 0$ is different, as it also takes into account the input.

$$\tilde{u}_{t,0} = \tilde{u}_{t-1,0} + \mathcal{V}_0 \cdot (\tilde{\phi}_{t,0} - g_{t,0} - i_t) \qquad t \geq 1 \tag{3.57}$$

$$\tilde{u}_{t,n} = \tilde{u}_{t-1,n} + \mathcal{V}_n \cdot (\tilde{\phi}_{t,n} - g_{t,n}) \qquad t \geq 1 \wedge 0 < n \leq N \tag{3.58}$$

The relationship for $y_{t,n}$

The equation for $y_{t,n}$, the displacement of the membrane at point n, remains equal to equation (3.44):

$$\tilde{y}_{t,n} = \tilde{y}_{t-1,n} + \Delta\tilde{t} \cdot \tilde{u}_{t,n} \qquad t \geq 1 \tag{3.59}$$

The complete set of equations has now been described. Note that all equations consist of simple operations which can be calculated indendently of each other in parallel, except the calculation of $\tilde{\phi}_{t,n}$. The next chapter will discuss efficient ways to solve the system of equations in order to calculate $\tilde{\phi}_{t,n}$.

4 Solving a tridiagonal system

In the previous chapter we discussed the cochlea and the linear model of the cochlea as described by Van den Raadt [5]. One part of the model involved solving a system of equations. This system of equations can be written as a matrix-vector equation:

$$A \cdot \tilde{\boldsymbol{\phi}}_t = \boldsymbol{r}_t \tag{4.1}$$

Three important properties of the matrix A are:

- The matrix is square and diagonally dominant
- The matrix is tridiagonal: only the main diagonal and the two diagonals next to it have non-zero values
- The matrix is not dependent on timestep t or any other dynamic variable

The first property ensures that the inverse of A and a solution to the equation exists. The second property allows us to use efficient algorithms to solve the system which specifically make use of the fact that most of the elements in the matrix are zero. The third property also allows for specific optimizations, because all steps of an algorithm that only use the matrix A can be calculated offline.

This chapter discusses methods for solving the system and investigates their use for implementation on an FPGA. Important factors here are the ability to parallelize the operations that are needed and the number of operations that are needed. First, a straightforward method is discussed which calculates the inverse of matrix A and then does a matrix-vector multiplication to obtain the unknown vector. Second, the TDMA algorithm is discussed. This is a specialized version of Gaussian elimination, specifically designed for tridiagonal matrices. Last, the method of cyclic reduction is described. Cyclic reduction can be seen as a more parallel version of TDMA. A fourth method which has similar properties to cyclic reduction exists which is called Bondeli's algorithm. However, it is not discussed here because previous research has found that it is more complex to implement than cyclic reduction and therefore better suited for CPUs [11].

4.1 Matrix inverse

A simple and straightforward method of determining the solution of a matrix-vector equation is to rewrite the equation with a matrix inverse. When we have a matrix A, a known vector $\boldsymbol{r}$ and an unknown vector $\boldsymbol{\phi}$, this can be written as:

$$A \cdot \boldsymbol{\phi} = \boldsymbol{r} \tag{4.2}$$

$$\boldsymbol{\phi} = A^{-1} \cdot \boldsymbol{r} \tag{4.3}$$

Effectively, this method consists of determining the inverse of matrix A and then performing a matrix-vector multiplication. Moreover, the matrix A is constant over time, and can be calculated offline. Therefore, the inverse of A can also be calculated offline. This reduces the steps needed to solve the system to one matrix-vector multiplication.

One advantage of this method is that it is highly parallelizable: a matrix-vector multiplication with n rows and columns consists of n^2 multiplications and $n \cdot (n-1)$ additions. Given enough resources, all of the multiplications can be done in parallel, while the additions can be done in $\lceil \log n \rceil$ sequential steps when an addition tree is used. Let us consider an example of a four by four matrix to see why these formulas hold:

$$A^{-1} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & d_1 & b_2 & b_3 \\ c_0 & c_1 & c_2 & c_3 \\ d_0 & d_1 & d_2 & d_3 \end{bmatrix} \tag{4.4}$$

$$\boldsymbol{r} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{4.5}$$

$$\boldsymbol{\phi} = A^{-1} \cdot \boldsymbol{r} = \begin{bmatrix} a_0 \cdot x_0 + a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 \\ b_0 \cdot x_0 + d_1 \cdot x_1 + b_2 \cdot x_2 + b_3 \cdot x_3 \\ c_0 \cdot x_0 + c_1 \cdot x_1 + c_2 \cdot x_2 + c_3 \cdot x_3 \\ d_0 \cdot x_0 + d_1 \cdot x_1 + d_2 \cdot x_2 + d_3 \cdot x_3 \end{bmatrix} \tag{4.6}$$

The example clearly shows no data dependencies between any of the multiplications. Thus, these can be done in parallel. The additions do have some dependencies on each other, because they have to sum up n terms. A summation of n terms requires at least $\lceil \log n \rceil$ sequential steps with a standard divide and conquer approach. Therefore, $\lceil \log n \rceil$ is the number of sequential addition steps required for the matrix-vector multiplication.

There is one problem with the matrix-vector multiplication approach. This problem is due to the properties of the inverse matrix. As said before, one of the important properties of matrix A is that it is tridiagonal: only $3n - 2$ elements of the matrix are non-zero. It turns out that the inverse, A^{-1} does not have this property. Instead, all elements in the inverse matrix are non-zero (although they do have the largest values on the main diagonal and exponentially decays to zero towards the top-right and down-left corners of the matrix). Still, the property that A is tridiagonal is not used at all in this approach. Another downside to this approach is that the elements of the inverse matrix become very small towards the corners of the matrix. In fixed-point calculations, this will lead to poor precision as the small numbers cannot be represented accurately.

The total number of operations of this approach has complexity $\mathcal{O}(N^2)$, because of the multiplications. There are specific implementation optimizations possible here, because one of the two operands of the multiplication are always constant. However, in general, this has worst-case complexity of n^2. The number of sequential steps of this approach has complexity $\mathcal{O}(\log N)$, because of the additions.

4.2 TDMA and Gaussian elimination

4.2.1 Gaussian elimination for arbitrary matrices

A sequential algorithm that is often used for solving a system of linear equations is Gaussian elimination. It can be used for various other problems too, for example to calculate the determinant of a matrix, find the rank of a matrix or calculate the inverse of an invertible square matrix. As a first step, the known vector is usually added as the right-most column of the known matrix. This is called an augmented matrix. Gaussian elimination works by performing elementary row operations on rows of the augmented matrix until the matrix is in reduced row echelon form. A matrix is in reduced row echelon form when the lower left-hand corner of the matrix is filled with zeros as much as possible, every leading coefficient (non-zero) in a row equals 1 and every column containing a leading coefficient contains only zeroes except for the leading coefficient itself. An example of an augmented matrix in reduced row echelon form is:

$$\begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 7 \end{bmatrix}$$

It can be seen that this system of equations is trivial to solve. Every 1 represents an element of the vector that needed to be solved and its value equals the value in the rightmost column on that row.

For Gaussian elimination, there are three elementary row operations that can be systematically applied to convert the augmented matrix to reduced row echelon form:

1. Swap the positions of two rows
2. Multiply a row by a nonzero scalar
3. Add to one row a scalar multiple of another

Gaussian elimination is numerically stable for diagonally dominant matrices, thus it can be applied to our matrix. However, Gaussian elimination works for any matrix which means that it does not exploit the fact that our matrix is tridiagonal. There exists a specialized form of Gaussian elimination which does exploit this property.

4.2.2 TDMA

A tridiagonal matrix already consists of mostly zeroes, because only three diagonals have non-zero elements. This can save us a lot of steps, because the matrix is already close to reduced row echelon form. A specialized form of Gaussian elimination called TDMA (Tridiagonal Matrix Algorithm) or Thomas algorithm exists which exploits this property [12].

Consider the three by three tridiagonal system of equations given by:

$$\begin{bmatrix} b_0 & c_0 & 0 \\ a_1 & b_1 & c_1 \\ 0 & a_2 & b_2 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

The strategy to solve this system is to first eliminate the a_i starting with a_1 and ending with a_2. Next, the c_i can be eliminated starting with c_1 and ending with c_0. Elimination can be done by using rule three of Gaussian elimination: to add to one row a scalar multiple of another. This is an iterative process which updates the matrix in every step.

Suppose we define:

$$m_1 = \frac{a_1}{b_0}$$

We can multiply the first row with m_1 and subtract it from the second row to obtain the following updated matrix where a_1 is eliminated:

$$\begin{bmatrix} b_0 & c_0 & 0 \\ 0 & b_1 - m_1 c_0 & c_1 \\ 0 & a_2 & b_2 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 - m_1 d_0 \\ d_2 \end{bmatrix}$$

This process can be repeated for the other rows until all a have been eliminated. This is called the forward reduction stage. Note that first a_1 has to be eliminated before a_2 can be eliminated. There are data dependencies between these two consecutive steps which means that these steps cannot run in parallel. The algorithm can be written in an iterative way as follows (in each step overwriting the previous matrix, n is the number of rows in the matrix):

for $k = 1$ to $n - 1$ do
$$m = \frac{a_k}{b_{k-1}}$$
$$b_k = b_k - m c_{k-1}$$
$$d_k = d_k - m d_{k-1}$$
end loop

After the forward reduction step, the backward substitution step is used to compute the values of x. It is done similarly as forward reduction: in each step a scalar multiple of one row is added to another. However, the backward reduction step starts at the last row and works its way up to the first row. Because after forward reduction all a have been eliminated, the last row is trivial to solve. It only depends on itself. Thus:

$$x_{n-1} = \frac{d_{n-1}}{b_{n-1}}$$

The rest of the values can be computed iteratively:

for $k = n - 2$ downto 0 do
$$x_k = \frac{d_k - c_k x_{k+1}}{b_k}$$
end loop

Obviously, the complexity of both the number of sequential steps and total number of operations is in $\mathcal{O}(N)$. To be more precise, forward reduction takes $2N - 2$ multiplications, $N - 1$

divisions and $2N-2$ additions. Backward substitution takes $N-1$ multiplications, N divisions and $N-1$ additions. This makes for a total of $3N-3$ multiplications, $2N-1$ divisions and $3N-3$ additions.

The data dependencies are clearer if they are not specified in an iterative loop as above, but with recurrence relations instead. This is also more in tradition with the functional approach proposed in this thesis. For this, we add an extra index to the parameters. $a_{n,0}$, $b_{n,0}$, $c_{n,0}$, $d_{n,0}$ represent the values of the matrix-vector equation before applying TDMA where n is the row index and N is the number of rows. These values are the input to the algorithm. $a_{n,1}$, $b_{n,1}$, $c_{n,1}$ and $d_{n,1}$ are the values after forward reduction. $a_{n,2}$, $b_{n,2}$, $c_{n,2}$, $d_{n,2}$ and x_n are the values after backward substitution.

$$
\begin{aligned}
m_n &= \frac{a_{n,0}}{b_{n-1,1}} && 1 \leq n < N \\
a_{n,1} &= 0 && 1 \leq n < N \\
b_{0,1} &= b_{0,0} && \\
b_{n,1} &= b_{n,0} - m_n c_{n-1,0} && 1 \leq n < N \\
c_{n,1} &= c_{n,0} && 0 \leq n < N-1 \\
d_{0,1} &= d_{0,0} && \\
d_{n,1} &= d_{n,0} - m_n d_{n-1,1} && 1 \leq n < N
\end{aligned}
$$

$$
\begin{aligned}
a_{n,2} &= 0 && 1 \leq n < N \\
b_{n,2} &= 1 && 0 \leq n < N \\
c_{n,2} &= 0 && 0 \leq n < N-1 \\
d_{N-1,2} &= \frac{d_{N-1,1}}{b_{N-1,1}} && \\
d_{n,2} &= \frac{d_{n,1} - c_{n,1} x_{n+1,2}}{b_{n,1}} && 0 \leq n < N-1 \\
x_n &= d_{n,2} && 0 \leq n < N
\end{aligned}
$$

However, some optimizations are possible in the case where the matrix is known beforehand. In this case, all expressions consisting only of a_k, b_k and c_k can be precalculated (at the cost of using extra memory during execution to store the precalculated values). This can also convert all divisions into multiplications by calculating the multiplicative inverse of b_k beforehand. This reduces the number of multiplications to $3N-2$, the number of additions to $2N-2$ and completely eliminates all divisions. The only calculations that remain are the calculation of $d_{n,1}$ and $d_{n,2}$ in which the expressions for m_n, $c_{n,1}$ and $\frac{1}{b_{n,1}}$ are precalculated.

The obvious advantage in comparison to the simple matrix-vector product computation is the lower number of total operations: $\mathcal{O}(N)$ instead of $\mathcal{O}(N^2)$. However, they cannot all be done in parallel. The number of sequential steps required for matrix-vector multiplication is $\mathcal{O}(\log N)$, while TDMA has order $\mathcal{O}(N)$.

4.3 Cyclic reduction

Cyclic reduction is a more parallel version of TDMA. It is described by Hockney and Jesshope and has interesting properties [13]. Instead of iteratively eliminating all a and then all c, the matrix is divided into two parts: row and column combinations with even indices form one part, while the odd indices form the second part. This process is repeated recursively on one part of the divided matrix, thereby splitting the matrix in half at every step. The system can be solved trivially once it has been reduced to one by one. The result of this can then be used in back substitution to recursively solve the other halves of the matrix at every step. There is also an even more parallel form of cyclic reduction which does not need the back substitution step. Both versions are described in this section, starting with the version with back substitution.

4.3.1 Cyclic reduction with back substitution

Algebraically, cyclic reduction can be derived as follows. The tridiagonal system of size N has an equation for every $0 \leq n < N$ (we define $a_0 = 0$ and $c_{N-1} = 0$):

$$a_n x_{n-1} + b_n x_n + c_n x_{n+1} = d_n$$

Now we define two new variables:

$$\begin{aligned} w_n &= x_{2n} \\ u_n &= x_{2n+1} \end{aligned}$$

By taking three successive equations and substituting above variables in it, we obtain:

$$\begin{aligned} a_{2n} u_{n-1} + b_{2n} w_n + c_{2n} u_i &= d_{2n} \\ a_{2n+1} w_n + b_{2n+1} u_n + c_{2n+1} w_{n+1} &= d_{2n+1} \\ a_{2n+2} u_n + b_{2n+2} w_{n+1} + c_{2n+2} u_{n+1} &= d_{2n+2} \end{aligned}$$

The w's can be eliminated from the middle equation by substitution of the first and third equation. This leads to:

$$\left(-\frac{a_{2n+1}a_{2n}}{b_{2n}}\right) u_{n-1} + \left(b_{2n+1} - \frac{a_{2n+1}c_{2n}}{b_{2n}} - \frac{c_{2n+1}a_{2n+2}}{b_{2n+2}}\right) u_n + \left(-\frac{c_{2n+1}c_{2n+2}}{b_{2n+2}}\right) u_{n+1} = d_{2n+1} - \frac{a_{2n+1}d_{2n}}{b_{2n}} - \frac{c_{2n+1}d_{2n+2}}{b_{2n+2}}$$

This is again a tridiagonal system, but now in the u's only. It can be solved by applying the same formula again if the system is not yet one by one or it can be solved directly for a one by one system. Once it is solved, the remaining unknowns can be solved by rearranging the third formula to:

$$w_n = \frac{d_{2n+2}}{b_{2n+2}} - \frac{a_{2n+2}}{b_{2n+2}} u_n - \frac{c_{2n+2}}{b_{2n+2}} u_{n+1}$$

A numerical example is now given for clarification. Suppose we start with the tridiagonal system of five equations given by:

$$\begin{bmatrix} 1.5 & -0.5 & 0 & 0 & 0 \\ -0.5 & 1.0 & -0.5 & 0 & 0 \\ 0 & -0.5 & 1.0 & -0.5 & 0 \\ 0 & 0 & -0.5 & 1.0 & -0.5 \\ 0 & 0 & 0 & -0.5 & 1.0 \end{bmatrix} \begin{bmatrix} 0.5 \\ 1.5 \\ 0.0 \\ 2.5 \\ 4.0 \end{bmatrix}$$

Eliminating the odd-indexed variables from the even-numbered equations leads to:

$$\begin{bmatrix} 1.25 & 0 & -0.25 & 0 & 0 \\ 0.5 & -1.0 & 0.5 & 0 & 0 \\ -0.25 & 0 & 0.5 & 0 & -0.25 \\ 0 & 0 & 0.5 & -1.0 & 0.5 \\ 0 & 0 & -0.25 & 0 & 0.75 \end{bmatrix} \begin{bmatrix} 1.25 \\ -1.5 \\ 2.0 \\ -2.5 \\ 5.25 \end{bmatrix}$$

From this the three by three tridiagonal system can be extracted. It contains the columns 0, 2 and 4 from rows 0, 2 and 4:

$$\begin{bmatrix} 1.25 & -0.25 & 0 \\ -0.25 & 0.5 & -0.25 \\ 0 & -0.25 & 0.75 \end{bmatrix} \begin{bmatrix} 1.25 \\ 2.0 \\ 5.25 \end{bmatrix}$$

Repeating the reduction step for this matrix yields:

$$\begin{bmatrix} 1.125 & 0 & -0.125 \\ 0.5 & -1.0 & 0.5 \\ -0.125 & 0 & 0.625 \end{bmatrix} \begin{bmatrix} 2.25 \\ -4.0 \\ 6.25 \end{bmatrix}$$

Extracting the even row and colums again:

$$\begin{bmatrix} 1.125 & -0.125 \\ -0.125 & 0.625 \end{bmatrix} \begin{bmatrix} 2.25 \\ 6.25 \end{bmatrix}$$

This in turn can be reduced to:

$$\begin{bmatrix} 1.1 & 0 \\ 0.2 & -1.0 \end{bmatrix} \begin{bmatrix} 3.5 \\ -10.0 \end{bmatrix}$$

From which the single equation can be extracted:

$$\begin{bmatrix} 1.1 \end{bmatrix} \begin{bmatrix} 3.5 \end{bmatrix}$$

This single equation can be solved directly: its solution is $\frac{35}{11}$. This result can be used for substitution in the previous equations to obtain the other values.

This process of reduction and backsubstution can also be visualized. Figure 4.1 shows the different steps. From left to right, it can be seen that during the reduction phase the number of rows decreases by a factor two each time. The new values depend on the previous row at

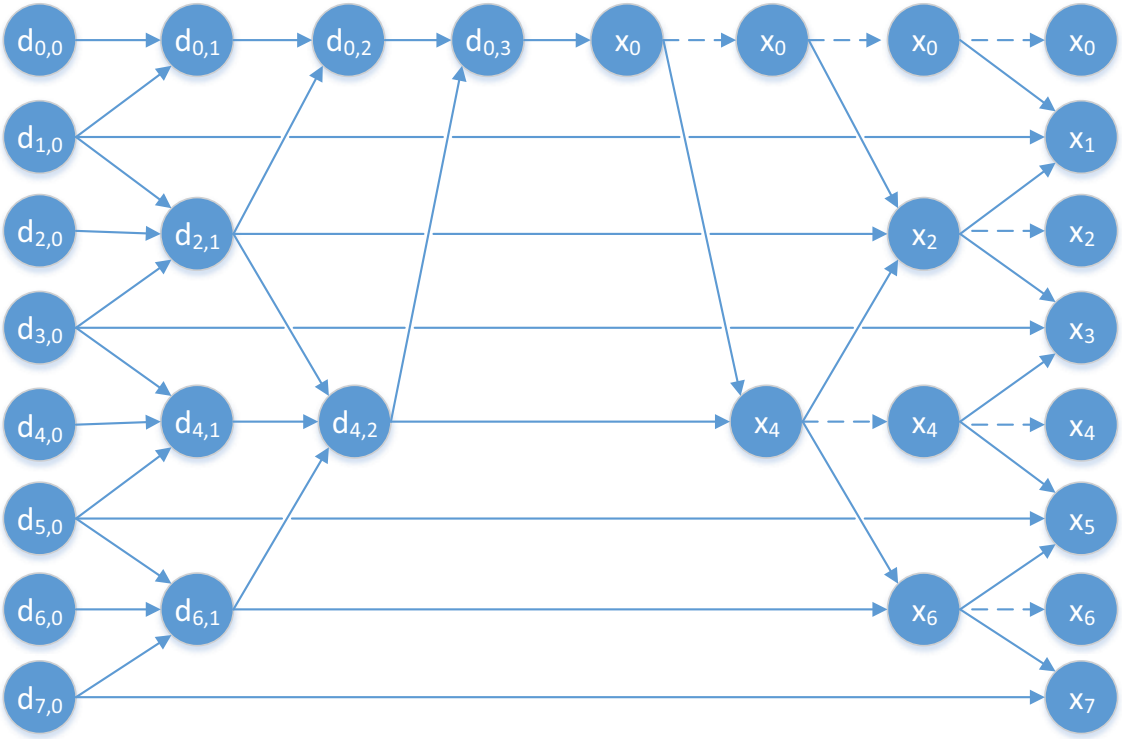


Figure 4.1: A visualization of cyclic reduction with back substitution for a vector of size eight

the index and the two neighbours. When a one by one system is left, it is solved directly and then the back substitution begins where the number of solved rows increases by a factor of two every step.

The total number of operations has order $\mathcal{O}(N)$, because the number of rows to work with decrease with a factor two each step. The number of sequential steps has order $\mathcal{O}(\log N)$, because of the divide and conquer approach. There are $\mathcal{O}(\log N)$ reduction steps, 1 direct solve step and $\mathcal{O}(\log N)$ back substitution steps.

As with TDMA, the exact number of multiplications and additions can be reduced, because the matrix is known beforehand. Thus, all expressions consisting only of known values can be calculated offline at the cost of some extra memory at runtime. This optimization will be discussed after an even more parallel method of cyclic reduction has been discussed in the next subsection.

4.3.2 Cyclic reduction without back substitution

In the previous subsection cyclic reduction with back substitution was discussed. This method of cyclic reduction needed $1 + 2\log n$ sequential steps. A more parallel version exists which only needs $1 + \log n$ sequential steps. However, this comes at the cost of a larger number of total operations, which increases from $\mathcal{O}(N)$ to $\mathcal{O}(N \log N)$.

The main difference between the two methods is that in the more parallel method, the reduction step is applied to all indices of the matrix, instead of just the even or odd indices. Thus, one reduction step of a matrix of size n results in another set of n three-term equations, half of which are equal to the equations that would have been obtained if cyclic reduction with back substitution was applied. Now another reduction step is applied independently to both the half

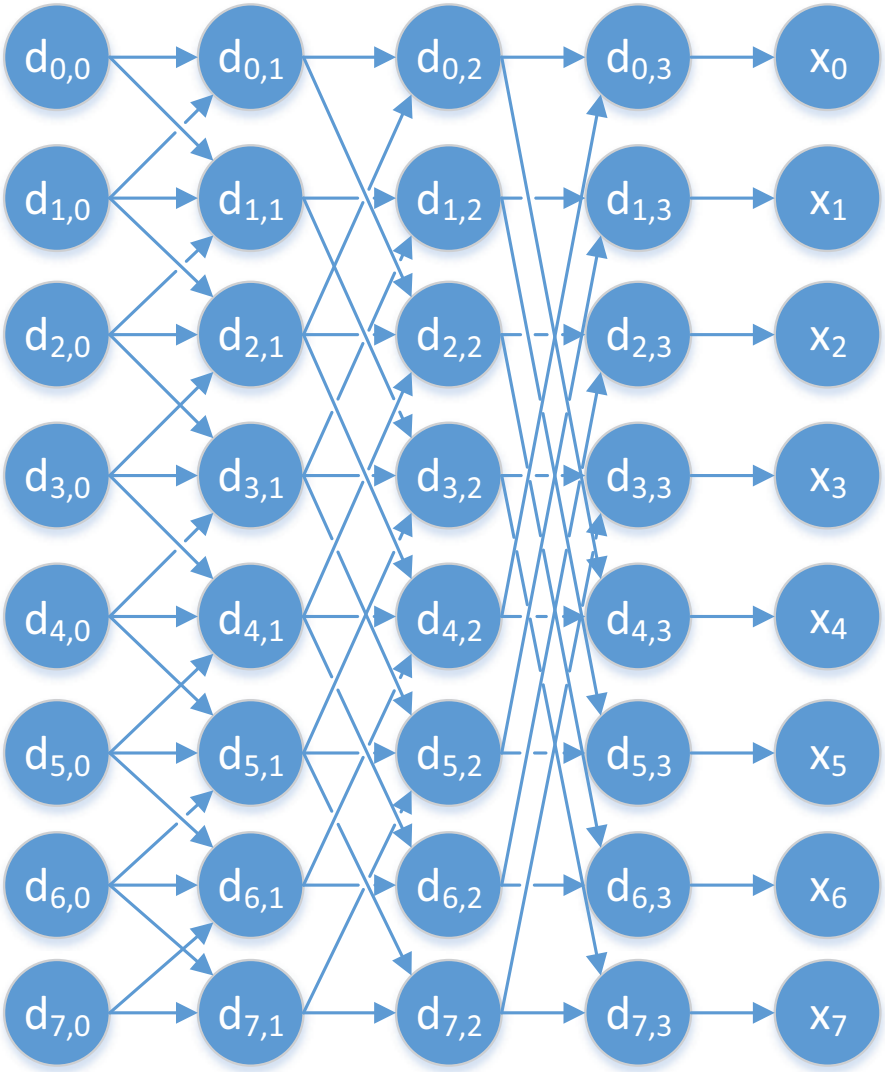


Figure 4.2: A visualization of cyclic reduction without back substitution for a vector of size eight

that was equal to normal cyclic reduction as well as to the other half. Repeating this $\log n$ times results in n equations which only depend on themselves and thus can be solved directly.

A visualization of this algorithm can be found in figure 4.2. In this figure, it is easy to see that the total number of steps has increased to $n \log n$, while the number of sequential steps decreased to $1 + \log n$.

It is important to note that it is possible to mix the two methods of cyclic reduction. For example, it is possible to first do one step with back substitution and the remaining without. An example of this can be found in figure 4.3.

4.3.3 Optimizations

As with TDMA, the exact number of multiplications and additions that are needed for both methods of cyclic reduction can be reduced, because the matrix is known beforehand. Thus, all

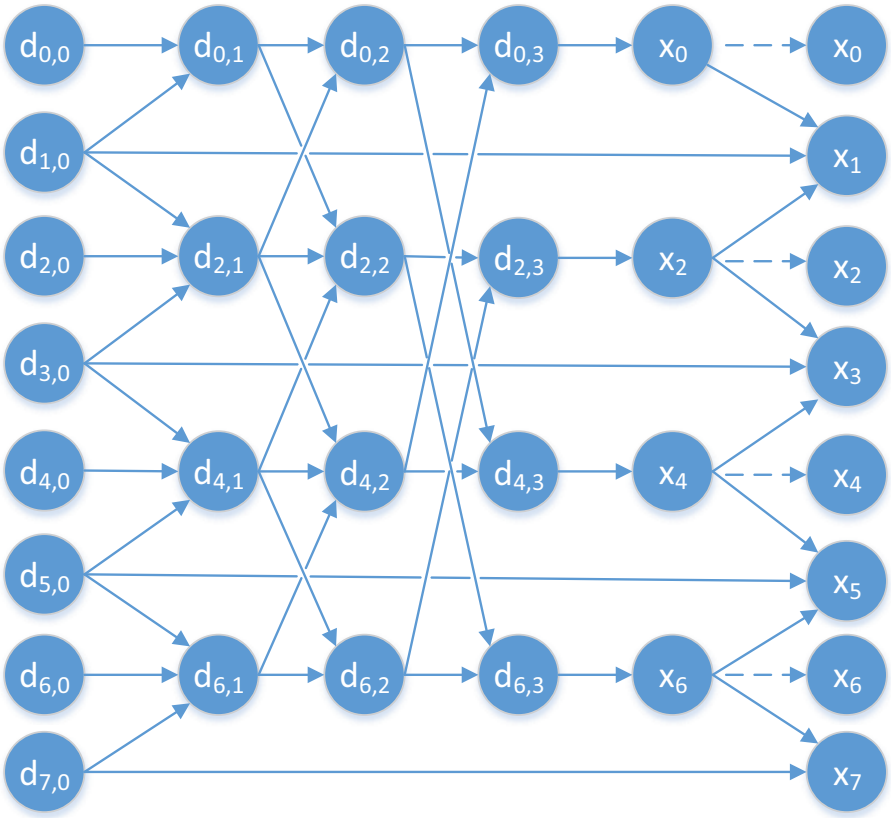


Figure 4.3: A visualization of hybrid cyclic reduction for a vector of size eight. There is one reduction step with back substitution, the other steps are done without back substitution.

steps that operate on the matrix only can be calculated offline. This means that all expressions except those containing d_i or x_i can be calculated beforehand.

For now, it is assumed that the matrix to be solved has a number of rows and columns equal to some power of 2. This means that it can be solved in exactly $1 + \log N$ steps using cyclic reduction without back substitution and in exactly $1 + \log 2N$ steps with back substitution. These are the recurrence relations that result for cyclic reduction without back substitution.

The initial condition:

$$\begin{aligned} a_{n,0} &= a_n \\ b_{n,0} &= b_n \\ c_{n,0} &= c_n \\ d_{n,0} &= d_n \end{aligned}$$

The recurrences:

$$\begin{aligned} a_{n,k} &= -\frac{a_{n,k-1}a_{n-2^{k-1},k-1}}{b_{n-2^{k-1},k-1}} \\ b_{n,k} &= b_{n,k-1} - \frac{a_{n,k-1}c_{n-2^{k-1},k-1}}{b_{n-2^{k-1},k-1}} - \frac{c_{n,k-1}a_{n+2^{k-1},k-1}}{b_{n+2^{k-1},k-1}} \\ c_{n,k} &= -\frac{c_{n,k-1}c_{n+2^{k-1},k-1}}{b_{n+2^{k-1},k-1}} \\ d_{n,k} &= d_{n,k-1} - \frac{a_{n,k-1}d_{n-2^{k-1},k-1}}{b_{n-2^{k-1},k-1}} - \frac{c_{n,k-1}d_{n+2^{k-1},k-1}}{b_{n+2^{k-1},k-1}} \end{aligned}$$

The final result:

$$x_n = \frac{d_{n,\log_2 N}}{b_{n,\log_2 N}}$$

Of these recurrence relations, only $d_{n,k}$ and x_n need to be calculated for every step. The other relationships depend only on parameters which are constant over time and only need to be calculated once. Therefore, the number of multiplications reduces to two per element per step for $d_{n,k}$ and one multiplication per element for the final calculation of x_n. There are also two additions per element during the calculation of $d_{n,k}$. This sums up to a total of $N(1+2\log N)$ multiplications and $2N \log N$ additions. To make it more clear which calculations can be done offline, we define:

$$\begin{aligned} \mathcal{F}_{n,k} &= \frac{a_{n,k-1}}{b_{n-2^{k-1},k-1}} \\ \mathcal{G}_{n,k} &= \frac{c_{n,k-1}}{b_{n+2^{k-1},k-1}} \\ \mathcal{H}_n &= \frac{1}{b_{n,\log_2 N}} \end{aligned}$$

With these, it is possible to rewrite the relationships for $d_{n,k}$ and x_n to:

$$\begin{aligned} d_{n,0} &= d_n \\ d_{n,k} &= d_{n,k-1} - \mathcal{F}_{n,k} d_{n-2^{k-1},k-1} - \mathcal{G}_{n,k} d_{n+2^{k-1},k-1} \\ x_n &= \mathcal{H}_{n,k} d_{n,\log_2 N} \end{aligned}$$

These are the only calculations that need to be done online. The relationships for $\mathcal{F}_{n,k}$, $\mathcal{G}_{n,k}$ and $\mathcal{H}_{n,k}$ can all be calculated offline.

4.4 Conclusion

The methods that have been discussed in this section all have their advantages and disadvantages. Matrix-vector multiplication can be very fast, because of the limited number of data dependencies, but it also requires a lot of resources to get these results, because the total number of operations has worst-case complexity $\mathcal{O}(N^2)$. Also, this method does not take advantage of the fact that the matrix is tridiagonal. On the other hand, TDMA requires a smaller number of total operations, but a larger number of sequential steps. Both these factors have worst-case complexity of $\mathcal{O}(N)$.

Algorithm	**#Operations**	**#Sequential steps**
Matrix-vector multiplication	$\mathcal{O}(N^2)$	$\mathcal{O}(\log N)$
TDMA	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Cyclic reduction with backsubstitution	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$
Cyclic reduction without backsubstitution	$\mathcal{O}(N \log N)$	$\mathcal{O}(\log N)$

Table 4.1: Comparison of worst-case complexity of the different algorithms

Cyclic reduction seems to be better suited for an FPGA, because it seeks a balance between parallelism and resource usage. The total number of operations has worst-case complexity $\mathcal{O}(N)$, just as TDMA, but the number of sequential steps has worst-case complexity of $\mathcal{O}(\log N)$, a large improvement compared to TDMA. It is also not that much worse than matrix-vector multiplication, because full parallelism with matrix-vector multiplication for a reasonable sized matrix is not possible, as it would use too much resources. Cyclic reduction without back substitution has a factor two less steps than cyclic reduction with back substitution, thus it is still in the complexity class $\mathcal{O}(\log N)$. However, its total number of operations increases to $\mathcal{O}(N \log N)$. Which of these two is best depends on the resources that are available.

Therefore, cyclic reduction is best suited for an implementation of this problem on an FPGA. For implementation, a mixture of cyclic reduction with back substitution and without back substitution can be used, depending on the resources that are available on the FPGA.

5 Transformations

This chapter discusses some of the possible transformations that can be applied to the recurrence relations. These transformations are applied with a possible hardware implementation in mind. The goal is to transform the recurrence relations in such a way that an efficient implementation on hardware becomes possible. Applying a transformation to a recurrence relation leads to a new recurrence relation which is mathematically equal to the first one, but could lead to a different, perhaps more efficient, implementation on hardware.

5.1 Recurrence relations

Recall the original recurrence relations as found in chapter 3 for the cochlea and chapter 4 for cyclic reduction in section 4.3.3. For the rest of this chapter, we assume cyclic reduction without back substitution to be used as method for solving the system of equations. The transformations can be applied similarly to cyclic reduction with back substitution or a hybrid version, because this is just a different method of solving the system of equations.

The initial conditions of the speed and displacement of the membrane are set to zero:

$$\begin{aligned} \tilde{u}_{0,n} &= 0 && 0 \leq n \leq N \\ \tilde{y}_{0,n} &= 0 && 0 \leq n \leq N \end{aligned}$$

To calculate the speed and displacement of the membrane for the next timestep, the values of the previous timestep are taken and combined to form $g_{t,n}$ and then combined with the input i_t to form $r_{t,n}$.

$$g_{t,n} = \mathcal{U}_n \cdot \tilde{u}_{t-1,n} + \mathcal{Y}_n \cdot \tilde{y}_{t-1,n} \qquad t \geq 1 \wedge 0 \leq n \leq N$$

$$\begin{aligned} r_{t,0} &= \mathcal{R}_0 \cdot (i_t + g_{t,0}) && t \geq 1 \\ r_{t,n} &= \mathcal{R}_n \cdot g_{t,n} && t \geq 1 \wedge 0 < n \leq N \end{aligned}$$

The values for $r_{t,n}$ are the right-hand side of the set of linear equations which need to be solved. The recurrence relations for cyclic reduction are as follows:

$$\begin{aligned} d_{t,n,0} &= r_{t,n} \\ d_{t,n,k} &= d_{t,n,k-1} - \mathcal{F}_{n,k} d_{t,n-2^{k-1},k-1} - \mathcal{G}_{n,k} d_{t,n+2^{k-1},k-1} \\ \tilde{\phi}_{t,n} &= \mathcal{H}_{n,k} d_{t,n,\log_2 N} \end{aligned}$$

The solved system can then be used to calculate the new speed and displacement of the membrane:

$$\begin{aligned} \tilde{u}_{t,0} &= \tilde{u}_{t-1,0} + \mathcal{V}_0 \cdot (\tilde{\phi}_{t,0} - g_{t,0} - i_t) && t \geq 1 \\ \tilde{u}_{t,n} &= \tilde{u}_{t-1,n} + \mathcal{V}_n \cdot (\tilde{\phi}_{t,n} - g_{t,n}) && t \geq 1 \wedge 0 < n \leq N \end{aligned}$$

$$\tilde{y}_{t,n} = \tilde{y}_{t-1,n} + \Delta\tilde{t} \cdot \tilde{u}_{t,n} \qquad t \geq 1$$

This is the complete set of equations that needs to be transformed and implementated on hardware. In the rest of this chapter, we will apply the transformations on these equations.

5.2 Lifting

We define lifting as an aggregate operation: it takes some function f which expects some inputs and transforms it into another function which expects a vector of inputs and applies the original function f to every element in the vector. Thus, instead of applying the function only once to one input, it is applied multiple times to different inputs. Lifting is an important transformation, because it allows grouping similar computations together.

As can be seen in the recurrence relations, the first segment of the cochlea is treated differently than the rest of the segments, because this is where the input is applied. However, the other segments are treated identically. Therefore, we can lift the functions for segments of the cochlea. The lifted functions that we specify here only hold for segments with index 1 and above, because the first was treated a little differently, but we will later show how to include the first segment. It is not very difficult, because the structure of the computation does not differ much from the other segments. The recurrence relations here are indexed with t only, because the function implicitly works on all n. All relationships are vectors of length $N + 1$. Names in bold indicate that we are dealing with vectors instead of scalars. The normal addition and multiplication functions are also defined to operate elementwise on two vectors. The shift function ($\ll$ and $\gg$) takes a vector as left operand and an integer i as right operand. It shifts the vector i places to the left or right and inserts zeros in the empty places. Last, the tildes on names are now omitted to make the relationships easier to read.

The initial conditions of the membrane are still zero for speed and displacement, but now they are defined as a vector (bold).

$$\begin{aligned} \boldsymbol{u}_0 &= \mathbf{0} \\ \boldsymbol{y}_0 &= \mathbf{0} \end{aligned}$$

The relations for $g_{t,n}$ and $r_{t,n}$ are also defined as vectors after the lifting operation.

$$\boldsymbol{g}_t = \boldsymbol{\mathcal{U}} \cdot \boldsymbol{u}_{t-1} + \boldsymbol{\mathcal{Y}} \cdot \boldsymbol{y}_{t-1} \qquad t \geq 1$$

$$\boldsymbol{r}_t = \boldsymbol{\mathcal{R}} \cdot \boldsymbol{g}_t \qquad t \geq 1$$

The cyclic reduction relations have also been lifted. In the non-lifted equations for cyclic reduction, some slices were indexed like $d_{t,n-2^{k-1}}$ or $d_{t,n+2^{k-1}}$. This can be represented in vector notation by shifting the vector to the right or to the left.

$$\begin{aligned}
\boldsymbol{d}_{t,0} &= \boldsymbol{r}_t \\
\boldsymbol{d}_{t,k} &= \boldsymbol{d}_{t,k-1} - \boldsymbol{\mathcal{F}}_k(\boldsymbol{d}_{t,k-1} \gg 2^{k-1}) - \boldsymbol{\mathcal{G}}_k(\boldsymbol{d}_{t,k-1} \ll 2^{k-1}) \\
\boldsymbol{\phi}_t &= \boldsymbol{\mathcal{H}}_k \boldsymbol{d}_{t,\log_2 N}
\end{aligned}$$

The relations for $u_{t,n}$ and $y_{t,n}$ are also lifted:

$$\boldsymbol{u}_t = \boldsymbol{u}_{t-1} + \boldsymbol{\mathcal{V}} \cdot (\boldsymbol{\phi}_t - \boldsymbol{g}_t) \qquad t \geq 1$$

$$\boldsymbol{y}_t = \boldsymbol{y}_{t-1} + \boldsymbol{\Delta t} \cdot \boldsymbol{u}_t \qquad t \geq 1$$

5.3 Folding

Folding is a transformation for minimizing the number of functional blocks in a design. The transformation allows similar operations to be scheduled on a single component. For example, suppose we first have N identical functional units, without data dependencies between any of them, which can process input in 1 time step. If folding is used N times, this can be transformed into 1 functional unit where the processing takes N time steps. Alternatively, it can be folded $\frac{N}{2}$ times, which results in $\frac{N}{2}$ functional units and $\frac{N}{2}$ processing time.

The dataflow graph that results from the equations above can be found in figure 5.1. It is clear that this dataflow graph cannot be implemented for real-time processing as it is infinitely large: for every instance of the computation (even for every time step) a different functional unit is used. To fit this on a device, the dataflow graph first needs to be folded.

Even though folding can be used to minimize the number of functional blocks, it also has several drawbacks. Mainly, the introduction of folding can increase the number of multiplexers and registers in the design. Registers need to be introduced to store intermediate results and multiplexers are required for switching different operation paths. Therefore, when folding there should be a consideration between number of functional units and register/multiplexer usage.

5.3.1 Folding over time

Folding over timestep t is straightforward and is used implicitly for all FPGA designs. It does not increase the number of timesteps that the calculation takes, because there is a data dependency between timestep t and $t+1$. It does decrease the size of the design though. By

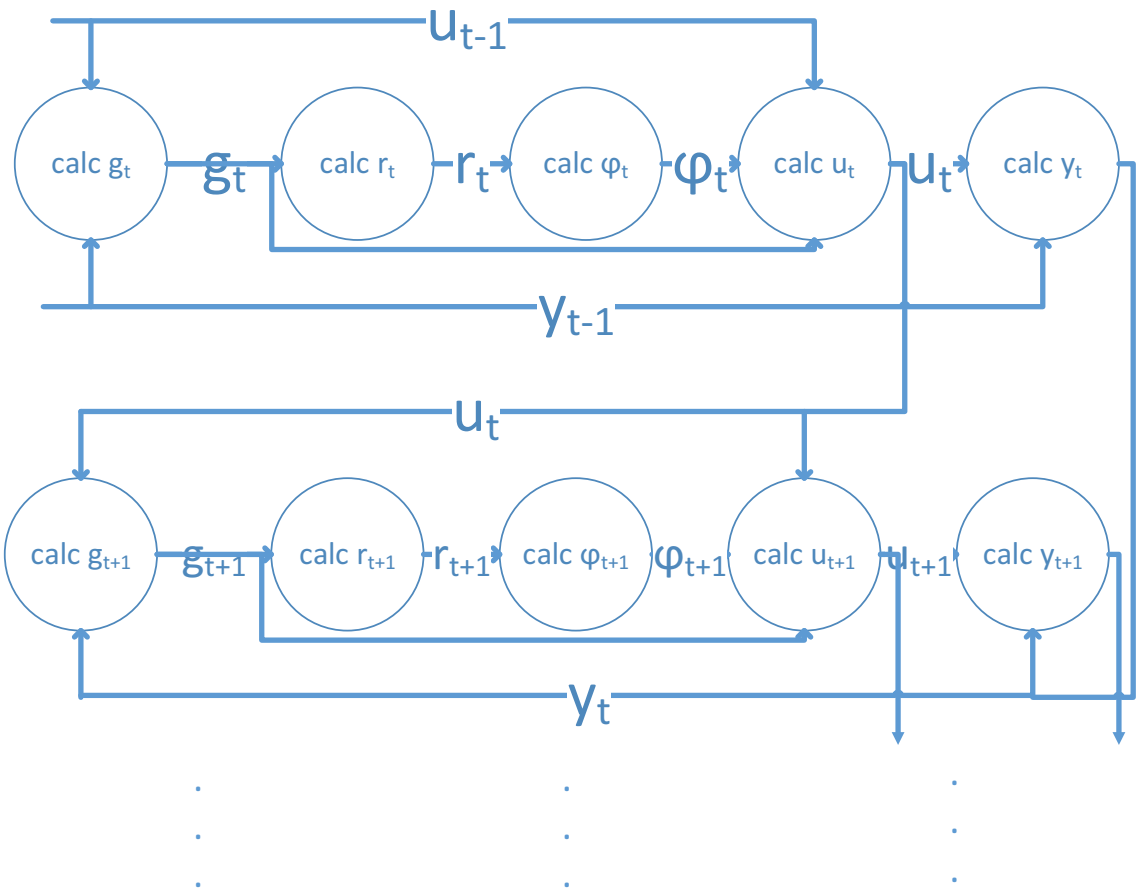


Figure 5.1: The unfolded data dependency graph

introducing a delay element after the calculation of u_t and y_t, the functional units can be used in the next time step for calculation of the next output.

The folded dataflow graph can be found in figure 5.2. This transformation reduces the number of functional units and does not increase the number of multiplexers. It does however increase the number of registers, because the vectors u_t and y_t need to be stored. Thus, the increase in registers is $2MN$ where N is the number of partitions of the cochlea and M is the number of bits of one element.

5.3.2 Folding equations

The folded design does the following calculations each time step for a model of the cochlea with N partitions:

- Calculation of $\boldsymbol{g}_t$: $2N$ multiplications, N additions
- Calculation of $\boldsymbol{r}_t$: N multiplications
- Calculation of $\boldsymbol{d}_t$: $\log_2 N$ steps with $2N$ multiplications, $3N$ additions per step
- Calculation of $\boldsymbol{\phi}_t$: N multiplications
- Calculation of $\boldsymbol{u}_t$: N multiplications, $2N$ additions (or $2N$ multiplications and $3N$ additions when expanding the expression in parentheses)

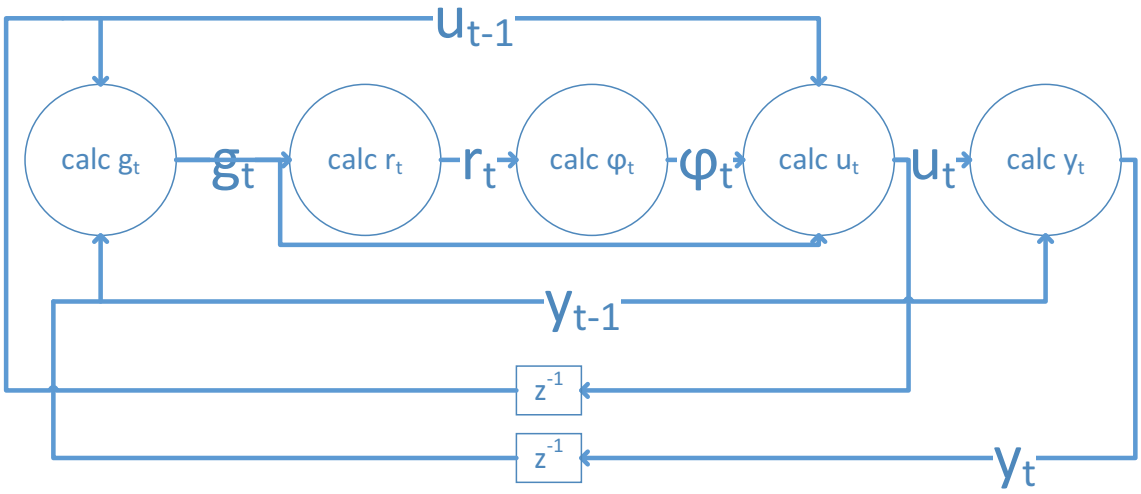


Figure 5.2: The data dependency graph when folded over time

- Calculation of $\boldsymbol{y}_t$: N multiplications, N additions

Thus, the total number of multiplications sums up to $6N + 2N \log_2 N$ and the total number of additions sums up to $4N + 3N \log_2 N$. For a realistic number of slices, for example $N = 128$, the total number of computations is very large. If these are all performed by different functional units, the design will fit on most FPGAs.

Therefore, we would like to be able to fold the design further. Another possible way of folding is to break the calculation of $\boldsymbol{u}_t$ and $\boldsymbol{y}_t$ for one timestep apart in substeps of calculating g_t, r_t, $\phi_{t,i}$ and then finally $\tilde{u}$ and $\tilde{y}$. The first step to make folding possible is to ensure that the calculation of all these steps use some common function. This will then become the functional unit that is shared among the computations.

From the recurrence relations it can be observed that the calculation in every step can be done with two multiplications and three additions, when these are used in such a way that every recurrence relation can be written in the form of:

$$f(u, v, w, x, y) = u \cdot v + w \cdot x + y \tag{5.1}$$

This general structure is chosen, because all calculations that need to be done can be written in such a way that they fit in this structure. Another possible structure would be $f(u, v, w, x, y, z) = u \cdot v + w \cdot x + y \cdot z$, which is more regular. However, it would require one more multiplication. Therefore, $f(u, v, w, x, y) = u \cdot v + w \cdot x + y$ is chosen as structure. In its lifted form, this is written as:

$$\widehat{f}(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{u} \cdot \boldsymbol{v} + \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{y} \tag{5.2}$$

Now, the recurrence relations can be rewritten to use this function f instead of their normal multiplications and additions. When one step does not use all operations, for example the calculation of $\boldsymbol{r}_t$ which only uses one multiplication per element, the remaining parameters of the function f can be set to zero.

$$\begin{aligned}\boldsymbol{u}_0 &= \mathbf{0}\\ \boldsymbol{y}_0 &= \mathbf{0}\end{aligned}$$

$$\boldsymbol{g}_t = \widehat{f}(\boldsymbol{\mathcal{U}}, \boldsymbol{u}_{t-1}, \boldsymbol{\mathcal{Y}}, \boldsymbol{y}_{t-1}, \mathbf{0})$$

$$\boldsymbol{r}_t = \widehat{f}(\boldsymbol{\mathcal{R}}, \boldsymbol{g}_t, \mathbf{0}, \mathbf{0}, \mathbf{0})$$

$$\begin{aligned}\boldsymbol{d}_{t,0} &= \boldsymbol{r}_t\\ \boldsymbol{d}_{t,k} &= \widehat{f}(-\boldsymbol{\mathcal{F}}_k, \boldsymbol{d}_{t,k-1} \gg 2^{k-1}, -\boldsymbol{\mathcal{G}}_k, \boldsymbol{d}_{t,k-1} \ll 2^{k-1}, \boldsymbol{d}_{t,k-1})\\ \boldsymbol{\phi}_t &= \widehat{f}(\boldsymbol{\mathcal{H}}_k, \boldsymbol{d}_{t,\log_2 N}, \mathbf{0}, \mathbf{0}, \mathbf{0})\end{aligned}$$

$$\boldsymbol{u}_t = \widehat{f}(\boldsymbol{\mathcal{V}}, \boldsymbol{\phi}_t, -\boldsymbol{\mathcal{V}}, \boldsymbol{g}_t, \boldsymbol{u}_{t-1})$$

$$\boldsymbol{y}_t = \widehat{f}(\boldsymbol{\Delta t}, \boldsymbol{u}_t, \mathbf{0}, \mathbf{0}, \boldsymbol{y}_{t-1})$$

This folding technique does have additional complexity to select the right input for the function f and storing the output in the correct place. Thus, the number of multiplexers and the number of memory elements increases. However, the number of functional units decreases greatly, because what used to be done with $N \cdot (1+1+\log_2 N+1+1+1) = N \cdot (5+\log_2 N)$ functional units (1 for g_t, r_t, ϕ_t, u_t and y_t and $\log_2 N$ for $d_{t,k}$) can now be done with N functional units if the function f is reused. Figure 5.3 shows this folding technique graphically.

5.3.3 Folding over n

Calculating the result of the function $\widehat{f}$ can be done in parallel for every n, because there are no data dependencies between elements. Therefore, when there is enough area it is generally not desirable to fold over n. However, for large values of N, it may be useful to fold over n as well, because there are not enough multipliers available to do everything in parallel.

For example, the calculation of function $\widehat{f}$ for $N = 128$ can be folded into two steps: the first step calculating $n = 0..63$ and the second step calculating $n = 64..127$. Similarly, for larger values of N, it can be broken down into 4, 8 or more blocks of equal size. Again, this folding procedure comes at the cost of extra multiplexers, but reduces the number of functional units that are required.

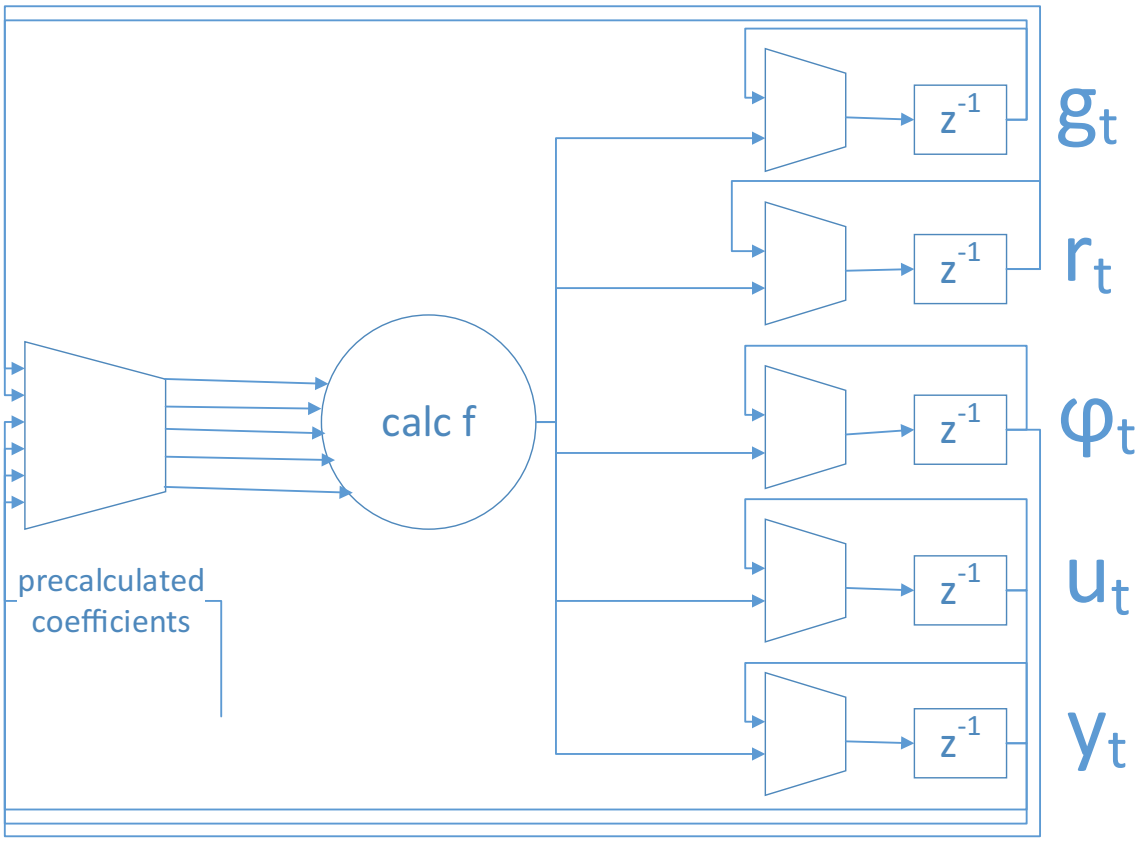


Figure 5.3: The data dependency graph when folding the equations

5.3.4 Hybrid cyclic reduction

When the number of functional units required is greater than the number of functional units available, folding over n is needed. Folding over n is possible for all steps in the process. However, for calculating ϕ_t a better method is available. It is not a transformation to the current equations, but it is noted in this chapter anyway, because it corresponds with folding over n. Recall our discussion of cyclic reduction in chapter 4. In all previous equations we assumed that cyclic reduction without back substitution was used to solve the system of equations. Cyclic reduction without back substitution is superior to cyclic reduction with back substitution whenever one step can be done completely in parallel, because it requires fewer number of steps.

However, when one step cannot be done completely in parallel, as is the case when folding over n is used, then it is better to use cyclic reduction with back substitution. More specifically, cyclic reduction with back substitution can be used up until the point that the matrix is reduced to a size that can be handled competely in parallel. From that point, the reduced matrix can be solved using cyclic reduction without back substitution. This hybrid form of cyclic reduction is described in chapter 4 and will minimize the number of steps that are needed to solve the system of equations.

Suppose that $N = 128$ and that 32 elements can be processed in parallel. Cyclic reduction without back substitution normally takes $1 + \log_2 N = 8$ sequential steps when 128 elements can be processed in parallel (see chapter 4). However, because every step has to be split in four it now takes $8 \cdot 4 = 32$ steps.

Suppose that instead of using cyclic reduction without back substitution, hybrid cyclic reduction is used. An illustration of hybrid cyclic reduction can be found in chapter 4 in figure 4.3. First the matrix is reduced to 64 and subsequently to 32. Next, the remaining matrix of 32 is solved with cyclic reduction without back substitution in $1 + \log_2 32 = 6$ steps. Last, the back substitution step is used to obtain first a matrix of 64 and then 128. The first reduction and last back substitution step have to be split in two, because only 32 elements can be processed at a time. The second reduction and first back substitution only take one step. Thus, the total number of steps when hybrid cyclic reduction is used sums up to $2 + 1 + 6 + 1 + 2 = 12$: a big reduction compared to the 32 required for cyclic reduction without back substitution.

6 Implementation

This chapter will discuss the implementation of the algorithm on an FPGA. The last chapter ended with the transformed recurrence relations which can be translated quite straightforwardly to CλaSH. However, there are still several issues depending on the exact implementation requirements. In this chapter the implementation requirements are described. Also, the problems that arise when converting the mathematical model to fixed-point are described. The mathematics assume infinite precision which is not possible in hardware. There are either floating point or fixed point computations. This section will discuss the possibilities and choose a suitable fixed point range. Last, the actual conversion to CλaSH is discussed.

6.1 Implementation requirements

There are several implementation requirements. Not only requirements like the type of FPGA that it needs to be implemented on, but also model parameters like the number of sections of the cochlea. The requirements consist of:

- The parameter N, the number of sections of the cochlea minus one (the input)
- The type of FPGA that the algorithm needs to run on. This determines the number of multipliers, M, that are available and also the are that is available for logic elements.
- The range of the input
- The frequency of the input
- The range of the output
- The frequency of the output

The number of sections of the cochlea determines to a large extent the size of the implementation: doubling the number of sections more than doubles the total number of operations. Furthermore, the larger the parameter N, the more the design needs to be folded to fit with the available number of hardware multipliers. Simulations at INCAS^3 have shown that sound recognition would still be possible for 128 sections, thus $N = 127$, so this is what is chosen for N. A smaller number of sections would make the signal too imprecise for recognition. A greater number of sections would not make the signal that much better to justify the greater number of computations required.

The algorithm has to run on the Xilinx ZYNQ-7000 board, which contains a XC7Z020 Artix-7 FPGA. This board was chosen, because it also contains an ARM CPU on the chip, which can be used to analyze the output of the cochlea algorithm for sound recognition. Furthermore, it is not very expensive and has a decent FPGA with 220 18x25 multipliers.

The input comes from a soundboard which delivers samples of 24 bits. The sample frequency is set to be 192 kHz as simulations at INCAS^3 have shown that this is a reasonable sample frequency. It is oversampled, because this makes the numerical integration methods more

stable. The output of the model is the speed and displacement of the membrane in a precision equal to the precision that the calculations are done in. The output frequency is equal to the input frequency.

6.2 Fixed point

All calculations in the mathematics are assumed to be done with infinite precision. Unfortunately, when implementing such algorithm on real hardware or software, there is always a limitation to the precision. Software generally uses 32-bit or 64-bit floating point precision, which is enough for almost all tasks. However, in hardware, fixed-point is much more common due to the larger area costs of the implementation of floating point calculations. Making full-blown floating point multipliers is therefore not a suitable option. Then, two options remain:

- Fixed-point calculations with one fixed-point domain
- Fixed-point calculations with multiple fixed-point domains

An advantage of the first approach is that multipliers can be easily shared between different calculations, because all calculations are in the same domain. After each multiplication, the results needs to be shifted back by the same number of bits. A disadvantage is that all calculations need to be in the same range. If this is not the case and different calculations have different ranges, then the number of bits required needs to increase to fit all calculations in the same fixed-point range.

The second approach makes it more difficult to share multipliers, because there is extra logic involved to shift back the result by the correct number of bits. However, a smaller number of bits is required in total, because calculations with different ranges can be done in different fixed-point domains.

Becaue it is necessary to share multipliers, fixed-point with only one domain is chosen for this algorithm. In order to find the correct number of integer bits and fractional bits that are necessary, simulations have been done with an existing Matlab script. This Matlab script was written at INCAS^3 and simulates the model. It would also have been possible to write this script in, for example, Haskell, but because a Matlab script already existed this was the preferred option. Also, Matlab has excellent easy-to-use plotting libraries.

Plots were created using the Matlab script which show the effect of several fixed-point options for different input signals. These plots are shown in figure 6.1 through figure 6.4. The input signals are all sine waves at 1 KHz but with different amplitudes. The fixed-point format has to be able to handle both small and large signals with a small error. Whether or not the error is small enough is determined qualitatively by examining the plots. If two lines do not deviate much, then the error is small enough.

One step in the calculation of the algorithm in Matlab is still done in floating point: this is the calculation of ϕ. For the implementation on the FPGA, cyclic reduction will be used in fixed-point. However, the Matlab script uses simple matrix-vector multiplication in floating point. Converting the matrix-vector multiplication to fixed-point yields results much worse than cyclic reduction, because the inverse of the tridiagonal matrix has a lot of small elements. These small elements contribute to the final result in floating-point, but are all rounded to zero in fixed-point. Therefore, it is chosen to do the matrix-vector multiplication in floating

point and use the results as a best-case scenario for fixed-point results: cyclic reduction will also introduce a small error, but this will be smaller than the errors in the other calculations.

A comparison of the Matlab fixed point and floating point implementations for several sine input signals is found in the figures below. The figures show plots for the speed of the membrane, u_t, and one for the displacement, y_t. Simulations have shown that in order to simulate very small as well as very large input signals, a word length of 25.15 is needed: 25 for the integer part (twos complement) and 15 for the fractional part. Because this is a very large word length, the choice has been made to restrict the amplitude of the input signal to reduce resource usage. If the word length is restricted to 17.15, signals between 25 dB SPL and 85 dB SPL can be processed accurately.

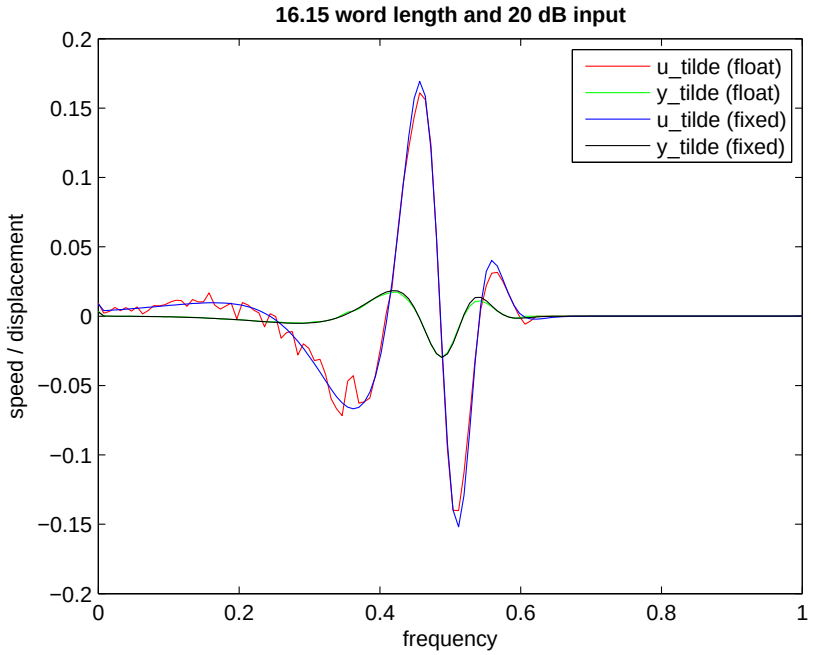


Figure 6.1: Comparison between fixed-point and floating-point of a 20 dB SPL input signal

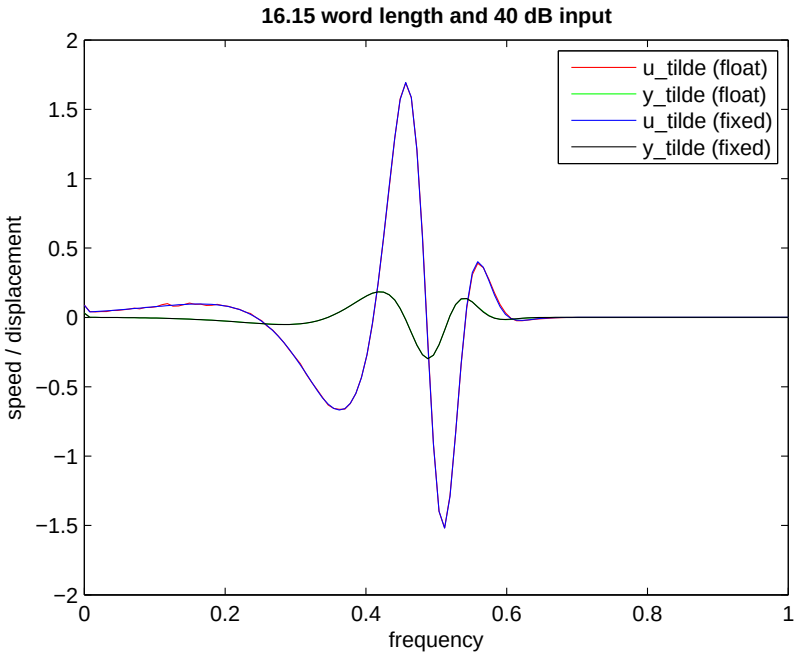


Figure 6.2: Comparison between fixed-point and floating-point of a 40 dB SPL input signal

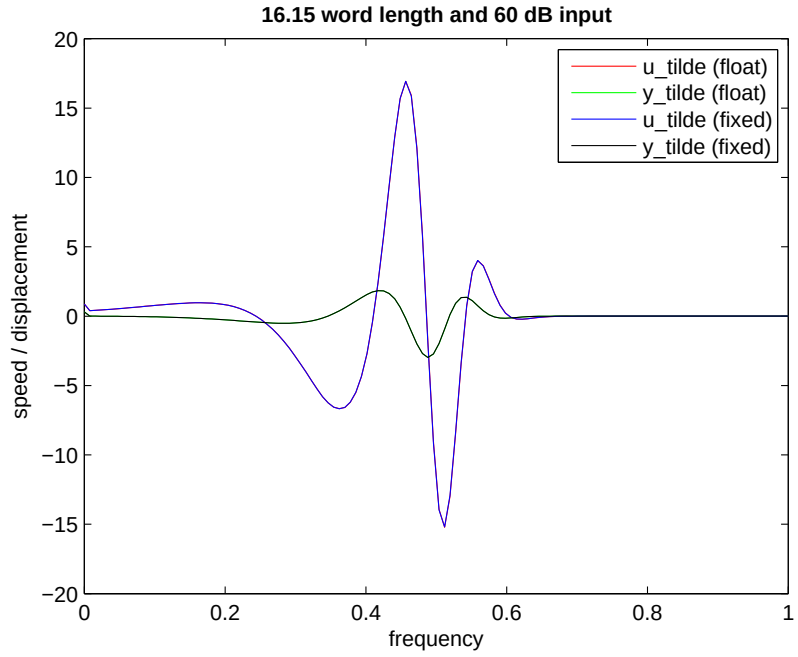


Figure 6.3: Comparison between fixed-point and floating-point of a 60 dB SPL input signal

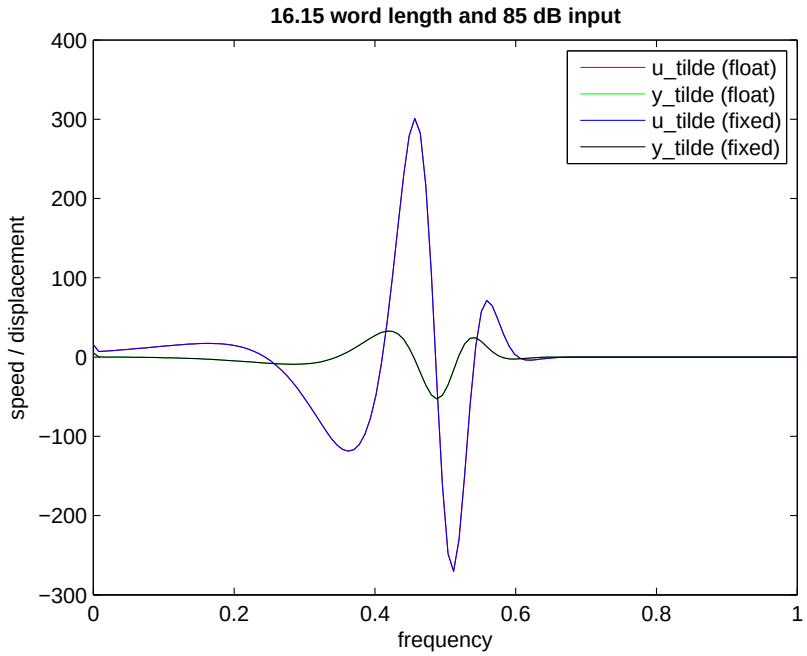


Figure 6.4: Comparison between fixed-point and floating-point of a 85 dB SPL input signal

6.3 Conversion to CλaSH

Using the the word length and the FPGA requirements, it is possible to make an implementation in CλaSH based on the equations from chapter 5. Recall that the transformations were applied in order to obtain a more efficient implementation on hardware. The level of folding that is required depends on the number of functional units that are available. Thus, first we need to determine how many multipliers our design would require and how many are available.

The FPGA has 220 25x18 bits multipliers. Because of our large choice of word length, every multiplication is 32x32 bits wide. In order to use hardware multipliers, four 25x18 bits multipliers are needed for one 32x32 bits calculation. There are 220 multipliers available and the function f needs two multiplications of which every multiplication takes four multipliers. Thus, the maximum number of elements that can be processed in parallel is $\frac{220}{2 \cdot 4} = 27.5$. Rounding down to a number that is divisible by 128, the total number of elements of the cochlea, we get 16 elements in parallel at a time. With this, it takes eight steps for one calculation of $\widehat{f}$ over 128 elements.

Thus, the transformations that are needed to reduce the number of functional units required are as follows:

- Folding over t
- Folding the equations
- Folding over n by a factor 8

Also, hybrid cyclic reduction needs to be used with three steps back substitution (the reduction from 128 to 16).

It should be clear that it requires quite a lot of folding to make it fit for the number of multipliers on the FPGA. The number of functional units available is one of the major limitations, because the algorithm requires a lot of functional units. The aggresive folding strategy will lead to a large increase of multiplexers. This could make the number of logic elements or the wiring during place and route a limitation. However, an exact estimation cannot be given at this point in the design process.

The resulting CλaSH design is similar to figure 5.3, except that it is further folded over n by a factor of 8. The CλaSH code corresponds almost one on one with this design and is straightforward to implement. The function f is taken as a basic block. It takes five arguments which it multiplies and adds together. For every clock cycle, the correct inputs to the function f are selected depending on which computation we want to do that clock cycle. The computation is done by mapping the function f over all the inputs. After the computation, the results are written to the corresponding registers.

6.3.1 Generalization for M and N

Suppose there are M functional units available each of which can perform function f with two multiplications and three additions and the cochlea is divided into N partitions. Which transformations need to be applied to reduce the number of functional units required to the number that are available?

Folding over time needs to be applied anyway, as we noted in the previous chapter that this is a transformation that all FPGA designs implicitly use. Furthermore, if $M < N \cdot (5 + \log_2 N)$ some form of folding the equations needs to be used, because there are not enough functional units to dedicate one functional unit to one calculation in the process. Also, if $M < N$ then folding over n needs to be applied as well and it becomes beneficial to use one or more steps of cyclic reduction with back substitution.

Ideally, we would like to be able to define a function in CλaSH that takes M and N as arguments and synthesizes the best implementation with the least amount of folding that still fits with the given number of functional units. This is preferred to manually writing the CλaSH code for one specific M and N for two reasons. First, when synthesizing for a different FPGA with a different number of multipliers, or for a different N, the code would need to be rewritten competely. Second, in writing this manually, it is easy to introduce subtle errors which are hard to detect and this is exactly what we would like to avoid. Unfortunately, because it is quite a complex function it is not possible to write such function in CλaSH itself, because of type restriction. Therefore, a script was written that generates CλaSH code that is needed for a given M and N. The script generates all the precalculated constants and logic that are needed by the algorithm. The resulting CλaSH code can be fed to the CλaSH compiler to produce synthesizeable VHDL.

With this script, all that needs to be done is to input the variables M and N and then it will generate the CλaSH code. This code can be tweaked manually if needed after which the CλaSH compiler converts it to VHDL.

6.3.2 The CλaSH code

This section discusses the structure of the generated CλaSH code. This section will take parts of the code and explain it.

The script generates 5 files:

- `Cochlear128.hs`, the main file. It contains the function that describes the top entity of the CλaSH design.
- `Types.hs`, contains the type definitions that are used in the CλaSH code.
- `CoeffsCochlear.hs`, a file containing vectors of constants used in the calculation of g, r, u and y.
- `CoeffsFull.hs`, a file containing vectors of constants used in the cyclic reduction steps for cyclic reduction without back substitution.
- `CoeffsHalf.hs`, a file containing vectors of constants used in the cyclic reduction steps for cyclic reduction with back substitution.

Because the latter three files only contain constants, these are for the most part self-explanatory. The file `Types.hs` defines the types that are used in the design. It contains the following important type definitions:

- `CountType`, defined as an unsigned number consisting of enough bits to count up to the maximum number of clock cycles needed for calculation one time step of the algorithm
- `Sample`, a signed 32-bits number. This type is used for all calculations.
- `SampleVect`, a vector of N `Sample`s. This type is used for storing for example the vector u_t and y_t.
- `CoeffVect`, a vector of `Sample`s which has a size equal to the maximum number of calculations of f that can be done in parallel. The generated vectors of constants have this type.

The interesting file is `Cochlear128.hs` which contains the actual design. Just like any CλaSH clocked design, it is defined as a Mealy machine taking an input and state and producing an output and new state. The definition of the top entity is:

```
arch :: (SampleVect, SampleVect, SampleVect, SampleVect, CountType,
    Sample, Bit, SampleVect) -> (Sample, Bit) -> ((SampleVect, SampleVect,
    SampleVect, SampleVect, CountType, Sample, Bit, SampleVect), (Sample,
    Bit))
arch ((u, y, g, rphi, i, inp, prevInpReady, result)) (input, inpReady) =
    ((u', y', g', rphi', i', inp', inpReady, result'), (output, ready))
```

The important types here are the four leading `SampleVect`s in the state which store the g, r and ϕ (shared), u and y. It is a vector of size N. The `CountType` stores a simple counter to control which calculations are performed. The input consists of a `Sample`and `Bit` which are the input sample and a bit to indicate whether the offered input is currently valid.

The implementation is just as figure 5.3 suggests. There is a block of functional units which compute the result of function f. Function f, named `calc` here, is defined as:

```
calc :: (Sample, Sample, Sample, Sample, Sample) -> Sample
calc (a,b,c,d,e) = fpmult a b + fpmult c d + e
```

It takes a tuple of five values as input and multiplies and adds them together to produce one output. Depending on variable i, the current step in the computation, different inputs to the function f have to be selected. The input can consist of one of the precalculated vectors of constants which are defined in the other files or (part) of one of the state vectors. Selection of the inputs are done with a simple guard expression in Haskell. The inputs `i1` through `i5`

are defined. An example of function `i1` is shown next. It can be seen that for different values of i, different inputs are selected.

```
i1
    | i == 0    = u -- calculation of new g
    | i == 1    = ((inp + vhead g) :> vdrop d1 g)
                  -- calculation of new r
    | i == 2    = vshiftR rphi d1 -- start cyclic reduction
    | i == 3    = vshiftR rphi d2
    | i == 4    = vshiftR rphi d4
    | i == 5    = vshiftR rphi d8
    | i == 7    = vshiftR rphi d32
    | i == 8    = vshiftR rphi d64
    | i == 9    = rphi -- end cyclic reduction
    | i == 10   = ((vhead rphi - inp) :> vdrop d1 rphi) -- new u
    | i == 11   = dt_tilde -- new y
    | otherwise = vshiftR rphi d16
```

The computation can then be defined as the `map` over the `zip` of these five inputs:

```
calc_out = vmap calc (vzip5 i1 i2 i3 i4 i5)
```

Last, the result of the computation needs to be stored in either `g'`, `rphi'`, `u'` or `y'` depending on what was calculated in that step. Another guard expression will do, for example:

```
calc_out = vmap calc (vzip5 i1 i2 i3 i4 i5)

g'    | i == 0    = calc_out
      | otherwise = g

rphi' | i >= 1 && i <= 9   = calc_out
      | otherwise          = rphi

u'    | i == 10    = calc_out
      | otherwise = u

y'    | i == 11   = calc_out
      | otherwise = y
```

7 Results

This chapter is divided in three sections. First, the results of the simulation in Haskell are presented. These results are compared to the fixed-point Matlab model that was already available. Next, the CλaSH code is compiled to VHDL with the CλaSH compiler and results of the compilation and VHDL simulation are shown. Last, the VHDL is synthesized and resource usage and other synthesis results are given.

7.1 Simulation in Haskell

The CλaSH design of the cochlea that is generated by the script as described in chapter 6 can be straightforwardly run on any machine with a Haskell compiler. This serves as a first verification step. The same input signals were tested as with the fixed-point test in chapter 6. The outputs at several timesteps t of u_t and y_t, the speed and displacement of the membrane, were compared to those of the fixed-point Matlab model.

The output of the Matlab fixed-point model will slightly differ from the output of the CλaSH design, because the Matlab model is not completely fixed-point: the cyclic reduction step is done in floating-point. Thus, the simulations should verify that the response to a specific input signal is similar to the Matlab model response within a small margin. It should especially test this for some signals close to the boundaries: with large and small amplitudes. Therefore, testing is done with three 1 kHz sine waves at different amplitudes: 40 dB, 60 dB and 80 dB.

The code to simulate the design in CλaSH is as follows:

```
topEntity = arch <^> (vcopyI 0, vcopyI 0, vcopyI 0, vcopyI 0, 0, 0, L,
    vcopyI 0)

inpbit = concat (repeat (L:(take (numCycles-1) (repeat H))))
inpvec = concat (map (\x->take numCycles (repeat x)) inpv)
myinp = zip inpvec inpbit

myres :: [(Sample, Bit, SampleVect, SampleVect)]
myres = simulateP topEntity myinp
```

This code assumes that `inpv` is the sampled list of inputs that represent the sine wave. The output `myres` contains u_t and y_t as last two elements of the tuple.

7.1.1 40 dB sine input

Figure 7.1 shows a plot with the output of the CλaSH design as well as the output of the Matlab model for u_t and y_t after 2000 timesteps for a 40 dB input signal. It can be seen that the lines do not exactly match, though the overall curve of the plot is the same. The peak in the middle is caused by the 1 kHz input signal.

After the peak at 1 kHz the plot for y_t does not go to zero, but stays below zero. This is caused by small integration errors in the fixed-point computation of the cyclic reduction method.

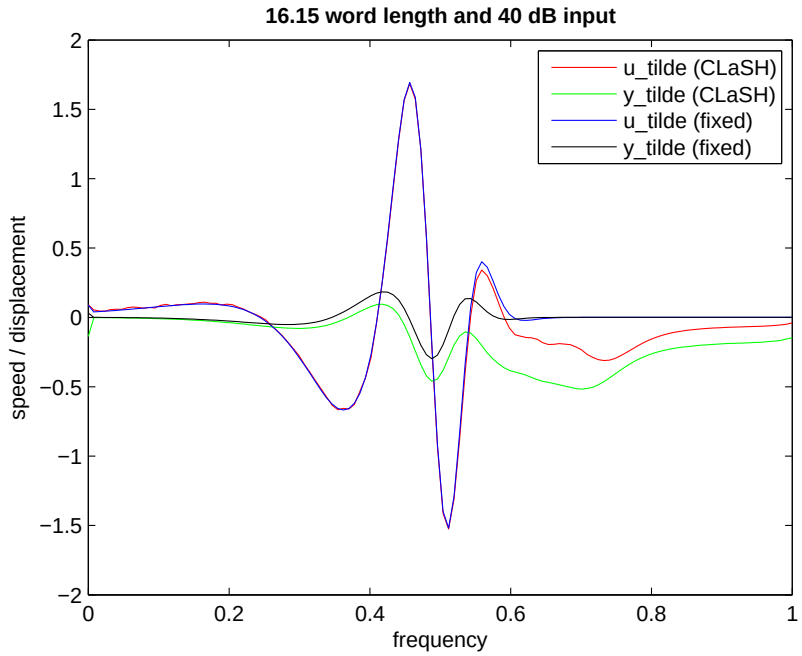


Figure 7.1: Comparison between Matlab fixed-point and CλaSH implementation of a 40 dB SPL input signal

The Matlab model does not compute the solution of the equation with cyclic reduction, but computes it by the inverse of the matrix A. In the model, this step is not calculated in fixed-point, thus the results of the Matlab model are slightly better than the full fixed-point calculation of the CλaSH model.

7.1.2 60 dB sine input

Figure 7.2 shows a plot with the output of the CλaSH design as well as the output of the Matlab model for u_t and y_t after 2000 timesteps for a 60 dB input signal. The two plots match better than the plot for the 40 dB input signal, but are still not completely identical.

7.1.3 80 dB sine input

Figure 7.3 shows a plot with the output of the CλaSH design as well as the output of the Matlab model for u_t and y_t after 2000 timesteps for a 80 dB input signal. Here, the Matlab model and CλaSH model are nearly identical.

Tests for input signals with different amplitudes showed that there is a reasonable match between 50 and 80 dB. Signals smaller than 50 dB do not converge precisely to zero, while signals larger than 80 dB experience peaks at 1 kHz which are too small.

7.2 VHDL simulation

The CλaSH-compiler can generate VHDL code from the CλaSH-code. Generally, no errors are introduced in this translation unless bugs exist in the CλaSH-compiler or hand-written VHDL

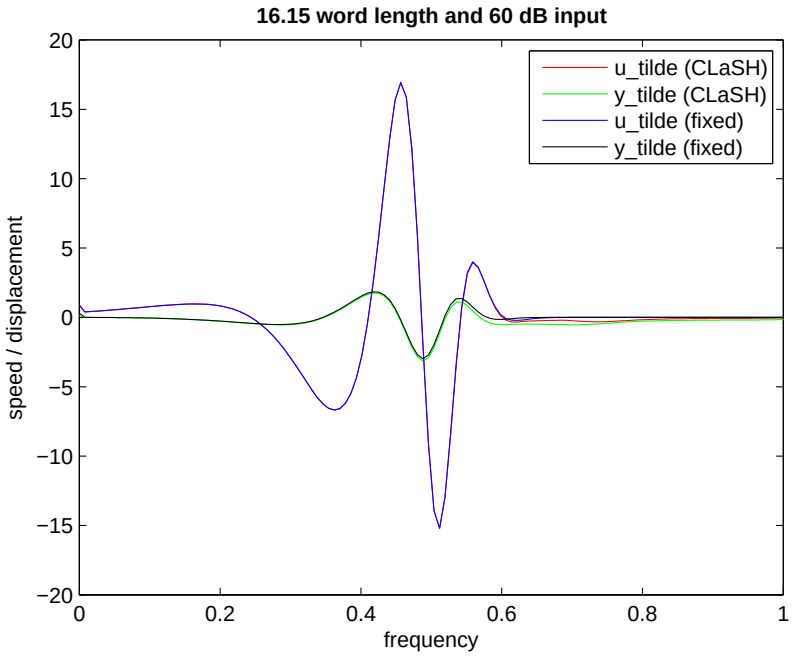


Figure 7.2: Comparison between Matlab fixed-point and CλaSH implementation of a 60 dB SPL input signal

code is added to the design. The VHDL code for 32 elements was tested in a simulation with ModelSim. The same inputs were applied as with the simulation of the CλaSH model and it was verified that the outputs at every timestep were equal to the outputs of the CλaSH simulation.

7.3 Synthesis

After simulation on both CλaSH level and VHDL level and verification showed that the design was working as expected, synthesis could be started. Synthesis of the design is needed to translate the VHDL code to a fully routed design which can be put on the FPGA. There are several constraints that need to be met, all of which are concerned with the available resources on the FPGA. These can roughly be divided into these categories:

- Number of multipliers; an FPGA has a limited number of hardware multipliers. Additional multipliers can be generated in logic, but these are much slower than the hardware multipliers. Usually it is best to stick with hardware multipliers except for simple multiplications which consists of shifts only.
- Number of configurable logic blocks (CLBs); a CLB generally consists of several logic cells which can be configured to perform a simple logic function. They can be seen as some sort of configurable lookup table. The number of CLBs an FPGA has generally limits the number of logic elements and multiplexers a design can have.
- Fan-out; the output of a logic gate can only drive a certain number of gate inputs. The maximum value can generally be obtained from the FPGAs datasheet. If a design has a large number of gates with a high fan-out, it might run into trouble when synthesizing the design.

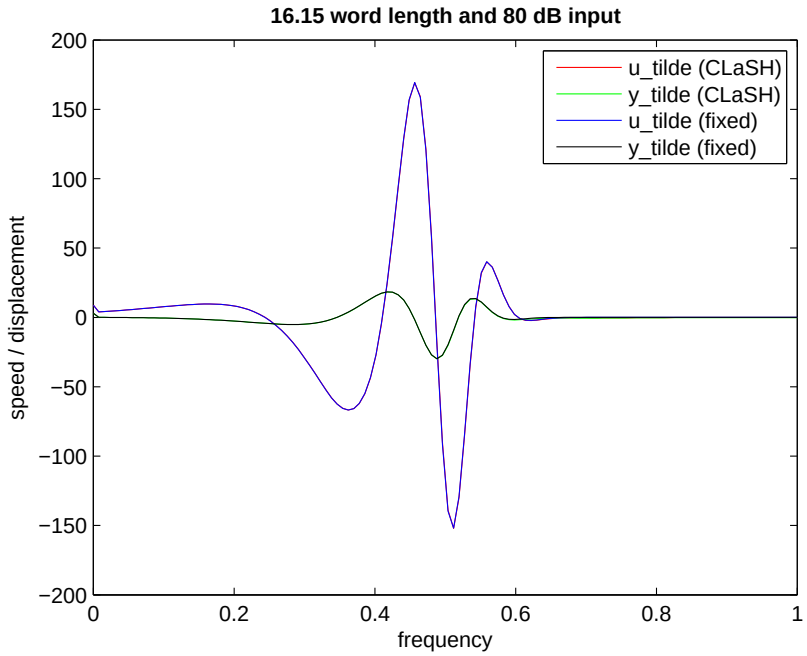


Figure 7.3: Comparison between Matlab fixed-point and CλaSH implementation of a 80 dB SPL input signal

- Wiring; designs with a lot of interconnections between logic gates, for example designs with a lot of multiplexers, might run into trouble with the wiring. Just like with CLBs, there is only a limited number of connections possible between them. Large designs might not fit because there is no routing possible for the connections between CLBs, while the number of CLBs does not exceed the maximum number available on the FPGA.

The number of multipliers can easily be constrained in the design with the methods described in the chapter about transformations. However, as noted there, reducing the number of multipliers by applying folding does increase the number of registers and multiplexers. It is not easy to manually give an estimate how much this increase in multiplexers will affect the number of CLBs and the wiring. A synthesis run is required to obtain reliable information about these numbers.

7.3.1 Xilinx ZYNQ-7000 board

The design was synthesized for the Xilinx ZYNQ-7000 board, which has a XC7Z020 Artix-7 FPGA. The design that was synthesized has 128 slices (the cochlea is divided into 128 parts) and a word length of 32 bits, as described in the previous chapter. The design can process 16 elements at a time, resulting in a multiplier usage of $16 \cdot 4 \cdot 2 = 128$ multipliers. Results of the synthesis showed that the design was too large to fit on the board. While the number of multipliers was within boundaries (220 multipliers are available), the number of LUTs needed was more than available.

As an attempt to make a design fit on the FPGA, the synthesis was also run for a design with 64 elements and one with 32 elements. In the design with 64 elements the number of elements that could be processed in parallel was kept at 16, which means that the number of multipliers remains the same as in the previous design. In the design with 32 elements, this was increased

to 32. Normally, this would require $32 \cdot 4 \cdot 2 = 256$ multipliers, however the Precision synthesis tool was able to optimize this to fit it in the 220 multipliers that are available. By lowering the number of total elements of the cochlea, the number of sequential steps decreases and thereby the number of LUTs that are required for multiplexers also decreases.

Preliminary synthesis results showed that LUT usage was around 70% for the design with 64 elements. However, place and route ran into problems while trying to route the design. Heavy multiplexer usage led to many wires, which made the place and route process too hard.

For 32 elements, the design was synthesizeable by Precision RTL. However, Xilinx ISE did not perform the multiplier optimization like Precision. Thus, it was not synthesizeable by Xilinx ISE. Results of the synthesis in Precision RTL were as follows:

```
**************************************************************
Device Utilization for 7Z020CLG484
**************************************************************
Resource                          Used    Avail   Utilization
--------------------------------------------------------------
IOs                               68      200      34.00%
Global Buffers                    2       32        6.25%
LUTs                              15302   53200    28.76%
CLB Slices                        3826    13300    28.77%
Dffs or Latches                   5157    106400    4.85%
Block RAMs                        0       140       0.00%
DSP48E1s                          212     220      96.36%
--------------------------------------------------------------

                    Clock Frequency Report

Domain                   Clock Name                              Min Period
    (Freq)             Required Period (Freq)
------                   ----------
    -----------------              ----------------------

ClockDomain0             clk                                     14.822
    (67.467 MHz)           50.000 (20.000 MHz)
```

7.3.2 Xilinx 6VLX240TFF784

Because the ZYNQ board could only fit a design with 32 elements, synthesis was also run for a larger board. The Xilinx 6VLX240TFF784 board has 768 multipliers, enough to process 64 elements in parallel or 128 elements in two steps. However, for the design with 128 slices, there were still problems with place and route. The design with 64 elements did fit on the device, with the following results. The results will further be discussed in the next chapter.

```
Device Utilization Summary:

Slice Logic Utilization:
  Number of Slice Registers:              10,285 out of 301,440    3%
    Number used as Flip Flops:            10,281
    Number used as Latches:                    0
    Number used as Latch-thrus:                0
    Number used as AND/OR logics:              4
  Number of Slice LUTs:                   34,051 out of 150,720   22%
    Number used as logic:                 33,985 out of 150,720   22%
      Number using O6 output only:        33,959
      Number using O5 output only:            14
      Number using O5 and O6:                 12
      Number used as ROM:                      0
```

```
    Number used as Memory:                        0 out of  58,400    0%
    Number used exclusively as route-thrus:      66
      Number with same-slice register load:       0
      Number with same-slice carry load:         66
      Number with other load:                     0

Slice Logic Distribution:
  Number of occupied Slices:                11,037 out of  37,680   29%
  Number of LUT Flip Flop pairs used:       34,053
    Number with an unused Flip Flop:        23,768 out of  34,053   69%
    Number with an unused LUT:                   2 out of  34,053    1%
    Number of fully used LUT-FF pairs:      10,283 out of  34,053   30%
    Number of slice register sites lost
      to control set restrictions:                0 out of 301,440    0%

IO Utilization:
  Number of bonded IOBs:                        68 out of     400   17%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:                  0 out of     416    0%
  Number of RAMB18E1/FIFO18E1s:                  0 out of     832    0%
  Number of BUFG/BUFGCTRLs:                      2 out of      32    6%
    Number used as BUFGs:                        2
    Number used as BUFGCTRLs:                    0
  Number of ILOGICE1/ISERDESE1s:                 0 out of     720    0%
  Number of OLOGICE1/OSERDESE1s:                 0 out of     720    0%
  Number of BSCANs:                              0 out of       4    0%
  Number of BUFHCEs:                             0 out of     144    0%
  Number of BUFIODQSs:                           0 out of      72    0%
  Number of BUFRs:                               0 out of      36    0%
  Number of CAPTUREs:                            0 out of       1    0%
  Number of DSP48E1s:                          367 out of     768   47%
  Number of EFUSE_USRs:                          0 out of       1    0%
  Number of FRAME_ECCs:                          0 out of       1    0%
  Number of GTXE1s:                              0 out of      12    0%
  Number of IBUFDS_GTXE1s:                       0 out of      12    0%
  Number of ICAPs:                               0 out of       2    0%
  Number of IDELAYCTRLs:                         0 out of      18    0%
  Number of IODELAYE1s:                          0 out of     720    0%
  Number of MMCM_ADVs:                           0 out of      12    0%
  Number of PCIE_2_0s:                           0 out of       2    0%
  Number of STARTUPs:                            1 out of       1  100%
  Number of SYSMONs:                             0 out of       1    0%
  Number of TEMAC_SINGLEs:                       0 out of       4    0%

 286691356612 paths analyzed, 61491 endpoints analyzed, 0 failing endpoints
 0 timing errors detected. (0 setup errors, 0 hold errors, 0 component
     switching limit errors)
 Minimum period is  31.060ns.
```

8 Discussion

The purpose of the research was twofold. On the one hand, a design of the cochlea model for an FPGA needed to be created. On the other hand, by doing this the effectiveness of CλaSH and the mathematical design methodology was tested. The discussion can also be split into these two parts. The first part discusses the results of the implementation of the cochlea model. The second part discusses the usability of the mathematical design methodology and CλaSH and the advantages and disadvantages of this methodology compared to writing VHDL directly.

8.1 Fixed-point

First the discussion of the fixed-point model versus the floating-point model. The plots in the implementation section show a large difference in input range for fixed-point versus floating-point. In order to obtain similar results for the full input range the desired fixed-point parameters were too large. Also, even for a smaller input range, the actual results of the CλaSH design were more off than the fixed-point model, because the fixed-point model still used floating-point operations in one part of the model.

We observe that this difficulty in using fixed-point is mainly due to the large dynamic range of the input and the intermediate results. In some operations of calculating the solution of the set of linear equations, very small values are needed as parameters, while the input and output can be very large. Also, the precalculated constants of most operations are usually large for slices of the cochlea close to the first slice and diminish to zero to the end. This occurs, because the damping and thus the frequency distribution across the cochlea is logarithmic. These large differences make it difficult to find good fixed-point parameters.

This makes it difficult to obtain a good design, because in fixed-point more precision leads to a larger word-length, which in turn leads to more multiplexers to get the inputs to the function blocks and to write the outputs back to the registers. However, we noted in the results section that area is already a bottleneck in the design. Further increasing area usage by increasing the word-length is not a good idea.

Because of the large required fixed-point range, it might be beneficial to use floating-point blocks instead. Using floating-point blocks would greatly reduce the required word-length and therefore would also reduce the required number of multiplexers in the design. However, this comes at the cost of the overhead of the floating-point units which are generally slower and require more area. This would be worth looking into as an improvement.

8.2 Area usage

Three designs were synthesized for the ZYNQ board, of which only the smallest design fit on the board. The designs for $N = 128$ and $N = 64$ did not fit and the design for $N = 32$ did

fit on the ZYNQ board. For the 6VLX240TFF784 board the design for $N = 64$ also fit, but the design for $N = 128$ is still too large. Several reasons exist for the large area usage of the designs and in this section we will discuss those.

The first reason is already discussed in the previous section. In order to obtain useful precision with fixed-point calculations, a large word-length is required. The large word-length leads to a larger design.

A second reason is the extensive folding that is done is in the design. Because of the limited number of multipliers available, these have to be shared for different operations. Folding saves multipliers, but introduces extra multiplexers to make it possible to share the multipliers among different operations. Because only a limited number of multipliers is available, the degree of folding is determined by the number of multipliers. A large word-length further increases folding, because the hardware multipliers are only 25x18 bits wide. Any word-length larger than that uses more than one hardware multiplier for a multiplication. However, too much folding led to problems with multiplexer usage and made place-and-route impossible.

A possible countermeasure for the first reason has already been discussed. It involves using floating-point operations instead of fixed-point operations. This also partly tackles the second reason, as the word-length can be reduced with floating-point operations. However, the area usage that is saved by a smaller word length then has to be used for the floating-point functional units. These consume much more area than fixed-point functional units. The main question is whether the area savings of word-length are greater than the area increase of floating-point units.

Another possible way to optimize the design is to use the built-in block RAMs to store the precalculated coefficients. Now, these are stored in distributed RAM and it might save space when keeping these in block RAMs. Indexing the block RAMs can be done with the builtin indexer, or with another block RAM to store the indexes. This reduces multiplexer usage, at the cost of using block RAMs. This method looks very promising to reduce area usage.

Also, one option is to clock the functional units at a higher speed than the rest of the design. The hardware multipliers can be clocked at a high frequency, because the slowest path is caused by the multiplexers. If this is split in two: multiplexers and multipliers, then the multipliers can be clocked at twice the frequency of the multiplexers. This way, the number of required multipliers can be reduced further, thereby saving space. This would require changes to the design to have different clock domains: one for the multipliers and one for the multiplexers. This could be combined with pipelining of the multipliers to save more resources.

8.3 ZYNQ-7000

The synthesis results for the ZYNQ-board for $N = 32$ show 96% multiplier usage. Normally, four multipliers are needed for a 32-bits multiplication. This would result in $4 \cdot 2 \cdot 32 = 256$ multipliers. However, Precision RTL synthesis tooling was able to reduce the number of multipliers by applying some optimizations. This results in 212 multipliers instead of the expected 256, which means it fits on the board.

Logic utilization is approximately 29%, which is mostly due to multiplexing. A higher utilization would lead to a more difficult to route design, thus this utilization is fine. The slowest path is approximately 15 ns, thus the clock speed can be 67 MHz, which is more than enough for the design.

Note that this could not be reproduced with standard Xilinx ISE tooling, because ISE does not optimize the multiplier usage.

8.4 6VLX240TFF784

The synthesis results for the 6VLX240TFF784-board for $N = 64$ show 47% multiplier usage. As with the design for the ZYNQ-board, synthesis tooling was able to reduce the number of multipliers by applying some optimizations. However, even without optimizations the design would have fit on the board. Logic utilization is approximately 22%. The slowest path is approximately 31 ns, which means clock speed could run at 32 MHz.

8.5 Mathematical design methodology

The complete design has been created with a design methodology based on mathematics. Its aim was to start from the mathematical model and apply transformations on the equations in order to finally end up with a straightforward transformation to CλaSH code.

In this case study, we found that remaining close to mathematics as long as possible has indeed numerous advantages. After several mathematical transformations, the design could be translated to CλaSH code without errors. Different designs, for example for FPGAs with more multipliers available, were just a minor adjustement to the code, because it just meant one different mathematical transformation.

This process of generating the CλaSH code was so straightforward that a simple script had been written to do it given the number of multipliers and the number of elements. This allowed for simple and fast testing of different designs. The design approach combined with this script made a great difference compared to writing traditional VHDL. It would have taken a lot longer to write one of the possible implementation in VHDL, while it was now possible to generate any of the implementations with a simple script. This process is similar to processes that synthesis tools use already: for example, Xilinx Coregen can generate VHDL code directly from user input. However, these are general tools that can only generate parts of a design. For example, when a FIFO queue is needed it can generate the VHDL for a FIFO queue. However, the rest of the code needs to be written manually. With my script, the CλaSH code for this specific application can be generated.

There also exist several drawbacks of this method, which have mostly to do with the lack of libraries and debug tools that currently exist. For example, a fixed-point library for CλaSH would prove a great addition. Also, it is hard to have a good overview of all signals simultaneously during simulations and large simulations can take a long time to run. However, tools that help the developer in debugging and libraries can always be developed. These drawbacks do not relate to the design method itself. In recent months, a large improvement in the tooling has already been accomplished. It is expected that more features will be added to the tooling in the future.

9 Conclusion

Hardware design can be a difficult task. Often, several tradeoffs have to be made between area and time. In addition, cumbersome design processes with transformations between different semantics can introduce subtle bugs in the design. In this case study we investigated a different design methodology based on transformations in mathematical form. The topic of the case study was the cochlea model, for which the goal was to design a working implementation of the cochlea model on an FPGA.

During development of the implementation, the mathematical design methodology was found to be very useful. The transformations are described in mathematical form and are fairly trivial. This led to a reduced probability for errors in the design. In addition, because the last step of transforming the mathematics to CλaSH-code was straightforward, it is faster to develop variations of a certain algorithm: the only thing that needs to be changed is the mathematical form, the CλaSH-code follows. This is a large advantage, as in the previous approach the algorithm would be changed in a sequential language like C-code, and transforming that into an efficient hardware design leads to the aforementioned errors because of the semantic mismatch between the two languages. Thus, the two main advantages of CλaSH and a mathematical hardware design methodology over the traditional design using C and VHDL are trivial, verifiable transformations and rapid development.

A disadvantage, at least in the current version of CλaSH, is the fact that some VHDL features are not yet supported in the CλaSH compiler. An example of this are efficient fixed-point libraries. Also, while the type system of Haskell is quite flexible, there are still limitations in CλaSH, due to the fact that not everything can be easily translated into a hardware design. Large simulations of designs with a lot of calculations can become slow at times. However, these disadvantages are related to the tooling instead of the design methodology itself and can easily be improved in the future.

The implementation of the cochlea model proved more difficult than expected. Because it is a computationally intensive model with a wide range of input values, a large word-length and a lot of multipliers are needed. There is plenty of parallelism to exploit, but there were not enough resources on the FPGA to process everything in parallel. Folding was needed, which led to a large area usage - too large for the target FPGA. Because of this, the model was scaled down to a smaller number of cochlea elements until it fit on the desired FPGA.

Future work could investigate the use of block RAMs instead of multiplexers to select the input to the functional units. Also, pipelining the multipliers in addition to running the multipliers at a different clock frequency than the rest of the design might save more resources. These might save enough resource to obtain a working design with 128 elements.

In conclusion, we found that the mathematical design methodology has many advantages. Even for large projects like the cochlea model, it is faster and less error-prone to stay as long as possible in a mathematical form. This project has not been performed with the traditional VHDL approach and thus cannot be compared directly to it. However, we strongly think that a traditional design approach would have led to more errors in the design. The CλaSH compiler already has a lot of functionality, but more additions to the tooling like fixed-point libraries

would be welcome. Unfortunately, the original design that was created for $N = 128$ did not fit on the target FPGA as the model is too computationally intensive. The design was scaled down to a smaller number of elements ($N = 32$), which led to a working implementation.

Bibliography

[1] A. J. Virginia, Y. D. Yankova, and K. L. Bertels, "An empirical comparison of ansi-c to vhdl compilers: Spark, roccc and dwarv," in *Anual Workshop on Circuits Systems and Signal Processing ProRISC*, pp. 388–394, 2007.

[2] P. R. Panda, "Systemc-a modeling platform supporting multiple design abstractions," in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pp. 75–80, IEEE, 2001.

[3] G. J. Smit, J. Kuper, and C. P. Baaij, "A mathematical approach towards hardware design," 2010.

[4] C. Baaij, "Clash: From haskell to hardware," Master's thesis, University of Twente, 2009.

[5] M. van den Raadt, "Formulering 1-dimensionaal cochlea model in termen van een netwerk," 1 1991.

[6] D. A. Turner, *Recursion equations as a programming language*. Cambridge University Press, 1982.

[7] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, *et al.*, "Report on the programming language haskell: a non-strict, purely functional language version 1.2," *ACM SigPlan notices*, vol. 27, no. 5, pp. 1–164, 1992.

[8] L. Chittka and A. Brockmann, "Perception spacethe final frontier," *PLoS biology*, vol. 3, no. 4, p. e137, 2005.

[9] S. Van Netten and H. Duifhuis, "Modelling an active, nonlinear cochlea," in *Mechanics of Hearing*, pp. 143–151, Springer, 1983.

[10] O. Schuring, "Enhancement of a model of the human cochlea," Master's thesis, Rijksuniversiteit Groningen, 2011.

[11] P. Quesada-Barriuso, J. Lamas-Rodríguez, D. B. Heras, M. Bóo, and F. Argüello, "Selecting the best tridiagonal system solver projected on multi-core cpu and gpu platforms," in *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (as part of WorldComp2011 Conference)*, 2011.

[12] S. Levin, "Footnote to parallel xt migration." `http://sepwww.stanford.edu/data/media/public/oldreports/sep41/41_14.pdf`.

[13] Jesshope and R. Hockney, "Parallel computers: architecture, programming and algorithms/rw hockney, cr jesshope," 1981.