Norwegian University of
Science and Technology

High-level implementation of a partial DSP algorithm chain used in a digital radio using functional programming

Kristoffer Hove Torsvik

Master of Science in Electronics
Submission date: June 2016
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Anja Niedermeier, Nordic Semiconductor ASA

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Abstract

This thesis investigates the advantages of using functional programming as a hardware description tool. The functional programming paradigm shares a lot of similarities with hardware, as it is highly parallel in nature and has a notion of structure in its descriptions. Furthermore, it operates on a higher-level of abstraction compared to commonly-used hardware description languages, which facilitates the design- and debugging process.

The goal has been to implement a decoding algorithm - the Viterbi algorithm (VA) - used in a variety of communication networks. An estimate done in 2005 revealed that approximately 10^{15} bits are being decoded by the Viterbi algorithm in digital TVs, every second of the day. This estimate neglects a billion of cellphones that are also a large user of the VA.

Throughout the design process, the functional implementation of a Viterbi decoder has been simulated directly within the interactive environment of the source language, without the need to create a customized testbench. A modularized design can simulate components in isolation to give written functions evidence of correctness.

The complete implementation has been transformed to a synthesizable SystemVerilog description, using an open-source tool called CλaSH. The resulting code has been verified with several testbenches that were written for a reference design, verifying correct timing- and decoding behavior. Furthermore, it has been compared with a reference design made in SystemVerilog, in terms of performance and resource utilization.

Functional programming has proven to be well-suited for hardware designs. However, it is still hard to achieve the same quality as traditional methodologies, and requires the designer to be skilled in the functional programming style. Nevertheless, it is a superb alternative for performing architectural exploration, and creating high-quality circuits.

Sammendrag

Denne oppgaven undersøker fordelene ved å bruke funksjonell programmering som et verktøy for å beskrive maskinvare. Den funksjonelle programmeringsparadigme deler mange likheter med maskinvare, ettersom det har en svært parallell natur og gir en form for struktur i dens beskrivelser. I tillegg, opererer det på et høyere abstraksjonsnivå i forhold til vanlig brukte maskinvare beskrivende språk, som forenkler design- og feilsøkings prosessen.

Målet har vært å implementere en dekodingsalgoritme - Viterbi-algoritmen - som brukes i en rekke forskjellige kommunikasjonsnettverk. Et anslag som ble gjort i 2005 viste at om lag 10^{15} bits blir dekodet av Viterbi-algoritmen (VA) i digitale TV apparater hvert sekund. Dette anslaget neglisjerer en milliard mobiltelefoner som også er en stor bruker av VA.

Gjennom hele designprosessen, har den funksjonelle implementasjonen av en Viterbi dekoder blitt simulert direkte i det interaktive miljøet til kildekoden, uten behov for å lage en tilpasset testbenk. Et modulærisert design kan simulere komponenter i isolasjon, for å bevise funksjonene som korrekt.

Den komplette implementasjonen har blitt transformert til en syntetiserbar SystemVerilog beskrivelse, ved hjelp av et verktøy kalt CλaSH, som er åpen kildekode. Den resulterende koden er blitt verifisert av flere testbenker, som har verifisert at tidsstyring og dekodingsoppførselen er korrekt. I tillegg har designet blitt sammenlignet med en tilsvarende løsning som er laget i SystemVerilog, med fokus på ytelse og ressursutnyttelse.

Funksjonell programmering har vist seg å være velegnet for å designe maskinvare. Det er likevel vanskelig å oppnå den samme kvaliteten som tradisjonelle metoder, og det krever i tillegg at designeren er dyktig i den funksjonelle programmeringsstilen. Det er imidlertid et ypperlig alternativ for å utforske ulike arkitekturer, og generere kretser av høy kvalitet.

Preface

This master's thesis is part of the Master of Science degree program in Electronics, Design of Digital Systems at the Department of Electronics and Telecommunication (IET) at the Norwegian University of Science and Technology (NTNU).

The project was proposed by Nordic Semiconductor ASA - a world-leading chip designing company based in Norway, specializing in bluetooth low energy. The goal was to further explore the possibilities of functional programming for use in hardware design. This project has been a completely new field for me to take on. It has been educational, exciting and - in a good way - challenging. Being able to tackle the conundrums within the project was not a stroke of luck.

Throughout the semester, Nordic Semiconductor provided me with a computer with all the required tools and resources for me to complete the project. This declaration of trust was a big motivation for me and forwarded both the speed and completion of the project. My supervisor Dr. Anja Niedermeier from Nordic Semiconductor and Prof. dr. Kjetil Svarstad from NTNU has given me invaluable help and avid guidance throughout the semester. Thank you.

Contents

1 Introduction and motivation **1**
1.1 Project description . . . 2
1.2 Structure of the thesis . . . 2
1.3 Contributions . . . 3

2 Background and Theory **5**
2.1 Background Information . . . 5
2.2 High-Level Synthesis . . . 6
2.3 Functional programming . . . 8
2.4 Functional HDLs . . . 9
2.5 CλaSH . . . 10
2.6 Forward error-correcting codes . . . 11
2.7 The Viterbi Algorithm . . . 11
2.7.1 Convolutional Encoder . . . 12
2.7.2 Viterbi Decoder . . . 13
2.7.3 Hardware Implementation Methods . . . 18
2.8 Related work . . . 20
2.8.1 Viterbi decoder methods . . . 20
2.8.2 Publications about CλaSH . . . 21

3 Design Methods and Tools **23**
3.1 Introduction to Haskell . . . 23
3.1.1 Syntax . . . 23
3.1.2 Data structures . . . 25
3.1.3 Choices . . . 27
3.1.4 Recursion . . . 28

3.1.5 Higher order functions 29
3.1.6 Lambda expressions 33
3.2 CλaSH 33
3.2.1 From Haskell to CλaSH 33
3.2.2 Higher-order functions and recursion 35
3.2.3 Sequential circuits 36
3.2.4 RTL generation 39

4 Methodology **41**
4.1 Overview 41
4.2 Implementation details 43
4.2.1 Decoder 45
4.2.2 Valid control 49
4.2.3 Flush state register 49
4.3 Simulation 50
4.4 Verification 52
4.5 IP integration 54

5 Evaluation **55**
5.1 Hardware design with functional programming 55
5.1.1 CλaSH 56
5.2 Quality of Result 57
5.2.1 Exhaustive testing 58
5.2.2 Bit error rate 58
5.2.3 Synthesis results 58
5.2.4 Generated SystemVerilog code 59

6 Conclusion **61**
6.1 Future work 62

Appendices **63**

A CλaSH implementation **65**

B CλaSH testbench **71**

C Encoder Perl script **73**

Abbreviations and acronyms

ACS Add-Compare-Select

ACSU Add-Compare-Select Unit

ASIC Application-Specific Integrated Circuit

BER Bit Error Rate

BM Branch Metric

BMU Branch Metric Unit

DAG Directed Acyclic Graph

EDSL Embedded Domain Specific Language

FEC Forward Error Correction

FPGA Field-Programmable Gate Array

FSM Finite State Machine

GHC Glasgow Haskell Compiler

HDL Hardware Description Language

HLS High-Level Synthesis

PM Path Metric

RE Register Exchange

RTL Register Transfer Level

SMU Survivor Memory Unit

SoC System-On-Chip

TB Trace-Back

VA Viterbi Algorithm

VLSI Very-Large-Scale Integration

Chapter 1

Introduction and motivation

The development in electronics play a key-role in the modern age, and is necessary to solve important societal challenges. The complexity of integrated circuits is ever growing, and the industry is racing to develop the best solutions and seize their window of opportunity. Because of this, there has been a continuous demand to save time along the "critical path" in the development process. A field of interest is the design- and debugging phase, where hardware tools of today date back to the late 1980s. Even though these are highly customized for hardware, their abstraction is close to the details of the modeled circuit, and requires a testbench for simulating their behavior.

For decades, there have been attempts to raise the abstraction level of hardware description languages, with a process known as high-level synthesis (HLS) [12]. It utilizes features of a high-level language for describing hardware, and can facilitate the debugging and verification process through better simulation possibilities. This can be especially useful for digital signal processing, like describing an algorithm used in a digital radio. Additionally, HLS allows designers to perform architectural exploration and compare post-synthesis results and performance between different implementation methods.

One well-established programming paradigm that has gained increasing interest from hardware designers, is functional programming. This is due to their

similarities with hardware, and highly parallel nature. There are several features that makes functional programming stand out from other programming paradigms in terms of hardware design. Firstly, their written programs are an expression of data dependencies rather than a sequence for the compiler to execute, which gives the program a notion of structure. Secondly, functional languages are strongly typed, which means that any violation of an argument's type or size will be caught at compile time. Both of these features are highly related to circuits, as these have a static structure with fixed-sized bit-width. What makes it even better is that functional languages are immutable, which means that objects are not allowed to change once they are created, and the transformation to hardware can be formally proven.

This thesis explores the functional hardware description language (HDL) CλaSH. It uses the functional language Haskell as source language, and allows a functional description to be transformed to a synthesizable hardware representation. For testing the usability and reliability of CλaSH, a Viterbi decoder is implemented and compared with a reference design. The Viterbi decoder is widely used in communication systems to prevent transmitted messages from being lost over noisy communication channels. In order detect errors in the received message sequence, redundant information is added with a forward error-correcting code known as convolutional code.

1.1 Project description

This thesis explores the Viterbi decoder in detail, and looks at related hardware implementation techniques. The decoder will be implemented in the functional programming language Haskell, and transformed to a synthesizable SystemVerilog description with CλaSH. The design methodology will be evaluated in terms of usability and reliability, based on the quality of the generated hardware representation and the freedom of expression that CλaSH allows.

1.2 Structure of the thesis

Chapter 2 provides the background information of high-level synthesis, and explains the features of functional programming and what makes it suitable for describing hardware. CλaSH and other functional HDLs are addressed,

describing their main differences and what makes CλaSH unique. Furthermore, the details of the Viterbi algorithm are described, with illustrative examples showing its main phases. Finally, some implementation specific methods are described and other techniques optimized for hardware are mentioned.

In chapter 3, an introduction to Haskell and CλaSH is given. The introduction covers the required information of the functional programming style, to understand the details of the presented implementation. Following, it describes the steps needed to customize the implementation to be compatible with the CλaSH compiler.

Chapter 4 gives an overview of the overall system, and presents the implementation details of the Viterbi decoder. Some examples are included to highlight the internal behavior of the functions, and keep the chapter free from in-depth explanations. Furthermore, the simulation procedure used throughout the design process is presented as examples, covering simulations of single components and the overall system. Moreover, the verification phase is described as several steps. First, the setup for generating a simple testbench with CλaSH is shown, then some of the more exhaustive verification steps - using provided testbenches - are described.

In chapter 5, the implementation is evaluated. A general note on functional programming as a hardware description tool is given, based on experiences throughout the project. Additionally, CλaSH is evaluated, where its usability and issues are addressed. Post synthesis results are presented in terms of performance and resource utilization.

1.3 Contributions

The functional hardware description is transformed into synthesizable SystemVerilog code, using the Haskell-based functional HDL CλaSH. The generated solution is verified with testbenches made for a reference design provided by Nordic Semiconductor, to give evidence of correctness. Furthermore, the resulting implementation is synthesized to gate-level hardware, and compared with a reference design in terms of performance and utilized resources. As a final step, the implementation is integrated into an existing digital radio, part of a complete system-on-chip (SoC). The chip is tested as a whole with several top-level testcases, and the digital radio is synthesized for an FPGA.

Chapter 2

Background and Theory

This chapter gives an overview of concepts and tools used in the hardware implementation presented in this paper. It discusses the fundamental features of functional programming, and its suitability as a hardware description tool. Furthermore, the chapter addresses error-correcting codes and decoding techniques based on the Viterbi algorithm.

2.1 Background Information

Traditionally, the first step involved in hardware design is often to model its functionality in a generally applicable software language such as MATLAB, C or SystemC. The model serves as the performance reference for the hardware implementation, where it is common to explore different design methods prior to the actual design process. Through simulations and comparisons, decisions can be made early and the high-level model can provide stimuli and response files for hardware testbenches. Additionally, a software model can later be used for equivalence checking[1]. The actual hardware implementation is commonly made from a hardware description language (HDL) such as Verilog or VHDL, which is syntactically and semantically very different from a software language.

[1]Equivalence checking is used to compare two representations of a circuit description, to prove that they exhibit the same behavior.

Their abstraction level is considerably closer to hardware, as they model logical operations on digital signals and their flow between hardware registers. This low level of abstraction is commonly referred to as the register transfer level (RTL), and has proven to be too verbose and concrete for some application domains. Considering the high increase of chip functionality and complexity, especially accounting for very-large-scale integrations (VLSI), the need to express hardware at a higher-level of abstraction become increasingly important. This have resulted in several new approaches to arise. Some include features from object oriented programming to already existing HDLs, resulting in languages such as SystemVerilog, while other approaches aim at translating high-level languages to gate-level hardware - a process known as *high-level synthesis*. This process often goes through several transformation steps before generating the actual gate-level hardware. It is common to translate a high-level description to a commonly-used HDL representation, in order to use available tooling for the remaining synthesis-steps. Although HDLs are still widely used and preferred in the industry for its quality assurance, it can still be rewarding to derive separate HDL modules from a high-level language, to work in conjunction with existing modules. Nevertheless, a synthesizable high-level software model can be useful when investigating the fidelity of area, power or resources, during the architectural exploration.

This thesis explores an alternative way of implementing the software model with a programming paradigm known as *functional programming*. With a supporting tool known as *CλaSH*, the functional description can be transformed into a synthesizable VHDL-, Verilog- or SystemVerilog representation.

2.2 High-Level Synthesis

High-level synthesis (HLS) is the process of translating a high-level language to gate-level hardware [12]. A high-level language refers to programming languages with strong abstraction from the details of the computer. HLS tools are able generate a fully timed implementation, even from an untimed or partially time high-level specification. In terms of hardware design, we differentiate between several abstraction levels, namely circuit-, logic-, register-transfer-, algorithmic- and system level. These levels are described by three distinct design domains, and is best visualized by an Y-chart [17], depicted in Figure 2.1.

The design methodology explored in this thesis are mostly targeting the struc-

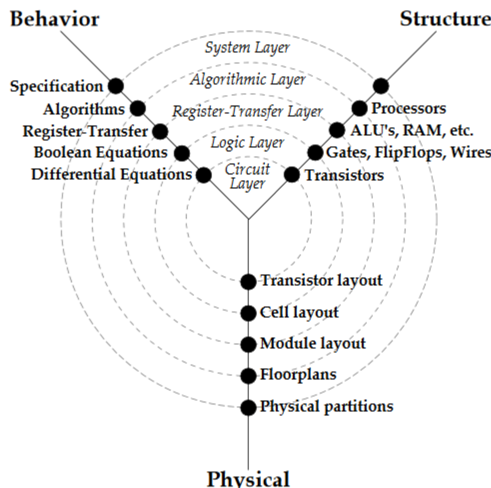

Figure 2.1: Gajski Y-Diagram: Figure copied from [5]

tural domain, with some aspects from the behavioral domain. Expressing hardware at a higher abstraction level serves many purposes for a hardware designer. It can for instance be time-saving in terms of both coding and debugging. Additionally, it allows the designer to explore and compare different design methods and architectures ahead of the actual implementation.

Higher-level synthesis started out as a research for academic purposes and has been a topic since the beginning of the 1970s. Although, commercially it did not become available until the beginning of the 1990s. This escalation started when the industry gained interest. Even then, Higher-level synthesis was tiresome to use and produced poor results, thus further pending its success. Today, previous problems associated with HLS tools have been addressed - making it increasingly important for large scale designs. Although, issues concerning the efficiency of this method is still salient in the field. [37]

Some of the more currently available HLS tools and languages are described in [28].

2.3 Functional programming

Functional programming is a programming paradigm that operates entirely on functions. It is essentially programming with mathematical functions, where functions are not allowed to keep state or change their objects once they have been created. This is a feature referred to as *immutability*, and makes functions completely safe from outside manipulation and free from *side effects*[2]. Because of this, it is guaranteed that functions will return the same value every time it is given the same set of inputs - a property referred to as *referential transparency*. This makes it easier to run parallel computations, and write programs that exploits multi-core CPUs, which alone is reason enough for functional programming being on the rise. Given that there are no data dependencies between functions, their order can be reversed or executed in parallel.

For the mainstream programmer, used to *imperative* languages (e.g. C, Java, Python), the though-process of a functional language may require some adaptation. A functional language treats functions as *first-class citizens*, allowing functions to be passed as arguments and be used as return types. Functions exploiting this, are known as *higher order functions*. They are not just a powerful way of solving problems, but may be considered the main building-block of a functional program.

It is common to distinguish between two types of functional languages, known as *pure* and *impure*. Impure languages, only adopt some functional features from the completely pure languages. This thesis uses the pure functional language, *Haskell*, which was developed to be *the* standard for functional languages. Common for all pure functional languages, is that they utilize a compiler technique called *lazy evaluation*, which avoids executing functions until it is specifically told to. This makes programs an expression of *what things are*, rather than what it should do. Furthermore, some pure languages - including Haskell - are *statically typed*, meaning that the compiler will automatically figure out the type signature of functions, so that it becomes unnecessary to label every part of the code with its own type. Moreover, all pure functional languages are strongly typed - making them more likely to report errors at compile time if there are any incongruity between the actual type and the expected type. Aesthetically, functional programs can appear quite elegant and concise, as they typically compresses loops and code blocks to a single line of code.

[2] When a function or expression modifies some state, it is said to have a side effect. This includes modifications of global variables and any interactions with the outside world.

It is fair to note that every feature of a purely functional language, is also part of the reason why it has not become as mainstream as imperative languages. Sometimes it is desirable to utilize side-effects and work with mutable data structures at the cost of keeping the data consistent. Moreover, there are far less programmers skilled in the functional programming style, and a large amount of maintainable systems built from imperative languages. Nevertheless, functional programming has a promising similarity with hardware, and might be a perfect match.

2.4 Functional HDLs

Functional HDLs are the common designation of languages, which uses functional programming as the basis for describing circuits. In the context of circuit design, they are the most studied, non-imperative paradigm, which uses available abstractions from functional languages to simplify the expression of hardware. Typically, a functional HDL will transform a functional hardware description to a commonly-known HDL, such as VHDL, Verilog or SystemVerilog. This way, they can easily be included as a module in a larger design and utilize existing tooling for synthesis and verification. Functional HDLs are able to simulate hardware components without the need to create a separate testbench. Because of this, the simulation time is much faster and it also allows single components to be tested separately.

Functions modeling a sequential circuit will typically execute on every tick of the system clock, or - if the language supports multiple-clock domains - it can be assigned to a specific clock source. Since functions are state-less, functions must connect their outputs with their inputs in order to model state. This creates a feedback loop that allows a function's calculated state to be given as input in the following clock cycle.

There have been many functional HDLs over the years, usually made obsolete by their successor. Some of the most known functional HDLs of today are *ForSyDe* [34], *Bluespec System Verilog* [31], *CλaSH* and the Lava family, including *Chalmers-Lava* [11], *Xilinx-Lava* [8] and *Kansas-Lava* [19]. ForSyDe and the Lava family, is implemented as an *embedded domain specific language*[3] (EDSL) on top of Haskell, which encodes the hardware in custom data-structures. This

[3] A language that is defined as a library, on top of a host language, which adds domain-specific primitives.

allows languages to have its own syntax and semantics, but requires the designer to learn it. Bluespec and CλaSH on the other hand, was both initially developed as a subset of Haskell, where the syntax and semantics were already defined. However, Bluespec later decided to add a layer of syntax on top, to make it look more like a HDL, and limited the use of abstraction that Haskell offered. As CλaSH have stayed true to the idea of leveraging an existing functional language, it has gained an increasing interest. Another interesting HDL which is worth mentioning is *Chisel*. It is embedded in the functional language *Scala*, and provides concepts from object orientated programming as well as functional programming [6]. The remainder of this thesis, will however only be focusing on CλaSH and its potential as a functional HDL.

2.5 CλaSH

CλaSH was developed as a research platform for hardware design methodologies at the University of Twente [5, 25, 4]. It is a functional HDL, which borrows already defined syntax and semantics from the functional language Haskell. This makes every CλaSH design a valid Haskell program, which facilitates the simulation process and provides an extensive support for commonly used functions [3]. Simulations in CλaSH are essentially done by executing a function representing the hardware, which allows modularized code to test sub-components in isolation. As CλaSH leverage an existing functional language, it adopts all of its syntax and semantics. However, this means that CλaSH also uses the "standard" compiler approach towards synthesis, which limits its support for recursive functions - frequently used in the functional programming style **??**. Luckily, this does not greatly limit the designer's freedom of expression, since Haskell's higher-order functions have been redefined in CλaSH to avoid non-synthesizable recursion techniques, and - in most cases - eliminates the need to define our own recursive functions. Although there are elements in Haskell that have no meaning in hardware, most of its commonly-used features are supported, and CλaSH extends its library with functions and types well-suited for hardware design. This is further described in Chapter 3, where a short introduction is given.

In addition, CλaSH supports testbench generation, multiple-clock domains, user-defined VHDL/(System)Verilog primitives, and top-level annotations, to exert some control of how the top level function is created.

2.6 Forward error-correcting codes

Some background information is required to understand the presented implementation of a Viterbi decoder. From this point forward, decoding concepts and hardware specific methods are discussed. First is a short introduction to codes allowing a decoder to detect errors in a transmitted message sequence.

Forward error-correcting (FEC) codes are used to prevent messages from being lost when sending messages over an unreliable or noisy communication channel. This is done by adding redundant information before each message is sent, and allows the receiving side to reconstruct the original message despite errors. It is a technique that is heavily applied in one-way communication links and other situations where retransmissions can be either impossible or costly. There are mainly two categories of FEC codes; *block codes* and *convolutional codes*. Block codes work on fixed-size packets of bits or symbols, while convolutional codes operates on bit streams of arbitrary length. This thesis is focusing on convolutional codes, which are decoded by the *Viterbi algorithm*.

2.7 The Viterbi Algorithm

The Viterbi algorithm was first introduced in 1967 by Andrew J. Viterbi [36]. It is an algorithm for message decoding over noisy communication links, and is a special case of the well-known *Bellman-Ford shortest distance* algorithm[4]. Common for both algorithms is that they use a method called *dynamic programming* - a method for solving a complex problem by breaking it into simpler overlapping problems. The basic algorithm was later redefined to make it a practical decoding technique [21], and has become the most popular decoding procedure for convolutional codes, due to its high accuracy. However, it is considered the most resource-consuming compared to alternative decoding algorithms, such as the *Fano algorithm* and the *Stack algorithm*. There has however been proposed several different implementation techniques for the viterbi algorithm, for reducing power consumption, reducing memory requirements and increasing the throughput. A better overview of existing methods is given in Section 2.8. In addition to radio communication, the algorithm has been applied in applications concerning speech recognition, modems and modern disk drives [16].

[4]The Bellman-Ford algorithm, is used to compute the shortest distance in a weighted graph, from a single source node to all other nodes.

The remainder of this chapter will describe the Viterbi algorithm in more detail, while taking on a running example. For an in-depth explanation of the Viterbi algorithm, see [35]. In order to understand how the Viterbi decoder operates, it is necessary to be familiar with the convolutional encoder - producer of the convolutional code. The overall communication system is best illustrated by a block diagram, shown in Figure 2.2.

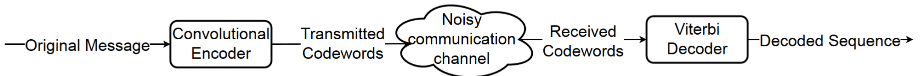

Figure 2.2: Overall system

2.7.1 Convolutional Encoder

The encoder operates on a stream of input bits, representing the original message, and transforms k bits to a series of n output bits. The relationship between the number of inputs and outputs are referred to as the bit rate $r = k/n$, where the number of output bits n always exceed the number of input bits k. A rate of $r = 1/2$ is the simplest form an encoder can operate with, and is used throughout this thesis, both in the examples and in the actual implementation. Such an encoder is best visualized by a FSM, controlling the generation of outputs based on its current state and input bit. Figure 2.3 illustrates this behavior, with a total number of eight possible states.

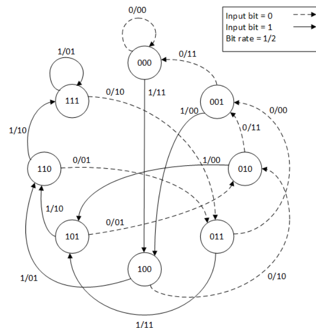

Figure 2.3: Encoder FSM

The number of states is determined by the number of memory elements used in the encoder, known as the *constraint length* K. The relationship between K and the number of states is: $nr_of_states = 2^{K-1}$, hence $K = 4$ results in eight states. The actual encoding is performed with modulo-2 adders, and a multiplexer, as Figure 2.4 illustrates.

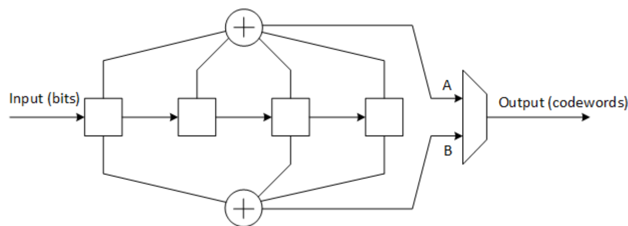

Figure 2.4: Convolution encoder diagram

Note that connections between the memory elements and the modulo-2 adders determines the polynomials, which must be known by the decoder. For this convolution encoder, the polynomials will be 15 and 11 for A and B, respectively.

Given the aforementioned encoder, an input message of 0100110100 will result in the encoded sequence 00 11 10 11 00 01 01 11 01 11, where each bit-pair is referred to as a *codeword*. However, the sequence received by the decoder may deviate from what was originally transmitted, due to noise and instability on the communication link. The task of the Viterbi decoder is to figure out the state transitions that was taken by the encoder, even though there are faulty bits in the codewords.

2.7.2 Viterbi Decoder

The task of the Viterbi decoder is to reproduce the original message processed by the encoder. It does this by keeping track of the encoder's state transitions for each received codeword, and reproduce the encoded message based on this. Since a received codeword may deviate from a transmitted codeword, the decoder needs to evaluate a number of successive codewords in order to determine the most likely state transition. For instance, receiving the first codeword from its initial state 000, there are two possible codewords that the encoder can produce (i.e. 00 or 11. See Figure 2.3). If however the *received* codeword is neither of

these, the decoder knows that one of the bits are faulty. In order to determine if the codeword was originally 00 or 11, the decoder looks at the successive codewords - resulting in several possible paths in the encoder's FSM. Each path will be associated with an error metric, which helps the decoder chose the *most likely path* to reproduce the original message from. This is best visualized by a *Viterbi trellis*, which is essentially the encoder's FSM represented as a directed acyclic graph (DAG). A trellis based on the aforementioned encoder is shown in Figure 2.5.

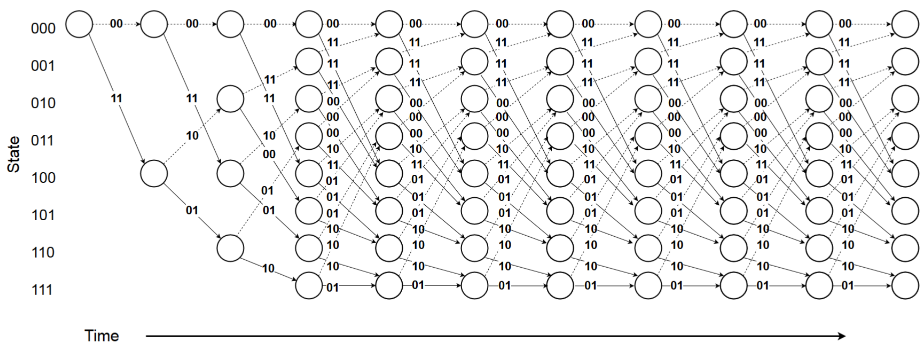

Figure 2.5: Viterbi trellis based on the encoder from the previous section

Each state in the trellis is placed vertically and each branch represents a valid transition to a new state. The adjacent number to each branch is the encoded codeword, which the received codeword is compared with. Each path leading from the initial state to a final state will correspond to a message, though only one path will produce the original message. The length of the trellis is determined by the implementation. As a rule-of-thumb, it is typically $5*K$ and is referred to as the *traceback depth*. Hence, the illustrated trellis should ideally be twice its size, and will only serve as a model for demonstration purposes.

The steps included in the decoding process, can be divided into three main phases. It is appropriate to look at these as separate units: *Branch Metric Unit*, *Add-Compare-Select Unit*, and *Traceback Unit*. A typically hardware implementation will also require a FILO, as the block diagram in Figure 2.6 illustrates.

Each unit will execute in order whenever a new codeword is received by the decoder. Their task is further described in the following subsections:

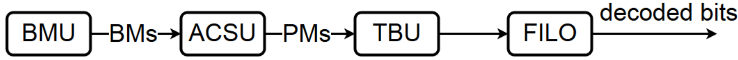

Figure 2.6: HW implementation

Branch Metric Unit (BMU)

The *branch metric* (BM) is a weight assigned to each branch in the trellis. It is a measure of the likelihood of a state transition based on the received codeword. A BMU will calculate these weights as the hamming distance between each state transition and the received codeword. For a bit rate of $r = 1/2$, the weight will be a value between $0 - 2$. This approach assumes that the received codewords have already been converted to digital values, and is commonly known as *hard decision decoding*.

Figure 2.7 illustrates the calculation of branch metrics for an arbitrary time-step in the decoding process.

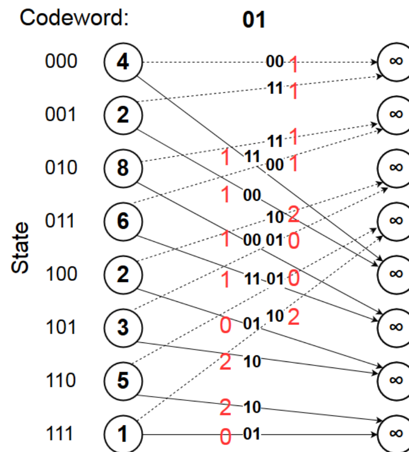

Figure 2.7: Branch metric calculations

There is another approach to this, known as *soft decision decoding*, where the received codewords have not yet been converted to digital values. It is more accurate method which is further described in Section 2.7.3.

Add-Compare-Select Unit (ACSU)

The Add-Compare-Select (ACS) Unit will keep track of the accumulated BMs for each path in the trellis. These metrics are referred to as *path metrics* (PMs), and reflects the total number of bit errors detected along a path. Based on the calculated BMs and PMs of a stage, the ACSU will decide which of the incoming branches to a state that should **survive**, and form a path back to the initial state. The calculation of the next PM for each state, is as follows:

$$
\begin{aligned}
PM_0' &= \texttt{Min}\{PM_0 + BM_{0\rightarrow 0},\ PM_1 + BM_{1\rightarrow 0}\} \\
PM_1' &= \texttt{Min}\{PM_2 + BM_{2\rightarrow 1},\ PM_3 + BM_{3\rightarrow 1}\} \\
PM_2' &= \texttt{Min}\{PM_4 + BM_{4\rightarrow 2},\ PM_5 + BM_{5\rightarrow 2}\} \\
PM_3' &= \texttt{Min}\{PM_6 + BM_{6\rightarrow 3},\ PM_7 + BM_{7\rightarrow 3}\} \\
PM_4' &= \texttt{Min}\{PM_0 + BM_{0\rightarrow 4},\ PM_1 + BM_{1\rightarrow 4}\} \\
PM_5' &= \texttt{Min}\{PM_2 + BM_{2\rightarrow 5},\ PM_3 + BM_{3\rightarrow 5}\} \\
PM_6' &= \texttt{Min}\{PM_4 + BM_{4\rightarrow 6},\ PM_5 + BM_{5\rightarrow 6}\} \\
PM_7' &= \texttt{Min}\{PM_6 + BM_{6\rightarrow 7},\ PM_7 + BM_{7\rightarrow 7}\}
\end{aligned}
$$

Figure 2.8 illustrates an example for an arbitrary time step, where the received codeword is 01. Note that each node contains the path metric value for its respective time-step.

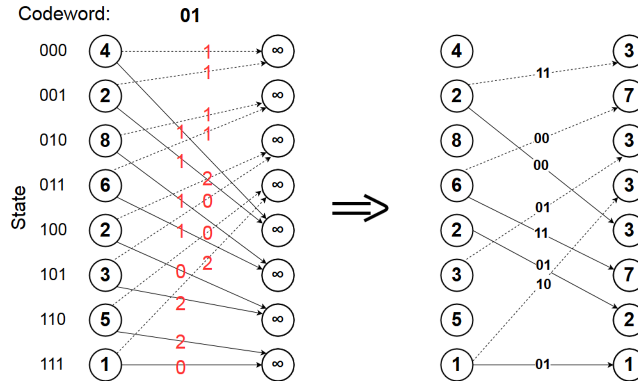

Figure 2.8: Path metric calculations

Traceback Unit (TBU)

The most likely path is first chosen at the end of the trellis. If the received codewords is equal to the codewords produced by the encoder, there will be one state at the final time-step with **zero** PM. From this state, there will be a single path leading back to the initial state, from which the decoder will start back-tracing from. The state transitions in this path will correspond to the original message processed by the encoder. While back-tracing through the trellis, the decoder will output a decoded bit based on each state transition. Since the decoded bit sequence is read in reverse order, it is required to have a FILO to reverse the order. This method is commonly known as the traceback (TB) method, and is a well known method which have been optimized for hardware in several papers (see Section 2.8). Figure 2.9 depicts a Viterbi trellis where all PM's have been calculated for each time-step, and the received codewords equals the transmitted ones.

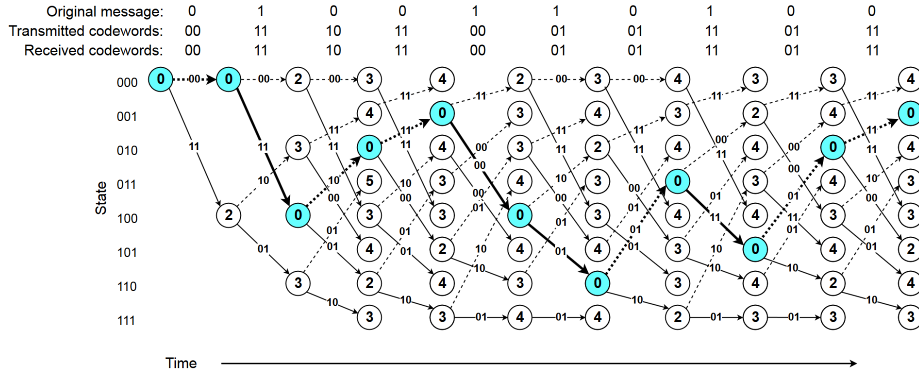

Figure 2.9: Trellis example: Received codewords = Transmitted codewords

If we introduce two incorrect bits to the received codewords, it will still result in the same path as depicted above. This is because the final state 001 will still have the least PM compared to the others, and the correct message is recreated. Figure 2.10 illustrates this case.

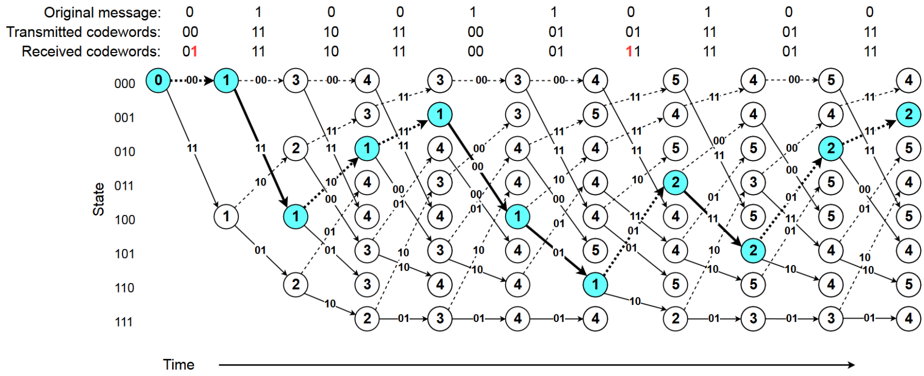

Figure 2.10: Trellis example: Received codewords ≠ Transmitted codewords

2.7.3 Hardware Implementation Methods

This section describes some implementation methods utilized in the actual design. More implementation specific details is described in Chapter 4.

Register Exchange

Register Exchange (RE), is an alternative implementation method for the trace-back unit, which is conceptually simpler and faster than the aforementioned TB method. The method assigns a register to each state and uses them to record the decoded output for each time-step in the trellis. This requires every bit in each register to be read and rewritten whenever a new codeword is received by the decoder, which is the main disadvantage of the original RE method. The decoded bits are appended to their register - shifting them to the left. At the end of the trellis, when each register will be filled, the leftmost bit will be the first decoded bit for their respective path. Figure 2.11 illustrates the previously shown trellis example, implemented as a RE architecture.

Notice that the registers in the last time-step contains the decoded sequence for its associated path. The read-out will happen during any successive codewords from when the registers became full, by reading the leftmost bit of the state register with the least PM. This read-out process can be seen in Figure 2.12 for the first five bits in the original message. Note that the leftmost container

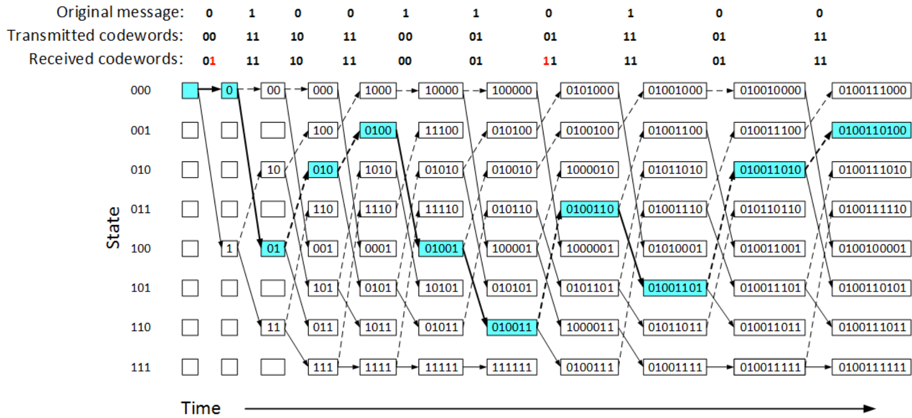

Figure 2.11: Register exchange

of each step, is the PM for each state, which decides which state register to read.

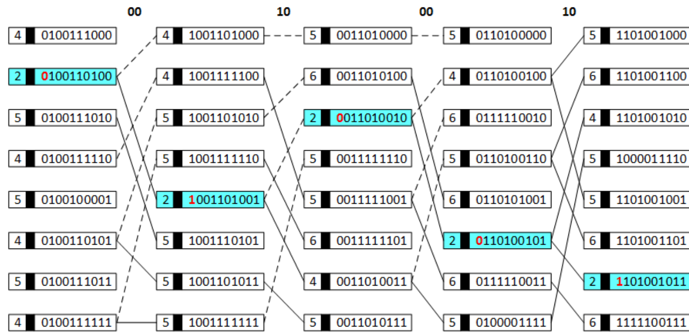

Figure 2.12: Decoded bit read-out.

Since the RE method does not require back-tracing, it eliminates the need for additional RAM blocks for traceback, and increases the throughput. In addition, it eliminates the need for a FILO since the output order is correct. It is common to name the traceback-unit as the survivor memory unit (SMU) when using the register exchange architecture.

Soft Decision Decoding

Soft decision decoding is a decoding process where the received codewords are represented directly as the voltage samples, *before* digitizing them. Because hard decision decoding throws away information in the digitizing process, soft decision decoding will be more accurate and reduce the overall probability of bit errors. Each codeword will contain information about *reliability* of each received symbol. For instance, in a 4-bit encoding, this *reliability* information can be encoded as Figure 2.13 illustrates.

Figure 2.13: Softbit reliability information

2.8 Related work

This section highlights work related to CλaSH and decoding methods using the Viterbi algorithm. One publication in particular is especially related, as it investigates a Viterbi decoder implemented in a different functional HDL, Kansas Lava [9]. However, the architecture is completely different, and not related to the implementation presented in this paper.

2.8.1 Viterbi decoder methods

Many implementation variations of the Viterbi decoder are possible. They are trading off implementation complexity, decoding latency, area and power consumption.

Traceback optimization

As the conventional traceback approach is known to use more area than the RE method, an optimized traceback architecture is proposed in [18]. It is known as the pre-traceback architecture which in addition to saving area, reduces the number of read operations and achieves a more energy efficient solution. [10] presents another solution which aims at reducing the decoding latency of the traceback method. Additionally, another power efficient architecture is proposed.

ACS Lookahead

The add-compare-select (ACS) lookahead method, is an implementation for high-speed applications [27, 24, 15, 14]. It allows the ACS unit to compute several number of time-steps in parallel. It results in much greater area for the design, but the method can be profitable for high-speed applications.

Register Exchange optimization

[13] presents a new implementation technique for the register exchange (RE) method. As previously mentioned, the disadvantage of the RE method is that every bit needs to be read and rewritten when a new codeword is received by the decoder. The modified method uses a pointer concept instead, so that it is not necessary to copy the contents of one register to another.

Soft-output Viterbi algorithm

The soft-output Viterbi algorithm (SOVA) is producing a soft output instead of a decoded output bit. There are several methods for implementing this architecture [20, 7, 23, 38].

2.8.2 Publications about CλaSH

CλaSH can be used for a number of things, from quick algorithm prototyping to complete architectures. There have been several publications about

CλaSH, featured at the homepage of clash [2]. Here, some of them are briefly described.

[30] shows how a simple dataflow processor can be implemented by using CλaSH. The design was transformed to VHDL code and compared with a reference design to evaluate the outcome of the CλaSH implementation. It was shown that CλaSH generated a VHDL code that resembled the functionality described by Haskell, and that it was synthesizable for both ASIC and FPGA.

[29] presents a highly configurable *coarse-grained reconfigurable array* (CGRA), implemented in CλaSH.

CλaSH has also been used for implementing an algorithm used for frequency estimation, known as the MUSIC algorithm [22]. The thesis investigates how CλaSH compares to VHDL, and proves the usability of CλaSH descriptions for a non-trivial hardware implementation.

In [33], a proposed solution for handling recursive functions is presented. It describes a method for transforming a recursive function description to a corresponding circuitry. It has proven that data-dependent recursive functions can be transformed with formal rewrite rules, though there are aspects of the methodology that still need to be researched further.

Chapter 3

Design Methods and Tools

This chapter aims at giving the reader a basic understanding of Haskell and provide the necessary knowledge needed to understand the implemented solution made in CλaSH. First is a short introduction to Haskell, displaying some example code as new concepts are introduced. The second part of this chapter, specifically targets CλaSH, and explains some important things to consider when implementing hardware.

3.1 Introduction to Haskell

This section aims to give a quick introduction to the syntax of Haskell and show some of its most useful features, utilized by functional HDLs. A complete tutorial can be found in [26], [32].

3.1.1 Syntax

When writing a function, one must first specify the function name and then list all its parameters. These are separated by spaces without any surrounding

brackets. The return value is determined by the right hand side of the equality sign. For instance, a function *multiplyTogether*, that multiplies three values together could be defined as shown in Listing 3.1. Here, the function has three parameters, named x, y and z.

```
multiplyTogether x y z = x * y * z
```

Listing 3.1: Function with three parameters

Within the interactive environment of GHC (GHCi), these arguments can be applied a specific value to study the outcome:

```
ghci> multiplyTogether 2 5 10
100
```

GHCi also allows a user to define its own functions and variables directly within the interactive environment. This is done by writing *let* in front of the function declaration:

```
ghci> let minus a b = a - b
ghci> let pi = 3.14
ghci> minus pi 2
1.14
```

As both parameters of *minus* is followed by the function name, it can be confusing to know which of the parameters that is subtracted from the other. Functions like these are called *prefix functions*, and are the default for user declared functions. However, since there are only two parameters, it can be used as a *infix function* - placed in between its given parameters - to make it more readable. This can be done in the following way:

```
ghci> pi `minus` 2
1.14
```

Notice that the defined functions do not specify the type of the their parameters or return values. This makes the functions *polymorphic*, and enables them to determine the result type based on its given parameters. This means that the function works with more than one type, which can be revealed when inspecting the type of *multiplyTogether* in GHCi (:t):

```
ghci> :t multiplyTogether
multiplyTogether :: Num a => a -> a -> a -> a
```

The left hand side of the => sign specifies the *typeclass* of the function's parameters. **Num a** denotes that a is a *numeric* type, which includes Integer and Double. Knowing this, we interpret that *multiplyTogether* can accept any *numeric* type but expects all parameters to have the same type, as seen from the right hand side of the => sign. The $->$ signs separates the parameters from each other, where the last one always displays its return type. Naturally, the return value has same type as the given parameters. By including a type signature in the function definition, we can specify the input/output type as shown in Listing 3.2.

```
multiplyTogether :: Double -> Double -> Double -> Double
multiplyTogether x y z = x * y * z

multiplyTogether :: Int -> Int -> Int -> Int
multiplyTogether x y z = x * y * z
```

Listing 3.2: Type signatures

Even though the compiler figures out the type by itself, it is good practice to explicitly state a function's type, especially when it is used for hardware generation.

The basic Haskell library provides a lot of predefined functions for performing common operations in our programs. In fact, +, -, /, etc. are functions as well, that goes between the parameters, which makes them infix functions. A complete overview of the base package of Haskell can be found at [1].

3.1.2 Data structures

Sometimes, a function is required to return several values. This is the case for the well-known abc-formula, made as a Haskell function in Listing 3.3. The output values are encapsulated in parentheses that makes it a data structure called a *tuple*. A tuple combines a number of elements into a structure that can contain several different types. Also notice how the code has been made more readable with a *where* clause.

```haskell
functionRoot :: Int -> Int -> Int -> (Int, Int)
functionRoot a b c = (x1,x2)
  where
    x1 = ((-b) + sqrt d) / (2*a)
    x2 = ((-b) - sqrt d) / (2*a)
    d  = b*b - 4*a*c
```

Listing 3.3: ABC-formula

It is also possible to define our own tuples with appropriate names. E.g, a set of coordinates consisting of different values would be appropriate to encapsulate in such a data structure. Geographically coordinates for instance, are often expressed in either latitude and longitude or DMS (degrees, minutes and seconds) with an orientation symbol. We can make tuples to represent this in the following way:

```haskell
type LatLong = (Double, Double)
type DMS = (Char, Int, Int, Double)
type Coordinates = (DMS, DMS)
```

Note that the keyword for a tuple is *type*. Elements in a tuple are accessed by their position, which can work fine for few elements, but may become unclear when it becomes more. In such cases, there is another more applicable structure called *records*, which gives each element its own name. Hence, elements are accessed by their given name rather than their position. The aforementioned tuples can be expressed as records with the *data* keyword, as follows:

```haskell
data LatLong = LatLong {lat::Double, long::Double}
data DMS = DMS {ori::Char, degrees::Int, minutes::Int, sec::Double}
data Coordinates = Coordinates {north_south::DMS,east_west::DMS}
```

These records can be used as part of a functions type signature as shown in Listing 3.4. Notice how the output is defined and how records are instantiated in line 10. This example makes use of three predefined functions: *floor* which returns the number in front of a decimal point, *abs* which returns the absolute value of a given argument, and *fromIntegral* which converts from any *Integral* type (i.e. Integer or Int) to a *numeric* type (including Double).

```haskell
deg_to_dms :: Double -> DMS
deg_to_dms lat = DMS {ori=o,degrees=d,minutes=m,sec=s}
  where
    o = if lat > 0 then 'N' else 'S'
    d = floor lat
    md = abs (lat - (fromIntegral d)) * 60
    m = floor md
    s = (md - (fromIntegral m)) * 60

myTreasure = LatLong {lat=63.25498,long=10.23421}
myLatitude = deg_to_dms (lat myTreasure)
```

Listing 3.4: Degree to DMS

In order to display the data within the *myLatitude* record, we access its elements in the following way, from GHCi:

```
ghci> ori myLatitude
'N'
ghci> minutes myLatitude
15
```

For someone familiar with other programming languages, the *if then else* construction used in Listing 3.4 may already be well-known. However, Haskell has other useful choice constructs to chose from.

3.1.3 Choices

In addition to the *if then else* construction, Haskell makes use of a choice construct called *guards*. This is symbolized with the | symbol, and can replace the choice from Listing 3.4 as shown in Listing 3.5.

```haskell
o | (lat > 0)   = 'N'
  | otherwise   = 'S'
```

Listing 3.5: Guards

Guards also allows additional branches to be added, and should in every case include an *otherwise* option to capture every other case. There are also something called *pattern matching* and *case constructs* for matching a certain value or constructor. Listing 3.6 gives an example of how they can be used to determine a function's outcome.

```haskell
guess 5 = "You guessed the correct number!"
guess _ = "Try again."

guess x = case x of 5 = "You guessed the correct number!"
                    _ = "Try again."
```

Listing 3.6: Pattern matching and case constructs

The underscore indicates a *don't care* condition and covers all possible inputs. The lines are evaluated sequentially, meaning that it would always result in the first branch if the lines where switched. These methods are essential when operating with *recursion* as we will see in the following subsection.

3.1.4 Recursion

Recursion is a much used methodology in functional programming, and introduces an effective method for evaluating lists of data. Consider the quicksort algorithm presented in Listing 3.7.

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
  where
    lesser = filter (< p) xs
    greater = filter (>= p) xs
```

Listing 3.7: Quicksort algorithm

This is a well-known algorithm for sorting a list, by using a so-called divide-and-conquer technique. It divides a list into smaller parts which can be quickly sorted and merged together in the correct ordering. The list is divided based on a reference value (pivot element), which in our example is chosen to be the first element of the list (notice how this is done using the : notation). For each function call, the remaining elements in the list are divided into two smaller lists - lesser and greater. The function then calls itself recursively for each of these new lists, and continues the process until the list becomes empty ([]). The ++ function ensures that the resulting lists (consisting of only one element) are combined into one list in ascending order. Figure 3.1 is included to illustrate the behavior of this algorithm.

It might not be obvious from the implementation of the quicksort algorithm,

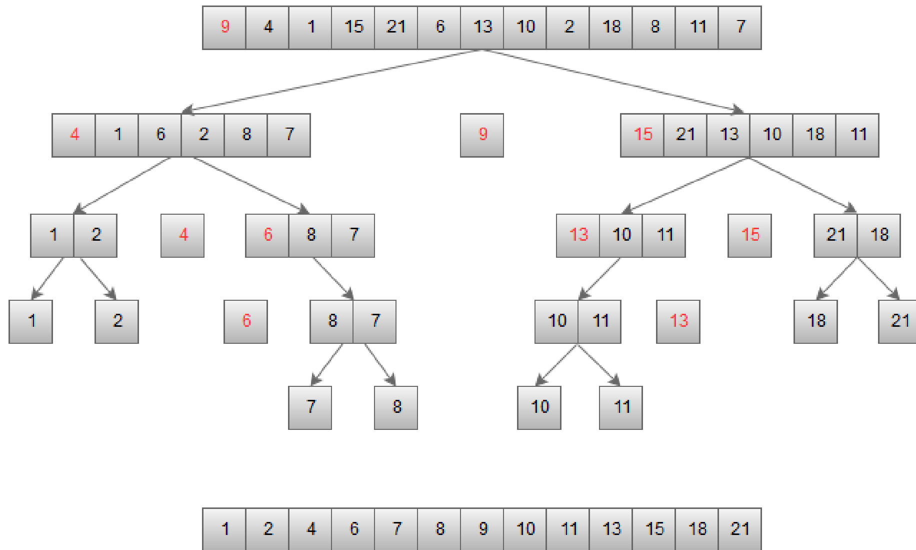

Figure 3.1: Quicksort algorithm example

but the function that does the filtering called *filter*, is actually accepting a *function* as an argument, making it a topic for the next subsection - *higher order functions*.

3.1.5 Higher order functions

A *higher order function* is a function that either accepts other functions as arguments, or has a function as return type. As mentioned in the previous subsection, the *filter* function is expecting a function as one of its argument, thus making it a higher order function. This can be revealed when examining its type:

```
ghci> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

The parentheses in the type signature is an argument expected to be a function, which evaluates some type *a* and returns a *Bool*. When studying the function calls in Listing 3.7, we see that the arguments that are expected to be a function

are given as $(< p)$ and $(>= p)$. Both $<$ and $>=$ are functions that expects two parameters, but in this case, they are only given one, namely p. In cases like these, the function is actually returning **a different function**, which serves a different purpose. The new function will in this case expect one argument to compare its "previous argument" p, with. In fact, every function in Haskell officially only takes **one** parameter. The reason why functions works with several parameters is because they evaluate the first argument, and then returns a new function, crafted to handle the next argument. Not until the last argument is evaluated, will the result share type with the return value of the original function - which may as well be another function. This means that the type signatures presented in Listing 3.8 can be used interchangeably for the $<$ functions.

```
(<) :: (Ord a) => a -> a -> Bool
(<) :: (Ord a) => a -> (a -> Bool)
```

Listing 3.8: Interchangeable type signatures

All functions that accept several parameters are known as *curried functions*. This includes the very first function *multiplyTogether* presented in this chapter, which means that we are actually evaluating a sequence of functions rather than all of its arguments at once. Listing 3.9 shows an alternative type signature for *multiplyTogether*.

```
multiplyTogether :: Num a => a -> a -> a -> a
multiplyTogether :: Num a => a -> (a -> (a -> a))
```

Listing 3.9: Alternative type signature for multiplyTogether

It is important to be aware of how curried functions are evaluated, but for simplicity's sake, we can assume that functions are evaluating several parameters at once.

There are a few commonly-used higher order function that often recur, when operating with a list in of data. These are discussed in the following sections:

map

The **map** function is heavily used when working with a single list. It applies a function to each element of a given list and returns the result as a new list. Figure 3.2 illustrates its behavior.

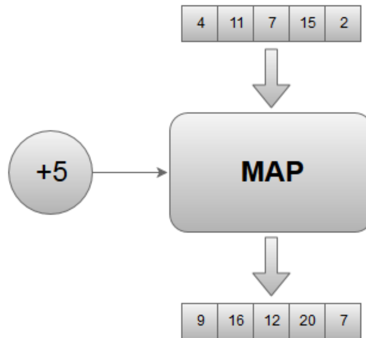

Figure 3.2: Example - map function

Note that a list can consist of other lists as well, or even tuples. Some of the function's application area is shown below:

```
ghci> map (+4) [2,3,6,1,7]
[6,7,10,5,11]
ghci> map (replicate 4) [2,4..10]
[[2,2,2,2],[4,4,4,4],[6,6,6,6],[8,8,8,8],[10,10,10,10]]
ghci> map snd [(3,"blue"),(7,"yellow"),(2,"red"),(5,"green")]
["blue","yellow","red","green"]
```

foldl and foldr

When it is required to go sequentially through a list from the left or the right, we can use either *foldl* or *foldr*, respectively. From an initial value, a function is applied to each element in order of the list. This is illustrated by Figure 3.3 for the *foldl* function.

Some examples for the *foldl* function is given below:

```
ghci> foldl (+) 0 [2,5,4,3]
14
```

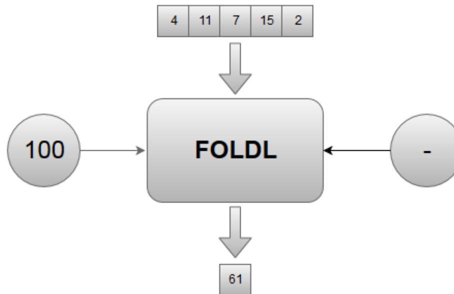

Figure 3.3: Example - foldl function

zipWith

When operating with two lists, the **zipWith** function can be useful. As the name suggests, the two lists are zipped together into one list, as the result of a given function. The elements of both lists are given as arguments to the function, which creates a new list based on its results, as Figure 3.4 illustrates.

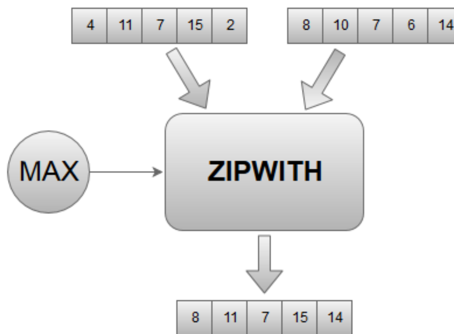

Figure 3.4: Example - zipwith function

The *zipWith* function can for instance be used in the following scenarios:

```
ghci> zipWith (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
```

3.1.6 Lambda expressions

Lambda expressions are used to define anonymous functions that are only used once. They are typically fairly simple, and are used together with higher-order functions. Lambda expressions starts with a \, followed by its parameters. Instead of an equal-sign as ordinary functions uses, we use $->$ to separate the function parameters from its body. Some examples are given in Listing 3.10.

```
map (\x -> multiplyTogether 3 5 x) xs
zipWith (\x y -> if (x*2 > y) then True else False) xs ys
```

Listing 3.10: Lambda examples

The first example passes each element of *xs* to the previously defined *multiply-Together* function. The second example compares the elements of two lists, *xs* and *ys* with each other, where the elements of *xs* is doubled. If the doubled element of *xs* is greater than the element from *ys*, then it returns *True*, in any other case it returns *False*.

3.2 CλaSH

3.2.1 From Haskell to CλaSH

This section addresses some of the differences between CλaSH and pure Haskell. As previously mentioned, hardware design requires values to have a fixed-size throughout the design, in order to derive circuits from the description. Hence, it is required to specify the number of bits for number types, and operate on fixed-sized lists. CλaSH introduces a library which provides the designer with types such as *Signed n*, *Unsigned n*, *Bit*, *BitVector n* and *Vector n a*.

Number types

Instead of operating with e.g. *Integer*s or *Double*s, we specify the type to be either *Signed n* or *Unsigned n*. An example is given in Listing 3.12.

```
multiplyTogether :: Signed 16 -> Signed 16 -> Signed 16 -> Signed 16
multiplyTogether x y z = x * y * z
```

Listing 3.11: Signed type

If it is desirable to resize the resulting value to be represented as a different number of bits, we can for instance use *signExtend* or *truncateB*, as Listing 3.12 shows.

```
my_value     = multiplyTogether 3 4 2
my_value_s32 = (signExtend my_value) :: Signed 32
my_value_s8  = (truncateB my_value)  :: Signed 8
```

Listing 3.12: Resizing values

It is also possible to convert a Signed type into Unsigned, and vice versa. In such cases, we use the function *fromIntegral* to convert between types, as Listing 3.13 shows.

```
my_value = (fromIntegral my_value) :: Unsigned 16
```

Listing 3.13: From Signed to Unsigned

In cases where we are operating with a single bit, we can use the **bit** type instead of **Unsigned 1**.

Lists

As Haskell operates with infinite lists, it is not suited for hardware design. Instead, CλaSH introduces the **Vector** type for representing lists. A Vector is simply a list with its length encoded in its type, such that the number of elements may never change. To instantiate a Vector, we can use the examples shown in Listing 3.14, interchangeably.

```
my_vector1 = (1:>2:>3:>4:>5:>Nil) :: Vector 5 (Signed 8)
my_vector2 = $(v [1,2,3,4,5])     :: Vector 5 (Signed 8)
```

Listing 3.14: Instantiation of Vectors

Note that the second example uses **v** to transform a list to a Vector. The elements of a Vector can be accessed by the **!!** function. For reading the third

index of a Vector, do the following:

```
ghci> my_vector1 !! 2
3
```

When the elements of a Vector represents bits, the Vector type can be changed to the *BitVector n* type. In the implementation, it is possible to convert between the two types with **v2bv** and **bv2v**, as shown in Listing 3.15.

```
my_vector    = (1:>0:>0:>1:>1:>Nil) :: Vector 5 (Unsigned 1)
my_bitvector = 0b10011 :: BitVector 5
my_vector_as_bv   = v2bv my_vector
my_bitvector_as_v = bv2v my_bitvector
```

Listing 3.15: Conversion between Vector and BitVector

If a circuit is said to have a Vector as input/output instead of a BitVector, the CλaSH compiler will generate n different input/output signals. Therefore, it is important to use either the BitVector- or Signed/Unsigned type at the top function, in order to group the bits to the same input/output port.

3.2.2 Higher-order functions and recursion

As previously established, CλaSH has poor support for recursive functions. This is made up for by the built-in support for higher order functions, that have been redefined in CλaSH to work with the Vector type instead of lists. This way, the compiler avoids unsupported recursion calls, and can produce finite circuit designs. The function names are the same, i.e. **map**, **foldl** and **zipWith**, and are used in the same way as the original functions. In addition to the well-known higher-order functions, CλaSH includes altered versions of these which returns the index in addition to the element. They are indicated by a leading **i** in the function name, i.e. **imap**, **ifoldl** and **izipWith**.

```
ghci> let xs = (5:>4:>8:>1:>2:>4:>5:>Nil)
ghci> let ys = imap (\i a -> (xs !! (i+2)) * a) (2:>5:>7:>3:>Nil)
ghci> ys
<16,5,14,12>
ghci> izipWith (\i x y -> if (i==0) then 1 else x*y) xs ys
<1,20,112,12>
```

3.2.3 Sequential circuits

All sequential circuits in CλaSH, work on values of type: **Signal a**. The **Signal** type specifies that it is an infinite stream of samples, where its components are synchronized to a global clock. Hence, a function with **Signal**s in its type signature, is executed at each tick of the clock. To delay the stream by one clock cycle, we use a **register-** or **regEn** function, as shown in Listing 3.16.

```
foo :: Signal Bit -> Signal Bit -> Signal Bool
foo input enable = out
    where
        a = register 0 input
        b = regEn 0 enable input
        out = a && b
```

Listing 3.16: Register functions

The first argument of the register and regEn function specifies the initial value at time 0. As the regEn function models a register with an enable signal, it requires an additional argument.

State

Due to the immutable nature of Haskell, functions can not keep state or internal values. In order to model state, we therefore need to "tie the knot" between the inputs and outputs of the function. This way, the function can calculate the next state based on its input, and receive the new state as input in the following clock cycle. This creates a feedback loop on the outside of the function, and is an idiom that corresponds to the actual schematics of a sequential circuit. Such functions can be modeled as a Mealy machine, as Figure 3.5 illustrates.

To model sequential functions in CλaSH, the combinational function needs to be written in the following way:

```
foo state inputs = (next_state, outputs)
```

Assuming that we want to have a valid signal dependent on the value of a counter, we can define the function *count_to_100*, as depicted in Listing 3.17. Note that this function does not use the **Signal** type, since it is combinational.

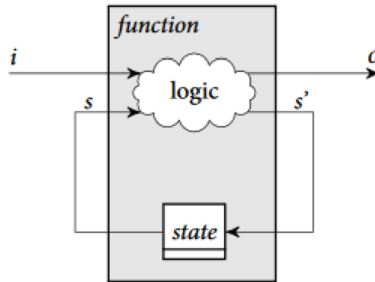

Figure 3.5: Mealy machine: Figure copied from [29]

```
count_to_100 :: Unsigned 8 -> Bool -> (Unsigned 8, Bool)
count_to_100 c enable = (c',reached_100)
    where
        c' = if (enable && c' < 100) then (c + 1) else c
        reached_100 = if (c' >= 100) then True else False
```

Listing 3.17: Combinational function

The next step is to apply a helper-function to model the combinational function as a synchronous function, for describing a Mealy machine. This is done with the **mealy** function available in CλaSH, and can be seen in Listing 3.18.

```
valid_signal1 = mealy count_to_100 0 enable_signal
```

Listing 3.18: Modeling a function as a Mealy machine

The parameters of the mealy function is the transfer function followed by its initial state, and then the function's input.

In the actual design presented in this paper, we use a version of the mealy function called **mealyB**, which does automatic *bundle*ing. This means that functions with several inputs and outputs will automatically *bundle* the input signals, and *unbundle* the output signals, to suit the type signature of the mealy function. Consider a function **foo** with the type signature given in Listing 3.19.

```
foo :: Int -> (Signal Bit,Signal Bool) -> (Int, (Signal Bool,Signal Bit))
```

Listing 3.19: Type signature of a function

The mealy function expects an input type of **Signal i**, and requires the input signals to be bundled with a **bundle** function. As this will produce an output type of **Signal o**, the output needs to be unbundled with an **unbundle** function. In this specific case, the bundle and unbundle function will perform the following conversions:

```
bundle :: (Signal Bit,Signal Bool) -> Signal (Bit,Bool)
unbundle :: Signal (Bool,Bit) -> (Signal Bool,Signal Bit)
```

To model the function **foo** as a Mealy machine, we can either use the bundle and unbundle functions manually with the mealy function, or let the mealyB function handle this automatically, as shown in Listing 3.20.

```
(outA, outB) = unbundle (mealy foo 0 (bundle (inpA, inpB)))
(outA, outB) = mealyB foo 0 (inpA, inpB)
```

Listing 3.20: Automatic bundling and unbundling

Note that CλaSH can also model Moore machines, in a similar way.

Explicitly clocked functions

For designs utilizing multiple clock-domains, we attach a ’ to the end of the function-name (e.g. mealy’, register’, regEn’) to specify the driving clock of the function/circuit. The function’s input signals then needs to be synchronized to the same clock, which can be specified with the **Signal’ clk a** type. In order to synchronize signals between the clock-domains, we make use of either the **unsafeSynchronizer**- or the **dualFlipFlopSynchronizer** function[1]. Listing 3.21 shows an example of this.

[1] dualFlipFlopSynchronizer is a synchroniser based on two sequentially connected flip-flops to reduce metastability. unsafeSynchronizer should never be used to synchronize signals between two asynchronous clock-domains.

```
type ClkA = 'Clk "clkA" 500
type ClkB = 'Clk "clkB" 1000

clkA :: SClock ClkA
clkA = sclock
clkB :: SClock ClkB
clkB = sclock

foo :: Signal' clkA (Signed 8)
    -> Signal' clkB Bool
    -> Signal' clkA (Signed 8)
foo input valid = output
    where
        valid_sync = dualFlipFlopSynchronizer clkB clkA False valid
        output = if (valid_sync) then
                    mealy' clkA foo2 init_state input
                 else
                    0

foo2 state input = (next_state, output)
    where
        ...
```

Listing 3.21: Multiple clock-domains and synchronization

3.2.4 RTL generation

In order for the CλaSH compiler to transform the functional description to RTL code, it needs to have a starting point. This is specified with a *topEntity* function as seen in Listing 3.22.

```
topEntity :: Signal Bool
          -> Signal Bool
topEntity enable = mealy count_to_100 0 enable

count_to_100 c enable = (c',reached_100)
    where
        c' = if (enable && c' < 100) then (c + 1) else c
        reached_100 = if (c' >= 100) then True else False
```

Listing 3.22: TopEntity function

In the interactive environment of CλaSH, one of the following commands can be issued to generate the desired RTL code:

```
ghci> :vhdl
ghci> :verilog
ghci> :systemverilog
```

To generate a simple RTL testbench in addition to the circuit description, two additional functions needs to be specified, namely *testInput* and *expectedOutput*. The testInput function will contain the stream of input values, while the expectedOutput function will - as the name suggests - list the expected output. For creating the input stream, another helper function *stimuliGenerator* is required in order to feed the circuit with a new input on each tick of the clock. Similarly, the expectedOutput function needs an *outputVerifier* function in order to evaluate one output at a time. Listing 3.23 shows an example for creating a testbench for the above topEntity function.

```
testInput = stimuliGenerator (repeat True :: Vec 100 Bool)
expectedOutput = outputVerifier (repeat False :: Vec 100 Bool) ++ (True:>Nil)
```

Listing 3.23: Automatic testbench generation

It can be helpful to generate a testbench in order to examine the timing behavior of the circuit description.

Chapter 4

Methodology

This chapter presents the Haskell/CλaSH implementation of a Viterbi decoder. First, an overview of the implementation is given, with the signal interface and a block diagram. Next, each function is given as a code snippet, and their functionality is briefly described. Furthermore, we show how simulations were performed throughout the design process, and discuss the details of the verification phase.

4.1 Overview

The functional architecture of the Viterbi decoder is based on the implementation methods described in Section 2.7.3. The top-module consists of four main functions modeled as Mealy machines, namely: **decoder**, **valid_ctrl**, **flush_ctrl** and **flush_sr**. All outer signals are synchronized to the same clock, *ck*. In addition, there is another clock *ckDiv*, driving the **decoder** function and all its internal functions, since they are operating on softbit-pairs instead of single softbits. This clock is synchronous to *ck*, and operating on half the frequency when handling a stream of valid softbits, but is switched off when the decoder is inactive. This is done with a clock gating unit which is not part of the Haskell implementation, but provided by Nordic Semiconductor and not shown in this thesis. The unit enables the clock based on the *enckDiv* signal, which goes high when a valid softbit-pair is received (or in the case of clear or flush). This

way, unnecessary switching of the *ckDiv*-clock is avoided. The overall system is illustrated by Figure 4.1, which also depicts the data dependencies between functions.

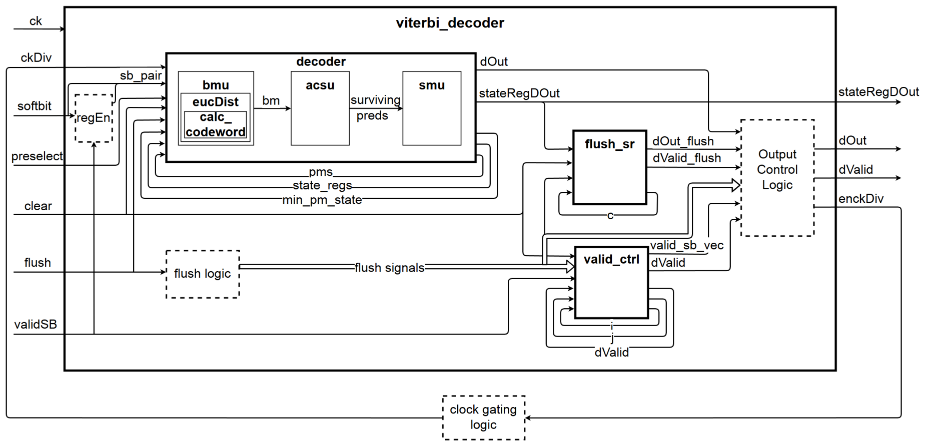

Figure 4.1: Viterbi decoder block diagram

Note that this block diagram does not include synchronization logic between the clock-domains and the functions' internal behavior. The signal interface of the top-module is listed below:

Signal	In/Out	Description
ck	In	Clock signal with frequency corresponding to the softbit input rate
ckDiv	In	Clock signal driven by enckDiv
ck_rstn	In	Asynchronous reset
ckDiv_rstn	In	Asynchronous reset
softBit[3:0]	In	Input data signal
validSB	In	Signal indicating that the softbit is valid
clear	In	Signal used to clear the decoder's internal state
flush	In	Signal used to read the remaining bits in the state register with the least path metric (PM)
preselect[2:0]	In	Only relevant if clear is high: selects the initial state of the trellis by setting its initial PM to zero
dOut	Out	Decoded output bit
validB	Out	Signal indicating that dOut is valid
enckDiv	Out	Output that enables the ckDiv clock
stateRegDOut[19:0]	Out	Parallel readout of state register with the least PM, for applications requiring near zero latency flush.

4.2 Implementation details

This section presents the details of the implemented Viterbi decoder. Each function will be presented and shortly described. Some examples are shown to illustrate the internal behavior of the functions. The entire code can be seen in Appendix A.

The top-module **viterbi_decoder** is presented in Listing 4.1 (Figure 4.1). Its inputs and outputs are synchronized to the *ck* clock and the function executes on every clock tick. In short, the softbit-pair and the signals are synchronized to ckDiv and given to the **decoder** function (line 12-18). The **decoder** function and the **valid_ctrl-** and **flush_sr** functions are all modeled as mealy machines (line 24-32). Together, they make up the main functionality of the decoder and provide the outputs for the top-module. Some registers have been added to the top-module to ensure correct timing behavior. Note that decoded output bits

are either read from the **decoder** function or the **flush_sr** function, depending on the *flush* signal (line 38-40).

```haskell
viterbi_decoder :: SClock ck -> SClock ckDiv                                    1
                -> Signal' ck (Signed 4)                                        2
                -> Signal' ck (Bool)                                            3
                -> Signal' ck (Bool)                                            4
                -> Signal' ck (Bool)                                            5
                -> Signal' ck (BitVector 3)                                     6
                -> Signal' ck (Bit, Bool, Bool,(BitVector 20))                  7
viterbi_decoderck ckDiv softbit validSB clear flush preselect = out             8
  where                                                                         9
    sb1 = (regEn' ck 0 validSB softbit)                                         10
    sb2 = softbit                                                               11
    -- Signals synchronized to ckDiv                                            12
    sb_pair        = dualFlipFlopSynchronizer ck ckDiv (0,0)                    13
                     (bundle' ck (sb1,sb2))                                     14
    flush_sync     = dualFlipFlopSynchronizer ck ckDiv False flush              15
    clear_sync     = dualFlipFlopSynchronizer ck ckDiv False clear              16
    preselect_sync = dualFlipFlopSynchronizer ck ckDiv                          17
                     (0 :: BitVector 3) preselect                               18
    -- Flush Signals                                                            19
    flushDelayed = mux flush (register6' ck False flush) flush                  20
    flushPosEdge = isRising' ck False (register' ck False flush)                21
    flushPosEdged = isRising' ck False (register' ck False flushPosEdge)        22
    -- Mealy machines                                                           23
    (dOut, sr_dOut)              = mealyB' ckDiv (decoder)                      24
                                     (init_pms,init_state_regs,0)               25
                                     (sb_pair,flush_sync,clear_sync,preselect_sync) 26
    (validB_temp,validSB_vec)   = mealyB' ck (valid_ctrl)                       27
                                     (0,0,False)                                28
                                     (clear,validSB,flushPosEdge,flushPosEdged) 29
    (dOut_flush,validB_flush)    = mealyB' ck (flush_sr)                        30
                                     (0)                                        31
                                     (flushDelayed,stateRegDOut,clear)          32
    -- Outputs                                                                  33
    validB       = (register' ck False validB_temp)                             34
    dOut_sync    = (dualFlipFlopSynchronizer ckDiv ck 0 dOut)                   35
    stateRegDOut = unsafeSynchronizer ckDiv ck sr_dOut                          36
    enckDiv      = (validSB_vec .||. flushPosEdge .||. clear)                   37
    out = mux flushDelayed                                                      38
              (bundle' ck (dOut_flush,validB_flush,enckDiv,stateRegDOut))       39
              (bundle' ck (dOut_sync,validB,enckDiv,stateRegDOut))              40
```

Listing 4.1: Top-module of the Viterbi decoder

Following, the functions modeled as a mealy machine are presented. First, we

look at the **decoder** function and all of its sub-functions.

4.2.1 Decoder

The **decoder** function is modeled as a Mealy machine, and is responsible for calculating the decoded output on each rising edge of ckDiv. Listing 4.2 shows the implemented function. Note that **pms**, **state_regs** and **min_pms_state** are its state inputs, and are used to calculate their next state with the **bmu** (line 9), **acsu** (line 17) and **smu** (line 20) functions. The decoded output bit *dOut*, is read from the leftmost position of the state register with the least PM, given by the value of *min_pms_state* (line 22). Additionally, the same state register is returned by the function, so that in the case a flush is issued, all remaining bits in the state register can - if required - be read simultaneously.

```
decoder (pms,state_regs,min_pm_state)                                          1
        (sb_pair,flush,clear,preselect)                                        2
        = ((pms',state_regs',min_pm_state'),                                   3
                       (dOut,stateRegDOut))                                    4
  where                                                                        5
    sb1 = (signExtend (fst sb_pair))::Signed 8                                 6
    sb2 = (signExtend (snd sb_pair))::Signed 8                                 7
    sb_pair_se = (sb1,sb2)                                                     8
    bms = bmu sb_pair_se                                                       9
    (pms',min_pm_state',surviving_preds') =                                    10
        if (flush) then                                                        11
            (pms,min_pm_state,init_surviving_preds)                            12
        else if (clear)                                                        13
            ((imap (\i x -> if (i == (unpack preselect::Index 8))              14
                            then (0::Unsigned 8)                               15
                            else x) init_pms), 0, init_surviving_preds)        16
        else (acsu pms bms)                                                    17
    state_regs' | (flush)     = state_regs                                     18
                | (clear)     = init_state_regs                                19
                | otherwise   = smu state_regs surviving_preds'                20
    -- Outputs                                                                 21
    dOut = (state_regs' !! (min_pm_state')) !! (0 :: BitVector 3)              22
    stateRegDOut = v2bv (state_regs' !! min_pm_state')                         23
```

Listing 4.2: Decoder function with all the decoding logic

Note that the initial states *init_pms*, *init_surviving_preds* and *init_state_regs* are not defined in this code snippet. See Appendix A for more details. In the following, the three main units (BMU,ACSU and SMU) of the **decoder** function

is presented and further described.

Branch Metric Unit (BMU)

The **bmu** function must be seen together with two other functions - **eucDist** and **calc_codeword**. Listing 4.3 presents the implementation of the **bmu** function. It takes a softbit-pair (sb_pair) as an argument and calculates the BMs based on these. This is done by looping through *pred_states1* and *pred_states2*, (i.e. the first and second predecessor of each state), and calculates their BMs: *bm1s* and *bm2s*, respectively. At the end, their elements are combined into a Vector of tuples, so that the first tuple in the Vector contains $BM_{0\to0}$ and $BM_{1\to0}$.

```
bmu sb_pair = bms                                                          1
  where                                                                    2
    bm1s = imap (\i a -> eucDist a (encInps !! i) sb_pair) pred_states1    3
    bm2s = imap (\i a -> eucDist a (encInps !! i) sb_pair) pred_states2    4
    bms = zipWith (\x y -> (x,y)) bm1s bm2s                                5
```

Listing 4.3: BMU function

Each predecessor state (*pred*) will be passed to the **eucDist** function together with the softbit-pair (*sb_pair*) and the encoder input (*encIn*). The function then calculates the euclidean distance (i.e. the branch metric (BM)) for one of the two outgoing branches from the given predecessor state, based on the given encoder input. The **eucDist** function is shown in Listing 4.4.

```
eucDist pred encIn sb_pair = bm                                            1
  where                                                                    2
    sb1 = fst sb_pair                                                      3
    sb2 = snd sb_pair                                                      4
    codewordA = calc_codeword polya pred encIn                             5
    codewordB = calc_codeword polyb pred encIn                             6
    e_distA = if      ((codewordA == 1) && (sb1 < 0))  then sb1 * (-1)     7
              else if ((codewordA == 0) && (sb1 >= 0)) then sb1            8
              else 0                                                       9
    e_distB = if      ((codewordB == 1) && (sb2 < 0))  then sb2 * (-1)     10
              else if ((codewordB == 0) && (sb2 >= 0)) then sb2            11
              else 0                                                       12
    bm_temp = e_distA + e_distB                                            13
    bm = fromIntegral (bm_temp)::Unsigned 8                                14
```

Listing 4.4: Function calculating the branch metric for one branch

The **eucDist** function uses the **calc_codeword** function for calculating the codewords associated to the branch. These are seen in relation with the softbits, to calculate the branch metric. The **calc_codeword** function can be seen in Listing 4.5. Note that the encoder's polynomials must be known in order to calculate this. In this thesis, *polya* and *polyb* are 15 (0b1111) and 11 (0b1011), respectively.

```
calc_codeword poly state encIn = bit                     1
  where                                                  2
    temp = (poly .&. state)::Unsigned 4                  3
    bit = encIn `xor` (reduceXor temp)                   4
```

Listing 4.5: Function calculating a codeword produced by the encoder

Add-Compare-Select Unit (ACSU)

The **acsu** function calculates the next PMs based on the previous PMs and the calculated BMs. Listing 4.6 shows the implementation. It first adds together the current PMs with the BMs (line 5-6), then selects the branches resulting in the least PM (line 7) for the next time-step. This way, each state will only have one incoming branch. The smallest PM are subtracted from each state's PM in order to avoid overflow (line 12), and returned by the function. In addition, it returns the state with the least PM (line 13), and the surviving predecessor of each state (line 14-18).

```haskell
acsu pms bms = (pms',min_pm_state,surviving_preds')
  where
    bm1s = map fst bms
    bm2s = map snd bms
    pm1s = imap (\i x -> (+) (pms !! (pred_states1 !! i)) x) bm1s
    pm2s = imap (\i x -> (+) (pms !! (pred_states2 !! i)) x) bm2s
    pms_temp = zipWith (min) pm1s pm2s
    min_pm = fst (mapAccumL (\acc x -> if (x < acc)
                                        then (x,acc)
                                        else (acc,acc))
                                        (pms_temp !! 0) pms_temp)
    pms' = map (\x -> (x - min_pm)) pms_temp
    min_pm_state = fromMaybe 0 (findIndex (== min_pm) pms_temp)
    surviving_preds' = (izipWith (\i pm1 pm2 -> if (pm1 < pm2) then
                                                    (pred_states1 !! i)
                                                  else
                                                    (pred_states2 !! i))
                                                  pm1s pm2s) :: Vec 8 (Unsigned 8)
```

Listing 4.6: Function calculating the path metrics (PMs) and surviving predecessor states

Survivor Memory Unit (SMU)

The **smu** function seen in Listing 4.7, updates the state registers based on the surviving predecessor states returned by the ACSU. Each state copies the register of its predecessor, then bit shifts it with the decoded bit (i.e. the encoder's input bit for the state transition).

```haskell
smu state_regs surviving_preds = state_regs'
  where
    state_regs' =
        imap
            (\i x -> ((state_regs !! (surviving_preds !! i)) <<+ (encInps !! i)))
            states
```

Listing 4.7: Function calculating the next state registers from the surviving predecessor states.

4.2.2 Valid control

The **valid_ctrl** function is responsible for flagging a decoded output bit as valid. It does this by keeping count of how many valid softbit-pairs have been received by the decoder. When the state registers becomes filled, the decoded output bit is flagged valid each time a new valid softbit-pair is received. The function can be seen in Listing 4.8.

```
valid_ctrl (i,j,validB) (clear,validSB,flushPosEdge,flushPosEdged,flush) =
           ((i',j',validB'), (validB',validSB_vec))
  where
    j' = if (validSB && (j < 2))     then (j + 1)
         else if (j >= 2 && validSB) then 1
         else if (j >= 2 || clear)    then 0
         else j
    validSB_vec = if (j' >= 2) then True else False
    i' = if (clear || flush)                     then 0
        else if (validSB_vec && (i < td + 2)) then (i + 1)
        else i
    validB'  | (validB)        = False
             | (flush)         = False
             | (clear)         = False
             | ((i' >= td) && flushPosEdge)       = True
             | ((i' >= td + 2) && validSB_vec)   = True
             | ((i' >= td + 2) && flushPosEdged) = True
             | otherwise       = False
```

Listing 4.8: Function controlling the validB signal, by counting the number of valid softbit-pairs.

4.2.3 Flush state register

Listing 4.9 shows the **flush_sr** function, which reads the remaining bits in the current state register (stateRegDOut). Since this function's signals are synchronized to the *ck*-clock, the read-out is accelerated and the *validB* signal is constantly kept high for 20 clock cycles (traceback depth of decoder). Note that *stateRegDOut* is a BitVector, where its elements are accessed by a single **!**.

```
flush_sr c (flush,stateRegDOut,clear) = (c',(dOut,validB))                              1
  where                                                                                  2
    c' = if (not flush) then 0 else (if (flush && c < td) then (c + 1) else c)           3
    dOut = (stateRegDOut ! (td - c'))                                                    4
    validB = if (clear) then False else (if (flush && c < td) then True else False)      5
```

Listing 4.9: Function reading the remaining bits of a state register when the flush signal goes high

4.3 Simulation

As mentioned in chapter 2, the design can be simulated within the interactive environment of Haskell (GHCi). The implementation is modularized, such that each unit is represented as its own function, and can be tested in isolation of each other. This makes the debugging process easier as it only targets the separate module rather than the overall system. Simulations are performed by executing the function and provide it with its expected input type.

The **bmu** function can be simulated with various combinations of softbits (see Figure 2.13) to examine the resulting BMs, in the following way:

```
ghci> bmu (7,7)
<(14,0),(0,14),(7,7),(7,7),(0,14),(14,0),(7,7),(7,7)>
ghci> bmu (-8,7)
<(7,8),(8,7),(15,0),(0,15),(8,7),(7,8),(0,15),(15,0)>
ghci> bmu (-1,0)
<(0,1),(1,0),(1,0),(0,1),(1,0),(0,1),(0,1),(1,0)>
```

Note that the values close to the least confident 0s and 1s, produces smaller branch metric values. The same simulation process can be done for the function that represents the ACSU as well, but since this function operates on the calculated branch metrics, we let the BMU calculate these for us, instead of manually writing them:

```
ghci> let bms = bmu (-8,7)
ghci> let (pms,min_pm_st,surv_preds) = acsu init_pms bms
ghci> pms
<7,7,0,0,7,7,0,0>
ghci> min_pm_st
2
```

```
ghci> surv_preds
<0,3,5,6,1,2,4,7>
```

The calculated surviving predecessor states can be further used to simulate the **smu** function. In addition, the state registers needs to be filled with bits. This can be done within the interactive environment of CλaSH in the following way:

```
ghci> let sr0 = $(v [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1])
ghci> let sr1 = $(v [0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1])
ghci> let sr2 = $(v [0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,0])
ghci> let sr3 = $(v [0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0])
ghci> let sr4 = $(v [0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1])
ghci> let sr5 = $(v [0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,1,1])
ghci> let sr6 = $(v [1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1])
ghci> let sr7 = $(v [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
ghci> let stRegs = (sr0:>sr1:>sr2:>sr3:>sr4:>sr5:>sr6:>sr7:>Nil)
ghci> smu stRegs surv_preds
<<1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0>
,<0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0>
,<0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,1,1,0>
,<1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,0>
,<0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,1>
,<0,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,0,1>
,<0,0,0,0,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1>
,<1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1>>
```

To simulate the **decoder** function, the previously calculated states can be used as input together with a softbit-pair and the other input signals. The following is an example of this:

```
ghci> let sbPair = ((4::Signed 4),(-6::Signed 4))
ghci> let pre_sel = 0 :: BitVector 3
ghci> decoder (pms,stRegs,min_pm_st) (sbPair,False,False,pre_sel)
((<11,4,7,0,11,4,7,0>,
<<1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0>
,<0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0>
,<0,0,0,0,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,0>
,<1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0>
,<0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,1>
,<0,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,0,1>
```

```
,<0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,1,1,1>
,<1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,1>>
,3)
,(1
,1111_1111_1111_1111_1110))
```

Note that the decoded output is 1 and that state 011 have the least PM, since its state register is returned as the *stateRegDOut*.

4.4 Verification

The generated SystemVerilog implementation has been through several verification steps. The first step was to verify the timing model with a testbench generated by CλaSH. In order to provide the test inputs, a simple Perl script was made to mimic an encoder (See Appendix C). The generated input values was contained in a separate file, which could be read by the **testInput** function in CλaSH. Listing 4.10 illustrates an example of this.

```
import Language.Haskell.TH.Syntax                                                  1
import CLaSH.Prelude                                                               2
import CLaSH.Prelude.Explicit                                                      3
import System.IO                                                                   4
import qualified Prelude as P                                                      5
                                                                                   6
testInput :: (Signal' ClkSys (Signed 4)                                            7
             ,Signal' ClkSys (Bool)                                                8
             ,Signal' ClkSys (Bool)                                                9
             ,Signal' ClkSys (Bool)                                                10
             ,Signal' ClkSys (BitVector 3))                                        11
testInput = (testSoftbits, testValidBits, testClear, testFlush, testPreselect)     12
                                                                                   13
testSoftbits = stimuliGenerator' ckSys z                                           14
   where                                                                           15
     z = $(do a <- qRunIO (readFile "sb_vec.txt")                                  16
              let x = P.map read (lines a) :: [Int]                                17
                  y = P.map fromIntegral x :: [Signed 4]                           18
              v y)                                                                 19
                                                                                   20
testValidBits = stimuliGenerator' ckSys (repeat True :: Vec 100 (Bool))            21
testClear     = stimuliGenerator' ckSys (repeat False :: Vec 100 (Bool))           22
testFlush     = stimuliGenerator' ckSys (repeat False :: Vec 100 (Bool))           23
testPreselect = stimuliGenerator' ckSys (repeat 0 :: Vec 100 (BitVector 3))        24
```

Listing 4.10: Stimuli for auto-generated testbench

Each softbit are read from a file (sb_vec.txt) and converted to its expected type **Signal' ClkSys Signed 4**. Note that this setup does not test the **clear-**, **flush-** or **preselect** signals, and expects that every softbit is valid.

The **expectedOutput** function is also required in order for CλaSH to generate a simple testbench. This function can be seen in Listing **??**. Note that *encoder_input_bits* is a Vector that is not shown here, but will contain the bits to compare the decoded output bits with.

```
expectedOutput :: (Signal' ClkSys (Bit, Bool, Bool, BitVector 20))
                -> Signal' ClkSys Bool
expectedOutput (output) = expected_dOut
  where
    unbundled_output = unbundle' ckSys output
    dOut          = (\(x,_,_,_) -> x) unbundled_output
    validB        = (\(_,x,_,_) -> x) unbundled_output
    enckDiv       = (\(_,_,x,_) -> x) unbundled_output
    stateRegDOut = (\(_,_,_,x) -> x) unbundled_output
    expected_dOut = outputVerifier' ckSys z dOut
      where
        z = encoder_input_bits
```

Listing 4.11: Function specifying the expected output for the auto-generated testbench

We can even runs simulations with the specified testInput:

```
ghci> let output = sampleN 100 (topEntity testInput)
[(0,False,False,0000_0000_0000_0000_0000),...]
ghci> let valid_output = (Prelude.filter (\(_,x,_,_) -> x) output)
ghci> let dOut = Prelude.map (\(x,_,_,_) -> x) valid_output
ghci> dOut
[1,0,0,1,0,0,1,0,0,1,1,1,0,1,1,0,0,1,1,1,0,1,1]
```

Exhaustive testing

The next verification step was performed by a testbench developed at Nordic Semiconductor, and made specifically for a Viterbi decoder. The testbench was reduced to only operate within the constraints of the implemented design. The

testbech provides over 10 000 000 test-bits, which were read from input files and used to compare the decoded bits with. This resulted in some bit errors, but all within the specified limits of the testbench. Figure 4.2 displays a portion of the generated waveform. Notice that the testbench provides the softbits as a continuous stream, as indicated by the **validSB** signal.

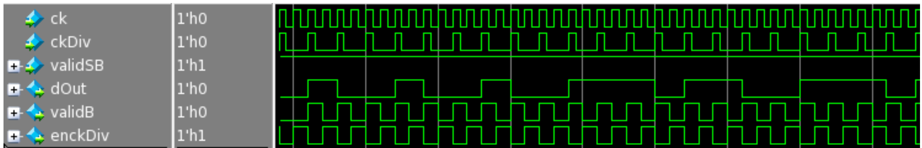

Figure 4.2: Waveform from SystemVerilog testbench 1

A second testbench - also provided by Nordic Semiconductor - was used for testing some of the corner-cases of the decoder. For instance, it would test what would happen if the **flush** signal was given as a single pulse. Furthermore, it would verify that the decoder was able to operate with softbits given several clock cycles apart. Figure 4.3 shows an example of this.

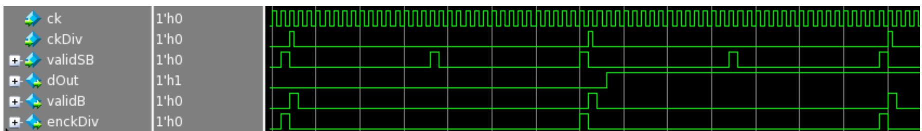

Figure 4.3: Waveform from SystemVerilog testbench 2

The performance was measured as the bit error rate (BER) between the CλaSH- and the reference design with a third testbench.

4.5 IP integration

As a final test, the Viterbi decoder was integrated in a complete SoC design, hereby replacing the existing decoder. The resulting SoC was then tested as a whole, with testbenches targeting the top-level system.

Chapter 5

Evaluation

This chapter evaluates CλaSH as a hardware description tool, and discusses its advantages and current issues. The presented results are based on experiences throughout the design process, and the features of functional programming. Furthermore, the quality of the generated SystemVerilog solution is compared with a reference design. The quality is based on the calculated bit error rate, synthesis results and from exhaustive testing cases.

5.1 Hardware design with functional programming

Common for the execution of digital hardware and the evaluation of a functional language is that they are both highly parallel. This semantic equivalence, gives functional programming an edge in describing hardware compared to imperative languages. As combinatorial circuits are immutable and can be directly modeled as mathematical functions, they translate well into a functional language. Sequential circuits on the other hand, can not keep their internal states in a function, as all functions are state-less. However, it is far from hard to model state, as it is all about creating a feedback loop between a function's input and its calculated output. In fact, this idiom is exactly what a feedback loop looks like in the schematics, but is not explicit in common HDLs as they

are in functional HDLs.

As hardware is described at a higher abstraction level in a functional language compared to a traditional HDL, it commonly results in far less code than a corresponding solution made in a HDL. However, the code length may not be representative for the time spent during the design process. Typically, functional languages tend to compress calculations and code blocks to a single line, much due to higher-order functions and the use of lambdas (see Section 3.1.5). Nevertheless, the code is still easier to debug in most cases, because of the interactive environment and the opportunity to test functions in isolation. Once a function has been thoroughly tested, it is not necessary to re-verify everything, as the features of Haskell does not allow it to ever change its internal behavior. When all required functions are created and verified, it is all about putting the pieces together and achieve correct timing behavior of the system.

As the correctness of design transformation from a functional HDL to RTL code can be mathematically proven, their written programs is highly reliable. This was also experienced when testing the design with several testbenches, and from a successful integration into a complete SoC.

5.1.1 CλaSH

The most commonly used features of Haskell seem to be supported by CλaSH. It gives the designer a freedom of expression, and allows newcomers to CλaSH to get started without learning a new language. CλaSH also introduces some new types and functions, tailor-made for hardware purposes. Sequential signals are given as a **Signal** type, which makes functions execute every tick of their associated clock. Moreover, CλaSH allows complex sequential circuits to be easily modeled as a Mealy- or a Moore machine to create the aforementioned feedback loop.

One of the issues that CλaSH currently has, is the poor support for recursive functions. The workaround for this has been to redefine the higher-order functions of Haskell to support the **Vector** type, and avoid unsupported recursion calls. Still, it limits the possibilities for the designer and might be considered the greatest disadvantage that CλaSH currently has.

Furthermore, the introduced **Signal** type which models a sequential signal, is not able to utilize all of Haskell's predefined functions. To deal with this, CλaSH provides alternative functions with a slightly different syntax. For instance, it

is required to use **.&&.** instead of **&&** and **.||.** instead of ||. Additionally, the **Signal** type does not support the use of *guards* or the *if-else-then* construct, and must use the **mux** function for making choices. It is not a big issue, but working directly with the **Signal** type does require a fairly good overview of the library of CλaSH.

Another thing worth mentioning about functional programming and CλaSH, is that any interaction with the outside world is not very intuitive. This interaction is a possible cause of side-effects (see Section 2.3), and must be avoided by using a structure known as a *monad*. Because of this, reading and writing to files, or reporting internal values during simulations can be hard.

As a whole, the experience with CλaSH has been very satisfying. From the beginning of the project, it has not been any big "change of plans", as all the ideas have been implementable in CλaSH. The poor support for recursive functions has not been an issue, as the predefined higher order functions has been able to tackle every problem. Moreover, since most functions from Haskell are supported, it has been possible to find extensive support for these.

5.2 Quality of Result

The CλaSH implementation is compared with a reference design, provided by Nordic Semiconductor. This is to give a measure of quality, since the implementation of multiple designs are out of scope for this thesis.

As there are a lot of available functions in Haskell/CλaSH, there are multiple ways to go about when implementing a Virterbi decoder. This makes it hard to argue whether the presented solution could have been made easier and more understandable for the common programmer. Another aspect that has not been concluded, is if a better quality of result could have been achieved with another design approach. In terms of area, number of registers, and the BER, there may be solutions which results in more efficient SystemVerilog code. As the details of the synthesis process is unclear to the designer, it is hard to know exactly what is generated, unless VHDL/(System)Verilog primitives are defined by the designer.

In this section, the quality of the solution made in CλaSH is compared to the reference design and discussed in more detail.

5.2.1 Exhaustive testing

As mentioned in Chapter 4, the CλaSH generated implementation has been through several verification phases. The testbenches were originally made for the reference design, and were performing exhaustive testing cases. Every testbench passed, which proves that the presented implementation is working as expected, and that CλaSH is in fact able to generate a fully-functioning SystemVerilog design.

5.2.2 Bit error rate

A separate testbench was used to calculate the bit error rate (BER) for both the CλaSH- and the reference design. The same input stimuli was given to both designs in form of softbits, so that the presented numbers are representative for the achieved quality:

Design	BER
CλaSH	128/204000
SystemVerilog	121/204000

It was found that the CλaSH design was producing 7 more bit errors compared to the reference design, which is a difference of 5%. Hence, the difference is not very significant.

5.2.3 Synthesis results

The CλaSH generated SystemVerilog code was synthesized as an application-specific integrated circuit (ASIC) and for an FPGA. Both synthesis processes compiled successfully from the generated SystemVerilog code. The resulting FPGA netlist - containing a complete digital radio - was transferred to an FPGA, but never fully tested due to time limitations. However, the digital radio was tested with a quick sanity check and found to be working properly.

The synthesis results for the ASIC design was compared to the reference design provided by Nordic Semiconductor. It was found that the CλaSH generated solution was using approximately 22% more flip-flops in total, compared to the design made in SystemVerilog. In terms of area, measured as the NAND-gate equivalence, the difference was 42%. It should be noted that the CλaSH

design was never optimized for area and number of registers, and that better results might be achievable in CλaSH. For instance, the CλaSH design uses synchronizers between the clock-domains, even though this could have been avoided.

5.2.4 Generated SystemVerilog code

The generated SystemVerilog code is not very readable, as there typically are a lot of files and auto-generated signal names[1]. The designer can exert some control of how the top level function is created with **TopEntity** annotations, which is all that is needed to integrate the IP in a larger system. However, it is often desirable to look at internal signals as well, when inspecting the waveform of a testbench. This can be cumbersome as every internal signal is given an auto-generated name. It also makes it harder to probe the design when creating properties with SystemVerilog Assertions (SVA).

In addition, one of the generated files required some editing in order to synthesize the design as an ASIC. These changes were very small, and did not delay the synthesis process significantly.

[1] A newer version of CλaSH seem to address this issue, by using the names given in Haskell for the auto-generated signal names.

Chapter 6

Conclusion

This thesis starts of by introducing the reader to the functional programming paradigm and the theory behind the Viterbi algorithm. Following, we elaborate on the functional programming style, and how this can be utilized for designing hardware with CλaSH. This lays the foundation for understanding the implementation details described in the following chapter, where simulation and verification steps are included. The final chapter evaluates the design methodology based on experiences throughout the project, and presents a measure of the achieved quality.

A Viterbi decoder has been successfully implemented in the functional HDL CλaSH. It has been an experimental project, to see if the current state of the CλaSH is reliable and useful for hardware engineers. From the beginning of the project, the approach towards a complete implementation has been based on the three main units of the Viterbi decoder discussed in Section 2.7.2. Along the way, there has not been any big changes and CλaSH has been able to transform the ideas to fully-functioning SystemVerilog implementation. It has shown that the freedom of expression that Haskell offers can describe hardware in a variety of ways, without going too much into the details of the circuit. The downside however, is that it is not very flexible for describing the timing behavior, and the support for recursive functions are poor. Moreover, the generated SystemVerilog implementation for this thesis is not able to match the quality of the provided solution. Nevertheless, CλaSH has proven to be highly reliable and is an excellent choice for implementing a software model and performing

architectural exploration.

It has been proven that CλaSH is able to generate a fully-functioning Viterbi decoder from a Haskell implementation. This has been verified with several testbenches, and been concluded from a successful integration into a complete SoC. Furthermore, the resulting gate-level hardware have been compared to a solution made in SystemVerilog, which gives CλaSH a measure of quality. It should be noted that the design has been implemented by a newcomer to the functional programming style and the Viterbi algorithm.

The presented implementation is a visual example of how easily circuits can be described in a functional language. It shows a rather complex implementation, and how functional programming enables the designer to approach the problem with a step-wise design methodology. This is thanks to the interactive environment of CλaSH, which allows parts of the circuit to be implemented and simulated in isolation. This makes it easy to explore different implementation techniques without the need to create a customized testbench.

6.1 Future work

Although the presented solution has proven to be working and could potentially be used in a SoC, the implementation could be further explored and optimized. It would be interesting to see if better resource utilization could have been achieved with another design approach. Furthermore, the decoder could have been made parametrizable in order for it to work with different constraint lengths and polynomials. Different implementation methods could also have been explored, with methods such as ACS lookahead or an optimized RE method (see Section 2.7.3).

Another aspect that could have been further explored, is if it would be possible to generate a more complex testbench with CλaSH.

Appendices

Appendix A

CλaSH implementation

```
module RxViterbiDecoder where

import Language.Haskell.TH.Syntax
import CLaSH.Prelude
import CLaSH.Prelude.Explicit
import System.IO
import qualified Prelude as P
import Data.Maybe

{-# ANN topEntity
  (defTop
    { t_name = "RxViterbiDecoder"
    , t_inputs = ["softBit","validSB","clear","flush","preselect"]
    , t_outputs = ["dOut","validB","enckDiv","stateRegDOut"]
    }) #-}

polya = 15
polyb = 11
numstates = 8
numstates_SNat = d8
td = 20

type ClkSys = 'Clk "ck" 500
type ClkDiv = 'Clk "ckDiv" 1000
```

```haskell
clk :: SClock ClkSys                                                              25
clk = sclock                                                                      26
clkDiv :: SClock ClkDiv                                                           27
clkDiv = sclock                                                                   28
                                                                                  29
register6' ck start x = register' ck start (register' ck start (                  30
                        register' ck start (register' ck start (                  31
                        register' ck start (register' ck start x)))))             32
                                                                                  33
-- Init vectors                                                                   34
init_pms = (repeat 128 :: Vec 8 (Unsigned 8))                                     35
init_state_regs = (repeat (repeat 0 :: (Vec 20 (Bit))):: Vec 8 (Vec 20 (Bit)))    36
init_surviving_preds = (repeat 0 :: Vec 8 (Unsigned 8))                           37
                                                                                  38
-- Constant vectors                                                               39
states = iterate numstates_SNat (+1) 0                                            40
encInps = map (\state -> if ((state+1)>(numstates `div` 2)) then 1 else 0) states 41
pred_states1 = (map (\x -> ((2 * x) `mod` numstates)) states)                     42
pred_states2 = (map (\x -> ((2 * x+1) `mod` numstates)) states)                   43
                                                                                  44
-- Decoder functions                                                              45
topEntity :: (Signal' ClkSys (Signed 4)                                           46
             ,Signal' ClkSys (Bool)                                               47
             ,Signal' ClkSys (Bool)                                               48
             ,Signal' ClkSys (Bool)                                               49
             ,Signal' ClkSys (BitVector 3))                                       50
          -> Signal' ClkSys (Bit, Bool, Bool,(BitVector 20))                      51
topEntity (softbit,validSB,clear,flush,preselect) =                               52
          viterbi_decoder clk clkDiv softbit validSB clear flush preselect        53
                                                                                  54
                                                                                  55
viterbi_decoder :: SClock ck -> SClock ckDiv                                      56
                -> Signal' ck (Signed 4)                                          57
                -> Signal' ck (Bool)                                              58
                -> Signal' ck (Bool)                                              59
                -> Signal' ck (Bool)                                              60
                -> Signal' ck (BitVector 3)                                       61
                -> Signal' ck (Bit, Bool, Bool,(BitVector 20))                    62
viterbi_decoderck ckDiv softbit validSB clear flush preselect = out               63
  where                                                                           64
    sb1 = (regEn' ck 0 validSB softbit)                                           65
    sb2 = softbit                                                                 66
    -- Signals synchronized to ckDiv                                              67
    sb_pair        = dualFlipFlopSynchronizer ck ckDiv (0,0)                      68
                     (bundle' ck (sb1,sb2))                                       69
    flush_sync     = dualFlipFlopSynchronizer ck ckDiv False flush                70
    clear_sync     = dualFlipFlopSynchronizer ck ckDiv False clear                71
    preselect_sync = dualFlipFlopSynchronizer ck ckDiv                            72
                     (0 :: BitVector 3) preselect                                 73
```

```
    -- Flush Signals                                                              74
    flushDelayed = mux flush (register6' ck False flush) flush                    75
    flushPosEdge = isRising' ck False (register' ck False flush)                  76
    flushPosEdged = isRising' ck False (register' ck False flushPosEdge)          77
    -- Mealy machines                                                             78
    (dOut, sr_dOut)              = mealyB' ckDiv (decoder)                        79
                                     (init_pms,init_state_regs,0)                 80
                                     (sb_pair,flush_sync,clear_sync,preselect_sync) 81
    (validB_temp,validSB_vec)   = mealyB' ck (valid_ctrl)                         82
                                     (0,0,False)                                  83
                                     (clear,validSB,flushPosEdge,flushPosEdged)   84
    (dOut_flush,validB_flush)    = mealyB' ck (flush_sr)                          85
                                     (0)                                          86
                                     (flushDelayed,stateRegDOut,clear)            87
    -- Outputs                                                                    88
    validB       = (register' ck False validB_temp)                               89
    dOut_sync    = (dualFlipFlopSynchronizer ckDiv ck 0 dOut)                     90
    stateRegDOut = unsafeSynchronizer ckDiv ck sr_dOut                            91
    enckDiv      = (validSB_vec .||. flushPosEdge .||. clear)                     92
    out = mux flushDelayed                                                        93
              (bundle' ck (dOut_flush,validB_flush,enckDiv,stateRegDOut))         94
              (bundle' ck (dOut_sync,validB,enckDiv,stateRegDOut))                95
                                                                                  96
                                                                                  97
valid_ctrl (i,j,validB) (clear,validSB,flushPosEdge,flushPosEdged,flush) =        98
           ((i',j',validB'), (validB',validSB_vec))                               99
  where                                                                           100
    j' = if (validSB && (j < 2))     then (j + 1)                                 101
         else if (j >= 2 && validSB) then 1                                       102
         else if (j >= 2 || clear)    then 0                                      103
         else j                                                                   104
    validSB_vec = if (j' >= 2) then True else False                               105
    i' = if (clear || flush)                    then 0                            106
        else if (validSB_vec && (i < td + 2)) then (i + 1)                        107
        else i                                                                    108
    validB'  | (validB)        = False                                            109
             | (flush)         = False                                            110
             | (clear)         = False                                            111
             | ((i' >= td) && flushPosEdge)       = True                          112
             | ((i' >= td + 2) && validSB_vec)  = True                            113
             | ((i' >= td + 2) && flushPosEdged) = True                           114
             | otherwise       = False                                            115
                                                                                  116
                                                                                  117
flush_sr c (flush,stateRegDOut,clear) = (c',(dOut,validB))                        118
  where                                                                           119
    c' = if (not flush) then 0 else (if (flush && c < td) then (c + 1) else c)    120
    dOut = (stateRegDOut ! (td - c'))                                             121
    validB = if (clear) then False else (if (flush && c < td) then True else False)122
```

```
decoder (pms,state_regs,min_pm_state)                                   123
        (sb_pair,flush,clear,preselect)                                 124
        = ((pms',state_regs',min_pm_state'),                            125
                        (dOut,stateRegDOut))                            126
  where                                                                 127
    sb1 = (signExtend (fst sb_pair))::Signed 8                          128
    sb2 = (signExtend (snd sb_pair))::Signed 8                          129
    sb_pair_se = (sb1,sb2)                                              130
    bms = bmu sb_pair_se                                                131
    (pms',min_pm_state',surviving_preds') =                             132
        if (flush) then                                                 133
            (pms,min_pm_state,init_surviving_preds)                     134
        else if (clear)                                                 135
            ((imap (\i x -> if (i == (unpack preselect::Index 8))       136
                            then (0::Unsigned 8)                        137
                            else x) init_pms), 0, init_surviving_preds) 138
        else (acsu pms bms)                                             139
    state_regs' | (flush)     = state_regs                              140
                | (clear)     = init_state_regs                         141
                | otherwise   = smu state_regs surviving_preds'         142
    -- Outputs                                                          143
    dOut = (state_regs' !! (min_pm_state')) !! (0 :: BitVector 3)       144
    stateRegDOut = v2bv (state_regs' !! min_pm_state')                  145
                                                                        146
                                                                        147
bmu sb_pair = bms                                                       148
  where                                                                 149
    bm1s = imap (\i a -> eucDist a (encInps !! i) sb_pair) pred_states1 150
    bm2s = imap (\i a -> eucDist a (encInps !! i) sb_pair) pred_states2 151
    bms = zipWith (\x y -> (x,y)) bm1s bm2s                             152
                                                                        153
                                                                        154
eucDist pred encIn sb_pair = bm                                         155
  where                                                                 156
    sb1 = fst sb_pair                                                   157
    sb2 = snd sb_pair                                                   158
    codewordA = calc_codeword polya pred encIn                          159
    codewordB = calc_codeword polyb pred encIn                          160
    e_distA = if      ((codewordA == 1) && (sb1 < 0))  then sb1 * (-1)  161
              else if ((codewordA == 0) && (sb1 >= 0)) then sb1         162
              else 0                                                    163
    e_distB = if      ((codewordB == 1) && (sb2 < 0))  then sb2 * (-1)  164
              else if ((codewordB == 0) && (sb2 >= 0)) then sb2         165
              else 0                                                    166
    bm_temp = e_distA + e_distB                                         167
    bm = fromIntegral (bm_temp)::Unsigned 8                             168
```

```
calc_codeword poly state encIn = bit
  where
    temp = (poly .&. state)::Unsigned 4
    bit = encIn `xor` (reduceXor temp)


acsu pms bms = (pms',min_pm_state,surviving_preds')
  where
    bm1s = map fst bms
    bm2s = map snd bms
    pm1s = imap (\i x -> (+) (pms !! (pred_states1 !! i)) x) bm1s
    pm2s = imap (\i x -> (+) (pms !! (pred_states2 !! i)) x) bm2s
    pms_temp = zipWith (min) pm1s pm2s
    min_pm = fst (mapAccumL (\acc x -> if (x < acc)
                                        then (x,acc)
                                        else (acc,acc))
                                        (pms_temp !! 0) pms_temp)
    pms' = map (\x -> (x - min_pm)) pms_temp
    min_pm_state = fromMaybe 0 (findIndex (== min_pm) pms_temp)
    surviving_preds' = (izipWith (\i pm1 pm2 -> if (pm1 < pm2) then
                                                  (pred_states1 !! i)
                                                else
                                                  (pred_states2 !! i))
                                                pm1s pm2s) :: Vec 8 (Unsigned 8)


smu state_regs surviving_preds = state_regs'
  where
    state_regs' =
        imap
          (\i x -> ((state_regs !! (surviving_preds !! i)) <<+ (encInps !! i)))
          states
```

Appendix B

CλaSH testbench

```
testInput :: (Signal' ClkSys (Signed 4)
             ,Signal' ClkSys (Bool)
             ,Signal' ClkSys (Bool)
             ,Signal' ClkSys (Bool)
             ,Signal' ClkSys (BitVector 3))
testInput = (testSoftbits, testValidBits, testClear, testFlush, testPreselect)

testSoftbits = stimuliGenerator' clk z
   where
     z = $(do a <- qRunIO (readFile "sb_vec.txt")
              let x = P.map read (lines a) :: [Int]
                  y = P.map fromIntegral x :: [Signed 4]
              v y)
testValidBits = stimuliGenerator' clk (repeat True :: Vec 100 Bool)
testClear = stimuliGenerator' clk (repeat False :: Vec 100 Bool)
testFlush = stimuliGenerator' clk (repeat False :: Vec 100 Bool)
testPreselect = stimuliGenerator' clk (repeat 0 :: Vec 100 (BitVector 3))
```

```
expectedOutput :: (Signal' ClkSys (Bit, Bool, Bool, BitVector 20))            18
               -> Signal' ClkSys Bool                                          19
expectedOutput (output) = expected_dOut                                        20
  where                                                                        21
    unbundled_output = unbundle' clk output                                    22
    dOut           = (\(x,_,_,_) -> x) unbundled_output                        23
    validB         = (\(_,x,_,_) -> x) unbundled_output                        24
    enckDiv        = (\(_,_,x,_) -> x) unbundled_output                        25
    stateRegDOut = (\(_,_,_,x) -> x) unbundled_output                          26
    expected_dOut = outputVerifier' clk z dOut                                 27
       where                                                                   28
         z = encoder_input_bits                                                29
                                                                               30
encoder_input_bits = $(v [...])                                                31
```

Appendix C

Encoder Perl script

```
#!/usr/bin/perl -w                                    1
                                                      2
use strict;                                           3
use warnings;                                         4
use Getopt::Long qw(GetOptions);                      5
use Switch;                                           6
                                                      7
sub check_arguments;                                  8
my $current_state = 0;                                9
my @encInp;                                           10
my @encOut;                                           11
my $argument;                                         12
my $random = 0;                                       13
                                                      14
check_arguments();                                    15
```

```
for (my $i = 0; $i < scalar(@encInp); $i++){                                        16
  my $inp = $encInp[$i];                                                            17
                                                                                    18
  switch($current_state){                                                           19
    case 0 {if ($inp == 0) {$current_state = 0; push @encOut, "00";}                20
            else {$current_state = 4; push @encOut, "11";}}                         21
    case 1 {if ($inp == 0) {$current_state = 0; push @encOut, "11";}                22
            else {$current_state = 4; push @encOut, "00";}}                         23
    case 2 {if ($inp == 0) {$current_state = 1; push @encOut, "11";}                24
            else {$current_state = 5; push @encOut, "00";}}                         25
    case 3 {if ($inp == 0) {$current_state = 1; push @encOut, "00";}                26
            else {$current_state = 5; push @encOut, "11";}}                         27
    case 4 {if ($inp == 0) {$current_state = 2; push @encOut, "10";}                28
            else {$current_state = 6; push @encOut, "01";}}                         29
    case 5 {if ($inp == 0) {$current_state = 2; push @encOut, "01";}                30
            else {$current_state = 6; push @encOut, "10";}}                         31
    case 6 {if ($inp == 0) {$current_state = 3; push @encOut, "01";}                32
            else {$current_state = 7; push @encOut, "10";}}                         33
    case 7 {if ($inp == 0) {$current_state = 3; push @encOut, "10";}                34
            else {$current_state = 7; push @encOut, "01";}}                         35
  }                                                                                 36
}                                                                                   37
print "Encoder Input:  ", @encInp, "\n";                                            38
print "Encoder Output: ", @encOut, "\n";                                            39
my @encOutSB;                                                                       40
                                                                                    41
if ($random){                                                                       42
  for (my $i = 0; $i < scalar(@encOut); $i++){                                      43
    my $sb1 = int(rand(8));                                                         44
    my $sb2 = int(rand(8));                                                         45
    if ($encOut[$i] eq "00"){                                                       46
      push @encOutSB, "-$sb1";                                                      47
      push @encOutSB, "-$sb2";                                                      48
    }                                                                               49
    if ($encOut[$i] eq "11"){                                                       50
      push @encOutSB, "$sb1";                                                       51
      push @encOutSB, "$sb2";                                                       52
    }                                                                               53
    if ($encOut[$i] eq "01"){                                                       54
      push @encOutSB, "-$sb1";                                                      55
      push @encOutSB, "$sb2";                                                       56
    }                                                                               57
    if ($encOut[$i] eq "10"){                                                       58
      push @encOutSB, "$sb1";                                                       59
      push @encOutSB, "-$sb2";                                                      60
    }                                                                               61
  }                                                                                 62
}                                                                                   63
```

```
else{
  for (my $i = 0; $i < scalar(@encOut); $i++){
    if ($encOut[$i] eq "00"){
      push @encOutSB, "-8";
      push @encOutSB, "-8";
    }
    if ($encOut[$i] eq "11"){
      push @encOutSB, "7";
      push @encOutSB, "7";
    }
    if ($encOut[$i] eq "01"){
      push @encOutSB, "-8";
      push @encOutSB, "7";
    }
    if ($encOut[$i] eq "10"){
      push @encOutSB, "7";
      push @encOutSB, "-8";
    }
  }
}

open my $fh, '>', "sb_vec.txt" or die "Cannot open sb_vec.txt: $!";

foreach (@encOutSB)
{
  print $fh "$_\n";
}

######################## CHECK ARGUMENTS ###########################
sub check_arguments(){
  GetOptions(
    'input|i=s' => \$argument,
    'random|r' => \$random
  )or die "Invalid argument.\n";

  if ($argument){
    @encInp = split //, $argument;
  }
}
```

Bibliography

[1] *Haskell base package.* https://hackage.haskell.org/package/base.

[2] *CλaSH homepage.* http://www.clash-lang.org/.

[3] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. C? ash: structural descriptions of synchronous hardware using haskell. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 714–721. IEEE, 2010.

[4] Christiaan Pieter Rudolf Baaij. Digital circuit in cλash: functional specifications and type-directed synthesis. 2015.

[5] CPR Baaij. Cλash: From haskell to hardware. 2009.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[7] Claude Berrou, Patrick Adde, Ettiboua Angui, and Stephane Faudeil. A low complexity soft-output viterbi decoder architecture. In *Communications, 1993. ICC'93 Geneva. Technical Program, Conference Record, IEEE International Conference on*, volume 2, pages 737–740. IEEE, 1993.

[8] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

[9] Tristan Michael Bull. *Forward Error Correction and Functional Programming.* PhD thesis, University of Kansas, 2011.

[10] Chun-Yuan Chu, Yu-Chuan Huang, and An-Yeu Wu. Power efficient low latency survivor memory architecture for viterbi decoder. In *VLSI Design, Automation and Test, 2008. VLSI-DAT 2008. IEEE International Symposium on*, pages 228–231. IEEE, 2008.

[11] K Claessen and M Sheeran. A slightly revised tutorial on lava: A hardware description and verification system, 2007.

[12] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, (4):8–17, 2009.

[13] Dalia A El-Dib and Mohamed I Elmasry. Modified register-exchange viterbi decoder for low-power wireless communications. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 51(2):371–378, 2004.

[14] Gerhard Fettweis and Heinrich Meyr. Parallel viterbi algorithm implementation: breaking the acs-bottleneck. *Communications, IEEE Transactions on*, 37(8):785–790, 1989.

[15] Gerhard Fettweis and Heinrich Meyr. High-speed parallel viterbi decoding: Algorithm and vlsi-architecture. *IEEE Communications Magazine*, 29(5):46–55, 1991.

[16] G David Forney Jr. The viterbi algorithm: A personal history. *arXiv preprint cs/0504020*, 2005.

[17] Daniel D Gajski and Robert H Kuhn. Guest editors' introduction: New vlsi tools. *Computer*, 16(12):11–14, 1983.

[18] Yao Gang, Ahmet T Erdogan, and Tughrul Arslan. An efficient pre-traceback architecture for the viterbi decoder targeting wireless communication applications. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 53(9):1918–1927, 2006.

[19] Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing kansas lava. In *Implementation and Application of Functional Languages*, pages 18–35. Springer, 2009.

[20] Joachim Hagenauer and Peter Hoeher. A viterbi algorithm with soft-decision outputs and its applications. In *Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'(GLOBECOM), 1989. IEEE*, pages 1680–1686. IEEE, 1989.

[21] Jerrold A Heller and Irwin Jacobs. Viterbi decoding for satellite and space communication. *Communication Technology, IEEE Transactions on*, 19(5):835–848, 1971.

[22] Xiaopeng Jin. Implementation of the music algorithm in cλash. 2014.

[23] Olaf J Joeressen, Martin Vaupel, and Heinrich Meyr. Soft-output viterbi decoding: Vlsi implementation issues. In *Vehicular Technology Conference, 1993., 43rd IEEE*, pages 941–944. IEEE, 1993.

[24] Jun Jin Kong and Keshab K Parhi. Low-latency architectures for high-throughput rate viterbi decoders. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(6):642–651, 2004.

[25] Matthijs Kooijman. Haskell as a higher order structural hardware description language. 2009.

[26] Miran Lipovaca. *Learn you a haskell for great good!: a beginner's guide.* no starch press, 2011.

[27] Renfei Liu and Keshab K Parhi. Low-latency low-complexity architectures for viterbi decoders. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 56(10):2315–2324, 2009.

[28] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[29] Anja Niedermeier. *A fine-grained parallel dataflow-inspired architecture for streaming applications.* University of Twente, 2014.

[30] Anja Niedermeier, Rinse Wester, Kenneth Rovers, Christiaan Baaij, Jan Kuper, and Gerard Smit. Designing a dataflow processor using cλash. In *NORCHIP, 2010*, pages 1–4. IEEE, 2010.

[31] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70. IEEE, 2004.

[32] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real world haskell: Code you can believe in.* " O'Reilly Media, Inc.", 2008.

[33] I Raa. Recursive functional hardware descriptions using cλash. 2015.

[34] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(1):17–32, 2004.

[35] Bernard Sklar. *Digital communications*, volume 2. Prentice Hall NJ, 2001.

[36] Andrew J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, 1967.

[37] Rinse Wester. A transformation-based approach to hardware design using higher-order functions. 2015.

[38] Engling Yeo, Stephanie Augsburger, Wm Rhett Davis, and Borivoje Nikolic. Implementation of high throughput soft output viterbi decoders. In *Signal Processing Systems, 2002.(SIPS'02). IEEE Workshop on*, pages 146–151. IEEE, 2002.