Forward Error Correction and Functional Programming

By

Tristan M. Bull

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Committee:

Dr. Andy Gill: Chairperson

Dr. Erik Perrins

Dr. Perry Alexander

Date Defended

The Thesis Committee for Tristan M. Bull certifies
That this is the approved version of the following thesis:

Forward Error Correction and Functional Programming

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Abstract

This thesis contains a collection of work I have performed while working on Dr. Erik Perrins' Efficient Hardware Implementation of Iterative FEC Decoders project. The following topics and my contributions to those topics are included in this thesis. The first topic is a Viterbi decoder implemented in the Haskell programming language. Next, I will briefly introduce Kansas Lava, a Haskell DSL developed by my advisor, Dr. Andy Gill, and other students and staff. The goal of Kansas Lava is to generate efficient synthesizable VHDL for complex circuits. I will discuss one such circuit, a large-scale LDPC decoder implemented in Kansas Lava that has been synthesized and tested on FPGA hardware. After discussing the synthesis and simulation results of the decoder circuit, I will discuss a memory interface that was developed for use in our HFEC system. Finally, I tie these individual projects together in a discussion on the benefits of functional programming in hardware design.

Acknowledgments

First, and foremost, I would like to thank my advisor, Dr. Andy Gill. He has provided encouragement and support to me, not only on this thesis, but through my entire time as a graduate student here at the University of Kansas. I would also like to thank Dr. Erik Perrins for allowing me the opportunity to work on the HFEC project, for serving as a member of my thesis committee, and for acting as my advisor during my first semester of graduate school. Thirdly, I would like to thank Dr. Perry Alexander for serving on my thesis committee. I am also grateful to the other students and staff I have worked with on the HFEC project at ITTC. Finally, I would like to thank my family for their love and moral support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The purpose of this thesis is to show how Functional Programming can be used to benefit the implementation of Error Correction Coding (ECC) systems. This is accomplished in several chapters which discuss different contributions I have made to the HFEC project. This remainder of this document is organized as follows. Chapter 2 describes a functional implementation of a soft-decision Viterbi decoder in Haskell. This decoder was written when Kansas Lava was in it's infancy and was developed in pure Haskell using a library of matrix operations that predates those found in the sized-types [9] library. The purpose of this excercise was to gain an understanding of the algorithm as well as functional programming. Kansas Lava is introduced in chapter 3. Kansas Lava began as a teaching tool for an introductory functional programming class. However, my advisor, Dr. Andy Gill, saw untapped research potential in using Lava to assist in the implementation of complex algorithms in hardware. Over time, Kansas Lava was extended to it's current state. This chapter includes a high level overview of Kansas Lava, some small examples, and my thoughts on the viability of Kansas Lava as a tool

for FPGA developers. Chapter 4 contains a description of Low Density Parity Check (LDPC) codes and a description of an LDPC decoder. Also included is an example demonstrating how new functionality is added to Kansas Lava. Chapter 5 discusses the testing framework used to test our LDPC decoder, some issues we faced when designing the testing framework, as well as some performance results for our LDPC decoder. Finally, chapter 6 discusses how a memory interface was developed for off chip DRAM on our development board, and also how Kansas Lava could interface to this memory.

1.2 Graduate Work

In this section, I will describe the work I've done as a graduate student. I started my Master's degree in the Fall of 2008. During this semester, I taught two sections of EECS128: Foundations of Information Technology as a graduate teaching assistant. This class is a service course for non-engineering majors. During this time I also took classes in digital communications and digital signal processing to prepare for my work with Dr. Erik Perrins on the High-speed Forward Error Correction (HFEC) project.

In the Spring of 2009, I took a functional programming class with Dr. Andy Gill and a software defined radios class with Dr. Gary Minden. Also, during this semester I started working as a graduate research assistant on the HFEC project. During this time I began writing a paper on the implementation of a Viterbi decoder using the functional programming language Haskell for the International Telemetering Conference (ITC). Implementing the Viterbi algorithm in Haskell was the first step towards using Kansas Lava to implement error correcting circuits. This project is discussed further in chapter 2.

In the Fall of 2009, I took a class in network routing architectures and a class in error correction coding. Also, in October of 2009, I attended the ITC conference in Las Vegas, and presented my paper on the Viterbi algorithm. It was also during this time that I began working with Dr. Andy Gill on LDPC in Kansas Lava.

In the Spring of 2010, I took a class in implementing functional programming languages and a class in computer graphics. I continued working with Dr. Gill on the LDPC project. While prototyping the LDPC decoder, I extended different matrix representations in our Haskell Sized Types library. Some of my contributions to the LDPC decoder are discussed in chapter 4. Also, as a part of the CSDL lab, I attended the Trends in Functional Programming conference in Norman, Oklahoma.

In the Fall of 2010 and Spring of 2011, I enrolled in Master's thesis hours, and worked on my thesis while working on the HFEC project. In particular, I worked on testing our LDPC implementation on FPGA hardware. Our testing methods and the results of these tests are discussed further in 5. I also worked on implementing a reliable DRAM interface for the FPGAs on our Wildstar 5 development board. This DRAM interface is discussed in 6. During this semester a paper detailing the implementation of an LDPC decoder on an FGPA [8] was accepted to the IEEE Symposium on Field-programmable Custom Computing Machines (FCCM). Figures 4.1 and 3.1 were adapted from this paper with permission from the primary author, Dr. Andy Gill.

Chapter 2

Viterbi Decoder

This section discusses a Haskell implementation of a Viterbi decoder. The contents of this section came from a paper [2] which I wrote and presented at the International Telemetering Conference in Las Vegas in 2009. This project served to better familiarize myself with Haskell and also prepare to implement a Forward Error Correction decoder in Kansas Lava. Ultimately, we decided to implement LDPC first in Kansas Lava, but this proved to be a good introduction to Error Correction Coding and Haskell.

2.1 Background Information

Forward error correction(FEC) codes allow a receiver to correct errors in a received bit sequence by introducing parity bits into the transmitted signal. FEC codes are commonly used in wireless communications and offer benefits including the ability to provide reliable communication at very low signal-to-noise ratios. The motivation behind *this* implementation is a telemetry application, and as such, future telemetry systems will used trellis-based demodulators that will require an FEC decoder.

This chapter focuses on one type of FEC code: convolutional codes (CC). Specifically, we focus on the Viterbi algorithm, a decoding algorithm for convolutional codes [14, Ch 12]. This chapter is the first step in a larger project, to implement the CC decoder on field-programmable gate array (FPGA) hardware. Traditionally, a software model is first implemented in an imperative programming language such as MATLAB or C++. This model can later be used for equivalence checking with the hardware implementation. However, in general, none of the coding done on the software model is reuseable in the hardware implementation. The hardware implementation is usually written in a hardware description language such as VHDL[1], which is very different syntactically and semantically from an ordinary programming language.

In this paper, we discuss a Viterbi algorithm implementation in the functional programming language, Haskell. Functional programming languages offer many benefits over imperative languages like C or MATLAB. Functions in Haskell are more like mathematical expressions than a list of steps for the computer to iterate through. Functions can be passed as arguments very easily and neatly. These properties make Haskell appealing as a hardware description language as well. Chapter 3 discusses how Kansas Lava draws on these properties, and others, to effectively describe digital circuits in Haskell. Haskell has a library called QuickCheck available that simplifies and automates equivalence checking [4]. So, designing our software model in Haskell and the hardware description in Lava will allow us to easily check the hardware implementation for correctness. Additionally, future research may include methods of transforming normal Haskell functions into Lava.

[1]VHSIC (Very High Speed Integrated Circuits) hardware description language

In summary, our contributions are as follows:

- We implement the Viterbi algorithm in the Haskell programming language.
- We investigate the advantages and caveats of implementing the Viterbi algorithm in Haskell.
- We compare bit error rate (BER) performance of our implementation with a known implementation to give evidence of correctness.

2.2 Viterbi Algorithm

The Viterbi algorithm is a decoding algorithm for convolutional codes [14, Ch 12]. A simple block diagram of the system used to simulate the algorithm is presented in Figure 2.1. The simulation program first generates a block of 4096 bits and encodes the bits using a rate 1/2 (5,7) convolutional code (CC) encoder [14, Ch 12]. The encoded bits are then modulated using binary pulse-amplitude modulation (PAM), essentially just turning them into antipodal bits(+/-1). The modulated data are then sent through an additive white Gaussian noise (AWGN) channel. Finally, the noisy data are supplied as input to the soft decision Viterbi algorithm (SDVA) decoder which, by definition, takes antipodal data as input and outputs decoded bits. Further details of the simulation process are included in the Performance Validation section.

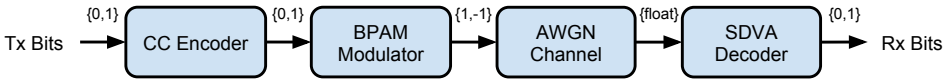


Figure 2.1. Block diagram of Viterbi algorithm simulation.

The Viterbi algorithm is implemented in two steps. First, we generate a trellis from the received bits. An example trellis is shown in Figure 2.2. At any point in

the transmission, there are four possible states. Each dot corresponds to a state. Each edge corresponds to a state transition, or branch. Each branch is assigned a branch metric, which is related to the Euclidean distance between the branch state and the last transmitted symbol.

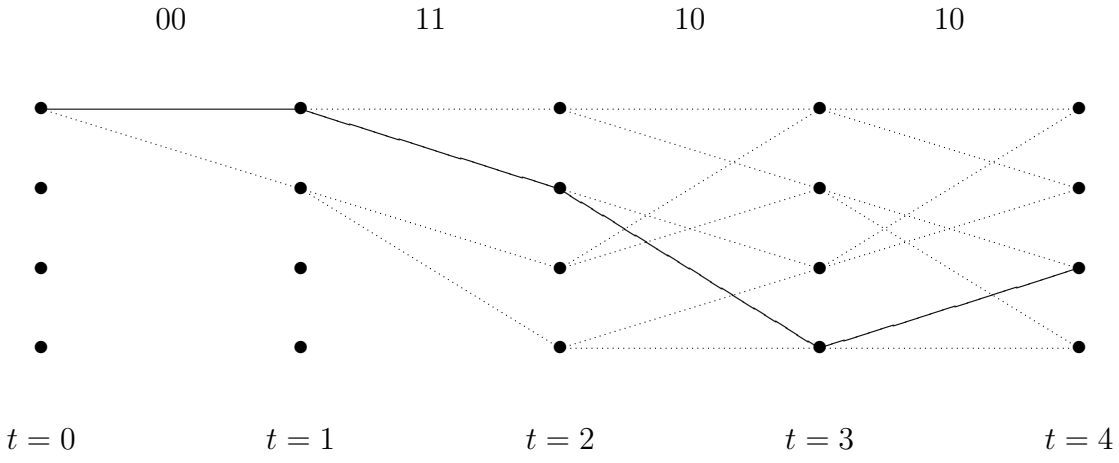


Figure 2.2. Viterbi Algorithm Trellis

The equation we used to calculate the branch metric is simple:

$$\begin{aligned} \mathrm{BM}(rr, bb) &= (r_0 - b_0)^2 + (r_1 - b_1)^2 && (2.1) \\ &= r_0^2 - 2r_0b_0 + b_0^2 + r_1^2 - 2r_1b_1 + b_1^2 \\ &= \cancel{r_0^2} - \cancel{2}r_0b_0 + \cancel{b_0^2} + \cancel{r_1^2} - \cancel{2}r_1b_1 + \cancel{b_1^2} && (2.2) \\ &\equiv r_0b_0 + r_1b_1. && (2.3) \end{aligned}$$

In (2.1), rr corresponds to the received symbol while bb corresponds to the branch symbol, which is dependent on the convolutional encoder used. In (2.2), all scalar terms (including -1) are cancelled out. We are able to simplify the calculation because we are not concerned with finding the actual value for Euclidean distance. Properties of equality state that $\mathrm{argmin}(a\times x+b, a\times y+b) == \mathrm{argmin}(x, y)$.

Assuming that there is no noise, for values of r and b between -1 and 1, the branch metric will be between -2 and 2. If $rr = bb$, then the branch metric will be 2. Likewise, if $r_0 = -b_0$ and $r_1 = -b_1$, the branch metric will be -2. Thus, the minimum Euclidean distance in (2.1) corresponds with the maximum correlation in (2.3).

Each state in the trellis contains an ordered pair of Floating point numbers, which represent the path metric (λ) for each branch entering the state:

$$\lambda_{\text{new}} = \lambda_{\text{prev}} + r_0 b_0 + r_1 b_1. \tag{2.4}$$

The path metric at any state is a cumulative value representing the path of minimum distance to that state. Remember, to minimize distance, we maximize the branch metric. Thus, λ_{prev} is simply the path metric of the previous state, which is equal to the maximum value in the ordered pair at the previous state. The second part of the path metric equation is the branch metric derived in Equation (2.3).

Once the trellis is generated, the next step is to find the survivor path. The survivor path is the path with minimal Euclidean distance (and maximum path metric). To find the survivor path, we start at the end of the transmission and find the branch with the highest path metric. We then follow that branch to its previous state. At this state, we once again pick the path with the highest path metric and follow that branch to it's previous state. This process continues until we reach the starting state. In Figure 2.2, the solid black line represents the survivor path. The branch symbols for that path are includeded at the top of the figure. The Viterbi algorithm is discussed in much further detail in [14, Ch 12].

2.3 Haskell Implementation

Traditional programming languages such as C or Java are considered imperative languages. Algorithms are expressed in an iterative fashion. Functional programming languages, like Haskell, express algorithms more as mathematical expressions than as lists of steps [16]. Normal Haskell functions and variables have no side effects. In computer science, a side effect is a change of state or an interaction with the outside world. This means that all functions are deterministic: the output is dependent only on the input. Also, variables do not change. In an imperative language a variable is a label for a block of memory or the value stored in that block of memory. In Haskell, a variable is a label for some value. This means that Haskell variables do not actually vary at all! Unsurprisingly, functions are a very important part of functional programming. Functions are values just like variables. A function can have another function as input or output. While all of this may seem unnatural at first to the C or MATLAB programmer, it allows for higher levels of abstraction and much more concise code. An example of a Haskell function for calculating the branch metric is presented below:

```
ed :: (Float,Float) -> (Float,Float) -> Float
ed (x1,y1) (x2,y2) = x1*x2 + y1*y2
```

First, the type of the function is given. Function types are listed in order from the first argument to the last argument followed by the type of the output. The Euclidean distance function, `ed`, accepts two-tuples as arguments, each containing two floating point numbers, and returns a floating point number. A tuple is a data structure similar to a list with a few exceptions. A list in Haskell can only contain elements of a single type, while a tuple can contain elements of different types. Also, a list can be of variable length, while the number of elements in a tuple must

be known. In this case, each tuple contains a pair of values. The line after the type contains the actual function. We simply multiply the `x` and `y` values from the two ordered pair inputs and sum the results. This function, `ed`, is used to calculate the branch metric discussed in the previous section.

Haskell has a built-in type called `Array`, which can be used to create indexed multi-dimensional arrays. Haskell includes functions to get/update elements of an array. However, for this project we wanted additional functionality. We wanted to be able to use many of the matrix operations such as reshape, cross product, add, and subtract that are available in MATLAB. So we created a matrix module including a new type called `Matrix` and wrote functions that performed these operations on our new data structure. For this project, we used an early implementation of the `Matrix` library that is included in the `sized-types` [9] Haskell package which is used extensively in Kansas Lava. The current `Matrix` library contains all of the functionality included here, but also much more functionality making matrices easier to work with. The following definition of `newtype Matrix` contains a "boxed up" two-dimensional `Array`:

```
newtype Matrix a = Matrix (Array (Int,Int) a)
```

`newtype` is a keyword in Haskell used to declare a new data type. In this case, we are building a new data type called `Matrix` which can contain elements of any type. The pair, `(Int,Int)` correspond to row and column indices in a two-dimensional array. In our Haskell implementation of the Viterbi algorithm, the trellis is represented as a $4 \times N$ matrix of ordered pairs, where N is the number of symbols in the received data. Each row in the matrix corresponds to one of the four states, and each column corresponds to a symbol. Each element contains the path metric data at that state. An example path metric matrix is presented

below:

```
    (Just 2.0,Nothing)     (Just 0.0,Nothing) (Just 0.0,Just (-2.0))    (Just 0.0,Just 2.0)
(Just (-2.0),Nothing)     (Just 4.0,Nothing) (Just 0.0,Just (-2.0))    (Just 0.0,Just 2.0)
    (Nothing,Nothing)  (Just (-2.0),Nothing)    (Just 2.0,Just 0.0) (Just (-2.0),Just 8.0)
    (Nothing,Nothing)  (Just (-2.0),Nothing) (Just 6.0,Just (-4.0))    (Just 2.0,Just 4.0)
```

This matrix was generating using the four symbol sequence described in Figure 2.2. Haskell contains a built-in type called `Maybe` which is essentially an optional value. A value of type `Maybe Float`, for example, either contains a value of type `Float` (called `Just Float`) or `Nothing`. The `Maybe` type is discussed further in [16]. The `Maybe` type was used in the Viterbi algorithm to denote invalid branches in the first two symbols. The first column in the matrix above corresponds to the states at $t = 1$ in the trellis in Figure 2.2. The `Nothing` values correspond to states which have zero or one branch entering them.

The `vaTrellis` function is presented below:

```
vaTrellis :: Matrix Float -> Matrix ((Maybe Float),(Maybe Float))
vaTrellis rx = (Matrix a)
  where a = array ((0,0),(3,un-1))
            $[((x,0), (bm1 x 0,Nothing)) | x <- [0,1]] ++
            [((x,0), (Nothing,Nothing)) | x <- [2,3]] ++
            [((x,1), (((bm1 x 1) `mAdd` p1 x 1),Nothing)) | x <- [0..3]] ++
            [((x,y), (((bm1 x y) `mAdd` p1 x y),(bm2 x y) `mAdd` p2 x y))
             | y <- [2..un-1], x <- [0..3]]
        bm1 x y = Just $ ed (list2Tuple (getRow bs (l2r@@(x*2,0)))) (list2Tuple (getRow rxDat y))
        bm2 x y = Just $ ed (list2Tuple (getRow bs (l2r@@(x*2+1,0)))) (list2Tuple (getRow rxDat y))
        p1 x y  = max (fst (a!(l2r@@(2*x,0) `div` 2,y-1))) (snd (a!(l2r@@(2*x,0) `div` 2,y-1)))
        p2 x y  = max (fst (a!(l2r@@(2*x+1,0) `div` 2,y-1))) (snd (a!(l2r@@(2*x+1,0) `div` 2,y-1)))
        un      = numRows rxDat
        xDat    = matReshape rx (numRows rx `div` 2) 2
        l2r     = genL2R gcc
        bs      = matMap antip $ genBS gcc
```

The `vaTrellis` function accepts a list of received symbols in the form of a $1 \times 2N$ matrix of floating point numbers, and outputs a $4 \times N$ `Matrix` of type `(Maybe Float, Maybe Float)` where N is the number of transmitted (2-bit) symbols. Haskell allows the programmer to include a `where` clause containing defintions of local functions and variables. In the code above, the `where` clause includes the majority of the functionality. The variable `a` is defined as the array inside the matrix that is returned by the function. The first line `array ((0,0),(3,un-1))` includes the array bounds. The lower bounds are `(0,0)` and the upper bounds are `(3,un-1)` where `un` is equal to the number of symbols in the transmission. The next four lines build the contents of the array. The contents of an array are stored in a list where each element is of the form `((idx,idy),v))` where `idx` is the column index, `idy` is the row index, and `v` is the value stored in that element. To build this list, we use a series of Haskell list comprehensions, which are similar to set comprehensions (or set-builder notation) in mathematics. A complete explanation of Haskell list comprehensions is beyond the scope of this paper. We encourage the reader to read the appropriate section in [16] for more information. Suffice it to say that a list comprehension is a syntactic shortcut for building a list from one or more lists. In this case, we are building a list of matrix elements from lists of row and column indices. After `a` is defined, we define a series of helper functions which are used to get the branch metric and path metric data. `bm1` and `bm2` are used to calculate the branch metric for even and odd branches respectively. The type of the `bm` functions is given below:

```
bm :: Int -> Int -> Maybe Float
```

The functions accept an x and y index for a state in the trellis and calculates the branch metric data at that state. The functions `p1` and `p2` determine which

path metric (referred to as λ_{prev} in the previous section) to use at a given state. Their type is the same as the `bm` functions. They accept an x and y index for a state and return the maximum value of the previous state. `rxDat` reshapes the input data (a single column matrix) into a two column matrix so that there is one 2-bit symbol on each row. `l2r` generates a list of left-to-right indices that are used to determine a branch's left index, given its right index. `bs` generates antipodal branch state data.

After the trellis is generated, the next step is to determine the survivor path. We wrote a function that accepts a matrix generated by the vaTrellis function and outputs decoded bits. That function is called `getSym`:

```
getSym :: Matrix (Maybe Float, Maybe Float) -> Int -> [Int]
getSym m y = flatten $ reverse $ getSym' ss m y
    where possSS = getCol m y
          ss = foo possSS
          getSym' x m y
             | y >= 0 = issueBS x (m@@(x,y)) : getSym' (st x (m@@(x,y))) m (y-1)
             | otherwise = []
```

The function `getSym` relies on three helper functions: `foo`, `st`, and `issueBS`. The type of `foo` is given below:

```
foo :: [(Maybe Float, Maybe Float)] -> Int
```

This function accepts a column from the trellis and returns the index of the row containing the highest value. This is used to determine the starting state in the last column of the trellis (the first symbol to be decoded).

The type of `st` is given below:

```
st :: Int -> (Maybe Float,Maybe Float) -> Int
```

`st` takes a current state (an `Int`) and an ordered pair containing path metric data, and outputs the next state. This function is used to move from right to left in the trellis, following the survivor path.

The type of the function `issueBS` is given below:

```
issueBS :: Int -> (Maybe Float, Maybe Float) -> [Int]
```

This function takes a current state and an ordered pair of path metric data, and outputs the branch state. The branch state is the decoded bit(s) and is dependant on the convolutional encoder used. In the `getSym` function, we use the `where` clause to define some functions and variables used in main function definition. The variable `possSS` contains the last column in the trellis. The variable `ss` determines the starting state by running `foo` on `possSS`. Finally, `getSym'` combines all of the functions discussed above to move from right to left in the trellis, generating a list of decoded bits. The type of getSym' is included below:

```
getSym' :: Int -> Matrix (Maybe Float, Maybe Float) -> Int -> [[Int]]
```

The first argument to `getSym'`, x, is the row index of the current state. The next argument is the trellis matrix, and the final argument, y, is the column index of the current state. There are two things that must be considered before the result of `getSym'` is output to the user. First, since getSym' is recursing from the beginning to the end of the data transmission, we must reverse the decoded bit sequestion. Seond, the function returns a list of lists of `Int` because `issueBS` returns the decoded bits for each symbol in the form of a list. Even though the rate 1/2 convolutional code only has one decoded bit per 2-bit symbol, other convolutional encoders may have multiple bits. Thus, we return a list of bits. To solve these two problems we call the built-in function `reverse` to reverse the bit

sequence returned by `getSym'` and we call the function `flatten` which turns a list of type `[[Int]]` into a list of type `[Int]`.

The two functions `vaTrellis` and `getSym` make up the majority of our Viterbi algorithm implementation in Haskell. By calling `vaTrellis` on a column matrix of encoded data, and then calling `getSym` on the trellis data structure, we are able to decode a sequence of convolutionally encoded bits.

2.4 Performance Validation

We used simulation to measure algorithm correctness. In the simulation, a rate 1/2 (5,7) convolutional code was used with 4096 bit codewords. We ran the simulation using input data with energy per bit to noise power spectral density ratios (E_b/N_0) of 3dB to 6dB in 0.5dB increments. For each E_b/N_0 value, we ran the simulation until at least 25,000 bit errors were recorded. Results are shown in Figure 2.3. This bit-error rate (BER) curve matches the curve presented in [12, Ch 11], giving confidence that our implementation is correct.

2.5 Lava Implementation

The final goal of this project was to implement a Viterbi decoder on a field-programmable gate array (FPGA). The original plan was to iteratively translate the Haskell implementation of the Viterbi decoder to Kansas Lava. For various reasons, I went on to work with my advisor on the LDPC decoder discussed in chapter 4. Independently, two more implementations of the SDVA arose. Another student, Brett Werling, implemented the SDVA in VHDL directly. Also, Ed Komp, an engineer working in our lab implemented the algorithm in Kansas

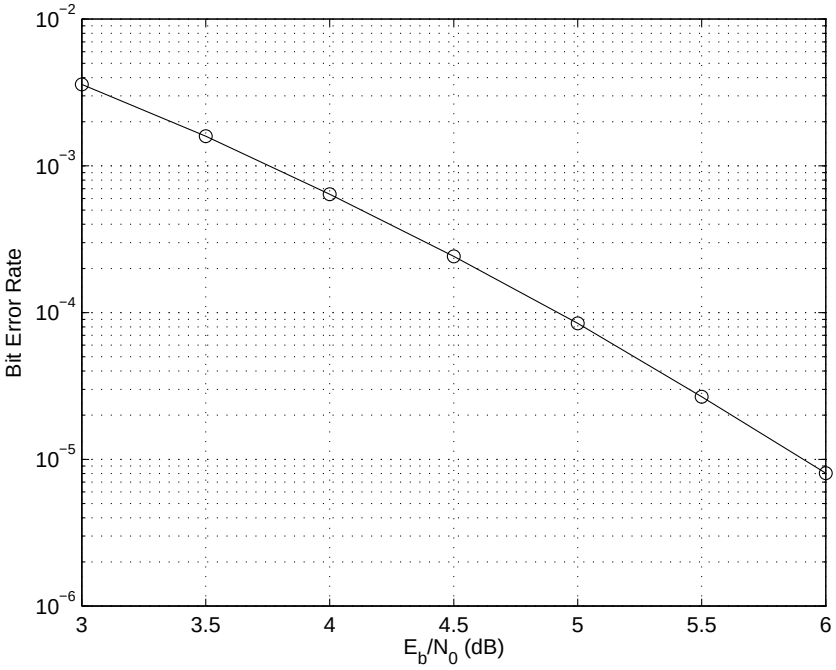


Figure 2.3. Bit error rate as a function of E_b/N_0 of Rate = 1/2 convolutional code.

Lava. Comparisons between the two showed similar performance. While the plan of iteratively stepping from a pure Haskell implementation to a Kansas Lava implementation was not used for this project, it was used to great success in the implementation of the LDPC decoder discussed in chapter 4.

Chapter 3

Kansas Lava

This section contains a high-level description of Kansas Lava and examples of the Kansas Lava design pattern. Some contents from this chapter, including figure 3.1 and the counter example, were adapted from a paper [8] in which I was a contributing author.

3.1 Description

Functional programming provides a convenient host for hardware description. There are several parallels between functional programming and hardware design [18]: the stream idiom in functional programming mirrors synchronous circuits in hardware, functional programmers and hardware designers both take a black-box approach to components in that components (or functions) are often considered in terms of their inputs and outputs, a functional program can provide a macro language on top of hardware description languages like VHDL and Verilog. There have been several uses of Haskell in providing Domain Specific Languages (DSLs) for hardware description. Historically, these include: Chalmers Lava [3]

which focuses on verification of circuits, Xilinx Lava [19] which focuses on Xilinx FPGA circuit layout, ForSyDe [17] which focuses on modeling systems and the connections between systems, and Hydra [15] which was one of the first attempts at modeling hardware using functional programming. Kansas Lava is a flavor of lava which was designed with the synthesis of complex circuits in mind and uses static types to describe communication between components.

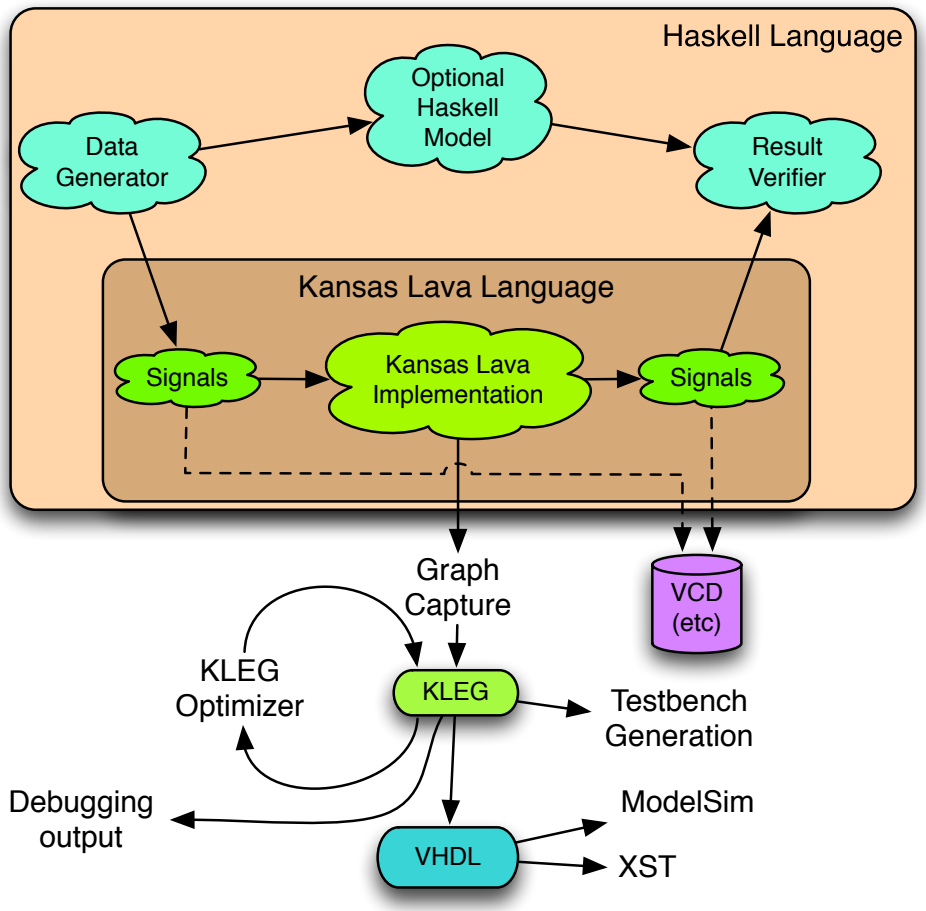


Figure 3.1. KansasLava Architecture

Kansas Lava provides a dual shallow and deep embedding [10] which allows for software simulation (in GHCi) and netlist generation through a technique called reification [7]. Figure 3.1 describes the common Kansas Lava design paradigm.

This paradigm is as follows. First, A Haskell model of the desired algorithm is developed. This model can later be used to verify correctness of the Kansas Lava implementation. Additionally, an iterative process can be used to develop versions of the algorithm that are closer and closer to the final synthesizable version. In fact, this is exactly how the LDPC decoder described in chapter 4 was developed. There were actually 17 versions of the decoder, from a straightforward pure Haskell model to the final, efficient, hardware implementation. After the Kansas Lava implementation and optional Haskell model have been developed, a data generator can be developed in Haskell to provide test data. Kansas Lava provides built-in functions to generate `Signal`s (streams) of data from the common Haskell `List` type. These signals can be provided as input to the Kansas Lava implementation for a cycle-accurate simulation inside GHCi. The output signals of the simulation can then be compared to the output of the Haskell model to verify correctness. An example of one such testing infrastrucutre is discussed further in chapter 4. Alternatively, the input and output signals can be written to a VCD file for waveform analysis. After the developer is satisfied with the results of the software simulation, they may choose to generate a netlist output. This netlist can then be used to generate VHDL code that can be simulated in a tool like ModelSim or synthesized to hardware.

3.2 Examples

Kansas Lava contains a type class, `Signal`, which supports combinitorial and sequential circuits through types called `Comb` and `Seq` respectively. An additional type, called `HandShake`. provides a reliable mechanism for communication between sub-components of a larger circuit. First, we will explore the simplest of these

three types, `Comb`.

```
halfAdder :: (Comb ~ sig) => sig Bool -> sig Bool -> (sig Bool, sig Bool)
halfAdder x y = (s, c)
    where s = (x `xor2` y)
          c = (x `and2` y)
```

Above is a simple half-adder circuit defined in Kansas Lava. The circuit is defined as a Haskell function, and the first line of any Haskell function lists the type of the function. In this case, `(Comb ~ sig)` indicates that `sig` is an alias for a `Comb`inatorial value. The portion of the type signature after the `=>` indicates that the circuit accepts two `Bool`ean `Comb`inatorial inputs and outputs a tuple containing two `Bool`ean `Comb`inatorial outputs. Next, we can use two half-adders to construct a full-adder.

```
fullAdder :: (Comb ~ sig) => sig Bool -> sig Bool -> sig Bool -> (sig Bool, sig Bool)
fullAdder x y c_in = (s_out, c_temp1 `or2` c_temp2)
    where (s_temp, c_temp1) = halfAdder x y
          (s_out,  c_temp2) = halfAdder s_temp c_in
```

The `fullAdder` function above uses `halfAdder`, as defined previously, to build a full-adder. In the `fullAdder` function, we've added a carry input called `c_in`. This is indicated in the additional `sig Bool` on the first line, and by `c_in` on the second line. Next, Haskell can be used to construct a truth table for our `fullAdder` circuit.

```
> [ (x,y,cin,fullAdder x y cin)
  | x <- [low,high], y <- [low, high], cin <- [low, high] ]
[(0,0,0,(0,0)), (0,0,1,(1,0)), (0,1,0,(1,0)),
 (0,1,1,(0,1)), (1,0,0,(1,0)), (1,0,1,(0,1)),
 (1,1,0,(0,1)), (1,1,1,(1,1))]
```

The first two lines above is the input to the GHCi interpreter. Both of these would ordinarily be entered on a single line, but they have been split to better fit on the page of this dicument. These two lines form a list comprehension. As mentioned in chapter 2, list comprehensions are a common type of syntactic sugar used to construct lists in Haskell. A full explanation is included in [16]. The syntax details are not important, but we are constructing a list containing all possible combinations of inputs to the `fullAdder` circuits and listing those inputs with the circuit outputs. In other words, we are constructing a truth table. The output of this function (the truth table) is included on the three lines following the input to the interpreter. This is a simple example, but it is important to note that this process of constructing large systems (programs) from small blocks (functions) is pervasive in functional programming. It is also worth noting that a Kansas Lava circuit is a Haskell function, and therefore can use many of the abstractions available to standard Haskell functions.

The next example is a simple sequential circuit:

```
counter :: (Rep a, Num a, Clock clk, CSeq clk ~ sig) => sig Bool -> sig Bool -> sig a
counter restart inc = loop
    where reg  = register 0 loop
          reg' = mux2 restart (0,reg)
          loop = mux2 inc (reg' + 1, reg')
```

In this case, the first line of the code block indicates that the output type is both `Rep`resentable in hardware and `Num`eric. `Rep`resentable types include most fixed length integer types (signed and unsigned), boolean values, and tuples containing these types. Second, we indicate that this is a `Se`quential circuit with `CSeq clk ~ sig`. We have two sequential Boolean inputs and a single sequential output. Instead of generating a truth table, we can execute the circuit from GHCi with some sample input to see if we get the expected output.

```
> counter low (toSeq (cycle [True,False])) :: Seq U4
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 ...
```

`low` is a function which outputs a Boolean `Signal` that is always false (or low). Here, we are setting the `restart` input to `low` which will cause the counter to never restart, unless it overflows. `toSeq` is a function which takes a list of `Rep`resentable values, and outputs a stream of values. This function can be thought of as a bridge between Haskell types and Kansas Lava types. In this case, we are simple cycling the `inc` input between `True` and `False`. The `inc` input determines whether the output increments on *this* clock cycle. Finally, we must define a type for our output. In this case, we choose a `Seq`uential `U`nsigned output that is `4` bits wide. The output, included on the second line, indicates that the counter is incremented every other clock cycle. This matches the expected output.

Now that the circuit has been tested in software, synthesizable VHDL code can be generated. This code can then be simulated in ModelSim or synthesized for running on an FPGA. Given that the system for generating a netlist is always available, there is another issue that must be dealt with. In a Haskell function, arguments are unnamed and ordered. Conversely in a VHDL circuit, ports are named and unnordered. We could generate names automatically, but even small code changes can cause problems with this approach. Additionally, the association between a Kansas Lava function argument and a VHDL port becomes unclear. Kansas Lava has employed a solution in the form of a functional programming construct called a monad. A monad is a construct used to express computation. The details of monads are beyond the scope of this document and are discussed further in [16]. In this case, the `Fabric` monad can be thought of as a data structure that is used to describe the interfaces to a Kansas Lava circuit.

There are several functions defined which provide an interface to `Fabric`. A subset of these are included below:

```
inStdLogic :: String -> Fabric (Seq Bool)
inStdLogicVector :: (Size x) => String -> Fabric (Seq (Unsigned x))
outStdLogic :: String -> Seq Bool -> Fabric ()
outStdLogicVector :: (Size x) => String -> Seq (Unsigned x) -> Fabric ()
```

`inStdLogic` and `inStdLogicVector` each name an input, while `outStdLogic` and `outStdLogicVector` each name an output and return a `Fabric`. This interface can be used to build a `Fabric` for the `counter` example discussed above.

```
counterFabric :: Fabric ()
counterFabric = do
  restart <- inStdLogic "restart"
  inc <- inStdLogic "inc"
  let cir = counter rst inc :: Seq U4
  outStdLogicVector "count" ((coerce) cir)
```

`counterFabric` is a function which builds a `Fabric` for the `counter` circuit. In the code above, `inStdLogic` is used to name two binary inputs "restart" and "inc". Next, these named inputs are provided to the `counter` function. Lastly, `outStdLogicVector` is used to name the output of our circuit "count". The `coerce` function is simply used to coerce from the `Unsigned` type to the `StdLogicVector` type. This step is necessary because Kansas Lava circuits can only have inputs of type `StdLogicVector` or `StdLogic`. Next, VHDL code can be generated from the `Fabric`.

We will use two functions `reifyFabric` and `writeVhdlCircuit` to generate VHDL code for our circuit. The type for `reifyFabric` and `writeVhdlCircuit` are as follows:

```
reifyFabric :: Fabric () -> IO Circuit
writeVhdlCircuit :: [String] -> String -> FilePath -> Circuit -> IO ()
```

`reifyFabric` takes a `Fabric` and returns an `IO Circuit`. `IO` is another monad which, unsurprisingly, provides I/O to Haskell programs. For our purposes, `IO` can be thought of as a box in which we are storing a `Circuit`. The `Circuit` is a datastructure which is essentially a netlist for our circuit. `writeVhdlCircuit` has four arguments. The first is a list of optional arguments, which can be ignored for now. The second is a name for the generated VHDL `entity`. The third is a file name for the output, and the final argument is the `Circuit` for which we want to generate VHDL.

```
main = do
  k <- reifyFabric counterFabric
  writeVhdlCircuit [] "counter" "counter.vhd" k
```

The code sample above generates VHDL for the `counter` function. At this point, the process used to go from a Kansas Lava function to VHDL should be clear. First, we create a wrapper for our function called a `Fabric`. Next, we use reification to generate a netlist for our function called a `Circuit`. Finally, we can generate VHDL code from the `Circuit`.

```
...
entity counter is
  port(rst : in std_logic;
       clk : in std_logic;
       clk_en : in std_logic;
       inc : in std_logic;
       restart : in std_logic;
       count : out std_logic_vector(3 downto 0));
end entity counter;
architecture str of counter is
  signal sig_2_o0 : std_logic_vector(3 downto 0);
...
  signal sig_6_o0 : std_logic_vector(3 downto 0) := "0000";
begin
  sig_2_o0 <= sig_3_o0;
  sig_3_o0 <= sig_4_o0 when (inc = '1') else
```

```
              sig_5_o0;
  sig_4_o0 <= std_logic_vector((unsigned(sig_5_o0) + "0001"));
  sig_5_o0 <= "0000" when (restart = '1') else
              sig_6_o0;
  proc14 : process(rst,clk) is
  begin
    if rst = '1' then
      sig_6_o0 <= "0000";
    elsif rising_edge(clk) then
      if (clk_en = '1') then
        sig_6_o0 <= sig_3_o0;
      end if;
    end if;
  end process proc14;
  count <= sig_2_o0;
end architecture str;
```

VHDL for our `counter` example is included above. It is apparent that there are some redundnant assignments in the generated code. A future version of Kansas Lava will optimize away these redundant statements. In our experiences, the synthesis tools are quite good at catching these cases and optimizing them away, so it has not been a high priority.

3.3 Retrospective

In my opinion, Kansas Lava can benefit hardware developers in a number of ways:

- The type system: The Haskell type system tends to catch a lot of type errors which can make it past VHDL compilers. Additionally, types can be easily constructed which describe a control structure or communication protocol. VHDL does allow for user defined types, but the Haskell type system is much more powerful.

- The shallow embedding: Being able to test circuits using the GHCi interpreter and being able to construct test data using normal Haskell functions has been an invaluable time saving tool to our design efforts.

- The deep embedding: One aspect of Kansas Lava which was not discussed in this chapter, but which was used in optimizing the LDPC decoder discussed in chapter 4 is that the netlist can be analyzed to determine the critical path of a circuit which is causing the greatest delay, and lowering the maximum clock rate for the circuit.

- The benefits of a high-level language: It is common knowledge in computer science that high-level programming languages generally make the programmer's job easier, though sometimes at the cost of performance. The same can be said of Kansas Lava and traditional hardware description languages like VHDL. However, just as is the case with Haskell, steps can be taken to mitigate the associated performance loss. In fact, Chapter 4 discusses one example of how Kansas Lava has already been used to meet the real-world performance requirements of a complex algorithm. Also discussed in this chapter is how an inefficient and naive implementation can be optimized in Kansas Lava.

Naturally, there are still a few issues with hardware development:

- Haskell as a host: Currently, the developer must be an exptert in Haskell development to efficiently use Kansas Lava. Convincing developers in industry to learn a programming paradigm that is fundamentally different from anything they've probably seen is a difficult proposition.

- VHDL as an intermediary: This subject is discussed further at the end of

chapter 6. It is easy to instantiate a Kansas Lava module inside a larger VHDL design, but the reverse is not true. There is still a fair amount of work to be done to allow a VHDL entity to be instantiated inside a Kansas Lava circuit.

- The penalty of abstraction: The metric example demonstrated how a user could very easily write an inefficient implementation of a seemingly straightforward algorithm. Additionally, the Kansas Lava team is tasked with ensuring that primitives behave the same way in Kansas Lava as they do when realized in hardware. Andrew Farmer discusses some work he has done to verify consistency between the shallow and deep embedding in [5] and he has extended these efforts since this publication. This is still a non-trivial task considering that even HDL simulators like ModelSim sometimes produce different results than hardware.

Chapter 4

Low-Density Parity Check Codes

This chapter contains a description of LDPC codes and the LDPC decoder algorithm we are using for the HFEC project. The implementation of the decoder was discussed in a paper [8] in which I was a contributing author. The equations used in this chapter were adapted from this paper. I was involved in the initial prototyping of the decoder in Haskell as well as the development of some of the Kansas Lava primitives. In the final section of this chapter, I discuss a portion of the decoder, and contrast a naive Kansas Lava implementation, an optimized Kansas Lava implementation, and a native VHDL implementation.

4.1 Description

Low-Density Parity Check (LDPC) codes are a type of forward error correcting code that were invented by Robert G. Gallager in the 1960's [6]. LDPC codes belong to a subset of error correcting codes called block codes. Block codes are distinct from the convolutional codes discussed in chapter 2. Block codes are generally discussed in terms of their message length k, and their codeword length

n. A message consists of k information symbols (usually bits), while a codeword consists of n symbols. The ratio k/n is used to describe the *rate* of the code. An LDPC code consists of two matrices: a generator matrix and a parity check matrix. The generator matrix contains k rows and n columns of bits, and is used to generate a length-n codeword from a length-k message through a matrix-vector multiplication. The parity check matrix contains $n - k$ rows and n columns and is used to check the validity of a codeword through a matrix-vector multiplication. However, these matrices are generally quite large, and until recently, LDPC codes were too computationally complex to use in real-time systems. Because of advances in hardware and advances in LDPC code design, LDPC codes can now be used in real-time systems.

4.2 Decoding LDPC Codes

We used an iterative log-likelihood decoder for our implementation. The iterative decoding algorithm accepts a noisy codeword as input and provides a decoded message as output. This iterative decoding algorithm can be described by the following three equations [14]:

For each (m, n) where $A(m, n) = 1$:

$$\eta_{m,n}^{[l]} = -2 \tanh^{-1}\left(\prod_{j \in \mathcal{N}_{m,n}} \tanh\left(\frac{\eta_{m,j}^{[l-1]} - \lambda_j^{[l-1]}}{2}\right)\right)$$

For each n:

$$\lambda_n^{[l]} = \lambda_n^{[0]} + \sum_{m \in \mathcal{M}_n} \eta_{m,n}^{[l]}$$

For each n:

$$\hat{c}_n^{[l]} = 1, \text{if} \lambda_n^{[l]} > 0, \text{otherwise} = 0.$$

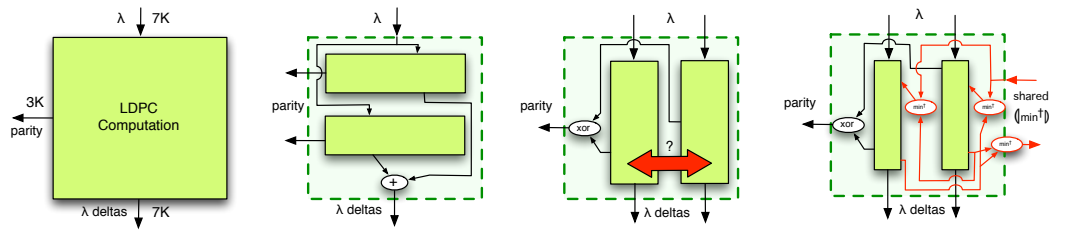
A above refers to the parity check matrix mentioned in the previous section. η refers to a matrix that contains the same number of elements as the parity

check matrix. However, each element is a soft value. η is first initialized to 0's where $A(m,n) = 1$. λ can be thought of a vector of probabilities. Each element in λ corresponds to a symbol in the codeword and is initialized to the recieved codeword.

For each decode iteration, η is updated according to the equation above. Next, λ is updated according to the equation above. Finally, $\hat{c}$, a vector of bits, is obtained by sending λ through a sign function which operates as described above. At the end of each iteration, a parity check is performed: if $A * \hat{c} = 0$, a valid codeword has been found. Else, we increment the iteration counter and repeat. The algorithm continues until either a valid codeword is found, or a maximum number of iterations has been reached. The maximum number of iterations is known before decoding begins. In our case, we used 200 iterations as our maximum. So, if after 200 iterations, a valid codeword has not been found, a decoding failure is declared, and we output the message packet with errors.

4.3 Our Decoder Design

In the case of our HFEC project, we are using an LDPC code that is a part of a family of codes called AR4JA codes. These codes were developed at NASA's JPL [1]. These codes have a few properties which ease implementation. First, the codes are considered *regular* codes. This means that all of the information bits in a codeword are stored in-order at the beginning of the codeword. Thus, a codeword consists of n information bits (the message) followed by m parity bits. This makes the encoder implementation slightly easier because the information sequence can just be stored in a register. Only the parity bits have to be calculated. This also simplifies the decoder implementation slightly because, given a valid codeword,



(a) Basic LDPC Fabric (b) Fabric Split Horizontally (c) Fabric Split Vertically (d) Refined Vertical Split

Figure 4.1. Abstract Computational Fabrics

the original information sequence can be obtained by simply truncating off the parity bits. No matrix operations are necessary. Next, and most importantly, these codes are called *block-circulant* codes. This means that the parity check and generator matrices are made up of smaller submatrices, which we refer to as *cells*. Each *cell* consists of an all-zeros matrix, an identity matrix, or a cyclically-shifted identity matrix. This *block-circulant* property will contribute greatly to our implementation of an LDPC decoder and is discussed further below.

Our LDPC code has a message size of 4096 bits and a codeword size of 6144 bits. The code actually has an extra 1024 parity bits which we do not transmit, resulting in a generator matrix size of 4096×7168 bits and a parity-check matrix size of 3096×7168. Given that the η matrix contains the same number of elements as the parity check matrix, it goes without saying that a naive decoder implementation storing the entire η matrix would never fit on an FPGA. Luckily, the matrices are sparse, so we could use a map-like data structure to store only the non-zero values. However, this still does not take advantage of the massively parallel nature of FPGAs. Thus, we take a divide and conquer approach to the decoding algorithm.

Figure 4.1(a) illustrates the LDPC decoder as a fabric. Our fabric accepts λ (the noisy input sequence), and outputs parity bits and processed λ-deltas. We

output delta values, instead of the updated λ because the original λ is stored when a packet is received.

It is straightforward to see how control logic could be added around this block to both compute $\hat{c}$, and provide the λ-deltas as feedback if the parity check fails. The most complex part of the decoder implementation is the splitting of the LDPC fabric. Figure 4.1(b) illustrates how the fabric can be split horizontally by initially loading the λ values into each block in a column and combining λ-delta values after computation has completed. Splitting vertically is not as straightforward, because sharing occurs bidirectionally. I was involved in the early prototyping of this design in Haskell, and was a part of the discussions during the transition to a fully synthesizable Kansas Lava implementation. My advisor, Dr. Andy Gill, was the primary developer of our final Kansas Lava implementation. The remaining details of the final design are discussed further in [8].

4.4 Metric Function

The tanh and $\tanh^{-1}$ functions used in the calculation of λ are very expensive operations in hardware. Either a large lookup table or a linear approximation must be used. Additionally, the algorithm described above requires that the input codeword first be scaled by a factor that is proportional to the signal-to-noise ratio of the input. Thus, a hardware implementation must also contain a signal-to-noise estimator. Luckily, there is a simplified implementation which uses only comparators and basic arithmetic operators:

For each (m, n) where $A(m, n) = 1$:

$$\eta_{m,n}^{[l]} = -\frac{3}{4}\left(\!\!\left(\min^{\dagger}\right)\!\!\right)_{j \in \mathcal{N}_{m,n}} \left(\eta_{m,j}^{[l-1]} - \lambda_{j}^{[l-1]}\right)$$

$$\min^{\dagger}(x, y) = sign(x) * sign(y) * min(|x|, |y|)$$

The notation above indicates that the values on the right side of $(\!|\ |\!)$ should be combined in the same was as $\sum$ or $\prod$. Thus, $\sum$ could be written as $(\!|+|\!)$. In our case, the function $\min^{\dagger}$ is used to combine values. This notation is borrowed from the notation for a catamorphism in functional programming [13].

We have the option of implementing the above function either in Kansas Lava or directly in VHDL. Kansas Lava has the advantage that the definition will be far more concise than a direct VHDL implementation. Additionally, since the circuit is a Haskell function, it can be tested in GHCi directly. One advantage of using VHDL for implementation is that the final implementation may be slightly more efficient since workingg at a lower level allows for greater control of implementation details, just as in software programming. Both implementations will be presented and compared in this section.

```
metric :: (Size ix, Integral ix, Signed ix ~ a, Clock clk, CSeq clk ~ sig) =>
          sig a -> sig a -> sig a
metric x y = (sign x) * (sign y) * (min (abs x) (abs y))
    where sign x = mux2 (x .>. 0) (1, (-1))
```

The Kansas Lava code is presented above. As with previous Kansas Lava examples, the first statement describes the type. In this case, we are stating that the two inputs are `Sequential` `Signed` signals. `Size ix` indicates that the signal has a fixed width, and `Integral ix` indicates that the signal is an integer type. The next line contains the value returned by this function. This definition is almost identical to the definition presened above. The `sign` function is not present

in Kansas Lava (or Haskell for that matter), so we had to define it ourselves. In this case, we used `mux2`, a built-in Kansas Lava function which is a two-input multiplexer. The first argument is a Boolean function and the second argument is a tuple containing the output when the first argument evaluates to true followed by the output when the first argument evaluates to false. This is analogous to an if-then-else statement in programming. In this case, `sign` is a function which returns 1 in the case of a positive number and -1 in the case of a negative number. Next, the function can be tested in GHCi.

```
> let a = toSeq [(-10)..10] :: Seq S8
> let b = toSeq [0..20] :: Seq S8
> metric a b
0 -1 -2 -3 -4 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 ...
```

The first line generates a incremental `Seq`uence of `S`igned `8`-bit values from -10 to 10. The second line generates a `Seq`uence of values with the same type from 0 to 20. The next line executes the `metric` function using `a` and `b` as inputs. We can analyze this output mathematically to see if it matches our expected output, and in this case it does. Next, we can generate VHDL code for this circuit. However, during VHDL generation, we found that the `min` and `abs` functions did not have deep embedding definitions in Kansas Lava. Here is an example of where the deep and shallow embeddings diverge.

```
instance (Show a, Rep a, Num a) => Num (CSeq c a) where
    (+) = liftS2 (+)
    (-) = liftS2 (-)
    (*) = liftS2 (*)
    negate = liftS1 negate
    abs = liftS1 abs
    signum = liftS1 signum
```

```
instance (Ord a, Rep a) => Ord (CSeq c a) where
    max = liftS2 max
    min = liftS2 min
```

The shallow embedding functionality is described above. The functions `liftS1` and `liftS2` are used to *lift* an ordinary Haskell function into the Lava world. However, this method only creates a shallow embedding for the function, as there is no way to automatically derive the functionality in VHDL. We must add the deep embedding functionality ourselves. In this case, that is exactly what we've done:

```
genInst :: M.Map Unique (Entity Unique) -> Unique -> Entity Unique -> [Decl]
genInst _ i (Entity (Prim "min") [("o0",_)] [("i0",xTy,x),("i1",yTy,y)])
        = [NetAssign (sigName "o0" i)
                     (ExprCond cond
                      (toStdLogicExpr xTy x)
                      (toStdLogicExpr yTy y))]
  where cond = ExprBinary LessThan (toTypedExpr xTy x) (toTypedExpr yTy y)

genInst _ i (Entity (Prim "abs") [("o0",_)] [("i0", xTy, x)])
    = [NetAssign (sigName "o0" i)
                     (ExprCond cond
                      (toStdLogicExpr xTy (ExprUnary Neg (toTypedExpr xTy x)))
                      (toStdLogicExpr xTy x))]
  where cond = ExprBinary LessThan (toTypedExpr xTy x) (toTypedExpr xTy (0 :: Integer))
```

`genInst` is a function whose type is given in the first line above. The underlying details are complex, but it is essentially used to generate a netlist for Kansas Lava primitives. Thus, any primitive which we wish to generate VHDL for must have a definition inside genInst. The above examples are similar in that they both rely on a binary expression to determine their output. The difference is that `min` takes 2 arguments, while `abs` only takes 1 argument. The generated VHDL code is included below:

```
...
entity metric is
  port(x : in std_logic_vector(7 downto 0);
       y : in std_logic_vector(7 downto 0);
       z : out std_logic_vector(7 downto 0));
end entity metric;
architecture str of metric is
  signal sig_2_o0 : std_logic_vector(7 downto 0);
...
begin
  sig_2_o0 <= sig_3_o0;
  inst3 : entity lava_signed_mul
  generic map (width => 8)
  port map (i0 => sig_4_o0,i1 => sig_15_o0,o0 => sig_3_o0);
  sig_15_o0 <= sig_16_o0 when (signed(sig_16_o0) < signed(sig_17_o0)) else
               sig_17_o0;
  sig_17_o0 <= std_logic_vector(-(signed(sig_12_o0))) when (signed(sig_12_o0) < "00000000") else
               sig_12_o0;
  sig_16_o0 <= std_logic_vector(-(signed(sig_7_o0))) when (signed(sig_7_o0) < "00000000") else
               sig_7_o0;
  inst4 : entity lava_signed_mul
  generic map (width => 8)
  port map (i0 => sig_5_o0,i1 => sig_10_o0,o0 => sig_4_o0);
  sig_10_o0 <= "00000001" when (sig_11_o0 = '1') else
               sig_14_o0;
  sig_14_o0 <= std_logic_vector(-("00000001"));
  sig_11_o0 <= '1' when (signed(sig_12_o0) > "00000000") else
               '0';
  sig_12_o0 <= sig_13_o0;
  sig_13_o0 <= y(7 downto 0);
  sig_5_o0 <= "00000001" when (sig_6_o0 = '1') else
              sig_9_o0;
  sig_9_o0 <= std_logic_vector(-("00000001"));
  sig_6_o0 <= '1' when (signed(sig_7_o0) > "00000000") else
              '0';
  sig_7_o0 <= sig_8_o0;
  sig_8_o0 <= x(7 downto 0);
  z <= sig_2_o0;
end architecture str;
```

Again, we find that there are some redundant assignments in this code. As mentioned in chapter 3, we intend to optimize these out of the generated code

eventually. However, the synthesis tools seem to do a sufficient job of optimizing them out on their own.

The above VHDL code synthesizes and provides the expected output. However, there is some inefficiency to the code. Most notably, there are two multiplications which are only multiplying by either 1 or -1. So we can easily optimize out these multiplies by using a negate instead when appropriate.

```
metric' :: (Size ix, Integral ix, Signed ix ~ a, Clock clk, CSeq clk ~ sig) =>
           sig a -> sig a -> sig a
metric' x y =  mux2 flipSign (-ans,ans)
    where flipSign = (isPositive x) `xor2` (isPositive y)
          ans      = (min (abs x) (abs y))
          sign x   = mux2 (x .>. 0) (1, (-1))
```

We will call our optimized circuit `metric'`. To determine whether we need to flip the sign of the output, we've added a function called `flipSign` which is a Boolean function that uses a built-in Kansas Lava function called `isPositive` which takes a `Signed Signal` as an argument and outputs `high` when the `Signal` is positive and `low` when the `Signal` is `low`. We can simply `xor` the results of the `isPositive` function to determine whether we need to flip the output sign. Finally, we can use `mux2` again to simply output `ans` or the negation of `ans`.

The VHDL resulting from `metric'` is included below:

```
...
entity metric is
  port(x : in std_logic_vector(7 downto 0);
       y : in std_logic_vector(7 downto 0);
       z : out std_logic_vector(7 downto 0));
end entity metric;
architecture str of metric is
  signal sig_2_o0 : std_logic_vector(7 downto 0);
...
begin
  sig_2_o0 <= sig_3_o0;
```

```
  sig_3_o0 <= sig_13_o0 when (sig_4_o0 = '1') else
              sig_14_o0;
  sig_13_o0 <= std_logic_vector(-(signed(sig_14_o0)));
  sig_14_o0 <= sig_15_o0 when (signed(sig_15_o0) < signed(sig_16_o0)) else
               sig_16_o0;
  sig_16_o0 <= std_logic_vector(-(signed(sig_11_o0))) when (signed(sig_11_o0) < "00000000") else
               sig_11_o0;
  sig_15_o0 <= std_logic_vector(-(signed(sig_7_o0))) when (signed(sig_7_o0) < "00000000") else
               sig_7_o0;
  sig_4_o0 <= (sig_5_o0 xor sig_9_o0);
  sig_9_o0 <= not(sig_10_o0);
  sig_10_o0 <= sig_11_o0(7);
  sig_11_o0 <= sig_12_o0;
  sig_12_o0 <= y(7 downto 0);
  sig_5_o0 <= not(sig_6_o0);
  sig_6_o0 <= sig_7_o0(7);
  sig_7_o0 <= sig_8_o0;
  sig_8_o0 <= x(7 downto 0);
  z <= sig_2_o0;
end architecture str;
```

Analyzing the VHDL code, we see that there are no longer any instances of the `lava_signed_mul` entity, which is our multiplication entity for the `Signed` signal type. For the sake of comparison, a version of the metric function written directly in VHDL is included in the Appendix at the end of this document. This is the version of the function we used in our initial design of the LDPC algorithm, and provides results consistent with the optimized Kansas Lava design. In the final part of this chapter, we will synthesize these the two Kansas Lava implementations and compare their relative utilization of the on the FPGA.

Table 4.1. Device Utilization of 1024 Serially Conected Metric Blocks

	# LUTs Used	Max Clock Frequency
Optimized	33837	187MHz
Original	59950	130MHz

Table 4.1 compares the device utilization of 1024 Metric blocks which are connected serially. This configuration was used to provide a good indicator of the relative performance of the metric function without letting the rest of the LDPC circuit obfuscate the results. These tests were performed on a Xilinx Virtex 5vlx110tff1738-1 FPGA. In our tests, the circuit using the optimized algorithm used about 56% of the number of LUTs as the original circuit. Additionally, the maximum frequency for the optimized circuit was 57MHz higher (a 44% improvement). This improvement in clock frequency is a result of a shorter path from beginning to the end of the circuit. This relatively simple example illustrates the care that must be taken when implementing even simple hardware circuits. This issue was highlighted at the end of chapter 3. Though Kansas Lava may make hardware development easier for functional programmers, they must still be aware of the caveats of hardware design.

Chapter 5

Testing Infrastructure

This chapter contains a description of the testing framework we used to test the LDPC decoder on our development board. This testing framework was developed by my advisor and myself. I was highly involved in the optimization efforts of the testing framework. I also used the testing framework to test the decoder and ultimately generate the results in figure 5.6.

5.1 LDPC Packet Format

As discussed in the previous chapter, the LDPC decoder requires, as input, a maximum number of decoding iterations to attempt. This value could be statically coded into the decoder. However, we wanted to have the ability to tune this aspect of the decoder during operation. To provide this information to the LDPC decoder, we could either add an additional input to the decoder or we could include the maximum iterations as a header before sending our noisy codeword to the decoder. In the end, we elected to use the header approach as there were other benefits which will be discussed further in this section.

```
data Packet d = Packet
        { packet_id    :: Word8
        , iter_id      :: Word8
        , packet_count :: Word32
        , payload :: [d]
        }
```

Above is the Haskell data type definition for our LDPC Packet. The packet contains a 48-bit header followed by a payload. The header is very straightfowrad. The `packet_id` is an 8-bit value which indicates the type of packet. This is important as it describes the contents of the payload. Table 5.1 lists the `packet_id`s and the associated payload. For packets containing noisy data, the `iter_id` indicates the maximum number of iterations that should be attempted by the decoder. In the case of an output packet from the decoder, the `iter_id` indicates the number of iterations used by the decoder. The `packet_count` is essentially an upcounter which can be used to determine if a packet was dropped.

Table 5.1. Packet Information

Packet Type	`packet_id`	`iter_id`	`packet_count`	`payload`
Message	0x1	N/A	512	8 bits/byte
Codeword (CW)	0x2	N/A	768	8 bits/byte
Noisy CW	0x5	max iters	12288	16 bits/symbol
Quantized CW	0x6	max iters	6144	8 bits/symbol
Successful decode	0x7	iters used	512	8 bits/byte
Failed decode	0x8	iters used	512	8 bits/byte

Since a packet decoding success or failure could occur on the last decode iteration, we could not use the `iter_id` alone to determine decoding success. Thus, we created two seperate `packet_id` values for success and failure (0x7 and 0x8 respectively). This value can be used by the receiver to handle decoder failures. In our system, we don't do anything in the case of a decoding failure. However, one could imagine a system in which failed packets are retransmitted. In this case,

the `packet_count` can be used to request a specific packet be retransmitted.

When testing our LDPC decoder on our FPGA development board, we quickly found that our test program was the bottleneck in our overall testing framework. We had the option to either optimize our Haskell code, or rewrite portions of the test program in C. Given the simple packet format and short timeframe we had to work with, we elected to rewrite portions of our program in C. In the end, we developed a C program which consisted of about 800 lines of code. Table 5.2 contains a listing of which components were and were not reproduced in the C program. As mentioned in the previous section, we have decoders written in Haskell and Kansas Lava. Of course, from Kansas Lava, we generated VHDL which was synthesized and loaded onto an FPGA on the Wildstar development board.

Table 5.2. Processes in the Testing Infrastructure

	Data	Encode	AddNoise	Quantize	Decode	Compare
C	YES	YES	YES	YES	NO	NO
Haskell	YES	YES	YES	YES	YES	YES

We developed the C program with a similar command line interface to the Haskell interface to make integration easier. The Makefile sample below performs the same operation as the Haskell version discussed previously.

The command line interface is nearly identical. We elected to include optional arguments for input and output this time, rather than just using Unix functions to redirect `stdin` and `stdout`. This choice was made primarily for debugging purposes during development.

Table 5.3 compares the relative performance of C and unoptimized Haskell for each block in our test program. It is clear that the computation in either program is dominated by the Encode operation. This is not surprising since the

```
allc::
        rm -f $(BITS)
        mkfifo $(BITS)
        $(CLDPC) --process=Data --rounds=$(ROUNDS) --output=$(BITS)  &

        rm -f $(ENCODED)
        mkfifo $(ENCODED)
        $(CLDPC) --process=Encode --rounds=$(ROUNDS) --input=$(BITS) --output=$(ENCODED)  &
        make addnoise

        rm -f $(NOISE)
        mkfifo $(NOISE)
        $(CLDPC) --process=AddNoise --noise=2.18 --input=$(DATA)/$(ENCODED) --output=$(NOISE) &

        rm -f $(RAWIN)
        mkfifo $(RAWIN)
        $(CLDPC) --process=Quantize --iterations=200 --input=$(NOISE) --output=$(RAWIN) &

        rm -f $(RAWOUT)
        mkfifo $(RAWOUT)
        ./*exe -f 30.0 $(RAWIN) $(RAWOUT)
```

Figure 5.1. Makefile for executing test using the optimized C Code

Table 5.3. Execution Time for Individual Blocks of Test Framework

	Data	Encode	AddNoise	Quantize
C	00:00.11	02:09.15	00:00.61	00:00.03
Haskell	00:01.33	15:55.39	02:48.42	01:12.14

encode operation is essentially a vector-matrix multiply between a 1×4096 vector and a 4096×7168 matrix while the rest of the operations are linear in time. Even after using the C version of our test program, we found that the system overall was still CPU limited. We decided that the easiest solution was to encode a very large number of packets, and store them in a file. Then these packets could be reused in our tests by generating random noise in real-time. To verify performance down, it was necessary to test the decoder down to a bit error rate of 10^{-6}. To provide a statistically sound result, it was necessary for the test to produce at least 1000 bit errors at this bit error rate. Thus, it would be necessary to simulate at least one billion bits. Given that each packet contains 6K bits, a sample of one million packets would give us a large enough set of data to perform all of the measurements we needed. After generating this data set, only the AddNoise and

Quantize processes would need to be performed in real-time. These operations are very fast in the C implementation, so our program was easily able to keep up with the output of our development board.

5.2 Description of the Testing Infrastructure

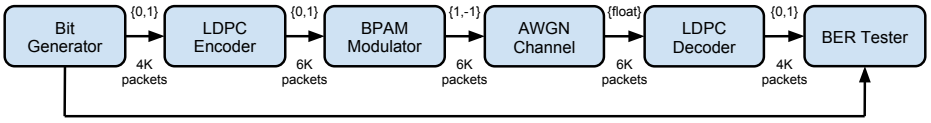


Figure 5.2. LDPC Test Framework

A model of our testing infrastructure is depicted in figure 5.2. At a high level, the testing infrastructure consists of a bit generator, an LDPC encoder, an Additive White Gaussian Noise (AWGN) block, an LDPC decoder, and a BER tester. We implemented all of these blocks in a Haskell program which uses a command-line interface to specify which operation will be performed and the associated options for the selected operation. The program takes an argument called `process` which determines which operation will be performed. The program has the option to read packets from a binary file pointer and writes packets to a binary file pointer (stdout). A simple Makefile was used to pipeline the testing infrastructure. The Makefile redirects input and output to unix fifos (named pipes). This pipelined Makefile system provided an easy way to allow us to take advantage of multiple CPUs in the host system. A sample of the makefile is included below:

The code sample above invokes the testing framework from beginning to end using our Haskell executable. We generate message packets, encode the packets, add noise, quantize the packets for decoding, and send the packets to the decoder

```
allh::
        rm -f $(BITS)
        mkfifo $(BITS)
        $(LDPC) --process=Data --rounds=$(ROUNDS) > $(BITS)  &

        rm -f $(ENCODED)
        mkfifo $(ENCODED)
        $(LDPC) --process=Encode --rounds=$(ROUNDS) $(BITS) > $(ENCODED)  &
        make addnoise

        rm -f $(NOISE)
        mkfifo $(NOISE)
        $(LDPC) --process=AddNoise --noise=2.18 $(DATA)/$(ENCODED) > $(NOISE) &

        rm -f $(RAWIN)
        mkfifo $(RAWIN)
        $(LDPC) --process=Quantize --iterations=200 $(NOISE) > $(RAWIN) &

        rm -f $(RAWOUT)
        mkfifo $(RAWOUT)
        ./*exe -f 30.0 $(RAWIN) $(RAWOUT)
```

Figure 5.3. Makefile for executing test using the unoptimized Haskell Code

running on our test board. The method of invoking each process is very similar. First, create a Unix FIFO for output, and then invoke the desired process. We've used variables as much as possible in the Makefile to maximize modularity. There is a companion process which acts as the BER tester. This process reads from the final output (`$(RAWOUT)`) and compares the output packets to the originally generated message packets.

```
[tbull@togo exec]$ make compare
/projects/fpg/data/ldpc4k --process=Compare /projects/fpg/data/Datam.4K.1M.dat Decoded.raw 1.0
[0.029108s] Packet (0,0) : (0,200) : 333 bers, 8.129883% [total: 333 bers, 8.129883%, 4096 ] #
[1.056257s] Packet (45,45) : (0,67) : 0 bers, 0.000000% [total: 2598 bers, 1.378864%, 188416 ]
[2.12255s] Packet (93,93) : (0,57) : 0 bers, 0.000000% [total: 5583 bers, 1.450039%, 385024 ]
[3.173391s] Packet (139,139) : (0,27) : 0 bers, 0.000000% [total: 7568 bers, 1.319754%, 573440 ]
[4.240439s] Packet (187,187) : (0,73) : 0 bers, 0.000000% [total: 8895 bers, 1.155123%, 770048 ]
[5.288494s] Packet (234,234) : (0,34) : 0 bers, 0.000000% [total: 10876 bers, 1.129904%, 962560 ]
```

A sample of the comparison output is included above. The compare process expects as arguments two filenames and a numeric value to indicate the period, in seconds, in which to write results to stdout. Above, the program is configured

to send output to the console once every second (see the final argument, 1.0). We output the packet count,number of iterations, BER for this packet, total bit errors, BER, and total bits. Additionally, the `#` symbol is used to indicate a decode failure occured in *this* packet. This is enough information to allow us to do debugging in real-time and also provides the necessary information for performing BER measurements such as those discussed in the next section.

For development, we used a Wildstar 5 development board from Annapolis Micro. This board contains three Xilinx Virtex 5vlx110tff1738-1 FPGAs, and uses a PCI-Express interface to communicate with the host computer. A photo of the development board is included in figure 5.4.

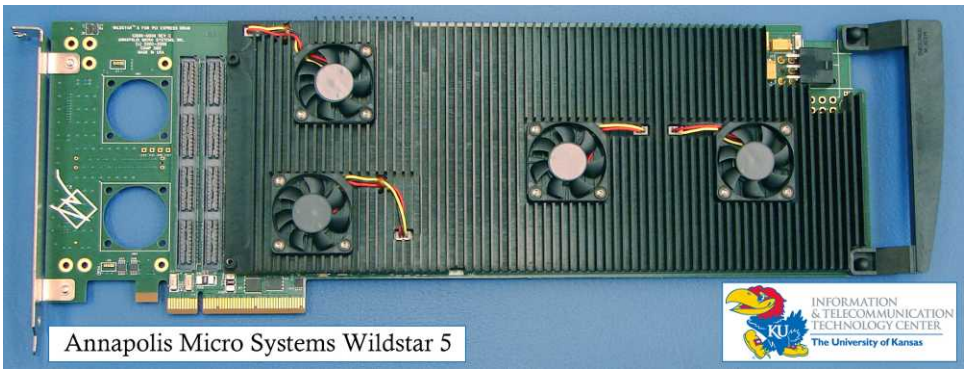


Figure 5.4. Wildstar 5 PCI-Express Development Board

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26794 tbull     25   0 26604 1252  836 R 99.9  0.0   0:27.48 basic_wrapper_d
26798 tbull     15   0 84996 7360 3612 S  3.7  0.1   0:00.93 ldpc4k
26786 tbull     18   0 55524  632  524 S  3.0  0.0   0:00.72 cldpc4k
 4392 tbull     18   0 55516  488  408 S  0.0  0.0   0:00.00 cldpc4k
```

Figure 5.5. Output from the top command during testing

Figure 5.5 contains the partial output of the Unix `top` command. Using effectively 100% of the time on a single CPU is the `basic_wrapper_driver` command which is responsible for communication with the Wildstar board. The `ldpc4k`

command is the comparison process discussed in the previous paragraph. The remaining processes are the C code which add noise and quantize packets. All of these processes use vary little resources compared to the communications process.

Table 5.4. Throughput in Packets/Second for Test Framework

	All Tasks	Noise + Quantization Only
C	7.97	48.20
Haskell	1.05	5.92

Table 5.4 compares the throughput of our decoder on the board while using our test framework. This table indicates that our C implementation can generate test data completely from scratch faster than the unoptimized Haskell implementation can perform only the Noise and Quantization processes while reading encoded packets from a file. If we use the C implementation to perform only the Noise and Quantization processes, we can run at an average of 48.2 packets/sec. In figure 5.5, we verified that the throughput is no longer limited by the test framework at this speed by verifying that the CPU time was being dominated by the communication process and not by data generation. It is also worth noting that the C implementation used less memory. Peak memory usage for the C implementation was around 45MB while peak memory usage for the unoptimized Haskell implementation was around 75 MB.

Once again, it is worth mentioning that the performance figures in this section are not an indicator of Haskell performance in general. In fact, Haskell is capable of producing very efficient code. The options for optimizing Haskell programs are vast. The GHC compiler contains very good profiling tools. Additionally, there are tools available such as HPC [11] for viewing code coverage as well as a number of libraries containing heavily optimized versions of the built-in Haskell data structures. We simply decided to use C where appropriate because we were

dealing with a small program that could just as easily be expressed in an imperative language. In circumstances where functional programming proved more beneficial, it would make more sense to optimize the original Haskell code.

5.3 Performance Results

We used the test framework described in the previous section to simulate the LDPC decoder at a range of signal-to-noise levels and iteration counts. Our decoder contains 8-bits of quantization, uses the $min^{\dagger}$ function discussed in chapter 4 and ran up to the maximum number of iterations for each packet received.

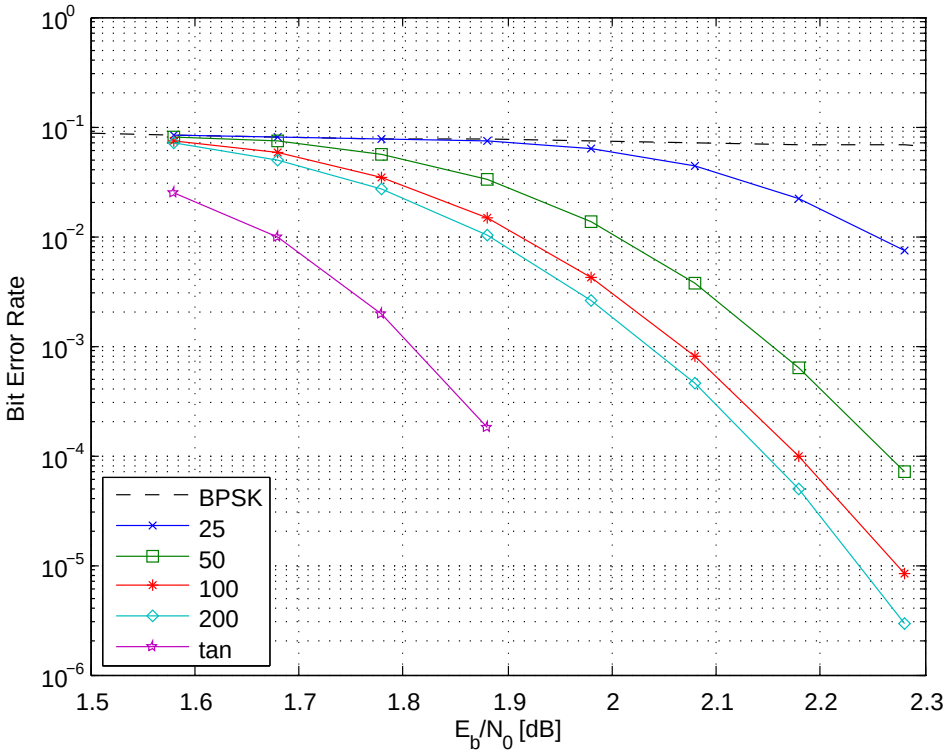


Figure 5.6. BER curve for 4096/6144 LDPC code

A plot containing our simulation results is included in figure 5.6. We included results from a software simulation (labeled tanh) in the plot. it is worth noting

that the tanh algorithm provides approximately 0.2dB better performance than $min^{\dagger}$, which is the expected result. Also, as we expected, raising the maximum number of iterations improves performance. However, in terms of bit error rate, we did see diminishing returns on increases to the number of iterations allowed.

Chapter 6

Memory

This chapter contains a description of a DDR2 DRAM interface which we developed for use in the HFEC project. Our wildstar 5 development board provides a single-port interface to off-chip DRAM. However, to make integration with the Kansas Lava system easier, it was necessary to develop a dual-port wrapper around this interface. This section describes the steps that were followed in this development process.

6.1 Annapolis Micro Wildstar 5 DDR2 DRAM Interface

The vendor, Annapolis Micro, provided a simple DRAM interface for the board which is illlustrated in figure 6.1. Limited documentation was provided and since the DRAM interface was on the board, it was not possible to use a simulator such as ModelSim for direct simulation. Through experimentation, the details of this interface became more clear and reliable communication was achieved with the DRAM. The read and write operations are described in detail in this section.

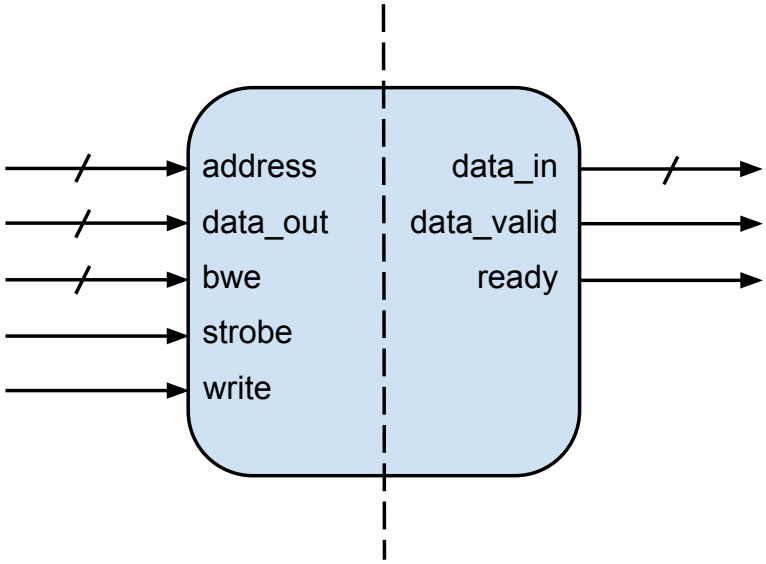


Figure 6.1. Annapolis Wildstar 5 DRAM Interface

To write to the DRAM, perform the following:

1. Wait for the `ready` signal to go `high`.
2. Assign desired values to `address` and `data_out`.
3. Set `bwe` (block write enable) to indicate which input bytes should be written.
4. Set `strobe` and `write` to `high`.

To read from the DRAM, perform the following:

1. Wait for the `ready` signal to be `high`.
2. Assign desired read address to `address`.
3. Set `strobe` to `high` and `write` to `low`.

The write process is straightforward, though the DRAM does not acknowledge that a write has occured. The user should only perform I/O when `ready` is `high`,

and it is understood that if `ready` is `high` then the data will indeed be written on *this* clock cycle. In the case of a read operation, the `data_valid` signal indicates that a read has occured. In general, the read output will appear on `data_in` on the cycle after a read is requested. There is not a multiple cycle read delay, which is common with on-chip block RAMs(BRAMs).

Annapolis Micro have chosen to name the data signals from the perspective of the FPGA. Thus, `data_in` is the input to the FPGA (and output of the DRAM), while `data_out` is the output of the FPGA (and input to the DRAM). Also, the widths of the `address`, `data_in`, `data_out`, and `bwe` vary depending on which FPGA on the board is used. In our case, the data signals are 64-bits wide, which meant that `bwe` is 8-bits wide (each bit corresponding to a data byte). One point worth mentioning is that if no data mask is desired, `bwe` should be set to all 1's. In our initial tests, it took some time to determine why we were always reading 0's from our memory. The reason for this turned out to be the `bwe` signal, which was not being set. Finally, the `address` width is 26-bits, which allows for a maximum of 512MB of addressable memory given that each address contains 64-bits of data.

6.2 Dual-Port DRAM Wrapper

The DRAM interface described above is a single-port interface. In Kansas Lava, we have a standard dual-port memory interface which we have used successfully to build FIFOs using BRAMs. The comparatively large capacity of off-chip DRAM provided motivation for extending this interface to the DRAM. Thus, it was necessary to provide a dual-port wrapper around the single-port DRAM interface provided by Annapolis. This dual-port interface is described in figure 6.2.

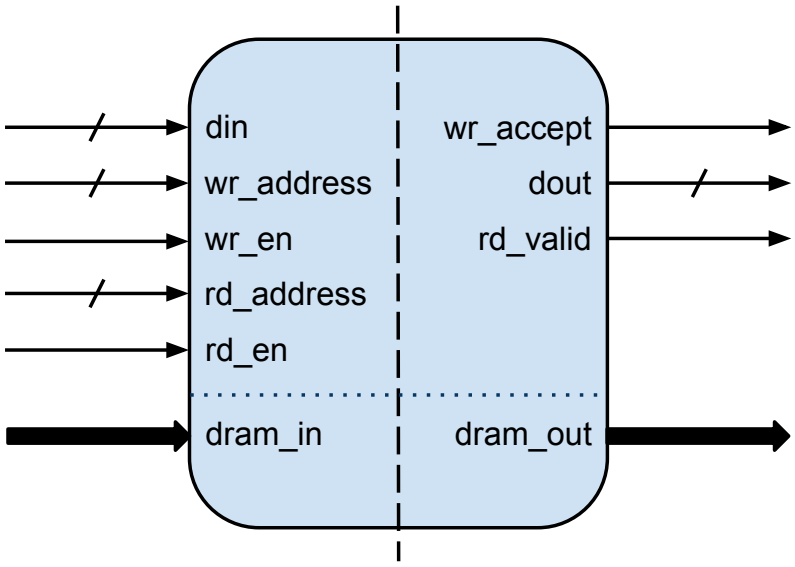


Figure 6.2. Dual-port DRAM Interface

The first step was adding seperate address lines for the read and write operations (`rd_address` and `wr_address`, respectively). Accompanying each address line is an enable signal (`rd_en` and `wr_en`) which should be set `high` whenever a read or write operation is desired. Also added is a `wr_accept` signal which is `high` on any cycle in which a successful write operation occurs. Additionally, there is a `rd_valid` signal which is `high` on any cycle in which the `dout` signal is valid. The `dram_in` and `dram_out` ports at the bottom correspond to the signals in figure 6.1. This interface is provided by Annapolis in the case of our Wildstar development board. The signals are bundled into a `record` which is a collection of signals in VHDL, similar to a `struct` in C. Thus, a larger arrow is used to depict a record in the figure. Since we are interfacing into a single-port DRAM, a read or write operation can occur on every clock cycle. However, if a read *and* write operation is desired on the same clock cycle, a delay must be introduced. In our desgin, if both read and write are requested, the write occurs on this cycle and the read occurs

on the following cycle. The choice of doing write-before-read was arbitrary, and it would not be difficult to reconfigure the wrapper to perform read-before-write. The current interface is illustrated in the following timing diagram.

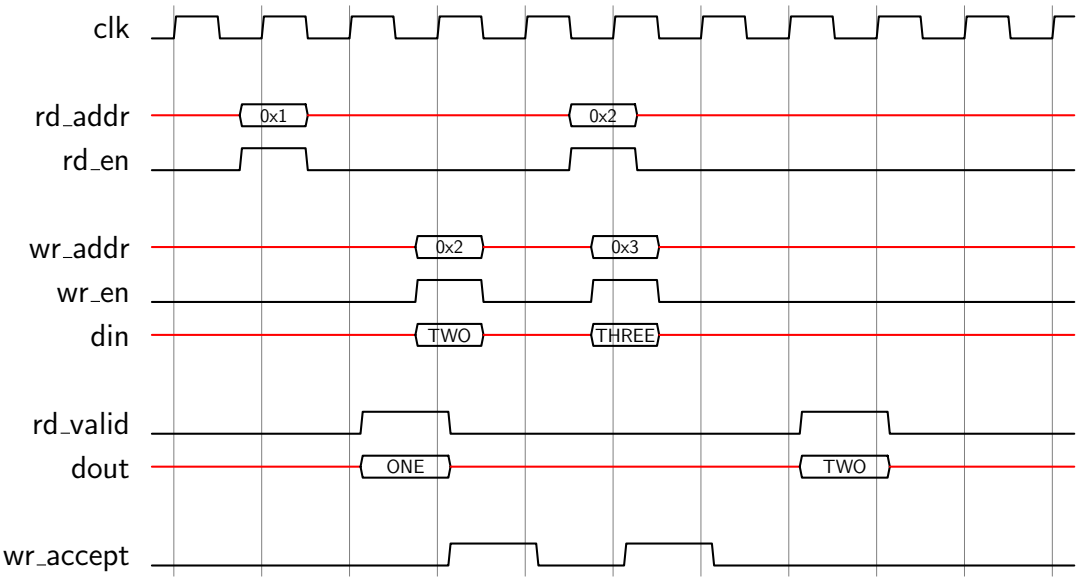


Figure 6.3. Dual Port DRAM Simulation Example

The first operation above is a read operation on address 0x1. The input signals are sampled on the rising edge of the clock cycle, and the read output (the symbolic value ONE) appears on `dout` on the following clock cycle. The next operation is a write operation. A request to write the value TWO to the address 0x2 is sent, and `wr_accept` goes high as soon at the rising edge of the clock cycle, and remains high until the following clock cycle. The final operation is actually a pair of operations: a read and a write. We will refer to this as a read/write operation in the remainder of this document. In this case, a read is requested on address 0x2 while a write is requested on address 0x3. Even though these input signals don't change at exactly the same time, the operations will be received at the same time since they both are sampled on the same rising edge of the clock.

The write is processed first, as indicated by `wr_accept` going high at the start of the clock cycle. The read is processed on the following clock cycle. The data read (TWO) appears on `dout` on the next clock cycle, effectively delayed an extra cycle. An important consideration is that any input to the DRAM on this extra cycle following the read/write request will be ignored. This is communicated to the user through the `wr_accept` and `rd_valid` signals. If a write operation were attempted, the `wr_accept` signal would not be set `high`, indicating that the `write` was not accepted. If a read were attempted, `rd_valid` signal would go `high` on the following clock cycle, but the circuit accessing the memory should still be waiting on the result of the previous read, indicating that this one was not accepted. In our test system, which performed thousands of sequential reads, writes, and read/writes, this protocol proved reliabile.

The implementation of this interface amounted to about 100 lines of VHDL code. The entity definition is given in figure 6.4. The architecture is included in the Appendix. The architecture contains a single process running on the rising edge of the clock. This process provides the logic necessary to manage the DRAM as described in the previous example.

6.3 Kansas Lava DRAM Interface

Given the entity in figure 6.4, we could provide a Kansas Lava interface in one of two ways. First, we could simply build a circuit (and `Fabric`) which has input and output signals corresponding to each of the signals in the `dram_dual_port` entity. This has the advantage that it does not require any further technology to be added to Kansas Lava. However, this would require we write some VHDL code to manually wire the two circuits together.

```
entity dram_dual_port is
    generic (
      WIDTH : integer := 64;
      ADDR_WIDTH : integer := 26;
      );
    port (
      clk            : in  std_logic;
      reset            : in  std_logic;
      din            : in  std_logic_vector(WIDTH-1 downto 0);
      wr_addr           : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
      wr_en      : in  std_logic;
      wr_accept  : out std_logic;

      rd_addr           : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
      rd_en      : in  std_logic;
      dout           : out std_logic_vector(WIDTH-1 downto 0);
      rd_valid   : out std_logic;

      dram_in    : in  dram_32b_interface_in_type;
      dram_out   : out dram_32b_interface_out_type
      );
end dram_dual_port;
```

Figure 6.4. VHDL Entity definition for a dual-port DRAM interface

Ideally, we would like to instantiate the `dram_dual_port` entity in our circuit directly. This would help eliminate a potential for error in the handwritten VHDL code. Additionally, handwriting such VHDL code quickly becomes tedious if the entity is instantiated multiple times. There are a couple of issues with this approach. First, `Fabric` currently only supports ports of type `std_logic` and `std_logic_vector`. Thus, we would need to add two more interface functions for `Fabric`:

```
inAbstract :: (Rep a) => String -> String -> Fabric (Seq a)
outAbstract :: (Rep a) => String -> String ->  Seq a -> Fabric ()
```

We've added an additional `String` argument to `inAbstract` and `outAbstract`

which would be used to describe the VHDL type. One example type is the `dram_32b_interface_in_type` used in `dram_dual_port` entity. This isn't the only issue however. Additionally, a circuit that instantiates `dram_dual_port` in Kansas Lava would also require a model of the entity in Kansas Lava to allow for simulation in GHCi. If we don't care about simulation, there is another issue with the actual instantiation of an external entity. Currently, `Fabric` gives us a way to name ports on an entity which we can then instantiate inside another VHDL circuit. However, we don't have a way of instatiating a VHDL entity inside of Kansas Lava, and that is necessary to solve this problem. This type of instantiation may take the form of another monad, like `Fabric`, or it may take the form of a function containing a list of ports on the external entity and their types. These issues are being explored currently, and certainly this functionality will be added to a future version of Kansas Lava.

Chapter 7

Conclusion

In the introduction of this thesis, it was stated that functional programming can be used to benefit the development of error correction coding systems. I have demonstrated this in several ways. First, in chapter 2, I demonstrated the use of Haskell in the implementation of a Viterbi decoder. In this case, Haskell was beneficial in that it provides a mathematical syntax and also a concise definition of the algorithm. It was proposed that a Haskell model of the Viterbi algorithm would allow for easier testing of a Kansas Lava implementation. Since a VHDL implementation was already in development, my attention was turned to the LDPC decoder. Next, in chapter 3, I introduced Kansas Lava and showed how GHCi can be used as a light-weight test environment for Kansas Lava circuits and also how synthesizable VHDL code can be generated from Kansas Lava circuit definitions. In chapter 4, I provided an overview of a complex example in which Kansas Lava was used to generate space and time efficient VHDL code. I also demonstrated the relatively painless process of adding additional functionality to Kansas Lava in the form of the `min` and `abs` functions. In chapter 5, I showed how Haskell was used to develop a rich test environment for our LDPC decoder

running on a development board. This chapter also showed one common shortcoming of unoptimized Haskell code in execution time. I showed how we were able to mitigate against this shortcoming by rewriting portions of our code in C. One may point to this as a failure in functional programming, or in Haskell. However, as was discussed at the end of the chapter, there were optimization opportunities available in Haskell. We simply chose to take the route of a C implementation because we knew that C would meet our goals in this circumstance. Finally, the end of chapter 6 discusses how, in the future, Kansas Lava will be able to directly instantiate entities developed outside of Kansas Lava allowing for a greater degree of interoperability and less time spent hacking wrappers and interfaces in VHDL.

Through my work on this project, I have had the opportunity to work in two very different worlds: functional programming and hardware design. In doing so, I have learned the power of abstraction in programming. In terms of abstraction, we have successfully used a very high level language in Haskell to make the low-level, and sometimes tedious, task of hardware development easier. Functional programming has provided several benefits: a more mathematical syntax for describing algorithms in Haskell, a light-weight software simulation platform in GHCi, and hardware optimization opportunities in Kansas Lava that would likely require a great deal of code rewriting in VHDL. With these benefits come some caveats: the Kansas Lava developer must also be a skilled Haskell programmer, care must be taken to ensure efficiency, and testing on hardware is still messy. These issues are discussed further in the following section.

7.1 Future Work

Arguably, the most important and most difficult task is making Kansas Lava more accessible. Learning a language like Haskell is a time consuming task for a hardware designer with no prior experience in functional programming. However, a deep understanding of Haskell is a requirement for efficient use of Kansas Lava. Second, more efficient methods of testing Kansas Lava circuits is an area in need of improvement. Testing in software is easy. Testing inside Kansas Lava is accomplished in GHCi, and efforts have been made to make testing in ModelSim easier [5], but testing on hardware becomes much more complex. A testing framework, including a lot of handwritten VHDL and C code, must be developed for every combination of circuit and development board. Some work has been done to provide a standard test interface in Lambda Bridge [8], but ideally one would test circuits running on hardware from GHCi or a GHCi-like interface. In chapter 4, we demonstrated how a simple Haskell function can be *lifted* into Kansas Lava, but there are certainly opportunities to investigate the generation of Lava circuits, and potentially VHDL, from more general Haskell programs. Lastly, as a lab, have successfully implemented two different HFEC algorithms in Kansas Lava (LDPC, discussed in chapter 4 and SDVA, implemented in Kansas Lava by team member, Ed Komp). It is certain that different algorithms in different fields of study will present unique challenges that we have not even considered.

Chapter 8

Appendix

8.1 Metric Function in VHDL

This section contains a version of the Metric function discussed in chapter 4. This code takes a similar approach to the optimized Kansas Lava approach except that the `sign` signal is the opposite of `flipSign` in our Kansas Lava implementation, and the output of the `nabs2` is flipped correspondingly. Likewise, the `min` function was changed to a `max`. This choice was quite arbitrary and does not change the resulting output.

```
entity ldpc_metric is
  generic (
    width_size : natural := 8;      -- signal width
    max_value : natural := 8);      -- value for max * min
  port (i0 : in  std_logic_vector(width_size-1 downto 0);
        i1 : in  std_logic_vector(width_size-1 downto 0);
        o0  : out std_logic_vector(width_size-1 downto 0));
end ;

architecture Behavioral of ldpc_metric is
  signal sign : std_logic := '0';       -- sign of output signal

  -- define some temp signals to satisfy the compiler...
```

```
  signal i0s, i1s, i0temp, i1temp, ztemp : signed (width_size-1 downto 0);
  signal negZtemp : std_logic_vector(width_size - 1 downto 0);


  -- min calculates the minimum of two signed signals
  function max(signal a, b : in signed) return signed is
  begin
    if a < b then
      return b;
    else
      return a;
    end if;
  end function max;

  -- abs2 calculates the absolute value of a signed signal
  function nabs2(signal a : in signed) return signed is
  begin
    if a < 0 then
      return a;
    else
      return -a;
    end if;
  end function nabs2;

begin

    -- figure out whether we need to flip the sign of the output
    sign <= i0(width_size-1) xor i1(width_size-1);

    i0s <= signed(i0);
    i1s <= signed(i1);

-- define temp signals, and output
    i0temp <= nabs2(i0s);
    i1temp <= nabs2(i1s);
    ztemp  <= max(i0temp,i1temp);

    neg : entity Sampled_negate
    generic map (
      width_size => width_size,
      max_value  => max_value)
     port map (i0 => std_logic_vector(ztemp),
```

```
            o0 => negZtemp);

    -- flip the sign on output only if the inputs have two different signs.
    o0 <= std_logic_vector(ztemp) when sign = '1' else negZtemp;

end Behavioral;
```

8.2 DRAM Dual-Port Wrapper

Below is the VHDL architecture definition for the `dram_dual_port` wrapper discussed in chapter 6.

```
architecture std of dram_dual_port is

  constant dram_bwe : std_logic_vector(7 downto 0) := x"FF";

  signal dram_dout, dram_din    : std_logic_vector(WIDTH-1 downto 0);
  signal dram_addr, rd_addr_reg : std_logic_vector(ADDR_WIDTH-1 downto 0);
  signal dram_write, dram_ready  : std_logic;
  signal dram_strobe : std_logic;

  signal rw_sig : std_logic_vector(1 downto 0) := "00";
  signal skip : std_logic;
  signal rd_valid_1, rd_valid_2, rd_valid_current : std_logic;

begin

  dram_out.data_out <= dram_din;
  dram_out.bwe      <= dram_bwe;
  dram_out.address  <= dram_addr;
  dram_out.strobe   <= dram_strobe;
  dram_out.write    <= dram_write;
  dram_dout <= dram_in.data_in;
  rd_valid <= dram_in.data_valid;
  dram_ready <= dram_in.ready;

  rw_sig <= rd_en & wr_en;

  control_proc: process (clk, reset)
  begin  -- process control_proc
```

```
if reset = '1' then
  dram_strobe <= '0';
  dram_write <= '0';
  skip <= '0';
elsif rising_edge(clk) then
  if dram_ready = '1' then
    if skip /= '1' then
      case rw_sig is
        when "01" =>
          dram_strobe <= '1';
          dram_write  <= '1';
          dram_addr   <= wr_addr;
          dram_din    <= din;
          skip <= '0';
        when "10" =>
          dram_strobe <= '1';
          dram_write  <= '0';
          dram_addr   <= rd_addr;
          dram_din    <= din;
          skip <= '0';
        when "11" =>
          dram_strobe <= '1';
          dram_write  <= '1';
          dram_addr   <= wr_addr;
          dram_din    <= din;
          rd_addr_reg <= rd_addr;
          skip <= '1';
        when others =>
          dram_strobe <= '0';
          dram_write  <= '0';
          dram_addr   <= rd_addr;
          dram_din    <= din;
          skip <= '0';
      end case;
    else
      dram_strobe <= '1';
      dram_write  <= '0';
      dram_addr   <= rd_addr_reg;
      dram_din    <= din;
      skip <= '0';
    end if;
  end if;
```

```
    end if;
  end process control_proc;

  wr_accept <= '1' when ((skip = '0') and (rw_sig = "01" or rw_sig = "11")) else
               '0';
  dout <= dram_dout;

end std;
```

References

[1] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones, and F. Pollara. The development of turbo and LDPC codes for deep-space applications. *Proc. IEEE*, 95(11):2142–2156, Nov. 2007.

[2] T. Bull, E. Perrins, and A. Gill. Implementation of the viterbi algorithm using functional programming languages. In *Proceedings of the International Telemetering Conference*, Oct 2009. (Student Paper).

[3] K. Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology, April 2001.

[4] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.

[5] A. Farmer, G. Kimmell, and A. Gill. What's the matter with Kansas Lava? In *Proceedings of Trends in Functional Programming*, May 2010.

[6] R. G. Gallager. Low density parity check codes. *Transactions of the IRE Professional Group on Information Theory*, IT-8:2l–28, January 1962.

[7] A. Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[8] A. Gill, T. Bull, D. DePardo, A. Farmer, E. Komp, and E. Perrins. Using functional programming to generate an LDPC forward error corrector. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, May 2011.

[9] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing Kansas Lava. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.

[10] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing Kansas Lava. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.

[11] A. Gill and C. Runciman. Haskell Program Coverage. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Haskell*. ACM Press, September 2007.

[12] S. Lin and D. Costello. *Error Control Coding*. Prentice Hall, New York, 2004.

[13] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[14] T. K. Moon. *Error Correction Coding: MatheMatical Methods and Algorithms*. Wiley, New Jersey, 2005.

[15] J. O'Donnell. Overview of Hydra: a concurrent language for synchronous digital circuit design. In *Parallel and Distributed Processing Symposium*, pages 234–242, 2002.

[16] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[17] I. Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, April 2003.

[18] M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

[19] S. Singh. Designing reconfigurable systems in lava. *VLSI Design, International Conference on*, page 299, 2004.