7-2018

Exploring High Level Synthesis to Improve the Design of Turbo Code Error Correction in a Software Defined Radio Context

Bradley E. Conn
bec5030@rit.edu

Exploring High Level Synthesis to Improve the Design of Turbo Code Error Correction in a Software Defined Radio Context

Bradley E. Conn

Exploring High Level Synthesis to Improve the Design of Turbo Code Error Correction in a Software Defined Radio Context

Bradley E. Conn

July 2018

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

R·I·T | Kate Gleason
College of ENGINEERING

Department of Computer Engineering

Exploring High Level Synthesis to Improve the Design of Turbo Code Error Correction in a Software Defined Radio Context

Bradley E. Conn

Committee Approval:

Dr. Sonia Lopez Alarcon *Advisor* Date
RIT, Department of Computer Engineering

Dr. Marcin Lukowiak Date
RIT, Department of Computer Engineering

Dr. Andres Kwasinski Date
RIT, Department of Computer Engineering

"Unless someone like you cares a whole awful lot,

Nothing is going to get better. It's not."

- The Lorax

Abstract

With the ever improving progress of technology, Software Defined Radio (SDR) has become a more widely available technique for implementing radio communication. SDRs are sought after for their advantages over traditional radio communication mostly in flexibility, and hardware simplification. The greatest challenges SDRs face are often with their real time performance requirements. Forward error correction is an example of an SDR block that can exemplify these challenges as the error correction can be very computationally intensive. Due to these constraints, SDR implementations are commonly found in or alongside Field Programmable Gate Arrays (FPGAs) to enable performance that general purpose processors alone cannot achieve. The main challenge with FPGAs however, is in Register Transfer Level (RTL) development. High Level Synthesis (HLS) tools are a method of creating hardware descriptions from high level code, in an effort to ease this development process. In this work a turbo code decoder, a form of computationally intensive error correction codes, was accelerated with the help of FPGAs, using HLS tools. This accelerator was implemented on a Xilinx Zynq platform, which integrates a hard core ARM processor alongside programmable logic on a single chip.

Important aspects of the design process using HLS were identified and explained. The design process emphasizes the idea that for the best results the high level code should be created with a hardware mindset, and written in an attempt to describe a hardware design. The power of the HLS tools was demonstrated in its flexibility by providing a method of tailoring the hardware parameters through simply changing values in a macro file, and by exploration the design space through different data types and three different designs, each one improving from what was learned in the previous implementation. Ultimately, the best hardware implementation was over 56 times faster than the optimized software implementation. Comparing the HLS to a manually optimized design shows that the HLS implementation was able to

achieve over a 19% throughput, with many areas for further improvement identified, demonstrating the competitiveness of the HLS tools.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

1.1.1 Software Defined Radio

The ever increasing power of digital technology as well as newly developed tools have allowed software defined radio (SDR) to grow from an idea to a reality. SDRs provide many advantages over traditional hardware radios including, flexibility, customizability, longevity, reliability, and repurposability to name a few. SDR is a way of implementing typical radio communication hardware, such as mixers, filters, gain controllers, modulators, demodulators, using software.

The flexibility of SDR might be the most compelling aspect of the concept. SDRs often allow a wide range of frequencies ranging from KHz to GHz to be used in a single radio, which is uncommon for hardware radios. More importantly, SDRs can potentially implement any protocol, or a range of protocols at once. In a common cell phone for example, one might find different hardware specifically for Bluetooth, Wifi, GPS, and GSM or CDMA communication. With SDR, all of these could potentially be implemented in one, eliminating the need for separate dedicated hardware for the different protocols.

The upgradability, reconfigurability, and repurposability are also enticing aspects of SDRs. These capabilities allow radios to gain new features, new protocols, fix

bugs, or change to suit entirely new applications. Additionally, this can also occur remotely, which can have numerous benefits.

Reliability is another significant aspect of SDRs. Increased reliability can occur as there is limited component tuning required, there are fewer components that can fail, such as diodes or capacitors, and parameter differences, such as temperature change or manufacturing variations, are limited.

Cost saving is another element that can be considered for SDRs. Cost savings can occur as software can easily be reused for free, cutting down on design time and material cost. This eliminates the need for expensive new hardware designs, enabling low cost research development and testing. Additionally to this, little to no maintenance is required.

Although SDRs have numerous advantages they also has very challenging disadvantages. The biggest one is that SDRs require a lot of processing power, increasing power consumption, and potentially increasing cost. The technology behind it, such as processors, Analog to Digitl Conveters (ADCs), and Digital to Analog Converters (DACs), are not always fast enough for SDR applications. Additionally, the software code can quickly become very complex.

1.1.2 Turbo Codes

One interesting SDR component is the error correction code block. Specifically, turbo code error correction. Turbo codes are a widely used form of forward error correction that provides near channel capacity performance[1]. Their uses can be seen from low signal to noise deep space satellites to real time cell phone communication. For instance, the Long Term Evolution (LTE) standard uses turbo codes for error correction[2] which requires low latency, and high reliability. Most implementations of turbo codes can be found in integrated circuits due to their real time constraints and computational intensity, which do not hold the advantages of SDR. Due to these

requirements however they pose a significant challenge for SDRs, but remain an important aspect for radio communication.

1.1.3 Field Programmable Gate Arrays

Even though software defined radios are accessible like never before, many applications are not able to run on truly general purpose processors alone. SDRs often require accelerators in the forms of Digital Signal Processors (DSPs) or Field Programmable Gate Arrays (FPGAs). FPGAs allow custom digital hardware functionality to be implemented, and can be reconfigured to implement new functionality when desired. They are particularly interesting for software defined radios due to their ability to accelerate specific tasks. Turbo codes provide a great module for FPGA acceleration for their complexity, computational requirements, and frequent use described previously. Using an FPGA can have the advantage that the implementation can be tailored to the application requirements, handling different parameters such as latency, resource usage, power consumption, and accuracy.

1.1.4 High Level Synthesis

Even with the tremendous performance improvements FPGAs can provide, the most daunting aspect is developing for them. Development can be long and difficult. An ever evolving method for developing for FPGAs is through the use of High Level Synthesis (HLS) tools. HLS provides a design flow in which algorithms are implemented using high level languages such as C or C++. These implementations get interpreted by the HLS tools, which generate Register Transfer Level (RTL) models of the algorithm in a hardware description language such as VHDL or Verilog. This removes the developer from many of the low level nuances required by manually optimized designs, which comes at the cost of complete customization. Manually optimized designs will generally outperform HLS designs but require significant development time. Addi-

tionally, manually optimized designs require developers with knowledge of hardware description languages whereas HLS can be used with only knowledge of high level language. This opens the benefits of FPGAs to a new user base many times larger than those with Hardware Description Language (HDL) knowledge. The decreased development time allows for extensive exploration into the design space with much less effort. Alongside this, the verification is greatly simplified, as higher level code can be written more easily for the different test cases, and tested in software as well as RTL simulation from the same tests files.

For the best results, a design process should be followed which describes the hardware desired in a way that is effective for HLS translation. This paper examines an approach to accelerate turbo code decoding with an FPGA using HLS tools. The design process behind it is explored and the trade offs are analyzed. It also exemplifies the advantages, flexibility and customization with significantly less development time than that of a manual optimized hardware design.

1.2 Related Works

Previous research has explored the idea of implementing turbo code decoding in the context of SDR. In [3-8] the hardware capabilities are considered when implementing the decoder. Moreover, the implementation of the algorithm is tailored to take advantage of the specific hardware platform. In [3], [4], and [5], DSP are used as the underlying hardware, while in [6] a GPU is used, in [7] a CPU is used, and in [8] multiprocessor systems are used. Even with these carefully considered implementations, the performances achieved may still not be enough.

Although ASIC design for turbo codes is a commonly researched topic [9], and can achieve the highest performance, the lack of flexibility does not lend itself to SDR. Because of this, FPGA implementations are popular for implementing turbo codes for SDR applications. Still, most of the techniques used in turbo code ASIC

designs and turbo code FPGA designs can be used independent of the platform for implementation. Much of the research in this domain relates to improving the algorithm by reducing the computational intensity, reducing the resource requirements, and improving the throughput. These techniques commonly focus on using the log domain to reduce the computational intensity [10], and use a sliding window [11] or other parallel implementations [12] to reduce resource requirements and improve throughput. Even still, the actual hardware implementation is not easy to create, and exploring new design techniques can be a difficult and time consuming task.

High Level Synthesis has been studied to see its effectiveness compared to traditional manual development [13, 14, 15], and has been specifically used for SDR implementation [16]. In [16] an SDR implementing the Zigbee protocol is constructed using HLS. This work provides a proof of concept for the ability of HLS for SDR implementations. A manually optimized design was used as the baseline, and was created in conjunction with the HLS to verify functionality. HLS has also been leveraged for Error Correction Codes. In [17] a turbo code decoder is implemented. This work takes a C implementation and attempts to implement it on an FPGA using HLS for simulation purposes, and to gain an understanding of how quick development can be achieved. The implementation does not come from a hardware design and does not take advantage of any of the standard hardware techniques for effectively implementing turbo codes such as a Sliding Window technique. As such the implementation is not very optimized.

Low Density Parity Check (LDPC) codes are a competing code for forward error correction code that also approaches the Shannon limit and is computationally complex. HLS has been utilized to implement LDPC codes in several works [18] [19] [20]. In [18] three different HLS tools are compared and demonstrate performance within the same order of magnitude as manually optimized designs. In [19] two LDPC decoder architectures are proposed and implemented using HLS to achieve throughputs

greater than state or the art designs which are compared. In [20] Vivado HLS tools are examined to see their effectiveness when implementing LDPC codes. The paper discusses the mapping of the code to the hardware in detail and demonstrates the ability to implement non trivial designs.

Chapter 2

Software Defined Radio

2.1 Software Defined Radio

Software Defined Radio (SDR) has become an increasingly attractive radio implementation. The flexibility, reliability, and upgradability provide distinct advantages over traditional radio hardware configurations. SDR allows a single platform to be used for many different radio applications and protocols. SDR uses software to implement radio components that are usually implemented in hardware. This allows incredible opportunity for numerous different applications and protocols to work on a single piece of hardware, whereas hardware radios are limited by the fixed architecture used for its implementation.

One of the first well known software defined radios was the product of the Department of Defense and went by the name of Speakeasy [21]. Speakeasy was programmable waveform, multiband, multimode radio, capable of emulating more than 15 existing military radios. It proved the concept that the same hardware can be used for many different forms of radio communication through different software.

A simple example of a potential SDR application can be seen in modern day smartphones. Smartphones can communicate over many different protocols such as GSM, Wifi, and Bluetooth. Generally, to implement these protocols there is specific hardware required which is unique to each protocol. Utilizing SDR, only a single

piece of hardware could cover a range of protocols at once.

One of the most compelling uses of software defined radio is in satellite communication. Main satellites fail over time due to hardware maintenance issues. This probability is drastically decreased with SDR as there are less hardware components to fail. Along with this, as technology moves forward over the years, a satellite can still benefit from the software radio advancements, without ever needing to physically access the satellite, something not possible with hardware radios. This updating benefit can be seen in two examples. First, the Mars Recon Orbiter updating its communication to use to Adaptive Data Rates, allowing much faster more efficient communication[22]. Second, Voyager updated its Error Correction Code (ECC) after launch which was critical to continued mission success[23].

2.1.1 Common Radio Principals

In order to understand typical SDR implementations it is helpful to have an understanding of some common radio principals. This is particularly helpful for understanding the partitioning between the hardware and where the software starts.

2.1.1.1 Heterodyning and Superheterodyning

Heterodyning and superheterodyning are common techniques found in most radio communication today. Heterodyning mixes two frequencies to make a new frequency. Often this is a carrier frequency mixed with a signal to create a modulated signal, or to demodulate a signal. Superheterodyning is used in receiving radio communication, and takes the Radio Frequency (RF) signal and uses heterodyning to convert it into an Intermediate Frequency (IF) signal. From here more processing is completed and it is converted to the baseband signal. There are a few reasons why this approach is taken. First, if processing such as filtering and amplification are done at the high RF frequencies they have poorer performance than the lower frequencies. Second, if

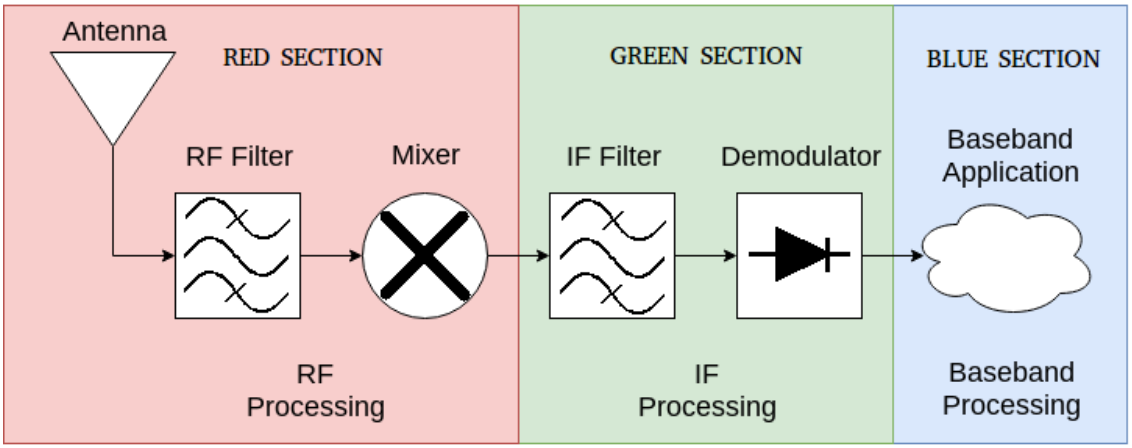

Figure 2.1: An example superheterodyning receiver to display the partitioning of the RF, IF, and baseband signals. An SDR can sample at any of the points along the path.

a radio can be tuned to different frequencies, it brings it to a common frequency for further processing, eliminating the challenges that various frequencies pose. This is typically accomplished by varying the oscillator that is used for heterodyning, and keeping all of the tuning hardware constant for a fixed frequency. From here after processing the signal is demodulated to obtain the baseband signal. A potential radio reciever can be seen in Figure 2.1. In the figure, the RED SECTION shows the processing in the RF signal, the GREEN SECTION shows the processing in the IF signal, and the BLUE SECTION shows the baseband signal. The filters and amplifiers are used to eliminate aliasing and unwanted frequencies, and tune the desired signal to make it stronger.

2.1.2 Ideal SDR

Software defined radio requires some way for the software to interact with the real RF world. Analog to Digital Converters (ADCs) and Digital to Analog Converters (DACs) are used to convert the signal from RF to digital and digital to RF. Thus, the perfect SDR transmitter would have three components. A processing system, a DAC, and an antenna. The perfect SDR receiver would also have three components. An

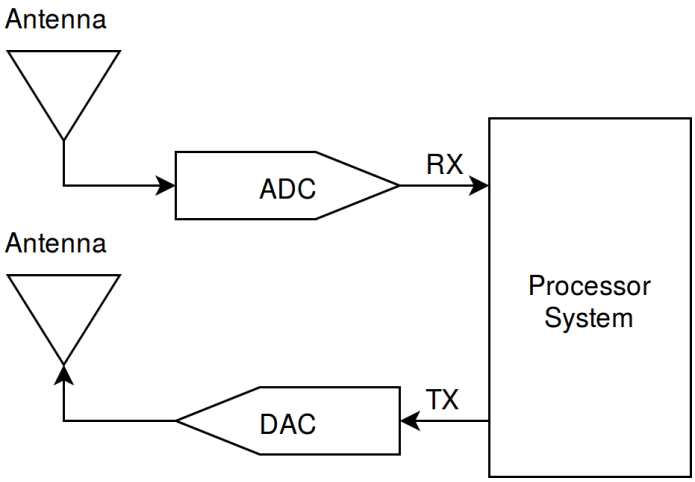

Figure 2.2: The ideal SDR receiver and transmitter made up of an antenna, ADC/DAC and processing system.

antenna, an ADC, and a processing system. This SDR setup can be seen in Figure 2.2. DACs and ADCs in reality are limited, and thus limit the frequencies that can be captured. Even if this weren't an issue, processing systems do not have unlimited computational power. This causes a typical SDR to have more hardware than an ideal SDR would have. Many SDR receivers implement much of the superheterodyne receiver shown in Figure 2.1 where the sampling is done at any point along that data path. It can be done in the RF, IF, baseband or later as desired.

2.1.3 Open Source SDR C and C++ Libraries

Different libraries were examined to see which was the best candidate to integrate with FPGA acceleration using HLS. The following is a small synopsis of three of the libraries examined to see how suited they were for the requirements.

2.1.3.1 GNU Radio

GNU Radio is one of the most widely used SDR libraries available. It provides many signal processing blocks that can be connected together and used to create SDRs. There is a useful visual tool that can be used to quickly connect SDR blocks for rapid

prototyping. The library is mainly written in C++, with many of the user tools being written in python. The SDR implementations can run with actual hardware, or in a simulated environment. Leveraging HLS with GNU radio can be difficult as it relies on dynamic memory and pointers, has its own data passing mechanism, has its own scheduler, and has many external dependencies. These overheads are not well suited for HLS and could be difficult to overcome.

2.1.3.2 Liquid DSP

Liquid DSP is a lightweight open source library for DSP written in C. It was designed to be a standalone framework for embedded SDR implementations with minimal external dependencies, and minimal overhead. It provides blocks that are scalable, flexible, and dynamic. The library provides no underlying infrastructure for connecting components, managing memory, or proprietary datatypes. That is left to the user to handle, to eliminate unnecessary overhead and keep it light weight. It compiles using CMAKE which may take some converting to be compatible with HLS tools.

2.1.3.3 CSDR

CSDR is a library written in C that was created for the cost effective RTL2832U-SDR dongles. It is a comparatively small library and is restricted to receiving transmission only as the RTL2832U-SDR dongles are incapable of transmitting. As such, it is a very simple library with decent code documentation. This library was used for the initial SDR prototyping with HLS.

Chapter 3

Turbo Codes

3.1 Turbo Code Error Correction

Turbo codes provide a technique for error correction with near Shannon limit performance that can often be found in the context of SDR. Turbo codes are iteratively decodable codes allowing them to be customized for accuracy and latency. The codes consist of an encoder and a decoder, with the encoder being relatively simple, and the decoder being much more complex. A simplified system integration can be seen in Figure 3.1. In this figure, a bit sequence gets encoded by a turbo code encoder, modulated, sent over a noisy channel, demodulated, error corrected using a turbo code decoder, and a bit sequence is recovered. The Encoder and the Decoder blocks are used to represent the turbo codes.

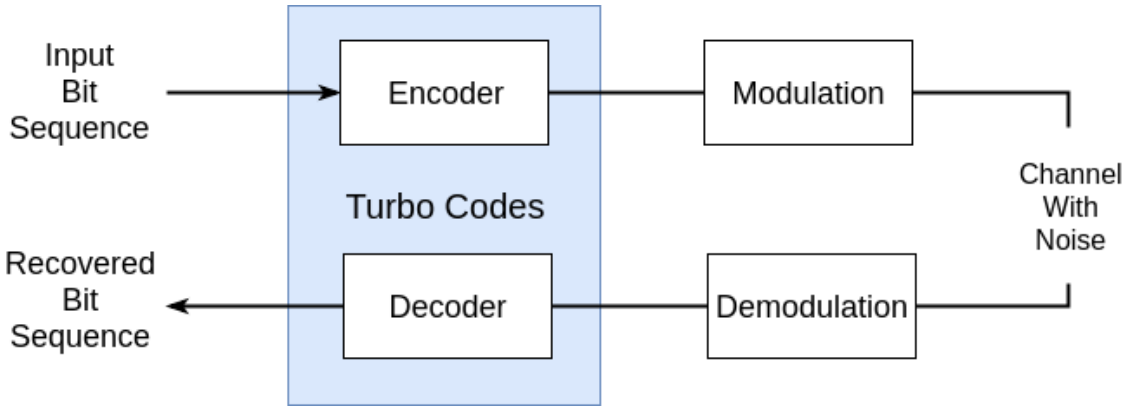

Figure 3.1: A simplified example of how turbo codes are integrated into radio communication.

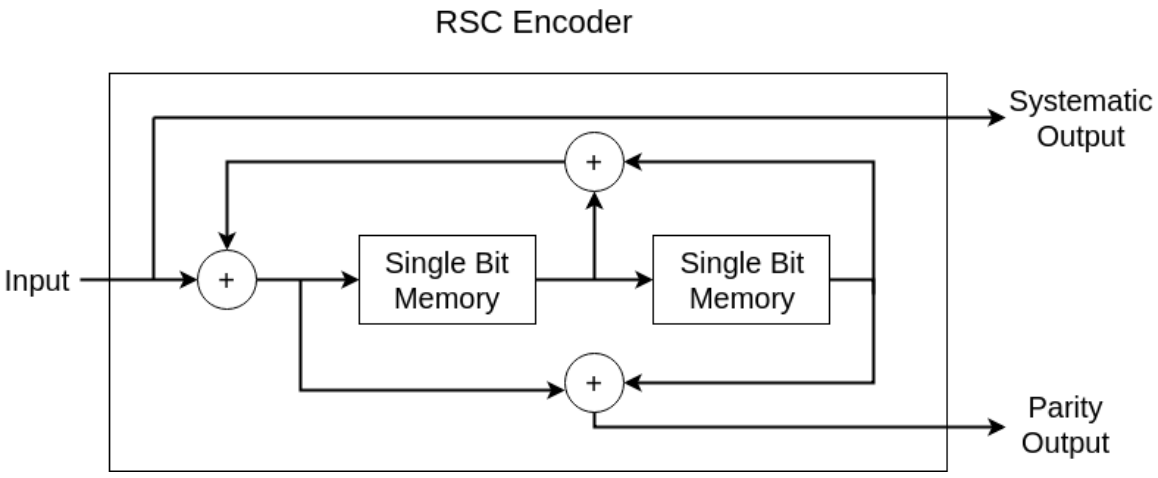

Figure 3.2: An example of a memory 2 Recursive Systematic Convolutional (RSC) Encoder.

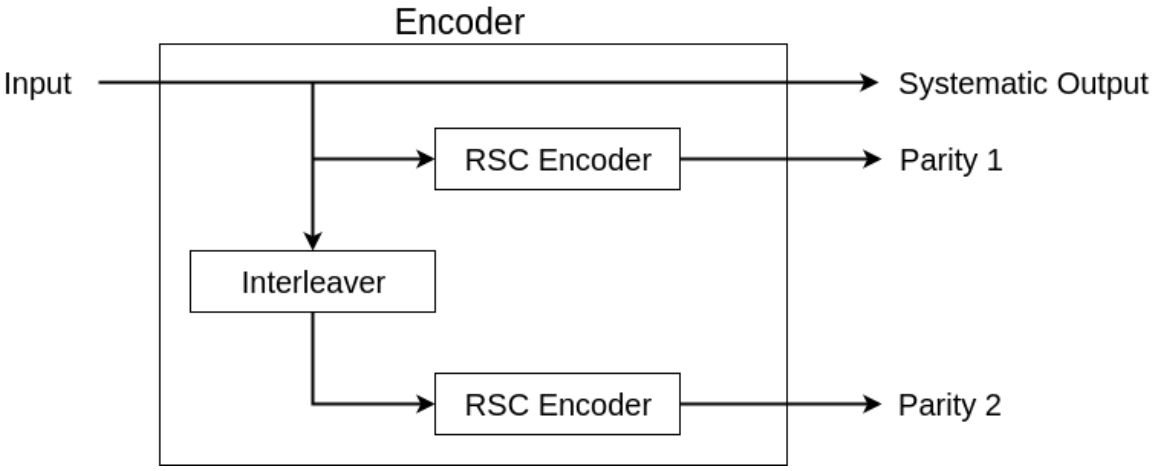

Figure 3.3: An example of a turbo code encoder containing two RSC encoders.

3.1.1 Turbo Code Encoder

Turbo code encoding at the lowest level uses a Recursive Systematic Convolutional (RSC) encoder. An example is shown in Figure 3.2. An RSC encoder takes an input sequence of bits and produces two output sequences of bits of equal length. One sequence exactly matches the input sequence, known as the systematic bits, and the other sequence is the parity bits, or redundant information used to help recover the input sequence. The recursive aspect of RSC encoders uses a feedback which creates an infinite impulse response which is beneficial for encoders. This infinite impulse

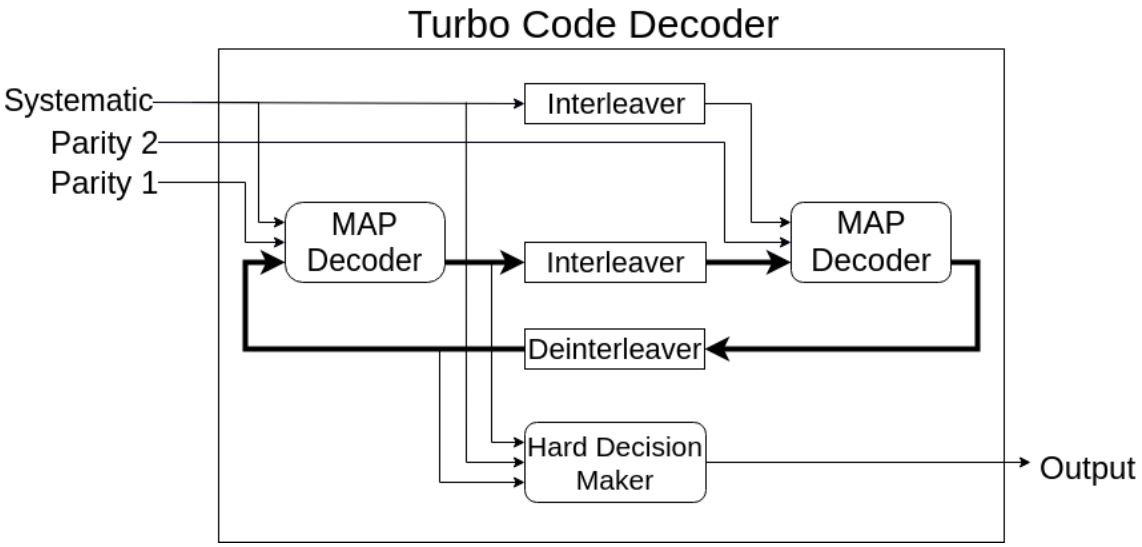

Figure 3.4: A high level view of the turbo code decoder consisting of MAP decoders, interleavers, deinterleavers, and a hard decision maker. The bold lines highlight the feedback loop present in the turbo decoder.

will continue to have the parity bits change even if in the presence of a long string of the same bit values which is beneficial for error correction.

Multiple RSC encoders can be concatenated in parallel to increase the amount of parity information. In order to decorrelate the redundancy bits from the two encoders, an interleaver is used at the input of the second RSC encoder to change the order of the bits. This is done for each subsequent RSC encoder. The additional RSC encoders systematic outputs do not provide new information and are ignored. An example turbo code encoder with two RSC encoders can be seen in Figure 3.3. This example encoder is used for the remainder of this section.

3.1.2 Turbo Code Decoder

In this example the turbo code decoder contains two Maximum A Posteriori (MAP) decoders, otherwise called Bahl, Cocke, Jelinek, and Raviv (BCJR) decoders [24]. Each MAP decoder is used for one parity sequence.

The outputs of the MAP decoders are used as inputs to the subsequent decoders forming a feedback loop. As the number of iterations through the feedback loop

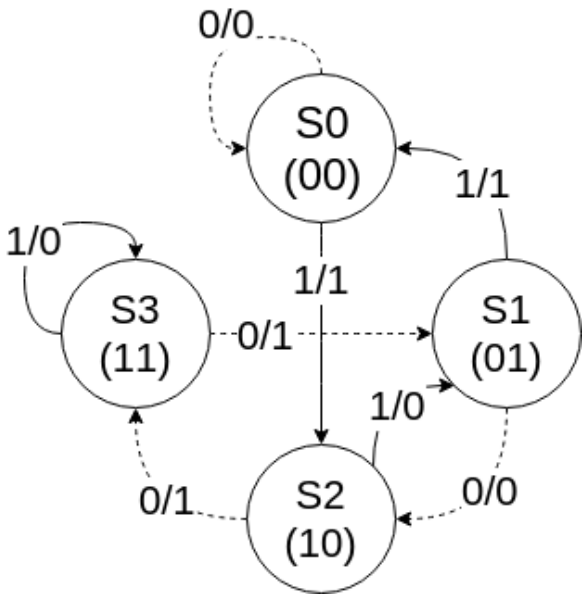

Figure 3.5: The state machine representing the example RSC encoder. The encoder memory is shown in the parenthesis. Inputs of zero are shown as dotted lines, and inputs of one are shown with full lines. The input and output format is represented as input/output marked on the transition lines.

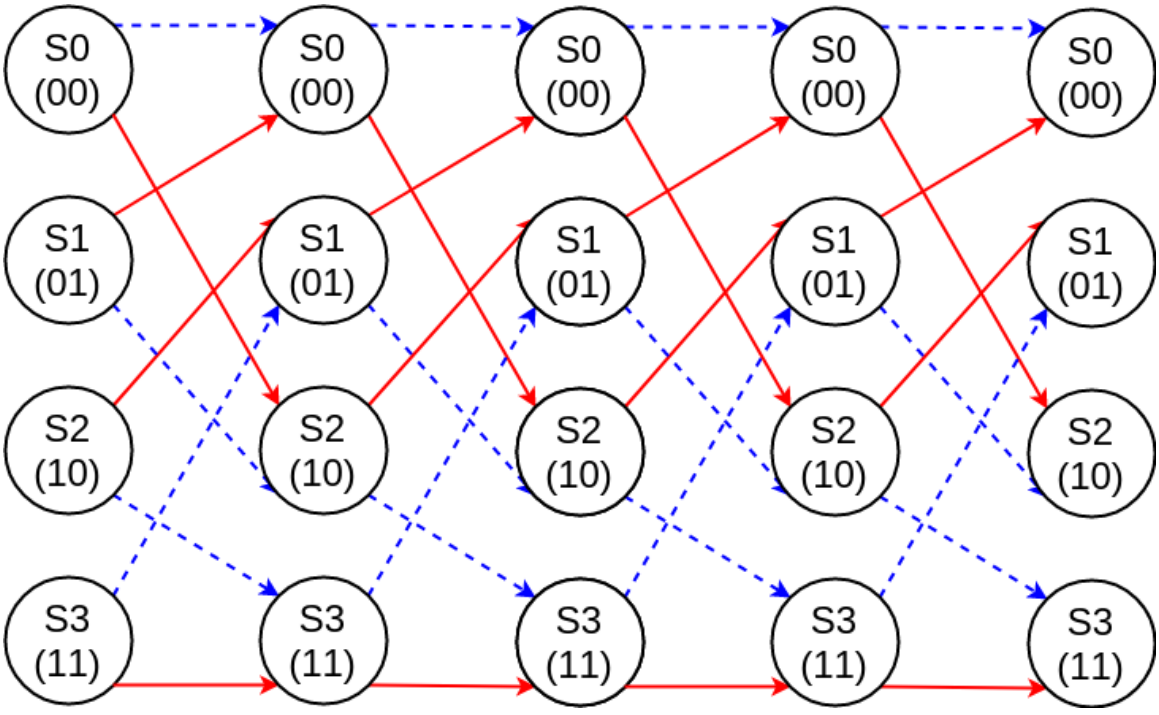

Figure 3.6: This diagram represents a trellis with 5 states over 4 bits. The red lines represent an input bit of one and the dotted blue lines represent and input bit of zero.

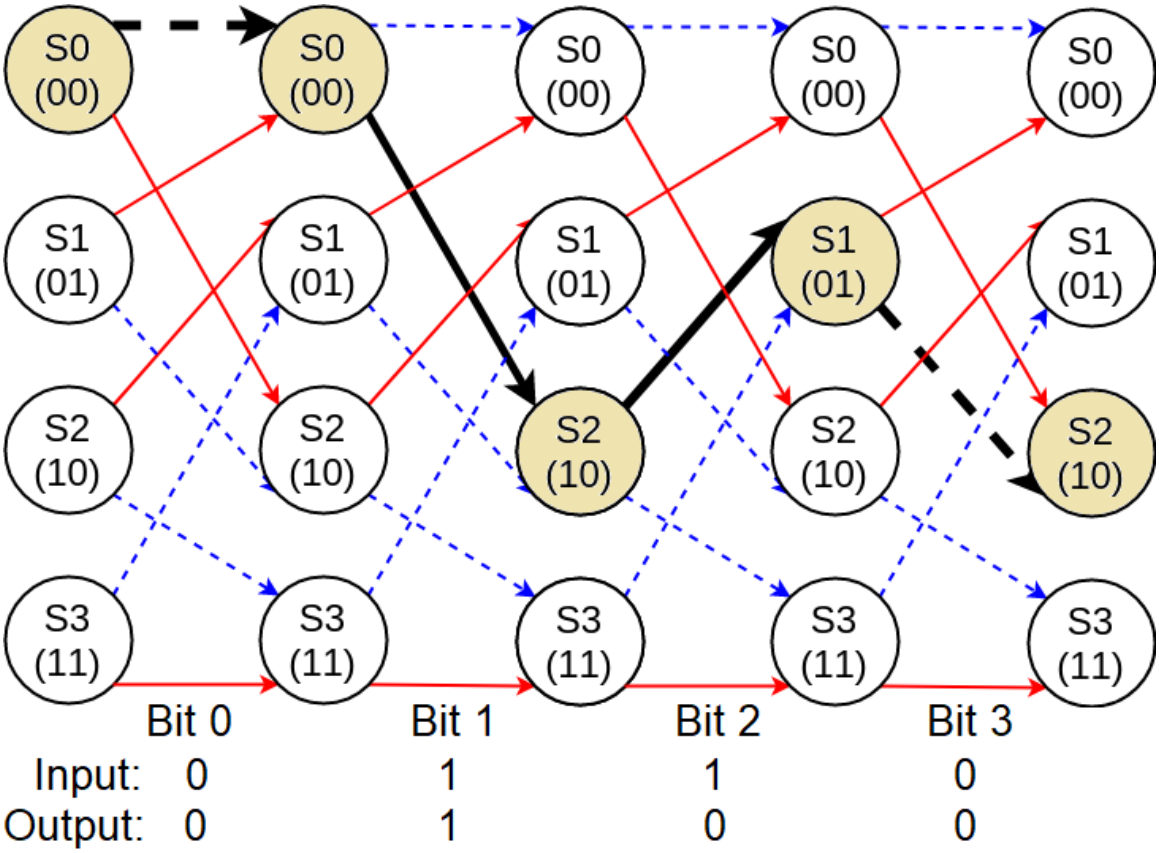

Figure 3.7: This diagram shows an example of how an encoded sequence can be represented with the trellis. The bolded black lines represent a path that the sequence could take.

increases so does the confidence in the output. When the desired number of iterations through the feedback loop is met, both outputs are fed into a hard decision maker which gives the final output.

The MAP decoder makes up the complex part of this algorithm. It constitutes the majority of the decoder, and relies heavily on the trellis structure. A trellis is a time-invariant state machine used to represent an RSC encoder. The truth table for every possible state for the previous RSC encoder can be seen in Table 3.1. A state machine can be extracted from this truth table. The state machine representation can be seen in Figure 3.5. The RSC encoder used has a memory of two bits, or four possible states, as seen in the figure. This state machine over time makes up the trellis. The trellis used for this example can be seen in Figure 3.6. Examining the figure shows the four states at each time interval, along with the transitions between each time interval. When encoding, a single path is taken through the trellis, where a new parity bit is produced at each step. An example of this is displayed through Figure 3.7. The MAP decoder uses this trellis structure which represents how the input sequence was encoded, and tries to find the path taken using the systematic sequence and the parity sequence.

The MAP decoder calculates the Log Likelihood Ratio (LLR) for each bit, or in other words the probability a bit is a 0 or a 1. The LLR is calculated as

$$LLR(u_k|y) = \log \frac{P(u_k = 1|y)}{P(u_k = 0|y)}, \tag{1}$$

here $P(u_k = 1|y)$ is the probability that the bit u_k is 1 given the entire information sequence, y, has been received and where $P(u_k = 0|y)$ is the probability that the bit u_k is 0 given the entire information sequence, y, has been received. This can be

Table 3.1: Truth table representation of the RSC encoder shown previously.

Input	Memory 1	Memory 2	State	A = Memory 1 XOR Memory 2	B = Input XOR A	Output = Memory 2 XOR B
0	0	0	0	0	0	0
0	0	1	1	1	1	0
0	1	0	2	1	1	1
0	1	1	3	0	0	1
1	0	0	0	0	1	1
1	0	1	1	1	0	1
1	1	0	2	1	0	0
1	1	1	3	0	1	0
0	0	0	0	0	0	0
0	0	1	1	1	1	0
0	1	0	2	1	1	1
0	1	1	3	0	0	1
1	0	0	0	0	1	1
1	0	1	1	1	0	1
1	1	0	2	1	0	0
1	1	1	3	0	1	0

expressed using three terms as follows

$$LLR(u_k|y) = \ln \frac{\sum(\alpha_{k-1}(s')\gamma(s', s)\beta_k(s))}{\sum(\alpha_{k-1}(s')\gamma(s', s)\beta_k(s))}, \tag{2}$$

where $\alpha_{k-1}(s')$ is the probability that the trellis is in state s' at $t = k - 1$, starting from $t = 0$ and moving forward in time, where $\beta_k(s)$ is the probability that the trellis is in state s at $t = k$, starting from $t = N$ and moving backwards in time, and where $\gamma_k(s', s)$ is the probability that if the trellis is in state s' at time $t = k - 1$, it moves to state s at $t = k$. The γ terms are branch transition probabilities known as the branch metrics. The α terms are computed recursively moving forward in time, called forward recursion, and the β terms are computed recursively moving backward in time, called backward recursion.

These terms can be understood conceptually with the help of the trellis structure.

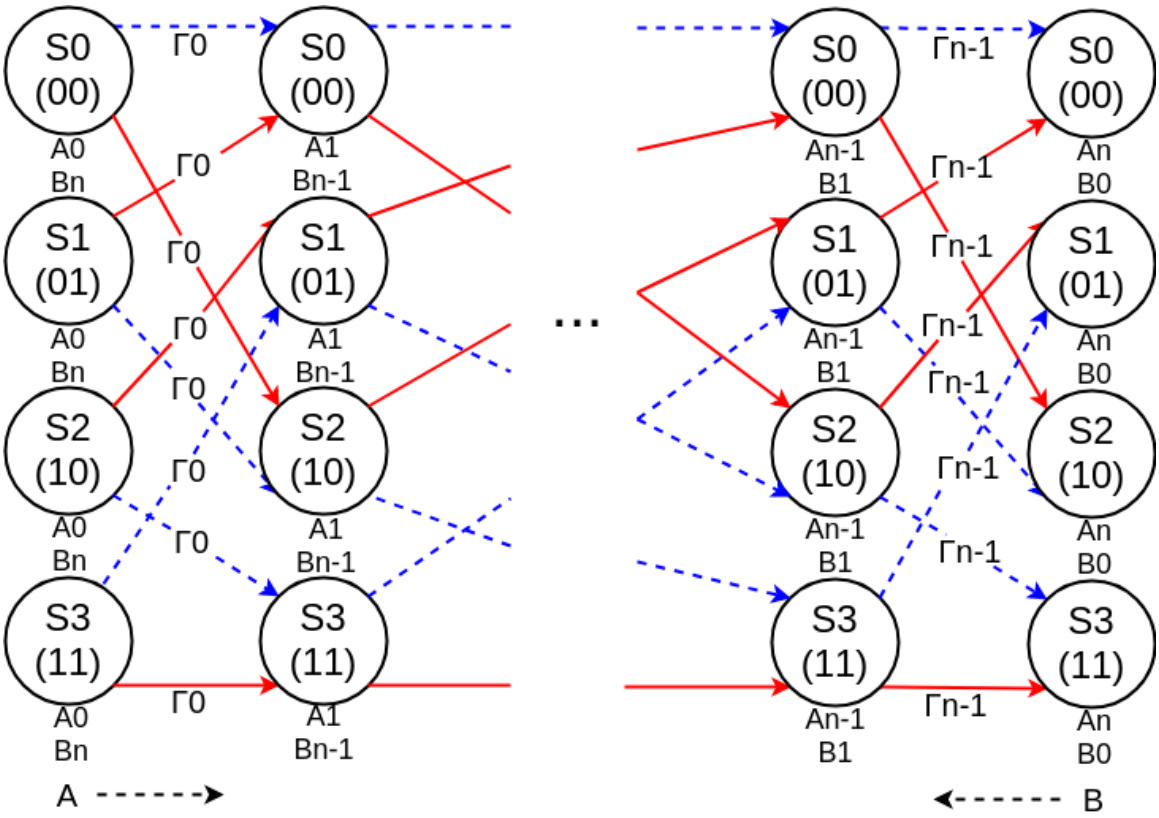

Figure 3.8: This diagram shows how the Gamma, Alpha, and Beta terms are represented in the trellis. The Gamma terms are the state transitions represented by the Γ_n in the diagram. The Alpha terms are denoted by A_n and move left to right. The Beta terms are denoted by B_N and move right to left.

Figure 3.8 displays each of the terms by labeling pieces of the trellis. In the figure the gamma terms are represented by the transitions between the states, while the Alpha terms are represented by the states going forward, and the Beta terms are represented by the states going backwards.

The branch metrics or gamma terms are calculated

$$\gamma_k(s', s) = C_k \exp\left(\frac{1}{2}(L_c y_k u_k + u_k L(u_k) + L_c \sum_{i=1}^{n} y_{ki} x_{ki})\right), \tag{3}$$

where u_k is the trellis input, $L(u_K)$ is the extrinsic information from the previous MAP decoder, L_c is the channel reliability, n is the number of bits being considered (in this case 2 for the systematic bit and parity bit), y_k is the bit received (systematic

bit or parity bit), x_k is the trellis output, and C_k is a constant that gets divided out later and can be ignored.

The forward recursion or alpha terms are calculated

$$\alpha_k(s) = \alpha_{k-1}(s')\gamma_k(s', s), \tag{4}$$

where α_0 is initialized

$$\alpha_0(s) = \begin{cases} 1, & \text{if } s = 0 \\ 0, & \text{otherwise.} \end{cases}$$

The backward recursion or beta terms are calculated

$$\beta_{k-1}(s)) = \beta_k(s')\gamma_k(s', s), \tag{5}$$

where β_{k-1} is initialized

$$\beta_{k-1}(s) = \begin{cases} 1, & \text{if } s = 0 \\ 0, & \text{otherwise.} \end{cases}$$

The MAP decoding algorithm requires long, resource intensive exponential functions, and a large number of multiplication functions. Improvements using the log domain can greatly reduce the complexity without sacrificing any accuracy [10]. Using the log domain, exponentials can be removed, the large amount of multiplications can become additions, and additions can become the max* operation seen in equation 3.

$$\max *(a, b) = \max(a, b) + ln(1 + e^{-|a-b|}), \tag{6}$$

This variation is commonly referred to as the MAX-LOG-MAP algorithm. Further simplifications can be made while still maintaining exceptional results by using the MAX operation and a correction factor, where this correction factor is stored in

a small 8 value look up table[25]. The decoder can be simplified even further by ignoring the correction factor entirely. This is commonly referred to as the MAX-MAP algorithm. For simplicity, the MAX-MAP variant of the algorithm was used for the FPGA acceleration.

The Γ terms in the log domain can be computed as follows

$$\Gamma_k(s', s) = \frac{1}{2} u_k L(u_k) + \frac{L_c}{2} \sum_{i=1}^{n} y_{ki} x_{ki} \tag{7}$$

The alpha terms in the log domain can be computed as follows

$$log(\alpha_k(s)) = A_k(s) = \max_{s'} *(A_{k-1}(s') + \Gamma_k(s', s)), \tag{8}$$

where A_0 is initialized

$$A_0(s) = \begin{cases} 0, & \text{if } s = 0 \\ -\infty, & \text{otherwise.} \end{cases}$$

The beta terms in the log domain can be computed as follows

$$log(\beta_{k-1}(s)) = B_{k-1}(s) = \max_{s'} *(B_k(s') + \Gamma_k(s', s)), \tag{9}$$

where B_{k-1} is initialized

$$B_{k-1}(s) = \begin{cases} 0, & \text{if } s = 0 \\ -\infty, & \text{otherwise.} \end{cases}$$

Upon completion of all of three of these terms, the extrinsic information or LLR can be calculated in the log domain as

$$L(u_k|y) = \max_{(s',s) \text{for } u_k=1} *(A_{k-1}(s') + \Gamma_k(s', s) + B_k(s))$$

$$- \max_{(s',s)\text{for } u_k=0}{}^{*} (A_{k-1}(s') + \Gamma_k(s', s) + B_k(s)), \tag{10}$$

Again, this extrinsic information is used in subsequent decoders to improve the reliability of the error correction.

The outputs of the MAP decoders are fed to the hard decision maker.

$$u_k = \begin{cases} 1, & \text{if } L(u_k|y)_1 + L(u_k|y)_2 + L_c * u_k > 0 \\ 0, & \text{otherwise.} \end{cases}$$

The two outputs are added together along with the systematic bit multiplied by the channel reliability. If the value is positive it is considered a 1 otherwise it is a zero. The equation can be seen in the Equation above.

3.1.3 Implementations

There are many variants of the MAP decoder which have implementation advantages. Each of the variants were implemented one by one. Initially, the regular MAP decoder was implemented in software. The psuedocode for the MAP decoder can be seen in Appendix A. Then the LOG-MAP implementation was created, followed by the MAX-LOG-MAP implementation, MAX-LOG-LUT-MAP, and finally the MAX-MAP implementation. The LOG-MAP variant is the same as the MAP decoder but using the log domain. The MAX-LOG-MAP decoder is the same as the LOG-MAP decoder but uses the max* operator in place of the log function. The MAX-LOG-LUT-MAP is the same as the MAP-LOG-MAP but uses a look up table for the correction factor in place of calculating the max* correction factor. Finally, the MAX-MAP is the same as the MAX-LOG-LUT-MAP but ignores the correction factor entirely. The psuedocode for the MAX-MAP decoder can be seen in appendix B.

This MAX-MAP implementation was used as the basis for the remainder of the

research. Supporting code was written to test the turbo code decoder, including the different variants of the MAP decoder. This code produced random sequences of data, created interleaver patterns used for encoding and decoding of data, encoded the data using the turbo code encoder, added Gaussian noise to the signal, ran the turbo code decoder, recorded the decoding time, calculated the BER of the output, and printed the results with their corresponding input parameters. The block size and trellis size were compile time macros, but the testing framework allowed simple iterations through some run time parameters as well. The SNR start, stop, and step amounts, the number of turbo decoder iteration start, stop, and step amounts, and the number of simulations per parameter were all run time specified, allowing for many parameters to be checked in a single run. The turbo code decoder using the MAX-MAP decoder results can be seen in Figure 3.9. This figure clearly shows that performance of the error correction improves with increased iterations and approaches a limit, as expected.

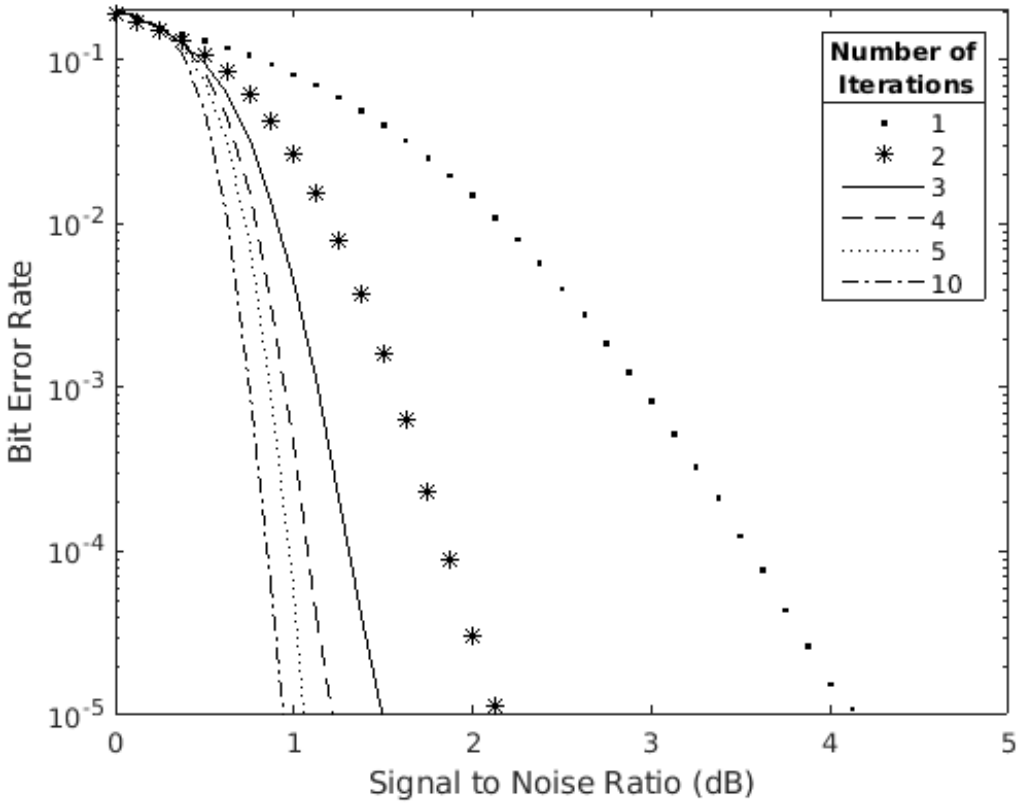

Figure 3.9: Bit Error Rate performance of a MAX-MAP turbo code decoder with a block size of 4000, and a trellis with 8 states.

Chapter 4

High Level Synthesis Guidelines

The goal of HLS tools is to facilitate the design process of hardware implementations on the FPGA by translating a software description of an algorithm into a hardware description language implementation. Taking software designed for sequential execution will likely not produce hardware implementations that are competitive with the ideal manually optimized designs. Instead, to maximize the effectiveness when utilizing HLS tools, it is important to take special care in the design approach, and to develop the software implementation with a hardware-centric mindset. This involves starting with a hardware design in mind and writing code that attempts to emulate it. Additionally, it involves adapting the coding style to the HLS requirements, which is not always in agreement with software best practices. In this work, important aspects of this design process are identified in order to use the HLS tool effectively.

One of the goals of this research was to identify the necessary aspects of HLS that lead to hardware comparable in terms of performance to manually optimized designs. Three different points of interest were identified to be very important and can be seen as guidelines that lead to good implementation.

1. A good understanding of the tools.

2. Having a coding style that is effective for HLS.

3. Starting with a hardware design in mind and writing code that attempts to

emulate it.

This section will discuss these point at a general level and provide some examples. After this, the implementation of the algorithm is discussed in detail.

4.1 Understand The Tool Offerings

An understanding of what is offered from the tools, and how the tools work in terms of synthesis, are essential elements necessary to take advantage of HLS. In the case of Vivado HLS that means understanding the available implementation choices such as directives, data types, and interfaces, and when to use them. It also means being able to read and understand HLS reports, and being able to analyze the output using the different analysis tools provided.

4.1.1 IDE Features

Vivado HLS has many features that can greatly enhance productivity and shorten development time. Some of these features include analysis tools which explain the mapping between the high level code and cycle by cycle operation, resource usage, timing information, and different simulation methods.

4.1.2 Directives

Arguably the most significant aspect of Vivado HLS to understand is the directives. As mentioned previously, using HLS to compete with hand coded RTL implementations is not as simple as selecting a function to synthesize. Part of the reasoning behind this is that there are many things that can be done in hardware that cannot be expressed in high level code alone. To accommodate for this HLS tools can be provided with directives. *Directives* are extensions of C and C++ that guide the tools to different hardware implementations. Several directives were identified to be

very impactful in HLS hardware designs, and thus greatly affecting performance and resource utilization:

- *Pipelining* directive allows fine grained parallelism by increasing utilization of hardware. It is used to pipeline loops, allowing future iterations of for loops to start before the previous ones have finished, increasing utilization and throughput. Pipelining isn't always free however. When pipelining a loop, all sub loops and sub functions may be unrolled completely which can dramatically increase resource usage.

- *Dataflow* directive allows coarse grained parallelism across functions or loops which increasing utilization of hardware. Dataflow will place a FIFO buffer between functions or between loops enabling subsequent functions or loops to receive and work with data before the previous function or loop has finished.

- *Loop unrolling* allows parallelism in loops by replicating hardware. This can be very effective, allowing single cycle computations, but can be costly in terms of resources.

- *Array partitioning* reduces access contention by splitting an array at the hardware level, which provides more parallel ports for access. The Array Partitioning directive requires intimate knowledge of the hardware design, and data access patterns, but without it severe penalties can be faced. For large arrays, block RAM is most commonly used as storage. Although this is often the appropriate method to use, block RAM only has two ports per block that can be used data access each cycle. This can be a huge bottleneck in a design. Partitioning splits the arrays among different blocks, allowing for more ports to access the memory at once. Arrays can be partitioned completely allowing for fully parallel access, but again comes at the cost of resources as it is implemented using FFs and LUTs.

- *Inlining* removes the hierarchy of the function call which in turn allows the tools more freedom to optimize. This can be quite effective for increasing performance, but can be more confusing for analysis.

- *Interface* specifies how the ports are created. This is a very significant directive when it comes to system integration as it directs how the data is passed into and out of the accelerator, and how it is interacted with.

- *Dependence* provides the tools with more information enabling false dependencies to be removed, which allows pipelining or improved pipelining. The tools choose functionality over performance and thus sometimes see dependencies that are not required. Eliminating those false or not required dependencies can enhance pipelining performance.

4.1.3 Data Types and Libraries

Xilinx provides arbitrary precision libraries which let the user have more control over data types. With these libraries a user can select the bit widths of data types, which can be very useful for minimizing resources, or creating data types larger than can be specified using native C or C++ types. These libraries are also useful for implementing fixed point data types. Fixed point data types are common in hardware designs as replacements for floating point as they can be calculated much faster and more efficiently.

In addition to that there are also other libraries provided by Xilinx that can be leveraged for effective implementations. Some of these libraries include DSP, Math, Linear Algebra, Streaming, and Video.

4.2 Coding Style

When writing the code for the hardware it is important to have a coding style tailored to hardware. In other words, when implementing a design it is best not to have a software perspective in mind, but instead think about what hardware will be created from the high level code. For example in software, it is good practice to avoid branches when possible as branch penalties can play a huge factor in slowing performance. Branches behave differently when consider for HLS. Using HLS, an *if* statement will usually be interpreted as a multiplexor in hardware. Multiple multiplexors or larger multiplexors do not face the same penalties as branch penalties. It is this kind of thinking, and understanding of what hardware the tools will interpret from the code that will guide good coding style and implementations. This can be a difficult mindset to adjust to at first, but is vital for good results. Along with this, there are many restrictions on coding styles as some things are not synthesizable. Some of the unsynthesizable aspects include operating system calls, STL functions, function pointers, and pointers without compile time size definitions. In addition to this recursion is not allowed, and pointer casting is very limited.

There are several ideas that come to mind for guiding good code. Code should be explicit and well defined. The tools will not optimize the hardware unless it can prove optimization will not break functionality. Being explicit provides more information and allows more optimizations to occur. Also keeping the control flow simple and avoiding complex code will produce better results. More complex control flow leads to higher resource usage requirements as well as longer latency delays. Understanding the building block of the FPGA such as FlipFlops, LUTs, DSP48s, or block RAM, and coding to take advantage of their properties can be beneficial as well.

A few examples taken from [26] of how to take advantage of these ideas in coding style can be seen below.

4.2.1 Bounded Loop Iterators

Bounding loops with a maximum number of iterations and using break statements for early termination will allow the tools to run optimizations that arent possible with dynamic for loops. Again the tools do not break functionality, so if a variable is specified as an 32 bit int but it only ever loops a maximum of 200 times, depending on how its written, the tools may not be able to prove that the counter never goes over 200 and thus will unnecessarily use 32 bit variables to count. These longer 32 bit variables take up more resources and take longer to evaluate than 8 bit variables. With perfectly defined loops the bit widths of the data path and control signals can be optimized.

Listing 4.1: Example code which does not provide definitive boundaries for the for loop resulting in subpar HLS results.

```
1 #define MAX 200
2 int total(int in_array[MAX], int size){
3   int total=0;
4   for(int i=0; i<size; i++){
5     total = total + in_array[i];
6   }
7   return total;
8 }
```

Listing 4.2: Example code with bounded loops and an early conditional break statement which allows the HLS to optimize the implementation.

```
1 #define MAX 200
2 int total(int in_array[MAX], int size){
3   int total=0;
```

```
4    for(int i=0; i<MAX; i++){
5      total = total + in_array[i];
6      if (i == size) {
7        break;
8      }
9    }
10   return total;
11 }
```

4.2.2 Being Explicit Where Possible

This bring about the point of being explicit and allowing the tools to make optimizations. For example, although the tools will interpret if statements as multiplexors, it does not mean if statements should be used without consideration. The tools will optimize where possible by sharing resources. In the example below there are two if statements that could be written using if else instead. Using two separate if statements will duplicate the foo hardware as the tools cannot prove the paths are mutually exclusive. Figure 4.1 shows a simplified idea of potential hardware that could be created from the description. As can be seen in the figure, the foo hardware is duplicated.

Listing 4.3: Example code disregarding potential mutual exclusion which restricts the tools from optimizing.

```
1 if (A == Value1)
2   A = Foo(X);
3 if (B == Value2)
4   B = Foo(Y);
```

Written this way, mutual exclusion cannot be proven, and potential optimizations such as resource sharing go untouched.

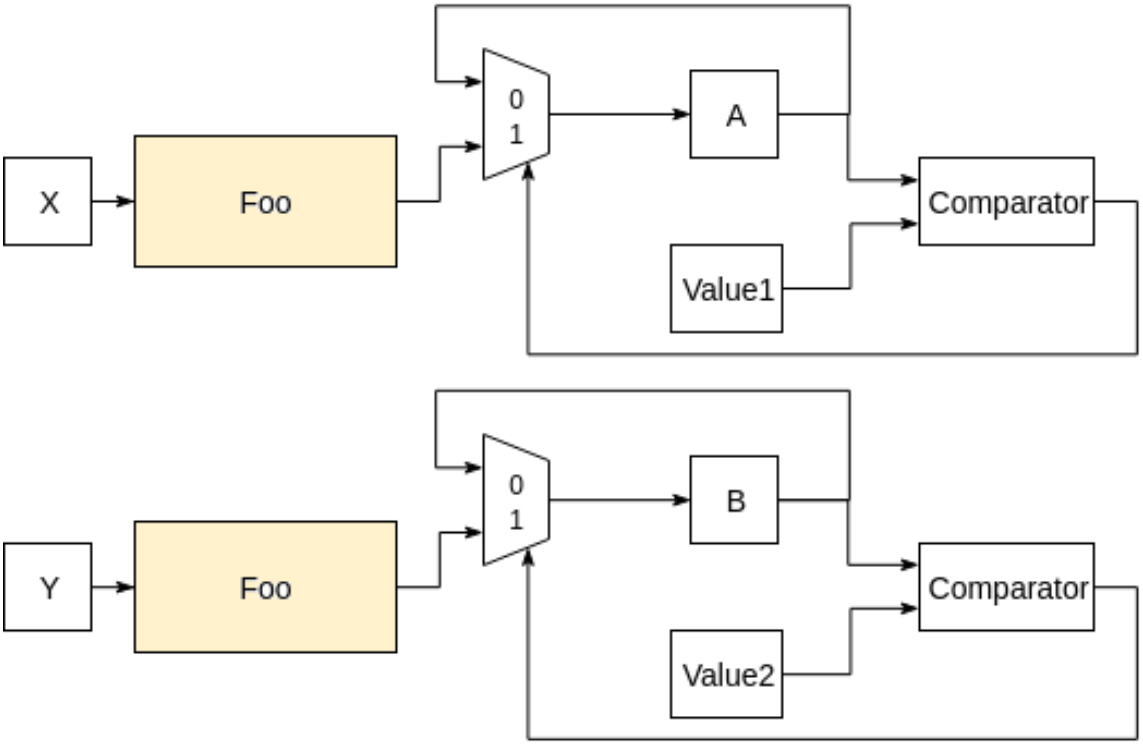

Figure 4.1: Illustration of the hardware interpretation when mutual exclusion is not proven. In this case two separate foo units are required.

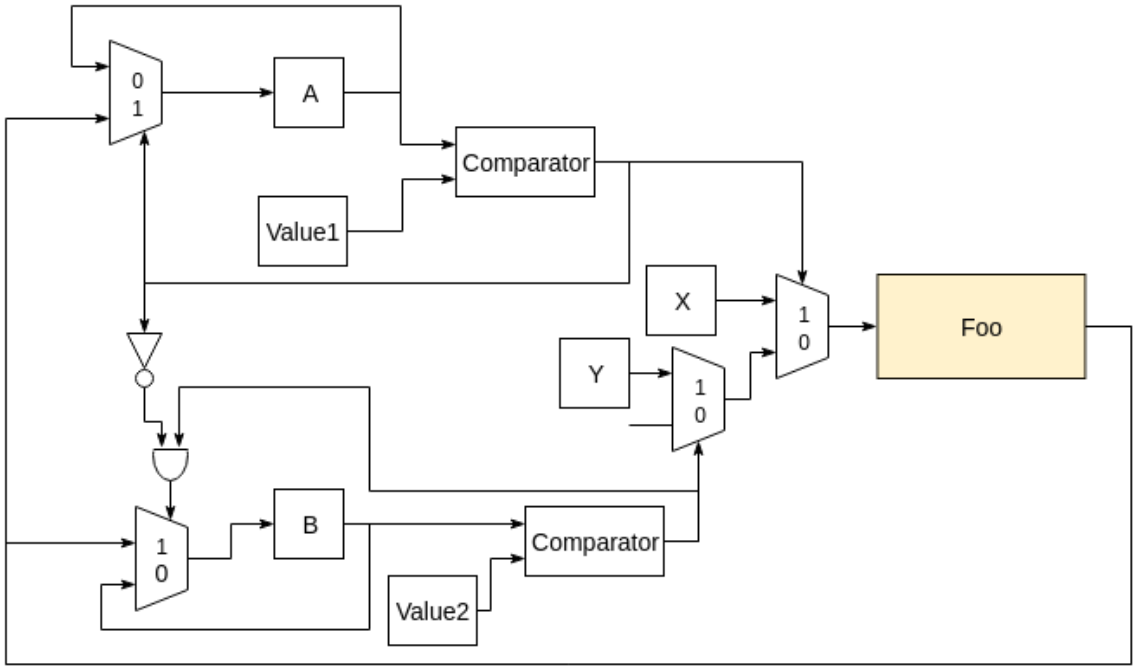

Figure 4.2: Illustration of the hardware interpretation when mutual exclusion is proven. In this case one foo unit can be shared.

The following listing provides more information that can be taken advantage of to enhance hardware implementation. A simplified idea of the hardware that could be created from this high level code can be seen in Figure 4.2. This figure has a more complicated control path but enables a single foo hardware unit to be implemented which may be very beneficial.

Listing 4.4: Example code being explicit and displaying mutual exclusion which allows the tools to optimize further. In this case by sharing resources.

```
1 if (A == Value1)
2   A = Foo(X);
3 else if (B == Value2)
4   B = Foo(Y);
```

Written this way, mutual exclusion is proven, and resource sharing optimizations can be made. This a very simple example but the principal of being explicit can be

applied more widely in design.

4.2.3 Single Return Point

Another good design practice is using a single return point. This can lower the complexity of the control path, and allows for the pipeline stages to be balanced better. This is beneficial for resource usage, latency, and debugging. If there is more than one possible exit point, flags can be used to skip extra computation. The example below first shows code that can return early which should try to be avoided for HLS. After this an alternative method of using flags is shown.

Listing 4.5: Example code showing multiple return points in a function which should be avoided for good HLS coding style.

```
1 if (exit_early == true) {
2   return A;
3 }
4 A = A + B;
5 return A;
```

Listing 4.6: Example code showing how flags can be used to easily bypass sections of code and cleanly create a single return point.

```
1 flag_add_B = true;
2 if (exit_early == true) {
3   flag_add_B = false;
4 }
5 if (flag_add_B == true) {
6   A = A + B;
7 }
8 return A;
```

Although this code can be simplified further, the coding style of using flags to bypass sections of code has shown to be effective for more complex examples.

4.3 Write Code That Emulates Hardware

One of the most significant aspects for HLS implementations that achieve near hand coded performance is writing code that tries to emulate a hardware design. This is very important as this type of coding doesn't always make sense from a software standpoint but is vital for good results. Software that achieves the same functional equivalency can likely be completed in a more straightforward implementation with fewer lines of code, but gives very different hardware results. For example increasing parallelism at the hardware level can lead to code which has a complex control flow, and calls the same function multiple times, which has no software benefit. Additionally, this may introducing extra variables or extra buffers that are unnecessary in software.

One example of this can be seen in a simplified case that was encountered through this work. In this case, dataflow and pipelining were not applicable due to the nature of the dependencies. The software functionality is capable of being expressed as shown in the listing below. The subsequent hardware created can be seen in Figure 4.3. The figure demonstrates how the code will produce hardware that runs sequentially.

Listing 4.7: Example code showing how hardware performance can be limited due to data dependencies on a single buffer.

```
1 for (LOOP) {
2     ACQUIRE_DATA(A);
3     USE_DATA(A);
4 }
```

Instead of this hardware, it was desired that the data acquisition and data usage

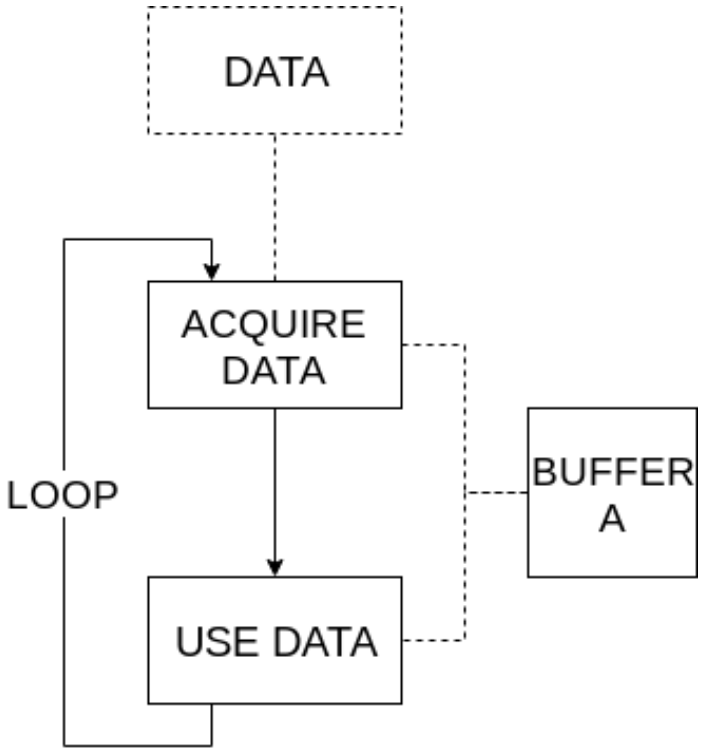

Figure 4.3: Example of resulting hardware from listing 4.7

occur in parallel to increase performance. To do this, a pingpong buffer scheme could be used. In this case, the acquire_data function would use one buffer, while the use_data buffer would use another buffer, and they would swap buffers for each loop. Code to represent this hardware can be seen in the listing below. The hardware representing of this can be seen in Figure 4.4. This figure clearly shows the two functions can run in parallel by removing the dependencies between them.

Listing 4.8: Example code showing how parallel functions could be used to reduce the memory access limitations.

```
1 //Initialization
2 ACQUIRE_DATA(B);
3 Toggle = true;
4
5 for (LOOP) {
```

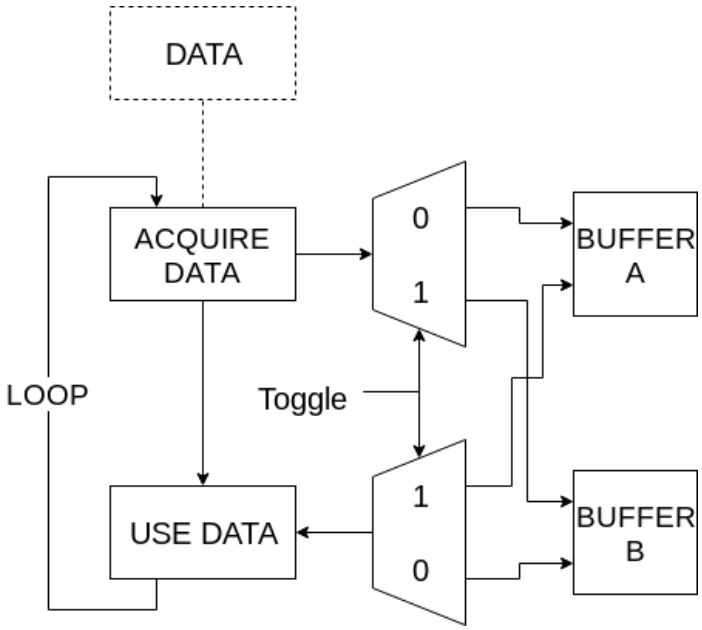

Figure 4.4: Example of resulting hardware from listing 4.8

```
6    if Toggle {
7      ACQUIRE_DATA(A);
8      USE_DATA(B);
9    }
10   else {
11     ACQUIRE_DATA(B);
12     USE_DATA(A);
13   }
14   Toggle = !Toggle;
15 }
16 //cleanup
```

As can be seen when comparing both code listings, the pingpong buffer scheme adds complexity to the code which would not add any benefit during software execution. The hardware on the other hand would be significantly different, gaining the ability for the functions to occur in parallel. This is only a simple example, but emphasizes the importance of writing the software to describe the desired hardware.

Chapter 5

HLS MAX-MAP Implementation

The turbo code decoder is the complex part of turbo code error correction. Within the turbo code decoder, the complex, compute intensive piece of the algorithm is the MAP decoder. This was chosen to undergo FPGA acceleration. In this case, the MAX-MAP variant of the MAP decoder described previously, was used as it contains many benefits for hardware implementation. This chapter describes the implementation details of the MAX-MAP decoder for targeting HLS. The high level hardware designs are explored first, followed by the low level building blocks that made up the implementation.

5.1 High Level Designs

The explanation of the implementation is easiest to comprehend from the top-down, looking at the high level design approach. Furthermore, starting with the software implementation, and moving through the hardware designs provides insight into the thought process that was behind the implementations, and demonstrates the ability to easily explore the design space. Ultimately, there were three different hardware designs that were created, building off what was learned from previous implementations.

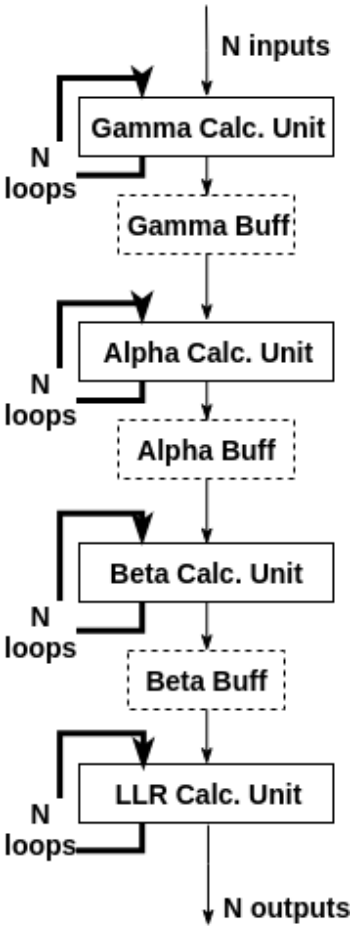

Figure 5.1: The high level flow of data for the original MAP decoder. The N inputs represents the entire block of information received. Each of the terms are calculated separately in their entirety before continuing to the next calculations.

5.1.1 Software

At the topmost level, the block diagram software implementation can be seen. The regular software flow of data through a MAX-MAP decoder can be seen in Figure 5.1. This figure shows the algorithm being broken down into 4 main functions, each one representing one of the terms of the algorithm. In this implementation, each of these calculations occurs for the entire N inputs, which typically ranges from 40 to 6144 as specified in the LTE standard [27], before starting the next calculations. This requires the entire input block to be received before continuing, and requires all of the calculation results to be stored.

Listing 5.1: Pseudo Code for the original software design.

```
MAP_decode(systematic[NUM_BITS],
           parity[NUM_BITS],
           extrinsic[NUM_BITS],
           noise,
           decoder_output[NUM_BITS])
{
    calculate_gamma(systematic[NUM_BITS],
                    parity[NUM_BITS],
                    extrinsic[NUM_BITS],
                    noise,
                    gamma_output[NUM_BITS]);

    calculate_alpha(gamma_output, alpha_output);

    calculate_beta(gamma_output, beta_output);

    calculate_LLR(gamma_output,
                  alpha_output,
                  beta_output,
                  decoder_output);
}
```

5.1.2 Hardware Design 1: Initial Sliding Window

The target hardware implementation for HLS uses a sliding window approach, which is common for hardware designs. In this case, a sliding window approach means only a number of bits are considered at one time, starting with the first bits and moving to the later bits, rather than using the entire sequence at once. A visual example of this using a trellis structure can be seen in Figure 5.2. Other than allowing for good hardware implementation, the siding window approach has many benefits to

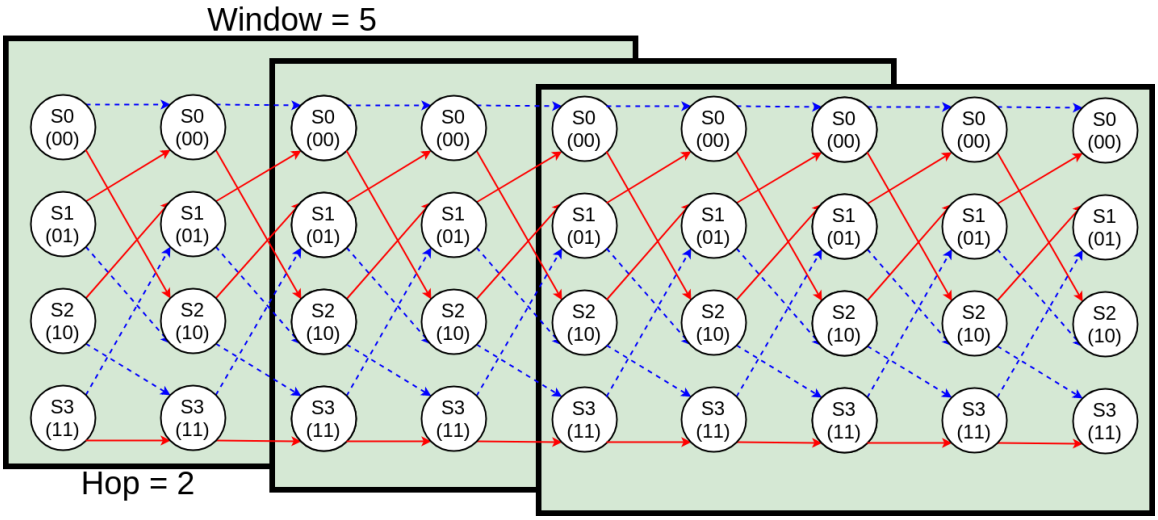

Figure 5.2: The sliding window approach shown on a trellis structure. In this example a window of 5 and a hop of 2 is used.

the algorithm itself. On one hand, the Beta values can start to be calculated before all of the bits are received, and thus before all the Gamma and Alpha calculations are completed. Because of this, continuous streams of data are possible, instead of requiring whole blocks at a time. Additionally, this approach allows for more parallelism, and less resource usage as less information is needed at one time. This approach is made possible by initializing the Beta values with either equal probability of being in each state, or the most recently calculated Alpha values for each state. This design takes a slight hit in accuracy as the starting state of the Beta values is unknown. To reduce the effect of this imperfect initialization, the first Beta values calculated are thrown away as they are not as accurate as the later values. They are recalculated again later, requiring extra

The high level design of the sliding window implementation can be seen in Figure 5.3. It is important to note that the figure ignores the initialization and cleanup required for the sliding window approach, but is required for the actual implementation. A significant aspect to the design is that to reduce the number of extra calculations, instead of having the window slide and produce one output per frame, it hops by H and produces H outputs for frame. The figure shows an overarching loop with

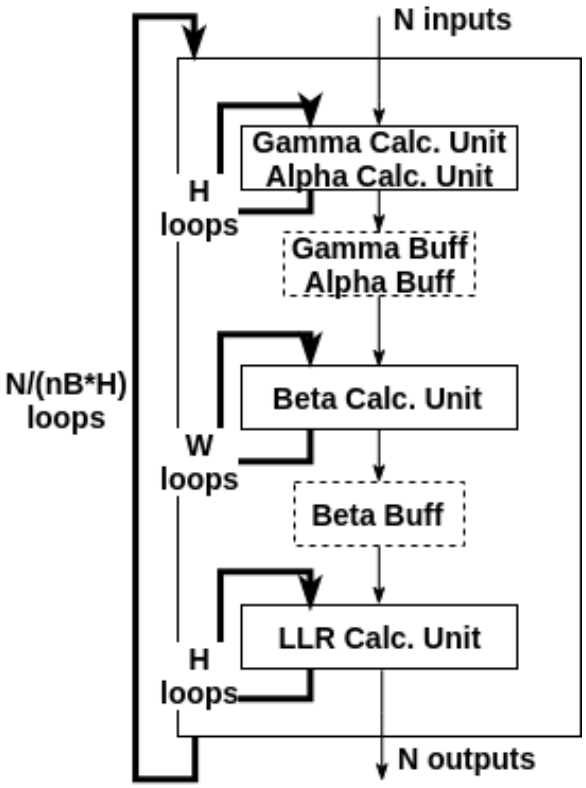

Figure 5.3: The high level hardware design for the initial sliding window approach. N represents the number of inputs in the block, W represents the number of inputs in the window, and H represents the amount to hop by for each window. Each of the inner loops is pipelined in this design.

nested loops inside of it. Each iteration will produce H results, while N/H iterations will produce outputs for all of the N inputs. In this design, the Gamma and Alpha Calculation Units are combined as they can occur concurrently. These terms do not require any extra calculations and thus loop H times. The results are then stored in a buffer, where the Gamma results are passed to the the Beta Calculation hardware. The Beta Calculation require the extra calculations and thus loops W times, using only H results for the next hardware. The extrinsic values are then calculated using all three buffers. This hardware loops H times and produces one output per loop. Only the last H calculated Beta values are used in the extrinsic value calculation.

Listing 5.2: Pseudo Code for the original sliding window design.

```
1  MAP_decode(systematic[NUM_BITS],
2             parity[NUM_BITS],
3             extrinsic[NUM_BITS],
4             noise,
5             decoder_output[NUM_BITS])
6  {
7
8      /**************************************
9                  initialization code
10      **************************************/
11
12     for NUM_BITS/WINDOW_HOP {
13
14         // Calculate the new gamma and alpha going forwards in time
15         for WINDOW_HOP {
16 //PIPELINE
17 #pragma HLS pipeline II=xx
18             new_gamma = calculate_gamma(systematic[bit], parity[bit], ↩
                   extrinsic[bit], noise_param, trellis);
```

```
19              bit++;
20              gamma_buffer_write(new_gamma);
21
22              new_alpha = calculate_alpha(last_alpha, gamma_buffer, ↩
                    trellis);
23              alpha_buffer_write(new_alpha);
24              last_alpha = new_alpha;
25          }
26
27          // Calculate beta terms going backwards in time
28          //initialize from the last calculated alpha
29          last_beta = last_alpha;
30
31          //Go through the entire window length for the Beta terms
32          for WINDOW {
33  //PIPELINE
34  #pragma HLS pipeline II=xx
35              new_beta = calculate_beta(last_beta, gamma_buffer, trellis↩
                    );
36              beta_buffer_write(new_beta);
37              last_beta = new_beta;
38          }
39
40          // Calculate Extrinsic LLRs
41          for WINDOW_HOP {
42  //PIPELINE
43  #pragma HLS pipeline II=xx
44              result = calculate_extrinsic(gamma_buffer, alpha_buffer, ↩
                    beta_buffer, trellis);
45              decoder_output_write(result);
46          }
47      } // END for NUM_BITS/WINDOW_HOP
48
```

```
49      /*************************************
50                          cleanup code
51       *************************************/
52  }
```

Special buffers were necessary and are explicitly shown in this design. The buffers only hold one window size of data. This enables the hardware design to use much less resources than storing the entire block of all the terms.

From the analysis of our design, unsurprisingly, the Beta calculations took most computation time. This is due to the multiple times more calculations being required than the Gamma and Alpha calculations. Two different directions were explored from this point.

5.1.3 Hardware Design 2: Replicated Hardware

The second design, shown in Figure 5.4, sought to alleviate the Beta Calculation Unit bottleneck by replicating hardware, enabling parallel Beta Calculation Units. This effectively hides the extra calculations. To accommodate for this approach, larger buffers are required, as well as extra resources for the duplicated hardware. Still, with this design only a single type of calculation unit could run at one time.

In this case, there are nB Beta Calculation Units. This requires nB times the amount of new data to work with each loop. For this reason, the Gamma and Alpha Calculation Units loop nB times as the previous version, represented by nB*H in the figure. The individual Beta Calculation loops remain at W loops as they are independent. Again nB times the amount of data per loop will be produced, requiring the LLR Calculation Unit to loop nB times as the previous version as well, represented by nB*H in the figure. As the inner loops is going through nB times as much data per loop, the outer loop needs to run fewer times to handle the same amount of data as the previous version, represented by N/(nB*H) in the figure.

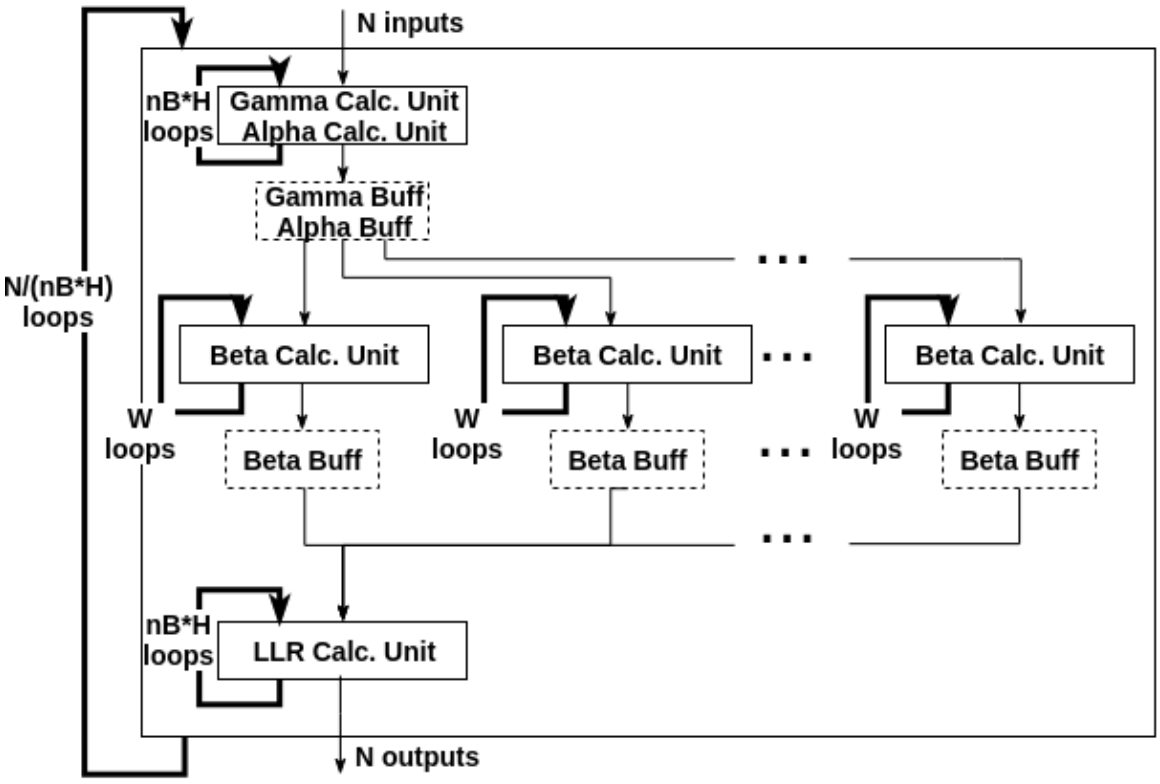

Figure 5.4: The second high level hardware design. This is the same as the previous design except with multiple parallel Beta Calculation Units. In this diagram nB represents the number of parallel Beta Calculation Units, N represents the number of inputs in the block, W represents the number of inputs in the window, and H represents the amount to hop by for each window. Again each of the inner loops is pipelined in this design.

Listing 5.3: Pseudo Code for the second sliding window design with two Beta Calculation Units.

```
1  MAP_decode(systematic[NUM_BITS],
2             parity[NUM_BITS],
3             extrinsic[NUM_BITS],
4             noise,
5             decoder_output[NUM_BITS])
6  {
7
8      /**************************************
9                  initialization code
10      **************************************/
11
12     for NUM_BITS/(WINDOW_HOP*NUM_BETA_CALCULATION_UNITS) {
13
14         // Calculate the new gamma and alpha going forwards in time
15         for WINDOW_HOP*NUM_BETA_CALCULATION_UNITS {
16 //PIPELINE
17 #pragma HLS pipeline II=xx
18             new_gamma = calculate_gamma(systematic[bit], parity[bit], ↩
                   extrinsic[bit], noise_param, trellis);
19             bit++;
20             gamma_buffer_write(new_gamma);
21
22             new_alpha = calculate_alpha(last_alpha, gamma_buffer, ↩
                   trellis);
23             alpha_buffer_write(new_alpha);
24             last_alpha = new_alpha;
25         }
26
27         // Calculate beta terms going backwards in time
28         //initialize from the last calculated alpha
```

```
29          last_beta = last_alpha;
30
31          //Go through the entire window length for the Beta terms
32          for WINDOW {
33  //PIPELINE
34  #pragma HLS pipeline II=xx
35              new_beta1 = calculate_beta(last_beta1, gamma_buffer, ↩
                    trellis);
36              beta_buffer1_write(new_beta1);
37              last_beta1 = new_beta1;
38
39              new_beta2 = calculate_beta(last_beta2, gamma_buffer, ↩
                    trellis);
40              beta_buffer2_write(new_beta2);
41              last_beta2 = new_beta2;
42          }
43
44          // Calculate Extrinsic LLRs
45          for WINDOW_HOP {
46  //PIPELINE
47  #pragma HLS pipeline II=xx
48              result1 = calculate_extrinsic(gamma_buffer, alpha_buffer, ↩
                    beta_buffer1, trellis);
49              decoder_output_write(result1);
50
51              result2 = calculate_extrinsic(gamma_buffer, alpha_buffer, ↩
                    beta_buffer2, trellis);
52              decoder_output_write(result2);
53          }
54      } // END for NUM_BITS/WINDOW_HOP
55
56      /**************************************
57                      cleanup code
```

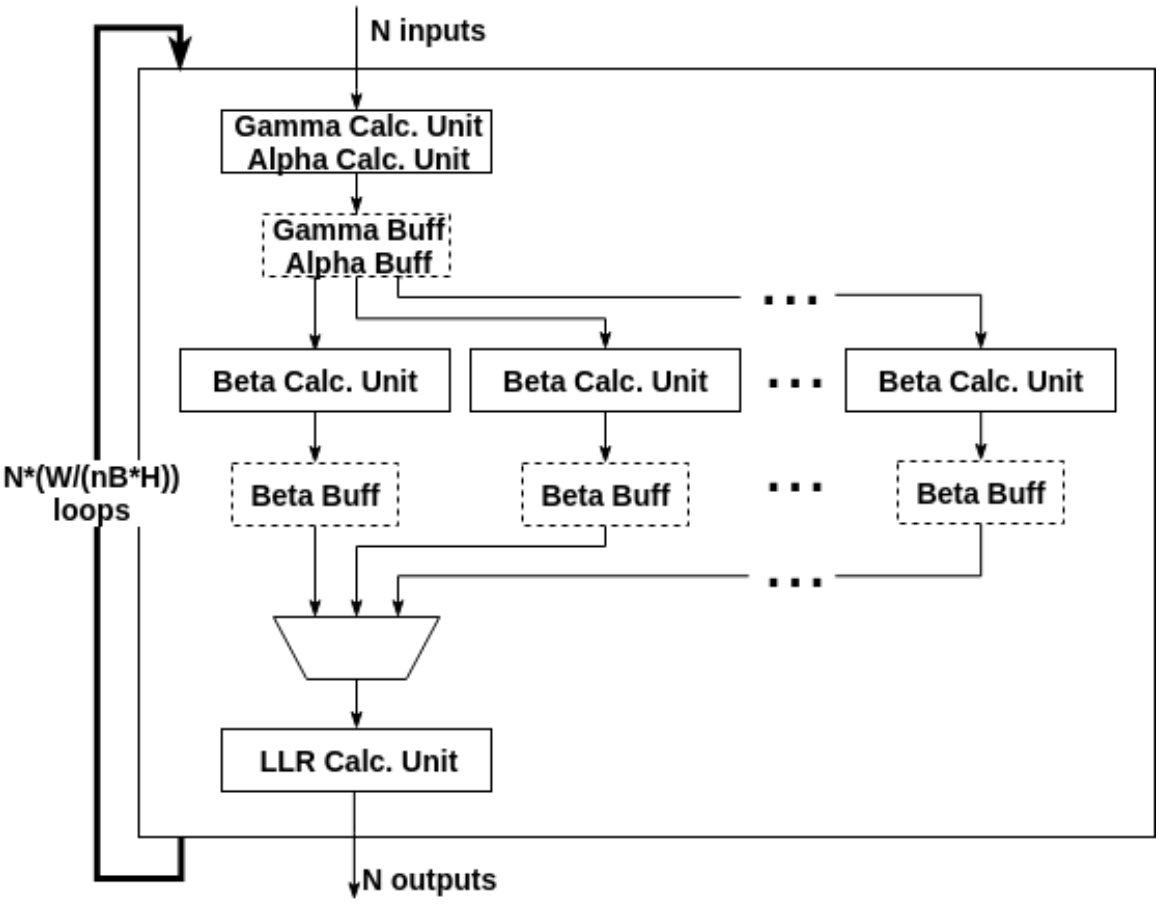

Figure 5.5: The third high level hardware design. This design also has parallel Beta Calculation Units but is different in that the entire design is pipelined instead of just the inner loops so each of the calculation units have the ability to run in parallel. In this diagram nB represents the number of parallel Beta Calculation Units, N represents the number of inputs in the block, W represents the number of inputs in the window, and H represents the amount to hop by for each window.

```
58      ****************************************/
59  }
```

5.1.4 Hardware Design 3: Fully Pipelined

A key feature of any hardware design is to utilize the hardware as much as possible. One limitation of the second design is that while one calculator is going, the others sit idle. An ideal design would be able to have complete hardware utilization at all times. With this in mind, the last design, shown in Figure 5.5, sought to have more utilization

by combining each of the calculation units into a single pipeline rather than pipelining on the individual calculation unit level. The figure shows only one looping structure which allows the entire design to be pipelined. To achieve this the Beta Calculation Units are required to have more advanced control logic which determine if and when a Beta Calculation Unit should run, and keep track of the indexes into the Gamma and Alpha buffers. In addition to this the LLR Calculation Unit requires extra control logic which determines which Beta buffers and associated Gamma and Alpha values to process each loop. The total loops in this case is $N\ inputs * (W/(nB * H))$ where W is the window size, nB represents the number of Beta Calculation Units, and H represents the hop amount. Although this design may seem like the obvious best approach, it may not necessarily turn out to be better. With this design, the pipeline only runs as fast as its slowest calculation unit. If the slowest calculation unit is not running every cycle it could delay potentially faster calculation units unnecessarily, which is the case under some parameters. This is exemplified further in the Results section.

Listing 5.4: Pseudo Code for the fully pipelined sliding window design with 2 Beta Calculation Units.

```
1  MAP_decode(systematic[NUM_BITS],
2             parity[NUM_BITS],
3             extrinsic[NUM_BITS],
4             noise,
5             decoder_output[NUM_BITS])
6  {
7
8      /**************************************
9                  initialization code
10      **************************************/
11
```

```
12      for NUM_BITS-(INITIALIZATION_OVERHEAD+CLEANUP_OVERHEAD) {
13
14          // Calculate the new gamma and alpha going forwards in time
15          for WINDOW_HOP*NUM_BETA_CALCULATION_UNITS {
16  //PIPELINE
17  #pragma HLS pipeline II=xx
18              new_gamma = calculate_gamma(systematic[bit], parity[bit], ↩
                    extrinsic[bit], noise_param, trellis);
19              bit++;
20              gamma_buffer_write(new_gamma);
21
22              new_alpha = calculate_alpha(last_alpha, gamma_buffer, ↩
                    trellis);
23              alpha_buffer_write(new_alpha);
24              last_alpha = new_alpha;
25          }
26
27          // Calculate beta terms going backwards in time
28          //initialize from the last calculated alpha
29          last_beta = last_alpha;
30
31          //Go through the entire window length for the Beta terms
32          for WINDOW {
33              new_beta1 = calculate_beta(last_beta1, gamma_buffer, ↩
                    trellis);
34              beta_buffer1_write(new_beta1);
35              last_beta1 = new_beta1;
36
37              new_beta2 = calculate_beta(last_beta2, gamma_buffer, ↩
                    trellis);
38              beta_buffer2_write(new_beta2);
39              last_beta2 = new_beta2;
40          }
```

```
41
42          // Calculate Extrinsic LLRs
43          for WINDOW_HOP {
44              result1 = calculate_extrinsic(gamma_buffer, alpha_buffer, ↩
                    beta_buffer1, trellis);
45              decoder_output_write(result1);
46
47              result2 = calculate_extrinsic(gamma_buffer, alpha_buffer, ↩
                    beta_buffer2, trellis);
48              decoder_output_write(result2);
49          }
50      } // END for NUM_BITS/WINDOW_HOP
51
52      /**************************************
53                      cleanup code
54       **************************************/
55  }
```

5.2 Design Trade-off Analysis

Initial results while developing the different designs showed the second design with replicated Beta Calculation Units actually performed significantly better than the fully pipelined third design. This was a result was not intuitive at first. With a fully pipelined implementation, calculation units can run in parallel which is not the case with the second design. At this point in development, the Gamma Calculation Units could produce one output every 60 cycles, the Alpha and Beta Calculation Units could produce one output every 40 cycles, and the LLR Calculation Unit could produce one output every cycle. Ignoring initialization, cleanup, filling the pipeline, and other aspects, basic calculations demonstrate the counter intuitive performances.

For a window size of 20 and a hop of 2 with two Beta Calculation Units, to produce

100 outputs design 2 would take:

```
------ DESIGN 2 ------
GAMMA and ALPHA:
2*H inner loops *  N/2H outer loops = N total loops
100 total loops * 60 cycles per loop = 6000 cycles

BETA:
W inner loops *  N/(2*H) outer loops = (N*W)/(2*H) total loops
(100*20)/(2*2) = 500 total loops
500 total loops * 40 cycles per loop = 20000 cycles

LLR:
2*H inner loops *  N/2H outer loops = N total loops
100 total loops * 1 cycle per loop = 100 cycles

TOTAL:
total cycles = 6000+20000+100 = 26100 cycles
```

Ignoring the time it took to fill the pipeline, the design 3 fully pipelined hardware design could loop at a rate of 60 cycles per loop.

For a window size of 20 and a hop of 2 with two Beta Calculation Units, to produce 100 outputs design 3 would take:

```
------ DESIGN 3 ------
TOTAL:
N*(W/(2*H)) loops = 100*(20/(2*2)) = 500 loops
500 total loops * 60 cycles per loop = 30000 total cycles
```

```
------ COMPARE ------
```

```
30000 total cycles is greater than 26100 total cycles.
Design 2 is faster.
```

On the surface it is counter intuitive that the fully pipelined approach is slower than pipelining on the inner level. Looking closer, there are two aspects which create this scenario. The first is that the pipeline can only go as fast as its slowest stage. in this case, the Gamma Calculation Unit as is unable to run faster than 60 cycles per output. Along with that, the window size is 10, the hop size is 2, and the number of Beta Calculation Units is 2. A saturated number of Beta Calculation Units, W/H, in this case $10/2 = 5$ units, would be able to produce a Beta output every loop. As it is not saturated, there are extra loops required due to the extra Beta calculations needed. In these extra loops, there are no Gamma outputs being produced, yet the pipeline remains at 60 cycles per loop, whereas in the previous design the Beta Calculation Unit loops could achieve 40 cycles per loops. These extra cycles in this case have a big impact on overall performance. With a greater number of Beta Calculation Units the fully pipelined design 3 design would achieve a better performance. Calculations for a fully saturated number of Beta Calculation Units can be seen below.

```
------ DESIGN 2 ------
GAMMA and ALPHA:
5*H inner loops *  N/5H outer loops = N total loops
100 total loops * 60 cycles per loop = 6000 cycles

BETA:
W inner loops *  N/(5*H) outer loops = (N*W)/(5*H) total loops
(100*20)/(5*2) = 200 total loops
200 total loops * 40 cycles per loop = 8000 cycles
```

```
LLR:
5*H inner loops *  N/5H outer loops = N total loops
100 total loops * 1 cycle per loop = 100 cycles

TOTAL:
total cycles = 6000+8000+100 = 14700 cycles
```

```
------ DESIGN 3 ------
TOTAL:
N*(W/(5*H)) loops = 100*(20/(5*2)) = 200 loops
200 total loops * 60 cycles per loop = 12000 total cycles
```

```
------ COMPARE ------
12000 total cycles is less than 14700 total cycles.
Design 3 is faster.
```

As shown, with the fully saturated number of Beta Calculation Units, the fully pipelined approach performs better.

Ultimately however, this Gamma Calculation Unit bottleneck was eliminated by inlining the Gamma Calculation function, as inlining removes the hierarchy of function calls and allows the tools to optimize in ways it couldn't previously. This does however emphasize the importance of starting off simple without making any assumptions, and iterating and improving further based on hard evidence.

5.3 Low level implementations

The individual calculation units and the circular buffers were the low level components that made up the building blocks of the implementation. When creating these blocks, they were optimized individually before being implemented into the system. The

calculation unit implementation and optimization was straightforward and quick. The buffers underwent large changes and many iterations. Although these building blocks were important themselves, how they were used had a substantial impacted performance as well.

5.3.1 Calculation Units

The calculation units are important to examine as they make up the algorithm. The calculation units went through some changes but could mostly be used "as is" from the software implementation as they are mostly just mathematical computations. The initial software however was not optimized for implementation but was written to explicitly follow the algorithm. As such, some improvements were made that were also used to improve the software implementation. These improvements included manual loop unrolling, fully unrolling with the directives, and only calculating and storing half the values, as the other half is just the negative of the first. The Gamma Calculation Unit was able to gain extra optimizations by using the INLINE directive which proved to be very effective, as inlining allows the tools to make extra optimizations by removing the function call hierarchy. Due to the recursive nature of the Alpha and Beta Calculation Units, these became the biggest bottlenecks. Inlining did not produce any significant effect due to the recursive dependency requirements. One future optimization that could be used to improve performance would be to implement better normalization. Normalization prevents overflows but requires the calculation unit to wait for all of the calculations to be completed and then iterate over the elements an extra time. More advanced normalization could prove very beneficial. The LLR Calculation Unit was able to be pipelined at one cycle per output an thus wasn't looked at much for further optimization.

5.3.2 Circular Buffer

Circular buffers underwent the biggest, most significant changes. Initially, the buffers were implemented as shift registers. This functionality worked fine but consumed a lot of resources as it was implemented with FFs and LUTs. Instead, BRAM was desired as it is designed for this. BRAM can be used for storing large amounts of data but is limited to two ports per BRAM. This two port bottleneck can be minimized by strategically splitting the data among block RAMs using the array partition directive. In lieu of a shift registers, a circular buffer system was created to take advantage of the BRAM. Functions were created which could write to or read from the head or the tail of the buffer by taking a buffer as input.

Along with the optimizations that were made to this code, how it was used also played a large factor in performance. For example early on the entire circular buffers were copied at once which had a large negative impact on performance due to the port bottleneck. To alleviate this, instead of using different Gamma buffers to support the different Beta Calculation Units, a single larger buffer was used, and separate indexes were kept for each instance needing to access the buffer. This added complexity to the code, but greatly increased performance of the hardware, and actually reduced resource usage.

5.3.3 Dependencies

Initially, due to the way the MAX-MAP decoder was coded, dependencies played a substantial part in slowing performance. In sequential software execution, initializing a variable from a previous variable may make the most sense algorithmically, and not face any penalty. In hardware this created unnecessary dependencies. Independent temporary variables and independent arrays were created to remove these unnecessary dependencies wherever possible. This added complications to the code but enabled far more parallelism and better pipelining throughput. The HLS reports were often

used to identify the sources of these dependency issues.

5.3.3.1 Trellis Structure in Code

The calculation units rely on the trellis structure for their calculations, which is passed as a parameter. The trellis structure is used to determine three things. One is the output when going from one state to the next, given an input (0 or 1). Another is the previous state given a starting state, and an input (0 or 1). The last is the next state given a starting state, and an input (0 or 1). Each of these corresponds to one of the calculation units. The Gamma Calculation Unit uses the output, the Alpha Calculation Unit uses the previous state, and the Beta Calculation Unit uses the next state. Initially, the entire trellis structure containing all the information was passed to each of the calculation units. This was marked as a dependency by the HLS tools, limiting performance. To alleviate this, initially the DEPENDENCE directive was used to mark it as a false dependency which removed this limitation. The trellis representation was initially stored in BRAM which, as mentioned earlier, may only provide two ports for access. As the loops were being fully unrolled, complete parallel access of the trellis was required. This was accomplished by using the ARRAY PARTITIONING directive. When this was done, the trellis was split into three unique arrays to simplify the code, which eliminated the false dependency that was occurring previously.

5.3.4 Data Types

One area of design that has a big impact when utilizing HLS is the data types. Xilinx provides solutions which enable the user to have control over aspects of data types that are not possible using native C and C++ types. The data types used for HLS can either be native data types or Xilinx specific arbitrary precision data types. Xilinx arbitrary precision types are able to be simulated in C++ using the exact specified

representation. The HLS tools are able to handle and implement double precision and single precision floating point. Although this is powerful, many hardware designers choose to go with a fixed point representation of the data as fixed point is simpler and more efficient than floating point. Fixed point representations consume fewer resources and have shorter latency than floating point representations making them desirable for hardware designs. They can often suffer from accuracy or precision compared to floating point, but this can be minimized when optimizing fixed point for a specific algorithm.

The decoder design initially started off with double precision floating point to ensure the highest accuracy could be achieved. After this, floating point was used to see the trade offs. This was followed by a fixed point representation using the arbitrary precision library. For this representation, 8 integer bits and 16 decimal bits were used. This required switching from C to C++ to support the arbitrary precision fixed point library. Even though a C++ compiler was used, the only part of the code that took advantage of the C++ was the fixed point data type. This 8 integer bit and 16 decimal bit data type was used to represent all of the non integer variables. A somewhat common technique for optimization using fixed point in hardware designs is to tune your data types to the different stages of application. This means that the 24 bits used to represent the fixed point universally in this design might be larger than necessary at some stages, or might be too small and lose information at other stages due to the nature of the algorithm. Using different fixed point representations at different points in the algorithm is a technique that has been used when implementing turbo codes [28]. The particular fixed point BER performance used compared to the floating BER performance can be seen in Figure 5.6. The figure shows that the fixed point representation has little impact on the BER performance.

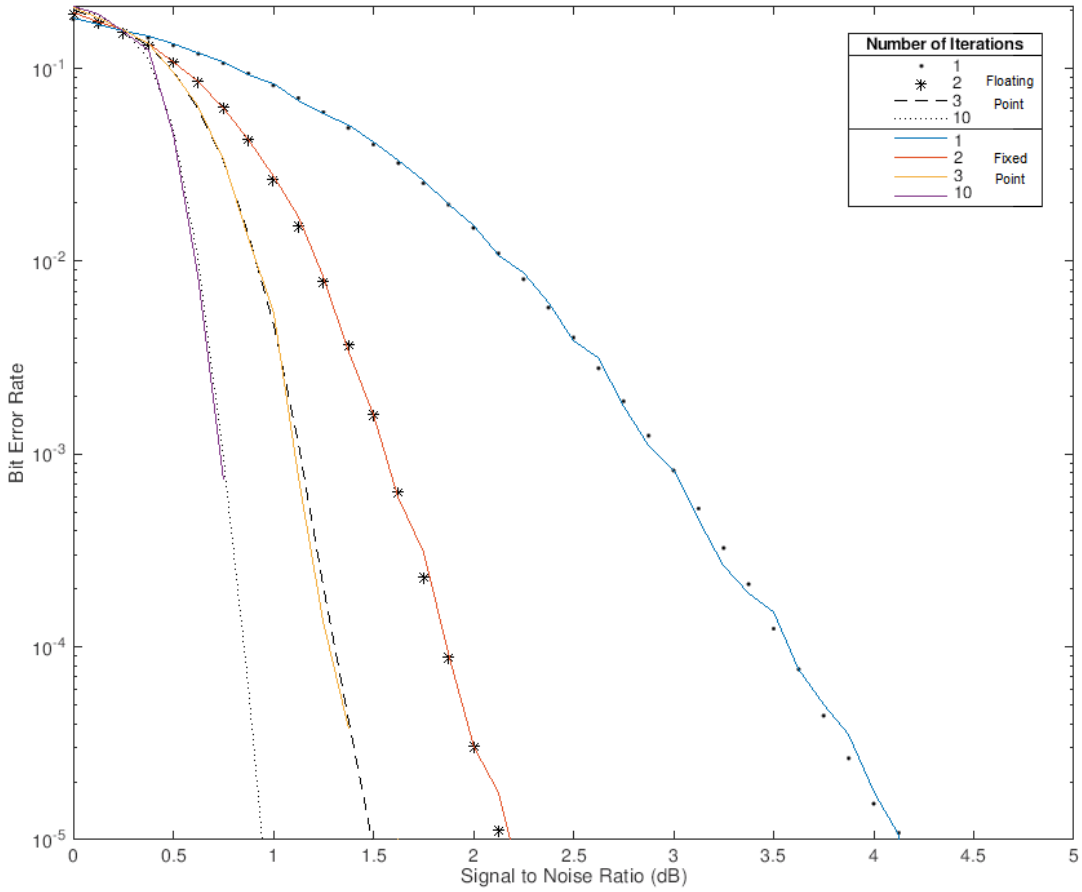

Figure 5.6: The fixed point BER performance compared to the floating point BER performance.

5.4 Flexibility

Along with reduced development time, one of the biggest benefits of HLS is the ease of flexibility that it provides. Having high level software describe the hardware allows techniques of high level code to be used. For this implementation, most of the flexibility came from a single file which defined most of the modifiable parameters. The quick single line modifications could change the implementation of the hardware with minimal effort. This can be very powerful as it provides the ability to tailor the hardware specifically for the application at hand. For example, the block size, window size, window hop amount, and number of Beta Calculation Units, could all be specified by changing a single line. Possibly more significantly, the data type representation could be easily specified as well, allowing double precision floating point representation, single precision floating point representation, or an arbitrary fixed point representation specified by the user, to be chosen by changing a single line. This enables a lot of flexibility that would be much harder to handle using traditional manually optimized design. The desired clock period is another parameter that could be set. In addition to this, the three different designs could be specified with only a few clicks as well.

Changing the parameters is not always that simple however. At least in this case, the parameter changes may require tuning to get optimized hardware implementations. Specifically, the directive values used for one set of parameters may not be suitable for another set of parameters. For example, the array partitioning and pipeline initiation interval may need to be modified to get the best results for the parameters at hand. In this case, this tuning would only need to occur once for a given set of parameters, and can be easily reused again if the same parameters are desired.

5.5 Accelerator Integration

Vivado was used to integrate the accelerator and generate the bitstream. The bitstream from Vivado was exported to XSDK. The environment setup from the software implementation was copied over to XSDK as well. XSDK was used to program the hard core ARM processor and configure the accelerator using the programmable logic. Everything was mostly kept the same with the code that was copied over except for two things. The first was the large arrays that were being input into the accelerated were specified at absolute addresses so their location could be handled manually. Second, the code to setup and run the accelerator was put in place to do so. If regular software execution was desired the accelerator calls were replaced with software function calls using #Defines.

5.5.1 Accelerator Interface

System integration played a large part in performance and need further consideration if the accelerator was to be implemented into an SDR as is. Initially, an AXI master interface was used to pass the data from memory to the accelerator. Analysis of this approach is discussed below. Ultimately, this interface became a huge bottleneck and the BRAM interface was used instead. The AXI master interface should not be counted out as the interface of choice, but was not considered further as system integration was not the main focus of this research. The BRAM interface required more explicit control from Processing System (PS) but eliminated this bottleneck so ultimately the true speed of the HLS implementation could be uncovered.

5.5.1.1 AXI Interface

The HLS INTERFACE directive was used to specify a master AXI interface in which, the programmable logic acted as the AXI master, and the PS, the AXI slave. The

Vivado integration can be seen in Figure 5.7. The figure shows the simple connection too the processing system and DDR over the AXI interface. Each of the parameter arrays is on its own separate bus, while the smaller parameters are tranfered over using an axi slave interface. On the hard core ARM processor side, at run time the addresses pointing to DDR memory were given to the accelerator so it could request data when it was running. These addresses represented the large arrays used as parameters to the accelerator. The code was written such that each loop a single element from each array was requested. The AXI interface excels at larger sequential burst (up to 256 elements) memory accesses. The code was restructured to read and write bursts at a time. Tests were conducted which examined how burst size effected performance and concluded that in this case bursts of 16 provided the best performance per programmable logic memory usage trade-off. Higher bursts got nearly the same performance but required greater programmable logic memory, while anything lower dropped performance significantly. Ultimately the bursting resulted in about a 2x speedup but memory accesses still consumed the majority of the running time. Designs were created which would continue to improve performance but were never implemented. One of these designs ran two functions in parallel, one being the decoder, the other being responsible for accessing memory. Using ping pong buffers, the memory access function would fill one buffer as the decoder would consume one, and buffers would swap for the next iteration, allowing them to run in parallel. This design should eliminate much of the bottleneck of the memory accesses as they are happening completely in parallel. The simplified pseudo code can be seen below. This design was used as a previous example in the HLS guidelines section.

Listing 5.5: Example code showing how parallel functions could be used to reduce the memory access limitations.

```
1 //Initialization
```

```
2  access_mem(b);
3  toggle = true;
4
5  for (loops) {
6    if toggle {
7      access_mem(a);
8      decode(b);
9    }
10   else {
11     access_mem(b);
12     decode(a);
13   }
14   toggle = !toggle;
15 }
16 //cleanup
```

The requirement to having them run in parallel is to ensure the code is written so that there are no dependencies between the functions. This is a very simplified design but shows the principals of how the design would work.

5.5.1.2 BRAM Interface

To eliminate this bottleneck, the BRAM interface was used. The Vivado integration can be seen in Figure 5.8. The figure shows the BRAM connection to the accelerator and processing system. The use of this interface is much different than using the AXI interface, and the integration is much more involved. Both the processing system and the accelerator required explicit access to the BRAM, creating more complexity in the integration. At runtime, before the accelerator is called, the parameters are read from DDR and copied into BRAM. When the accelerator is complete the data is read from BRAM and copied back into DDR. This becomes a little bit of a hindrance from the calling code as there are several copies that occur for the numerous iterations of

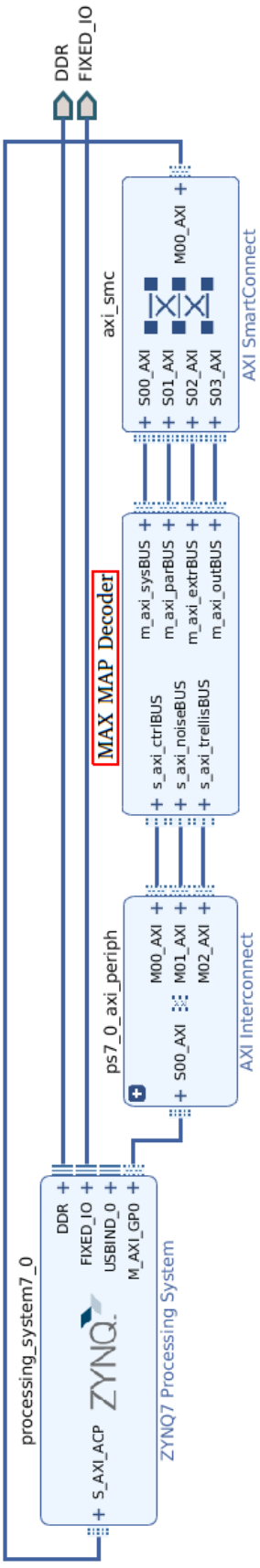

Figure 5.7: The MAX MAP decoder integration using the Axi interface on the Zynq board.

the decoder. This interface however makes much more sense if the entire turbo code is being accelerated, allowing BRAM to be used for all of the intermediate steps instead of accessing DDR over and over again. If this were the case, a single copy to BRAM at the start of the decoder, and a single copy from BRAM at the end of the decoder would be required. This approach was able to completely remove the memory access bottleneck and allowed the true speed of the accelerator to be seen.

5.5.2 Issues Running On The Board

It is important to note two issues were faced when trying to run the system. These issues are not specific to the accelerator and may occur in other work as well. The first issue faced was the stack was not large enough and would run out of room. To fix this the linker script needed modification to make the stack larger. The second issue faced was the running the accelerator initially produced values that were different than the simulation produced. A new simpler accelerator was created as a test to find the root cause of the issue. The Integrated Logic Analyzer was used to see the data being passed to the accelerator over the AXI interface and it was uncovered that the incorrect data was being transferred over the interface. Further inspection showed this data was not random but instead was outdated data from previous runs. It was eventually realized that cache was enabled by default and was causing this issue. This was turned off and the issue was fixed.

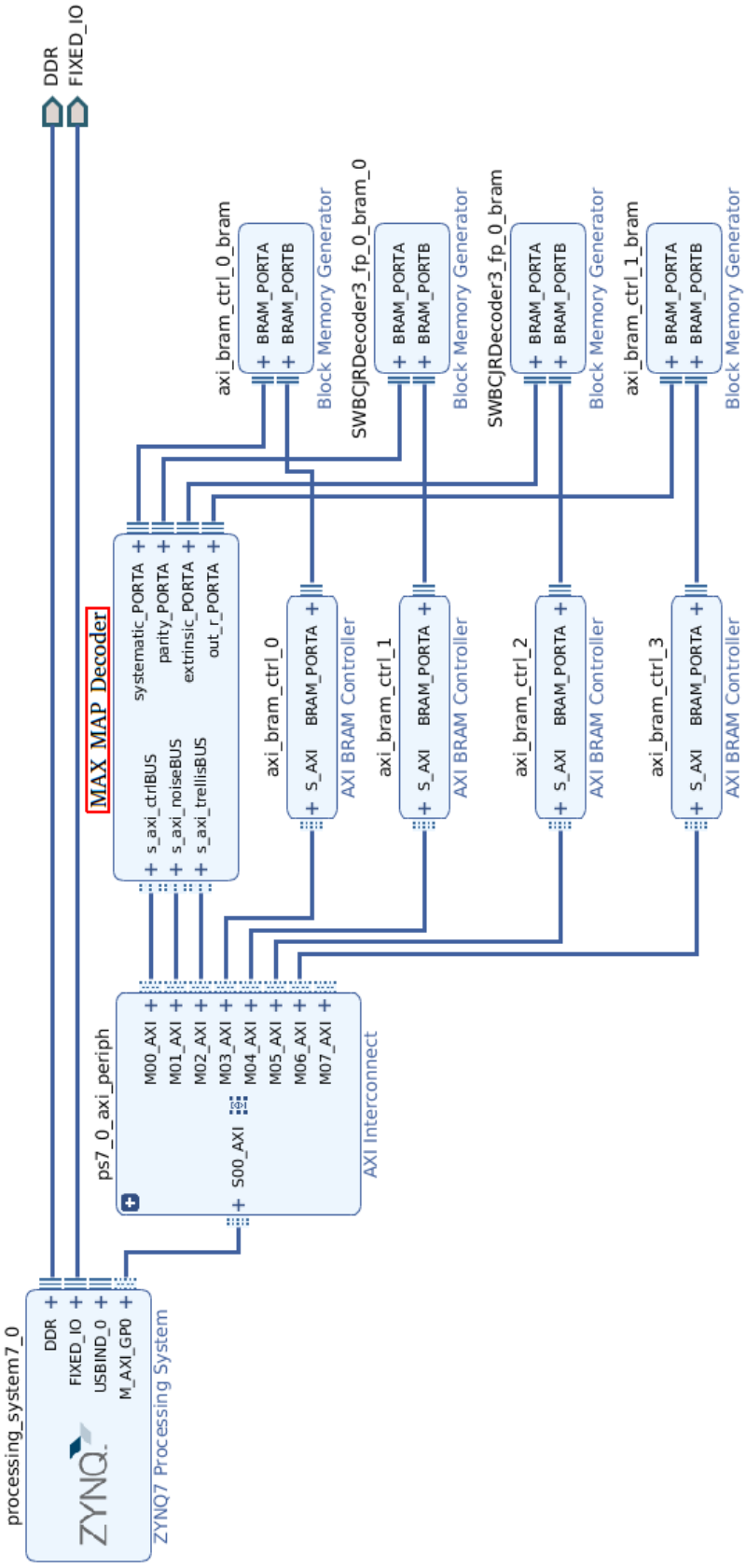

Figure 5.8: The MAX MAP decoder integration using the BRAM interface on the Zynq board.

Chapter 6

Results

It is important to first understand the platform used before appreciating the results. This chapter discusses the hardware used and then compares the results across different aspects. Most significantly, software implementations are compared to hardware to see how effective hardware acceleration can be utilizing HLS, and HLS throughput is compared with manually optimized designs to see how competitive HLS is against fully custom implementations.

6.1 Zynq Platform

The Xilinx Zynq platform was leveraged for this work. This platform integrates an ARM processor with programmable logic of an FPGA. This allows a very beneficial setup for an SDR. The software can run on the ARM processor and any tasks requiring acceleration can be run on the programmable logic of the FPGA. This decreased the time and complexity of building the system, as it is all integrated into a single chip. The board used for this woork was the ZCU102 evaluation kit, a Zynq-Ultrascale+ chip.

A simplified diagram of the integration between the hardcore application and the FPGA accelerator can be seen in Figure 6.1. The figure shows the MAX-MAP accelerator, the control lines used to start and stop the accelerator, and the interface to the BRAM where the accelerator accesses the bit sequences. The specifications

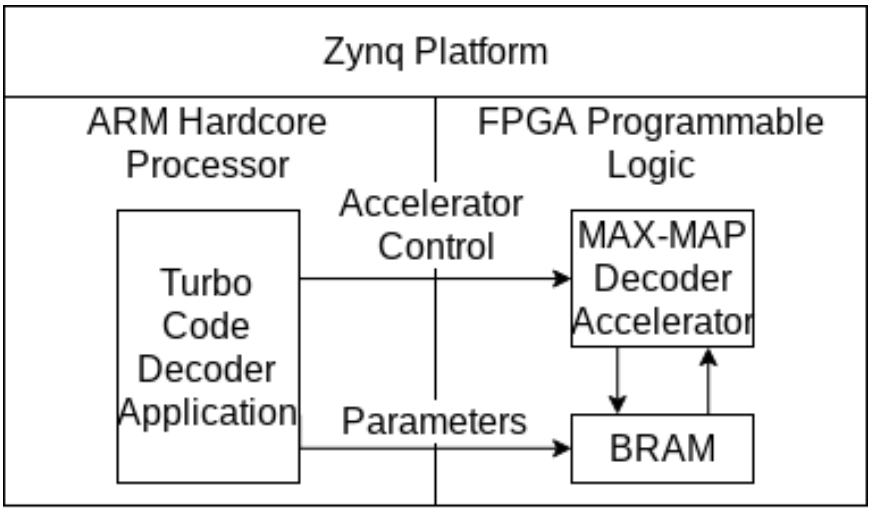

Figure 6.1: Simplified diagram of how the turbo code decoder and MAX-MAP Decoder accelerator is connected on the Zynq hardware.

of the hardware used for these tests can be seen in Table 6.1 and Table 6.2. Table 6.1 lists the hardcore processor specifications, while Table 6.2 lists the programmable logic specifications.

Table 6.1: ZCU102 Processor Specifications

Type	Name	Operation Frequency
General Purpose Core	Quad Core Cortex A53	up to 1.5GHz
Realtime Core	Dual Core Cortex R5	up to 600MHz
Graphics Processor	Mali-400 MP2	500MHz

Table 6.2: ZCU102 Programmable Logic Specifications

Type	Amount	
Programmale Logic Cells	599	k
Flip Flops	548	k
LookUp Tables	274	k
DSP Slices	2.5	k
Block Ram	32.1	Mb

6.2 Designs used for testing

In this section the different test cases will be described and justified. The test case implemented an entire turbo code encoder and decoder, but only accelerated and

profiled the MAX-MAP decoder. The hardware acceleration was broken into three different designs as explained previously.

- Design 1: sliding window approach with pipelining, unrolling and partition directives as a *baseline* hardware design,
- Design 2: *baseline* + replicated beta calculation units,
- Design 3: *baseline* + replicated beta calculation units + further loop pipelining.

Within each of these designs, the data representations was modifiable between *double precision floating point*, *single precision floating point*, and *fixed point*. The fixed point used in this case used 8 integer bits and 16 decimal bits. This provided 9 hardware acceleration tests. These same test cases, except for the fixed point representations, were recorded in software acceleration to present a baseline. This provides excellent insight into how effective FPGA acceleration can be in an SDR. These results were recorded for decoding a bit sequence from a memory 3 RSC encoder represented by a trellis with 8 states, and decoding a bit sequence from memory 4 RSC encoder represented by a trellis with 16 states.

6.2.1 Parameters

Each of the parameters is modifiable by simply changing the value in a single file, and can be tailored to meet the requirements of the application. The number of iterations as well as SNR, were run time parameters, while the flexible compile time parameters were the block size, the trellis size, the window size, the window hop, the number of Beta Calculation Units, the data type representation, and the clock period, which can be seen in Table 6.3. In Table 6.4 the parameter which remained constant for the different test cases can be observed. These parameters were selected to represent practical values and remained fixed throughout the research.

Along with the architecture selection, the test cases tested different data type representations and different trellis sizes, to see how these parameters affected the implementation. The data type implementations varied from double precision floating point, to single precision floating point, to fixed point, with fixed point being represented by 8 integer bits and 16 decimal bits. The trellis states altered between 8 states to 16 states.

Table 6.3: Modifiable Implementation Parameters

Parameter	Unit
Block Size	# Bits
Window Size	# Bits
Window Hop	# Bits
Parallel Beta Calculation Units	# Units (designs 2 and 3 only)
Trellis States	# States
Data Type Representation	Type
Clock Period	Nano Seconds

Table 6.4: Value of Parameters which Remained Constant

Block Size	3200	Bits
Window Size	32	Bits
Window Hop	8	Bits
Parallel Beta Calculation Units	4	Units (designs 2 and 3 only)
Clock Period	2.963	ns

6.3 Hardware vs Software

6.3.1 Software Designs

The first metrics looked at how the different designs compared for software execution. These results are significant for drawing larger conclusions later. The timing results were recorded using a global ARM timer available on the chip. The memory 3 double precision floating point and single precision floating point results for each of the three designs can be seen in Table 6.5. Fixed point was not included in these tests as fixed

point representations were not created for software execution. The different test cases are each within 7% of the mean. The memory 4 software results can be seen in Table 6.6. These results are all within 5% of the mean.

Table 6.5: Memory 3 encoder software results

Design	double precision software cycles	single precision software cycles
Design 1	1,880,455	1,807,305
Design 2	1,902,489	1,824,358
Design 3	1,938,593	1,884,609

Table 6.6: Memory 4 encoder software results

Design	double precision software cycles	single precision software cycles
Design 1	3,667,772	3,331,676
Design 2	3,689,522	3,349,087
Design 3	3,695,286	3,457,006

Comparing the two test cases shows that the larger memory test case consumed about a 2 times the amount of cycles which is expected as the trellis state doubles in size, requiring 2 times as many calculations. Examining the test cases individually shows there is not a huge difference between the implementations in software, even though there are significant differences in the designs, and differences in the data types. In both cases the double precision data type representation takes longer than the single precision, which is to be expected. The results also show that the data type representation accounts for a larger difference than the designs.

6.3.2 Hardware vs Software

An software optimized MAX-MAP decoder was created to be used as a baseline to compare to the hardware implementations. This version of the software leveraged the Single Instruction Multiple Data (SIMD) feature of the ARM core. The timing results for this implementation can be seen in Table 6.7. This optimized implementation was roughly 6 times faster than the implementations created for use with HLS.

The memory 3 double precision floating point, single precision floating point, and fixed point results for each of the three designs can be seen in Table 6.8. The speedups of the hardware implementations over the optimized software implementation is recorded in the table and also shown in Figure 6.2. The Figure illustrates the potential benefit of the hardware implementations, where In the best case the hardware is over 56 times faster.

Table 6.7: Optimized Software Results

Memory Size	Single Precision Software Cycles
Memory 3	281,553
Memory 4	570,526

Table 6.8: Memory 3 Hardware Results

	Hardware Cycles			Speedup (Multiple)		
Design	double	float	fixed	double	float	fixed
Design 1	317451	284048	162804	0.8869	0.9912	1.7294
Design 2	116349	107344	25302	2.4199	2.6229	11.1277
Design 3	70747	59006	4988	3.9797	4.7716	56.4461

The same was repeated for the memory 4 shown in Table 6.9. The optimized software implementations is used to compare the against the hardware designs for the memory 4 tests. The speedups are recorded in the table and also shown in Figure 6.3. The results are similar to the previous test cases. In the best case the hardware is over 43 times faster.

Table 6.9: Memory 4 Hardware Results

	Hardware Cycles			Speedup (Multiple)		
Design	double	float	fixed	double	float	fixed
Design 1	358944	326873	297088	1.5895	1.7454	1.9204
Design 2	168462	155150	23623	3.3867	3.6773	24.1513
Design 3	81518	77248	13014	6.9988	7.3856	43.8394

Comparing the designs, it is clear to see that design 2 increases performance over design 1, and design 3 increases performance even further. Again, design 2 is able to

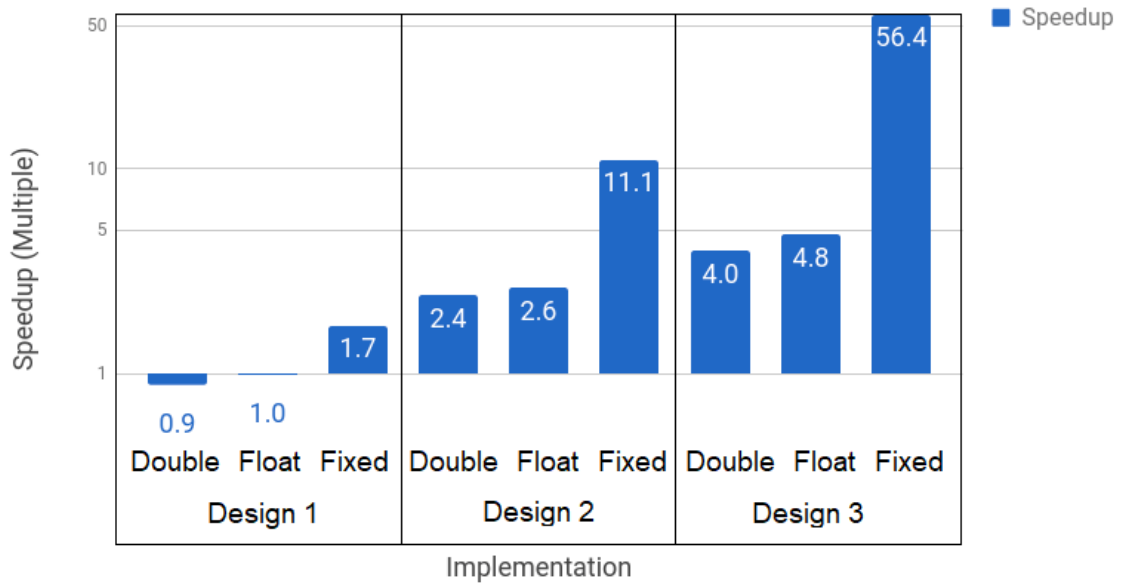

Figure 6.2: Performance of the memory 3 hardware accelerator implementations over the best software implementation. Block Size: 3200 Bits, Trellis Size: 8 States, Window Size: 32 Bits, Window Hop : 8 Bits, Beta Calculation Units: 4 Units

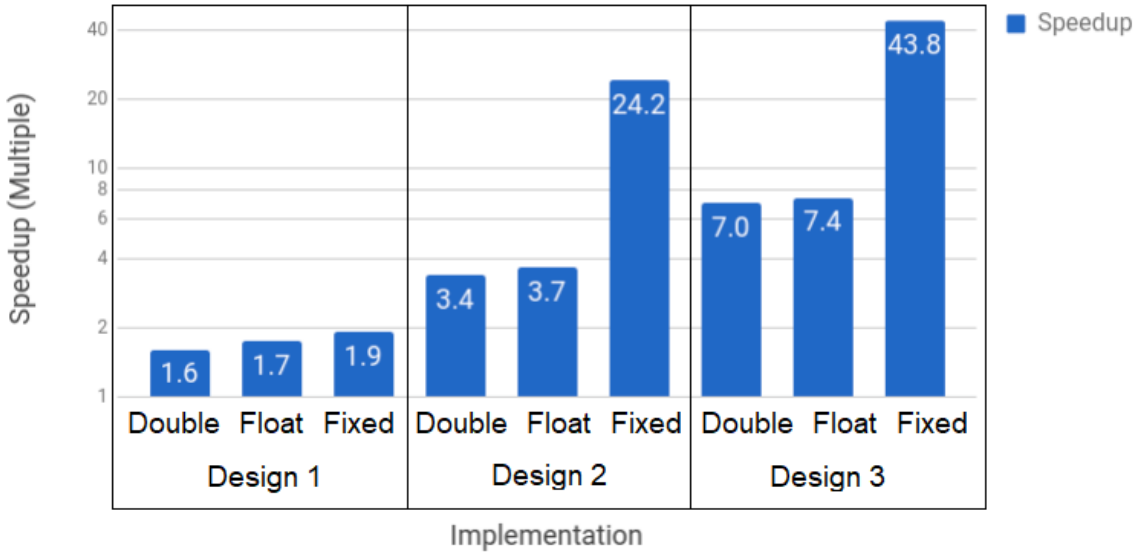

Figure 6.3: Performance of the memory 4 hardware accelerator implementations over the best software implementation. Block Size: 3200 Bits, Trellis Size: 16 States, Window Size: 32 Bits, Window Hop : 8 Bits, Beta Calculation Units: 4 Units

do this with hardware replication, increasing parallelism, and hiding the extra Beta calculations required by the sliding window approach. Design 3 is able to take this even further as the design is fully pipelined, allowing each of the calculation units to run in parallel, instead of only allowing one type of calculation unit to run at a time. In design 1, for the parameters chosen, the Beta Calculation unit has to calculate 4 times as many values as the Gamma and Alpha Calculation units, requiring 4 times as many cycles. The LLR Calculation Unit is very fast and comparatively the calculation time is negligible. Design 2 is able to hide the extra Beta calculations through parallelism, bringing it down to an equal number of cycles per output as the Gamma and Alpha calculations. Therefore it is estimated that design 2 should be able to achieve roughly a 2.5 times speedup over design 1. Analysis shows that this is accurate for the floating point implementations. Design 3 allows for parallelism between calculation units, halving the cycles per output, which is estimated to account for roughly a 2 times speedup of design 3 over design 2. Again this is seen for the floating point implementations. In each of the three designs for the floating point implementations, the bottleneck was speed of the evaluation of the mathematical calculations in the different calculation units, as reported by the Vivado HLS tools. Parallelizing the calculation unit bottleneck allowed for predictable results.

The fixed point representations enabled large leaps in performance as the simpler logic requires less resources, and can be evaluated faster. The bottleneck of floating point implementations was the calculation units, but this was not the case for the fixed point implementations. Instead, the bottleneck of the fixed point implementations, as reported from the Vivado HLS tools, was the reading and writing of the local variable dependencies between the Gamma and Alpha Calculation Units, and the Beta Calculation Units. The structure of these dependencies changed for each of the three designs, causing less predictable speedups.

One interesting outlier is in the case of the memory 4, design 1 implementations.

In all other cases, the fixed point representation is noticeably better than the floating point implementations. In this specific case, it is still the best, but not by nearly as significant an amount. In this case, it is speculated that less optimized memory partitioning is the cause.

In the previous subsection the software implementations were compared and showed there is some difference between the implementations, but nothing greater than 7% from the mean. The software, written different ways, changes the control flow and code complexity but does not greatly impact the performance in sequential execution. Comparing the hardware implementations on the other hand shows extraordinary differences upwards of 63 times faster if the best implementations is compared to the worst implementation. This re-enforces the previously discussed point that writing high level code for hardware often requires code that does not make sense for sequential software execution.

These results exhibit the profound abilities of FPGAs when accelerating tasks with hardware. Furthermore, it shows that HLS can be and effective method for developing hardware accelerators.

6.3.3 Hardware Resource Utilizations

It is important to look at resource utilization when creating hardware designs. The memory 3 double precision floating point, single precision floating point, and fixed point hardware usages for each of the three designs can be seen in Table 6.10. In this table, the Block RAM, DSP slices, Flip-Flops, and Look-Up Tables are reported as these make up the programmable logic of the FPGA. The memory 4 resource usage can be seen in Table 6.11.

These results show that generally, the fixed point designs used far fewer resources than the floating point counterparts. This is expected as fixed point representations are simpler and smaller than floating point representations. The fewer resources

Table 6.10: Memory 3 Resource Usage Table

Design	Data Type	BRAM		DSP48E		FF		LUT	
		Amount	%	Amount	%	Amount	%	Amount	%
Maximum Capacity	—	912	100	2,520	100	548,160	100	274,080	100
Design 1	double	113	12.39	54	2.14	61,170	11.16	60,160	21.95
	float	97	10.64	39	1.55	38,854	7.09	38,851	14.18
	fixed	85	9.32	4	0.16	26,772	4.88	36,435	13.29
Design 2	double	137	15.02	81	3.21	84,602	15.43	90,914	33.17
	float	109	11.95	61	2.42	50,751	9.26	54,500	19.88
	fixed	93	10.20	4	0.16	31,852	5.81	45,167	16.48
Design 3	double	121	13.27	30	1.19	89,928	16.41	85,897	31.34
	float	101	11.07	21	0.83	54,811	10.00	54,047	19.72
	fixed	101	11.07	4	0.16	36,620	6.68	51,767	18.89

Table 6.11: Memory 4 Resource Usage Table

Design	Data Type	BRAM		DSP48E		FF		LUT	
		Amount	%	Amount	%	Amount	%	Amount	%
Maximum Capacity	—	912	100	2,520	100	548,160	100	274,080	100
Design 1	double	145	15.90	212	8.41	121,180	22.11	127,669	46.58
	float	113	12.39	137	5.44	65,663	11.98	66,795	24.37
	fixed	95	9.32	4	0.16	31,602	5.77	52,250	19.06
Design 2	double	193	21.16	153	6.07	147,869	26.98	183,605	66.99
	float	137	15.02	109	4.33	77,696	14.17	95,630	34.89
	fixed	105	11.51	4	0.16	41,436	7.56	74,233	27.08
Design 3	double	161	17.65	53	2.10	120,329	21.95	161,679	58.99
	float	121	13.27	31	1.23	75,019	13.69	90,041	32.85
	fixed	57	6.25	4	0.16	62,633	11.43	97,439	35.55

required along with the tremendous speedups that were displayed previously, demonstrate why fixed point designs are desirable for hardware implementations. Further analysis shows that as the designs progressed, they generally increased in resource usage. Again, design 2 increased parallelism by replicating the Beta Calculation Units, so the increase in resources is expected. Design 3 added a more complex control flow to allow further pipelining, in addition to the replicated Beta Calculation Units, so the increase is expected as well.

Comparing across tables shows that the memory 4 implementation uses significantly more resources than the memory 3 implementation. This is because the mem-

ory 4 uses a trellis with 16 states whereas the memory 3 uses a trellis with only 8 states.

One interesting take away is that fixed point implementations hardly took advantage of the DSPs compared to the floating point representations, demonstrating the DSPs value in floating point implementations. When hand coding RTL the low level building blocks are something that might need to be highly considered, but when utilizing HLS this can be abstracted away.

The BRAM resource utilization is relatively equal in all of the memory 3 implementations as the implementations were created with more memory than required for the input and output parameters. In addition to this, the smaller 8 state trellis does not require much BRAM, hiding the details in the parameter overhead. More insight can be gained by looking at the memory 4 results. These results show that the parallelism of design 2 increases the BRAM usage as expected, but the fully pipelined implementations are able to optimize and actually use less resources. Also as expected, the double precision floating point implementations require more BRAMS than the single precision floating point representations, and the fixed point representations require the least number of BRAMs.

6.4 HLS vs Manually Optimized Design

The previous sections focused the acceleration advantages hardware can provide over sequential execution, but even more interesting conclusions can be drawn if the HLS implementations are compared against manually optimized designs. The HLS tools become even more favorable if they not only have the advantages in flexibility, development time, and design space exploration, but can compete with manually optimized designs.

The throughput of the best implementation can be seen in Table 6.12. This table compares the results recorded from an HLS implementation against the implemen-

tation results from [28]. In [28], a manually optimized Sliding Window MAX-MAP decoder is used, allowing fair comparisons as the architectures follow the same design principals.. In this case, the HLS implementation on the Zynq Ultrascale+ hardware achieves 19.53% of the throughput of the manually optimized design. Both implementations use a trellis with 8 states, and are fully parallelized with respect to the Beta Calculation Units. The manually optimized design however does this with a window of size 20, a hop of size 1, and 20 parallel Beta Calculation Units, whereas the HLS implementation in this case uses a window of size 32, a hop of size 8, and 4 parallel Beta Calculation Units. Besides this, there are three differences in design that can account for some of the performance improvement of manually optimized implementation.

- *Smaller fixed point representations which are tuned for different stages.* The manually optimized design uses a minimum of 7 bits with 4 integer and 3 decimal bits, and a maximum of 13 bits with 10 integer and 3 decimal bits. The HLS implementation uses a constant 24 bits with 8 integer and 16 decimal bits throughout the entire implementation.

- *Improved normalization.* The manually optimized design uses modulo normalization whereas the HLS implementation uses subtractive normalization.

- *Optimized Gamma Calculation Unit.* The manually optimized design optimizes the Gamma Calculation unit further than the HLS design, requiring 4x less calculations.

Each of these improvements are fully applicable to the HLS design and should be considered for future work. This is discussed again in the following chapter. The results show that although the manually optimized design is faster, HLS is still competitive while enabling numerous advantages. It is important to remember there is still room for improvement of the HLS design.

Table 6.12: Throughput Table

Work	Tech. Family	Max. Freq.	Throughput
Custom[28]	Virtex V	346MHz	346 Mbps
HLS	Zynq UltraScale+	337.5MHz	67.5 Mbps

```
#define NUM_BITS 1600
#define NUM_TRELLIS_STATES 4
#define WINDOW 17
#define WINDOW_HOP 2
#define NUM_BETAS 3

typedef ap_fixed<24,16> data_t;
//typedef float data_t;
//typedef double data_t;
```

Figure 6.4: The values that change in a single line to create unique implementations demonstrating the flexibility of HLS

Although the HLS implementation did not reach the same throughput as the manually optimized design, a huge advantage in flexibility is gained. To demonstrate the power of the flexibility, a new implementation was created with entirely new parameters. This implementation only required changing to the values in a single file, shown in Figure 6.4, and checking the reports to ensure there were no errors. Selecting new parameters took only seconds, and within a few minutes an entirely new MAX-MAP decoder accelerator was up and running. In this case, the design 3 fixed point implementation was selected, along with a 4 state trellis, which is smaller than the previous designs, requiring less calculations and enabling faster decoding. In addition to this, a smaller block size of 1600 bits was used, halving the calculation requirements. A window size of 17 was used, with a hop of 2, and 3 parallel Beta Calculation Units. The hardware cycles and resource usages can be seen in Table 6.13 and Table 6.14 respectively.

Table 6.13: Memory 2 Hardware Results

	Hardware Cycles
Design	fixed
Design 3	2,978

Table 6.14: Memory 2 Resource Usage Table

Design	Data Type	BRAM		DSP48E		FF		LUT	
		Amount	%	Amount	%	Amount	%	Amount	%
Maximum Capacity	—	912	100	2,520	100	548,160	100	274,080	100
Design 3	fixed	81	8.88	4	0.16	26,324	4.80	31,054	11.33

Chapter 7

Conclusion and Future Work

7.1 Conclusions

High Level Synthesis can be a powerful tool. This work focused on exploring the potential of HLS for accelerating SDR tasks. In particular, turbo codes, a popular form of forward error correction, were described and underwent hardware acceleration using an FPGA and HLS tools. Different aspects necessary for effective HLS development were identified, mainly emphasizing the idea that a hardware mindset is required for effective results. The advantages available for HLS were seen throughout the development, including flexibility, design space exploration, and reduced development time. The final implementations allowed quick and easy parameter selection from modifying a single macro file, demonstrating the flexibility gained using HLS tools. The design space was explored and compared using three different designs, with the capability of using 3 different data types each. The designs started with a simple sliding window approach, then added hardware replication to enable parallelism and hide extra calculations, and finally used an improved pipelining design to increase throughput even further. The data types included double precision floating point, single precision floating point, and fixed point with 8 integer and 16 decimal bits. Two different trellis sizes were used for test cases. The results showed that in the best case, the hardware acceleration was over 8500 times faster than the best software

implementation, and in the worst case the hardware acceleration was still over 130 times faster than the best software implementation. The software implementations however did not vary by more than 8% of the mean, supporting the idea that to most effectively use HLS, and approach with a hardware mindset is required. The HLS implementation was also compared with a manually optimized design and achieved 19% of the throughput comparatively, with further room for improvements identified in the next section. Thus emphasizing the competitiveness of HLS compared to traditional manually optimized designs.

7.2 Further Design Improvements

The various results display the very promising potentials of HLS. The acceleration capability over software is outstanding at up to 8597 times faster, and is still competitive to manually optimized designs running at 19.53% throughput. Even with the current results, there are several areas which can be investigated to further improve the results.

The three biggest areas for improvement identified are with the normalization, access contention, and fixed point tuning. *Normalization* is not explicit in the algorithm but is used to removed overflow susceptible by computer representations of values. It requires finding the maximum value in the calculation results, and then modifying all of calculation results a second time before finalizing them so they can be used at the next stages. Due to the recursive dependency nature of the Alpha and Beta Calculation Units, this extra calculation time cannot be hidden. On the other hand, *memory* access was an important aspect of the implementations and was handled carefully, but the access contention can still be improved further. Lastly, the *fixed point representation* used a single representation all throughout the design. It can be beneficial to tailor specific sections of the algorithm to different fixed point representations as demonstrated in [28]. This fine tuning customization of the repre-

sentation will provide optimizations that enable even greater performance, and also reduce resource utilization.

The current designs also make the assumption that a single new bit is available for each loop. Removing this assumption could allow new designs to take advantage and improve throughput further, but may not be possible depending on application requirements. In addition to this, one immediate improvement that can be taken advantage of with very little human effort is pushing the tools further. There is room left for the tools to be pushed further to increase programmable logic clock speed and possibly reduce resource usage by simply setting harsher constraints for Vivado implementation. These are some of the current improvements that could be made to enhance this MAX-MAP decoder, but further improvements to the overall turbo code implementation is discussed in detail in the next section.

7.3 Future Work

In addition to the previous section, there are many areas to expand on this research. Most significantly, future work should investigate implementing the entire turbo code decoder in hardware instead of the just the MAX-MAP decoder. This will create a more practical implementation and enable direct comparisons between other turbo code decoders. Additionally, parallel MAP decoders implementations should be explored for increased throughput that more closely matches state of the art designs. Manually optimized implementation should be created to have perfectly comparable results. This will provide more insight into the trade offs between manually optimized implementations and HLS implementations. Lastly, the turbo code error correction should be integrated as part of an SDR design to analyze the effects of FPGA acceleration on the whole system. The system integration was a big piece that was largely ignored which could have a significant impact on results in real systems.

Bibliography

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Communications, 1993. ICC '93 Geneva. Technical Program, Conference Record, IEEE International Conference on*, vol. 2, May 1993, pp. 1064–1070 vol.2.

[2] M. May, T. Ilnseher, N. Wehn, and W. Raab, "A 150mbit/s 3GPP LTE turbo code decoder," in *2010 Design, Automation Test in Europe Conference Exhibition*, March 2010, pp. 1420–1425.

[3] Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, A. Reid, and K. Flautner, "Design and implementation of turbo decoders for software defined radio," in *2006 IEEE Workshop on Signal Processing Systems Design and Implementation*, Oct 2006, pp. 22–27.

[4] F. Kienle, H. Michel, F. Gilbert, and N. Wehn, "Efficient MAP-algorithm implementation on programmable architectures," in *Adv. Radio Sci.*, 2003, pp. 259–263.

[5] B. Kang, N. Vijaykrishnan, M. J. Irwin, and T. Theocharides, "Power-efficient implementation of turbo decoder in SDR system," in *IEEE International SOC Conference, 2004. Proceedings.*, Sept 2004, pp. 119–122.

[6] D. R. N. Yoge and N. Chandrachoodan, "GPU implementation of a programmable turbo decoder for software defined radio applications," in *2012 25th International Conference on VLSI Design*, Jan 2012, pp. 149–154.

[7] L. Huang, Y. Luo, H. Wang, F. Yang, Z. Shi, and D. Gu, "A high speed turbo decoder implementation for CPU-based SDR system," in *IET International Conference on Communication Technology and Application (ICCTA 2011)*, Oct 2011, pp. 19–23.

[8] F. Gilbert, M. J. Thul, and N. Wehn, "Communication centric architectures for turbo-decoding on embedded multiprocessors," in *2003 Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 356–361.

[9] S. Belfanti, C. Roth, M. Gautschi, C. Benkeser, and Q. Huang, "A 1gbps LTE-advanced turbo-decoder ASIC in 65nm CMOS," in *2013 Symposium on VLSI Circuits*, June 2013, pp. C284–C285.

[10] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Communications, 1995. ICC '95 Seattle, 'Gateway to Globalization', 1995 IEEE International Conference on*, vol. 2, Jun 1995, pp. 1009–1013 vol.2.

[11] C. Schurgers, F. Catthoor, and M. Engels, "Memory optimization of MAP turbo decoder algorithms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 2, pp. 305–312, April 2001.

[12] Y. Sun and J. R. Cavallaro, "Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder," *Integration, the VLSI Journal*, vol. 44, no. 4, p. 305315, 2011.

[13] S. Lahti, P. Sjvall, J. Vanne, and T. D. Hmlinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018.

[14] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE International Conference on ASIC*, Oct 2011, pp. 1102–1105.

[15] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? a case study," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–8.

[16] V. Bhatnagar, G. S. Ouedraogo, M. Gautier, A. Carer, and O. Sentieys, "An FPGA software defined radio platform with a high-level synthesis design flow," in *2013 IEEE 77th Vehicular Technology Conference (VTC Spring)*, June 2013, pp. 1–5.

[17] M. Rler, H. Wang, U. Heinkel, N. Engin, and W. Drescher, "Rapid prototyping of a DVB-SH turbo decoder using high-level-synthesis," in *2009 Forum on Specification Design Languages (FDL)*, Sept 2009, pp. 1–6.

[18] J. Andrade, N. George, K. Karras, D. Novo, F. Pratas, L. Sousa, P. Ienne, G. Falcao, and V. Silva, "Design space exploration of LDPC decoders using high-level synthesis," *IEEE Access*, vol. 5, pp. 14 600–14 615, 2017.

[19] Y. Sun, J. R. Cavallaro, and T. Ly, "Scalable and low power LDPC decoder design using high level algorithmic synthesis," in *2009 IEEE International SOC Conference (SOCC)*, Sept 2009, pp. 267–270.

[20] E. Scheiber, G. Bruck, and P. Jung, "Implementation of an LDPC decoder for IEEE 802.11n using Vivado high-level synthesis," 2013.

[21] R. I. Lackey and D. W. Upmal, "Speakeasy: the military software radio," *IEEE Communications Magazine*, vol. 33, no. 5, pp. 56–61, May 1995.

[22] D. Bell, S. Allen, N. Chamberlain, M. Danos, C. Edwards, R. Gladden, D. Herman, S. Huh, P. Ilott, T. Jedrey, T. Khanampornpan, A. Kwok, R. Mendoza, K. Peters, S. Sburlan, M. Shihabi, and R. Thomas, "MRO relay telecom support of Mars science laboratory surface operations," in *2014 IEEE Aerospace Conference*, March 2014, pp. 1–10.

[23] J. Taylor and R. Ludwig, *Voyager Telecommunications*, p. 66. [Online]. Available: https://descanso.jpl.nasa.gov/monograph/series13/DeepCommoOverall--141030A_ama.pdf

[24] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, Mar 1974.

[25] S. Papaharalabos, M. Sybis, P. Tyczka, and P. T. Mathiopoulos, "Modified log-MAP algorithm for simplified decoding of turbo and turbo TCM codes," in *VTC Spring 2009 - IEEE 69th Vehicular Technology Conference*, April 2009, pp. 1–5.

[26] M. Fingeroff, *High-level synthesis Blue book*. Xlibris Corporation, 2010.

[27] "Evolved universal terrestrial radio access (e-utra); multiplexing and channel coding," in *3GPP TS 36.212 version 8.8.0 Release 8*, vol. 8, 2010, p. 14.

[28] R. Shrestha and R. Paily, "Design and implementation of a high speed MAP decoder architecture for turbo decoding," in *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*, Jan 2013, pp. 86–91.

Chapter 8

Appendix

8.1 Apprendix A: MAP Decoder Psuedocode

Listing 8.1: Psuedocode for the MAP decoder.

```
1
2  calculate_gamma(systematic[NUM_BITS],
3                  parity[NUM_BITS],
4                  extrinsic[NUM_BITS],
5                  noise,
6                  gamma_output[NUM_BITS][NUM_STATES][NUM_INPUTS])
7  {
8      for NUM_BITS {
9          for NUM_STATES {
10             //input is either 0 or 1
11             for NUM_INPUTS {
12                 //uk = input
13                 xk = trellis_get_output(state,input);
14                 gamma_output[bit][state][input] = exp(0.5*noise*parity↩
                       [bit]*xk + 0.5*uk*(extrinsic[bit]+noise*systematic[↩
                       bit]))
15
16             }
```

```
17          }
18      }
19  }
20
21  calculate_alpha(gamma_output[NUM_BITS][NUM_STATES][NUM_INPUTS],
22                  alpha_output[NUM_BITS][NUM_STATES])
23  {
24      for NUM_BITS {
25          for NUM_STATES {
26              previousState0 = trellis_get_previous_state(state, INPUT_0↩
                    );
27              previousState1 = trellis_get_previous_state(state, INPUT_1↩
                    );
28
29              alpha[bit+1][state] = (alpha[bit][previousState0] * gamma[↩
                    bit][previousState0][INPUT_0]
30                                      + alpha[bit][previousState1] * gamma[↩
                                            bit][previousState1][INPUT_1]);
31
32          }
33      }
34  }
35
36  calculate_beta(gamma_output[NUM_BITS][NUM_STATES][NUM_INPUTS],
37                  beta_output[NUM_BITS][NUM_STATES])
38  {
39      for NUM_BITS down to 0 {
40          for NUM_STATES {
41              nextState0 = trellis_get_next_state(state, INPUT_0);
42              nextState1 = trellis_get_next_state(state, INPUT_1);
43
44              beta[bit-1][state] = (beta[bit][nextState0] * gamma[bit][↩
                    nextState0][INPUT_0]
```

```
45                                  + beta[bit][nextState1] * gamma[bit][↩
                                        nextState1][INPUT_1]);
46
47          }
48      }
49  }
50
51  calculate_LLR(gamma_output,
52                alpha_output,
53                beta_output,
54                decoder_output)
55  {
56      for NUM_BITS {
57          input_of_0 = 0;
58          input_of_1 = 0;
59
60          // for an input of 0
61          for NUM_STATES {
62              nextState0 = trellis_get_next_state(state, INPUT_0);
63              input_of_0 += alpha[bit][state] * gamma[bit][state][↩
                    INPUT_0] * beta[bit][nextState0];
64          }
65
66          // for an input of 1
67          for NUM_STATES {
68              nextState1 = trellis_get_next_state(state, INPUT_1);
69              input_of_1 += alpha[bit][state] * gamma[bit][state][↩
                    INPUT_1] * beta[bit][nextState1];
70          }
71
72          decoder_output[bit] = log(input_1/input_0);
73      }
74  }
```

```
75
76 MAP_decode(systematic[NUM_BITS],
77            parity[NUM_BITS],
78            extrinsic[NUM_BITS],
79            noise,
80            decoder_output[NUM_BITS])
81 {
82     calculate_gamma(systematic[NUM_BITS],
83                     parity[NUM_BITS],
84                     extrinsic[NUM_BITS],
85                     noise,
86                     gamma_output[NUM_BITS]);
87
88     calculate_alpha(gamma_output, alpha_output);
89
90     calculate_beta(gamma_output, beta_output);
91
92     calculate_LLR(gamma_output,
93                   alpha_output,
94                   beta_output,
95                   decoder_output);
96 }
```

8.2 Apprendix B: MAX-MAP Decoder Psuedocode

Listing 8.2: Psuedocode for the MAX-MAP decoder variant.

```
1
2 calculate_gamma(systematic[NUM_BITS],
3                 parity[NUM_BITS],
4                 extrinsic[NUM_BITS],
```

```
5                 noise,
6                 gamma_output[NUM_BITS][NUM_STATES][NUM_INPUTS])
7 {
8     for NUM_BITS {
9         for NUM_STATES {
10            //input is either 0 or 1
11            for NUM_INPUTS {
12                //uk = input
13                xk = trellis_get_output(state,input);
14                //Just remove the exponent from the MAP implementation
15                gamma_output[bit][state][input] = 0.5*noise*parity[bit↩
                      ]*xk + 0.5*uk*(extrinsic[bit]+noise*systematic[bit↩
                      ])
16
17            }
18        }
19    }
20 }
21
22 calculate_alpha(gamma_output[NUM_BITS][NUM_STATES][NUM_INPUTS],
23                alpha_output[NUM_BITS][NUM_STATES])
24 {
25    for NUM_BITS {
26        for NUM_STATES {
27            previousState0 = trellis_get_previous_state(state, INPUT_0↩
                  );
28            previousState1 = trellis_get_previous_state(state, INPUT_1↩
                  );
29
30            // Turn the multiplications into additions
31            alpha0 = alpha[bit][previousState0] + gamma[bit][↩
                  previousState0][INPUT_0];
```

```
32              alpha1 = alpha[bit][previousState1] + gamma[bit][↩
                    previousState1][INPUT_1];
33              // Additions turn into the max operator
34              //take the max of the two
35              alpha[bit+1][state] = max(alpha0, alpha1);
36          }
37      }
38  }
39
40  calculate_beta(gamma_output[NUM_BITS][NUM_STATES][NUM_INPUTS],
41                  beta_output[NUM_BITS][NUM_STATES])
42  {
43      for NUM_BITS down to 0 {
44          for NUM_STATES {
45              nextState0 = trellis_get_next_state(state, INPUT_0);
46              nextState1 = trellis_get_next_state(state, INPUT_1);
47
48              // Turn the multiplications into additions
49              beta0 = beta[bit][nextState0] + gamma[bit][nextState0][↩
                    INPUT_0];
50              beta1 = beta[bit][nextState1] + gamma[bit][nextState1][↩
                    INPUT_1];
51
52              // Additions turn into the max operator
53              //take the max of the two
54              beta[bit-1][state] = max(beta0, beta1);
55          }
56      }
57  }
58
59  calculate_LLR(gamma_output,
60                alpha_output,
61                beta_output,
```

```
62                decoder_output)
63 {
64     for NUM_BITS {
65         max0 = -inf;
66         max1 = -inf;
67 
68         // for an input of 0
69         for NUM_STATES {
70             nextState0 = trellis_get_next_state(state, INPUT_0);
71             // Multiplications turn into additions
72             input_of_0 = alpha[bit][state] + gamma[bit][state][INPUT_0↩
                   ] + beta[bit][nextState0];
73             // Additions turn into max operator
74             max0 = max(max0, input_of_0);
75         }
76 
77         // for an input of 1
78         for NUM_STATES {
79             nextState1 = trellis_get_next_state(state, INPUT_1);
80             // Multiplications turn into additions
81             input_of_1 = alpha[bit][state] + gamma[bit][state][INPUT_1↩
                   ] + beta[bit][nextState1];
82             // Additions turn into max operator
83             max1 = max(max1, input_of_1);
84         }
85 
86         // Log domain use subtraction
87         decoder_output[bit] = max1 - max0;
88     }
89 }
90 
91 MAP_decode(systematic[NUM_BITS],
92            parity[NUM_BITS],
```

```
93              extrinsic[NUM_BITS],
94              noise,
95              decoder_output[NUM_BITS])
96  {
97      calculate_gamma(systematic[NUM_BITS],
98                      parity[NUM_BITS],
99                      extrinsic[NUM_BITS],
100                     noise,
101                     gamma_output[NUM_BITS]);
102
103     calculate_alpha(gamma_output, alpha_output);
104
105     calculate_beta(gamma_output, beta_output);
106
107     calculate_LLR(gamma_output,
108                   alpha_output,
109                   beta_output,
110                   decoder_output);
111 }
```