UNIVERSITY OF CALIFORNIA

Los Angeles

FPGA Implementation of Decoders for CRC-Aided Tail-biting Convolutional Codes.

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Electrical and Computer Engineering

by

Chester Hulse

2022

ABSTRACT OF THE THESIS

FPGA Implementation of Decoders for CRC-Aided Tail-biting Convolutional Codes.

by

Chester Hulse
Master of Science in Electrical and Computer Engineering
University of California, Los Angeles, 2022
Professor Richard D. Wesel, Chair

The reliable communication of short messages provides a foundation for today's information ecosystem. Text messages, control messages that initiate and manage calls on the cellular network, and messages from millions of sensors in the internet of things all need to communicate short messages promptly and reliably. For short messages, list Viterbi decoding (LVD) of tail-biting convolutional codes (TBCCs) aided by a cyclic redundancy check (CRC) has been shown to approach the random-coding union bound on frame error rate. There are two alternative approaches to LVD, serial LVD (S-LVD) and parallel LVD (P-LVD). Both S-LVD and P-LVD approach maximum-likelihood (ML) decoding performance as the maximum list size is increased. While several recent papers have focused on serial LVD, parallel LVD offers significant structural advantages for an implementation on a field-programmable gate-array (FPGA) board. This thesis presents a complete FPGA implementation of P-LVD and analyzes its performance. The thesis begins by introducing the general LVD paradigm and comparing P-LVD with S-LVD in terms of throughput and computational complexity. An adaptive version of P-LVD allows the list size to grow as with S-LVD. The thesis investigates various hardware architectures, culminating with selection of the best trade-off between throughput and FPGA resource requirements. The conclusion describes several directions for future work on this exciting and relevant area at the crossroads of cutting edge communication theory and practical communication system implementation.

The thesis of Chester Hulse is approved.

University of California, Los Angeles

2022

To my family, and the EE Buds.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

This thesis is a culmination of my work on Parallel Viterbi Decoding during my time at UCLA. I'd like to thank Professor Wesel for originally getting me interested in communications during my undergraduate education. He has been a very kind and patient advisor as I've learned as much as I could from him. Professor Wesel been a great resource in all things communication, but also soft skills like communicating and working with large interdisciplinary teams. He helped connect me with my first internship and kick-started my career both academically and professionally, and I can't thank him enough. Additionally, I'd like to thank everyone else I've worked with at CSL, including Jonathan Nguyen, Caleb Terril, Sean Chen, Linfang Wang, and Calvin Kuo.

Finally, I'd like to acknowledge Bill Ryan's great mentorship on Viterbi Decoding. He has been invaluable in contributing intuition and years of experience in the field. Additionally, Jacob King was a great help in helping me get introduced to simulating and working with different forms of the CRC-Aided Convolutional Code used in this paper.

VITA

2021	B.S. (Computer Engineering), UCLA, Los Angeles, California
2019	Lab Intern, Physical Optics Corporation
2020, 2021	Firmware Intern, SpaceX
2022	DSP & Communications Intern, SpaceX

PUBLICATIONS

C. Terrill, L. Wang, S. Chen, **C. Hulse**, C. Kuo, R. Wesel, D. Divsalar, ”FPGA Implementations of Layered MinSum LDPC Decoders Using RCQ Message Passing”, in *2021 IEEE Global Communications Conference (GLOBECOM)*, Dec. 2021

L. Wang, C. Terrill, M. Stark, Z. Li, S. Chen, **C. Hulse**, C. Kuo, R. D. Wesel, ”Reconstruction-Computation-Quantization (RCQ): A Paradigm for Low Bit Width LDPC Decoding,” in *IEEE Transactions on Communications*, Apr. 2022

J. Nguyen, L. Wang, **C. Hulse**, S. Dani, A. Antonini, T. Chauvin, D. Dariush, R. Wesel, “Neural Normalized Min-Sum Message-Passing vs. Viterbi Decoding for the CCSDS Line Product Code”, in *IEEE International Conference on Communications (ICC)*, May 2022

CHAPTER 1

Background

Tail-biting convolutional codes (TBCC) are codes used in LTE and other communication standards because of their easily configure-able block length and rates, combined with their low communication overhead and simple decoding process[KKY22]. Previous work by our lab has shown great success in creating TBCCs aided with CRCs to achieve excellent performance, and this thesis aims to explore a practical hardware accelerator for these types of decoders. This thesis focuses on rate 1/k convolutional codes, which have their uses in channels with low capacities, as shown by Shannon[Sha48]. That said, work done by Srinivasan[SP09], has shown that these low rate codes can be expanded to have much more flexibility than taken at first glance.

1.1 Convolutional Coding

Convolutional Codes are codes that have been used for decades to encode streams or frames of data. This encoding process can be a continuous convolution over a stream of bits, but in wireless communications the focus is on framed data. Here, a stream of data is split up into many chunks of a specific size. When a chunk is sent, the receiver can properly decode it and use it, or if it misses a chunk it can succesfully request retransmission, or handle the missing chunk another way. This fits naturally to most computer and data transmission architectures.

A convolutional encoder at a basic level can be represented with k memory elements, each storing 1 bit. An input bit is fed in from the left, and the outputs and inputs of each memory element are fed into various XOR blocks defined by the polynomial representing the convolutional code[JZ15]. An example encoder is shown below, where U is the incoming message stream (1 bit at a time), and C1 and C2 are the corresponding 2 output bits. This example code was taken and modified with permission from Professor Wesel's lecture on Convolutional Coding.

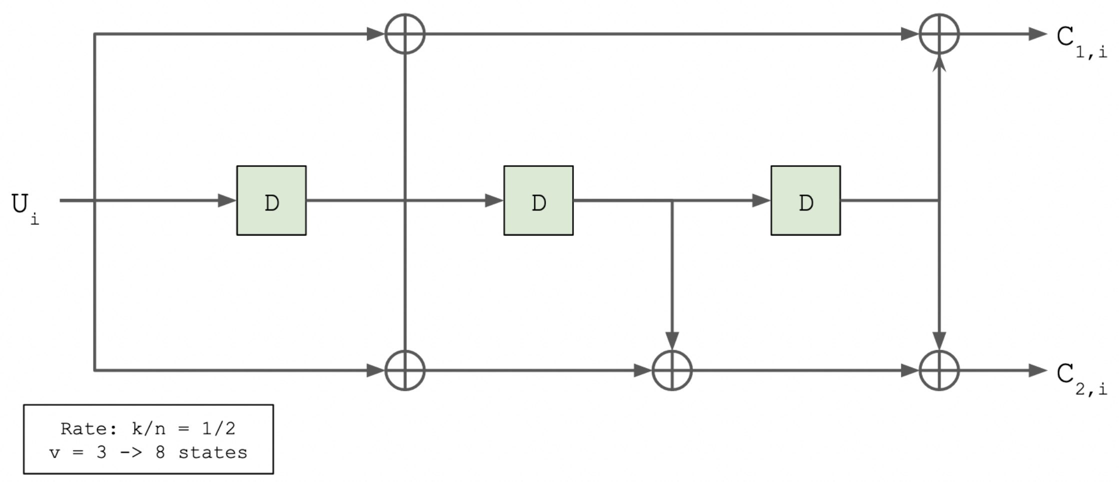


Figure 1.1: A rate 1/2 convolutional encoder with 3 memory elements. The output bits are related to both what is currently in a memory element, and what is being passed into the input data stream.

1.1.1 State Diagram

The convolutional code, represented with memory elements, can also be depicted as a state diagram. In this state diagram, the transition from one state to another is where the encoded messages are actually stored. A state diagram for an encoder with 3 memory elements will have 2^3 states to represent the 8 possible configurations those memory elements could be in, and edges between states will be determined by the input bit U.

These edges each correspond to a specific symbol, here there is a rate 1/2 code, so for every 1 bit of our message there are 2 encoded bits that correspond to 1 of 2 state transitions. For example, an input bit of 1 at state "101" will have an output symbol of "11".

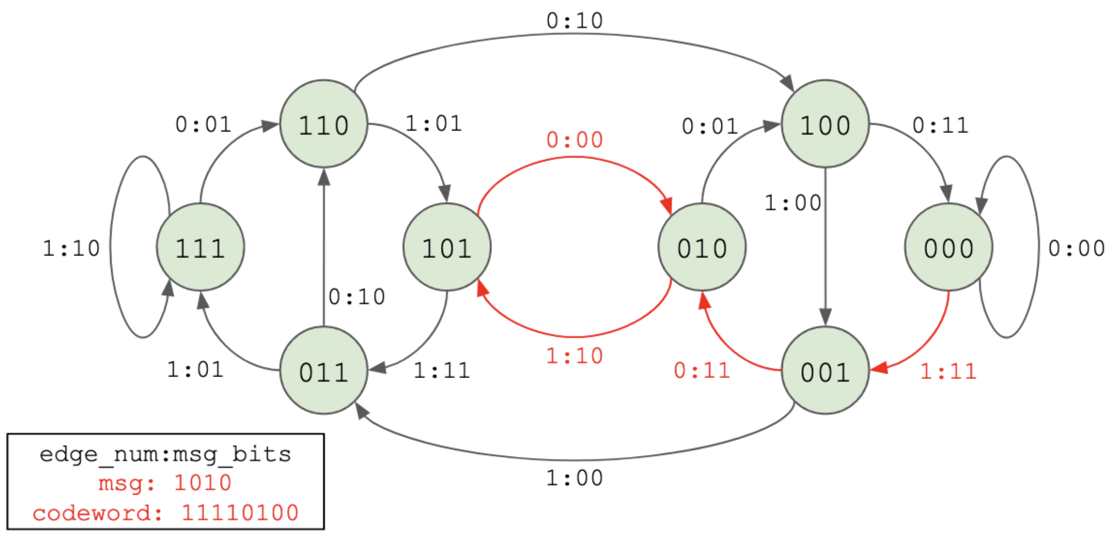


Figure 1.2: A state diagram for the same rate 1/2 code, showing 8 possible states. The edges between each state represent the corresponding output and state transition from each possible input.

1.1.2 Trellis Interpretation

From this state diagram, one can take the set of 8 states and replicate it for every bit expected in an output message. Edges between each replication still connect each state the same way they did before. Each edge has an edge weight corresponding to the expected symbol assuming no noise. This is now called a trellis, and an example with this same rate 1/2 code is demonstrated in Figure 1.3.

The edges between each state, as in the state diagram, are transitions that correspond to the encoded message. If the received message had no errors, there would be one path that exactly matches the received codeword.

In the real world, there will be noise added to the received message. There needs to be

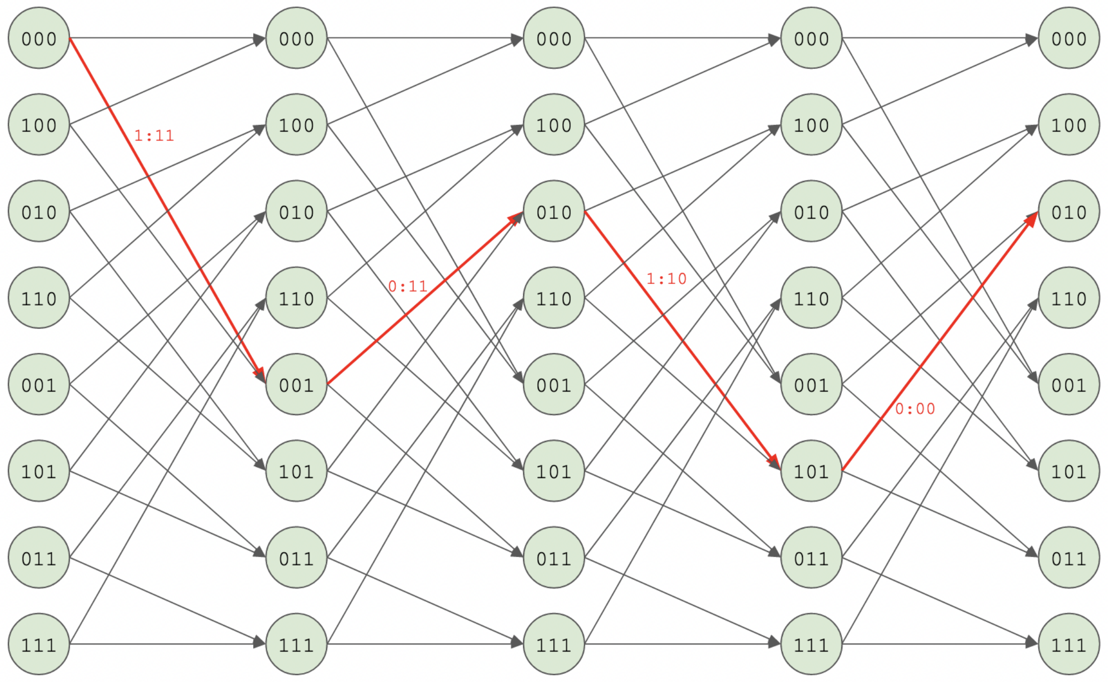
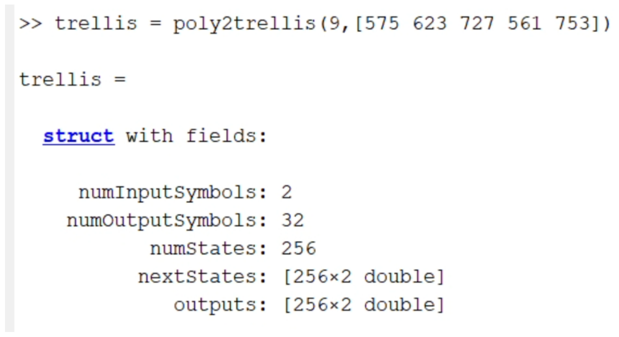


Figure 1.3: A trellis representation of the same rate 1/2 code. The red lines still represent the same path through the state diagram, but now it is easy to see each time point and the path spread out in trellis form.

a quantitative way to measure how close the received symbol matches the edge, we call this edge cost a "metric". A better branch metric means that the symbols received more closely match the expected symbol, and a worse metric means the symbols are further away.

The simplest way to quantify this metric would be the minimum euclidean distance from the symbol, $\sqrt{\sum_i (R_i - C_i)^2}$ where R_i is the received bit and C_i is what was transmitted for that branch, but later this thesis will present a way of simplifying this computation to make it more feasible in hardware.

Many of the examples in this thesis go over a trellis or code that is 8 or fewer states. This is easier to illustrate and follow, but the code that the actual hardware accelerator uses is a rate 1/5 convolutional code consisting of 32 message bits and 11 CRC bits. The CRC will be introduced in section 1.3, but it is simply another step in the encoding process to improve error correction capabilities. This means, in total, there are 43*5 received bits per frame. These bits are represented with Log-Likelihood Ratios (LLRs), which is $Log(P(X = 0)/P(X = 1)$. The polynomial specifying this convolutional code is: (575, 623, 727, 561, 753) in octal. To generate a trellis, including state mappings and edge weights, MATLAB can be very helpful, as seen in the figure below.



```
>> trellis = poly2trellis(9,[575 623 727 561 753])

trellis = 

  struct with fields:

     numInputSymbols: 2
    numOutputSymbols: 32
           numStates: 256
          nextStates: [256×2 double]
             outputs: [256×2 double]
```

Figure 1.4: MATLAB code for the convolutional code used in the paper, with 256 states.

1.1.3 Maximum Likelihood Decoding

To decode a received message, the goal is to find the most likely valid path through the trellis, given a noisy input. In terms of metric, this is the path with the lowest "cost" through the trellis. To actually find the most valid path, for a block length of k bits, one would need to compare 2^k different paths. For a point of reference, much of the work done in this paper relates to a code of length 43, which would have $8.29 * 10^{12}$ different possible paths. This would incredibly computationally expensive. Vitirbi invented an algorithm for decoding convolutional codes, which was popularized by Forney as a practical method to actually decode it[Vit67][For73]. Instead, the List Viterbi Algorithm is commonly used as an approximation of Maximum Likelihood Decoding.

1.1.4 Viterbi Decoding

Given a received message that is n bits long, the decoder needs to find the most-likely path through the trellis to arrive at the k-bit message that was encoded. First the convolutional encoding must be reversed with the List Viterbi Algorithm (LVA). This is spelled out very well in Seshadri's paper[SS94], but the key parts will be summarized below.

1.1.5 Parallel List Viterbi Decoding

The Parallel List Viterbi Algorithm is a parallel method for finding the best L paths through the trellis that minimize the Euclidean distance. It is defined as following:

1. Initialize all states to have a list of incoming paths. In this case, each state will have 1 incoming path of metric 0 and initial state set to the current state.

2. Of the maximum 2L incoming paths to each state, take the L paths with best metrics. This will grow as a power of 2 until reaching L (1, 2, 4, ... L).

3. Each state with incoming paths has 2 outgoing edges. Each edge has a corresponding metric associated with this time point given the received bits. Add this branch metric to every path, and send it along the edge to the next state.

4. Repeat from step 2 until reaching the ending (K+m) time points. There is now a separate list of L paths for each state that could be the valid decoded message.

1.1.6 Serial List Viterbi Decoding

Though not the focus of this thesis, serial LVA is another way to decode these codes that should be mentioned for comparison. From a high level, it finds the first best path through the trellis, and then stores enough information with this path to be able to back track to the second best path, while also knowing not to come back the same way. It stores enough information in complicated data structures that it can backtrack exactly L times, resulting in L best paths. This is not particularly easy to implement in a hardware accelerator due to the dynamic memory and non-deterministic runtime of such an approach[LY14]. Additionally, because it isn't easy to tell if a path is correct until the end, backtracking can't be parallelized and metric calculations need to be all done sequentially.

After calculating the first best path, the second best path must be identical to the first best path except for diverging once and then coming back. Any other divergences will cause a longer path. This is illustrated well in this figure from Seshadri's paper:

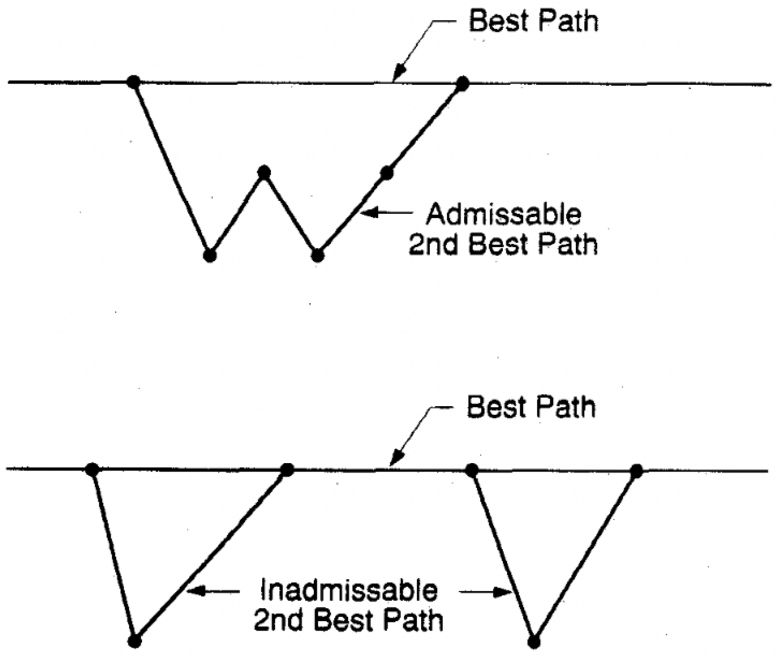


Figure 1.5: Seshadri illustrates a diverging second best path.

This is proved elegantly in Seshadri's paper, but to realize it simply note that every divergence must be equivalent or worse in total metric, so one divergence will have a strictly better overall metric than that same divergence with an additional one somewhere else.

This algorithm requires L * num_timepoints memory of divergent points, along with a "merge" array of the total number of time points.

This approach only computes the metrics necessary for up-to L paths through the trellis. After each path, the decoder has to verify it is correct or find a new one and backtrack. This is computationally very cheap compared to PLVD, but cannot be parallelized easily and has varying runtime depending on how far it has to backtrack. In comparison, PLVD can compute all the metrics for every state in a timep oint at once. It also has a constant runtime and very rigid hardware requirements, whereas the serial version could spend a long time backtracking and re-trying new paths.

1.2 Frame Identification

The data coming in with this approach will be framed to a given frame length (n bits). The receiver will need to sync their symbols to start at the correct part of the frame in order to properly decode every frame. Seshadri's description of LVA is for a more general trellis, with no restriction on beginning or finishing state. To separate frames, this beginning and end state needs to be restricted[CS94].

1.2.1 Traditional Zero-Padding

One very common method of frame syncing is called zero-padding, where every message is padded on either end with zeros to form a very recognizable boundary between frames. This has the relatively intuitive overhead equal to the length of your padding. This would also result in a trellis where the initial and final state is fixed, and all paths branch from it.

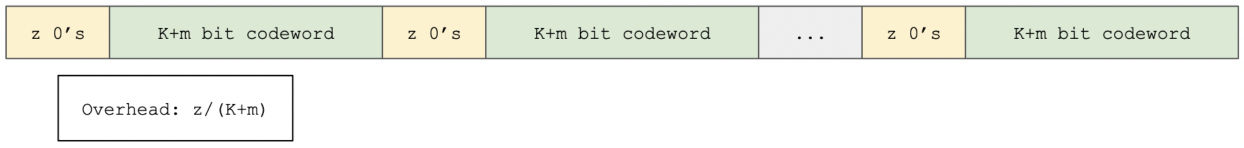


Figure 1.6: Illustration of a sequence of codewords with an all zeros framesync word.

One closely related version of this is inserting a frame-sync symbol instead. This symbol can be constructed to increase performance in certain kinds of channels, but it still leads to wasted overhead compared to a system that can avoid these symbols altogether.

1.2.2 Tail-biting Approach

The alternative that this thesis takes, called tail-biting (TB), is able to frame sync without these sync words. This method was originally proposed by Howard Ma[MW86]. TB codewords are codewords that begin and end at the same state. In practice, this means the decoder will run the decoding algorithm as stated, but at the end instead of just picking the path with the best metric, it will also verify that the path is in fact starting and ending at the same state. If not, it will simply check the next best path until it finds one that does satisfy this TB condition. Once the decoder locks on to a correct frame, it can find the next codeword easily by just skipping the number of bits equal to the pre-defined codeword length.

This TB condition will require some adaptations to Seshadri's LVA approach. Running the decoding process once for every state to find all tail-biting paths would not be feasible, so instead this decoder will use a process called Wrap Around Viterbi Algorithm (WAVA) to increase the number of TB paths. More details are in Chapter 2.

1.3 Cyclic Redundancy Checks

Finally, convolutional coding on its own may not have good enough error correction capabilities. These capabilities can be increased by wrapping the convolutional encoder with a Cyclic Redundancy Check (CRC). The design of this CRC is a whole topic on its own, but examples of suitable CRCs and the varying performance gains are given by [YLP22]. At the end of the LVA process, a decoder would then need to verify the CRC of the best paths that it gets, in addition to the tail-biting condition, before finally returning the path that satisfies both of these as its best-guess most-likely codeword. In this paper, K is defined to be the length of the message, and m is the length of the appended CRC. The total input message length is then K+m. This message is then passed through the convolutional encoder to get an encoded message of length (K+m)/r, where r is the rate of the convolutional code.

In summary, we want to find the minimum cost path through the trellis, using LVA, that is both tail-biting and has a valid CRC.

CHAPTER 2

Optimizations and Complexity Analysis

This chapter takes the established P-LVD and presents computational optimizations and available parallelism before analyzing the computational and memory complexities. Additionally, it introduces Adaptive P-LVD and describes how the complexity can scale differently as the maximum list size (L) changes.

2.1 Optimizations

The algorithm specified in Seshadri's paper doesn't go into actual implementation details, but some of these are crucial in reducing the minimum execution time and simplifying hardware. Additionally, these optimizations will be included in complexity analysis because they are important pieces of the hardware architectures proposed in the next sections.

2.1.1 Metric Calculation

Earlier, a "metric" was introduced as the cost to transition from one state to another given the input symbols. In traditional communications, this would be represented by Euclidean Distance, the closer state is more likely and would have a smaller Euclidean Distance in the space that includes all valid codewords.

One quick simplification to reduce computational complexity lies in how we compute the cost to go along an edge. Consider minimizing the Euclidean Distance, where R is the received set of bits and C is the correct set of bits for that state transition, $(C_1, C_2, ...C_n)$

where n is set by the rate of the code:

$$\arg\min_C \sum_i (R_i - C_i)^2 \tag{2.1}$$

Which expands to:

$$\arg\min_C \sum_i (R_i^2 - 2R_iC_i - C_i^2)$$

For a given time point (or a given set of 5 LLRs for a rate 1/5th code), R_i is constant when comparing between all states, so the R_i^2 term can be dropped:

$$\arg\min_C \sum_i (-2R_iC_i - C_i^2)$$

From here, C will always be ± 1, so minimize the following:

$$\arg\min_C \sum_i (-2R_iC_i - 1)$$

Again, the constant 1 can be dropped:

$$\arg\min_C \sum_i (-2R_iC_i)$$

Finally, drop the constant -2 and just maximize:

$$\arg\max_C \sum_i (R_iC_i) \tag{2.2}$$

Each edge will have a different C, so there will be a distinct metric value computed for every edge, but these are all easy to compute in parallel.

When moving this to hardware, it is as simple as taking the input LLRs, deciding to invert them based on the value of C for the current edge (BPSK means C is ± 1), and then summing it to the metric of the incoming path. The overall metric or cost of a path through a trellis will be the sum of all these edge weights when multiplied with their corresponding LLRs for each time point.

2.1.2 Branch to Path Metrics

After using the above method to compute a metric that is just inverting a number ($\pm R$), there is another simplification to be made. In this type of code, every state has 2 outgoing edges, and the weights corresponding to these edges are bit-wise opposites of each other. For example, a state may have, in binary, one edge weight that is 01010, and the other outgoing edge would be 10101. If the branch metric of one edge is computed as normal, the other one is simply the negative of the already computed branch metric. For every state, at every time point, only one branch metric needs to be computed instead of 2.

After the branch metric is computed for a state, simply take the list of L best incoming paths, and then produce 2 outgoing paths from each incoming path; one outgoing path with the branch metric added going to the first output state, and the other with the branch metric subtracted going to the second.

2.1.3 Independent Parallelism

Though not strictly a computational optimization, PLVD has an advantage over SLVD in terms of parallelism. At every time point there is opportunity for incredible amounts of independent computation. Each time point for a rate 1/5th code is associated with 5 LLRs and 2 outgoing edges for every state. Each of these 5 LLRs needs to be summed corresponding to the proper edge weights to an intermediate edge cost, and this resulting cost needs to be added to every outgoing path along that edge. All of these additions can be pipelined to be ready by the time the path is at this point, and every state can add the intermediate cost to every path in parallel. This makes the decoder runtime fully dependant on the number of timepoints, or bits in the message, and the maximum list size, assuming the list is dealt with in series.

2.1.4 Merge Sort

One crucial piece of the PLVD algorithm is picking the best L paths from 2L incoming at every state. A naive approach would be to just take all the 2*L input paths, add their intermediate costs, and then sort them and take the top L off of it.

A better approach is to assume all the lists are sorted. When 2 batches of sorted lists from 2 distinct states arrive, merge sort them and keep the output sorted, dropping the worst L paths. This merge sort optimization assumes the decoder is dealing with each list entry in series, but for list sizes that can grow to the hundreds or even thousands, it would be impractical to parallelize with respect to L, so this optimization in keeping them sorted is worth it and doesn't leave performance on the table.

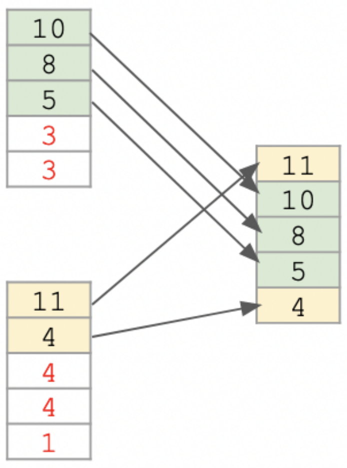


Figure 2.1: Merge sort example, taking in 2 batches of L paths and returning a single sorted batch of L.

Additionally, forcing the lists to be sorted doesn't restrict our algorithm in any way. The decoder starts with 1 path per state, which by definition is sorted. When merging paths from 2 distinct states, it is trivial to keep the output sorted and continue to grow from there. The algorithm starts with incoming path lists of size 1, and from there goes to 1,2,4,8,...L paths, so at every step it can stay sorted.

2.2 Adaptive Parallel LVA

Adaptive Parallel LVA is a paradigm where the maximum list size of the decoder grows until a correct codeword is found or the decoder gives up. Because runtime scales proportionally with the maximum list size, an ideal decoder would pick exactly the minimum list size that results in the correct codeword. In practice, there is no way to know what this L would be beforehand, but having an adjustable maximum list size allows system designers to make a valuable choice between latency and throughput averages and maximums.

As an example, because runtime is directly proportional to L, an adaptive decoder A could grow list sizes like the following: $1 + 2 + 3 + 4 = 10$, or have an adaptive decoder B that tries $1 + 8 = 9$. Depending on the expected list size and SNR these could have vastly different performance metrics. At high SNRs it is very likely to decode at just a list size of 1 or 2, as seen in Figure 2.4.

From these numbers, and desired performance targets, a systems designer could pick optimal numbers to maximize throughput and error correction capabilities based on the given SNR region. A hardware accelerator should be able to take advantage of such performance ideas by having a tune-able amount of parallelism and list size.

2.2.1 WAVA

The PLVD approach so far starts every state with equal footing and no way of ensuring TB paths. It hopes that at the end some of the states are tail-biting, and checks, but there is a chance the correct path lost to an incorrect, non tail-biting path somewhere early on and never made it. To ensure TB paths one could dedicate an entire trellis to only one initial state, and only find paths originating from that state. At the end, this algorithm would be guaranteed to have L TB paths for the specified state, and could then be repeated for every state. This is called the N-trellis approach[YLP22], and would be impractical as computational costs now scale proportionally S, which in this example is 256.

A better approach to maximize TB paths is something called the Wrap Around Viterbi Algorithm (WAVA)[SLF03]. This algorithm aims to decode using only 1 trellis, but boost the chances that the most likely ending states are also the most likely initial states. To do this, the decoder runs through the trellis with a list size of L=1, and finds the best metric for each ending state. This metric is then set to be the initial metric at the beginning of the decoding process, and the decoder starts from there instead of 0. If WAVA was wrong it might hurt some of the paths from that state, but Figures 2.2 and 2.3 show that even when wrong WAVA can still arrive at the right codeword, and it boosts the likelihood of TB paths significantly when compared to a non-WAVA PLVD decoder.

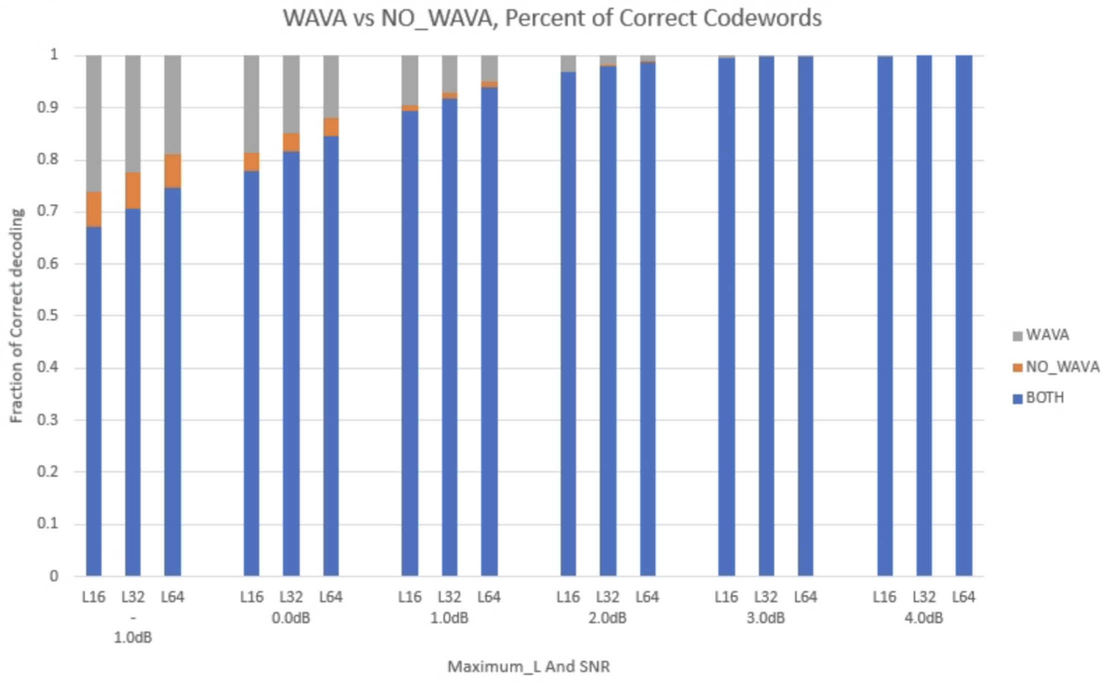


Figure 2.2: Stacked bar chart comparing the fraction of decoding attempts where WAVA achieves the same results as NOWAVA, WAVA succeeds while NOWAVA doesn't, or vice versa. As the SNR increases, they are both more likely to decode properly, but if one gets it the same trend follows where it is more likely to be WAVA.

Figure 2.2 illustrates of the correctly decoded words, what fraction required WAVA to get the right codeword. As SNR increases it gets easier to decode and harder to show this trend, so Figure 2.3 will show total error rate for an A-PLVD decoder. As you can see, WAVA has a significantly better frame error rate even when compared to WAVA of greater list sizes.

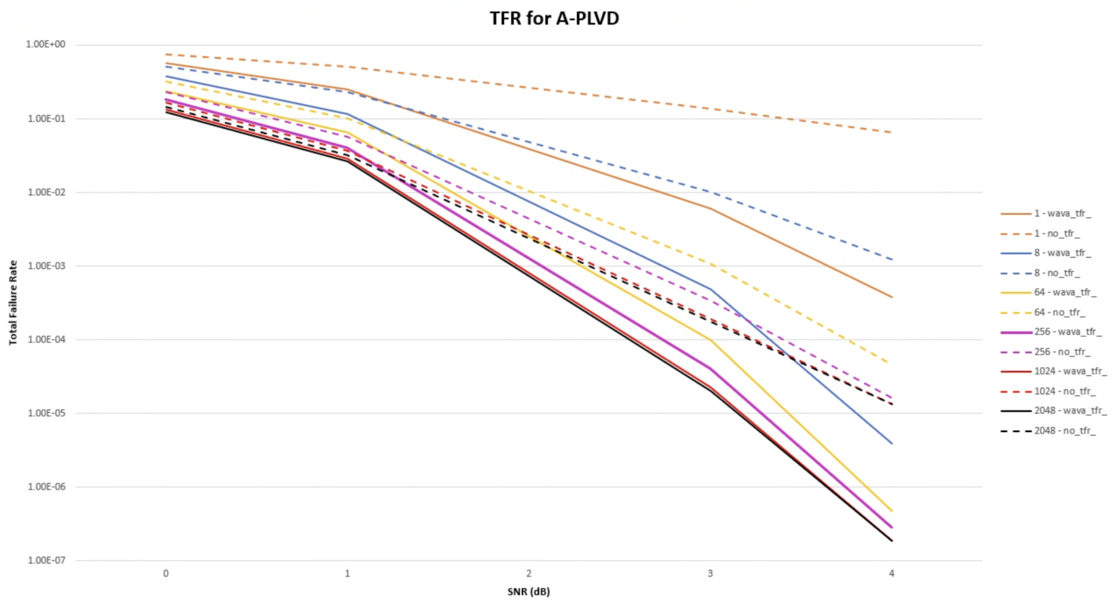


Figure 2.3: Total Failure Rate (TFR) curves comparing a WAVA initialized A-PLVD decoder to a non-WAVA A-PLVD one.

At high SNR this adaptive decoder is frequently decoding at a list size of 1, so when there is an occasional error, it is often due to an unlucky passing CRC rather than exhausting the list size and giving up. WAVA helps avoid this error by biasing the states that are more likely to be correct final states, but it doesn't fully correct for this problem. A new version of A-PLVD is discussed in the "Future Work" section below that can avoid this issue.

2.3 Parallel LVA Performance

The example TBCC-CRC code used in this paper consists of a few main parameters that will be helpful to define. Mainly:

1. num_states = S = 256
2. code_size = K = 32
3. crc_size = n = 11
4. block_length = N = 215
5. rate = r = (K+n)/N = 1/5

2.3.1 Memory

Because each state needs to keep track of 2*L paths, the overall LVA decoder needs 2*L*S memory elements. There are two ways of doing this, one where store the paths before comparison, and one where you store them after, but it makes computation and memory access patterns much simpler to store before comparing.

2.3.2 Computation

The computation for Parallel LVA can be greatly simplified if we assume all lists of paths are stored in sorted order. Moreover, if we keep things sorted from the start, we can keep them sorted the entire time very easily, so this comes for free.

In the case where all the lists are sorted, the best you can do computationally is S*L*(K+m) additions and comparisons, and S*L*(K+m)/r additions for computing the metrics.

2.3.3 Runtime

The only data dependency happens when moving from one time point to the next. Technically all additions for metrics, and even comparisons for keeping paths sorted could be done in parallel. This would be outrageously expensive. The hardware architecture in this approach instead looks to deal with 1 path from every state in parallel, so delay is totally dependant on L and the number of time points, and not dependant at all on how many states. This leads to a runtime that scales proportionally with (K+m)*L. All additions for the metrics involved in the rate can be pipelined such that the decoder is never stalling waiting for them to he ready.

2.4 Future Work

The A-PLVD decoder has significant performance benefits by accelerating the most common case where a short list size is sufficient, while still handling the rare cases that require a longer list. Ideally, an A-PLVD decoder would have the same TFR as a non adaptive PLVD, so it would have the speed of a lower list size for most cases but the TFR of a higher list size. As it is currently described, that is not the case; A-PLVD performs slightly worse than a decoder with a fixed list size equal to the maximum list size of the A-PLVD.

WAVA improves performance in some cases by increasing the number of TB paths that appear in smaller lists. However, for longer list sizes the non-ML metric of WAVA introduces decoding errors that increase the undetected error rate by about an order of magnitude.

Even without WAVA, there is a performance loss with A-PLVD that arises because of a subtlety involving how the current algoithm terminates. To achieve maximum likelihood a decoder should asymptotically approach the closest codeword as maximum list size increases. For very large list sizes, A-PLVD should return the lowest-cost tail-biting, crc-passing path through the trellis. Because these paths will always be competing with non-TB paths in the middle of the trellis, many end paths at the end of the trellis aren't TB, and even fewer of those are CRC-passing. The lists for some end states may not contain any paths that are both TB and CRC-passing.

Imagine a case where end state A has a TB path that is CRC-passing at cost C_a, while a state B has no TB, CRC-passing paths, but all of its paths have lower cost than C_a. If the list size is increased, there is the possibility for state B's longer list to include a TB path that is CRC-passing and has cost $C_b < C_a$. To make an A-PLVD decoder truly maximum likelihood, the stopping condition needs to be adjusted to not stop when a longer list might produce a TB, CRC-passing path with a lower cost.

As stated earlier, A-PLVD runs at a list size L, checks if there is a valid TB, CRC-passing codeword, and if not increases L and tries again. This stopping condition should be slight more strict. After identifying a valid TB, CRC-passing codeword, the decoder should check every other state to see if running the decoder with a larger list might find a valid TB, CRC-passing codeword with a lower cost. States that have a metric worse than the best"valid" codeword can be marked "resolved", as they will not produce a TB-CRC passing codeword that has lower cost than the current selection. Additionally, any state with that has identified a TB-CRC passing codeword codeword can be marked "resolved" and its path stored, it won't produce any better passing codewords. The remaining states are "unresolved" because they may still produce a TB, CRC-passing codeword with a lower cost.

PLVD can then be re-run without initializing any paths from the resolved states (and optionally with an even higher list size), until all states are resolved. Once all states have been resolved, the ML codeword has been identified. If the maximum list size is reached while there are still unresolved states, the lowest-cost TB, CRC-passing codeword can be selected as the decoding result, but it is not necessarily the ML decoding result.

This new approach will be a very interesting direction to explore. The following two figures consider the new approach, which we call resolution terminated adaptive parallel list Viterbi decoding, (RTA-PLVD) where the list size is increased until all states are resolved or the maximum list size is reached. Simulation results in Figures 2.4 and 2.5 show that RTA-PLVD outperforms the original A-PLVD decoder for a large enough maximum list size, and reduces the undetected error rate by a factor of 2. The TFR for RTA-PLVD is, of course, higher for small list sizes because A-PLVD has a weaker stopping requirement. So A-PLVD is quicker to decode a correct codeword but also can select non-ML codewords, which leads to the larger UER. Asymptotically as maximum list size is increased, RTA-PLVD will have both a lower TFR and UER.

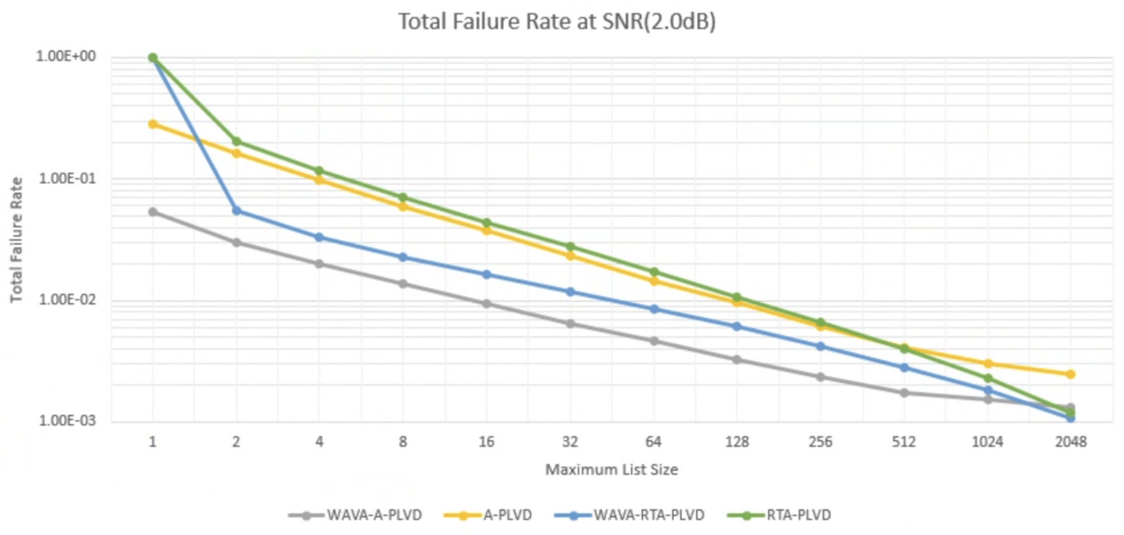


Figure 2.4: Total Failure Rate curves comparing a WAVA initialized A-PLVD decoder to a non-WAVA A-PLVD one, with both the normal stopping condition and the RTA-PLVD one.

WAVA sacrifices the use of an ML metric to reduce list size. Fig. 2.5 is revealing because it matches the intuition that with a large enough list size the non-ML metric of WAVA initialization comes at the cost of increased UER. Previously, we didn't see this effect in A-PLVD because a non-ML codeword would stop the decoding process too early leading to degraded results even with the ML metric. Not only does RTA-PLVD without WAVA match the RTA-PLVD with WAVA at larger list sizes, but it does so with nearly an order of magnitude lower UER, as seen in Figure 2.5 at maximum list size 2048.

This RTA-PLVD approach is a very rich area for further work. Right now, a quick simulation shows promising results, but more research needs to be done on the specifics of the implementation. Specifically:

1. What is the best way to grow the list to reach the first candidate codeword?

2. After reaching a candidate codeword, should the decoder re-run at that list size until

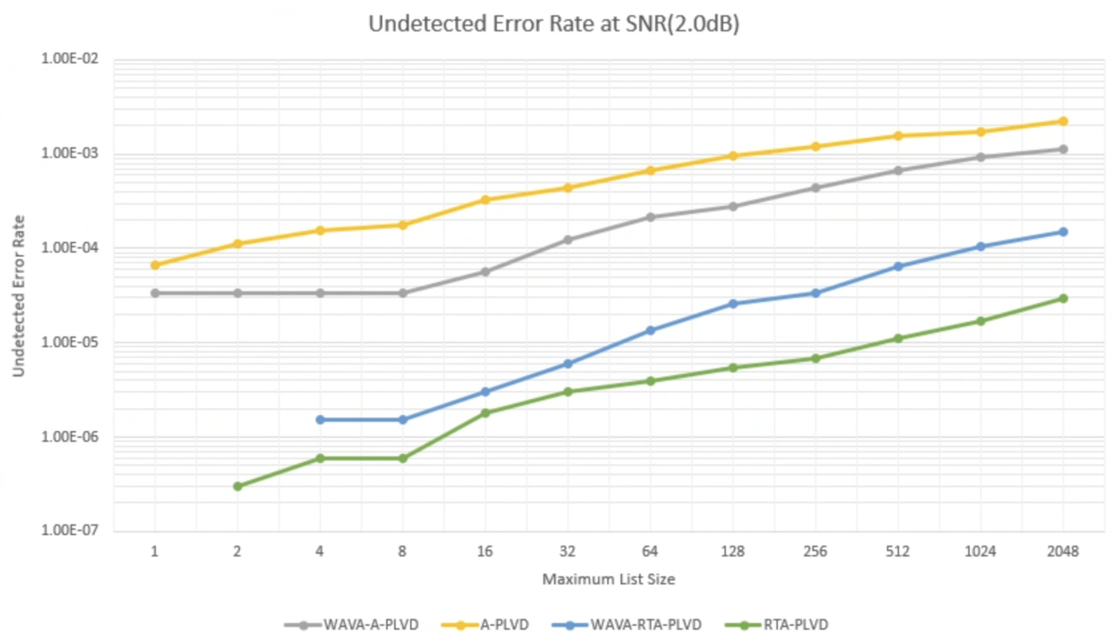


Figure 2.5: Undetected Error Rate curves comparing the probability for the decoder to return an incorrect codeword it thinks is correct.

it gets stuck, or should it increase L in hopes of resolving states faster?

3. Is there a threshold of "enough" resolved states to terminate, speeding up the decoder for a negligible performance hit?

4. Are states resolved more often by reaching "bad" metrics or from actually finding "worse" candidate paths?

These graphs show great improvement at a list size of 2048, but should be applicable with lower maximum list sizes too. Most of this thesis is focused on hardware acceleration of the decoder, and this realization will be applicable to the hardware accelerator, but more work needs to be done on finding the truly optimal RTA-PLVD approach.

CHAPTER 3

Current Accelerator Architecture

3.1 Motivation

Starting out the hardware design, there were two main goals: maximize parallelism within a trellis, and create the accelerator in a way where the maximum list size was configurable. To maximize parallelism, the hardware needed to tackle every state update for a particular time point in parallel, such that the complexity analysis would lead to a runtime purely dependant on L and not related at all to the number of states. As discussed in the next chapter, this may not have actually been the best idea but it was a starting point and the process of building it this way helped learn a couple things that will be applicable regardless of where the design goes in the future.

Figure 3.1 splits the decoder design into a few main parts, including a module for LVA, a tail-bite funnel, a CRC module, and more. Each of these modules are connected by AXI-Streams and described in detail in the following sections.

3.2 AXI-Streams

Communication between all basic building blocks to turn them into the high-level LVA accelerator benefits from standardization. Every module will have some form of data pushed into it and data pulled out of it, and it will need signals to know when data is being pushed from a producer or pulled by a consumer.

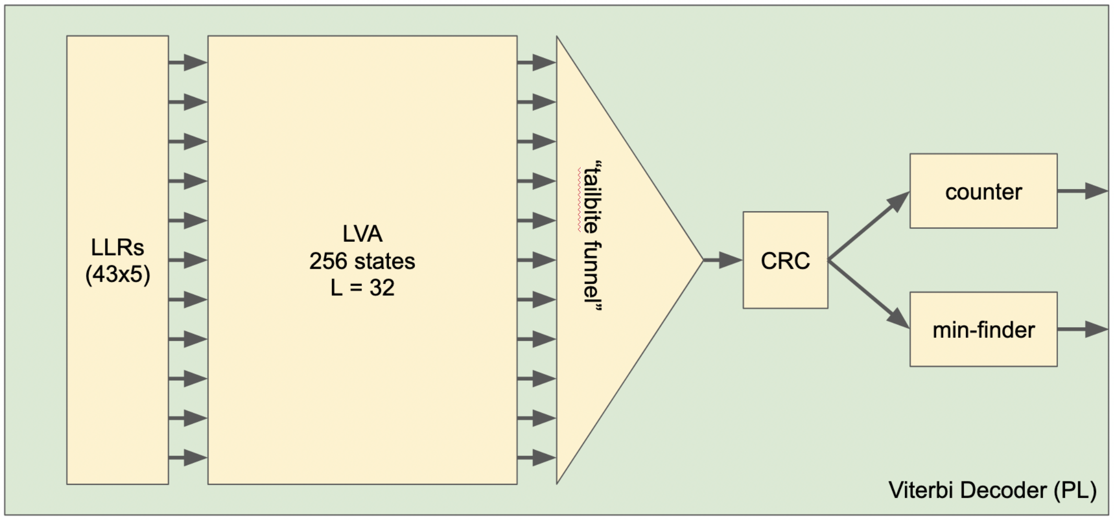


Figure 3.1: High level Programmable Logic (PL) diagram, illustrating the LVA module and then taking its outputs into the tail-bite funnel before CRCing them.

To achieve this, this design used something called AXI-Streams. AXI is a standard communication protocol used by Xilinx to interconnect many of their IPs, but it has been invaluable in connecting modules within the decoder IP.

AXI streams include 5 signals with rules on how they can change with respect to each other. Specifically:

1. **tdata**: Holds the data being sent. If tvalid is high, tdata cannot change until tready also goes high

2. **tvalid**: Set by the sender, this indicates the data is valid.

3. **tlast**: Set by the sender, indicates the end of a "packet". In this case, tlast indicates the end of a group of L paths, and anything after this is from a new time point.

4. **tuser**: Set by the sender, this is specific to whatever IPs are using it. The decoder uses it to identify the beginning of a list of sorted paths.

5. **tready**: Set by the receiver, this indicates that the receiver is ready for data. If both

tready and tvalid are high, the data was successfully transferred, otherwise it should be held valid until tready is high.

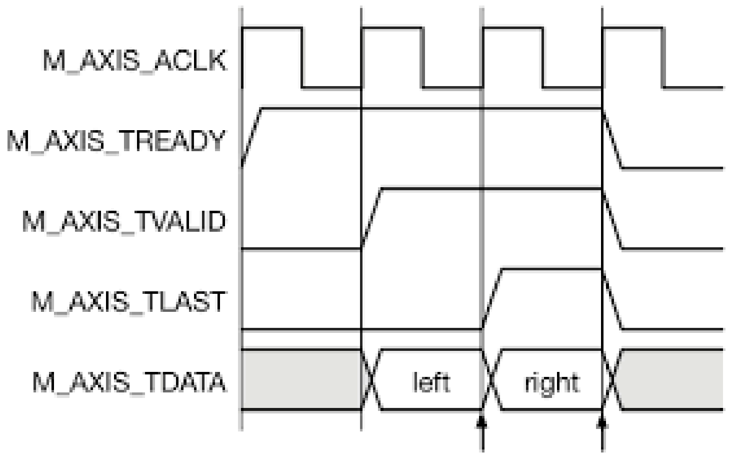


Figure 3.2: AXI-Stream Timing Diagram from MIT slides. tdata changes to a new data packet once tready and tvalid are both high. Additionally, tlast is high for the "right" section indicating it is the last in this packet.

By specifying this as a verilog interface with input and output ports, assertions can be created to enforce rules about when tdata can change, what happens at a reset, and more, and then everywhere in the accelerator is constantly checking for failures (during simulations). This makes it really easy to verify the design and catch bugs sending or receiving data. Before switching the source code to use AXI-Streams, this was done by redefining this communication in a slightly different way for each module. This was a lot more flexible but the flexibility often backfired when it made debugging orders of magnitude more complicated.

Having this standard interface also makes keeping track of certain statistics very easy. For example, counting how many paths are present at each time point requires a simple module that watches tready and tvalid, and increments a counter when they are both high. This same module could be moved to the input of a CRC module, or the output, and count those at the same time with the same, easy to use module. It can't be overstated how useful and helpful this AXI-Stream layer of abstraction was in implementing this decoder. It wasn't a core part of the algorithm or the computations, and could be applied to many different IPs, but it was incredibly useful and should be a starting point for many IPs, especially in there interfaces with other modules.

3.3 Path Object

A path for the decoder was defined by its metric as a fixed point number, a list of "decision" bits indicating either taking a '0' edge or a '1' edge, and finally its initial state to decide if it was tail-biting at the end of the decoding process.

In all, it was a 72 bit structure, made to fit the maximum width of a Block RAM (BRAM), which included 8 bits to define the initial state, 43 bits to define every decision, and then the rest were all dedicated to the metric. This metric width could be trimmed down to a lower value, but was just expanded to fit the rest of the remaining width of a BRAM.

All the AXI streams involving paths had tdata declared wide enough to fit an entire path, a tuser indicating the beginning of a list of paths, and a tlast indicating the end of that list or time point.

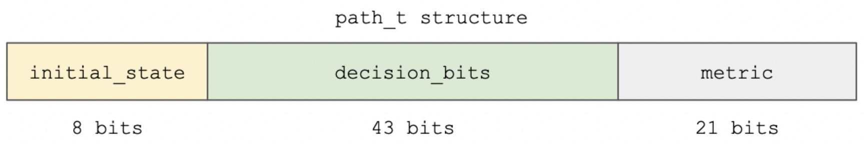


Figure 3.3: A diagram showing a 72 bit long path structure, where bits 0-7 represent the initial state, bits 8-50 as decision bits representing which edge was taken, and bits 51-71 representing the metric of the path as it travels through the trellis.

3.4 LVA Module

LVA in this decoder was done with a state module, edge modules, and some scheduling logic to keep track of the current symbol and what maximum list size the decoder is running at, which is used to run with WAVA before doing a run at a list size of 32. A CSV file describing the trellis, including state mapping and edge weights, was used to generate which states are connected to which edges.

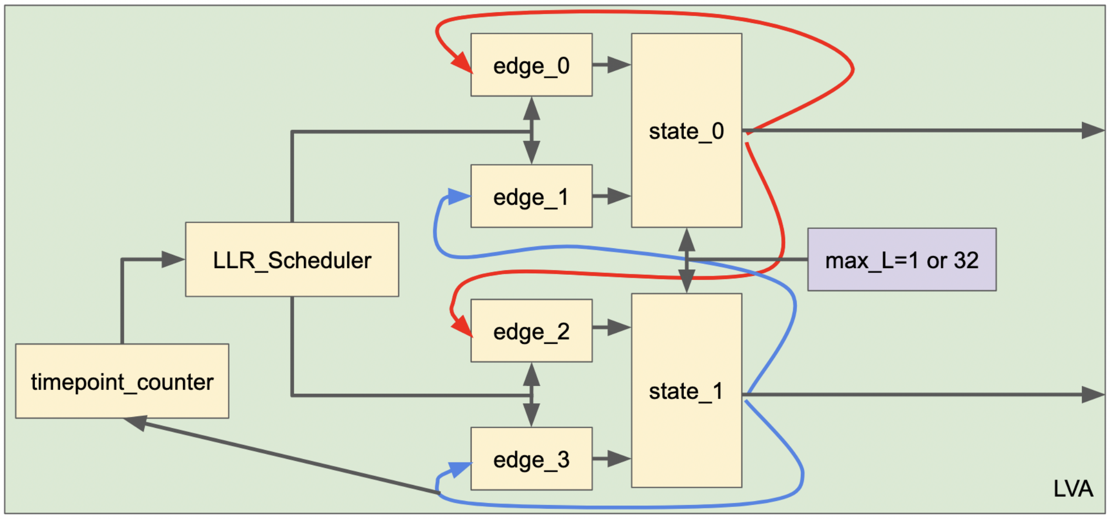


Figure 3.4: High level LVA diagram, illustrating all the different states for a given trellis and how they connect to each other.

3.4.1 State

A state would have 2 memory elements, referred to as Path FIFOs, used to store the 2 sets of incoming paths. It would properly pull the better of the two each clock cycle and eventually output the "L" best paths out of the 2L that came in. This on the surface isn't that difficult, but it was designed to handle edge cases where one set of paths arrives sooner than another, and along the way it verifies that the path is always coming in sorted and raising the proper tuser and tlast signals.

3.4.1.1 Path FIFO

The memory was designed to take in an arbitrary number of paths, assume they were sorted, and pop a new one out whenever requested. This meant that after pulling L paths out of 2 memories for a state, the 2 memories had to dump the rest of their paths out until they reached the next set of paths. This lead to stalls waiting for every state to clear out their memories, which, in the worst case, would take L clock cycles for every time point. Here the design was too flexible and should have restricted memory in such a way to avoid this stall and skip to the next batch of paths. The future design in the next chapter avoids this step by mapping every path to a specific address.

Every state had 2 incoming edges, each with L paths, and needed to combine those 2L paths into L best metrics. Here there was a tradeoff to be made on storing L paths in sorted order. The module could either have an "L" deep memory for each edge, and then combine those paths after the fact, or a single L deep memory for the state and combine them as the paths come in. These are referred to as pre-memory combining and post-memory combining in the following sections.

3.4.1.2 Pre-memory Combining

If the decoder sorts the paths in-place, it could use half as much memory. This is most easily explained with the figure below, where the two lists are stored starting at opposite ends of the memory and then move towards the center until they meet and overwrite the smaller ones that would get thrown away.

That said, after looking into this, if the decoder does the pre-memory combining the decoding process will be 1/2 the speed. BRAM can only access two addresses at once, ideally one of those addresses would be reading old paths and one will be writing new paths at all times. With pre-memory combining, the decoder would need to have a reading and writing phase and alternate between reading current paths and writing the next time-point's

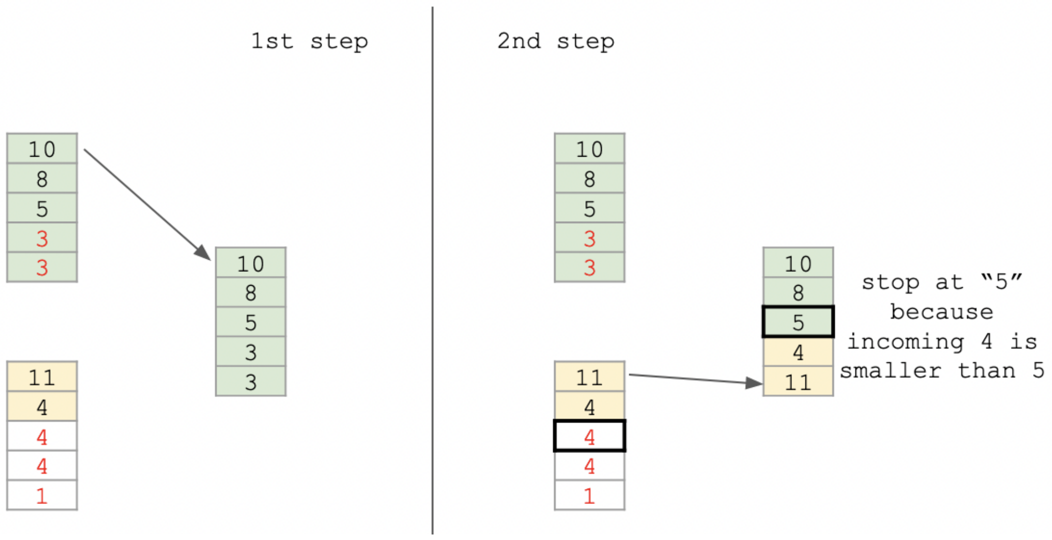


Figure 3.5: Pre-memory combining combines the paths as they are stored into memory, so the decoder would only need L memory per state rather than 2L to store both sets of incoming paths.

paths into the same memory. The next section will show post-memory combining allows for doing both simultaneously. Because the BRAMs are under-utilized in depth as-is, this decoder focused on combining after storing to make it easier to merge sort.

3.4.1.3 Post-memory Combining

This is the version that actually ended up in the synthesized decoder because it seemed faster and could fit. What this does is store the paths in memory as incoming paths, then pull them out to combine them, then store them back into a BRAM afterwards as separate outgoing paths.

The next chapter ends up sharing BRAMs between states, fully utilizing the maximum depth of the BRAM. Even in that case it still makes sense to do Post-memory combining because the limiting factor in all of these computations is how fast the output paths can be written to memory. Stalling writes while reading from the top and bottom side of each

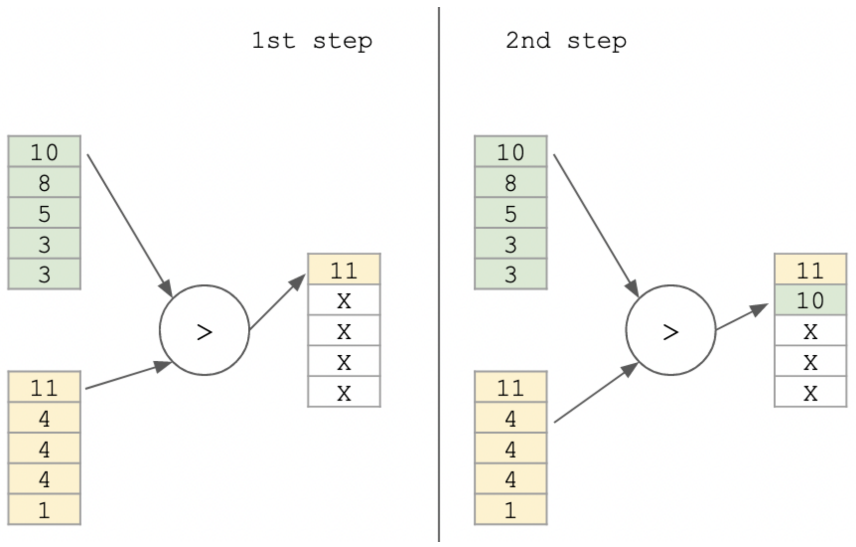


Figure 3.6: Post-memory combining combines the paths after they are stored into memory. This means that each state requires twice as many memory elements but is able to read and write from every BRAM simultaneously in every cycle.

BRAM with pre-memory combining would slow down the decoder more than the memory-savings are worth.

3.4.2 Edge

An edge takes in LLRs for the given time point, sums them up into a `branch_metric`. It holds tready low while it waits for LLRs to come in, and then takes a stream of paths and adds to all of their metrics the fixed `branch_metric` value. Additionally, it makes sure additions saturate to their maximum value without accidentally overflowing.

Because metrics will constantly be growing, a potential optimization includes subtracting off an offset from every edge as the average metric increases. This offset can increase or decrease dynamically depending on the magnitudes of current metrics. Since the decoder doesn't care about the value of the ending metric, just relatively whats best, this doesn't sacrifice any performance but could allow for less bits dedicated to the metric. For example, if a user wanted a trellis with 60 time points instead of 43 and wanted the path struct to fit in the same number of bits, it could take some of those from the overall metric value.

3.5 Tail-bite Funnel

The tail-bite funnel takes in one AXI stream for each state, and checks if they are tail-biting in parallel. If a path is tail-biting, it picks the first one available and spits it out to the next step in an AXI stream. If it is not tail-biting, it discards it and pulls the next path out. This lets the decoder very quickly dump out all the non tail-biting paths and just have tail-biting ones left in the very end.

3.6 CRC Module

The CRC computation for each path can be represented as a bunch of parallel XORs as shown by Campobello[CPR03]. This module takes in an AXI path stream, and for each valid path it simply calculates the crc with this XOR approach. The output AXI stream is just a copy of the input AXI stream with tvalid set to be the output of the parallel CRC.

3.7 Dataflow

For the actual coordination of the design, the high level module instantiated every state memory and edge to take care of scheduling what time point the decoder was in, including which LLRs would go to which edges, and at what times. It also took care of routing paths from one state to the next, and waiting until all states are ready before stepping on to the next time point. The routing, including which states are which and who they point to, was all handled by CSV files defining the trellis and its connections.

This makes the trellis and code it supports super easy to modify later, just change the number of states and where they connect to and you can just re-synthesize the whole decoder. That said, the one thing that is hard to change (for now) is the rate of the code. Everything right now is written assuming a rate 1/5th code, and changing that would mess up timings but could eventually be solved with some work (and made configurable like the rest). This would be ideal for simulating many of the other rate TBCRC codes CSL is working on, and a definite next step for this decoder. Solutions for this are mentioned in the next chapter.

3.8 CPU Interface

Until the encoder is also hardware accelerated, there needs to be a way to exercise the decoder and prove its performance without relying on a fully working system. To do this, the decoder utilized the Zynq architecture of the FPGA to encode codewords in software on the CPU, then send them over to the Programmable Logic (PL), through a BRAM. The BRAM defined a section of addressable memory with different addresses corresponding to different debug metrics and whatever LLRs were needed for the decoder. Both the hardware and software had to agree on these addresses, and then they could communicate between the two domains by writing and reading these sections of the Block RAM. In this case instead of using BRAM, the decoder used a URAM so that the BRAM could be fully dedicated to the decoder, but this has no difference in functionality for the decoder.

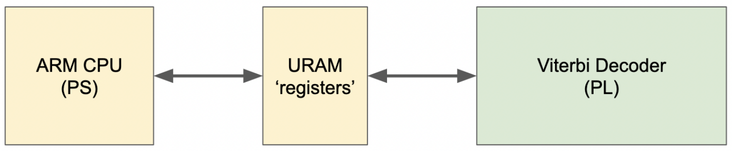


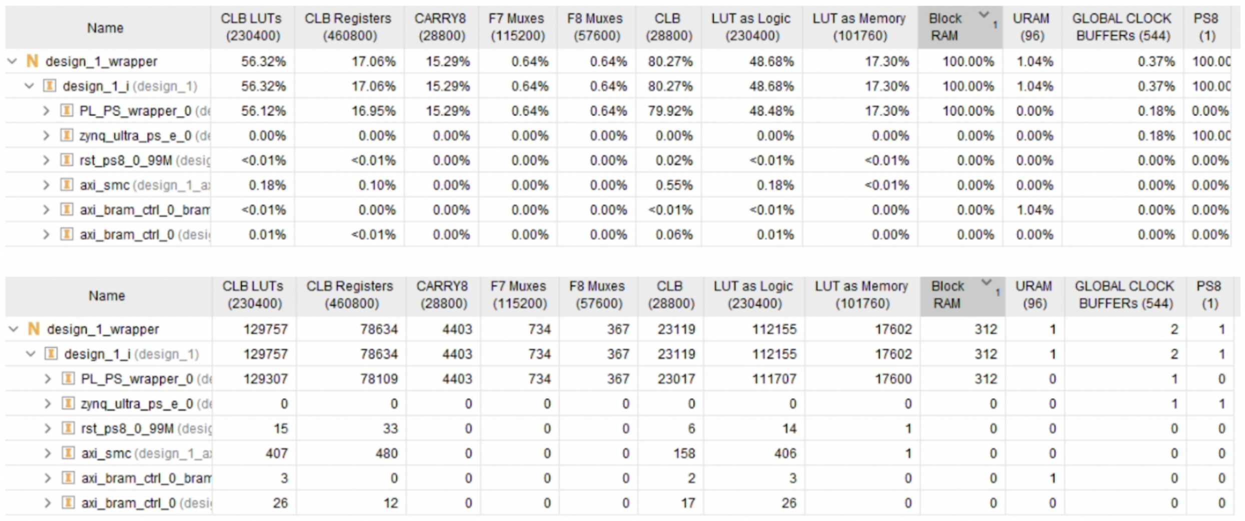
Figure 3.7: High level block diagram, where both the PS and PL communicate through different ports of a 2-port URAM memory element.

This did have the downside of being relatively slow. The decoder could decode in fractions of a second, but the software was having to poll this URAM to wait until the hardware was done, and then upload the new generated codeword, one LLR address at a time. Ideally, once an encoder is implemented, the software would simply send a ”start” flag, or a desired SNR, and then just read the resulting frame error rate metrics out of specific addresses, with very little communication or waiting on memory accesses.

3.9 Results

3.9.1 Utilization

The general implementation strategy for the decoder committed to trying to use every state at once in parallel rather than doing them in series. There weren't enough BRAMs to support this but Distributed RAMs could fill in the gaps. Distributed RAM is basically just FPGA LUTs being used as memory, and the hope was that this would be small enough that it could work well. However, when synthesized, this design used many more LUTs than expected.



Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	F8 Muxes (57600)	CLB (28800)	LUT as Logic (230400)	LUT as Memory (101760)	Block RAM	URAM (96)	GLOBAL CLOCK BUFFERs (544)	PS8 (1)
design_1_wrapper	56.32%	17.06%	15.29%	0.64%	0.64%	80.27%	48.68%	17.30%	100.00%	1.04%	0.37%	100.0C
design_1_i (design_1)	56.32%	17.06%	15.29%	0.64%	0.64%	80.27%	48.68%	17.30%	100.00%	1.04%	0.37%	100.0C
PL_PS_wrapper_0 (de	56.12%	16.95%	15.29%	0.64%	0.64%	79.92%	48.48%	17.30%	100.00%	0.00%	0.18%	0.00%
zynq_ultra_ps_e_0 (de	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.18%	100.0C
rst_ps8_0_99M (desig	<0.01%	<0.01%	0.00%	0.00%	0.00%	0.02%	<0.01%	<0.01%	0.00%	0.00%	0.00%	0.00%
axi_smc (design_1_a:	0.18%	0.10%	0.00%	0.00%	0.00%	0.55%	0.18%	<0.01%	0.00%	0.00%	0.00%	0.00%
axi_bram_ctrl_0_bram	<0.01%	0.00%	0.00%	0.00%	0.00%	<0.01%	<0.01%	0.00%	0.00%	1.04%	0.00%	0.00%
axi_bram_ctrl_0 (desi	0.01%	<0.01%	0.00%	0.00%	0.00%	0.06%	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%

Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	F8 Muxes (57600)	CLB (28800)	LUT as Logic (230400)	LUT as Memory (101760)	Block RAM	URAM (96)	GLOBAL CLOCK BUFFERs (544)	PS8 (1)
design_1_wrapper	129757	78634	4403	734	367	23119	112155	17602	312	1	2	1
design_1_i (design_1)	129757	78634	4403	734	367	23119	112155	17602	312	1	2	1
PL_PS_wrapper_0 (de	129307	78109	4403	734	367	23017	111707	17600	312	0	1	0
zynq_ultra_ps_e_0 (de	0	0	0	0	0	0	0	0	0	0	1	1
rst_ps8_0_99M (desig	15	33	0	0	0	6	14	1	0	0	0	0
axi_smc (design_1_a:	407	480	0	0	0	158	406	1	0	0	0	0
axi_bram_ctrl_0_bram	3	0	0	0	0	2	3	0	0	1	0	0
axi_bram_ctrl_0 (desi	26	12	0	0	0	17	26	0	0	0	0	0

Figure 3.8: Utilization table for the hardware design. Synthesized with Vivado for a ZCU106 board. The top figure shows percentages of hardware utilized while the bottom is raw numbers that are easier to compare to FPGAs with different amounts of hardware resources.

Small tests with Distributed RAM showed minimal impact on utilization, but when scaled up to 256 states it took up vast amounts of LUTs and over-complicated the routing between states. This made it a lot harder to meet timing, requiring a slower clock rate, and also resulted in a decoder that is far too big to use in practical systems. It would work but to have any other pieces of the radio attached the FPGA would need more standalone ASICs or FPGAs dedicated to them, they wouldn't fit on this board. Additionally, the Distributed RAM can't fit words as wide as BRAM. BRAMs can fit 72 bit words so it can't hurt to just use all of them for the path and its metric, but in distributed RAM the word size had to be trimmed significantly (taking bits out of the metric, which hurt accuracy if trimmed too far), and even then the distributed RAM wastes a lot of LUTs. Furthermore, list sizes are limited to 64 because of this Distributed RAM limitation, which means wasting 1000's of lines of depth in the BRAMs because the Distributed RAM can't be as deep.

3.9.2 Future Work

Overall, this Distributed RAM limitation is hindering the FPGA accelerator more than its helping. The trade-offs required to do an entire time point in parallel are not worth it. If the decoder could share BRAMs among states and do some of them sequentially there would be a number of benefits. The routing would be simpler which would make it easier to meet timing. This would allow for a higher clock frequency to offset some of the sequential runtime losses. Additionally, users of the accelerator could configure how many BRAMs they wanted to dedicate to their decoder, allowing for a configure-able trade-off between parallelism, maximum list size, and utilization.

In the next chapter, a new version of the decoder, fully designed but still being implemented, is presented to fix these issues.

CHAPTER 4

Future Accelerator Architecture

4.1 Motivation

Implementing the first version of the decoder has revealed many places for further optimizations towards a smaller, more versatile accelerator. The focus of this new decoder design is maximizing BRAM efficiency, but there are also places requirements can be more rigid, requiring less flexible hardware, and some places where it would be useful to make the hardware more flexible.

4.2 High Level Design

At its core, version 2 will use most of the same basic concepts. It can still be boiled down to parallel adders and merge sorting memory, with AXI streams and control logic connecting everything.

Where version 2 differs is how the memory is organized and how metrics will be computed to minimize stalls and hardware utilization, along with accommodating a flexible rate code.

4.3 Memory Layout

Memory on an FPGA is a limited resource that many IPs could use. Designing an IP to use all of the BRAM in an FPGA and then still require LUTs to synthesize more memory is incredibly impractical. There would be a single decoder without any room for a demodulator, SNR estimator to go from bits to LLRs, or really anything else. V2 aims to give the designer a choice for how many BRAMs to use for their decoder, and then aims to share these available BRAMs among states while maximizing throughput and FER performance.

For example, a designer could choose they wanted to use only 8 BRAMs, they are working on a very small FPGA with very limited resources and don't mind sacrificing throughput for a very small hardware footprint.

In a scheme like this, the address space of each BRAM would be split into 2, one section for "old" paths being read and added to, and one section for "new" paths, the output from the old edges going to the next time point.

Additionally, the address space will need to be split up in terms of edges and their corresponding paths. Of 256 total edges, the goal would be to split them all up evenly among the available BRAMs in such a way that there is always 1 write and 1 read to every BRAM happening in parallel, with no overlaps.

To start addressing this segmentation, the first step will be defining a trellis. This decoder used the same definition that Matlab's poly2trellis function uses so that its easier for designers to generate new trellises. For a trellis of size 8, see Figure 4.1.

From here, define the following parameters:

$$B = num\_total\_brams = 64$$

$$S = num\_total\_states = 256$$

$$E = num\_total\_edges = 2 * S$$

It is intuitive to now look at the trellis in terms of input and output edges.

Outgoing edges for state $S[s]$:

$$E[2 * s]$$

$$E[2 * s + 1]$$

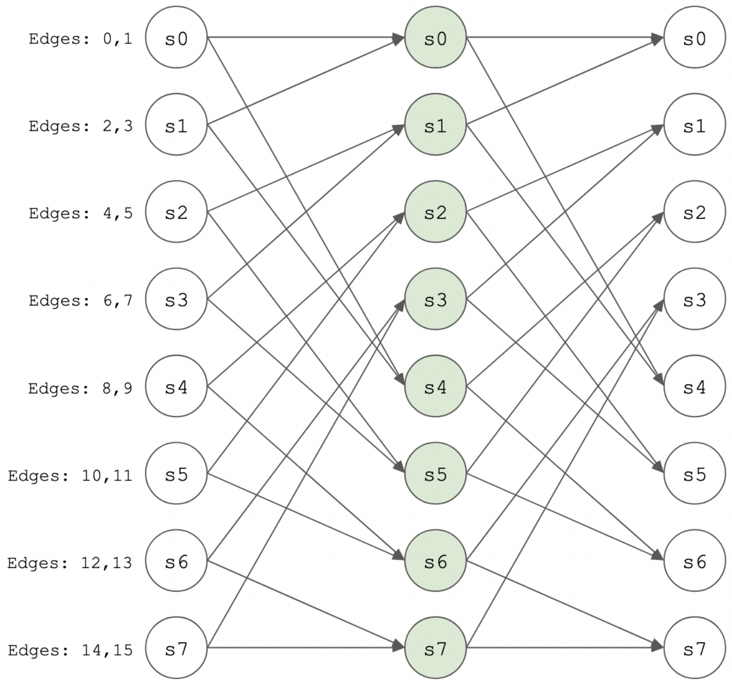


Figure 4.1: A small section of a trellis showing the states and their corresponding input and output edge numbering. Note that because this is a trellis, these edges will always connect the same states. An edge a going from state b to c will always go that direction between those two states regardless of the time-point.

Incoming edges for state $S[s]$:

$$E[((S/2) * s + 1)\%E]$$

$$E[((S/2) * s + 1)\%E + 2]$$

These equations look inefficient, but in hardware would be accomplished with a simple right shift and bit flip.

What really matters for the data flow is which input edges map to which output edges, the state that they are actually connecting to is just a number that is only used to check tail-biting conditions at the end.

With that in mind, the decoder needs to map every edge to a section of BRAM memory such that the same BRAM is never written to or read from twice in parallel. Each edge will get a section of L addresses to write new paths to and a section of L addresses to read old paths from, out of a certain BRAM. For example, the following figure color codes such an example for an 8 state, 4 BRAM approach, where different colors indicate different BRAMs:

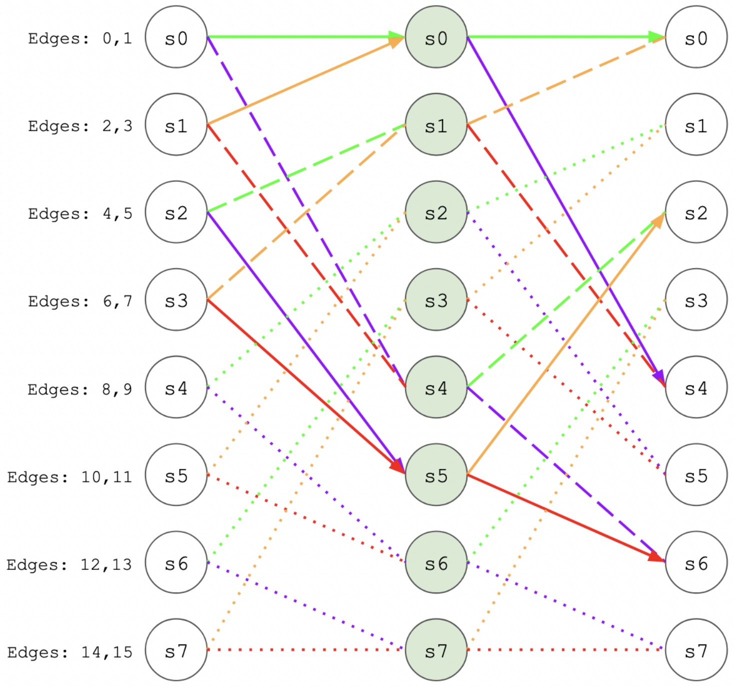


Figure 4.2: An example of how the edges can be divided in an 8 state trellis with 4 BRAMs. Each color represents a BRAM, while the solid lines represent reads and writes for cycle 1, thick dashes are cycle 2, and thin dots are everything else.

In this figure, the solid lines represent edges read and written to in the first step of the decoding process, the thicker dashed lines could then be a group for the second iteration, and then all the thinly dashed lines would be divided up for the remainder until they are all completed. There would be 4 iterations total because the decoder is doing 2 states per iteration and there are 8 states.

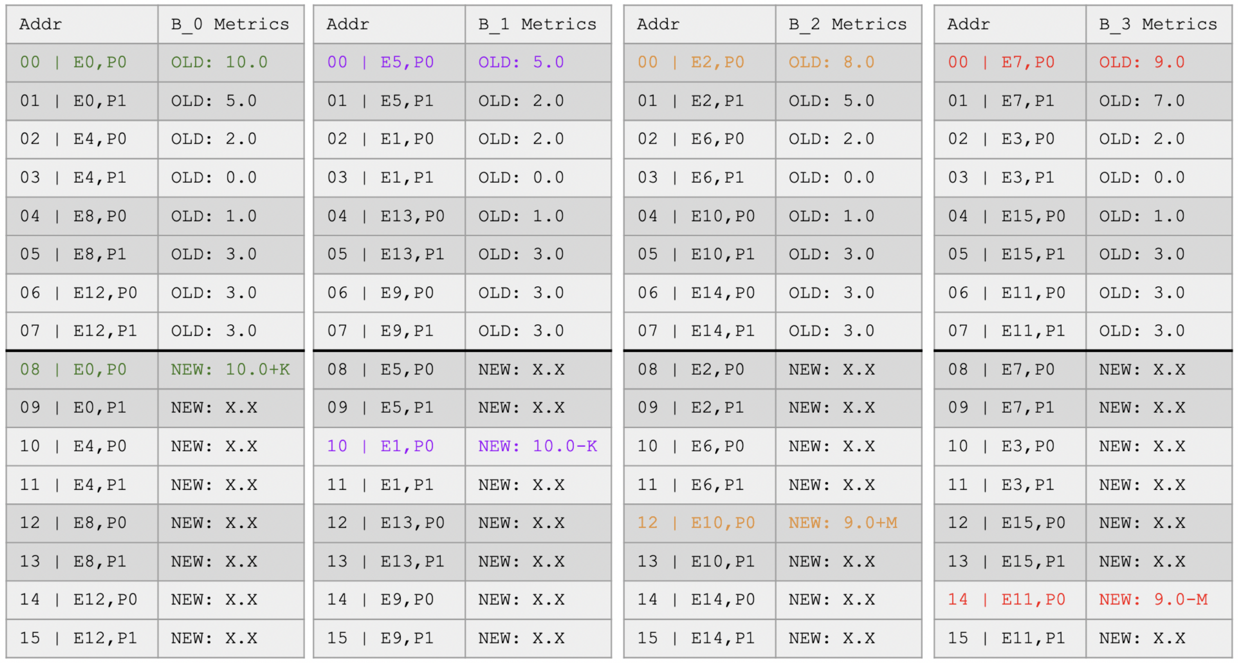
More generally, for an S state decoder, B BRAMs would be mapped to contain the following edges:

$$P = num\_edges\_per\_bram = E/B$$

$$B[k] \rightarrow E[P * n + k], k \in B, n \in E/P$$

Verbally, BRAM k contains edges that are integer multiples of P incremented by k.

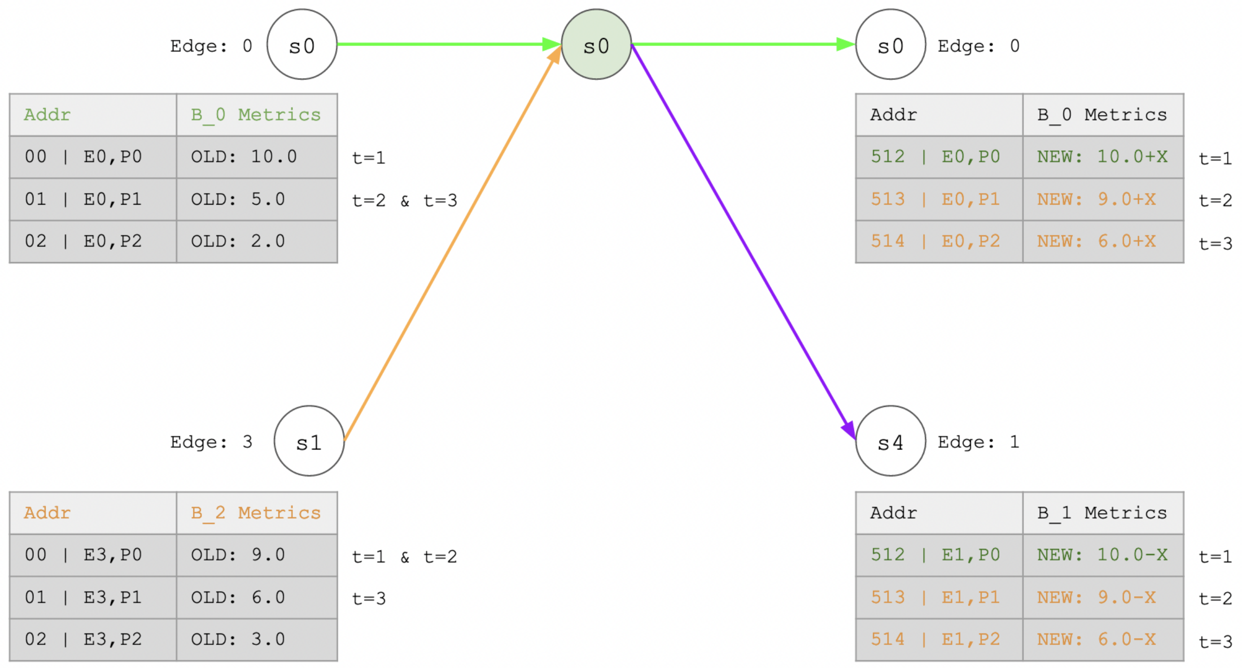
Splitting up the addresses of every BRAM, this would look something like 4.3:



Addr	B_0 Metrics	Addr	B_1 Metrics	Addr	B_2 Metrics	Addr	B_3 Metrics
00 | E0,P0	OLD: 10.0	00 | E5,P0	OLD: 5.0	00 | E2,P0	OLD: 8.0	00 | E7,P0	OLD: 9.0
01 | E0,P1	OLD: 5.0	01 | E5,P1	OLD: 2.0	01 | E2,P1	OLD: 5.0	01 | E7,P1	OLD: 7.0
02 | E4,P0	OLD: 2.0	02 | E1,P0	OLD: 2.0	02 | E6,P0	OLD: 2.0	02 | E3,P0	OLD: 2.0
03 | E4,P1	OLD: 0.0	03 | E1,P1	OLD: 0.0	03 | E6,P1	OLD: 0.0	03 | E3,P1	OLD: 0.0
04 | E8,P0	OLD: 1.0	04 | E13,P0	OLD: 1.0	04 | E10,P0	OLD: 1.0	04 | E15,P0	OLD: 1.0
05 | E8,P1	OLD: 3.0	05 | E13,P1	OLD: 3.0	05 | E10,P1	OLD: 3.0	05 | E15,P1	OLD: 3.0
06 | E12,P0	OLD: 3.0	06 | E9,P0	OLD: 3.0	06 | E14,P0	OLD: 3.0	06 | E11,P0	OLD: 3.0
07 | E12,P1	OLD: 3.0	07 | E9,P1	OLD: 3.0	07 | E14,P1	OLD: 3.0	07 | E11,P1	OLD: 3.0
08 | E0,P0	NEW: 10.0+K	08 | E5,P0	NEW: X.X	08 | E2,P0	NEW: X.X	08 | E7,P0	NEW: X.X
09 | E0,P1	NEW: X.X	09 | E5,P1	NEW: X.X	09 | E2,P1	NEW: X.X	09 | E7,P1	NEW: X.X
10 | E4,P0	NEW: X.X	10 | E1,P0	NEW: 10.0-K	10 | E6,P0	NEW: X.X	10 | E3,P0	NEW: X.X
11 | E4,P1	NEW: X.X	11 | E1,P1	NEW: X.X	11 | E6,P1	NEW: X.X	11 | E3,P1	NEW: X.X
12 | E8,P0	NEW: X.X	12 | E13,P0	NEW: X.X	12 | E10,P0	NEW: 9.0+M	12 | E15,P0	NEW: X.X
13 | E8,P1	NEW: X.X	13 | E13,P1	NEW: X.X	13 | E10,P1	NEW: X.X	13 | E15,P1	NEW: X.X
14 | E12,P0	NEW: X.X	14 | E9,P0	NEW: X.X	14 | E14,P0	NEW: X.X	14 | E11,P0	NEW: 9.0-M
15 | E12,P1	NEW: X.X	15 | E9,P1	NEW: X.X	15 | E14,P1	NEW: X.X	15 | E11,P1	NEW: X.X

Figure 4.3: Example memory layout of merge sort happening in real time. The colors are coordinated to be the same as the trellis above. Each BRAM does 1 read and 1 write simultaneously every cycle, and adds or subtracts the branch metric depending on which edge it is on.

In terms of merge sorting path metrics, there are two sets of stored paths for each state, they are read in series until L is reached, writing L outputs to the same BRAMs that are being read from (in a different location). After all states are dealt with, the new and old locations are swapped. The figure below illustrates this merge sort with example numbers and a list size of 3.



Addr	B_0 Metrics	
00 | E0,P0	OLD: 10.0	t=1
01 | E0,P1	OLD: 5.0	t=2 & t=3
02 | E0,P2	OLD: 2.0	

Addr	B_2 Metrics	
00 | E3,P0	OLD: 9.0	t=1 & t=2
01 | E3,P1	OLD: 6.0	t=3
02 | E3,P2	OLD: 3.0	

Addr	B_0 Metrics	
512 | E0,P0	NEW: 10.0+X	t=1
513 | E0,P1	NEW: 9.0+X	t=2
514 | E0,P2	NEW: 6.0+X	t=3

Addr	B_1 Metrics	
512 | E1,P0	NEW: 10.0-X	t=1
513 | E1,P1	NEW: 9.0-X	t=2
514 | E1,P2	NEW: 6.0-X	t=3

Figure 4.4: This figure shows the memories in terms of input and output paths for a specific subsection of a trellis. Edge 0 has the best input metric so it is read at cycle 1, then edge 3 with metric 9 at cycle 2, and then finally edge 3 again for cycle 3.

Note that for outgoing edges there is a constant $\pm X$ being added to the original path metric. This is due to the metric calculation simplification made earlier. The goal is to maximize RC, where C is determined by the edge weights of each edge. For two outgoing edges from the same state, they have exactly opposite C components.

Segmenting BRAM like this will mean sequentially processing some states, resulting in a performance penalty, but it also has a few performance benefits. There will be very rigid and uniform structure making routing much simpler. Additionally, to switch time points it is very simple to calculate which address new paths will be located. In the FIFO approach the accelerator had to stall and wait to drain the FIFO until the next time point is present.

In the second version of the decoder, the addresses are intentionally mapped so reading will be as simple as incrementing a counter by 1. On the writing side of the BRAMs, the edge computation will have to know which edge it is looking at, and knowing which edge it is on it will be able to determine which address to place the resulting paths into. Because the trellis structure is the same for each time point, it can be obtained from a lookup table of which edge the path came from, or a calculation involving the mappings stated above.

This setup guarantees that this pattern of memory-mapping will work for a trellis with this regularity in structure. These numbers can be used to calculate the runtime and maximum list sizes as a function of the number of BRAMs selected.

$$BRAM\_depth = 1024$$

$$max_L = 1024/(2 * P) = 64$$

$$latency = L * K * num_timepoints = L * S * 43/(2 * B)$$

The decoder can be instantiated with more BRAMs to exploit greater levels of parallelism, but still set to run with an L smaller than max_L to finish faster in higher SNR regions, or for running WAVA with a list size of 1.

Many of these numbers were specified for a specific BRAM quantity, but can be repeated for any other B between 2 and 256, as long as it can be split evenly into the number of states. This is a place where the new design can really shine, V1 had a maximum list size of 64 while taking up every single BRAM, while V2 can support the same thing with just 64 of them.

4.4 Metric Computation

This new approach to memory also leads to requiring a new, more complicated design for edge modules and their corresponding metric computations. At its core, it is still just a merge sort with an addition as shown before, but the pipeline to have these metrics ready for their corresponding BRAMs now needs to be modified to share hardware resources between states.

The first design was very decentralized, with each edge computing its own metric in parallel, but this version aims to centralize the metric computations in order to further reduce the hardware footprint. Additionally, a redesign allows for a new requirement to make the design more flexible: an adjustable rate decoder. Changing the rate changes the number of additions per time point but allows this decoder to support many more different kinds of codes just by changing a parameter for synthesis.

From a high level, the edge_module will take in L input paths (depending on how many BRAMs the designer specified there is a hardware limit for max L), a counter for which time point and state it is currently on, and input LLRs. It should properly output $2 * L$ paths and route them to their correct BRAMs. Any required metric increments should be properly pipelined such that they come out with a throughput of one result per cycle (to support WAVA), and also always be ready without any stalls to the data flow to wait for a metric computation. Ideally, this would be done without any rigid timing logic, and the edge computations could just use AXI to fill a buffer whenever it is empty and asserting tready.

Pictorially, this new edge computation module will look like Figure 4.5.

The comparison blocks are whole lists of paths being compared sequentially, as shown in Figure 4.4. Additionally, the $\pm$ blocks are adding and subtracting the R*C branch metric for each outgoing state, as discussed in earlier sections.

The intermediate adders and buffers holding intermediate results for each time point will act similarly to the path FIFOs of the v1 design, where they will compute up to 1 new intermediate metric per cycle, and store the 8 next required intermediate results. The number 8 was picked to fully hide the latency of the minimum rate code supported, which is a rate 1/32 code, and requires a latency of 5 cycles (rounded up to the nearest power of 2). Actual routing for output paths will be a multiplexer, depending on which input edges are currently being used.

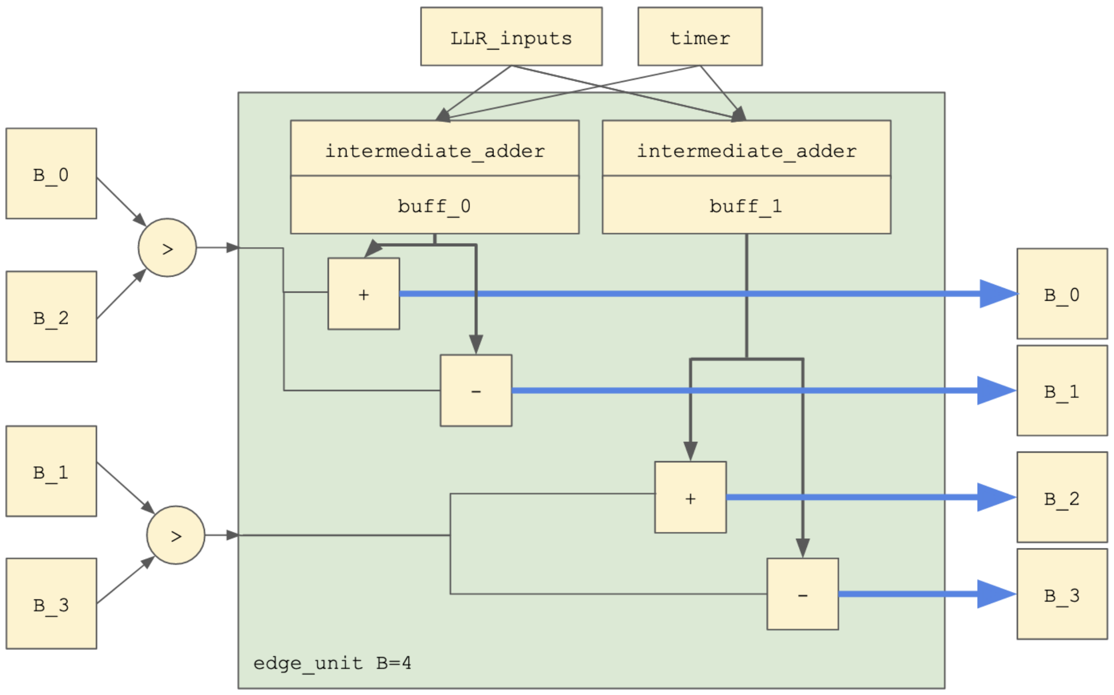


Figure 4.5: Edge unit block diagram.

4.5 Control Flow

Control flow of this new decoder is very simple. For each time point, increment a counter until running out of valid paths (slowly branching out the initial trellis), or hitting the current max_L. At this point, switch to the next state in the BRAM by navigating to that address, and repeat until running out of all possible states. After reaching all possible states, the current time point is completed, simply switch the address pointer to the ”new” section, and start iterating through paths again until the end of the decoded message, where the CRC modules from V1 can be used to check the validity of any tail-biting messages.

Outputs from the LVA module specifically will include the best metric from every state, along with a stream of paths, while inputs will include initial metrics, what list size to run at, and LLRs to use.

CHAPTER 5

Conclusion

This thesis introduces a variety of techniques to reduce the computational complexity of CRC-aided decoding of tail-biting convolutional codes. Some of these techniques, such as merge sorting and reduced-complexity metric computation, provide complexity reduction with no loss in performance. Other techniques, such as the wrap-around Viterbi algorithm require careful analysis to explore the trade-off between complexity reduction and performance degradation.

This thesis analyzes and simulates these techniques and ultimately implements a functioning parallel list Viterbi decoder using Vivado and a Xilinx ZCU106 board. This initial decoder confirms that CRC-aided list Viterbi decoding of tail-biting convolutional codes is achievable in a practical system. This first implementation is only a beginning. The thesis identifies many opportunities for potential improvements.

To lay a foundation for that future work, Chapter 4 proposes a new design that improves a variety of aspects of the pioneering original implementation, resulting in a decoder design that should be of practical interest. While work towards this new implementation is ongoing, this thesis is intended to be a resource for anyone looking to understand practical aspects of realizing the vision of CRC-aided list decoding of tail-biting convolutional codes.

REFERENCES

[CPR03] Giuseppe Campobello, Giuseppe Patane, and Marco Russo. “Parallel CRC realization.” *IEEE Transactions on Computers*, **52**(10):1312–1319, 2003.

[CS94] Richard V Cox and Carl-Erik W Sundberg. “An efficient adaptive circular Viterbi algorithm for decoding generalized tailbiting convolutional codes.” *IEEE transactions on vehicular technology*, **43**(1):57–68, 1994.

[For73] G David Forney. “The viterbi algorithm.” *Proceedings of the IEEE*, **61**(3):268–278, 1973.

[JZ15] Rolf Johannesson and Kamil Sh Zigangirov. *Fundamentals of convolutional coding.* John Wiley & Sons, 2015.

[KKY22] Jacob King, Alexandra Kwon, Hengjie Yang, William Ryan, and Richard D. Wesel. “CRC-Aided List Decoding of Convolutional and Polar Codes for Short Messages in 5G.”, 2022.

[LY14] Dongdong Li and Jun Yang. “Efficient implementation of the decoder for tail biting convolutional codes.” In *2014 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, pp. 623–626. IEEE, 2014.

[MW86] Howard Ma and Jack Wolf. “On tail biting convolutional codes.” *IEEE Transactions on Communications*, **34**(2):104–111, 1986.

[Sha48] C. E. Shannon. “A mathematical theory of communication.” *The Bell System Technical Journal*, **27**(3):379–423, 1948.

[SLF03] R.Y. Shao, Shu Lin, and M.P.C. Fossorier. “Two decoding algorithms for tailbiting codes.” *IEEE Transactions on Communications*, **51**(10):1658–1665, 2003.

[SP09] Sudharshan Srinivasan and Steven S Pietrobon. “Decoding of high rate convolutional codes using the dual trellis.” *IEEE transactions on information theory*, **56**(1):273–295, 2009.

[SS94] Nambirajan Seshadri and C-EW Sundberg. “List Viterbi decoding algorithms with applications.” *IEEE transactions on communications*, **42**(234):313–323, 1994.

[Vit67] Andrew Viterbi. “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm.” *IEEE transactions on Information Theory*, **13**(2):260–269, 1967.

[YLP22] Hengjie Yang, Ethan Liang, Minghao Pan, and Richard D. Wesel. “CRC-Aided List Decoding of Convolutional Codes in the Short Blocklength Regime.” *IEEE Transactions on Information Theory*, **68**(6):3744–3766, 2022.