The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

AMD Xilinx AI Engine-based High Speed Viterbi Decoding

AUTHOR	**Teo Sei Hau**
SUPERVISOR	**Dr. Yiqun Zhu**
MODERATOR	**Dr. Tianjie Zou**
DATE	**September 2024**

Project thesis submitted in part fulfilment of the requirements for the degree of Master of Science Electronic Communications and Computer Engineering, The University of Nottingham.

Table of Contents

Abstract

This project explores the implementation of the Viterbi Decoding algorithm on AMD's AI-Engine using the VEK280 evaluation kit, with the goal of significantly improving performance through parallel processing. The Viterbi Algorithm, a dynamic programming technique established in 1967, is essential for decoding convolutional codes in digital communication systems. This research aims to address the computational challenges of the Viterbi algorithm, particularly as data volumes increase and real-time processing becomes crucial. By transitioning the algorithm from C++ to AI-Engine code, the project leverages the AI-Engine's parallel processing capabilities to optimize execution speed and efficiency.

The project's primary objective is to compare the performance of the Viterbi decoding algorithm on traditional CPU architectures, specifically the Intel i9-12900H, versus the AI-Engine on board VEK280. The performance evaluation focuses on metrics such as transfer bandwidth and Bit Error Rate (BER). The implementation on the AI-Engine achieved an average speedup of 27.65 times compared to the CPU implementation, demonstrating a substantial reduction in execution time while maintaining high accuracy in decoding. This significant performance improvement underscores the advantages of utilizing specialized hardware for complex algorithms like Viterbi decoding.

In conclusion, this project successfully demonstrates the potential of parallel computing in enhancing the performance of the Viterbi decoding algorithm. The implementation on the AI-Engine not only achieves faster execution but also maintains high accuracy in decoding. This research contributes valuable insights into the benefits of using specialized hardware for algorithm optimization, paving the way for future innovations in digital communications and error correction technologies using AI Engines.

Keywords: AI Engine, Viterbi algorithm.

CHAPTER 1: Introduction

1.1 Background Information

1.1.1 Viterbi Algorithm (VA)

Proposed in 1967, the Viterbi Algorithm (VA) is a dynamic programming technique used to find the most likely sequence of transmitted bits based on the received signal. It is commonly employed for decoding bitstreams that have been encoded using convolutional codes or trellis codes. The Viterbi algorithm aims to perform maximum likelihood decoding. In other words, it seeks the most probable sequence of transmitted bits given the received signal. By considering all possible paths through the code, it calculates the likelihood of each path and selects the most likely one.

While the Viterbi algorithm is computationally intensive, it provides accurate decoding results. It is most often used for decoding convolutional codes with constraint lengths k = 3, although practical implementations can handle values up to k = 15. [1]

The architecture of the Viterbi algorithm, which uses the trellis diagram to determine state transitions, is particularly well-suited for parallel processing. This is because it allows computational tasks to be divided into smaller, independent tasks that can be executed simultaneously. By leveraging this feature, we can significantly enhance the algorithm's execution speed and efficiency.

In this project, we aim to harness the power of parallel processing to accelerate the Viterbi algorithm. This will be achieved by implementing the algorithm on the AI-Engine model VEK280 board from AMD, a state-of-the-art hardware platform known for its robust parallel programming capabilities. This endeavour will not only demonstrate the practical application of the Viterbi Decoding algorithm but also highlight the potential of parallel computing in enhancing the performance of the algorithm. This exploration into the intersection of convolutional algorithms and parallel computing promises to yield significant insights and advancements in the field.

1.1.2 Convolutional Codes

Convolutional codes are a type of error-correcting code used in digital communication systems. Unlike classic block codes, which operate on fixed-length blocks of data, convolutional codes work with continuous streams of data. The key concept behind these codes lies in their ability to generate parity symbols by applying a sliding Boolean polynomial function to the input data stream. This sliding operation is akin to a mathematical “convolution,” which gives rise to the term “convolutional coding.”

The encoder in a convolutional code employs linear shift registers (LSRs) to transform k input bits into n output bits, resulting in a code rate of R = k/n. Each output bit depends not only on the current input but also on the last K input bits, where K represents the constraint length. The remarkable feature of convolutional codes is their suitability for maximum-likelihood soft-decision decoding using time-invariant trellises. This efficient decoding process sets them apart from classic block codes, which typically rely on hard-decision decoding. Convolutional codes find widespread use in digital communication due to their flexibility in terms of block length, code rate modification, and economical soft decision decoding

1.2 Motivation

The Viterbi Decoding algorithm plays a crucial role in various fields, from communication systems to speech recognition and bioinformatics. However, as data volumes grow and real-time requirements become more stringent, the need for efficient and high-performance implementations of this algorithm intensifies. Leveraging parallel programming on specialized hardware like the AMD AI-Engine model VEK280 offers an exciting opportunity to address these challenges. By harnessing the power of parallelism, we aim to accelerate Viterbi Decoding, unlocking faster and more accurate hidden state sequence predictions. Our project not only showcases the capabilities of the AI-Engine but also contributes to advancing complex algorithms such as Viterbi decoding algorithm in practical applications. As we delve into parallelism, we open a path to optimize performance, reduce latency, and pave the way for future innovations in information processing.

1.3 Problem Statement

The Viterbi decoding algorithm is renowned for its computationally intensive. The computational demands of Viterbi decoding algorithm escalate exponentially with the constraint length (K). In practice, Viterbi decoding is typically limited to $K \leqslant 9$ due to these computational complexities.

1.4 Project Scope and Direction

This project centres around migrating the Viterbi decoding algorithm from C++ to AI-Engine code, ensuring compatibility with the VEK280 evaluation kit. Here are the key aspects:

1. **Algorithm Migration:**
    - The existing Viterbi decoding algorithm, currently implemented in C++, will be adapted to the AI-Engine environment.

- Online tutorials (in C or Python) will serve as a reference for developing the C++ version of the algorithm.

2. **Performance Metrics:**
   - We will evaluate the effectiveness of the ported code using the following metrics:
     - ✓ Execution Time: How efficiently the algorithm runs on the AI-Engine.
     - ✓ Bit Error Rate (BER): Accuracy in decoding information bits.
     - ✓ Signal-to-Noise Ratio (SNR): Robustness against noise and interference.
3. **Parallelization Exploration:**
   - The Viterbi algorithm's structure will be closely examined to identify opportunities for parallelization.
   - Specifically, we'll focus on optimizing the convolutional part of the algorithm.
4. **AI-Engine Configuration:**
   - The AI-Engine development environment will be configured to match the VEK280 board.
   - Learning about VEK280 architecture, AI-Engine development flow, and using Vitis IDE will be essential during this stage.
5. **Comprehensive Documentation:**
   - Detailed documentation will accompany the project, covering:
     - ✓ Algorithm Description: Explaining the Viterbi decoding process.
     - ✓ Code Implementation: How the algorithm was adapted for the AI-Engine.
     - ✓ Configuration Steps: Setting up the AI-Engine environment.
     - ✓ Benchmarking Results: A comprehensive report on performance gains achieved.

In summary, this project bridges the gap between algorithmic complexity and hardware implementation, showcasing the potential of parallel computing in enhancing Viterbi decoding. The final deliverables will include both functional code and thorough documentation.

1.5 Aims and Objectives

1.5.1 Aims

The aims of this project are twofold. Firstly, it seeks to implement the Viterbi decoder using AI-Engine techniques on the VEK280 evaluation kit from AMD. This involves leveraging the advanced parallel processing capabilities of the AI Engine to optimize the Viterbi decoding algorithm, ensuring it runs efficiently on the VEK280 hardware. Secondly, the project aims to conduct a comprehensive performance comparison between the CPU and AI-Engine implementations of the Viterbi decoder. This comparison will evaluate metrics such as transfer bandwidth and Bit Error Rate (BER) to determine the advantages and potential trade-offs of using AI-Engine techniques over traditional CPU-based implementations.

1.5.2 Objectives

The primary objectives of this project include:

1. Analyse the Viterbi architecture for parallelization purpose.
2. Implement the Viterbi Decoding algorithm on the AI-Engine using a parallel programming approach.
3. Optimize the Viterbi Decoding algorithm for optimal speed and memory utilization.
4. Run hardware emulation on VEK280 board.

5. Simulate the algorithm using both the AI-Engine and a CPU for a comparative performance analysis.

1.6 Impact, Significance and Contribution

Upon completion, this project will deliver a comprehensive implementation of the Viterbi decoding algorithm on AMD’s AI-Engine. Additionally, it will include an evaluation of the implemented solution.

In terms of contributions, this project offers valuable insights into parallelizing the Viterbi decoder to enhance algorithm performance. By providing a robust architecture for parallel Viterbi decoding and comparing the results to CPU-based approaches, researchers can gain a better understanding of efficient execution. Moreover, the well-organized implementation facilitates the identification of convolutional code architectures suitable for specific purposes, such as integrating error correction mechanisms in data centre applications. Ultimately, this work accelerates future research endeavours, making them more efficient and impactful.

1.7 Organization of Report

This report contains 6 chapters with the details of the project listed in the following chapters.

- **Chapter 2: Literature Review**

  This chapter presents a comprehensive literature review, covering the Viterbi decoder architecture as well as some related research and projects. It also delves into the development flow of the AMD AI Engine and the architecture of the AMD Versal AI Edge VEK280 Evaluation Kit.

- **Chapter 3: Proposed Methods / Approaches**

  This chapter provides an in-depth discussion of the proposed methods and approaches. It begins with an overview of the theoretical foundations and principles underlying the proposed techniques. Following this, the method is examined in detail, including the rationale behind its selection, the specific steps involved in its implementation, the technologies used and any relevant modifications or optimizations. Additionally, this chapter explores the potential advantages and limitations of the approach. The discussion aims to provide a clear understanding of how the methods contribute to the overall objectives of the project.

- **Chapter 4: System Design**

  This chapter delves into the system design of the project, focusing on the true implementation of the system. The chapter provides a detailed description of the system architecture, including hardware and software components, their interactions, and the overall workflow. Each subsystem is examined in depth, highlighting the design choices, implementation steps, and any challenges encountered during development. Additionally, this chapter discusses the integration of various components. The goal is to offer a comprehensive understanding of how the system was designed and implemented to meet the project's objectives.

- **Chapter 5: Simulation and Emulation Results**

  This chapter presents the simulation and emulation results of the Viterbi implementation throughout the entire project. The chapter provides a comprehensive analysis of the simulation and emulation outcomes, comparing the performance metrics against the expected benchmarks. The goal of this chapter is to offer a thorough understanding of the Viterbi implementation’s performance, supported by empirical data and detailed analysis.

- **Chapter 6: Conclusions and Future Work**

  This chapter provides comprehensive conclusions of the project’s achievements, summarizing the key findings and outcomes. It reflects on the objectives set at the beginning and evaluates how effectively they have been met. Additionally, this chapter discusses the implications of the results and their significance in the broader context of the field. The latter part of the chapter is dedicated to future works, outlining potential areas for further research and development. It highlights the limitations encountered during the project and suggests ways to address them in future studies. The goal is to offer a clear roadmap for continuing the work and building upon the foundation established by this project.

Chapter 2: Literature Review

2.1 Overview of Viterbi Decoder Architecture

The Viterbi decoder processes convolutionally encoded input symbols that have been transmitted through a noisy channel. These input symbols are typically corrupted by noise during transmission. The decoder's primary task is to extract the original information from these noisy input symbols. The Viterbi Algorithm (VA) aims to find the sequence of symbols within a given trellis that is closest in distance to the noisy received sequence. This computed sequence represents the global most likely path. Figure 2.1.1 shows the example trellis diagram of a 1 input bit and 2 encoded bit (1/2), constraint length, K = 3 Viterbi decoder.

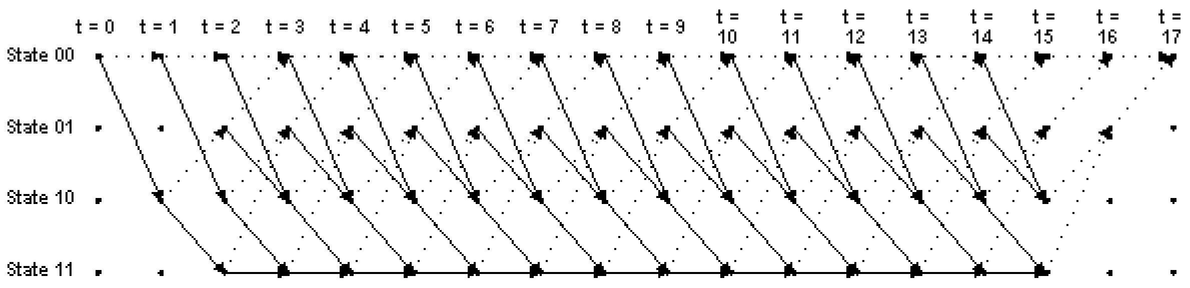


Figure 2.1.1: Example trellis diagram of a rate 1/2, K = 3 Viterbi decoder [2].

When using Euclidean distance as the distance measure, the VA becomes the optimal maximum-likelihood detection method in AWGN scenarios. However, in practical applications, the Hamming distance is often employed, even though it results in sub-optimal performance for the VA. Regardless of the chosen distance measure (Euclidean or Hamming), the procedure for searching the most likely sequence remains the same. For illustrative purposes, let's use Euclidean distance. To compute the global most likely sequence, the VA recursively computes the survivor path for each state. The survivor path for a given state consists of the symbols leading to that state, which are closest in distance to the noisy received symbols. The distance between the survivor path and the noisy symbol sequence is referred to as the path metric for that state. Once survivor paths for all states are computed, the VA selects the survivor path with the minimum path metric as the global most-likely path. [3], [4]

Consider a received sequence of noisy symbols denoted as y = (y_1, y_2, y_3, …, y_k). At each recursion, which corresponds to a stage in a trellis, the Viterbi Algorithm (VA) calculates the most likely transition entering each state. It then proceeds to update the survivor path and the path metric associated with that state. Figure 2.1.1 illustrates how the VA performs survivor path updates at state j.

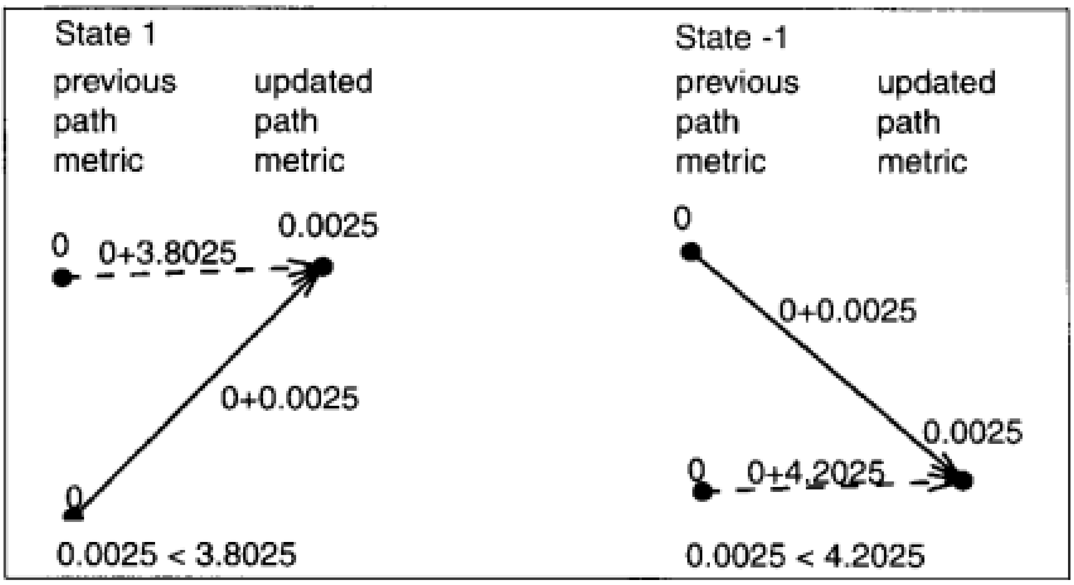


Figure 2.1.2: VA update survivor paths for state 1 and -1. [3]

In each recursion (denoted by n), the Viterbi Algorithm (VA) calculates the most likely transition entering state j. It achieves this by evaluating the metrics of all possible paths leading to state j. Subsequently, the VA selects the path with the minimum metric as the survivor path entering state j, and this chosen path's metric becomes the updated path metric for that state. When using Euclidean distance, the metric of a path represents the accumulated squared distances between the received sequence of noisy symbols and the ideal sequence of symbols along that path. Specifically, the metric for the path entering state j from the previous state i, at recursion n, is the sum of the branch metric for the transition from state i to state j and the path metric associated with state i computed at recursion (n - 1). This branch metric represents the squared distance between the received noisy symbol y_n and the ideal noiseless output symbol corresponding to that transition. It's important to note that this branch metric is calculated using Euclidean distance (Soft decision decoding technique). Specifically, for the transition from state i to state j at recursion n, the branch metric is:

$$B_{i,j,n} = \left(y_n - C_{i,j}\right)^2$$

where C_i_j represents the output symbol of the transition from state i to state j. Additionally, if we define M_j_n as the path metric for state j at recursion n, and {i} as the set of states that have transitions leading to state j, then the most likely path entering state j at recursion n corresponds to the path with the minimum metric,

$$M_{j,n} = \min_{\{i\}}\left(M_{i,n-1} + B_{i,j,n}\right)$$

Once the most likely transition to state j at recursion n has been computed, the path metric for state j, denoted as M_j_n, is updated. Additionally, the most likely transition — let's say from state i to state j—is appended to the survivor path of state i at recursion n - 1. This process forms the updated survivor path entering state j at the current recursion, n.

During each recursion, both the path metrics and the survivor paths are updated for all states. At the conclusion of the recursions (specifically, at recursion k in this case), the survivor path with the minimum path metric is chosen as the most likely path. This most likely path can be traced backward from the state that possesses the minimum path metric.

In practical scenarios, the received noisy symbols arrive continuously. Specifically, k + 1 in the context mentioned above. Based on the description provided earlier, one might assume that determining the most likely sequence would require an infinite amount of time. However, an interesting observation arises: if the survivor paths for all states converge to a single state—let's call it state j—and at recursion n, all states append the survivor path entering state j from recursion n to their existing survivor paths, then all states will share an identical survivor path sequence up to stage n. Consequently, we can trace back from any state and obtain the most likely sequence up to stage n. Given a specific trellis and the desired performance of the Viterbi decoder, it is possible to determine the number of recursions needed for the survivor paths to converge with high probability. This convergence point is referred to as the survivor path length, denoted as L. Consequently, after an initial delay of L recursions, we can reasonably assume that the survivor paths for all states converged L stages ago. As a result, we can start from any state and trace its survivor path back L stages to identify the most likely symbol at the (L - 1)-th stage. In practice, the value of L is determined through computer simulations. Normally L = 5 * K. [5]

2.1.1 Hard Decision Decoding

In the context of hard decision decoding, we work with a sequence of digitized parity bits. The branch metric, in this case, corresponds to the Hamming distance between the expected parity bits and the received ones. Below, Figure 2.1.1.1 illustrates a straightforward example of hard decision decoding, specifically focusing on computing the branch metric when the received bits are '00'.

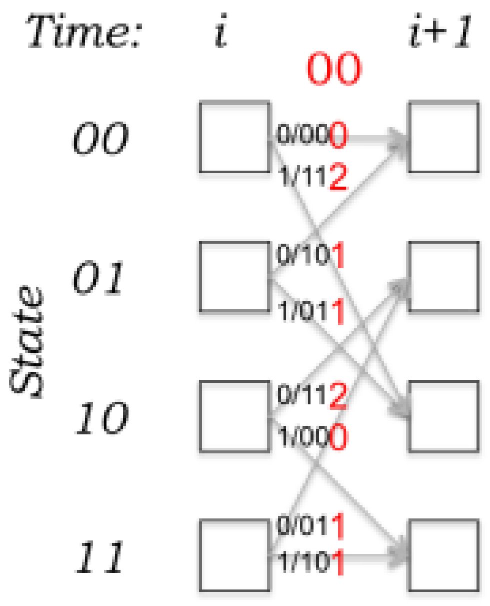


Figure 2.1.1.1: Branch metric computed using hard decision decoding. In this example, the decoder gets the parity bits 00. [6]

For each state transition, the number displayed on the arc represents the branch metric associated with that specific transition. Notably, two of these branch metrics are zero, aligning with the states and transitions where the corresponding Hamming distance is also zero. Conversely, the non-zero branch metrics correspond to scenarios involving bit errors.

The path metric is a value associated with a state in the trellis (i.e., with each node). In the context of hard decision decoding, it corresponds to the Hamming distance along the most likely path from the initial state to the current state within the trellis. By “most likely,” we refer to the path with the smallest Hamming distance between the initial and current states, considering all possible paths connecting the two states. This path, characterized by the smallest Hamming distance, effectively minimizes the total number of bit errors and is most likely when the Bit Error Rate (BER) is low.

The key insight of the Viterbi algorithm lies in the decoder's ability to incrementally compute the path metric for a (state, time) pair. This computation relies on the path metrics of previously computed states and the associated branch metrics. [6]

2.1.2 Soft Decision Decoding

Hard decision decoding involves digitizing the received voltage signals by comparing them to a threshold before passing them to the decoder. However, this process results in information loss. For instance, consider two scenarios: if the voltage was 0.50001, the confidence in the digitization is significantly lower than if the voltage was 0.99999. Despite this difference, both are seen as "1" by the decoder, even though it is overwhelmingly more likely that 0.99999 represents a "1" compared to the other value.

Soft decision decoding, also known as "soft input Viterbi decoding," capitalizes on this observation. Instead of digitizing the incoming samples prior to decoding, it employs a continuous function of the analogue sample as the input to the decoder. For example, if the expected parity bit is 0, and the received voltage is 0.3 V, we might use 0.3 (depending on the resolution of the decoder) as the value of the "bit" rather than discretizing it.

As mentioned in Chapter 2.1, soft decision decoding normally used Euclidean distance as the metric to determine best survival path. It is obvious that the decoding algorithm is identical to the one previously described for hard decision decoding, except that the branch metric is no longer an integer Hamming distance but a positive real number (if the voltages are all between 0 and 1, then the branch metric is between 0 and 1 as well).

In Chapter 2.1, we discussed soft decision decoding, which typically employs the Euclidean distance as the metric to determine the best survivor path. Notably, the decoding algorithm remains identical to the one previously described for hard decision decoding. However, there is a crucial difference: the branch metric is no longer an integer Hamming distance; instead, it becomes a positive real number. If the voltages lie within the range of 0 to 1, the branch metric also falls within that same interval. [6]

2.1.3 Comparison Between Both Decoding Method

Figure 2.1.3.1 below shows some representative performance results for a set of codes all of the same code rate of 1/2. The top-most curve shows the uncoded probability of bit error in terms of bit error rate (BER). The x-axis plots the amount of noise on the channel (lower noise is toward the right). The axis plots the signal-to-noise ratio (SNR) on a log scale. The unit of the x-axis is decibels, or dB, and N_0 is the noise, which for technical reasons is defined as $2\sigma^2$ (σ is the Gaussian noise variance— the factor of 2 is the technical definition).

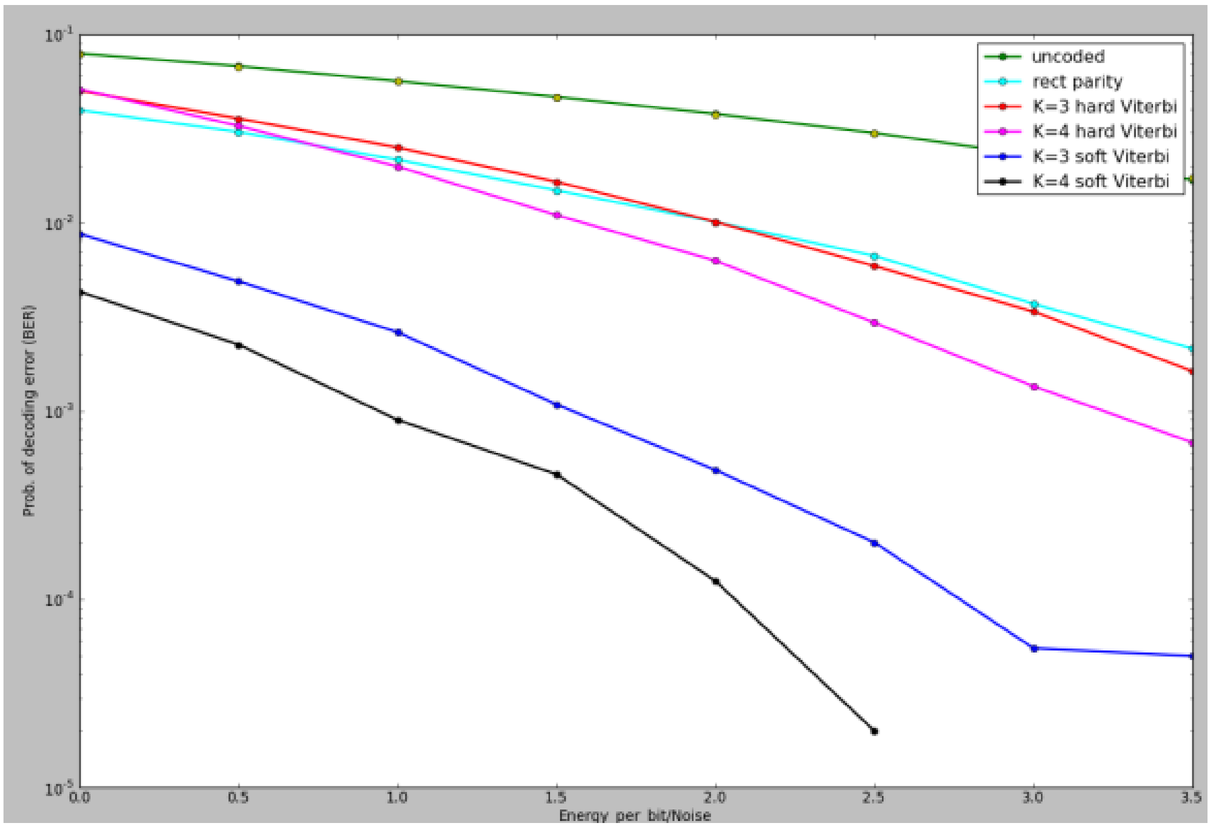


Figure 2.1.3.1: Error correcting performance evaluation for different codes under ½ rate decoder. [2]

From the graph in Figure 2.1.3.1 above, the key observations include that the uncoded line is consistently above the coded lines, indicating higher BER for uncoded transmission at equivalent Eb/No levels. Soft Viterbi decoding (both K = 3 and K = 4) generally outperforms hard Viterbi decoding, achieving lower BER for the same Eb/No. Additionally, the performance improves with increased constraint length, as seen with the K=4 soft decision decoding Viterbi performing better than the K=3 soft decision decoding Viterbi. Overall, the graph demonstrates that coding significantly improves BER performance, with soft Viterbi decoding providing the best results among the schemes compared.

2.2 Related Research and Projects

In this chapter, we will delve into the extensive body of related research and existing work on the implementation of the Viterbi decoding algorithm across various technologies. The key studies and breakthroughs that have contributed to advancements in implementing the Viterbi algorithm in different technological contexts will be reviewed in this chapter.

2.2.1 Efficient Parallel Implementation of Three-point Viterbi Decoding algorithm on CPU, GPU and FPGA [7]

The study conducted by Rongchun Li, Yong Dou, and Dan Zou provides a comprehensive analysis of the parallel implementation of the three-point Viterbi decoding algorithm (TVDA) across various computing platforms, including CPUs, GPUs, and FPGAs. This research highlights the potential for significant performance improvements through parallel processing and optimization techniques.

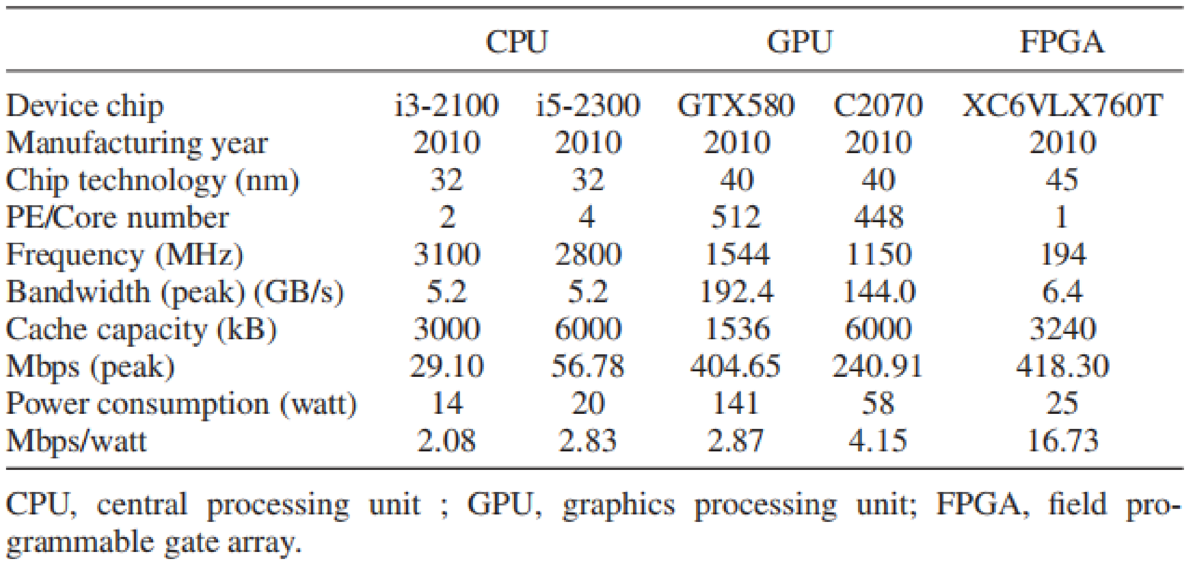
For CPU implementations, the authors leveraged techniques such as Single Instruction Multiple Data (SIMD) and multithreading to enhance the performance of the Viterbi decoder. These optimizations resulted in a speedup of over 145 times compared to a naive implementation on a quad-core CPU. This demonstrates the potential for substantial performance gains through parallel processing, which could be applied to enhance the efficiency of Viterbi decoders in AI engines like those from AMD Xilinx. The SIMD approach allows for multiple data elements to be processed simultaneously, which is particularly beneficial for the computationally intensive nature of Viterbi decoding.

In the GPU implementation, the authors utilized CUDA and implemented several optimizations including cached memory, coalesced global memory accesses, and asynchronous data transitions. These techniques improved the throughput to 404.65 Mbps on an NVIDIA GeForce GTX580, achieving a 7-fold improvement over an Intel quad-core CPU i5-2300. The use of GPUs for Viterbi decoding is advantageous due to their ability to handle numerous threads simultaneously, making them ideal for high-throughput applications. This aspect of the study could inform the integration of GPU-like parallel processing capabilities in AI engines, potentially leading to enhanced performance in real-time decoding tasks.

The FPGA implementation involved designing a custom radix-4 pipelined architecture, which was implemented on a 45nm Xilinx FPGA chip (XC6VLX760). This architecture achieved a throughput of 418.30 Mbps at a clock rate of 209.15 MHz. FPGAs offer the flexibility to tailor the hardware to specific applications, allowing for optimized performance and power efficiency. This flexibility and the ability to reconfigure the logic make FPGAs a compelling choice for implementing Viterbi decoders, particularly in AI engines where specific performance and power constraints must be met. [7]

The insights gained from this study can be applied to this project, as the AMD Xilinx AI Engine can benefit from these parallel processing techniques and optimizations. By leveraging the strengths of CPUs, GPUs, and FPGAs, this project can achieve a balance between performance, power efficiency, and flexibility, ultimately leading to a more robust and efficient Viterbi decoder implementation.

Table 2.2.1.1 and Figure 2.2.1.1 provides a detailed performance comparison of the Viterbi decoder with constraint length K = 7, among CPU, GPU, and FPGA implementations, as conducted by the research team. [7]



	CPU		GPU		FPGA
Device chip	i3-2100	i5-2300	GTX580	C2070	XC6VLX760T
Manufacturing year	2010	2010	2010	2010	2010
Chip technology (nm)	32	32	40	40	45
PE/Core number	2	4	512	448	1
Frequency (MHz)	3100	2800	1544	1150	194
Bandwidth (peak) (GB/s)	5.2	5.2	192.4	144.0	6.4
Cache capacity (kB)	3000	6000	1536	6000	3240
Mbps (peak)	29.10	56.78	404.65	240.91	418.30
Power consumption (watt)	14	20	141	58	25
Mbps/watt	2.08	2.83	2.87	4.15	16.73

CPU, central processing unit ; GPU, graphics processing unit; FPGA, field programmable gate array.

Table 2.1.1.1: Performance comparison between CPUs, GPUs and FPGA from different manufacturers. [7]

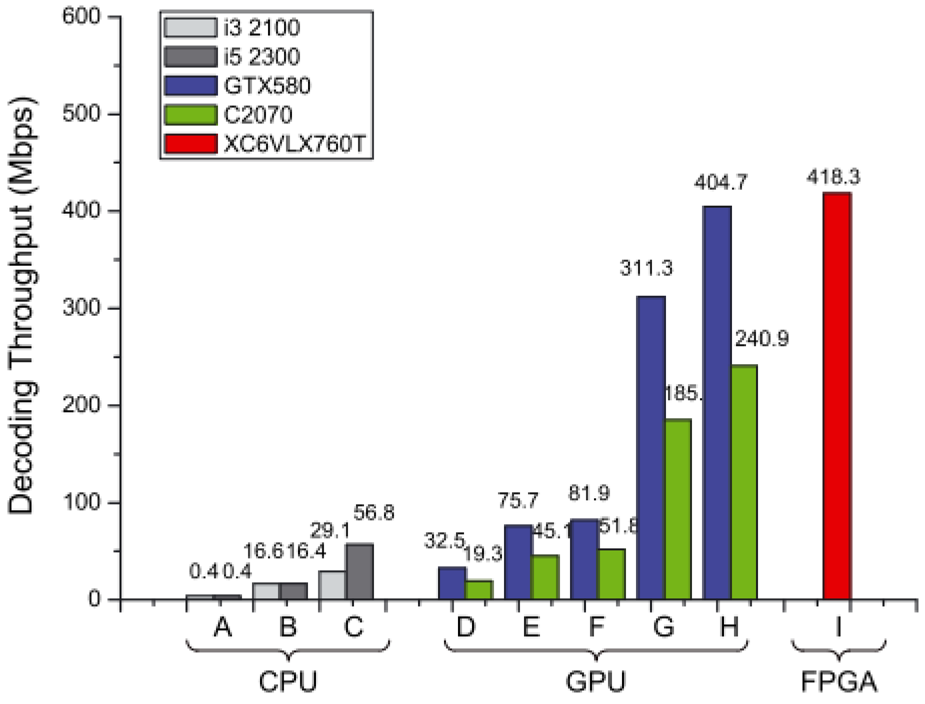


Figure 2.2.1.1: Performance growth of the Viterbi decoding algorithm with different optimization grades on three platforms. (A) no optimization, (B) SIMD, (C) multithread + SIMD, (D) no optimization, (E) cached memory optimization, (F) global memory optimization, (G) codeword packing scheme, (H) asynchronous data transition, (I) customized logic circuit. [7]

2.2.2 High-performance ACS for Viterbi decoder using pipeline T-Algorithm [8]

The paper by D. Vaithiyanathan, J. Nargis, and R. Seshasayanan presents significant advancements in the implementation of the Viterbi decoder algorithm, particularly focusing on improving performance and efficiency for systems with large constraint lengths.

The researchers proposed a modified T-Algorithm with a pre-computation architecture specifically designed for a rate 3/4 convolutional code. This approach is depicted in Figure 2.2.2.1 and Figure 2.2.2.2 below, which illustrate the two-step pre-computation T-algorithm architecture and the functional block diagram of the proposed system, respectively. By incorporating a pipelined architecture with three stages, the authors managed to improve the speed and efficiency of the decoder. The critical path of the algorithm was reduced by strategically repositioning delay elements and relocating the minimum value finder block, which precedes one set of delay elements. This reconfiguration not only enhances the speed but also reduces the computational complexity and power consumption, making the design more suitable for high-performance applications.

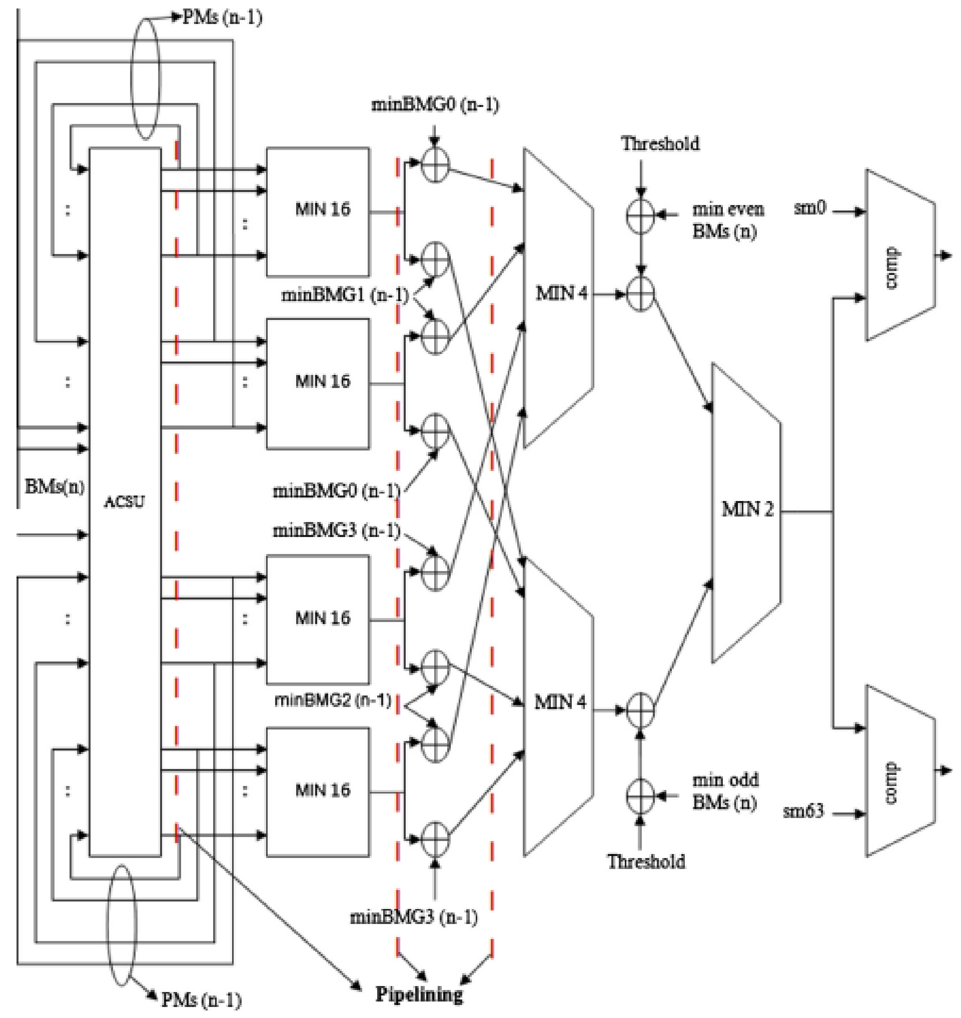


Figure 2.2.2.1: Two-step pre-computation T-algorithm architecture for rate 3/4 convolution codes. [8]

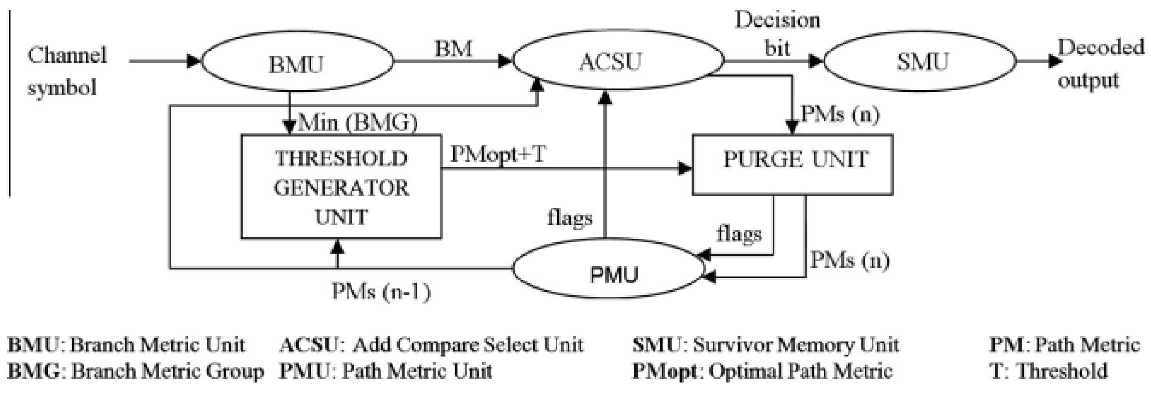


Figure 2.2.2.2: Proposed functional block diagram of VA decoder with added pre-computational architecture. [8]

Moreover, the researchers developed a reconfigurable Add-Compare-Select Unit (ACSU) architecture, enabling the decoder to support multiple constraint lengths, specifically from K=3 to 7. This flexibility is crucial for adapting to various wireless standards and optimizing the decoder's performance across different applications. The design was implemented on FPGA platforms, including Xilinx XCV800 and Xilinx XC2VP30, demonstrating significant improvements in resource utilization, delay, and power consumption compared to existing architectures, the results of the implementations are tabulated in table 2.2.2.1 below.

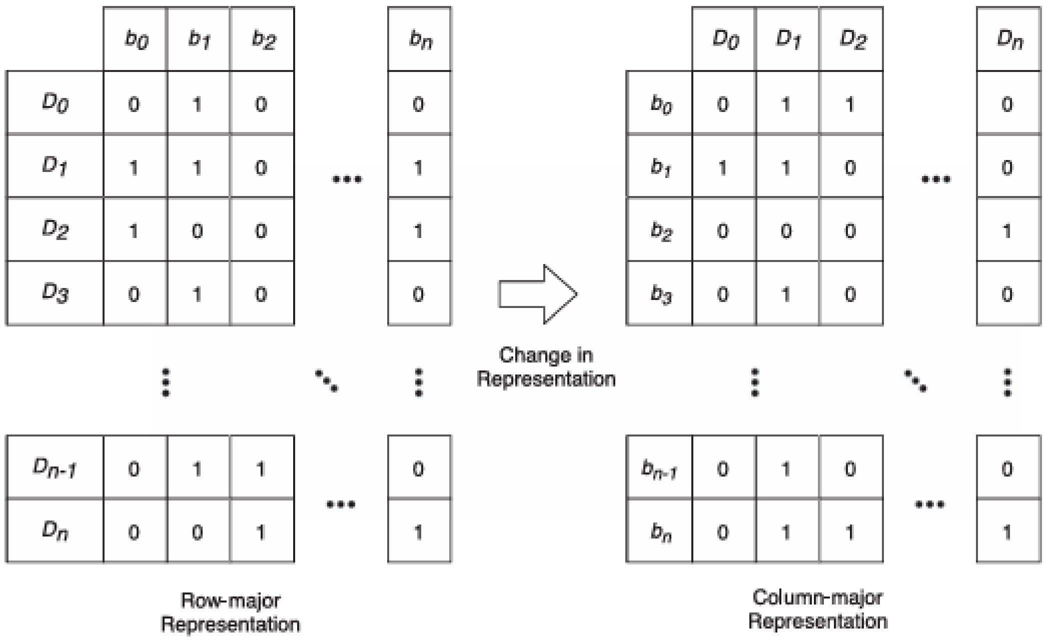
FPGA model	Area (number of gates)	Throughput (Mbps)
XCV800	57927	80.142
XC2VP30	66828	35.876

Table 2.2.2.1: Comparison of high-performance Viterbi decoder implemented on different FPGA model. [8]

These innovations align well with this project objectives, as the AMD Xilinx AI Engine can benefit from such efficient and flexible architectures. By adopting similar techniques, this project could achieve enhanced adaptability, crucial for applications requiring high-speed and constraint length flexible decoding solutions. The insights from this study can inform the development of more robust and efficient Viterbi decoders within AI engines, ultimately contributing to advancements in communication systems and other related fields.

2.2.3 A High-throughput Parallel Viterbi Algorithm via Bitslicing [9]

The paper titled "A High-throughput Parallel Viterbi Algorithm via Bitslicing" presents an innovative approach to implementing the Viterbi algorithm by utilizing bitslicing and GPU acceleration to greatly improve decoding throughput. The authors introduce a bitsliced data representation to facilitate efficient parallel processing on GPUs. This technique involves converting the input data format from row-major to column-major. Consequently, each processing unit can handle 32-bit data chunks simultaneously, instead of processing data bit by bit. Figures 2.2.3.1 and 2.2.3.2 illustrate the transformation from row-major to column-major format and the memory partitioning of the path metric using the proposed methods.



	b_0	b_1	b_2	...	b_n
D_0	0	1	0		0
D_1	1	1	0	...	1
D_2	1	0	0		1
D_3	0	1	0		0
⋮		⋮		⋱	⋮
D_{n-1}	0	1	1	...	0
D_n	0	0	1	...	1

	D_0	D_1	D_2	...	D_n
b_0	0	1	1		0
b_1	1	1	0	...	0
b_2	0	0	0		1
b_3	0	1	0		0
⋮		⋮		⋱	⋮
b_{n-1}	0	1	0	...	0
b_n	0	1	1	...	1

Figure 2.2.3.1: Transformation to column-major representation from row-major representation. [9]

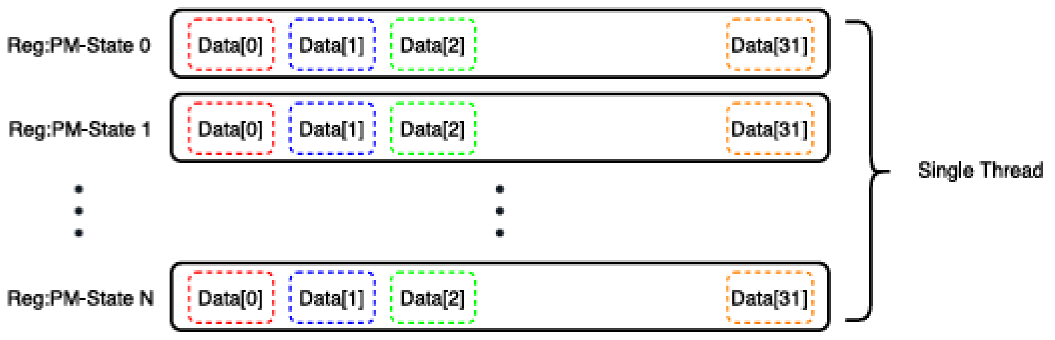


Figure 2.2.3.2: The memory partitioning representation of path metric (PM) stored in a single thread along with proposed method. [9]

The bitsliced Viterbi decoder leverages SIMD (Single Instruction, Multiple Data) operations to process multiple bits in parallel. Implemented using CUDA, the algorithm comprises two primary kernels: the feedforward kernel, which performs the Add-Compare-Select (ACS) operations, and the traceback kernel, which oversees the path selection process. The complete architecture is depicted in Figure 2.2.3.3.

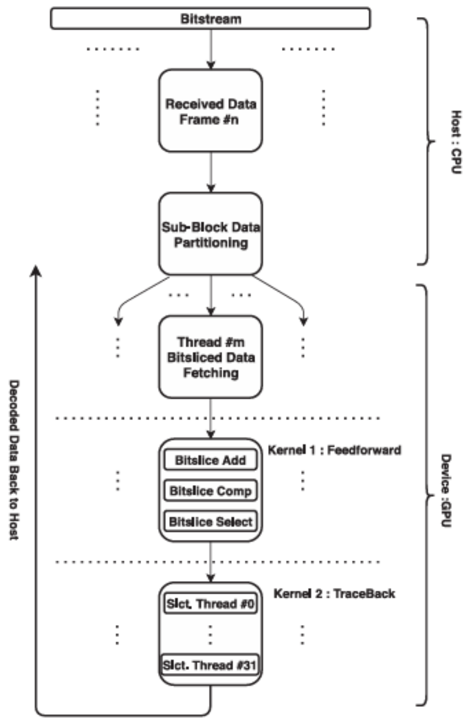


Figure 2.2.3.3: Full architecture of bitsliced Viterbi Decoder. [9]

In a nutshell, the proposed bitsliced Viterbi algorithm represents a major leap forward in GPU-based Viterbi decoding, delivering high throughput and efficiency. This makes it ideal for contemporary high-throughput communication systems and error correction applications. By incorporating similar strategies, this project could achieve enhanced performance and adaptability, crucial for applications requiring high-speed and low-power decoding solutions. The performance evaluation is detailed in Table 2.2.3.1. [9]

Method	**Throughput (Gbps)**	**Improvement Factor**
Traditional GPU Viterbi Decoder	1.8 - 6.64	-
Proposed Bitsliced Viterbi Decoder (Hard decision)	21.41	4.3x
Proposed Bitsliced Viterbi Decoder (Soft decision)	8.24	2.3x

Table 2.2.3.1: Performance comparison between proposed method implemented on Tesla V100 with traditional GPU implementations [9]

2.3 AMD AI Engine

The Versal AI Core series introduces groundbreaking AI inference acceleration through its AI Engines, which provide over 100x greater compute performance compared to current server-class CPUs. This versatile series is purpose-built for a wide range of applications, including cloud environments handling dynamic workloads and network setups requiring massive bandwidth. Additionally, it incorporates advanced safety and security features.

AI Engines are designed as 2D arrays composed of multiple AI Engine tiles, providing a highly scalable solution across the Versal portfolio, with configurations ranging from tens to hundreds of AI engines in a single device to meet diverse computational needs. Key benefits of using AI engines include software programmability, and a library-based design tailored for machine learning framework developers. The deterministic nature of AI Engines is ensured by dedicated instruction and data memories, along with dedicated connectivity and DMA engines for scheduled data movement between tiles. Additionally, AI Engines offer significant efficiency improvements, delivering up to 8 times better silicon area compute density compared to traditional programmable logic DSP and ML implementations, while reducing power consumption by approximately 40%. Figure 2.3.1 below show the architecture of AI Engine with multiple AI Engine tiles arranged in 2D array.

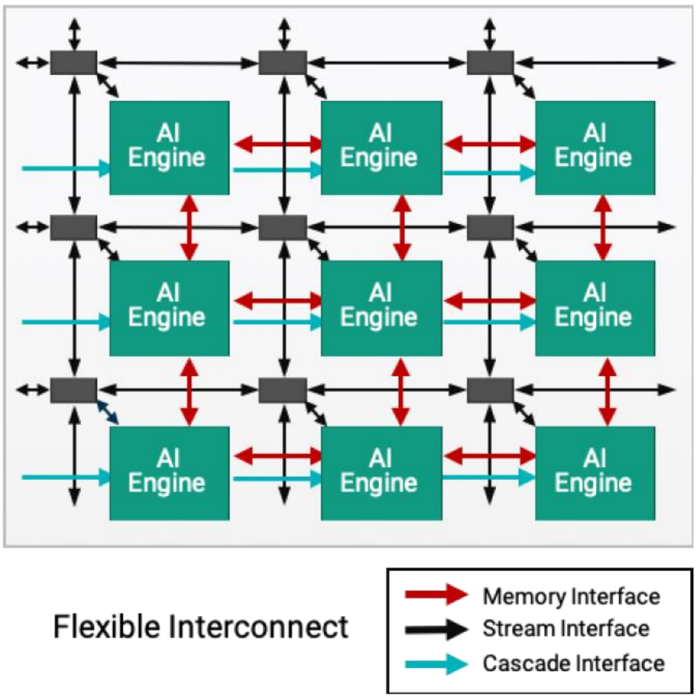


Figure 2.3.1: Schematic of AI engine architecture[10]

Each AI Engine tile comprises a Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) vector processor, optimized for machine learning and advanced signal processing applications. In addition to the VLIW vector processor, each tile includes program memory for storing instructions, local data memory for storing data, weights, activations, and coefficients, a RISC scalar processor, and various interconnect modes to handle different types of data communication. [10]

2.3.1 Inference of AI Engine

AMD developed the AI Engine as part of its Versal Adaptive Compute Acceleration Platform (ACAP) to address the high computational demands of modern AI workloads, which traditional FPGA solutions struggle to meet. While FPGAs offer flexibility and performance benefits over GPUs, they face scalability challenges due to the overhead associated with their bit-level interconnects and fine-grained programmability. The AI Engine, in contrast, is designed as a highly integrated, multicore, heterogeneous device that combines ASIC-class compute capabilities with adaptable hardware, providing superior compute density, power efficiency, and low latency for AI inference tasks, particularly those based on convolutional neural networks (CNNs).

The AI Engine architecture features a 2D array of software-programmable vector processors with flexible interconnects and tightly coupled local memory, enabling massive parallelism and efficient data movement. This design allows the Versal AI Core series to achieve 2.7 times the performance per watt compared to competing 10nm FPGAs, making it an ideal solution for compute-intensive AI applications. By integrating AI Engines, AMD can offer a scalable solution that supports a wide range of neural network models and adapts to evolving AI algorithms, thus overcoming the limitations of traditional FPGAs and meeting the increasing demands of AI workloads in data centres and other high-performance environments. [11]

2.3.2 AI Engine Applications Development Flow

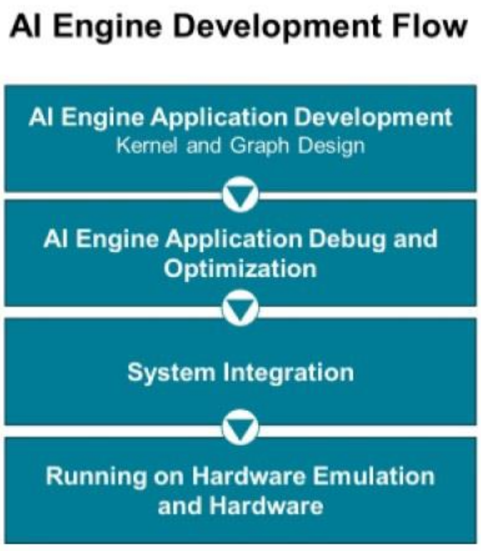

AI Engine Development Flow
AI Engine Application Development
Kernel and Graph Design
AI Engine Application Debug and Optimization
System Integration
Running on Hardware Emulation and Hardware

Figure 2.3.2.1: Comprehensive development flow of the AMD AI Engine. [12]

Figure 2.3.2.1 illustrates the comprehensive development flow of the AMD AI Engine. As shown in the figure below, the AI Engine development flow consists of four key stages, each building on the previous one to ensure a comprehensive and efficient development process [13]:

1. **AI Engine Application Development**:

   This initial stage involves designing the kernels (basic computational units) and the graph (which connects these kernels).

2. **AI Engine Application Debug and Optimization**:

   Once the kernels and graph are designed, the application undergoes rigorous debugging and optimization. This step ensures that the application runs smoothly, efficiently, and without errors. Performance tuning is also conducted to maximize speed and resource utilization.

3. **System Integration**:

After debugging and optimization, the application is integrated into the larger system. This involves create high level synthesis (HLS) kernels used to communicate between AI Engine kernels with other components such as memory on the board and conducting system-level testing to verify overall functionality.

4. **Running on Hardware**:

   The final stage involves deploying the application on the actual hardware, such as the VEK280 evaluation kit. This step includes hardware emulation, real-world testing, and final performance validation to ensure that the application meets all specified requirements and operates effectively in a real-world environment.

2.4 AMD Versal AI Edge VEK280 Evaluation Kit

The AMD Versal AI Edge VEK280 Evaluation Kit is a robust platform for ML and compute-intensive applications. It features the VE2802 device with over 200 dense TOPs, AI Engines, DSP Engines, Arm Cortex processors, and various peripherals. The kit offers connectivity options, software tools, and example designs for sensor-to-AI applications across markets. Key features include high-speed data communication, FMC+ connector, LPDDR4 memory, PCIe support, HDMI I/O, and a microSD card interface. Development tools include Vivado and Vitis, along with example designs and tutorials. The kit targets applications in vision, automotive, industrial, and aerospace domains. [14] Figure 2.4.1 illustrates the block schematic of VEK280 board.

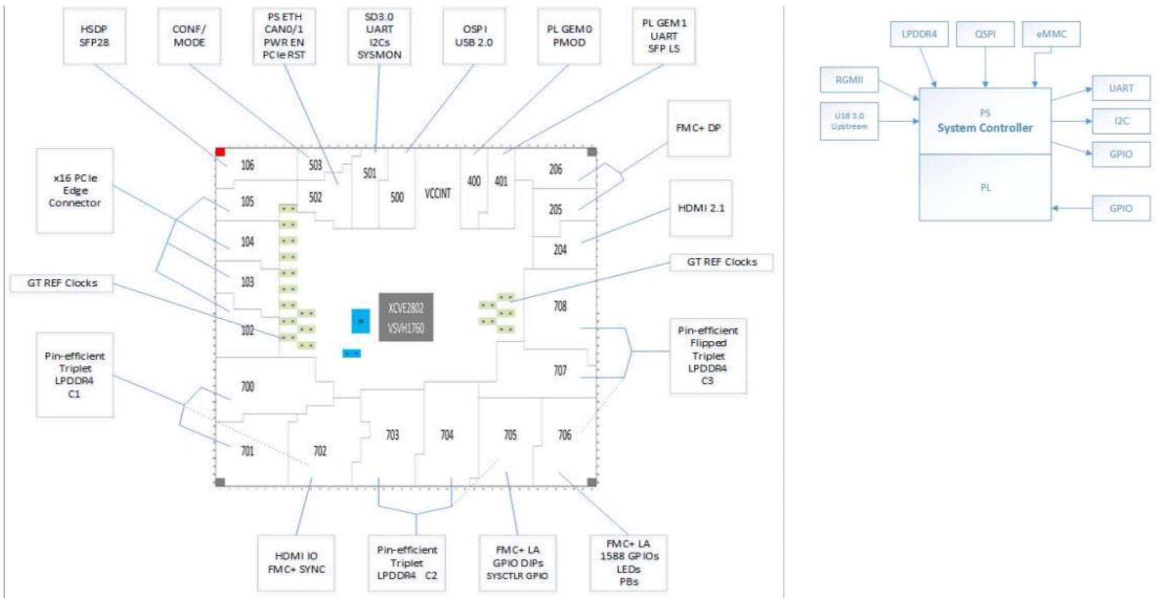


Figure 2.4.1: Block schematic diagram of AMD Versal AI Edge VEK280 Evaluation Kit. [15]

2.5 HACC

The Heterogeneous Accelerated Compute Clusters (HACCs) program, is a special initiative under the AMD Xilinx University Program, supports innovative research in adaptive compute acceleration for high-performance computing (HPC). HACCs are equipped with the latest AMD Xilinx hardware and software technologies, enabling academic teams to conduct cutting-edge HPC research. The program’s scope is extensive, covering systems, architecture, tools, and applications.

Five Heterogeneous Accelerated Compute Clusters (HACCs) have been established at some of the world’s most prestigious universities, including the National University of Singapore (NUS) and the University of California, Los Angeles (UCLA). [16]

In this research, the HACC cluster at NUS will be utilized for hardware emulation purposes, leveraging its advanced capabilities to test and validate the implementation of the Viterbi decoding algorithm. This access to high-end computational resources will facilitate detailed performance analysis and optimization, ensuring the robustness and efficiency of the developed solutions.

Chapter 3: Proposed Methods / Approaches

3.1 Proposed Solution

The proposed solution entails implementing the Viterbi Decoding algorithm in two distinct environments: C++ (CPU) and AI Engine. The AI Engine implementation is specifically designed to leverage its parallel computing capabilities to the fullest by employing three specialized kernels: the table create kernel, the feedforward kernel, and the traceback kernel. This strategic approach aims to optimize performance by utilizing parallelism, thereby enhancing processing speed and overall efficiency.

To thoroughly evaluate the effectiveness of both implementations, a comparative analysis will be conducted. This analysis will measure throughput in terms of Mega bits per second (Mbps) and accuracy in terms of bit error rate (BER). By examining these metrics, we can gain valuable insights into the efficiency and reliability of each approach. This comprehensive evaluation will not only highlight the strengths and weaknesses of the C++ and AI Engine implementations but also guide future optimizations and potential applications. The findings from this analysis will be instrumental in determining the most effective and reliable method for implementing the Viterbi Decoding algorithm, ensuring optimal performance in practical applications.

3.2 Research Methodology for the Project

3.2.1 Analyse the Viterbi Decoding Architecture

In order to perform parallelization, it is necessary to study the Viterbi decoder architecture and understand the parallelizable and unparallelizable parts of the architecture. The memory architecture of the trellis diagram also needs to be studied to transform into AI Engine compatible codes.

3.2.2 Setup the Vitis Tool Environment

In this project, Linux Ubuntu 22.04 LTS operating system is used as recommended by AMD to operate the Vitis software. To set up the Vitis tool environment, I need to install the Vitis library with the command “sudo installLibs.sh”. Next, the system needs to be installed with the Xilinx Runtime library (XRT) and the necessary platforms for interface development and connection to the VEK280 board. Following this, we will also need the embedded platform for the VEK280 board image which can be installed from AMD Xilinx product page.

Once these installations are complete, the system needs to be configured with the environment to run the Vitis software platform. This involves setting up the XRT path, platform repository path, and other necessary paths to identify the location of platform files and make them accessible to the design projects. This setup ensures that all required tools and files are properly linked and available for efficient project development.

3.3 Viterbi Decoder Modelling and Verification

The C++ level decoder modelling should be completed before the AI Engine level to ensure functional correctness. Each sub-function will be verified individually before being integrated into the full decoder.

Once the C++ level decoder meets the necessary specifications and expectations, the code will be analysed and divided into parallelizable and non-parallelizable parts. The parallelizable parts will then be coded into different kernels, which will be connected using a graph structure.

The AI Engine code will be simulated using Vitis 2024.1 to ensure compatibility with the AMD Xilinx VEK280 board. Noticeably, each kernel will also undergo functional verification before being integrated into the full decoder.

Next, to port the fully functional decoder to the VEK280 board, interfaces such as the system-to-memory port (s2mm), which transfers data from the AI Engine to memory, and the memory-to-system port (mm2s), which transfers data from memory to the AI Engine, need to be created. These interfaces are essential for moving data between different structures within the board. Their functionalities will be verified through simple write and read data movements from memory tiles to AI Engine kernels and from AI Engine kernels to memory tiles.

3.4 Benchmarking the Viterbi Decoder

To benchmark the Viterbi Decoder, two key performance metrics will be utilized: bit error rate (BER) and transfer bandwidth, measured in mega bits per second (Mbps). The bit error rate is a critical measure of accuracy, calculated by dividing the number of erroneous bits by the total number of transmitted bits. This metric provides insight into the reliability of the decoder by indicating the proportion of errors in the transmitted data. A lower BER signifies higher accuracy and better performance of the decoder. In this project, it is expected that both the C++ and AI Engine implementations of the Viterbi Decoder will yield similar BER data, as they will be tested with the same set of input data. The BER can be calculated using the following formula:

$$Bit\ Error\ Rate\ (BER) = \frac{Number\ of\ Errors\ Observed}{Total\ Transmitted\ Data\ (Bits)}$$

Conversely, the transfer rate, expressed in Mbps, measures the speed at which data is transferred. This metric is crucial for assessing the efficiency and speed of the decoder, as it indicates how rapidly data can be processed and transmitted. A higher transfer bandwidth signifies a more efficient and faster decoder. It is anticipated that the AI Engine implementation of the Viterbi Decoder will exhibit a higher transfer rate compared to the CPU implementation. The transfer bandwidth is calculated by measuring the amount of data successfully transferred per second, as shown in the formula below:

$$Transfer\ Rate\ (Mbps) = \frac{Number\ of\ Data\ Being\ Transferred\ (Bits)}{Time\ Taken\ for\ The\ Transfer\ Process\ (s)} \times \frac{1\ (M)}{1000000}$$

By analysing both BER and transfer bandwidth and compare the metrics gained from different implementations, we can gain a comprehensive understanding of the Viterbi Decoder's performance. These metrics will help identify areas for improvement and ensure that the decoder meets the required specifications for accuracy and speed. This thorough benchmarking process is crucial for optimizing the decoder's functionality and reliability in practical applications.

3.5 Technologies Involved

3.5.1 Xilinx Vitis Unified Software Platform

Xilinx Vitis Unified Software Platform is a development environment developed by AMD Xilinx for designing applications that involved Xilinx FPGA fabric, Arm processors subsystems and also AI Engines. This tool provides a comprehensive suite of tools, for example, Vitis Embedded for developing C / C++ applications that drive embedded Arm processors, Vitis HLS for creating C / C++ based IP blocks targeting various AMD Xilinx FPGA fabric, Vitis Model Composer for rapid design exploration within the MathWorks Simulink environment, and Vitis AIE / AIE-ML for efficient development and employment of AI Engine applications. This platform includes a set of open-source, performance-optimized library functions that can be implemented in FPGA fabrics and AI Engines. Vitis platform aims to offer a higher level of abstraction for design development, making it easier for developers to create efficient and high-performance applications. In this project, Xilinx Vitis Unified Software Platform will be used to develop and analyse the AI Engine implementation of Viterbi decoder. [17]

3.6.1 Xilinx Vivado

Xilinx Vivado is an integrated design environment (IDE) launched by AMD Xilinx for designing and implementing systems on a chip (SoCs) and field-programmable gate arrays (FPGAs). Vivado offers a comprehensive suite of tools for design entry, synthesis, place and route, and verification / simulation. It supports traditional hardware description languages (HDLs) like VHDL, Verilog and SystemVerilog, as well as high-level synthesis (HLS) for converting high-level programming languages like C and C++ codes into programmable logic.

Besides, Vivado also includes the IP Integrator, which allows for easy integration and configuration of intellectual property (IP) blocks, and a common debug environment to streamline the design processes. This suite is designed to enhance productivity while at the same time reduce the time required for design iterations, making it a powerful tool for FPGA and SoC development. [18]

In this project, the Vivado IDE will be used to analyse and verify the integration and connections of different kernels / IPs and memory graphically.

Chapter 4: System Design

4.1 Viterbi Decoder Flow Design

The system flow is meticulously designed to ensure systematic, efficient and accurate data decoding approach through a series of well-defined steps. The flowchart in Figure 4.1.1 below provides a simplified view of the entire process.

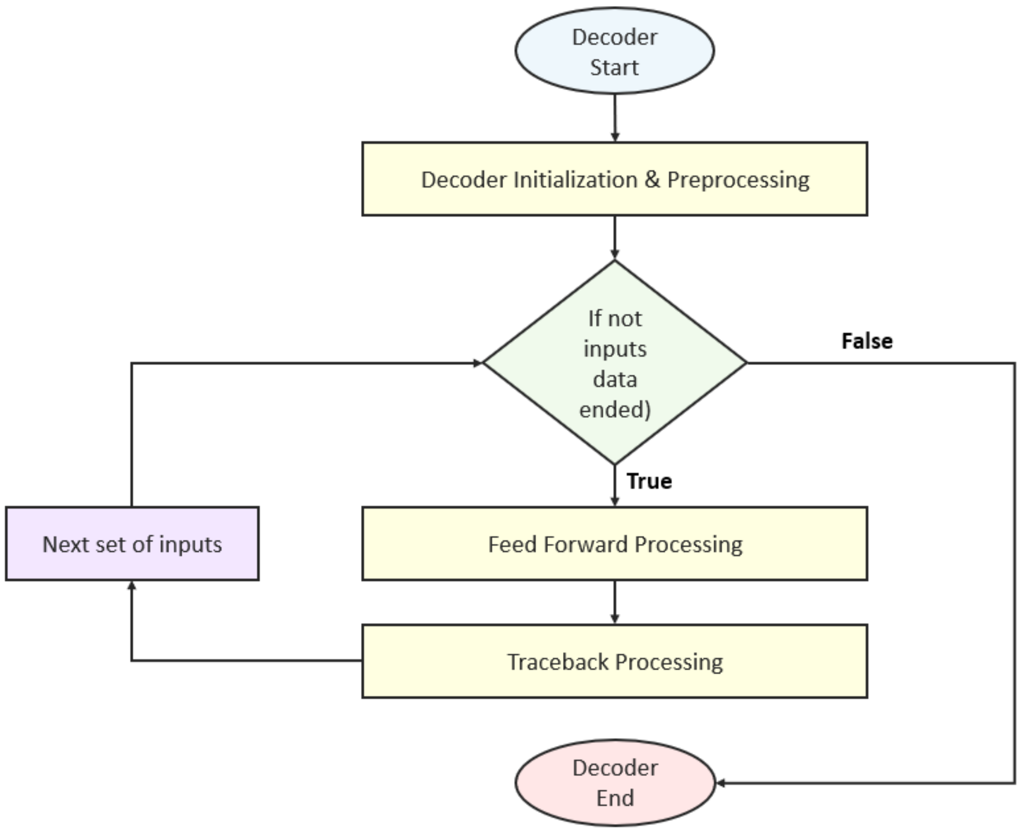


Figure 4.1.1: High-level overall flow chart for the designed Viterbi Decoder.

Below is a brief description of each stage in the system architecture:

1. **Decoder Initialization and Preprocessing:**

   The process begins with the initialization of the decoder. This step sets up the necessary environment and parameters required for the decoding process. Besides, this step is crucial as it prepares the raw input data and transform it into soft decision decoding format for subsequent processing stages.

2. **Input Data Looping:**

   Following preprocessing, the system encounters a loop to ensure that the system can continuously process new sets of data until all input data has been processed. The loop is a critical component of the architecture, enabling the system to handle large volumes of data efficiently.

3. **Feed Forward Processing:**

   In this stage, the pre-processed data undergoes feed forward processing. This involves passing the data through a series of computations to determine the accumulated error metric.

4. **Traceback Processing:**

   After feed-forward processing, the data is subjected to traceback processing. This step involves scanning through the trellis diagram and determine the most accurate data (data with lowest accumulated error metric) and return it as decoded data.

The comprehensive description of the architecture will be elaborated upon in the subsequent sections.

4.1.1 Decoder Initialization and Preprocessing

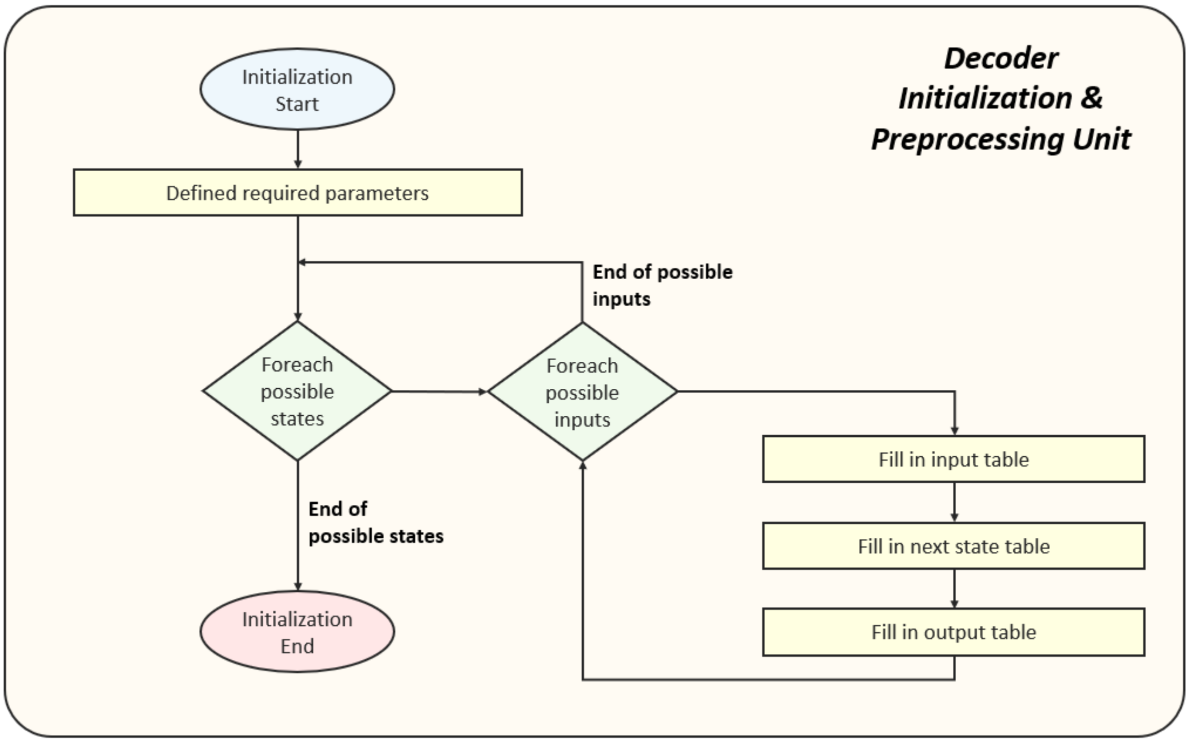


Figure 4.1.1.1: Comprehensive flow chart of the microarchitecture of decoder initialization and preprocessing unit.

The flow chart in Figure 4.1.1.1 above provides a detailed explanation of the steps involved in step "Decoder initialization and preprocessing". The analysis will break down the important component of the flowchart, explaining its role and significance within the overall decoder architecture and provide a clear understanding of the first step of the decoder.

1. **Defined Required Parameters**

   This step involves setting up the initial conditions and configuration parameters such as constraint length, K, number of input bits, k, number of output bits, n, number of possible states, etc. These configurations are needed and will be used for the entire decoding processes. These parameters are essential and will guide the subsequent steps and ensure that the system operates correctly.

2. **Filling up the tables**

Within the nested loop, input tables, output tables and next state table will be filled to smoothen the entire decoding processes. This step helps to minimize the repeating computation of same set of data over time when the later processes are undergoing. The description of each table is shown in Table 4.1.1.1 below:

Table Name	**Description**
Input Table	Describe how FEC encoder bits lead to next state.
Output Table	Shows the output bits of FEC encoder, given current presumed encoder state and encoder input bit. This will be compared to actual received symbols to determine the branch metric for corresponding branch of trellis.
Next State Table	Show the “next state” of the convolutional encoder, given current state and input bit.

Table 4.1.1.1: Description of each table that will be used in the decoding processes.

3. **Initialization End**

Once the nested loops have completed their respective paths, the system is now fully prepared to begin the actual decoding processes.

4.1.2 Feed Forward Processing

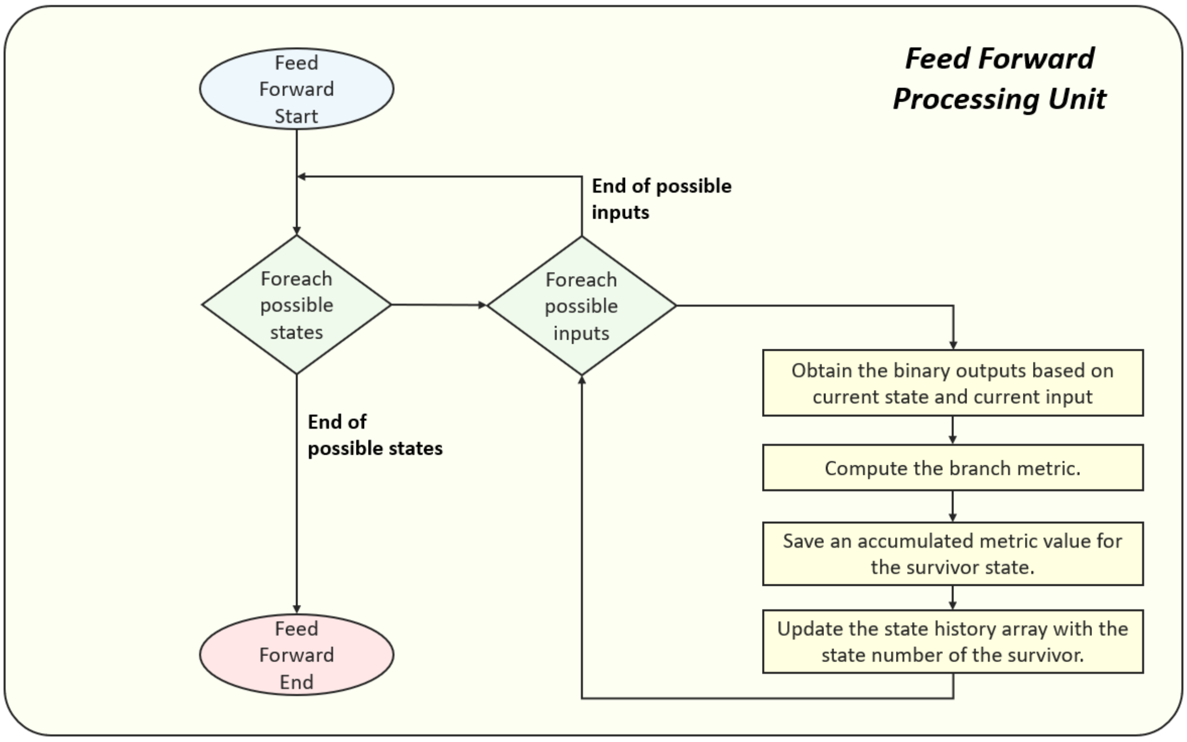

Feed Forward Processing Unit
Feed Forward Start
End of possible inputs
Foreach possible states
Foreach possible inputs
Obtain the binary outputs based on current state and current input
Compute the branch metric.
Save an accumulated metric value for the survivor state.
Update the state history array with the state number of the survivor.
End of possible states
Feed Forward End

Figure 4.1.2.1: Comprehensive flow chart of the microarchitecture of the feed forward processing unit.

The flowchart titled "Feed Forward Processing Unit" in Figure 4.1.2.1 above provides a detailed representation of the steps involved in the feed-forward processing stage. This analysis will break down the important components of the flowchart, explaining its role and significance within the overall decoder architecture. Clear explanations are provided as follows:

1. **Obtain Binary Outputs**

   This step transforms the "expected output" of the FEC encoder using the output table. For further details, please refer to section 4.1.1.

2. **Compute the Branch Metric**

   The subsequent step involves computing the branch metric. This metric is a measure of the likelihood between the actual output and the "expected" output associated with transitioning from one state to another based on the current input. Computing the branch metric is crucial for evaluating the best possible state transitions and ensuring accurate processing.

3. **Save Accumulated Metric for the Survivor State**

   Following the computation of the branch metric, the system proceeds to save the accumulator branch metric value for the survivor state, which is the state with lowest branch metric. This step ensures that the most likely or optimal state transitions are preserved for further processing.

4. **Update State History Array**

   The final step involves updating the state history array with the state number of the survivor. This step helps to build up the trellis diagram of the Viterbi decoder and is crucial for maintaining a record of the state transitions and ensuring that the system can accurately track the progression of states throughout the processing.

5. **End of Feed Forward Processing**

   Once all the steps have been completed, the system reaches the "Feed Forward End" point, signifying the termination of the feed-forward processing stage. The required data is now ready for the last stage, Traceback.

4.1.3 Traceback Processing

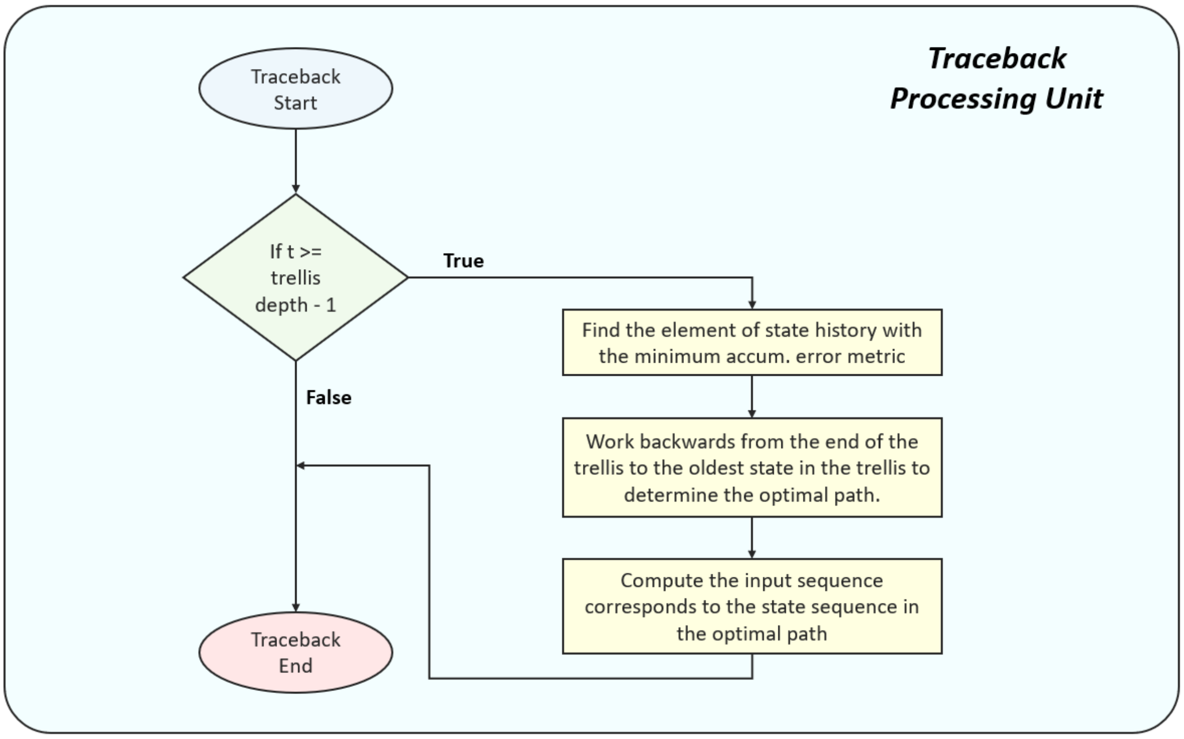

Traceback Processing Unit
Traceback Start
If t >= trellis depth - 1
True
False
Find the element of state history with the minimum accum. error metric
Work backwards from the end of the trellis to the oldest state in the trellis to determine the optimal path.
Compute the input sequence corresponds to the state sequence in the optimal path
Traceback End

Figure 4.1.3.1: Comprehensive flow chart of the microarchitecture of the traceback processing unit.

The flowchart titled “Traceback Processing Unit” in Figure 4.1.3.1 above provides a detailed representation of the steps involved in the traceback processing stage. This analysis will break down the important components of the flowchart, explaining its role and significance within the overall decoder architecture. Clear explanations are provided as follows:

1. **Decision Point - Trellis Depth Check**

   The first decision point checks if the current time step (t) is greater than or equal to the trellis depth minus one. This condition ensures that the traceback process only begins once the trellis has been fully filled with data.

2. **Find the Element of State History with Minimum Accumulated Error Metric**

   In this step, the decoder will loop through the possible states to find the state with the minimum accumulated error metric of the last entry of trellis / state history array. This step is crucial for determining the most likely state at the end of the trellis.

3. **Work Backwards from the End of the Trellis**

   The decoder then works backwards from the end of the trellis to the oldest state, tracing the optimal path. This step involves retracing the steps taken to reach the final state, ensuring that the most accurate path is identified.

4. **Compute the Input Sequence Corresponding to the State Sequence**

   Finally, the system computes the input sequence that corresponds to the identified state sequence in the optimal path by utilizing the input table. Please refer to section 4.1.1 for more details. This step translates the state transitions into the original input data, completing the traceback process.

5. **Traceback End**

   Once all the necessary steps have been completed, the system reaches the “Traceback End” point, signifying the termination of the traceback processing stage. At this point, the decoded data is now ready and will return as output of the decoder.

4.2 Implementation of Viterbi Decoder in C++

The C++ implementation for the Viterbi decoder serves as a baseline to ensure the functional correctness of the decoder and acts as a reference implementation to compare against other implementations, such as the AI Engines implementation which is part of the objectives in this project. To achieve maximum flexibility, dynamic memory allocation and the use of vector data structures in C++ are employed. This approach allows for easy adaptation to different test cases and parameters without requiring code modifications.

Besides, the design employs a simple and efficient file structure. This organization helps maintain a clean and manageable codebase. Please refer to section 4.2.1 for details of the file structure.

The two magic numbers, the transfer bandwidth and bit error rate (BER) are measured and compared with those obtained from an AI Engine implementation. These metrics help evaluate performance improvements and highlight differences between the implementations.

4.2.1 File Structure

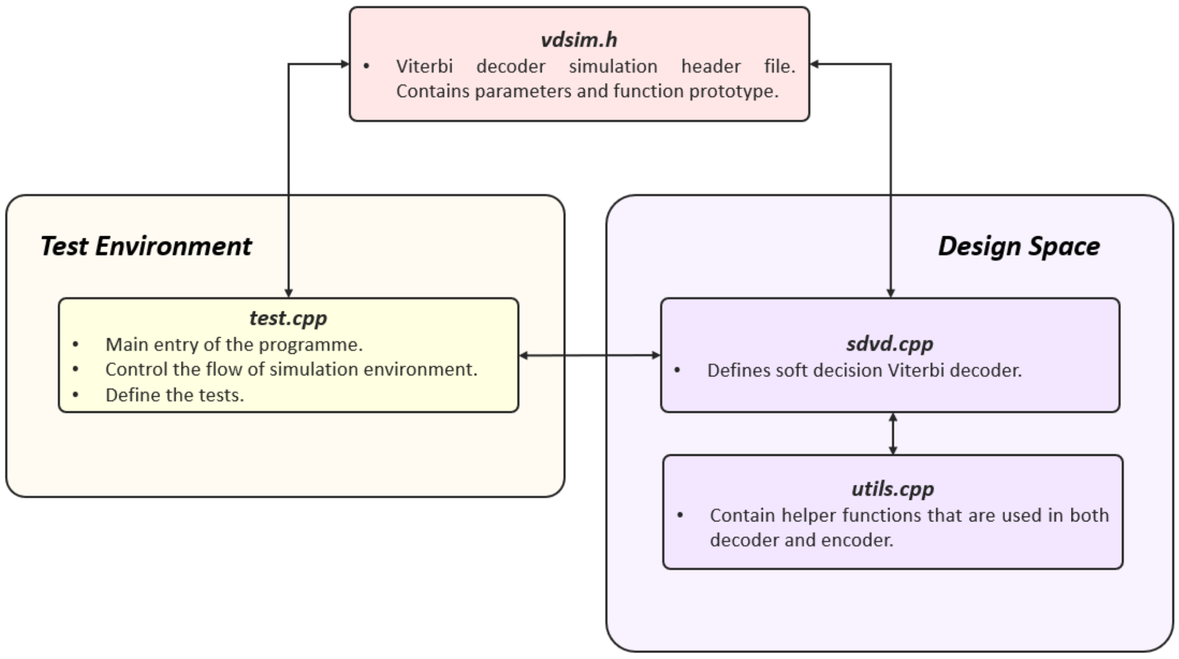

vdsim.h
• Viterbi decoder simulation header file. Contains parameters and function prototype.
Test Environment
test.cpp
• Main entry of the programme.
• Control the flow of simulation environment.
• Define the tests.
Design Space
sdvd.cpp
• Defines soft decision Viterbi decoder.
utils.cpp
• Contain helper functions that are used in both decoder and encoder.

Figure 4.2.1.1: File Structure of the full C++ Viterbi simulation environment.

Figure 4.2.1.1 illustrates the file structure of the testing environment for Viterbi decoder. It is divided into two main sections: **Test Environment** and **Design Space**.

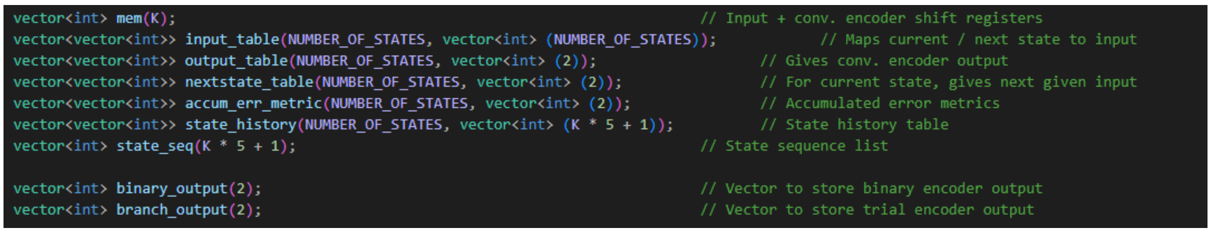
The header files, vdsim.h contains parameters and function prototypes necessary for the decoder’s operation.

The test environment, which consists of test.cpp, is responsible for the main entry of the program, controlling the flow of the simulation environment, and defining the tests. It ensures that the program operates correctly and efficiently under various conditions.

The design space includes the core components of the Viterbi decoder design. It is divided into two main files, which are the sdvd.cpp and utils.cpp. The sdvd.cpp file defines the soft decision Viterbi decoder, implementing the core decoding algorithm, while utils.cpp file contains helper functions used by both the decoder and encoder, providing common utilities to support the main decoding functions.

4.2.2 Actual C++ Design

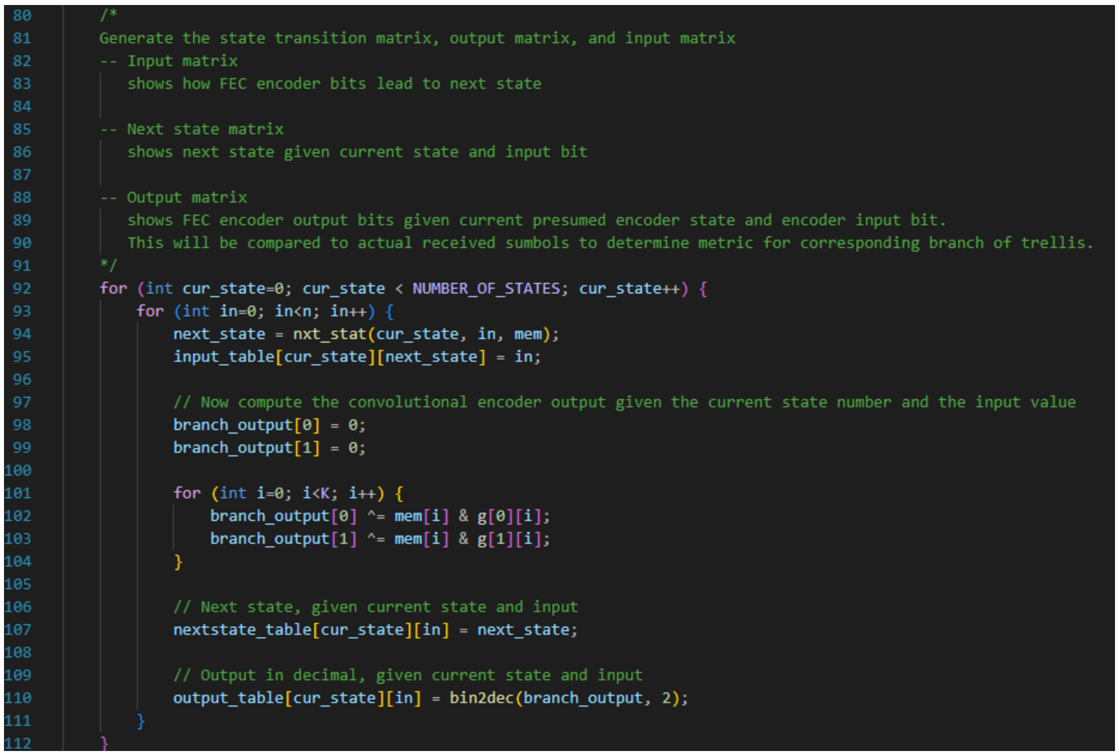
This section presents the real implementation of Viterbi Decoder in C++, following the flow discussed in section 4.1. The design ensures both functional correctness and flexibility by leveraging dynamic memory allocation and the std::vector data structure. This approach allows the decoder to handle varying input sizes and test cases without requiring code modifications. Figure 4.2.2.1 below shows the data structure used in the C++ design.



```cpp
vector<int> mem(K);                                                               // Input + conv. encoder shift registers
vector<vector<int>> input_table(NUMBER_OF_STATES, vector<int> (NUMBER_OF_STATES));           // Maps current / next state to input
vector<vector<int>> output_table(NUMBER_OF_STATES, vector<int> (2));                      // Gives conv. encoder output
vector<vector<int>> nextstate_table(NUMBER_OF_STATES, vector<int> (2));                   // For current state, gives next given input
vector<vector<int>> accum_err_metric(NUMBER_OF_STATES, vector<int> (2));                  // Accumulated error metrics
vector<vector<int>> state_history(NUMBER_OF_STATES, vector<int> (K * 5 + 1));             // State history table
vector<int> state_seq(K * 5 + 1);                                                 // State sequence list

vector<int> binary_output(2);                                                     // Vector to store binary encoder output
vector<int> branch_output(2);                                                     // Vector to store trial encoder output
```

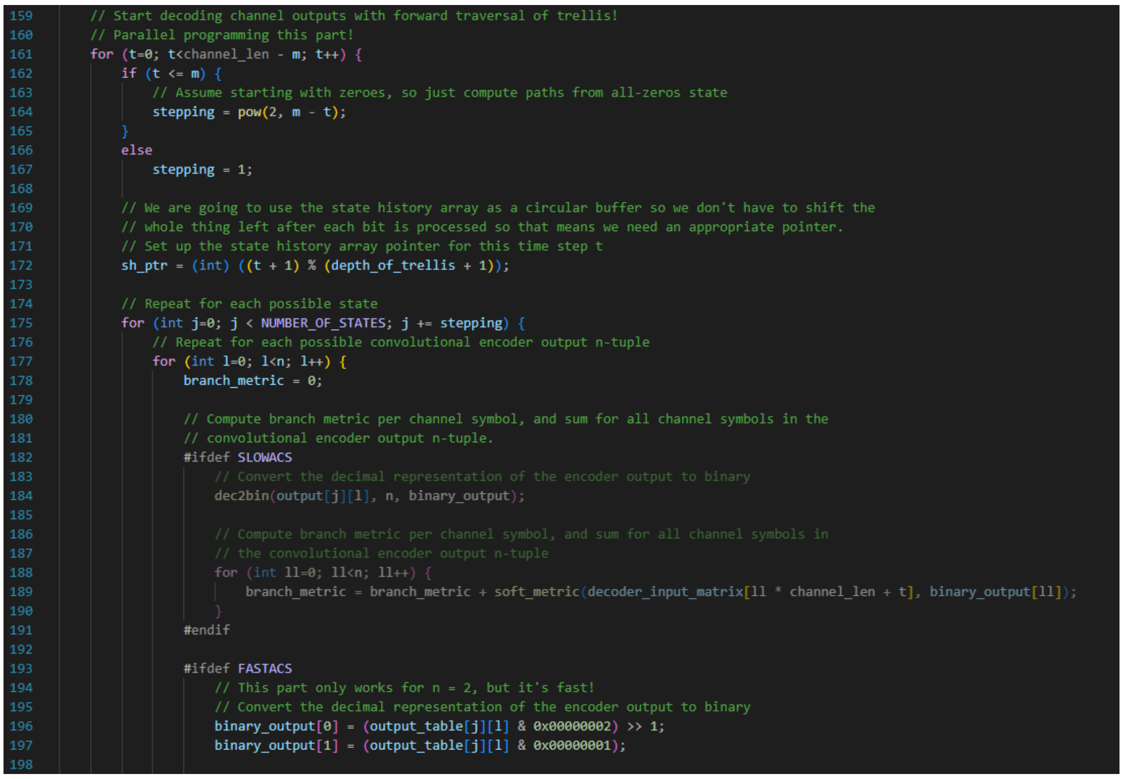
Figure 4.2.2.1: Data structure used in the C++ implementation.

Figure 4.2.2.2, Figure 4.2.2.3 and Figure 4.2.2.4 shows part of the implementation of decoder initialization and preprocessing, feed forward processing and traceback processing, please refer to section 4.1 to understand the flow.



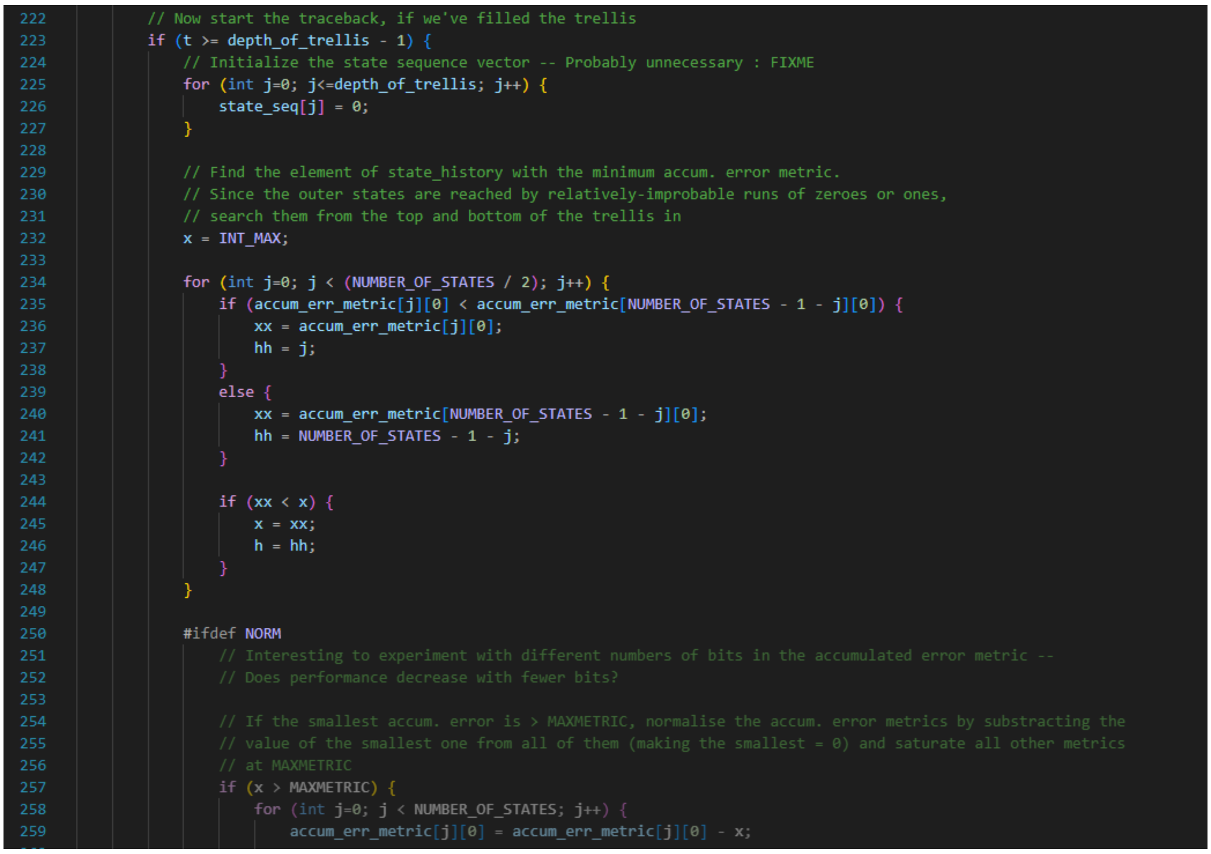
```cpp
/*
Generate the state transition matrix, output matrix, and input matrix
-- Input matrix
    shows how FEC encoder bits lead to next state

-- Next state matrix
    shows next state given current state and input bit

-- Output matrix
    shows FEC encoder output bits given current presumed encoder state and encoder input bit.
    This will be compared to actual received sumbols to determine metric for corresponding branch of trellis.
*/
for (int cur_state=0; cur_state < NUMBER_OF_STATES; cur_state++) {
    for (int in=0; in<n; in++) {
        next_state = nxt_stat(cur_state, in, mem);
        input_table[cur_state][next_state] = in;

        // Now compute the convolutional encoder output given the current state number and the input value
        branch_output[0] = 0;
        branch_output[1] = 0;

        for (int i=0; i<K; i++) {
            branch_output[0] ^= mem[i] & g[0][i];
            branch_output[1] ^= mem[i] & g[1][i];
        }

        // Next state, given current state and input
        nextstate_table[cur_state][in] = next_state;

        // Output in decimal, given current state and input
        output_table[cur_state][in] = bin2dec(branch_output, 2);
    }
}
```

Figure 4.2.2.2: C++ design of Decoder Initialization and Preprocessing Unit.



```cpp
// Start decoding channel outputs with forward traversal of trellis!
// Parallel programming this part!
for (t=0; t<channel_len - m; t++) {
    if (t <= m) {
        // Assume starting with zeroes, so just compute paths from all-zeros state
        stepping = pow(2, m - t);
    }
    else
        stepping = 1;

    // We are going to use the state history array as a circular buffer so we don't have to shift the
    // whole thing left after each bit is processed so that means we need an appropriate pointer.
    // Set up the state history array pointer for this time step t
    sh_ptr = (int) ((t + 1) % (depth_of_trellis + 1));

    // Repeat for each possible state
    for (int j=0; j < NUMBER_OF_STATES; j += stepping) {
        // Repeat for each possible convolutional encoder output n-tuple
        for (int l=0; l<n; l++) {
            branch_metric = 0;

            // Compute branch metric per channel symbol, and sum for all channel symbols in the
            // convolutional encoder output n-tuple.
            #ifdef SLOWACS
                // Convert the decimal representation of the encoder output to binary
                dec2bin(output[j][l], n, binary_output);

                // Compute branch metric per channel symbol, and sum for all channel symbols in
                // the convolutional encoder output n-tuple
                for (int ll=0; ll<n; ll++) {
                    branch_metric = branch_metric + soft_metric(decoder_input_matrix[ll * channel_len + t], binary_output[ll]);
                }
            #endif

            #ifdef FASTACS
                // This part only works for n = 2, but it's fast!
                // Convert the decimal representation of the encoder output to binary
                binary_output[0] = (output_table[j][l] & 0x00000002) >> 1;
                binary_output[1] = (output_table[j][l] & 0x00000001);

```

Figure 4.2.2.3: C++ design of Feed Forward Processing Unit.



```
        // Now start the traceback, if we've filled the trellis
        if (t >= depth_of_trellis - 1) {
            // Initialize the state sequence vector -- Probably unnecessary : FIXME
            for (int j=0; j<=depth_of_trellis; j++) {
                state_seq[j] = 0;
            }

            // Find the element of state_history with the minimum accum. error metric.
            // Since the outer states are reached by relatively-improbable runs of zeroes or ones,
            // search them from the top and bottom of the trellis in
            x = INT_MAX;

            for (int j=0; j < (NUMBER_OF_STATES / 2); j++) {
                if (accum_err_metric[j][0] < accum_err_metric[NUMBER_OF_STATES - 1 - j][0]) {
                    xx = accum_err_metric[j][0];
                    hh = j;
                }
                else {
                    xx = accum_err_metric[NUMBER_OF_STATES - 1 - j][0];
                    hh = NUMBER_OF_STATES - 1 - j;
                }

                if (xx < x) {
                    x = xx;
                    h = hh;
                }
            }

            #ifdef NORM
                // Interesting to experiment with different numbers of bits in the accumulated error metric --
                // Does performance decrease with fewer bits?

                // If the smallest accum. error is > MAXMETRIC, normalise the accum. error metrics by substracting the
                // value of the smallest one from all of them (making the smallest = 0) and saturate all other metrics
                // at MAXMETRIC
                if (x > MAXMETRIC) {
                    for (int j=0; j < NUMBER_OF_STATES; j++) {
                        accum_err_metric[j][0] = accum_err_metric[j][0] - x;
```

Figure 4.2.2.4: C++ design of Traceback Processing Unit.

4.2.3 Test Plans

The test plan for both C++ implementation is divided into two main groups. The first group focuses on verifying the functionality correctness of the design, ensuring that the Viterbi decoder operates as intended. The second group tests the decoder with large volumes of data, comparing the results against those obtained from the AI Engine implementation to evaluate performance and accuracy.

Table 4.2.3.1 below outlines the Group 1 test plan, which aims to verify the functional correctness of the design. By ensuring that the Bit Error Rate (BER) for each test remains below the intended threshold, we can conclude that the decoder is functioning as intended and is free from logical errors.

Test Number	Test Objectives	Constraint Length, K	Es/No Ratio	Message Length	Target
1	Verify the decoder under various constraint lengths	3	1.0	1000	BER < 0.01
2		5	1.0	1000	BER < 0.01
3		7	1.0	1000	BER < 0.005
4		9	1.0	1000	BER < 0.005
5	Verify decoder under various Es/No ratio.	5	0.5	1000	BER < 0.01
6		5	1.0	1000	BER < 0.01
7		5	1.5	1000	BER < 0.01
8		5	2.0	1000	BER < 0.005
9		5	2.5	1000	BER < 0.005
10		5	3.0	1000	BER < 0.005

Table 4.2.3.1: Group 1 test plan for functionalities verification.

Following this, Table 4.2.3.2 outlines the Group 2 test plan. This plan compares both the BER and transfer bandwidth against the results obtained from the AI Engine implementation. Consequently, this test plan does not specify an intended target but instead tests for large volumes of messages. Besides, the Es/No ratio has remained constant for group 2 since it does not affect the speed of Viterbi decoding algorithm.

Test Number	Constraint Length, K	Es/No Ratio	Message Length
11	3	0.2	10000
12	5	0.2	10000
13	7	0.2	10000
14	9	0.2	10000

Table 4.2.3.2: Group 2 test plan for comparative performance analysis.

4.3 Implementation of Viterbi Decoder using AI Engine

This section provides a detailed discussion on the implementation of the Viterbi Decoder using an AI Engine. The AI Engine approach leverages the parallel programming technique to enhance the performance and accuracy of the Viterbi decoding process. The implementation details cover the constraints of AI Engine application, micro-architecture of the design, graph, and system integration strategies to achieve efficient decoding.

Similar to C++ implementation, performance metrics like BER and transfer bandwidth will be compared between the AI Engine implementation and the C++ implementation. The test plan was written and is discussed in section 4.3.5.

4.3.1 Constraints of AI Engines Applications

When implementing the Viterbi Decoder using AI Engines, several constraints must be considered to ensure optimal performance and compatibility. These constraints arise from the specific architecture and capabilities of AI Engines, which differ from traditional C++ environments.

1. **Limited Input and Output Ports for AI Engine Kernel:**

   Each AI Engine kernel is restricted to a maximum of two input ports and two output ports. This limitation necessitates careful planning and design to ensure that data flows efficiently through the system. Developers must optimize the use of these ports to handle the required data throughput without causing bottlenecks or delays in processing.

2. **Restricted Data Types for Vectors:**

   Although AI Engines support C++ syntax, their vector data structures only accommodate a limited set of data types, which differ from the standard std::vector.

Besides, accessing vector elements in AI Engines is more complex, requiring additional considerations for indexing and manipulation. This constraint means that developers must adapt their code to work with the supported data types and manage vector operations more meticulously.

3. **Custom Floating/Double Data Types Processing:**

   AI Engines have their own methods for processing floating-point and double-precision data types, which are not compatible with the standard C++ library functions. As a result, developers cannot use typical C++ functions for mathematical operations involving these data types. Instead, they must either utilize equivalent functions from the AI Engine library or develop custom implementations to handle these operations.

4.3.2 Micro-Architecture of the Kernels

4.3.2.1 Table Construct Kernel

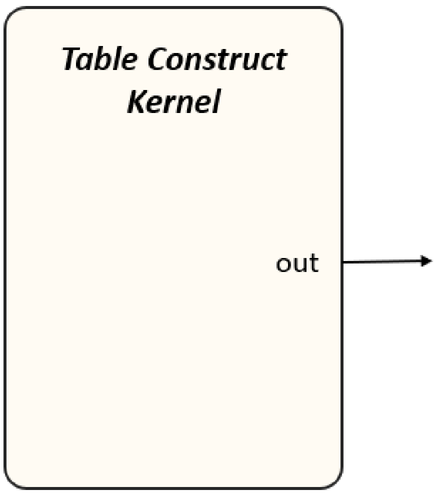
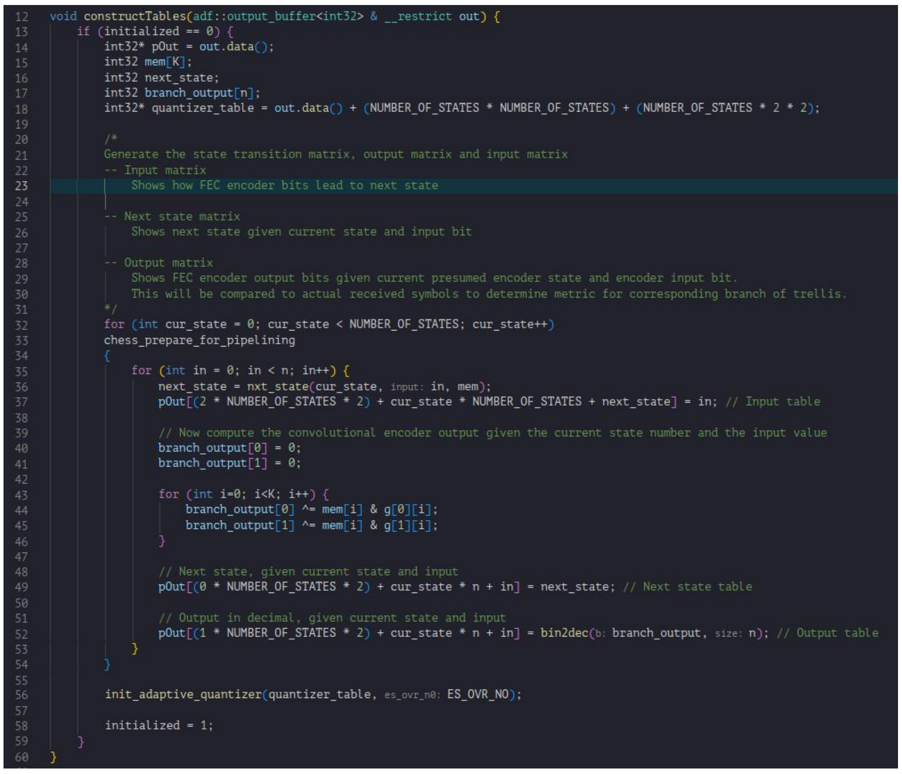


Figure 4.3.2.1.1: Schematic diagram of table construct kernel.

The Table Construct Kernel is responsible for initializing and generating key matrices / tables essential to Viterbi decoding processes. The kernel is designed to only execute once at the very beginning of the decoding process, and it will construct three primary table: the input table, next state table, and output table. The input matrix maps FEC encoder bits to subsequent states, the next state matrix indicates states transition based on current states and inputs, and the output matrix reflects FEC encoder outputs from presumed states and inputs. These matrices are crucial to determining metrics within trellis decoding structures by comparing the presumed encoder outputs with actual received symbols. This kernel is designed to follow the decoder initialization and preprocessing unit as discussed in section 4.1.1.

As shown in Figure 4.3.2.1.1, the table construct kernel has a single output port named "out". The data from all three tables will be organized and fed into a shared buffer used between kernels, adhering to the restriction of a maximum of two output ports. Figure 4.3.2.1.2 below shows the AI Engine implementation for the table construct kernel function.



```
void constructTables(adf::output_buffer<int32> & __restrict out) {
    if (initialized == 0) {
        int32* pOut = out.data();
        int32 mem[K];
        int32 next_state;
        int32 branch_output[n];
        int32* quantizer_table = out.data() + (NUMBER_OF_STATES * NUMBER_OF_STATES) + (NUMBER_OF_STATES * 2 * 2);

        /*
        Generate the state transition matrix, output matrix and input matrix
        -- Input matrix
            Shows how FEC encoder bits lead to next state

        -- Next state matrix
            Shows next state given current state and input bit

        -- Output matrix
            Shows FEC encoder output bits given current presumed encoder state and encoder input bit.
            This will be compared to actual received symbols to determine metric for corresponding branch of trellis.
        */
        for (int cur_state = 0; cur_state < NUMBER_OF_STATES; cur_state++)
        chess_prepare_for_pipelining
        {
            for (int in = 0; in < n; in++) {
                next_state = nxt_state(cur_state, input: in, mem);
                pOut[(2 * NUMBER_OF_STATES * 2) + cur_state * NUMBER_OF_STATES + next_state] = in; // Input table

                // Now compute the convolutional encoder output given the current state number and the input value
                branch_output[0] = 0;
                branch_output[1] = 0;

                for (int i=0; i<K; i++) {
                    branch_output[0] ^= mem[i] & g[0][i];
                    branch_output[1] ^= mem[i] & g[1][i];
                }

                // Next state, given current state and input
                pOut[(0 * NUMBER_OF_STATES * 2) + cur_state * n + in] = next_state; // Next state table

                // Output in decimal, given current state and input
                pOut[(1 * NUMBER_OF_STATES * 2) + cur_state * n + in] = bin2dec(b: branch_output, size: n); // Output table
            }
        }

        init_adaptive_quantizer(quantizer_table, es_ovr_n0: ES_OVR_NO);

        initialized = 1;
    }
}
```

Figure 4.3.2.1.2: AI Engine implementation for the table construct kernel.

4.3.2.2 Feed Forward Kernel

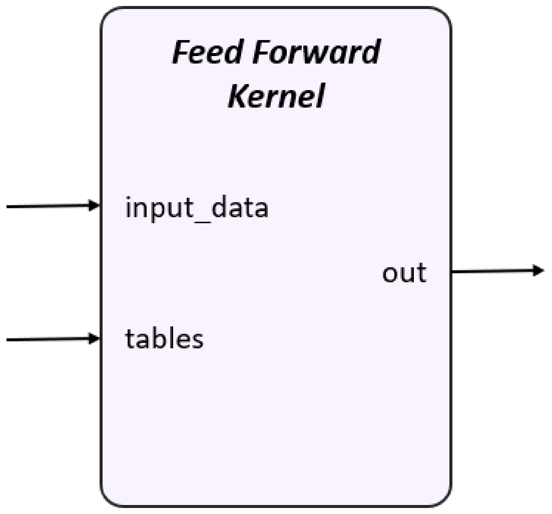
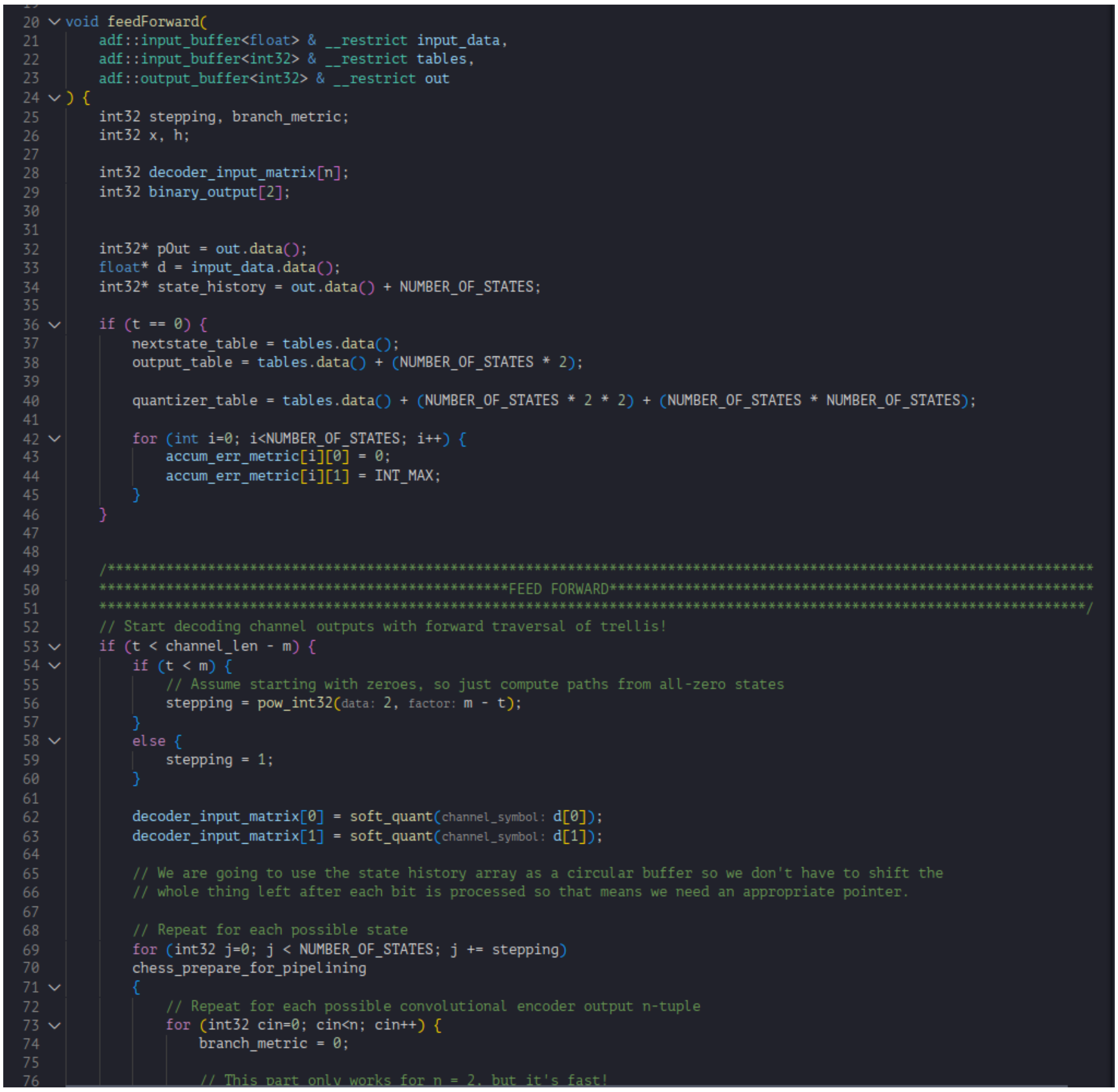


Figure 4.3.2.2.1: Schematic diagram of feed forward kernel.

The feedforward kernel is designed to process input data through a Viterbi decoder by traversing the trellis structure in a forward manner. As shown in Figure 4.3.2.2.1 above, the kernel has 2 input ports, the input_data port where this port will receive the noisy encoded signals, and tables, where this port will receive the initialized metrices from a shared buffer initialized by table construct kernel. Besides, an output port, named "out" is required to send the processed data to traceback kernel.

As aforementioned in Section 4.1.2, this kernel responsible to quantizes the input data in soft decision decoding format and calculates branch metrics for each possible state. The branch metrics will then be used to determine the surviving path with the smallest accumulated error metric. The state history array is updated accordingly to reflect the state transitions.

The kernel uses a circular buffer approach for the state history array to avoid shifting operations, enhancing efficiency. After processing each bit, the accumulated error metrics are updated, and the output buffer is populated with the current metrics. This kernel will continuously active until all the input data are decoded as intended. Figure 4.3.2.2.2 below shows part of the AI Engine implementation for the feed forward kernel function.



```
void feedForward(
    adf::input_buffer<float> & __restrict input_data,
    adf::input_buffer<int32> & __restrict tables,
    adf::output_buffer<int32> & __restrict out
) {
    int32 stepping, branch_metric;
    int32 x, h;

    int32 decoder_input_matrix[n];
    int32 binary_output[2];


    int32* pOut = out.data();
    float* d = input_data.data();
    int32* state_history = out.data() + NUMBER_OF_STATES;

    if (t == 0) {
        nextstate_table = tables.data();
        output_table = tables.data() + (NUMBER_OF_STATES * 2);

        quantizer_table = tables.data() + (NUMBER_OF_STATES * 2 * 2) + (NUMBER_OF_STATES * NUMBER_OF_STATES);

        for (int i=0; i<NUMBER_OF_STATES; i++) {
            accum_err_metric[i][0] = 0;
            accum_err_metric[i][1] = INT_MAX;
        }
    }


    /*************************************************************************************************************************
    *************************************************FEED FORWARD************************************************************
    *************************************************************************************************************************/
    // Start decoding channel outputs with forward traversal of trellis!
    if (t < channel_len - m) {
        if (t < m) {
            // Assume starting with zeroes, so just compute paths from all-zero states
            stepping = pow_int32(data: 2, factor: m - t);
        }
        else {
            stepping = 1;
        }

        decoder_input_matrix[0] = soft_quant(channel_symbol: d[0]);
        decoder_input_matrix[1] = soft_quant(channel_symbol: d[1]);

        // We are going to use the state history array as a circular buffer so we don't have to shift the
        // whole thing left after each bit is processed so that means we need an appropriate pointer.

        // Repeat for each possible state
        for (int32 j=0; j < NUMBER_OF_STATES; j += stepping)
        chess_prepare_for_pipelining
        {
            // Repeat for each possible convolutional encoder output n-tuple
            for (int32 cin=0; cin<n; cin++) {
                branch_metric = 0;

                // This part only works for n = 2, but it's fast!
```

Figure 4.3.2.2.2: The beginning part of AI Engine implementation for the feed forward kernel.

4.3.2.3 Traceback Kernel

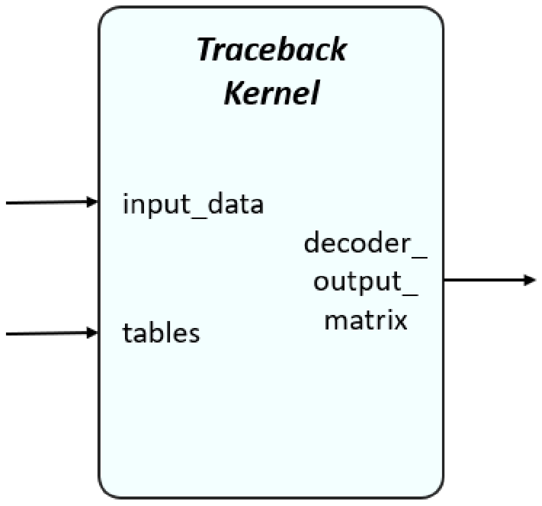
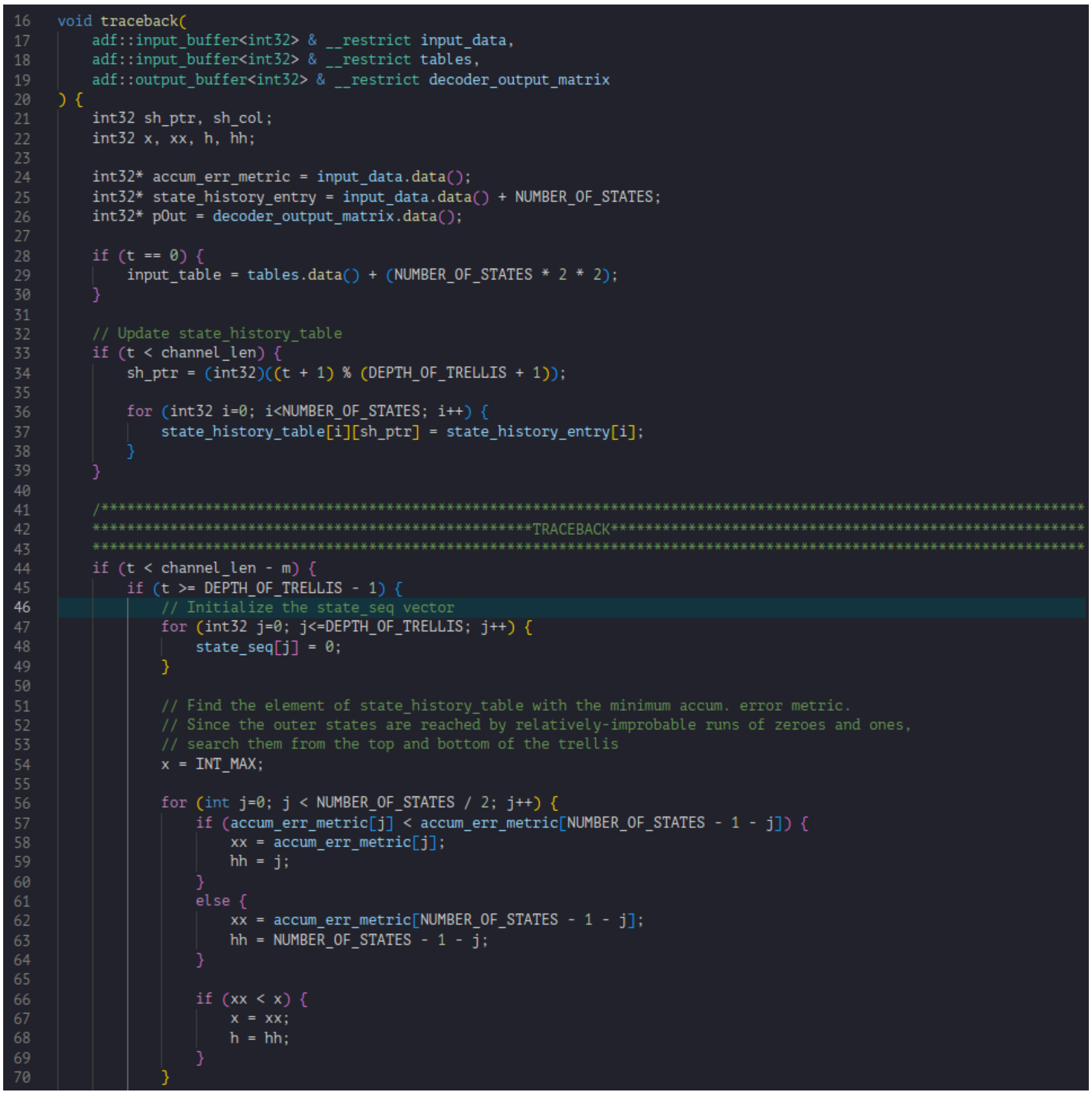


Figure 4.3.2.3.1: Schematic diagram of traceback kernel.

The traceback kernel is designed to determine the most likely state sequence in a Viterbi decoder by working backwards through the trellis structure. As shown in Figure 4.3.2.3.1 above, the traceback kernel has three ports: an “input_data” port where it will receive required data from feedforward kernel, an input port “tables”, where this port will receive the initialized metrices from a shared buffer initialized by table construct kernel, and an output port “decoder_output_matrix”, where the decoded data will output through this port.

Based on the state history entry from feed forward kernel, the traceback kernel will identifies the state with the minimum accumulated error metric and determine the most likely recent state. The kernel then works backwards from the end of the trellis to the oldest state, determining the optimal path based on the received channel symbols. It updates the state sequence vector accordingly. Finally, the kernel identifies the input sequence corresponding to the optimal state sequence and writes it to the output buffer. Kindly refer to Section 4.1.3 for more details of the logic of traceback kernel.

This process ensures that the Viterbi decoder accurately reconstructs the most likely sequence of states, providing reliable decoding of the input data. Figure 4.3.2.3.2 below shows the beginning part of the traceback kernel function.



```
void traceback(
    adf::input_buffer<int32> & __restrict input_data,
    adf::input_buffer<int32> & __restrict tables,
    adf::output_buffer<int32> & __restrict decoder_output_matrix
) {
    int32 sh_ptr, sh_col;
    int32 x, xx, h, hh;

    int32* accum_err_metric = input_data.data();
    int32* state_history_entry = input_data.data() + NUMBER_OF_STATES;
    int32* pOut = decoder_output_matrix.data();

    if (t == 0) {
        input_table = tables.data() + (NUMBER_OF_STATES * 2 * 2);
    }

    // Update state_history_table
    if (t < channel_len) {
        sh_ptr = (int32)((t + 1) % (DEPTH_OF_TRELLIS + 1));

        for (int32 i=0; i<NUMBER_OF_STATES; i++) {
            state_history_table[i][sh_ptr] = state_history_entry[i];
        }
    }

    /*****************************************************************************************************************
    ***************************************************TRACEBACK******************************************************
    ******************************************************************************************************************
    if (t < channel_len - m) {
        if (t >= DEPTH_OF_TRELLIS - 1) {
            // Initialize the state_seq vector
            for (int32 j=0; j<=DEPTH_OF_TRELLIS; j++) {
                state_seq[j] = 0;
            }

            // Find the element of state_history_table with the minimum accum. error metric.
            // Since the outer states are reached by relatively-improbable runs of zeroes and ones,
            // search them from the top and bottom of the trellis
            x = INT_MAX;

            for (int j=0; j < NUMBER_OF_STATES / 2; j++) {
                if (accum_err_metric[j] < accum_err_metric[NUMBER_OF_STATES - 1 - j]) {
                    xx = accum_err_metric[j];
                    hh = j;
                }
                else {
                    xx = accum_err_metric[NUMBER_OF_STATES - 1 - j];
                    hh = NUMBER_OF_STATES - 1 - j;
                }

                if (xx < x) {
                    x = xx;
                    h = hh;
                }
            }
```

Figure 4.3.2.3.2: The beginning part of AI Engine implementation for the traceback kernel.

4.3.3 ADF Graph Design

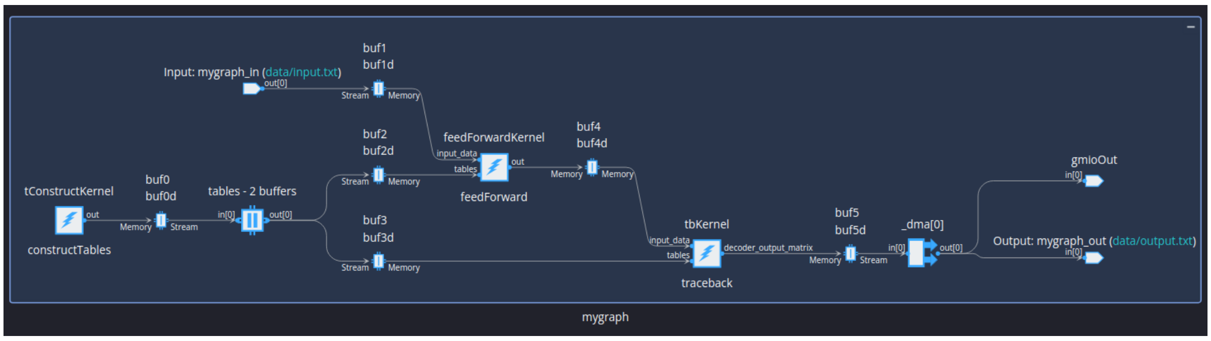


Figure 4.3.3.1: ADF graph that connected all the required components.

An adaptive data flow (ADF) graph represents the final design of AI Engine applications, creating a network of one or more AI Engine kernels interconnected by data streams. The graph, named “mygraph” shown in Figure 4.3.3.1, connects all the AI Engine kernels, internal data buffers, and input/output data streams to efficiently perform Viterbi decoding algorithms.

The graph begins with the tConstructKernel (table construct kernel), which initializes and generates the necessary tables for the decoding process. The output from this kernel is directed to a shared buffer node named “tables”, which stores the data and feeds it to the other kernels. Once the tables are ready, the feedForwardKernel (feed forward kernel) starts processing input data from “data/input.txt.”

The output data from the feed forward kernel is then sent to the tbKernel (traceback kernel), which performs the final layer of the Viterbi decoding algorithm, known as traceback processing. The decoded data is directed along two paths: gmioOut, which prints the results in the terminal for debugging purposes, and “data/output.txt,” which records the decoded data along with its simulation time.

This design maximizes the utilization of the AI Engine's parallel computing capabilities, allowing the feed forward processing to be executed concurrently with the traceback processing, as illustrated in Figure 4.3.3.2.

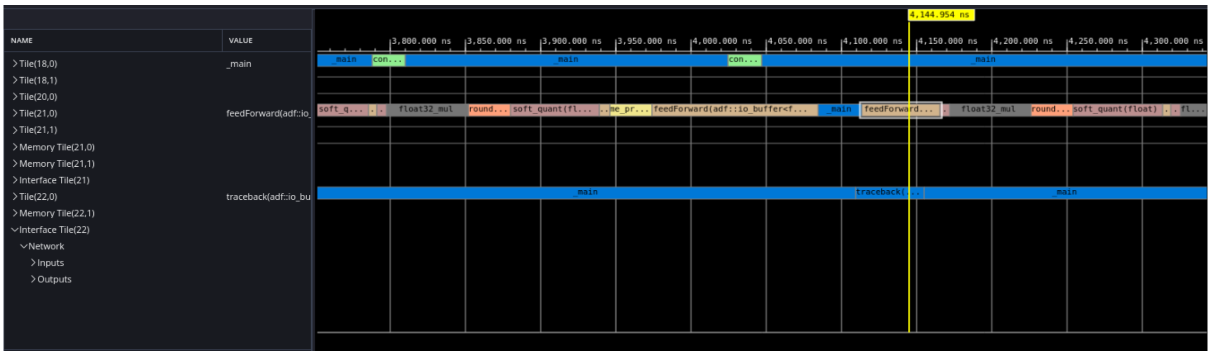


Figure 4.3.3.2: Trace signals of the design, where the second execution of feed forward function on Tile(21, 0) run simultaneously with first execution of traceback function on Tile(22, 0).

4.3.4 System Integration

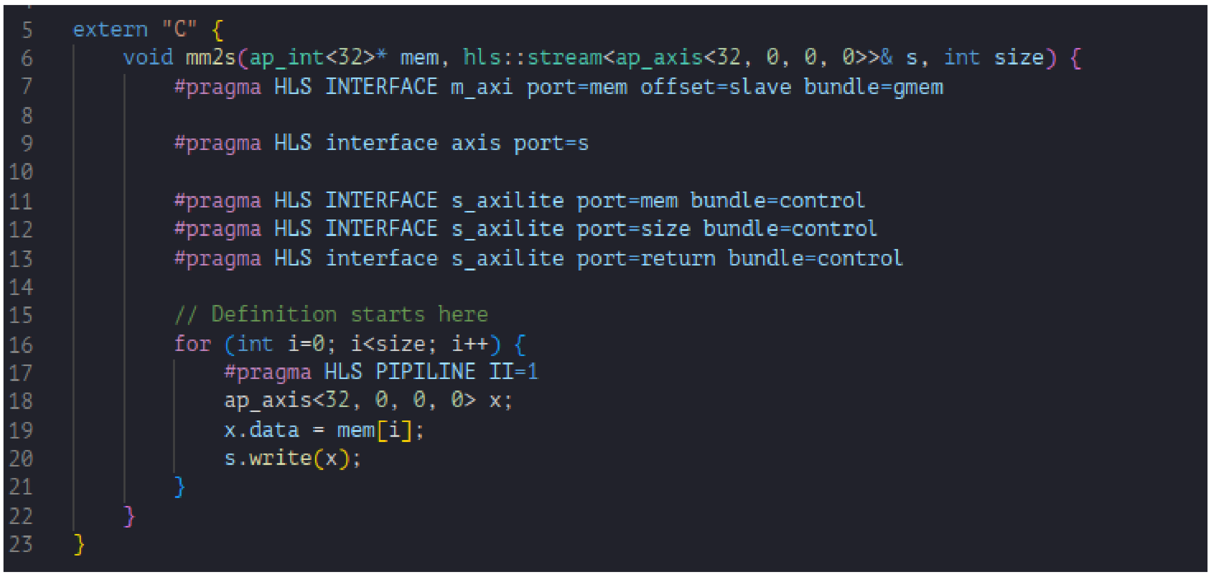
To enable communication between the AI Engine application and the VEK280 board, several programmable logic (PL) kernels need to be integrated into the system. AMD recommends using the PLIO port attributes, which represent external stream connections that cross the AI Engine-ML-PL boundary.

In this project, two high-level synthesis (HLS) data mover kernels, named memory-to-system (mm2s) kernel and system-to-memory (s2mm) kernel, were developed and connected to the PLIO ports of AI Engine. These kernels act as bridges between the memory NOC module and the AXI4-Stream interface, facilitating the input and output of data to and from the AI Engine. The kernels function as follows:

- The mm2s kernel reads data from memory and inputs it into the AI Engine array.

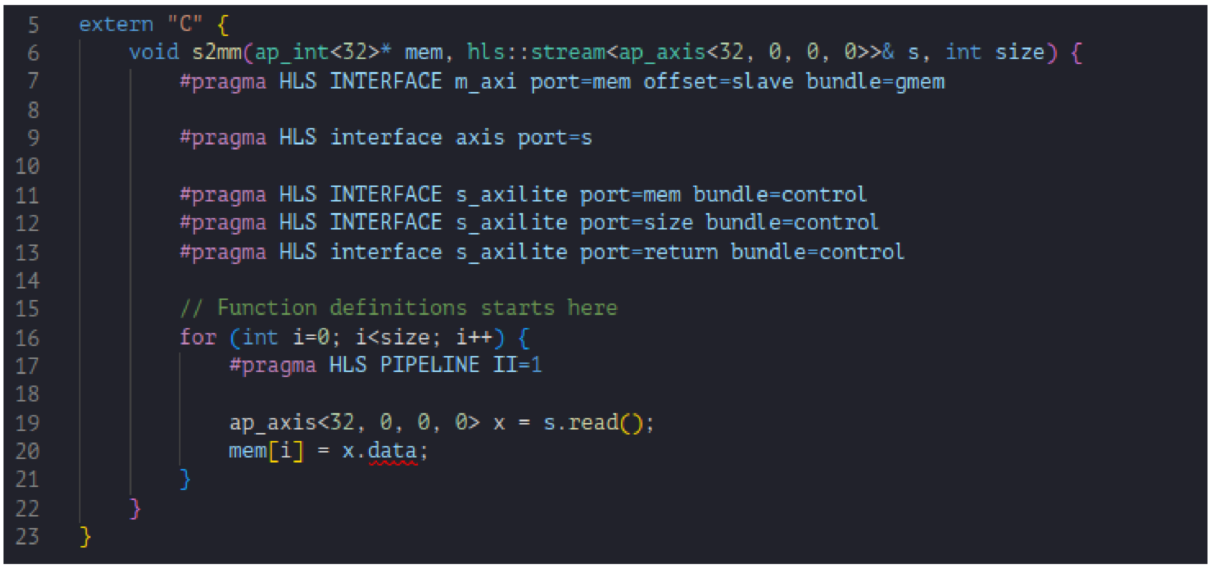
- The s2mm kernel receives output data from the AI Engine array and writes it back to memory.

Figure 4.3.4.1 and Figure 4.3.4.2 below show the implementation of mm2s kernel and s2mm kernel respectively.



```
extern "C" {
    void mm2s(ap_int<32>* mem, hls::stream<ap_axis<32, 0, 0, 0>>& s, int size) {
        #pragma HLS INTERFACE m_axi port=mem offset=slave bundle=gmem

        #pragma HLS interface axis port=s

        #pragma HLS INTERFACE s_axilite port=mem bundle=control
        #pragma HLS INTERFACE s_axilite port=size bundle=control
        #pragma HLS interface s_axilite port=return bundle=control

        // Definition starts here
        for (int i=0; i<size; i++) {
            #pragma HLS PIPILINE II=1
            ap_axis<32, 0, 0, 0> x;
            x.data = mem[i];
            s.write(x);
        }
    }
}
```

Figure 4.3.4.1: Implementation of mm2s kernel.



```
extern "C" {
    void s2mm(ap_int<32>* mem, hls::stream<ap_axis<32, 0, 0, 0>>& s, int size) {
        #pragma HLS INTERFACE m_axi port=mem offset=slave bundle=gmem

        #pragma HLS interface axis port=s

        #pragma HLS INTERFACE s_axilite port=mem bundle=control
        #pragma HLS INTERFACE s_axilite port=size bundle=control
        #pragma HLS interface s_axilite port=return bundle=control

        // Function definitions starts here
        for (int i=0; i<size; i++) {
            #pragma HLS PIPELINE II=1

            ap_axis<32, 0, 0, 0> x = s.read();
            mem[i] = x.data;
        }
    }
}
```

Figure 4.3.4.2: Implementation of s2mm kernel.

Although the HLS kernels have been developed, the project could not proceed with the hardware emulation due to insufficient computer hardware resources. The recommended RAM memory for this task is 80GB, which exceeds the capacity of my current system. Consequently, the hardware emulation processes could not be carried out at this time.

4.3.5 Test Plans

Like the C++ implementation, the test plans for the AI Engine implementation are divided into two main groups. The first group focuses on verifying the functional correctness of the design. This involves ensuring that the AI Engine implementation of the Viterbi decoder operates as intended, producing accurate results under various conditions. The second group tests the decoder with large volumes of data, comparing the results against those obtained from the C++ implementation. This comparison evaluates the performance and accuracy of the AI Engine implementation, highlighting any improvements or differences.

Due to insufficient computer hardware resources, the project could not proceed with the VEK280 board hardware emulation. As consequences, all data will be collected from AI Engine application simulations. These simulations will provide the necessary insights and performance metrics to assess the AI Engine implementation's effectiveness without the need for hardware emulation.

Table 4.3.5.1 below shows the Group 1 test plan, which focuses on verifying the functional correctness of the design. The design will be tested using the same data set as the C++ implementations, ensuring that the expected decoded codes and Bit Error Rate (BER) match those obtained from the C++ implementation. By ensuring that these values are identical, we can conclude that the decoder is functioning correctly and is free from logical errors.

Test Number	**Test Objectives**	**Constraint Length, K**	**Es/No Ratio**	**Message Length**	**Target**
15	Verify the decoder under various constraint lengths	3	1.0	1000	BER AI Engine = BER C++
16		5	1.0	1000	BER AI Engine = BER C++
17		7	1.0	1000	BER AI Engine = BER C++
18		9	1.0	1000	BER AI Engine = BER C++
19	Verify decoder under various Es/No ratio.	5	0.5	1000	BER AI Engine = BER C++
20		5	1.0	1000	BER AI Engine = BER C++
21		5	1.5	1000	BER AI Engine = BER C++
22		5	2.0	1000	BER AI Engine = BER C++
23		5	2.5	1000	BER AI Engine = BER C++
24		5	3.0	1000	BER AI Engine = BER C++

Table 4.3.5.1: Group 1 test plan for functionalities verification.

Following this, Table 4.3.5.2 details the Group 2 test plan. This plan compares both the Bit Error Rate (BER) and transfer bandwidth with the results obtained from the C++ implementation. Consequently, this test plan does not specify an intended target but instead focuses on testing large volumes of messages. Additionally, the Es/No ratio remains constant for Group 2, as it does not influence the speed of the Viterbi decoding algorithm. Please note that the test numbers remain the same for both the C++ and AI Engine implementations, as they are subjected to identical tests.

Test Number	Constraint Length, K	Es/No Ratio	Message Length
11	3	0.2	10000
12	5	0.2	10000
13	7	0.2	10000
14	9	0.2	10000

Table 4.2.3.3: Group 2 test plan for comparative performance analysis.

Chapter 5: Comparative Analysis

This chapter presents a comparative analysis of the Viterbi decoder implementations in C++ and AI Engine. The tests are developed based on the test plans outlined in Sections 4.2.3 and 4.3.5. The analysis is divided into three main sections.

First, Section 5.1 discusses the Group 1 tests of the C++ implementation, focusing on verifying the functional correctness of the C++ implementation. This involves ensuring that the decoder operates as intended, producing accurate results under various conditions.

Next, Section 5.2 examine the Group 1 tests of the AI Engine implementation. Similar to the C++ tests, these tests verify the functional correctness of the AI Engine design, ensuring it produces accurate and reliable results.

Finally, Section 5.3 compare the results obtained from the Group 2 tests of the C++ implementation and the AI Engine implementation. This comparison evaluates the performance and accuracy of both implementations by analysing metrics such as Bit Error Rate (BER) and transfer bandwidth. The analysis highlights any improvements or differences, providing insights into the effectiveness of each implementation.

5.1 Results of Group 1 Tests for C++ Implementation

Test Number	Constraint Length, K	Es/No Ratio	Message Length	Actual BER	Target	Status
1	3	1.0	1000	0.001	BER < 0.01	Passed
2	5	1.0	1000	0.000	BER < 0.01	Passed
3	7	1.0	1000	0.000	BER < 0.005	Passed

4	9	1.0	1000	0.000	BER < 0.005	Passed
5	5	0.5	1000	0.005	BER < 0.01	Passed
6	5	1.0	1000	0.000	BER < 0.01	Passed
7	5	1.5	1000	0.000	BER < 0.01	Passed
8	5	2.0	1000	0.000	BER < 0.005	Passed
9	5	2.5	1000	0.000	BER < 0.005	Passed
10	5	3.0	1000	0.000	BER < 0.005	Passed

*Table 5.1.1: Results of group 1 tests for C++ implementation on Intel i9-12900H CPU, with the trellis depth = K * 5*

Overall, the C++ implementation demonstrates robust performance across a range of constraint lengths (K) from 3 to 9 and Es/No ratios from 0.5 to 3.0, consistently achieving a low Bit Error Rate (BER) well below the specified targets. Each test, conducted with a message length of 1000, resulted in an actual BER significantly lower than the target BER, indicating high reliability and effectiveness in minimizing errors.

5.2 Results of Group 1 Tests for AI Engine Implementation

Test Number	**Constraint Length, K**	**Es/No Ratio**	**Message Length**	**Actual BER**	**Target**	**Status**
15	3	1.0	1000	0.001	BER AI Engine = BER C++	Passed
16	5	1.0	1000	0.000	BER AI Engine = BER C++	Passed
17	7	1.0	1000	0.000	BER AI Engine = BER C++	Passed

18	9	1.0	1000	0.000	BER AI Engine = BER C++	Passed
19	5	0.5	1000	0.005	BER AI Engine = BER C++	Passed
20	5	1.0	1000	0.000	BER AI Engine = BER C++	Passed
21	5	1.5	1000	0.000	BER AI Engine = BER C++	Passed
22	5	2.0	1000	0.000	BER AI Engine = BER C++	Passed
23	5	2.5	1000	0.000	BER AI Engine = BER C++	Passed
24	5	3.0	1000	0.000	BER AI Engine = BER C++	Passed

*Table 5.2.1: Results of group 1 tests for AI Engine implementation with VEK280 image, with the trellis depth = K * 5*

The data from Table 5.2.1 illustrates the performance of the AI Engine implementation with the VEK280 image across various test conditions. In each test, the actual Bit Error Rate (BER) achieved by the AI Engine matched the BER of the C++ implementation, consistently meeting the target BER. Notably, the AI Engine achieves a BER of 0.000 in most tests, indicating no errors, except for the test with K = 5 and Es/No ratio of 0.5, where the BER is 0.005. Despite this, the AI Engine demonstrated high reliability and robustness, maintaining low BER across different levels of complexity and signal-to-noise conditions, and successfully passing all the tests.

5.3 Results of Group 2 Tests for C++ and AI Engine Implementation

Test Number	K	Es/No	C++ BER	AI Engine BER	C++ Mbps	AI Engine Mbps	Speed up (x)
11	3	0.2	3.4×10^{-3}	3.4×10^{-3}	3.1153	54.2896	17.43
12	5	0.2	9.0×10^{-4}	9.0×10^{-4}	0.8703	27.2942	31.36
13	7	0.2	3.0×10^{-4}	3.0×10^{-4}	0.2674	7.6386	28.57
14	9	0.2	0	0	0.0621	2.0627	33.22

*Table 5.3.1: Results of group 2 tests for C++ implementation on Intel i9-12900H CPU and AI Engine implementation with VEK280 image, with the trellis depth = K * 5*

The data in the table highlights the performance comparison between the C++ implementation and the AI Engine across various constraint lengths (K). For K values of 3, 5, 7 and 9, both implementations achieved the exact same Bit Error Rates (BER), demonstrating the AI Engine's accuracy. In terms of throughput, the AI Engine significantly outperformed the C++ implementation, achieving speed-ups ranging from 17.43x to 33.22x, with the average of 27.65x. This substantial increase in processing speed underscores the efficiency of the AI Engine, where it maintains high accuracy and reliability while delivering much faster performance.

Chapter 6: Conclusions and Future Work

6.1 Conclusions

In conclusion, four out of five objectives have been fully met in this project, while the second objective and fourth objective have been partially completed.

The first objective, which is to analyse the Viterbi algorithm for parallelization purposes, has been achieved and became the groundwork for implementing the Viterbi algorithm on AI Engine with optimal performance.

The second objective, implementing the Viterbi decoding algorithm on the AI Engine using parallel programming techniques, has been accomplished for constraint lengths K = 3, 5, 7 and 9, which fully met the intended range of constraint lengths.

The third objective, which involved optimizing the Viterbi decoding algorithm for speed and memory utilization, has been successfully achieved. This process included refining the algorithm to eliminate inefficiencies and bottlenecks, and fine-tuning to ensure that the algorithm maintained a balance between speed and memory efficiency.

The fourth objective, running hardware emulation on the VEK280 board through HACC, has been partially completed. The interfaces to connect the AI Engine to the VEK280 board have been developed. However, the process could not be fully completed due to the high-end PC requirements of Vitis, which recommends 80GB of RAM to build the hardware image.

The final objective, simulating the Viterbi decoding algorithm using both the AI Engine and a CPU for comparative performance analysis, has been carried out successfully, with all data tabulated. The AI Engine implementation demonstrated high accuracy and achieved an average speed-up of 27.65x compared to the C++ decoder running on an Intel i9-12900H processor.

6.2 Future Work

To further advance this project, future work should focus on several key areas. Firstly, utilizing a high-end PC with more than 80GB of RAM will be essential to continuing the hardware emulation process, overcoming the current memory limitations. Next, ongoing efforts to discover additional optimization techniques for the Viterbi decoding algorithm itself will be crucial. This includes exploring new parallelization strategies, refining memory management, and leveraging advancements in hardware capabilities to further enhance both speed and efficiency. These steps will collectively contribute to the robustness and scalability of the Viterbi decoding implementation.

References

[1] "Viterbi decoder." Accessed: Aug. 27, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Viterbi_decoder

[2] F. Chip, "A Tutorial on Convolutional Coding with Viterbi Decoding," Spectrum Application. Accessed: Jun. 26, 2024. [Online]. Available: https://linas.org/mirrors/home.netcom.com/2002.09.18/~chip.f/viterbi/algrthms2.html

[3] H.-L. Lou, "Implementing the Viterbi Algorithm," 1995.

[4] G. D. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973, doi: 10.1109/PROC.1973.9030.

[5] D. Akkidas, "VITERBI ALGORITHM," 2015. [Online]. Available: https://www.researchgate.net/publication/305402054

[6] MIT, *Viterbi Decoding of Convolutional Codes*. Accessed: Jul. 04, 2024. [Online]. Available: https://web.mit.edu/6.02/www/s2012/handouts/8.pdf

[7] R. Li, Y. Dou, and D. Zou, "Efficient parallel implementation of three-point viterbi decoding algorithm on CPU, GPU, and FPGA," *Concurr Comput*, vol. 26, no. 3, pp. 821–840, Mar. 2014, doi: 10.1002/cpe.3093.

[8] D. Vaithiyanathan, J. Nargis, and R. Seshasayanan, "High performance ACS for Viterbi decoder using pipeline T-Algorithm," *Alexandria Engineering Journal*, vol. 54, no. 3, pp. 447–455, Sep. 2015, doi: 10.1016/j.aej.2015.04.007.

[9] S. K. Monfared, O. Hajihassani, V. Mohsseni, D. Rahmati, and S. Gorgin, "A High-throughput Parallel Viterbi Algorithm via Bitslicing," *ACM Transactions on Parallel Computing*, vol. 8, no. 4, Dec. 2021, doi: 10.1145/3470642.

[10] AMD, "AMD AI Engine Technology." Accessed: Aug. 22, 2024. [Online]. Available: https://www.xilinx.com/products/technology/ai-engine.html#overview

[11] AMD, "AI Inference with Versal^{TM} AI Core Series," 2023. [Online]. Available: www.xilinx.com/versal-ai-core

[12] G. Archana, “Vitis Tutorial - AI Engine Development.” Accessed: Jun. 28, 2024. [Online]. Available: https://github.com/Xilinx/Vitis-Tutorials/tree/2024.1/AI_Engine_Development/AIE

[13] AMD, “AI Engine Tools and Flows User Guide UG1076 (v2023.2).”

[14] AMD, “AMD Versal^{TM} AI Edge Series VEK280 Evaluation Kit.” Accessed: Jun. 28, 2024. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/vek280.html#information

[15] AMD, “Versal Evaluation Platform”, Accessed: Jun. 28, 2024. [Online]. Available: https://account.amd.com/en/forms/downloads/design-license.html?cid=3aec3cac-2682-43f6-a6b3-f6063b5a3985&filename=xtp760-vek280-schematic.zip

[16] AMD, “Heterogeneous Accelerated Compute Clusters.” Accessed: Aug. 22, 2024. [Online]. Available: https://www.amd-haccs.io/index.html

[17] AMD, “The Vitis Software Platform Development Environment.” Accessed: Aug. 13, 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis.html

[18] AMD, “AMD Vivado Design Suite.” Accessed: Aug. 13, 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html