KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

CAMPUS GROUP T LEUVEN

High-Throughput Open Source Viterbi Decoder for OpenWiFi

Yingshuo XI

Baiming ZHANG

Supervisor(s): Josep Balasch
Co-supervisor(s): Xianjun Jiao

Master Thesis submitted to obtain the degree of Master of Science in Engineering Technology: Electronics Engineering

Academic Year 2021 - 2022

KU LEUVEN

Faculty of Engineering Technology Campus GROUP T Leuven — Master Thesis submitted to obtain the degree of Master of Science in Engineering Technology: Electronics Engineering
Academic Year 2021 - 2022

High-Throughput Open Source Viterbi Decoder for OpenWiFi

Zhang Baiming, Xi Yingshuo

Master in Electronics Engineering, Faculty of Engineering Technology, Campus GROUP T Leuven, Andreas Vesaliusstraat 13, 3000 Leuven, Belgium

Supervisor(s): Josep Balasch

Electronics Engineering, Faculty of Engineering Technology, Campus GROUP T Leuven, Andreas Vesaliusstraat 13, 3000 Leuven, Belgium, <josep.balasch@kuleuven.be>

Co-supervisor(s): Xianjun Jiao

IDLab, Ghent University, iGent Toren, Technologiepark Zwijnaarde 126, 9052 Gent, Belgium , <Xianjun.Jiao@UGent.be>

ABSTRACT

OpenWiFi is an open source software-defined radio (SDR) implementation on the IEEE 802.11 (Wi-Fi) standard. In the receiver module, it used Xilinx's Viterbi decoder v7.0 to decode convolutional codes. However, due to license limitations, the test board had to be reset every 1-2 hours, which caused much inconvenience for wireless network experiments. The paper aimed to port an open source Viterbi decoder to OpenWiFi and make it functionally equivalent to the proprietary Xilinx one. The performance of the ported Viterbi decoder is evaluated by simulation and synthesis on the Xilinx Vivado platform. In addition, noise resilience was verified by testing a series of data files of different protocols, bit rates, and lengths compared to the proprietary Xilinx IP core. The maximum throughput of the ported Viterbi decoder reached 100 Mbps with a processing delay of 213 clock cycles (2.13 μs) at a clock frequency of 100 MHz. Finally, the modified OpenWiFi was loaded on the target FPGA for validation. Currently, the onboard test is not fully functional, which is the next step in future work.

Keywords: Viterbi decoder, Wi-Fi, FPGA, Open source

1 INTRODUCTION

1.1 Nature and scope of the problem

Wireless local area networks (WLANs) provide wireless connections between electronic devices in modern digital communication systems. One of the most common WLAN technologies now in use is Wi-Fi. IEEE 802.11 is an essential technical standard that specifies the protocols for implementing WLANs. The standard is based on orthogonal frequency division multiplexing (OFDM), which modulates digital data over multiple carrier frequencies Chang et al. (2011). In addition, to control data errors on noisy communication channels, a forward error correction (FEC) coding is applied to their transmitted bitstreams. This FEC coding is based on industry-standard $\frac{1}{2}$-rate convolutional codes. The efficient decoding of convolutional codes is vital when designing wireless LAN receivers. Moreover, the best solution for decoding convolutional codes is the Viterbi algorithm.

OpenWIFI is an open source implementation of a Wi-Fi chip that supports the IEEE802.11a/g/n standard Jiao et al. (2020). Figure 1.1 shows the block diagram of the OFDM receiver module of OpenWiFi. This open source hardware design provides developers with additional free options to test new ideas more efficiently than ever before, without cracking commercial chips or buying expensive reference designs.

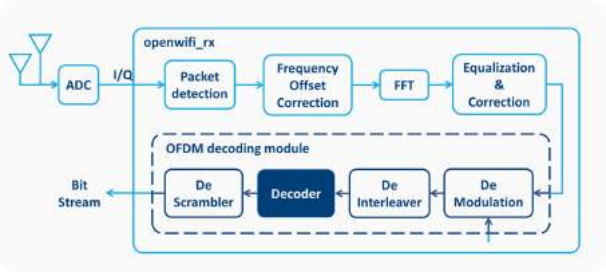


Figure 1.1: OFDM receiver in openWIFI Ganeshkumar and Rangasamy (2013)

Currently, OpenWIFI uses Xilinx's Viterbi decoder v7.0 in the decoder block under the OFDM decoding module. However, due to the limitation of the Xilinx license, the Viterbi decoder stops every 1-2 hours of running, which makes long-term operation and stress testing impossible. To overcome it, it is necessary to use an openly available version of the decoder and adapt it to make it compatible with the rest of the OpenWiFi system. Although the Xilinx datasheet releases the basic architecture of the Viterbi decoder, the detailed algorithm and parameters are still hidden, which affect the decoder's noise resilience and processing delay. Therefore it is necessary to compare the performance of the ported decoder with the proprietary Xilinx one in evaluation.

1.2 Objectives and challenges

1. Summarize the performance and functional requirements for the Viterbi decoder of the OpenWiFi project.

2. Select the open source Viterbi decoder with the most proper algorithm and architecture, according to the trade-off with different performance metrics.

3. Test the processing delay, maximum throughput, area/energy consumption, and noise resilience of the ported Viterbi decoder through simulation and synthesis.

4. Compare the performance of the ported Viterbi decoder with the proprietary Xilinx one.

5. Examine its compatibility with the rest of OpenWiFi by integrating it into OpenWiFi and simulation.

6. Verify the functionality of the modified OpenWiFi with the ported Viterbi decoder on an FPGA board.

1.3 Outline of the paper

This paper is organized as follows. Chapter 2 introduces the background knowledge that helps the reader understand this paper. In Chapter 3, a requirements analysis is performed, listing the metrics that need to be satisfied by the Viterbi decoder. Then, Chapter 4, from the algorithmic point of view, describes the architecture with four components: BMU, ACSU, SMU, and reorder unit, and explains the selection of algorithms for each component. Next, in Chapter 5, the implementation details of the RTL-level design are shown. Chapter 7 shows the simulation and synthesis results of the Viterbi decoder. Finally, in Chapter 7, the evaluation results are discussed and compared with the proprietary Xilinx one. In conclusion, the ported open source Viterbi decoder meets the performance requirements and is well compatible with the rest of the OpenWIFI in simulation and synthesis. This is a step forward in addressing the limitations imposed by the Xilinx license.

2 BACKGROUND

2.1 Communication systems

In a communication system, the communication medium that carries the information bits, called the channel. It affects the communication signal in three main ways Baptista (2021):

1. Attenuation of the signal by the channel.

2. The noise in the channel, which is superimposed on the signal, changes the amplitude, phase, and frequency of the signal, causing demodulation errors.

3. Multipath effect: the superposition effect caused by the reflection, refraction, and propagation of the signal along different paths during transmission.

For analysis and experimentation, there are many mathematical channel models.

2.2 Channel Models : Additive White Gaussian Noise (AWGN) Channel

The additive white Gaussian noise (AWGN) channel is one of the most common mathematical models of a communication channel used to simulate the channel between a transmitter and a receiver. It assumes that the channel is affected by Gaussian noise. Gaussian noise is a linearly increasing broadband noise with a constant spectral density and Gaussian distribution amplitude.

As mentioned in section 2.1, the signal is combined with errors when passing through the channel. Channel encoding improves the performance against error by inserting some redundant code elements at the transmitter side and detecting and correcting the errors at the receiver side. There are two types of channel encoding Baptista (2021):

1. Automatic repeat request (ARQ)

2. Forward error correction (FEC)

2.3 FEC (forward error correction)

Forward error correction (FEC) is used to control signal errors over unreliable or noisy communication channels. According to some algorithms, it generates new code elements from the original codeword at the transmitter side, which is called redundancy. Those redundant bits are then added to the transmitted message. After the code word reaches the receiver side, it detects and corrects the errors. Even if some code elements are corrupted by noise, the correct transmitted signal can still be extracted from the other uncorrupted received code elements Mizuochi (2006).

There are two main classes of ECC codes: block codes and convolutional codes. Block codes work on fixed-size bits blocks with a predetermined size. Convolutional codes work on a bitstream of arbitrary length.

2.4 Convolution codes

2.4.1 Basic concepts

Code words and code elements: The continuous bitstream should be coded at the transmitter side before being emitted to the channel. One approach is to group it and assign redundant information to each group. For instance, a data of *k* bits is coded to form an *n*-bit length stream $(n > k)$. This *n*-bit stream is defined as a code word, and each internal bit is called a code element.

Code rate: The proportion of the useful (non-redundant) data-stream, $R = \frac{k}{n}$.

Code distance: The number of different code elements in the corresponding positions between two code words. For example, the codewords ”11000011” and ”11010010” differ in two positions (No. 4 and No. 8), so their code distance is 2.

In telecommunication, a convolutional code generates redundancy elements via the sliding application of generator polynomials to a bitstream. Convolutional code encoding is expressed in the format (n, k, K), which is defined as:

n: number of code elements produced by the encoder

k: number of code elements coming into the encoder

K: constraint length of the code, which is the length of the shift register plus one input bit

Furthermore, there are three methods to describe a convolutional code:

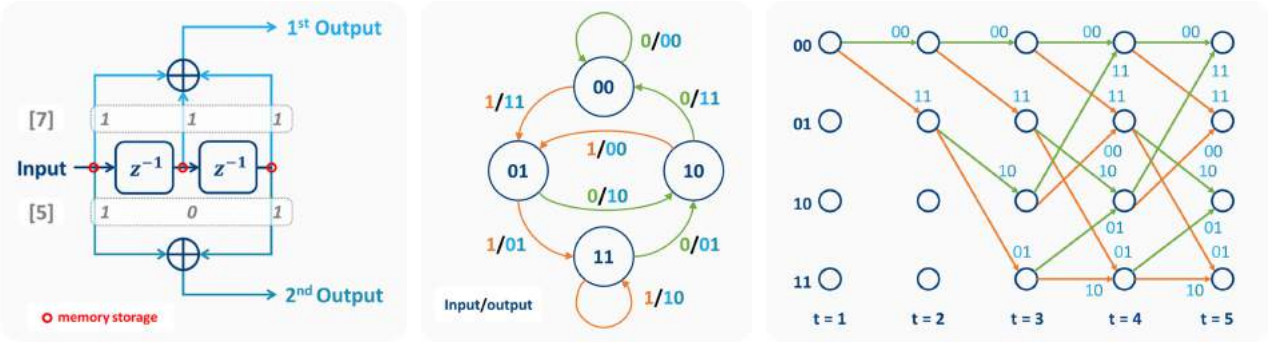
1. Polynomials representation

2. State transition diagram

3. Trellis diagram

2.4.2 Polynomials representation

The generator polynomials of a convolutional encoder describe the connections among shift registers and modulo-2 adders. If the encoder diagram has *k* inputs and *n* outputs, then the code generator matrix is a *k*-by-*n* matrix. The element (row *i*, column *j*) indicates how the *ith* input contributes to the *jth* output. Build a binary number representation by placing ’1’ in each spot where a connection line from the shift register feeds into the adder and ’0’ elsewhere. The leftmost spot in the binary number represents the current input, while the rightmost spot represents the oldest input remaining in the shift register (Gazi, 2020, pp. 283-285).



(a) Polynomials representation (b) State transition diagram (c) Trellis diagram

Figure 2.1: (2, 1, 3) Convolutional Encoder

For example, figure 2.1(a) illustrates a simple (2, 1, 3) encoder with one input, two outputs, and two shift registers. The binary numbers corresponding to the upper and lower adders are ”111” and ”101”, respectively. These binary numbers are equivalent to the octal numbers 7 and 5, respectively. Thus the generator polynomial matrix is [7 5].

In IEEE 802.11 description, the decoder needs to match the convolutional encoding polynomial [171 133], as shown in Figure 2.2, with the constraint length of 7 and the code rate of $\frac{1}{2}$ (Sun and Ding, 2012, p. 126). Its input is two 3-bit streams with de-puncturing (soft decision), and its output is a 1-bit stream.

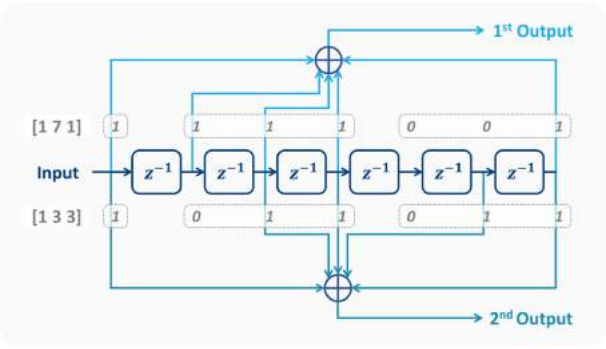


Figure 2.2: Convolutional Encoding with the polynomial [171, 133]

2.4.3 State transition diagram

The convolutional encoder is a Mealy finite state machine since its output depends on the input bit and its memory contents. It can be described by a state transition diagram where the states are the contents of its memory. In figure 2.1(b), states are represented as nodes, and state transitions are represented as arrows. Further, the encoder selects to enter two possible states depending on the current input (1 or 0) (Gazi, 2020, pp. 294-299).

2.4.4 Trellis diagram

The convolutional code can also be described as a trellis diagram by considering the transitions between states as a function of time Plosila (2005). In Figure 2.1(c), the nodes represent the memory contents of the encoder. Assuming that at time $t = 0$, starting from the initial state (00), two possible branches (to the following states) are generated according to the input. The green line represents the input as ’0’, and the orange line represents the input as ’1’. Therefore, the different paths on the trellis diagram represent all possible transmission sequences (Gazi, 2020, pp. 300-304).

2.5 Decoding convolutional codes

Assuming convolutional coding is applied to the transmitted signal in a channel with AWGN. Then decoding is performed at the receiver side. Two algorithms are most commonly used for convolutional code decoding:

- Viterbi decoding
- Sequential decoding

The Viterbi algorithm is commonly used for convolutional codes with relatively small constraint length. Codes with relatively long constraints are more practically decoded with sequential decoding algorithms. Unlike Viterbi decoding, the complexity of sequential decoding increases only slightly with the constraint length, so it allows the use of codes with long constraint lengths Banerjee et al. (2022). However, Viterbi decoding is a well-suitable hardware implementation option because it has a highly parallelizable structure and a fixed decoding time.

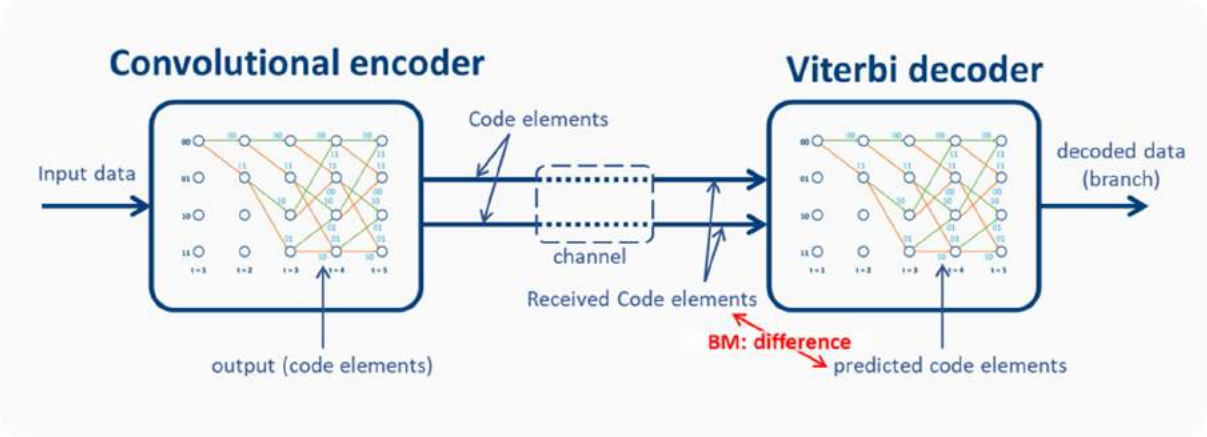


Figure 2.3: Data flow in encoder and decoder

2.6 Viterbi Methodology

The general idea of all convolutional decoders is to reproduce the encoded path. An exhaustive decoding algorithm would compare the received data with every possible path along the trellis and output the most probable path, but it is not feasible in most cases because of the enormous computational cost. The Viterbi algorithm uses the trellis structure to estimate the process. It only tracks paths that appear with maximum likelihood, which is why it's also called the maximum likelihood sequence estimator Forney (1972). Several key concepts will be defined in advance:

Branching metric (BM): Each branch has a branching metric that indicates how much the predicted codeword differs from the received codeword.

Path metric (PM): Each trellis path has a path metric, the sum of all BMs of the previous path. The smaller the PM is, the closer it is to the expected transmission sequence Abdallah and Shanbhag (2009).

Survivor path: The previous only possible path for one node,determined by the lower PM. Finally, it finds the most probable receiving sequence by finding the path based on the final minimum PM.

As shown in Figure 2.3, the convolutional encoder and Viterbi decoder utilize the trellis diagram differently. In the encoder, the only path is determined by the input, and the code elements are sent to the output. On the contrary, the Viterbi decoder reproduces the encoding process. The survivor paths are determined by PM, which is the sum of BM (the difference between received code elements and predicted code elements). The output of the decoder is the branch value.

According to different strategies, the decoding processes are divided into hard decision and soft decision., which are different in BM:

Hard decision: Hard decision directly determines the received signal waveform and outputs '0' or '1' according to its decision threshold and utilizes the Hamming distance as the metric. The Hamming distance indicates the number of different code elements in the corresponding position for two code words with the same length.

Soft decision: Soft decision quantizes the output waveform of the demodulator at multiple levels, which allows the received symbols to contain reliable information, and utilizes the squared Euclidean distance as the metric. For instance, in a 3-bit encoding, this reliability information can be encoded, as shown in Table 2.1. Euclidean distance means the true distance between two points in *n*-dimensional space, formulated as equation 2.1.

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{2.1}$$

p and *q* are two points in Euclidean n-space, q_i and p_i are Euclidean vectors starting from the origin of the space (initial points), and *n* is the dimensionality of space.

Probability	Hard decision value	Soft decision value
Strongest 1	1	111
Relatively strong 1	1	110
Relatively weak 1	1	101
Weakest 1	1	100
Weakest 0	0	000
Relatively weak 0	0	001
Relatively strong 0	0	010
Strongest 0	0	011

Table 2.1: Relationship between Hard decision value and Soft decision value for 3-bit inputs

2.7 Viterbi Algorithm working

Figure 2.4 shows how the Viterbi algorithm works. The example from Section 2.4 is utilized to illustrate the process. The number next to the lines means the calculated branch metrics. $Tn(n = 1,2,3,4)$ means the path metrics of the corresponding state. Assume the output is a data stream of five bits

Figure 2.4(a) shows two possible branches from the initial state, $00 \longrightarrow 00$ and $00 \longrightarrow 01$. Similarly, as presented in figures 2.4(b) and 2.4(c), every following state will generate two different branches, in which green refers to '0' and orange refers to '1'. Then, when $t = 4$, two paths diverging into one state in figure 2.4(c), it needs to eliminate the one with larger accumulated path metrics. The survivor paths of $t = 4$ are shown in figure 2.4(d). It is observed that there is only one branch left between $t = 1$ and $t = 2$. In other words, the correct state transmission is $00 \longrightarrow 01$. Therefore, the first output should be '1'.

Following the same steps, figures 2.4(e) to 2.4(h) show the survivor paths at $t = 5$ and $t = 6$ by generating the branches and eliminating the paths with larger accumulated metrics. Lastly, the most probable receiving sequence will be selected with the lowest path metrics (figure 2.4(i)). The output is read from the state transmission (colors), "11011".

Computational complexity: for a bitstream of *(n, k, K)* convolutional codes of length *N*. The decoder processes one bit per clock. There are 2^{K-1} states at each moment on the trellis graph, and each state takes twice comparisons, so 2^K times comparisons in total. Thus the computational complexity of Viterbi decoding *N* bits is $O(N \times 2^K)$.

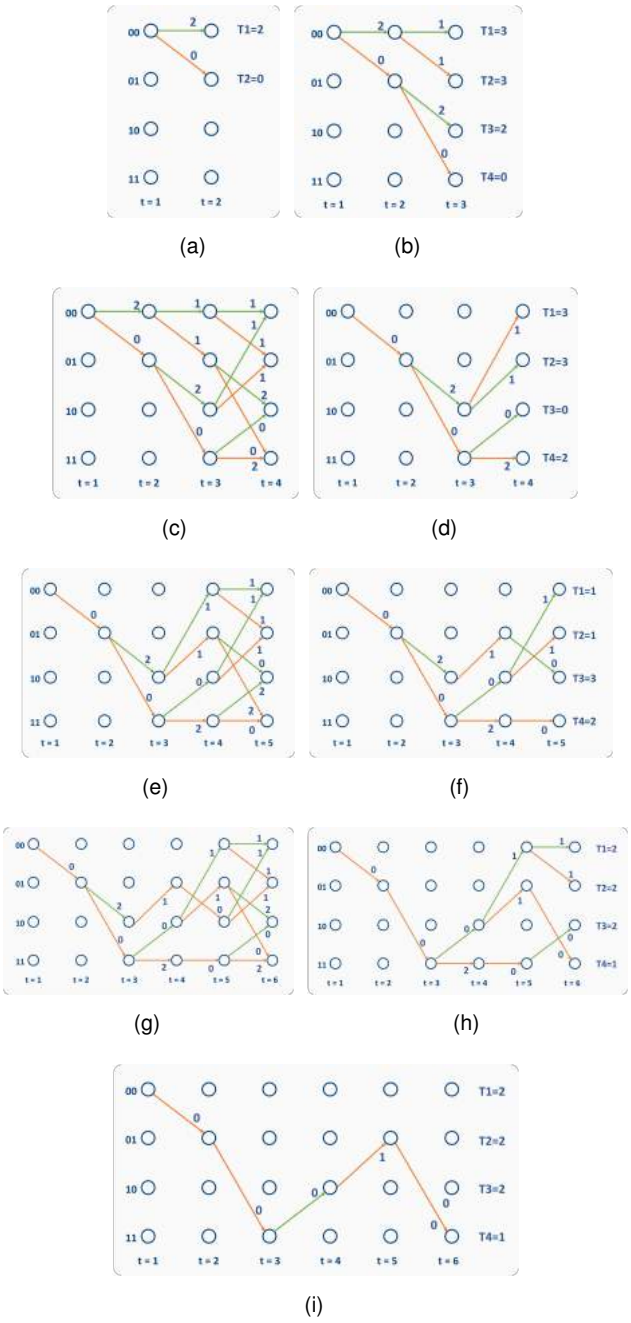


Figure 2.4: Viterbi Algorithm working example with decode output "11011"

2.8 Viterbi decoder

Figure 2.5 shows a general block diagram of the Viterbi decoder, which is composed of three functional blocks He et al. (2012):

Branch Metric Unit (BMU): calculate the branch metric (BM) between possible sequences and received sequences.

Add-Compare-Select Unit (ACSU): iterative calculations by adding the BM to the path metric (PM), comparing and selecting the path with the lower PM as the survivor path.

Survivor Management Unit (SMU): output the received sequence according to the survivor path.

The details of each block illustrates in Chapter 4, Viterbi Decoder Architecture.

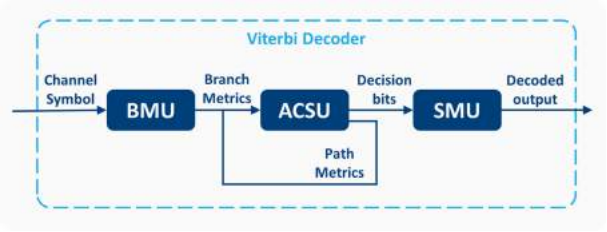


Figure 2.5: Block diagram of the Viterbi decoder

2.9 De-puncturing

To support efficient and flexible transmission methods, channel coding techniques need to take into account a

variety of different transmission data rates. For this reason, channel coding techniques often use punching to achieve rate matching based on $Datarate = clockrate \times puncturedrate$ Dong-Sun Kim et al. (2007). In coding theory, puncturing is the process of removing some of the parity bits after encoding with an error-correction code, which will provide the function of a high bit rate and less redundancy Solomon and Stiffler (1965).

De-puncturing is the reverse process of puncturing. The de-interleaving module (figure 1.1) performs the process, which inserts the punched bits back into the data stream and provides another data stream of puncturing marks. This process symbolizes the Viterbi decoder where the erased bits are. The Viterbi algorithm determines the survivor path based on the path metric, which is the accumulated linear distance. However, the linear distance of the punched bits is 0. In other words, it does not affect the path metric. The redundant information in the convolutional code can ensure the decoder follows the correct path, as long as the punched bits do not exceed the threshold. As a result, the de-punching process increases the bit rate without decreasing the noise resilience of the decoder.

Figure 2.6 is an example of puncturing at $\frac{3}{4}$ punctured rates IEEE (2008). The convolutional code shown in figure 2.6 generates two code bits for each input bit. Thus, a given convolutional code has a ratio of $\frac{1}{2}$. Instead of sending 6 bits of supervisory data per 3 original bits, only 4 bits need to be sent by removing a specific bit every 3 bits on the coded output. The coded output is $\frac{3}{4}$ code rate, which means every 3 bits of original data are represented by 4 bits of the actual sending data. The pattern of puncturing here is [1 1 1 0 0 1], where '1' means to keep the bit and '0' means to omit the bit. The new bitstream will be formed up as one single stream for further transmitting and receiving. The omitted bits will be inserted back into the sequence as null bits in the deinterleaving module in the receiving part.

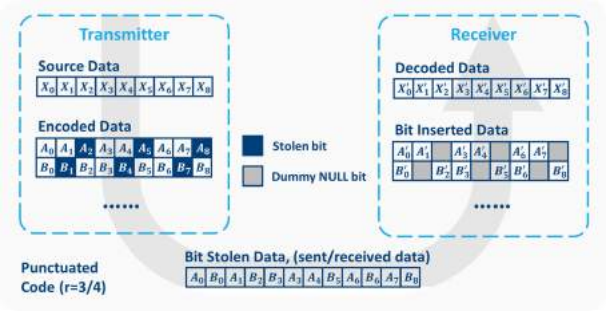


Figure 2.6: example of punched code of rate $\frac{3}{4}$

2.10 Open source hardware design

Open source hardware is a hardware design specification that is licensed so that anyone can study, modify, create and distribute the object. In essence, open hardware removes the common barriers to designing and building physical objects. It provides the ability for as many people as possible to build, remix, and share their knowledge of hardware design and functionality.

2.11 OpenWiFi

Software-defined radio (SDR) is a wireless communication design that uses software to implement physical layer connections Jiao et al. (2020). The basic concept is to use hardware as the basic platform for wireless communication and implement as many wireless communication functions as possible in software. Therefore, the system functions can be realized by software upgrade without changing the hardware equipment.

The Openwifi project is an open source SDR design. It is an open source Wi-Fi chip implementation on the IEEE 802.11a/g/n (Wi-Fi) standard. It supports communication with commercial Wi-Fi devices. The SDR approach makes the design very flexible, allowing changes in frequency, bandwidth, mode, etc. The solution is ideal for prototyping special demanding use cases with very complex requirements that cannot be met by off-the-shelf chips.

2.12 FPGA (Field-programmable gate array)

Field Programmable Gate Arrays (FPGA) are semiconductor devices based around a matrix of configurable logic blocks connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements by the user. Because of its design flexibility, FPGA is a very suitable choice for implementing many signal processing tasks.

Very High-Speed Integrated Circuit Hardware Description Language (VHDL) is a description language used to describe hardware. It is utilized in hardware designs on FPGAs to express digital systems.

In Chapter 6 Evaluation, the hardware test platform combines the ZedBoard and the AD-FMCOMMS2-EBZ, as shown in figure 2.7. The ZedBoard is a Xilinx Zynq-7000 series FPGA development board, and the AD-FMCOMMS2-EBZ is a high-performance radio frequency (RF) transceiver for SDR.

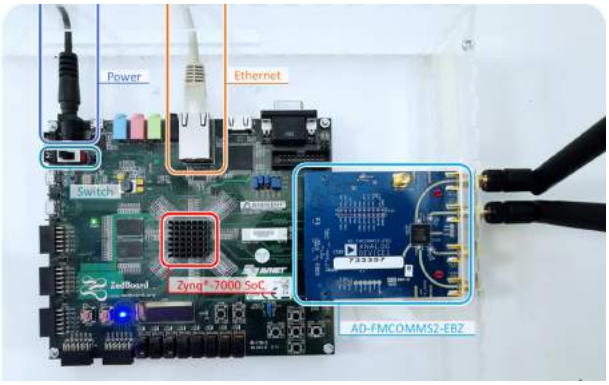


Figure 2.7: photo of the test FPGA (ZedBoard)

2.13 Related work

2.13.1 Xilinx Viterbi Decoder

The Xilinx Viterbi Decoder v7.0 is now used in OpenWIFI, a high-speed, compact decoder. It offers configurable constraint lengths from 3 to 9 and multiple code rates from $\frac{1}{2}$ to $\frac{1}{7}$. Moreover, it also supports de-punching functions. It supports multiple device families and is compatible with many common wireless communication standards Xilinx (2011).

2.13.2 Parallel Traceback algorithm

The Parallel Traceback algorithm has a high-throughput performance compared to the traditional decoding process Mohammadidoost and Hashemi (2020) detailed in Section 4.3.3. Figure 2.8 presents the structure of the parallel traceback algorithm with multiple threads. Longer window length (f_i) and acquisition length (v_2) effectively lower the BER but relatively increase the processing delay and memory consumption simultaneously because the delay and memory size are proportional to the sum of f_i and v_2, whose mechanism will be discussed in detail in section 4.3.3. Therefore, there is a trade-off between noise resilience, delay, and throughput.

The SMU structure algorithm may also strongly influence

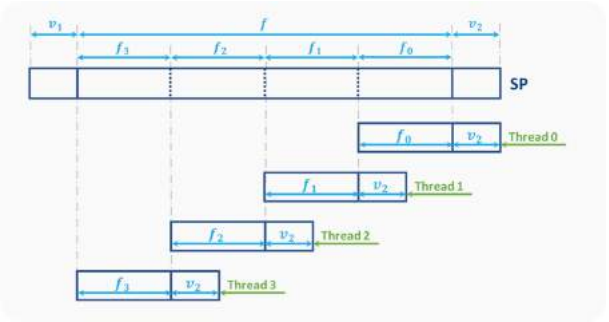


Figure 2.8: Parallel traceback algorithm with 4 threads

the performance Angarita et al. (2008). Among three different structures, RE, TB, and TB2f (Figure 2.9), register exchange (RE) requires double of slice area, while TB2f has a low data rate and high power consumption, and traceback (TB) is a middle compromise. The details of RE and TB will be presented in Section 4.3

3 REQUIREMENT ANALYSIS

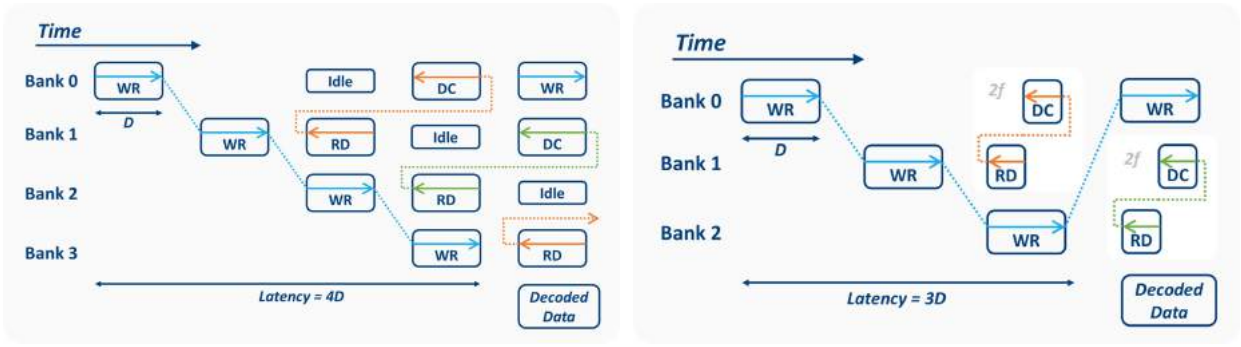
The Viterbi decoder performs decoding tasks in the OpenWiFi project, so it must comply with the IEEE 802.11a/g/n standard and the particular requirements of the OpenWiFi project. Moreover, the OpenWiFi decoding requirements are already known, with two 3-bit soft-decoded data streams as input, 1-bit output, the corresponding strobe signals, the decoding structure of (2, 1, 7) [171, 133], and a puncturing function. As a result, the ported decoder in this paper do not support the adjustable structure interface like the current using Xilinx one.

First is the selection of the evaluation parameters. Evaluation of a decoding scheme for any specific application is a process that requires consideration of several parameters Dizdar (2017). Table 3.1 lists all the parameters.

The delay and throughput are critical for the Viterbi decoder in a WLAN receiver. The openWiFi project supports up to 802.11n protocol (4th generation Wi-Fi). Wi-Fi 4 uses 52 OFDM data subcarriers, increases the most efficient coding rate to $\frac{5}{6}$ by compressing more bits, and shortens the protection interval to 400 *n*s, which defines a standard of 72.2Mbps Perahia (2008) to which the Viterbi decoder's throughput shall ideally adhere or exceed. The standard related to delay is the short inter-frame interval (SIFS), standardized channel access time-offs that ensure the receiver or transmitter has enough time to process the data. Consequently, the delay of a Wi-Fi system should be less than SIFS, which is specified in 802.11n as 10 μs.

In addition, the ported Viterbi decoder used different algorithms and parameters than the Xilinx proprietary one. Error performance, energy consumption, and area consumption also need to be considered. Although they are not explicitly required, low consumption and low BER are desired as they determine the efficiency and accuracy of the decoder.

Last but not least, it is necessary to implement a Viterbi decoder that can support a variety of different transmission code rates ($\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{5}{6}$) and block lengths (20 bytes up to thousands of bytes). In the implementation, this Viterbi decoder is supposed to support de-puncturing, further explained in section 4.



(a) TB diagram

(b) TB2f diagram

Figure 2.9: Viterbi Algorithm working

Metric	Typical Units	Explanation
Error performance	*BER v.s. *SNR	Error correction capability
Throughput	Mb/s	Number of decoded bits per second
Processing Delay	s (clock cycles)	The delay of each bit coming through the complete pipeline. In Wi-Fi System, it is measured as the time interval between the last bit going to and coming from the decoder
Power	mW	Power dissipation by the decoder circuit
Area	mm^2	The area spanned by the decoder circuit
Flexibility	-	The capability of a decoder implementation to support multiple code rates and block lengths

*Bit Error Rate (BER): the number of bit errors per unit time.
*Signal-to-noise Ratio (SNR): the ratio of signal power to the noise power.

Table 3.1: General evaluation parameters of the Viterbi decoder

4 VITERBI DECODER ARCHITECTURE

4.1 Branch Metric Unit

The branch metric unit calculates normed distances between the received code elements and the predicted code elements (figure 4.1). As mentioned in section 2.6, soft decision decoding has a better noise resilience by making full use of the information of the channel output signal (Tomlinson et al., 2017, pp. 25-40) . In the pursuit of better noise resilience to meet the requirements of OpenWiFi, soft decision decoding is a better choice.

As a clarification of how BMs work, the example in section 2.6 is utilized here. Considering the red point in the trellis diagram with the current state '01', it has two possible branches: 0 or 1, which correspond to the expected code elements '10' and '01'. In this case of $\frac{1}{2}$ coding rate, the total BM value will be the addition of two distance values. The example assumes the received symbols are '10'. For hard decision, the Hamming distance for branch 0 is $|1-1|+|0-0|=0$, and similarly, for branch 1 is $|1-0|+|0-1|=2$. This means the decoded bit is likely to be '0' rather than '1' for this node. While in soft decision, all the expected code elements are considered as strong bits, '10' will correspond to "111 011" according to Table 2.1. Here, for an example of 3-bit inputs, assuming the received code elements are "110 001" (relatively strong 1 and relatively weak 0), the Euclidean distance for branch 0 is

$$\begin{aligned}&\sqrt{(1-1)^2+(1-1)^2+(0-1)^2}\\&+\sqrt{(0-0)^2+(0-1)^2+(1-1)^2}=2.00\end{aligned} \tag{4.1}$$

And similarly, for branch 1 is

$$\begin{aligned}&\sqrt{(1-0)^2+(1-1)^2+(0-1)^2}\\&+\sqrt{(0-1)^2+(0-1)^2+(1-1)^2}=2.83\end{aligned} \tag{4.2}$$

Nevertheless, due to the high throughput of OpenWiFi, the computation of the multiple squared Euclidean distances will be the bottleneck of the soft decision Viterbi decoder. Compared to arithmetic, the complexity of square root operations is extremely high.

Linear distance is more advanced than Euclidean distance, reducing computational complexity by decomposing the constellation map into one dimension. Moreover, it does not affect the decoder's noise resilience Lou (2002).

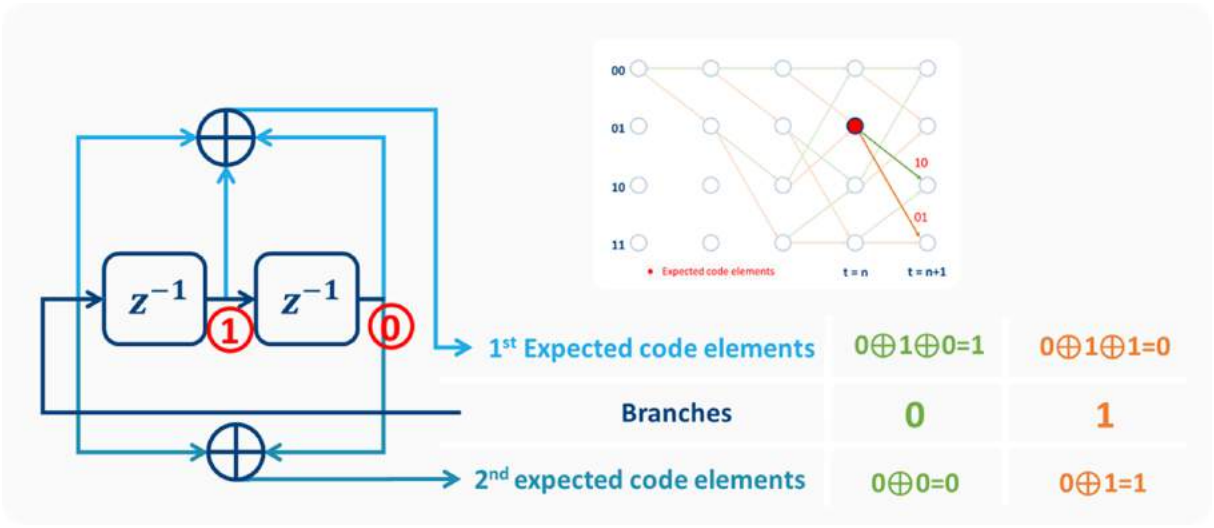


Figure 4.1: BMs example

Therefore, square and square root operations can be omitted. Every decimal addition in Euclidean distance will be replaced by a simple integer addition in linear distance to calculate the branch metric. In addition, there are 2^k branch metrics when the code rate is $\frac{1}{k}$ Ma (2010). The linear distances for 3-bit inputs are listed in Table 4.1. Since the code rate is $\frac{1}{2}$, there are four possible distances for one input. Equation 4.3 shows four linear distances, where the pair (x, y) represents the received soft decision bits.

$$\beta = \begin{bmatrix} x+y \\ x-y \\ -x+y \\ -x-y \end{bmatrix} \tag{4.3}$$

The plus or minus operations depend on the expected code elements, and the matching rule does not matter. If 0 means plus and 1 means minus, the possible branch will be the larger one, and vice versa.

Probability	Soft decision value	Linear distance
Strongest 1	111	-4
Relatively strong 1	110	-3
Relatively weak 1	101	-2
Weakest 1	100	-1
Weakest 0	000	0
Relatively weak 0	001	1
Relatively strong 0	010	2
Strongest 0	011	3

Table 4.1: Relationship between Soft decision value and Linear distance for 3-bit inputs

For the previous example, assuming the received code elements are still '110 010', the linear distance for branch 0 is $-(-3)+2=5$, and the linear distance for branch 1 is $-3-2=-5$. The decoded bit is likely to be 0 rather than 1, which is the same result as the soft decision.

4.2 Add-Compare-Select Unit

The Add, Compare, and Select Unit (ACSU) iteratively computes the path metric (PM) by adding the previous PMs to the corresponding BMs and sending the output to the comparator and selection unit. The output generates an arbitration bit and is stored in the memory. The following survivor management units use this to search for survival paths. As mentioned in the Viterbi algorithm in Section 2.4, the convolutional code consists of generating polynomials are [171 133] with a constraint length of 7 so that there are $2^{7-1}=64$ nodes for each trellis state Sun and Ding (2012).

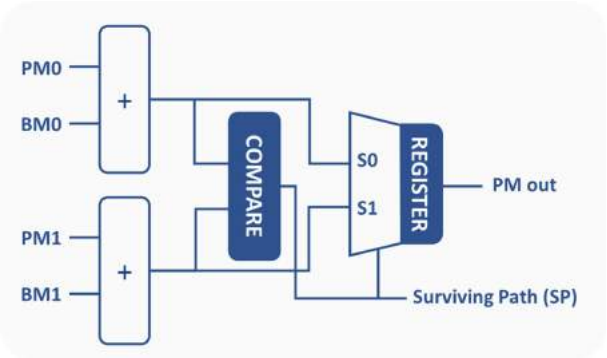


Figure 4.2: ACSU Architecture Angarita et al. (2008)

Therefore, each state needs to compute 64 nodes. One ACSU represents one node. A parallel structure consisting of 64 ACSU blocks is chosen for implementation to meet the delay requirements in the IEEE 802.11n standard. Therefore, each decoding clock cycle generates a 64-bit arbitration vector. In addition, due to the limited data

word length, the iterative process requires normalization of the path metric to avoid overflow. This normalization is performed at the output of each ACSU block by subtracting the PM of each survivor path.

4.3 Survivor Management Unit

The SMU decides the final survivor path and tracks the information bits associated with the survivor path to decoding the transmitted data. There are two basic methods for searching the survival path, Register Exchange (RE) and Traceback (TB), depending on how the information is stored and retrieved in the SMU Kamuf et al. (2007).

4.3.1 Register Exchange

RE writing process

RE uses a set of multiplexers and registers to store and update the sequence of survivors for each state. The registers will be fully reflashed after every new input comes. The register values are the bits from the previous node and an extra bit of the next path. Here, the example in Section 2.4 will still be utilized to illustrate how RE works. The branch metrics are omitted, and the blue square brackets represent the registers. Figure 4.3 a) - e) are corresponded figure 2.4 to a) b) d) f) and h) respectively in Section 2.7. Orange means the next branch is 1, while green means 0.

Figures 4.3 a) - e) show that the registers are fully updated while the survivor paths change. As a result, RE consumes much area and power due to the frequent switching of the states in the registers (bits are physically transferred from the previous stage to the next), especially for deeper traceback depths

RE reading process

The reading process of RE is extracting the data from the state register with the lowest path metrics. A series of shift registers are applied to process this step. For the previous example, because state four has the lowest path metrics, the corresponding registers, ”11011”, are read and sent to the output (figure 4.4).

RE generates the result immediately once the input ends. As a result, it has low processing delay and high area efficiency since it extracts the information bits directly from the encoded bitstream.

4.3.2 Traceback

TB writing process

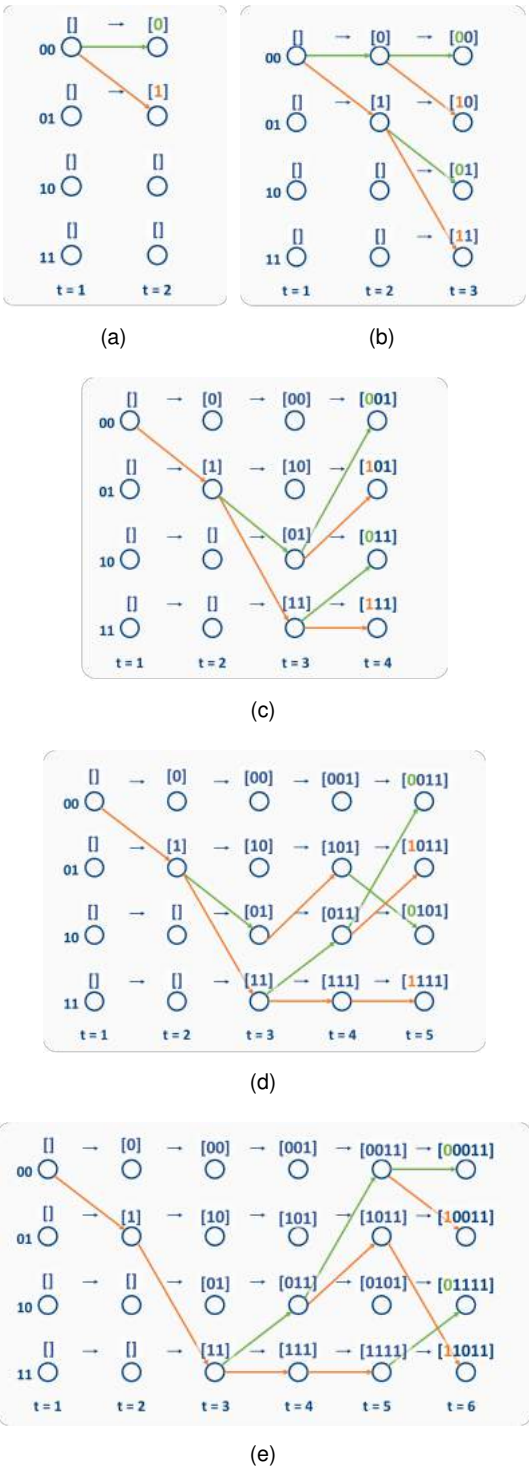


Figure 4.3: RE writing process

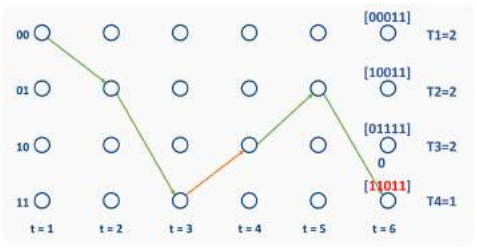
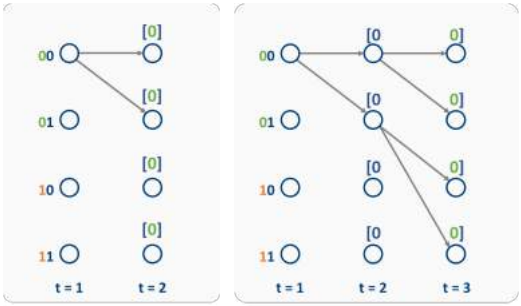
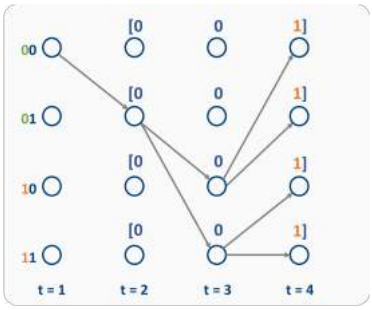


Figure 4.4: RE reading process

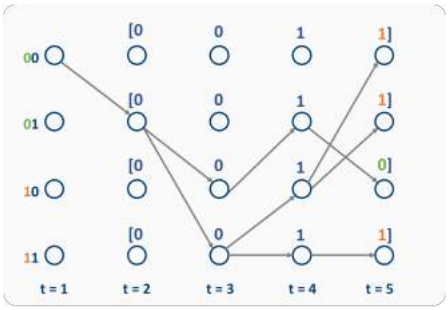
Unlike RE writing the survivor path to registers, TB writes the previous state's most significant bit or least significant bit. TB stores the decision vector in random access memory (RAM) and finds the survivor path by tracing back the RAM. Figures 15 present the writing process of the same RE example above, using the most significant bit of the previous state. Figure 4.5 a) - e) are also corresponded to a) b) d) f) and h) respectively in section 2.7.



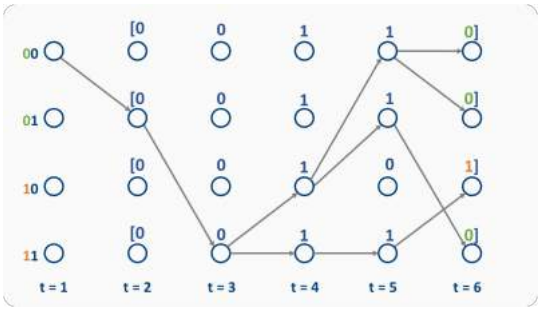
(a) (b)



(c)



(d)



(e)

Figure 4.5: TB writing process Putra et al. (2016)

It is observed that the previous registers do not be changed when new input comes. Hence, TB has a relatively lower energy consumption compared to RE.

TB reading process

The reading process of TB follows the principle as shown in figure 4.6. The initial state is the one with the lowest path metric, which is state four in the example. By reading the corresponding register, the previous state can be determined. Then, the state registers are shifted to a lower bit and TB output is generated

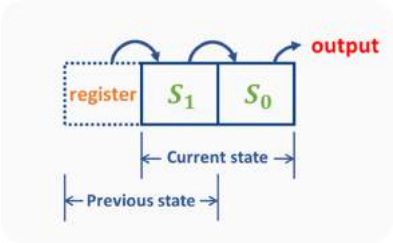


Figure 4.6: TB reading principle

Continuing the previous example, Figure set 4.7 illustrates how this principle works. Starting from the initial state 11 at $t = 6$ (figure 4.5(a)), 0 is read from the register, and the traceback goes to the previous state 01. Then, 1 is read from the register at $t = 5$ (figure 4.5(b)), so the previous state is updated to 10, and 1 is pushed to the output. By repeating the same process until $t = 0$ (figure 4.5(c)-4.5(e)), a sequence of output is obtained as "11011". Because TB is a reverse order operation, the sequence needs to be reordered to "11011" to get the actual output, which will be discussed in section 4.4.

4.3.3 Further discussion

Overall, considering the delay and consumption, the open source Viterbi decoder with TB algorithm was selected as the ported one. Furthermore, the SMU consists of two parts: the traceback unit (TBU) and the RAM. The TBU implements the traceback algorithm. The RAM stores the arbitration bits from the ACSU and reads them when the TBU works.

TBU reads the data from the very last bit of the RAM to the very first, which means it only processes after receiving a data block. However, in the actual case, the length of the Wi-Fi packet is up to thousands of bytes. The application is not able to allow the enormous delay. Start the traceback in the middle of the continuous data stream to solve this problem. However, the result will not be entirely correct in this case because the middle states are uncertain. For the previous example, even if the initial state is the one currently with minimal path value, the path could still be incorrect, as shown in figure 4.8.

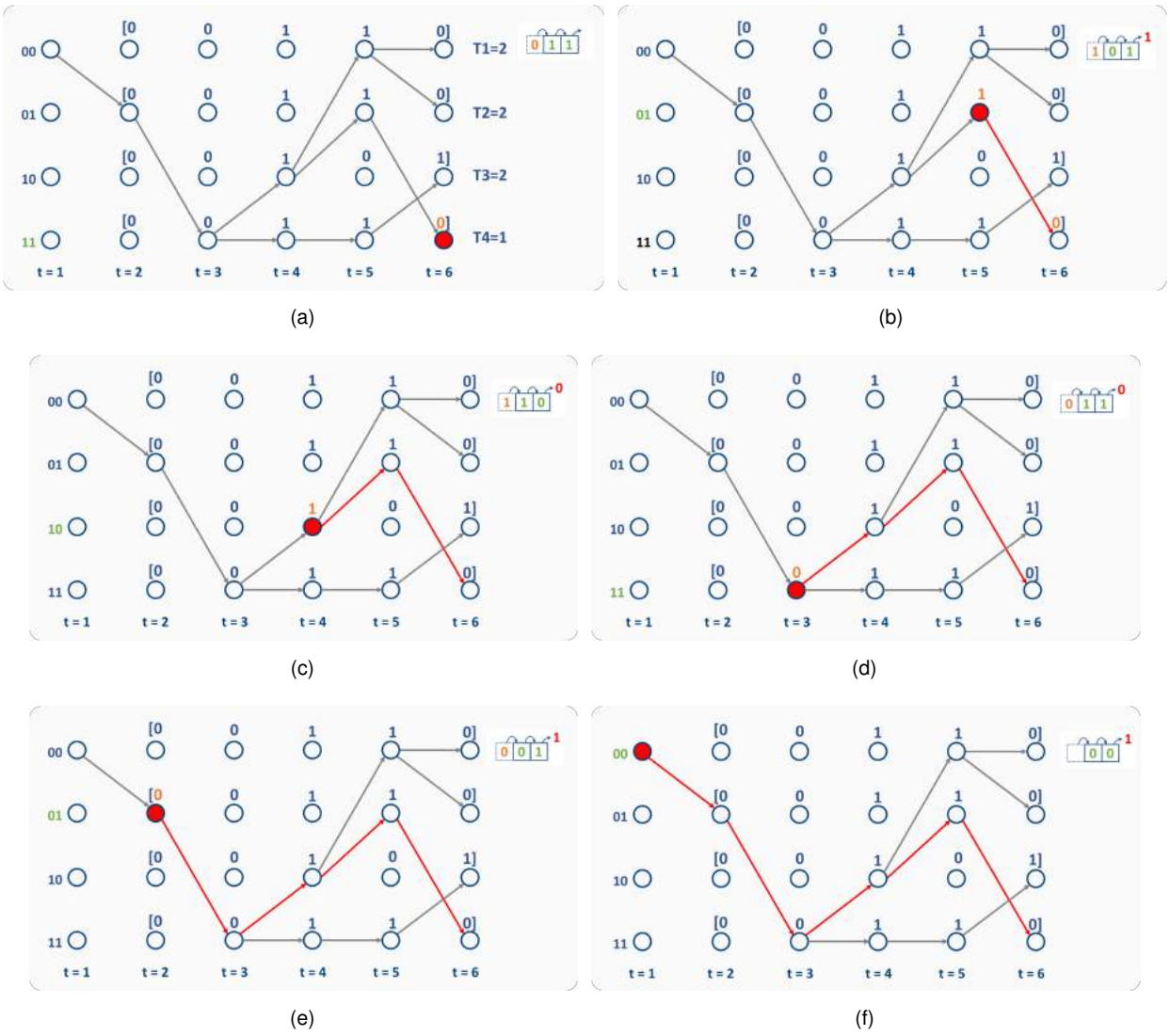


Figure 4.7: TB reading process

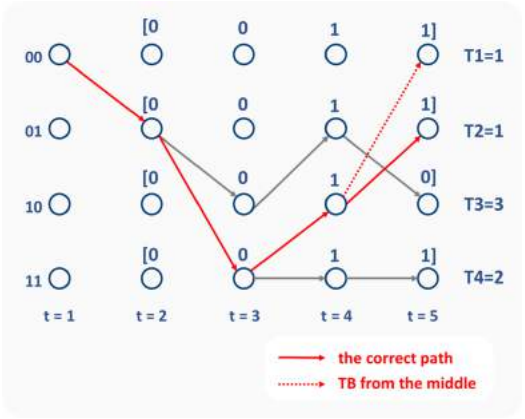


Figure 4.8: TB reading starting from the middle by PM values

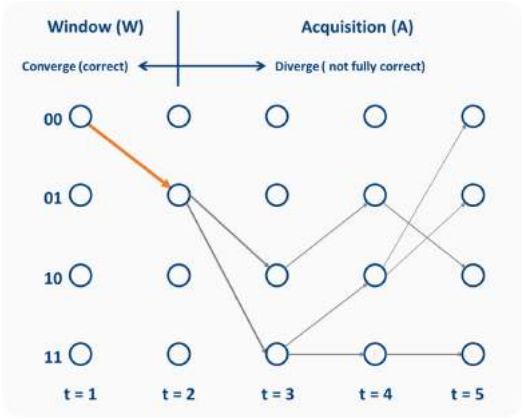


Figure 4.9: unification property of the TB

Usually, the decoded bits are unreliable from the beginning of the traceback to some later stages. Then, after some stages, the survivor path of the traceback starts to converge, and the decoded bits become reliable and usable, which is called the unification property Cypher and Shung (2002). This property can be seen in Figure 4.9.

As a result, the TB algorithm is divided into the acquisition and window phases. The blocks of the acquisition phase are used to make the survivor paths converge to the cor-

rect condition and are not used as outputs. The blocks of the window phase are reliable enough to be output to the next unit. The TB initial state is random because of the unification property but normally chosen to be state 0. Therefore, traceback the acquisition block first and then the window block in implementation.

However, the acquisition phase takes extra time to execute, which causes the decoder to skip part of the input data. As a result, the parallel traceback algorithm is used for continuous decoding Mohammadidoost and Hashemi (2020). Two TBUs should be working in turns to generate the output. While one of them works in the window phase, the other is simultaneously in the acquisition phase. The structure is shown in Figure 2.8. Blocks overlap their right-hand neighbors, and the overlapping part corresponds to the acquisition phase. It is recommended to be at least six times the number of states to ensure the acquisition has enough samples to converge to the correct values.

In addition, the length of the acquisition block (*A*) and the length of the window block (*W*) both impact the noise resilience. The longer *A* and *W* are, the lower BER (better accuracy) is. Moreover, *A* significantly influences BER compared to *W* Mohammadidoost and Hashemi (2020), and the processing delay is proportional to (*A*+*W*). As a result, it is a trade-off between accuracy and delay.

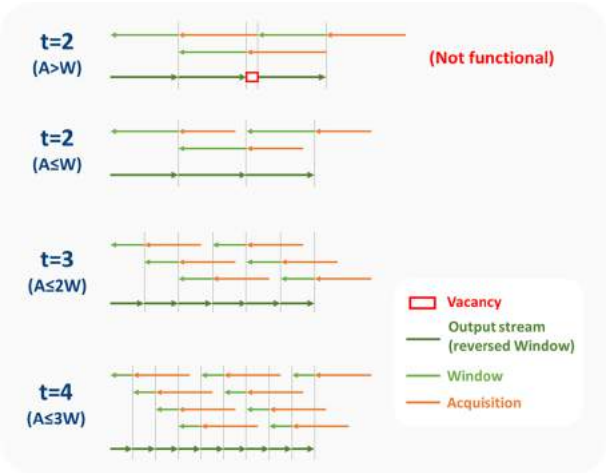


Figure 4.10: t threads traceback

Furthermore, if the number of parallel traceback threads t is equal to two, *W* ought to be larger than *A* to ensure the output is continuous when there is a continuous input stream, as shown in Figure 4.10. For the cases of parallel traceback threads of more than two $(t > 2)$, *W* can be shorter than *A*, and the output is still guaranteed to work continuously. In these cases, it reduces the delay to $\frac{2+t}{2t}(t > 2)$ but consumes extra $\frac{t-2}{2}(t > 2)$ times for the original memory. Consequently, it is not worth paying double or even more of the memory to exchange for a slight improvement in the delay. Referring to the data in the work of Alireza (Mohammadidoost and Hashemi (2020)), *A* and *W* are selected to be 50 and 55, respectively.

4.4 Reorder Unit

Since traceback operates in reverse of the received sequence, it is necessary for the reorder unit to invert the sequence. This is achieved by using shift registers to store the decoding results from the TBU. However, only when the TBU completes a window, the contents of the shift registers are output in the reverse order. Therefore it needs to wait for all data of a window to complete, which will increase the delay of the Viterbi decoder accordingly.

5 IMPLEMENTATION

FPGA is a reasonable choice for implementing signal processing tasks considering the design flexibility. Moreover, because the OpenWiFi receiver was built on FPGA, the ported decoder, part of the receiver, should also be implemented on FPGA. The implementations are performed on a ZedBoard, part of the Zynq-7000 Xilinx Zynq-7000 device family. The software used for RTL-level design, simulation, and synthesis is Xilinx Vivado 2018.3. The ported and adapted Viterbi decoder is coded in the VHDL Matthias and Markus (2012), aiming to be compatible with the rest of OpenWiFi and be functionally equivalent to the proprietary Xilinx one. All units of the decoder are connected to a top-level interface. Therefore modifying, exchanging, or adding a unit is simple during RTL design. The Viterbi decoder is introduced in the following section by the flow chart in Figure 5.1. With more specifications, a straightforward implementation will be provided module by module.

5.1 Branch Metric Unit

BMU calculates all possible branch distances by linear distance. As shown in Equation 4.3, four combinations are possible. In addition, the input soft bits from the previous module are three bits.

As mentioned in section 2.9, considering the puncturing situation, there needs to be a linear distance value between weak 0 and weak 1 to be assigned to the null bit, which should make no impact in ACSU (the value of 0). Considering that all the linear distance value performs as a relative value in the next unit ACSU, parallelly adding or subtracting the value will lead to the same result. As a result, the linear distance value from Table 4.1 will be adjusted to four bits length (Table 5.1). It is done by combining the two input 4-bit soft decision bits through an adder

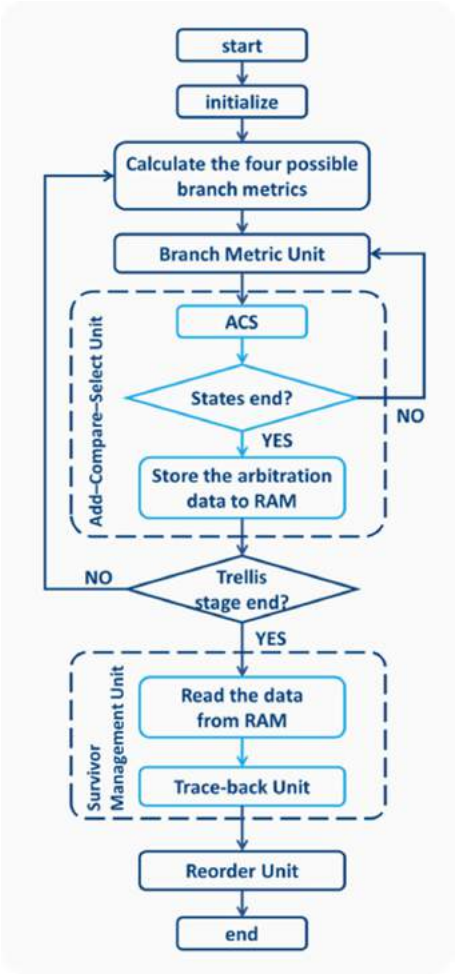


Figure 5.1: Viterbi decoder flow chart

or subtractor. In view of the sign bit and avoiding overflow, the output of BMU is set as 6 bits. Then the outputs are saved in shift registers, and a multiplexer controls the enable signal.

Probability	Linear distance	Bit representation
Strongest 1	-7	1001
Relatively strong 1	-5	1011
Relatively weak 1	-3	1101
Weakest 1	-1	1111
Puncturing bit	0	0000
Weakest 0	1	0001
Relatively weak 0	3	0111
Relatively strong 0	5	0101
Strongest 0	7	0111

Table 5.1: Linear distance and bit representation in ported decoder of 3-bit inputs, where the least significant bit depends on the punched signal

5.2 Add-Compare-Select Unit

The implementation uses 64 ACSUs in parallel to reduce the processing delay. As Figure 5.3 shows, the function of each ACSU first requires adding the two incoming branch distances from the high and low branches to the path metric accumulated by the previous high and low nodes. Then the high and low probabilities will be compared and selected for the more probable path. In this implementation, the most likely path is the one with the higher value (maximum search). Finally, the path metric for the survivor path is stored in a register within the ACS module. There are two required outputs: the updated path metric, and the decision bit of the previous state's most significant bit, as introduced in Section 4.3.2.

Furthermore, there is a normalization step to prevent the accumulated values from overflowing. The register v_diff length is the same bits as the two signed vectors: v_low and v_high. As presented in figure 5.3 as well as the code figure 5.4, instead of directly doing the comparison, the normalization method does subtraction and compares with 0. The following explain in detail how this approach effectively avoid the problems caused by overflow.

An register overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits, either higher than the maximum or lower than the minimum representable value Nidecki (2020). A maximum overflow causes the value to wrap and become negative, while a minimum overflow will lead to the opposite influence. Assume, after addition or subtraction, a number with n bits (including the sign bit) is ideally to be *P*, but it is actually read as *R* because of the overflow. It follow the equations 5.1 and 5.2 :

$$maximum\ overflow\ situation: R = P - 2^n - 1 \quad (5.1)$$

$$minimum\ overflow\ situation: R = 2^n + 1 + P \quad (5.2)$$

Mathematically, if the operation occurs with an equal times of maximum overflow and minimum overflow, the result is compensated to be correct.

In Figure 5.5, the circles represent the numbers on a number line. The hollow circles represent the ideal values, while the solid circles represent the real value in the *n*-bit registers. In the situation of maximum overflow situation, figure 5.5 a), although the orange circle (*O*) is ideally larger than the green one (*G*), it is actually smaller in comparison. However, if doing the subtraction $O-G$, the result will be $x+y$, because the minimum overflow happens. Then, comparing the result with zero as $O-G=x+y>0$, it is

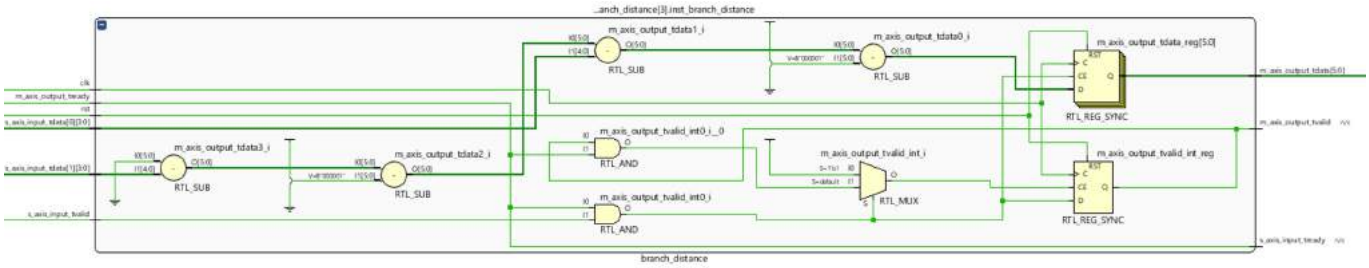


Figure 5.2: BMU Schematic

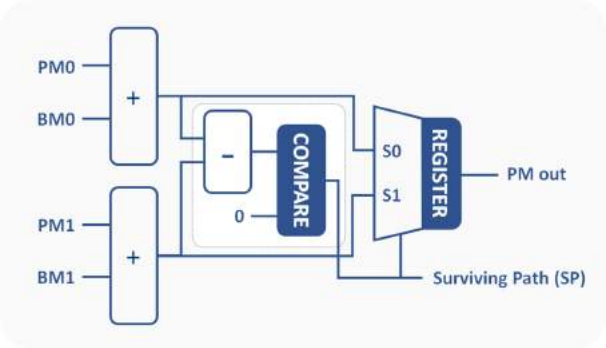


Figure 5.3: ACSU implementation

```
1  -- Add
2  v_low  := signed(
       s_axis_inbranch_tdata_low)
       + signed(
       s_axis_inprev_tdata_low);
3  v_high := signed(
       s_axis_inbranch_tdata_high)
       + signed(
       s_axis_inprev_tdata_high);
4
5   -- normalization
6  v_diff := v_low - v_high;
7
8  -- Compare with 0, and select
       the correct path
9  if v_diff < 0 then
10 m_axis_outdec_tdata  <= '1';
11 m_axis_outprob_tdata <=
       std_logic_vector(v_high);
12 else
13 m_axis_outdec_tdata  <= '0';
14 m_axis_outprob_tdata <=
       std_logic_vector(v_low);
15 end if;
```

Figure 5.4: ACSU Normalization in VHDL

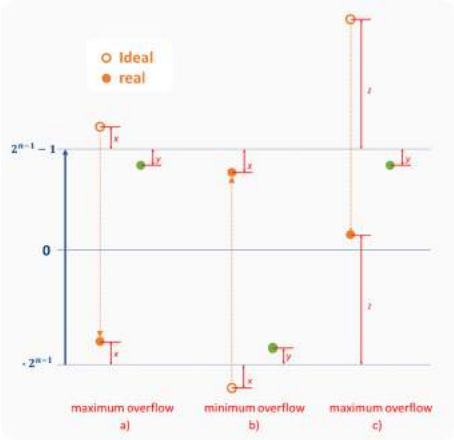


Figure 5.5: example of normalization

known that $O > G$, which is correct as the ideal situation. It is the same case in figure 5.5 b), the situation of minimum overflow. This time *O* is supposed to be smaller than *G*. Similarly, when $O - G$, a maximum overflow happens. The result $O - G = -x - y < 0$, which means $O < G$. However, if the overflow exceeds half of the range, the normalization method fails. In figure 5.5 c), ideally $O > G$, but this time compared with figure a), no overflow happens when $O - G$. As a result, in reality $O < G$, which is contrary to the ideal situation.

To prevent situation c) happening, the registers' range should be doubled than the difference between the largest value and the smallest value of PMs. It is hard to mathematically describe this value, hence, a test is generated in MATLAB to model this value, which code is shown in figure 5.6. By doing 10^8 times iterations, Figure 5.7 represents that the maximum difference of PMs reaches at most $2 \times \triangle b \times K$, where $\triangle b$ is the largest branch metrics difference and *K* is the number of constraint length as defined in Section 2.4. As a result, for the situation of this paper (*b*=14, *K*=7), the maximum PM difference is 196. As a result, the registers' range should be larger than 392 to give enough margin for normalization. The register length of 9 bits, range 512 in decimal, should be applied.

5.3 RAM control

The arbitration data generated by the ACSU needs to be stored in RAM before the traceback unit reads it. So there are three operations in total: the two read operations correspond to the acquisition phase (length 50) and the window phase (length 55), as described in Section 4.3.3. There is also the ASCU write operation to RAM. These three operations perform simultaneously with a parallel structure of four single-port RAMs to reduce delay, as shown in Figure 2.9(a) as well as the example in Table 5.2.

```
n_pick = [3,4,5,6,5,6,7,6,7,6,7];
state_pick = [2,2,2,2,3,3,3,4,4,5,5];
diff = zeros(11,1);
for k = 1:11
    n = n_pick(k);
    state = state_pick(k);
    state_nr = 2 ^ state;
    s_old = zeros(state_nr,1);
    s_new = zeros(state_nr,1);

    diff_max = 0;
    for i = 1:10^8
        s_old = s_new;
        random = floor(rand(state_nr,1)*(2*n+1))-n;
        for j = 1:state_nr
            if mod(j,2)==1
                s_new(j) = max(s_old((j+1)/2)+random((j+1)/2),
                    s_old((j+1)/2+state_nr/2)+random((j+1)/2+
                    state_nr/2)); % Generate Random BM
            else
                s_new(j) = max(s_old(j/2)-random(j/2), s_old(j/2+
                    state_nr/2)-random(j/2+state_nr/2)); % Select
                    Survivor Path
            end
        end
        diff_max = max(max(s_new)-min(s_new), diff_max); % Update
            Results
    end
    diff(k) = diff_max; % Save Results
end
```

Figure 5.6: MATLAB code for modeling the difference between the largest value and the smallest value of PMs with 10^8 times iterations

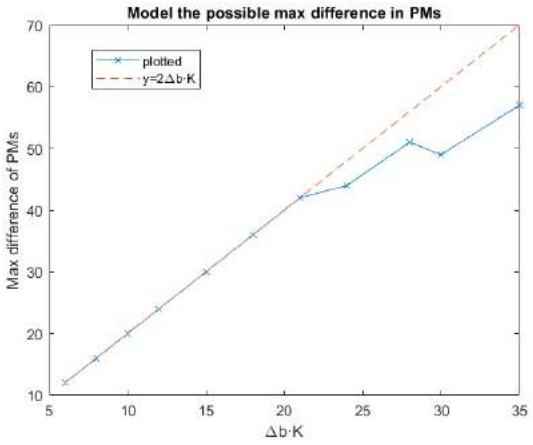


Figure 5.7: MATLAB results of the difference between the largest value and the smallest value of PMs

The minimum number of RAMs is four. If not, the reading and writing will operate the same memory, shown as in the ninth clock cycle nine of the example.

On the one hand, using four RAMs in parallel allows for a faster and more efficient design. However, on the other hand, memory management becomes more challenging because reading operations and writing operations need to be handled simultaneously. For each write operation, when the ACSU fills a block with RAM, the data access from all units must be switched. Moreover, each read operation has to wait for the previous write operation to complete, then proceed to the acquisition and window phases. Therefore, a RAM control unit is designed for this task. The controller entity *ram_ctrl* is implemented as a finite state machine (FSM). It is further divided into two FSMs: the write FSM and the read FSM. In the implementation, six pointers are operated. A RAM writing pointer indicates which one of the four RAMs is being used. An address writing pointer to indicate the current write address in the RAM. Similarly, two RAM reading pointers and two address reading pointers correspond to the two traceback units (TBU).

5.3.1 RAM Write Process

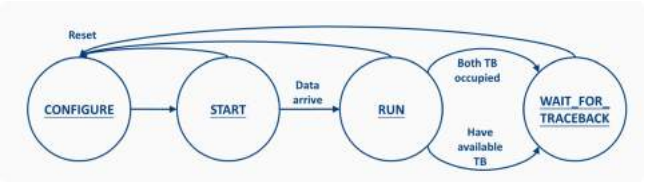


Figure 5.8: FSM diagram of RAM writing process

The write operation stores the output of the ACSU in RAM, and its FSM has four states which are seen in figure 5.8. It handles:

- State: CONFIGURE

Cycle	RAM1		RAM2		RAM3		RAM4	
	Unit	Addr	Unit	Addr	Unit	Addr	Unit	Addr
1	Write	3						
2	Write	4						
3			Write	0				
4			Write	1				
5			Write	2				
6			read *ACQ 1	2	Write	0		
7			read ACQ 1	1	Write	1		
8			read *WIN 1	0	Write	2		
9	read WIN 1	4			read ACQ 2	2	Write	0
10	read WIN 1	3			read ACQ 2	1	Write	1
11					read WIN 2	0	Write	2
12	Write	0	read WIN 2	4			read ACQ 1	2
13	Write	1	read WIN 2	3			read ACQ 1	1
14	Write	2					read WIN 1	0
15	read ACQ 2	2	Write	0	read WIN 2	4		
16	read ACQ 2	1	Write	1	read WIN 2	4		
17	read WIN 2	0	Write	2				
18			read ACQ 1	2	Write	0	read WIN 2	4
19			read ACQ 1	1	Write	1	read WIN 2	4
20			read WIN 1	0	Write	2		
21	read WIN 1	4			read ACQ 2	2	Write	0
22	read WIN 1	3			read ACQ 2	1	Write	1
*WIN: window phase; *ACQ: acquisition phase (1 and 2 represent different TBU)								

Table 5.2: RAM working Example (acquisition length = 2, window length = 3)

It is necessary to configure the decoder before each block. The $CONFIGURE$ state resets all parameters, sets the write address pointer to the initial address, and sets the write RAM pointer.

- State: START

After the decoder configuration is complete, the decoder waits for a new block. When the AXIS handshake occurs, packet transmission begins. The first write is a special case where only the acquisition length of data is written.

- State: RUN

Continuously writes blocks with window length to RAM until the end of the input data. However, processing 4 RAMs in parallel requires to prevent overwriting the RAM that is being read in the $TRACEBACK$ state. The implementation uses a conditional constraint. When the last bit of the current window is completely written, it will enter the $WAIT\_FOR\_TRACEBACK$ state if both TBUs are occupied. When the Viterbi decoder decodes a sufficient number of bits, a reset signal is generated externally to the decoder, pointing the write FSM to the CONFIGURE state.

- State: WAIT_FOR_TRACEBACK

In this state, keep waiting for a spare traceback unit.

5.3.2 RAM Read Process

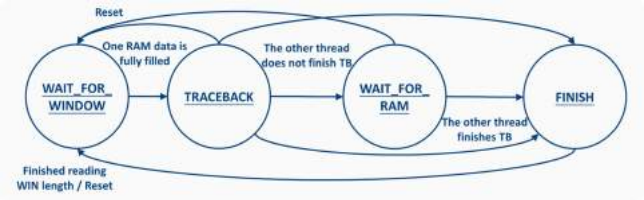


Figure 5.9: FSM diagram of RAM reading process

As figure 5.9 shows, the read FSM has four states. It handles:

- State: WAIT_FOR_WINDOW

Before performing traceback, it needs to wait for the write operation to fill a window length of data into RAM. Then, to read from the last RAM that was being written each time performing traceback. Two pointers are utilized to mark which RAM is being read and which RAM is being written.

- State: TRACEBACK

First, read the data with the acquisition length to allow the survivor paths to converge to the correct condition. Then, as the window length is greater than the acquisition length, the remaining few RAM bits are used as part of the window phase. At the same time, the RAM control module has to enable the TBU to receive the data for traceback. If another TBU is in the traceback state and is reading the previous RAM of the current read RAM pointer, it needs to jump to the wait state. There is a particular case when the first TBU is in the traceback state, and the RAM it is reading is the last RAM, the second TBU is required to read, then the second SMU needs to jump to the $WAIT\_FOR\_RAM$ state.

- State: WAIT_FOR_RAM

While in this state, keep waiting for the first TBU to end the traceback state, or the RAM being read by the first TBU is not the last one needed by the second one.

- State: FINISH

Get the remaining data from the second RAM we need for traceback (no acquisition data in this RAM).

5.4 Traceback Unit

The arbitration data from the ACS unit are stored in RAM. Furthermore, the traceback unit processes those arbitration data backward from a given starting state. Therefore, the output order is reversed. As mentioned in Section 4.3, only trailing is currently supported. It means that the traceback always starts from the zero states of the trellis. The TBU is implemented with a shift register, which stores the current trellis state, as shown in figure 5.10. The RAM arbitration data is sent into the shift register, and the earliest bit is sent to the output. Be aware that the output of the TBU is in the reverse order of the expected transmission.

5.5 Reorder Unit

The reorder unit uses shift registers to hold the traceback result. Then, the content of the shift registers is output in reversed order when the TBU finishes the whole window.

6 EVALUATION

The evaluation is divided into two parts: the performance of the target Viterbi decoder and functional verification on the board.

Performance: Through simulation and synthesis, observed processing delay, maximum throughput, area/energy consumption, noise resilience, and flexibility.

Functional verification: Integrated the Viterbi decoder in the OpenWiFi project as a replacement of the proprietary Xilinx one and prototype on FPGA.

Test environment: Set up in Xilinx Vivado 2018.3 with Ubuntu OS 18.04 on Linux. Simulation, synthesis, and on-board prototyping on ZedBoard which was mentioned in section 2.12. Used MATLAB R2022a to add the AWGN mentioned in section 2.2 to the signal and plot the BER against the output bitstream file of the decoder.

Test objects: Since this thesis not only tested the performance of the Viterbi decoder but also evaluated whether it matches with other modules in the OpenWiFi project. So different test objects were selected for different objectives. For delay, maximum throughput, and area/energy consumption, simulate and synthesize the target Viterbi decoder; For noise resilience, simulate and synthesize the OFDM receiver module ($openofdm\_rx$) as shown in Figure 1.1. As mentioned in Chapter 3, the noise, in reality, is generated in the channel, not inside OpenWiFi, so adding noise only at the input of the Viterbi decoder is meaningless. Therefore it is reasonable to test the OFDM receiver module ($openofdm\_rx$) instead of the decoder. For on-board verification, since it is necessary to verify whether the Viterbi decoder matches the other modules in OpenWiFi, the synthesis and implementation object is the OpenWiFi project.

Noise addition: The channel noise was an analog signal, but experimental conditions did not allow adding noise to the channel. The alternative was to add noise to the bitstream after sampling at the receiver side. The sampled signals were In-phase and Quadrature (I/Q) signals, which were digital signals. Used the AWGN function in MATLAB to add noise to this data file.

The In-phase and Quadrature (I/Q) sample signals were prepared as data files for testing the receiver. The observation was the recorded output result of the Viterbi decoder. The selected test I/Q files differed in protocols, data rate, packet length, number of packets, and noise. The following test files (Table 6.1) were selected without noise and with only one packet:

Besides the files above, some other simulation files were real-life recordings. The data of different lengths continuously flowed into the system with 6.5, 48, 52, 58.5, and 65 Mbps data rates.

6.1 Processing delay and throughput

In the theoretical derivation, the traceback algorithm first stored the data of acquisition length and the window length in RAM and then reversed the output sequentially. Thus it

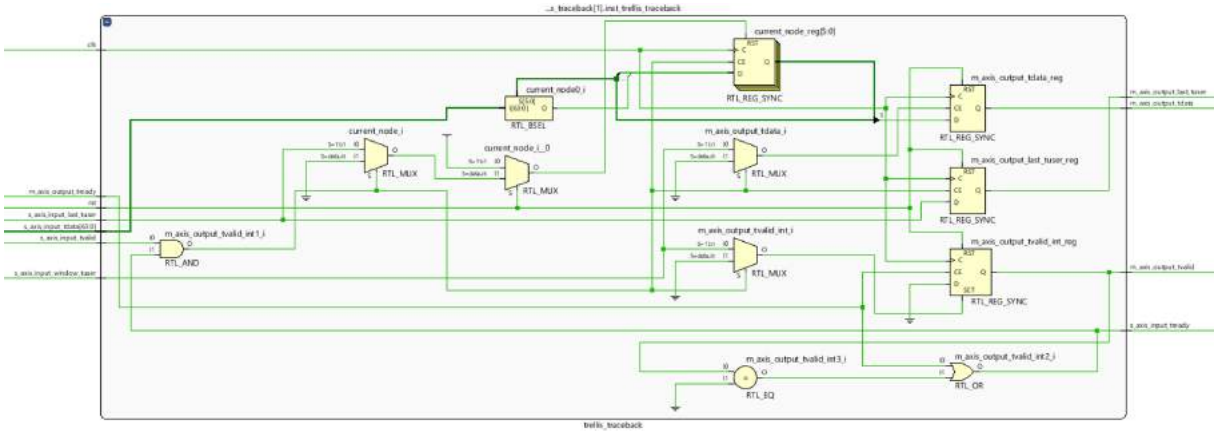


Figure 5.10: TBU Schematic

File No.	Protocol	Speed (Mbps)	Modulation type	Info		Data length (bytes)
				*MCS	*GI	
1	802.11 a/g	6	BPSK	-	-	14/1537/4000
2		54	64_QAM	-	-	14/1537/4000
3	802.11 n (HT-OFDM)	6.5	BPSK	0	0	14/1537/4000
4		7.2	BPSK	0	1	14/1537/4000
5		65	64_QAM	7	0	14/1537/4000
6		72.2	64_QAM	7	1	14/1537/4000

*Modulation and Coding Scheme (MCS): a certain combination of modulation and coding rate (Table 6.2)
*Guard interval (GI): 0 means normal guard interval (800 ns), 1 means short guard interval (400 ns)

Table 6.1: Information of the selected test I/Q files with different protocols, data rate, packet length, number of packets, and noise.

MCS index	Modulation type	Coding rate	Data rate (Mbit/s) with 20 MHz channel	
			800 ns GI	400 ns GI
0	BPSK	1/2	6.5	7.2
1	QPSK	1/2	13	14.4
2	QPSK	3/4	19.5	21.7
3	16-QAM	1/2	26	28.9
4	16-QAM	3/4	39	43.3
5	64-QAM	2/3	52	57.8
6	64-QAM	3/4	58.5	65
7	64-QAM	5/6	65	72.2

Table 6.2: Modulation and coding scheme with one Spatial streams Wikipedia contributors (2022)

required two times the number of clock cycles to operate. There was also a trailing length of 6 bits. To summarize,

$$\begin{aligned} &Delay\ (theoretical) \\ &= 2 \times (window\_length + acquisition\_length) + 6 \\ &= 216\ clock\ cycle \end{aligned}$$

The simulation results were shown in figure 6.1(a). The moment when the last bit entered the decoder was when both $conv\_in0\_dly$ and $conv\_in1\_dly$ signals were set to 3, as shown by the blue marker. Moreover, the last decoded bit moment was when the $dot11\_state$ signal was converted from *c* to 0, as shown by the yellow marker. Since the clock was 100 MHZ ($10ns$),

$$\begin{aligned} &Delay\ (simulation) \\ &= \frac{(924.585 - 922.455)10^{-6}}{10^{-9}} \\ &= 2130\ ns = \frac{2130}{10} = 213\ clock\ cycles \end{aligned}$$

As observed in figure 6.1(b), the pipeline acquires one bit per clock cycle and outputs one bit per clock cycle. Therefore the theoretical throughput of the decoder is equal to the clock frequency. To evaluate the actual throughput, the RTL design was synthesized with different clock constraints to find the maximum operational clock frequency for the design. The physical limitation is driven by the crit-

File No.	Length	Decoder	SNR			
			7	8	9	10
1	14	Xilinx	*N/A	0	0	0
		Ported	N/A	0	0	0
	1537	Xilinx	N/A	0	0	0
		Ported	N/A	0	0	0
	4000	Xilinx	-	N/A	0	0
		Ported	-	N/A	0	0

File No.	Length	Decoder	SNR									
			10	11	12	13	14	15	16	17	18	20
2	14	Xilinx	0.466	0.474	0.338	0.376	0.323	0.120	0	0	-	-
		Ported	0.391	0.353	0.338	0.158	0.226	0.120	0	0	-	-
	1537	Xilinx	0.493	0.478	0.458	0.417	0.327	0.187	0.084	0.027	0.002	0
		Ported	0.494	0.480	0.459	0.419	0.330	0.197	0.106	0.041	0.006	0
	4000	Xilinx	0.485	0.465	0.452	0.412	0.332	0.255	0.118	0.049	0.021	0.0006
		Ported	0.483	0.468	0.450	0.413	0.340	0.263	0.139	0.071	0.026	0.001

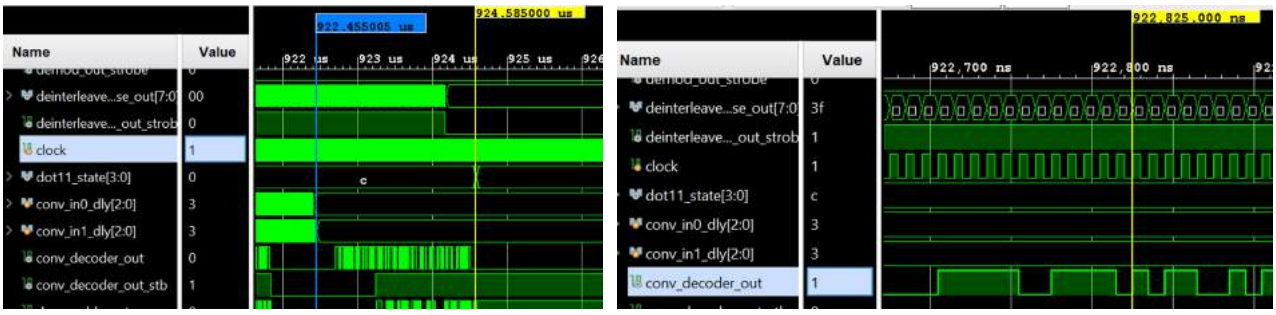
File No.	Length	Decoder	SNR						
			5	6	7	8	9	10	15
3	14	Xilinx	N/A	0	0	0	0	0	-
		Ported	N/A	0	0	0	0	0	-
	1537	Xilinx	-	-	-	N/A	0	0	-
		Ported	-	-	-	N/A	0	0	-
	4000	Xilinx	-	-	-	-	N/A	0	0
		Ported	-	-	-	-	N/A	0	0
4	14	Xilinx	N/A	0	0	0	-	-	-
		Ported	N/A	0	0	0	-	-	-
	1537	Xilinx	-	-	-	N/A	0	0	-
		Ported	-	-	-	N/A	0	0	-
	4000	Xilinx	-	-	N/A	0	0	0	-
		Ported	-	-	N/A	0	0	0	-

File No.	Length	Decoder	SNR									
			10	15	16	17	18	19	20	22	25	30
5	14	Xilinx	0.424	0.508	0.447	0.326	0.061	0	0	0	0	-
		Ported	0.470	0.470	0.454	0.235	0.061	0	0	0	0	-
	1537	Xilinx	0.502	0.476	0.465	0.436	0.404	0.328	0.226	0.041	0.002	-
		Ported	0.497	0.472	0.466	0.435	0.404	0.356	0.263	0.074	0.008	-
	4000	Xilinx	0.495	0.471	0.452	0.413	0.351	0.258	0.153	0.024	0.0007	-
		Ported	0.494	0.472	0.460	0.428	0.364	0.283	0.192	0.048	0.003	-
6	14	Xilinx	0.477	0.508	0.500	0.500	0	0	0	0	0	-
		Ported	0.500	0.508	0.470	0.348	0.068	0	0	0	0	-
	1537	Xilinx	0.499	0.470	0.459	0.400	0.331	0.230	0.136	0.022	0.0005	0
		Ported	-	0.469	0.448	0.414	0.363	0.280	0.193	0.036	0.002	-
	4000	Xilinx	0.495	0.473	0.454	0.410	0.338	0.257	0.148	0.017	0.0001	-
		Ported	-	0.469	0.448	0.414	0.363	0.289	0.193	0.036	0.002	-

N/A : The noise is so large that the system is not able to detect the header of the packets. No input or output is applicable.

*- : No need for simulation. The adjacent test points proved that this data should be either totally wrong (BER = 0.5) or totally correct (BER = 0)

Table 6.3: BER Comparison between the ported decoder and the commercial Xilinx Viterbi decoder 7.0



(a) (b)

Figure 6.1: simulation results of the ported Viterbi decoder in $openofdm\_rx$ module

Indicators	Ported viterbi decoder
Max Throughput (Mbps)	204
delay (data cycles)	213

Table 6.4: Summary of the delay and throughput

Data rates (Mbps)	Output length (in bits)	Bits Error
6.5	9344	0
48	8728	0
52	9600	6 (0.06%)
58.5	7728	5 (0.06%)
65	10192	303 (2.97%)

Table 6.5: Information of the simulation files with real-life recording

ical paths in the processing pipeline. Figure 6.2 showed that the clock constraint of worst negative slack was not satisfied; the maximum frequency of the decoder IP was thus limited to 204MHz.

6.2 Accuracy and Noise resilience

Firstly, the noisy simulation input files were generated by the Adding Gaussian White Noise (AWGN) function in MATLAB which was based on SNR to the selected input I/Q files (Table 6.1). The SNR values were chosen as [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20] for the files No. 1 and No. 2 as shown in Table 6.1 and [7, 8, 9, 10, 15, 16, 17, 18, 19, 20, 22, 25, 30] for files No. 3 to No. 6. Then, the SNR-to-BER graph was plotted in MATLAB. The BER was generated by comparing each bitstream output from the Viterbi decoder in the noisy case with the ideal case.

The simulation results (Table 6.3) were plotted in file number categories (figure 6.3), where the BER were plotted logarithmically. The more the line lies to the bottom left corner, the better performance the decoder has. Some of the lines were not plotted through the whole x-axis. Some of the lines were not plotted through the whole x-axis because they were truncated when the BER of the next sampled SNR was zero, which means that its logarithm was negative infinity.

Table 6.5 shows the error of the ported Viterbi decoder. Because the simulation files were real-life recordings with noise, decoding error shows while the data rate is over 48M.

6.3 Energy and Area consumption

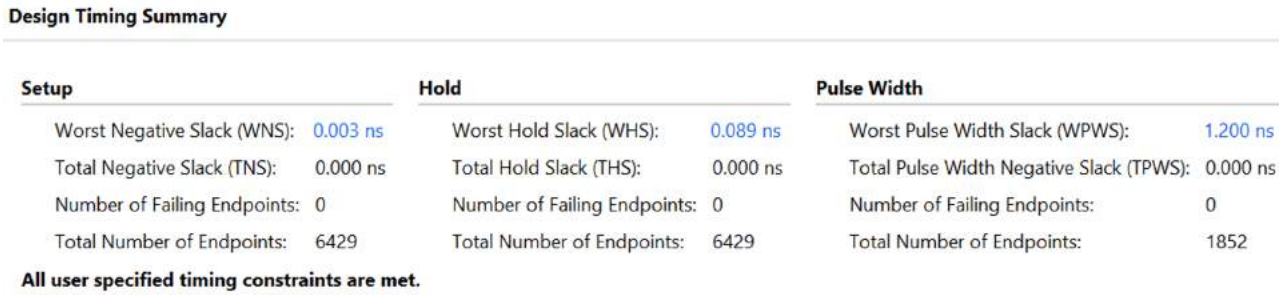
Through the utilization reports in the synthesis, the energy and area consumption of the Viterbi decoder were shown in Table 6.6; 100MHz was selected as the clock frequency. Also, the commercial IP core, Xilinx's Viterbi decoder v7.0, was observed as a reference for comparison. It was set to the same settings as the ported Viterbi decoder, with constraint length 7, output rate $\frac{1}{2}$, traceback depth 96, soft decode width 4, and parallel structure Xilinx (2011).

Part		Ported Viterbi decoder	Xilinx Viterbi decoder
Area	LUT	2595	2102
	LUTRAM	0	14
	FF	1083	1714
	BRAM	4	4
	IO	21	22
Clock Freq(MHz)		100	100
Power (mW)		210	140

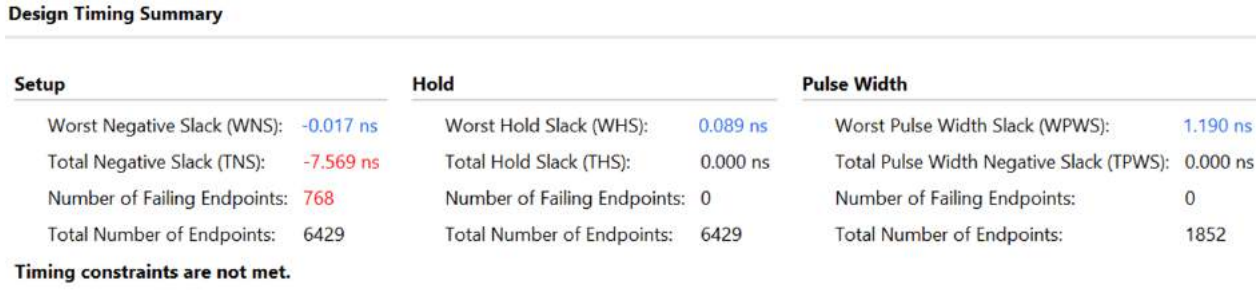
Table 6.6: Consumption comparison between the ported decoder and the commercial Xilinx Viterbi decoder 7.0

6.4 Validation on the board

As a final step of verification, the ported Viterbi decoder was integrated into the whole OpenWiFi project, and then simulated, synthesized, bit-stream generated, and verified



(a) clock frequency is 204MHz



(b) clock frequency is 205MHz

Figure 6.2: Design Timing Summary of ported Viterbi deocder

Ported	Utilization	Available	Utilization %
LUT	35474	53200	66.68
LUTRAM	1829	17400	10.51
FF	48691	106400	45.76
BRAM	107	140	76.43
DSP	108	220	49.09
IO	123	200	61.50
MMCM	1	4	25.00
Xilinx	**Utilization**	**Available**	**Utilization %**
LUT	34050	53200	64.00
LUTRAM	1332	17400	7.66
FF	48989	106400	46.04
BRAM	111	140	79.29
DSP	108	220	49.09
IO	123	200	61.50
MMCM	1	4	25.00

Table 6.7: Memory consumption of the whole OpenWIFI project integrated with the ported decoder and Xilinx Viterbi decoder 7.0

on the ZedBoard. It was also compared with the proprietary Xilinx one. Table 6.7 show the utilization report in the implementation step. Figure 6.4 shows the corresponding device view.

The on-board verification was performed as follows: First, the hardware design files were generated on a PC. Then transferred them via Ethernet to an SD card on the ZedBoard and used commands to handshake with the ZedBoard and allow Wi-Fi access. Although we detected the OpenWiFi's signal on the mobile phone, it showed network rejection when we tried to connect to it. Possible solution ideas were discussed in section 7.2.

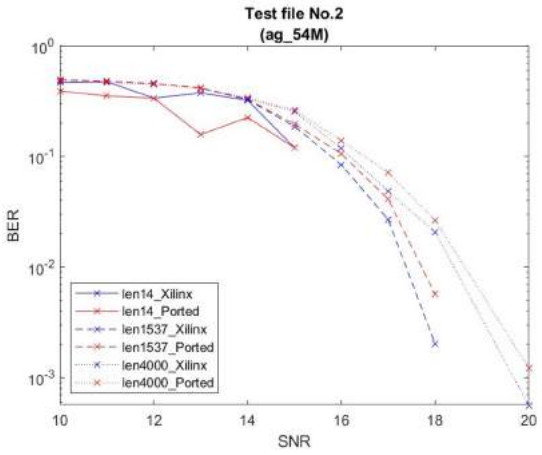
7 DISCUSSION

7.1 Discussion of this thesis

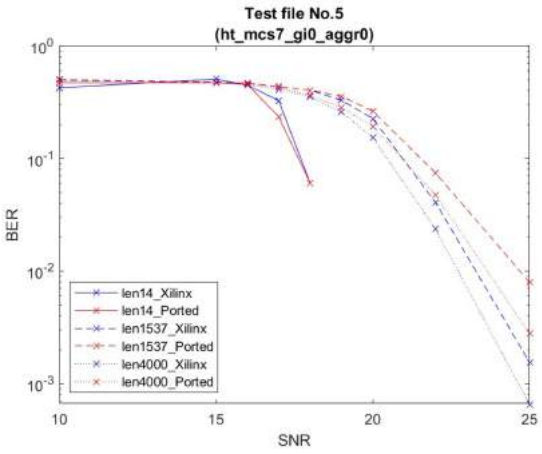
The discussion section included two aspects. First, discussed whether the performance of the ported Viterbi decoder met the requirements of the IEEE 802.11n standard and OpenWiFi. Then, compared it with the proprietary Xilinx IP core. The second part considered its significance as an open source hardware design.

For delay and throughput, the ported Viterbi decoder met the requirements in IEEE 802.11n. The delay (2.13 μs) was much smaller than the requirement (10 μs). The throughput was 204 Mbps, much larger than the required 72.2 Mbps.

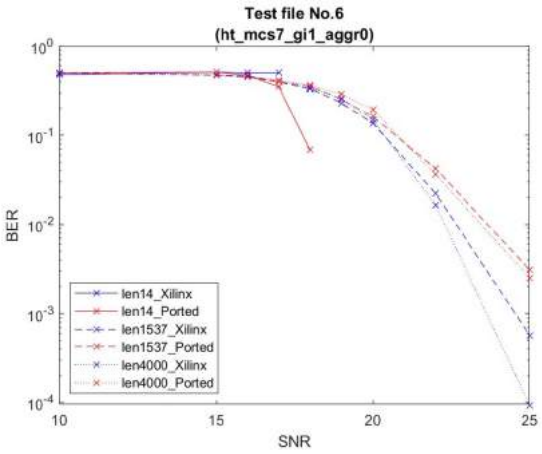
For noise resilience, the simulation results were compared with Xilinx's Viterbi decoder v7.0. According to Table 6.1, files 1, 3, and 4 with BPSK modulation type and low data rates (6, 6.5, 7.2 Mbps, respectively) were entirely correct (BER is always zero). Figure 6.3 showed that the BER of the proposed Viterbi decoder was always similar to Xilinx one under different simulations, including protocols, data rates, data lengths. In other words, the ported Viterbi decoder has comparable noise resilience with the Xilinx one.



(a) Test file No. 2

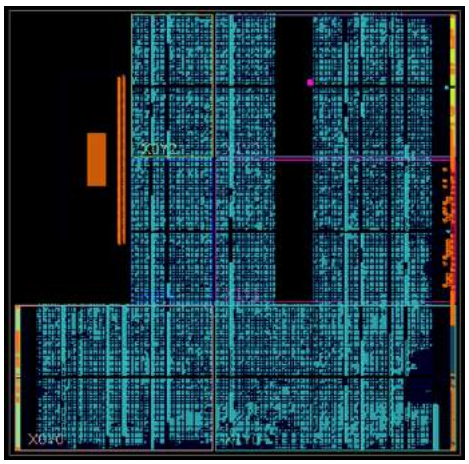


(b) Test file No. 5

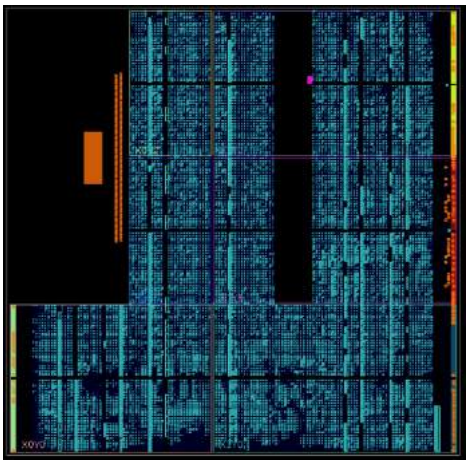


(c) Test file No. 6

Figure 6.3: SNR-BER plots of the ported Viterbi decoder in $openofdm\_rx$ modeule by MATLAB(file based)



(a) with ported Viterbi decoder



(b) with Xilinx Viterbi decoder

Figure 6.4: Device View of the OpenWiFi with Viterbi decoder

For area and energy consumption, Figure 6.4 presented that the ported Viterbi decoder had a larger area and higher energy consumption than the Xilinx one. It indicated that the ported one still has room for improvement.

For flexibility, thanks to the de-punching function, the ported Viterbi decoder supported various protocols and data rates mentioned in Table 6.1.

In a nutshell, the ported Viterbi decoder complied with the requirements of the IEEE 802.11n standard and the OpenWiFi. Simulation results showed it was compatible with the rest of OpenWiFi and obtained the correct decoding results, which was a step forward in addressing the limitations imposed by the Xilinx license.

Unlike the related studies, the highlight of this project was reflected in the open source hardware design. Closed source design was a massive obstacle for researchers. Building hardware platforms from scratch was time-consuming, and using commercial IP cores was costly. It was not conducive for freelancers to explore new theories and implementations. After porting open source Viterbi

decode in OpenWiFi, it will enable easier experimentation to try out new ideas and further develop the open source community.

7.2 Limitations and Future work

As mentioned in 6.4, although the simulation results showed that the ported Viterbi decoder decoded correctly, it did not work properly when integrated into OpenWiFi and loaded on the FPGA. It needed more time to debug and address the problem.

Although the performance met OpenWiFi system basic decoding requirements, the referenced overall architecture of the Viterbi decoder was released in 2014. The chosen parallel traceback structure was published in 2008, and the structure algorithms of other modules were even earlier. Many advanced architectures and algorithms can further improve the performance of decoders in terms of delay, throughput, area and energy consumption. Some of them were listed below as future research directions for the proposed Viterbi decoder.

Forward Traceback Method

Zhao et al. proposed a new forward traceback method, which combined the backward (TB) and forward (RE) methods together Zhao et al. (2017). The target started traceback state was obtained by forwarding recursive computing, which meant the acquisition phase was no longer needed. Compared with TB, such a structure reduced delay by 25%. Furthermore, the storage space (shift registers) paid in this way was far less than RE. This compromise method was suitable for application scenarios where delay and memory were restricted.

T-algorithm

Surya et al. proposed a design of a Viterbi decoder based on the T-algorithm Surya et al. (2021). The T-algorithm generated a threshold value and removed the redundant states from the ACSU for the next cycle. The area and power consumption were reduced by 20.90% and 18.18% for the decoder with a $\frac{1}{2}$ rate.

Moreover, the throughput of the Viterbi decoder highly relies on the maximum clock frequency, which is driven by the length of the critical paths in the pipeline, and to a lesser extent, the constraints of combinational circuits of a specific board. Utilizing more advanced boards might increase the decoding speed marginally. However, most improvement would come from analyzing the critical paths in the decoder's internal design and adding buffer registers as needed. Currently, OpenWiFi supports the following seven boards, as presented in Table 7.1, after only four years of development. It is reasonable to believe that more boards will be supported as this open source project continues to grow. Subsequently, further corresponding simulation and synthesis should be conducted to ensure the decoder functions properly. The structure can be further adjusted for boards with more significant memory and application regardless of the power consumption.

8 CONCLUSION

OpenWiFi is an open source Software Defined Radio (SDR) implementation on IEEE 802.11 (Wi-Fi) standard. Currently, the paid commercial IP core of Xilinx's Viterbi decoder v7.0 undertook the decoding work, which was restrictive and cumbersome for researchers. As a result, this article ported a parallel Viterbi decoder with the traceback algorithm. It utilized the linear distance for a three-bit soft decision. Furthermore, it supported the de-puncturing capabilities for the OpenWiFi, which offered flexibility and compatibility with different coding and modulation techniques and Wi-Fi standards.

This thesis first introduced the related background theories and analyzed the project requirements and indicators. Second, the structure and function of modules of the Viterbi decoder were explained. Third, details of the decoder's implementation methods were illustrated. Finally, the decoder was subsequently embedded into the OpenWiFi project and evaluated on the zed evaluation board. The simulation and synthesis results demonstrated that the proposed Viterbi decoder architecture was compliant with the IEEE 802.11a/g/n standard and performed comparably to the currently used Xilinx's Viterbi decoder v7.0, warranting a compatible replacement. Because of the pipeline architecture, the throughput of the decoder is equal to the operational clock frequency. The delay of the pipeline is 213 clock cycles, resulting in a 2.13 μs delay and 100 Mbps throughput at the target 100MHz frequency. Finally, the ported Viterbi decoder was integrated as part of the complete project and prototyped on the target FPGA platform. However, the final integration on board was not fully functional. This was probably because the interfaces were changed in the replacement, which might break the handshakes between the decoder and the rest of the system. To solve this problem, further debugging in future work is required.

The highlight of this project was shown in the open source hardware design. All the related code was released open source to the developer community. This work could enable more researchers to test new ideas in Wi-Fi systems and may spark other open source projects.

Name	Details	Vivado license
zc706_fmcs2	Xilinx ZC706 board + FMCOMMS2/3/4	Need
zed_fmcs2	Xilinx zed board + FMCOMMS2/3/4	No Need
adrv9364z7020	ADRV9364-Z7020 + ADRV1CRR-BOB	No Need
adrv9361z7035	ADRV9361-Z7035 + ADRV1CRR-BOB/FMC	Need
zc702_fmcs2	Xilinx ZC702 board + FMCOMMS2/3/4	No Need
antsdr	MicroPhase enhanced ADALM-PLUTO SDR	No Need
zcu102_fmcs2	Xilinx ZCU102 board + FMCOMMS2/3/4	Need

Table 7.1: SDR platforms supported by OpenWIFI

ACKNOWLEDGEMENTS

Baiming Zhang: I would first like to thank Professor Josep Balasch and Dr. Xianjun Jiao for providing us the opportunity to partake in this hugely interesting topic. Thank you to Maxim Yudayev for the continuous support and feedback on the experiments and thesis. Thank you to Yingshuo for spending an unforgettable time together as a handsome teammate. Last but not least, I would like to thank my parents for their support over the years and my girlfriend's companionship and love; her believing in me motivates me to persevere.

Yingshuo: I would like to thank my parents for their unconditional support and trust in me during my study aboard. Thanks Dr. Xianjun Jiao, for assigning us this meaningful topic. He and his colleague Dr. Michael were very patient in answering us many questions during the process. I would like to thank Prof. Josep Balasch and Mr. Maxim Yudayev for their great help and for spending quite a lot of time guiding us on this thesis. Also, I want to thank my teammate Baiming Zhang for being a terrific partner and friend. His open-minded and progressive attitude keeps the project organized and productive. What's more, I'd like to thank my best friend, Wangdong Guo. I'm glad you showed up in my life and grateful for what we have encountered so far. Last but not least, I would like to dedicate this thesis to my grandma, who has committed a lot to my upbringing and who would have been immensely proud.

BIBLIOGRAPHY

Abdallah, R. A. and Shanbhag, N. R. (2009). Error-resilient low-power viterbi decoder architectures. *IEEE Trans. Signal Process.*, 57(12):4906–4917.

Angarita, F., Canet, M. J., Sansaloni, T., Valls, J., and Almenar, V. (2008). Architectures for the implementation of a OFDM-WLAN viterbi decoder. *J. Signal Process. Syst.*, 52(1):35–44.

Banerjee, A., Lenz, A., and Wachter-Zeh, A. (2022). Sequential decoding of convolutional codes for synchronization errors.

Baptista, M. S. (2021). Chaos for communication. *Nonlinear Dyn.*, 105(2):1821–1841.

Chang, S.-H., Cosman, P. C., and Milstein, L. B. (2011). Performance analysis of n-channel symmetric FEC-based multiple description coding for OFDM networks. *IEEE Trans. Image Process.*, 20(4):1061–1076.

Cypher, R. and Shung, C. B. (2002). Generalized trace back techniques for survivor memory management in the viterbi algorithm. In *[Proceedings] GLOBECOM '90: IEEE Global Telecommunications Conference and Exhibition*. IEEE.

Dizdar, O. (2017). *High throughput decoding methods and architectures for polar codes with high energy-efficiency and low latency*.

Dong-Sun Kim, D.-S., Seung-Yerl Lee, S.-Y., Kyu-Yeul Wang, K.-Y., Jong-Hee Hwang, J.-H., and Duck-Jin Chung, D.-J. (2007). Power efficient viterbi decoder based on pre-computation technique for portable digital multimedia broadcasting receiver. *IEEE trans. consum. electron.*, 53(2):350–356.

Forney, D. (1972). Maximum-Likelihood seauence estimation of digital sequences i&he presence of intersymbol interference G. *Presence of Intersymbol Interference G*.

Ganeshkumar, S. and Rangasamy, K. S. (2013). Design of 2x2 mimo OFDM architecture for fixed WIMAX. `http://www.periyaruniversity.ac.in/ijcii/issue/Vol3No1June2013/IJCII%203-1-97.pdf`. Accessed: 2022-5-24.

Gazi, O. (2020). *Forward error correction via channel coding*. Springer Nature, Cham, Switzerland, 1 edition.

He, J., Liu, H., Wang, Z., Huang, X., and Zhang, K. (2012). High-speed low-power viterbi decoder design for TCM decoders. *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, 20(4):755–759.

IEEE (2008). IEEE standard for information technology - telecommunications and information exchange between systems - local and metropolitan networks - specific requirements - part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications: Higher speed physical layer (PHY) extension in the 2.4 GHz band. Technical report, IEEE, Piscataway, NJ, USA.

Jiao, X., Liu, W., Mehari, M., Aslam, M., and Moerman, I. (2020). openwifi: a free and open-source IEEE802.11 SDR implementation on SoC. 2020 IEEE 91st vehicular technology conference (VTC2020-Spring). pages 1–2.

Kamuf, M., Owall, V., and Anderson, J. B. (2007). Survivor path processing in viterbi decoders using register exchange and traceforward. *IEEE Trans. Circuits Syst. II Analog Digit. Signal Process.*, 54(6):537–541.

Lou, H.-L. (2002). Linear distances as branch metrics for viterbi decoding of trellis codes. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*. IEEE.

Ma, C. (2010). Digital communications III. https://www.eee.hku.hk/%7Esdma/elec7073/. Accessed: 2022-5-24.

Matthias, A. and Markus, F. (2012). Viterbi decoder (AXI4-Stream compliant). https://opencores.org/projects/viterbi_decoder_axi4s. Accessed: 2022-5-24.

Mizuochi, T. (2006). Recent progress in forward error correction and its interplay with transmission impairments. *IEEE J. Sel. Top. Quantum Electron.*, 12(4):544–554.

Mohammadidoost, A. and Hashemi, M. (2020). High-throughput and memory-efficient parallel viterbi decoder for convolutional codes on GPU.

Nidecki, T. A. (2020). What is integer overflow. https://www.acunetix.com/blog/web-security-zone/what-is-integer-overflow/. Accessed: 2022-5-24.

Perahia, E. (2008). IEEE 802.11n development: History, process, and technology. *IEEE Commun. Mag.*, 46(7):48–55.

Plosila, J. (2005). Asynchronous viterbi decoder in action systems. https://www.researchgate.net/publication/31595950_Asynchronous_Viterbi_Decoder_in_Action_Systems. Accessed: 2022-5-24.

Putra, R. V. W., Microelectronics Center, Institut Teknologi Bandung, Indonesia, Adiono, T., and Microelectronics Center, Institut Teknologi Bandung, Indonesia (2016). VLSI architecture for configurable and low-complexity design of hard-decision viterbi decoding algorithm. *J. ICT Res. Appl.*, 10(1):57–75.

Solomon, G. and Stiffler, J. J. (1965). Algebraically punctured cyclic codes. *Inf. Contr.*, 8(2):170–179.

Sun, Y. and Ding, Z. (2012). FPGA design and implementation of a convolutional encoder and a viterbi decoder based on 802.11a for OFDM. *Wirel. Eng. Technol.*, 03(03):125–131.

Surya, R., Balasubramanian, K., and Yamuna, B. (2021). Design of a low power and high-speed viterbi decoder using t-algorithm with normalization. In *2021 International Conference on Advances in Computing and Communications (ICACC)*. IEEE.

Tomlinson, M., Tjhai, C. J., Ambroze, M. A., Ahmed, M., and Jibril, M. (2017). *Error-correction coding and decoding*. Springer International Publishing.

Wikipedia contributors (2022). IEEE 802.11n-2009. https://en.wikipedia.org/w/index.php?title=IEEE_802.11n-2009&oldid=1083738461. Accessed: NA-NA-NA.

Xilinx (2011). Viterbi decoder v7.0. https://docs.xilinx.com/v/u/en-US/viterbi_ds247. Accessed: 2022-5-24.

Zhao, X., Li, H., and Wang, X. (2017). A high performance multi-standard viterbi decoder. In *2017 7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC)*. IEEE.