8-30-2011

A real-time, reconfigurable system for energy, error-resilient, and scalable lossless ECG coding

Paul Essenmacher

Paul Essenmacher
Candidate

Electrical and Computer Engineering
Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Marios Pattichis Μάριος Παττίχης , Chairperson

Howard Pollard Howard Pollard

Payman Zarkesh-Ha Payman Zarkesh

A REAL-TIME, RECONFIGURABLE SYSTEM FOR ENERGY, ERROR-RESILIENT, AND SCALABLE LOSSLESS ECG CODING

BY

PAUL ESSENMACHER

B.S., Electrical Engineering, University of New Mexico, 2008

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science in Electrical Engineering

The University of New Mexico
Albuquerque, New Mexico

August, 2011

ACKNEWLEDGEMENTS

I would like thank my advisor and thesis chair, Dr. Marios S. Pattichis, for his encouragement and teaching through classroom learning and the months of developing and writing this thesis. His passion for his work overflows into his interactions with his students and provided motivation and encouragement through this whole process.

I would also like thank Dr. Payman Zarkesh-Ha, and Dr. Howard Pollard for being a part of my thesis committee and for their valuable critiques and recommendations pertaining to this study.

And most importantly, I would like to thank my wife, Susan Essenmacher, for her unconditional love and support throughout this endeavor, and for her understanding and patience when I had to be gone long hours. Also, to my three sons, Ethan, Micah, and Levi, thank you for making me laugh and for asking daddy to “stay home”, and providing a constant reminder to take an occasional break and spend time with my family. I love you all.

A REAL-TIME, RECONFIGURABLE SYSTEM FOR LOW-ENERGY, ERROR-RESILIENT, AND SCALABLE LOSSLESS ECG CODING

by

Paul Essenmacher

B.S., Electrical Engineering, University of New Mexico, 2008
M.Sc., Electrical Engineering, University of New Mexico, 2011

ABSTRACT

Electrocardiogram (ECG) monitoring systems have evolved to the point where they are now portable and can monitor the patient 24/7 and transmit alerts and ECG data to parents and doctors as soon as a heart irregularity is detected. With the advances in these systems, there is a need for the incorporation of ECG coding systems to reduce the bandwidth used when data is transmitted and to incorporate methods to provide data recovery in the event of a transmission error. However, while ECG encoding systems for hospital or home care settings has been thoroughly researched, the application of ECG encoding systems to portable ECG monitoring systems where there is a much higher likelihood of noise interference during transmission of the data has not been fully investigated.

The goal of this work is to develop a real-time ECG encoding system that requires low hardware and power usage, provides lossless signal compression, and provides recovery of as much data as possible in the event of data corruption of packets during transmission. An entropy based compression algorithm is developed based on the

Huffman code which is then transformed to reversible variable length codes. This allows the data packets to be both frontward and backwards decodable allowing for greater data recovery in the event that portions of a packet are corrupted. The implementation is designed to be able to encode any sized bit width by utilizing a combination of 4, 6, or 8-bit entropy coders.

Two separate encoding systems are investigated using the before mentioned encoding algorithm. The first system recomputes the Reversible Variable Length Code (RVLC) tables periodically while the signal is being encoded in an effort to adapt to any changes in the signal. The second system uses a pre-calculated RVLC table that minimizes the delay and also significantly reduces the required hardware resources. We provide optimal, reconfigurable implementations for both systems. Furthermore, the effectiveness and error-resilient performance of both systems are validated on 12-bit and 16-bit ECG signals. The performance of the system is shown to be diagnostically lossless in noisy communications channels with significant bit errors. This represents a significant improvement over existing systems that do not employ the proposed error resilient encoding methods.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.0 Overview

Cardiac related health problems can occur at any age. The elderly and middle-aged alike can experience debilitating or fatal heart attacks; even young children are often the victims of heart related problems such as cardiac arrhythmias, which if not detected and treated in time can lead to tragic deaths. Especially troubling are the cases where a child will experience an arrhythmia, but by the time the child is taken to a doctor the symptoms have subsided and the doctor has no way of diagnosing the problem and getting the child the care that is needed. It is not feasible to keep a child under constant monitoring in a medical facility for days or weeks, but if the child never experiences an arrhythmia while being monitored then a correct diagnosis of the problem often cannot be made. Often a child will experience multiple arrhythmias that are days or weeks apart, until finally a severe enough arrhythmia proves fatal.

In recent years this problem has spurred the development of mobile ECG monitoring systems which can be carried by the patient to provide monitoring outside of a medical facility. One of the first such systems is the Holter monitor which can provide constant recording of ECG data, but is limited to 24 or 48 hours of recording. As technology advances, communications and computer system capabilities have become more efficient, smaller, and come at lower costs. A recent system that has taken advantage of these advanced capabilities is presented in [1]. It is designed to provide

constant ECG monitoring with automatic analysis of the ECG data either continuously, or at frequent intervals. Based on this analysis the monitoring system will transmit data to a doctor when a cardiac irregularity occurs. Other similar monitoring systems are presented in [2], [3], [4].

Analysis of the ECG data is vital to these monitoring systems as they must be able to determine whether the signal is normal, and what should be transmitted for analysis by a trained cardiologist. Figure 1.1 shows a single ECG beat broken down into the segments that are evaluated during analysis of the signal. Any abnormalities seen in the shape of the waves, the value of the voltage, or incorrect lengths in the time domain of the individual waves indicate that a cardiac irregularity may have occurred and that the data surrounding this event should be saved for further analysis.

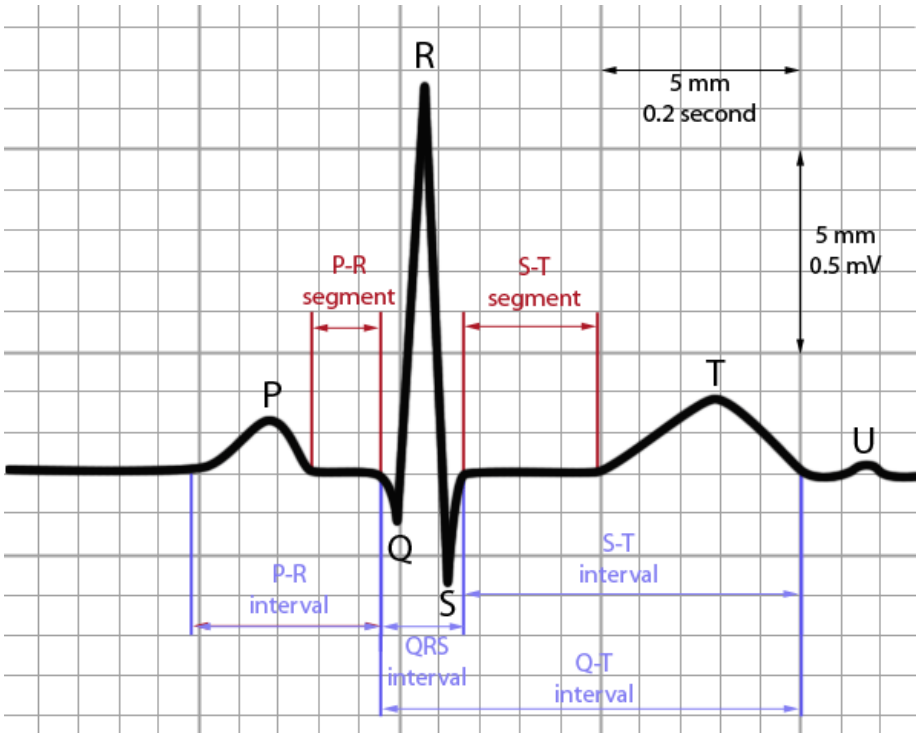


Figure 1.1: Single beat of an ECG signal showing critical points, intervals and segments [5]

One of the big components of the recent ECG monitoring systems is the ability to transmit the relevant ECG data over wireless channels thereby reducing or eliminating the need to store data in the monitoring system itself. This however introduces the possibility of data corruption or loss during transmission. Since the data being transmitted is likely to contain information vital to diagnosing and treating a cardiac irregularity, it is important to reduce the possibility of data loss as much as possible. Since monitoring systems are increasingly using wireless transmission features to stream the data they are collecting to a static location, incorporating ECG compression architecture with the monitoring system would allow for the data to be transmitted in a more efficient manner, and would also allow for the possibility of incorporating features such as identification and recovery from transmission errors.

In this thesis a low energy, error resilient, scalable, lossless ECG compression architecture is presented. The compression architecture is tested against selected signals from the PhysioBank Physiologic Signal Archives [6] which is the standard ECG database used to test compression systems that are targeting ECG signals.

1.1 Innovations and Contributions

A summary of the primary innovations and contributions includes:

- Signal-bitwidth scalable compression
    - Effectively demonstrated for 12 and 16 bit lossless ECG compressing by segmenting signal samples into smaller segments, and then recombining after decoding.

- Optimal implementations running in real-time, requiring a low amount of hardware resources, and low power usage

- Error-resilient hardware encoding and software decoding:

    - Optimal reversible variable length coding (RVLC) with minimal additional overhead (~7%, [49], [46])

    - Frontward and backward decoding

    - Sync markers at the packet level

- Allowing correct diagnosis of signals within certain levels of data corruption:

    - Interpolation of corrupted or lost data samples through new methods

Chapter 2

Related Work

2.0 Common Compression Techniques

Techniques for compressing ECG data can be broken down into three main types: direct and transform methods [7], and a wide range of other methods that require the signal to be pre-processed before encoding in order to extract certain information [8]. Direct methods perform compression on the signal in the time domain and include Huffman, arithmetic, Golomb, and differential pulse code modulation, techniques among others. Transform methods, on the other hand, transform the data to another domain and then compress the signal. The Fourier, wavelet, discrete cosine (DCT), Karhunen-Loeve (KL), Burrows-Wheeler, and Walsh are the main transforms used in ECG compression. The other methods in the third compression type that would not fall under the direct or transform methods generally have the characteristic that the signal must be pre-processed in order to extract important information. Information that would be extracted includes parameter extractions such as heart-beat averaging, long-term prediction, vector quantization, and subband decomposition (except for those based on transform methods) where spectral information is split into individual signals that can be individually encoded using a desired technique [8].

2.1 Lossy 2-D Compression Methods

One of the more recent trends in ECG compression has been to utilize lossy 2-D compression methods where the ECG signal is cut and aligned in 2-D “image” data

arrays, taking advantage of the fact that ECG signals show redundancy between successive beats and samples. An argument for 2-D compression methods to take advantage of the correlation in ECG signals is discussed in [9]. Using this 2-D array configuration allows the use of various algorithms that are particularly adept at exploiting correlation in 2-D array data. Several researchers utilize wavelet transform based methods, including 1-D and 2-D wavelet transforms in [10] and [11]. Filho *et. al.* presents a method incorporating DC equalization and complexity sorting in [12]. Chou *et. al.* converts 1-D ECG signals to 2-D representations, and then utilizes the image compression scheme JPEG2000 to encode the 2-D representation [13].

2.2 Lossy Transform Compression Methods

Over the past decade or so, lossy transform methods, especially the wavelet transform, have been the focus of a lot of research in the area of ECG signal compression [14-23]. Among these wavelet schemes are methods to provide quality on demand to meet a specified percent root-mean-square difference (PRD) [14],[15],[16], a wavelet based vector quantization approach [17], tree based quantization and encoding methods [18],[19],[20], a low delay compression scheme [21], and a scheme focusing on real time packetized cardiology [22]. A discrete cosine transform scheme is also presented in [23].

2.3 Lossy Non-Transform Compression Methods

Researchers have also explored a wide range of non-transform based methods for encoding ECG signals over the past decade as well [24-33]. Among these are a rate distortion optimal algorithm [24], a classification algorithm with quad level vector (QLV) [25], multiscale pattern matching [26], various filter structures [27],[28],[8], gain shape

[29] and adaptive [30] vector quantization, a high complexity multidimensional multiscale parser algorithm [31], an input delay neural network compression scheme [32], and a truncated singular value decomposition algorithm [33].

2.4 Lossless Compression Methods

Much of the work done on ECG compression techniques has focused on lossy compression since it yields compression results that are significantly better than by lossless methods. Most of the better lossy compression techniques result in compression ratios between 10:1 and 12:1, with some of the more recent implementations achieving average results up into the 18:1 to 20:1 range. However, this performance comes at a price as lossy compression schemes may lose critical information from the ECG signal. This can result in a loss of diagnostic ability since the lost information may pertain to critical features of the signal that could hinder an accurate diagnosis. Lossless compression of ECG signals is needed when comprehensive diagnostic ability is desired; however, very little work has been published in this area compared to the large amount of published work on lossy ECG compression.

Two of the most recently published works on lossless compression for ECG signals are multi-channel implementations, [34], [35]. A VLSI implementation using DPCM prediction, adaptive k-parameter estimation and Golomb-Rice entropy encoding and is implemented in 65 nm CMOS is presented in [34]. This implementation also has the ability to handle multiple input sample bit sizes. In [35], the MPEG-4 audio coding algorithm is implemented in software with the ability to handle only a single size of sample inputs. The implementations given in [36-41] are all single channel lossless

encoding algorithms. In [36] the author presents an algorithm based on K-means cluster based on the similarity measurement between samples. Inversion ranks of linear prediction and the Burrows-Wheeler transform are used in [37]. Several methods [38][39] use the entropy based Golomb-Rice encoding as the basis for their implementations; with [38] incorporating linear decorrelation from analysis of a block of samples, and [39] incorporating R-R interval estimation. [38] did incorporate error resiliency, but only at the block level. Two implementations based on the wavelet transform are presented in [17], which was initially a lossy scheme but can become lossless with the correct parameters being set, and [40] which is based on the lifting wavelet transform. Finally a linear block transform is used in [41]. A comparison of the lossless schemes is presented below in Table 2.1.

Author	Year	Ref.	Technique	Multi-Channel	Max CR	Avg. CR	Scala-bility	Error Resiliency	Hardware Imp.
Chua	2011	[34]	DPCM prediction with adaptive k-parameter estimation, and Golomb-Rice entropy encoding	Yes	-	2.50	Yes	No	VLSI Implementation
Kamamoto	2009	[35]	Enhanced codec software of the MPEG-4 Audio Lossless Coding international standard for compressing multichannel time-series signals	Yes	-	3.0	No	No	No
Zhou	2009	[36]	Lossless compression algorithm based on K-means cluster	No	3.90	3.20	No	No	No
Arnavut	2007	[37]	Algorithm based on Burrows-Wheeler transform and inversion ranks of linear prediction	No	-	4.525	No	No	No
Ottley	2005	[38]	Linear decorrelation selected from analysis of a block of samples, followed by Golomb-Rice coding.	No	3.13	2.67	No	Error resiliency at the block level	Soft-core CPU custom instruction for use in portable ECG monitoring
Miaou	2005	[17]	Wavelet based approach using 9/7 and 5/3 integer filters	No	3.75	3.068	No	No	No
Giurcâneanu	2001	[39]	*R–R* interval estimation with Golomb-Rice entropy encoding	No	4.41	3.071	No	No	No
Duda	2001	[40]	Lifting wavelet transform	No	3.722	3.156	No	No	No
Arnavut	2001	[41]	Linear transform algorithm (LTA) based on a new linear block transform (LBT)	No	~2.5	~2.22	No	No	No

Table 2.1: Comparison of lossless ECG encoding implementations

While there has been a variety of work done in lossless ECG coding, it is still far less than the amount of work done in lossy ECG coding. With less attention having been devoted to this area there are still areas that have not been adequately addressed, including:

- Incorporation of error resilient encoding techniques with the ability to recover significant amounts of data in the event of data corruption during signal transmission

- Adequate investigation of hardware implementation of ECG encoding algorithms with a focus on reducing the amount of hardware resources required and the amount of power consumed

- Implementations that can be easily scaled or reconfigured to accommodate various sized signal inputs

The work presented in [34] is the most recently published work on lossless ECG compression. The authors did allow for several different sizes of signal inputs, and also investigated a VLSI hardware implementation. But their implementation was focused on a broader scale design incorporating brain and heart monitoring. To truly be able to compare strictly ECG encoders implemented in hardware and their power requirements, there is a need for more investigation of ECG encoders implemented in FPGA/ASIC technology and the power required for those implementations. The work presented in [38] did incorporate error resiliency at the block level, but also chose large block sizes in order to reduce the amount of overhead incurred during transmission. Single bit errors during transmission could result in most or all of a block being lost since there is no provision for determining exactly where in the block the error occurred.

While there has been an increasing amount of investigation into lossless ECG coding methods in recent years, most of the work has been done without regard to the need for low power and hardware requirements, adaptability to various signal widths, and the need for error resiliency when transmitting data over error prone mediums. The implementation proposed in this thesis in an effort to broaden the scope of existing research in these areas.

Chapter 3

Methods

The compression algorithm presented in this thesis has the goal of providing compressed data for optimal data transmission, while providing the ability to reduce data loss as much as possible should an error occur during transmission of a data packet. There are other algorithms that would have provided better compression results, but these algorithms do not provide any way to minimize data loss when an error occurs. There are many applications that use data compression where a certain amount of data loss is acceptable; others, such as the transmission of ECG signals as discussed in this thesis, may require minimal data loss or else they run the risk of losing critical data. The following sections describe the individual methods used in the encoding system presented in this thesis.

3.0 Differential Pulse Code Modulation

$$e[n] = x[n] - x[n-1] \tag{1}$$

(1) shows the equation for calculating a Differential Pulse Code Modulation (DPCM) sequence, which is simply first order Linear Prediction (LP). DPCM is a method used to encode data by exploiting the correlation between successive samples of the input signal. Since the difference between successive samples is generally smaller than the samples themselves, the difference is encoded and normally requires fewer bits to represent it than are needed by the original samples. The block diagram of this implementation is shown in Figure 3.1(a).

$$e[n] = x[n] - \tilde{x}[n] \quad (2)$$

A natural extension of the DPCM operation is to predict the value of the current sample based on the previous *M* samples using LP, where the LP parameters are dynamically estimated. The equation for this is shown in (2) where $\tilde{x}[n]$ is the predicted value of the current sample, $x[n]$, and $\tilde{x}[n]$ is calculated using an *Nth* order linear predictor. The difference between the current sample and its predicted value, $e[n]$, is quantized with the hope that these difference values will have a lower entropy then the original signal . It is important to note that DPCM is the simplest form of linear prediction since it looks at the previous value, but it only uses a single coefficient with a value of one. Since the first and only coefficient in this case is always one, DPCM simply takes the difference between the current and previous sample. The coefficients of the LP model are determined by minimizing the error between the original and estimated signal in the least squares sense. The block diagram of the LP implementation is shown in Figure 3.1(b).

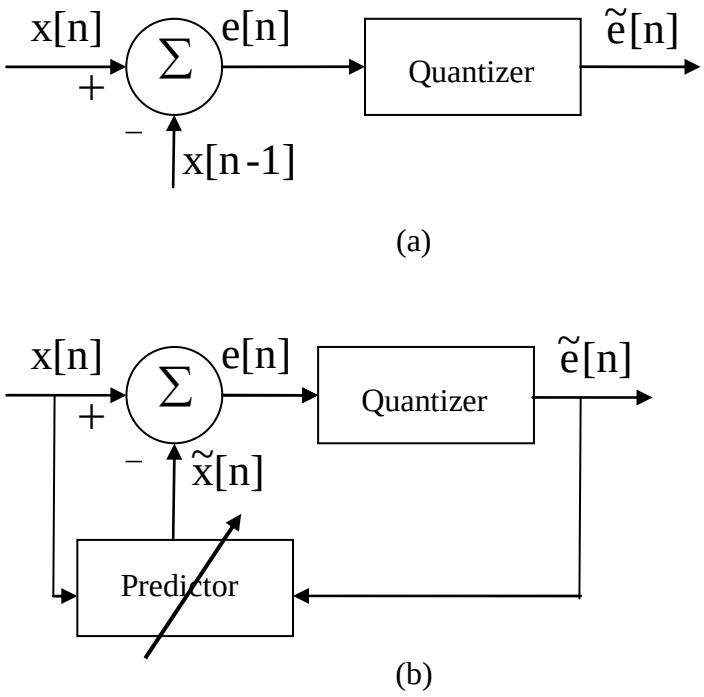


Figure 3.1: (a) DPCM – First order linear prediction model (b) Nth order linear prediction model

In both of the above DPCM systems, the difference sequence is normally quantized to a set number of bits. If the calculated difference between the current sample and the predicted value of the current sample is too large to be represented by the number of bits chosen for quantization, then data loss occurs. For this reason DPCM is normally considered to be a lossy coding scheme. However, if DPCM is not used strictly as a data compression tool, but rather as a tool to reduce the entropy of a signal so that an entropy coder can more effectively compress the signal, then it is not necessary to quantize the output of the DPCM to a smaller number of bits than that of the input. When used in this way the goal of the DPCM to reduce the entropy of the signal, so that a following entropy coder can achieve better compression results. Therefore, in order to keep the DPCM operation completely lossless, the output would need one more bit than the input, since

the DPCM generates some negative values, to guarantee that there will not be data loss. The actual compression of the data will be accomplished later by the entropy encoder.

ECG signals generally have a large amount of correlation between successive samples. In many cases, the ECG signal will have sequences that increase or decrease at a steady rate; the DPCM without linear prediction will encode each sample in this sequence with the same value since the difference between the samples is the same. This will lead to portions of data that can be effectively encoded using run-length coding. Additionally, in the implementation presented in this thesis the input ECG signals are broken up into smaller blocks, and then encoded and decoded individually. Therefore, very often the block containing the upper bits of the samples will be zero since only the lower bits are needed to represent the DPCM value. Run length encoding will be very effective in when this is the case.

The author in [37] discuss the fact that Jalaleddine *et. al.* [7] found that for ECG signals, a first order linear predictor (DPCM) yield better results than prediction models of higher orders, while [42] found, using a different set of signals, that a second order prediction model achieved the best entropy results. This research leads us to believe that due to the close correlation of sequential samples, that a lower order predictor will normally achieve better results than a higher order predictor.

This was tested and verified in software by running sets of 12-bit and 16-bit ECG signals through both the DPCM and LP implementations (Figure 3.1), and then computing the entropy of the output signals. As seen in Table 3.1 and Table 3.2, the implementation that only used single order linear prediction (DPCM) consistently

produced a lower entropy value than the implementation with linear prediction; this held true for both the 12 and 16 bit signals. This is a nice result as it allows the simpler DPCM implementation to be used which will require less hardware resources since a predicted value does not need to be calculated and the coefficients from the linear predictor do not need to be calculated and stored.

		Entropy Values				
ECG Signal	**Lead**	**Original Signal**	**LPC: # Coefficients**			
			1	**2**	**3**	**4**
418	ECG	8.7071	**5.0477**	5.9182	6.5669	7.0250
419	ECG	9.1869	**4.7987**	6.1033	7.0582	6.9955
420	ECG	9.6981	**4.6830**	5.0027	6.5445	6.0075
421	ECG	8.7012	**5.7515**	6.3500	7.4773	7.7677
422	ECG	8.8674	5.5145	**5.0846**	6.2938	7.0618
423	ECG	7.7288	4.5692	**4.4989**	6.2864	5.9547
424	ECG	9.5807	**4.4477**	4.5743	5.9273	5.7869
425	ECG	9.2784	**4.5721**	5.0516	6.2576	5.8598
426	ECG	9.5913	**4.4532**	4.6730	5.6604	5.6415
427	ECG	10.0566	**4.1904**	4.7187	6.2252	5.6052
428	ECG	9.5195	**4.2963**	4.8357	6.2831	5.7295
429	ECG	9.3842	**3.7853**	4.2939	5.6276	4.9651
430	ECG	8.9795	**3.8222**	4.7134	5.8945	5.3849
602	ECG	9.2163	**6.0324**	6.7347	7.5431	7.8941
605	ECG	7.3060	**5.0394**	5.9584	6.5863	6.7287
607	ECG	7.8574	**5.0523**	5.8967	6.8362	7.1261

Table 3.1: Comparison of entropy values of original, DPCM & DPCM with linear prediction for 12-bit ECG signals

ECG Signal	Lead	Entropy Values				
		Original Signal	DPCM with LPC: # Coefficients			
			1	2	3	4
I01	V1	11.0128	**5.2801**	5.8463	6.4793	6.6797
	V2	11.0571	**5.3527**	6.0115	6.7500	6.9486
I11	V1	9.0795	**3.2160**	4.1747	5.1246	4.8323
	V2	8.7161	**3.3185**	4.3287	5.2402	5.0432
I20	V1	8.8060	**4.0377**	4.6352	5.5899	5.5423
	V2	8.6443	**4.2152**	4.8425	5.7523	5.9156
I32	V1	9.7863	**3.5717**	4.6621	5.5638	5.5262
	V2	9.6871	**3.7507**	5.0098	5.8168	5.8321
I46	V1	10.9828	**5.9994**	6.2554	7.3305	7.7105
	V2	11.3890	**6.1716**	6.3882	7.4594	7.7784
I57	V1	11.3666	**6.9282**	7.4364	8.9670	8.3423
	V2	11.1157	**6.8805**	7.5318	8.9821	8.5187
I69	V1	10.6359	**4.2761**	5.3309	5.9863	5.9424
	V2	10.1248	**4.9686**	5.9078	6.5613	6.7879
I75	V1	10.4226	**5.4757**	6.7625	7.3139	7.4913
	V2	11.5332	**5.7083**	7.1702	7.6710	7.8511

Table 3.2: Comparison of entropy values of original, DPCM & DPCM with linear prediction for 16-bit ECG signals

Since the final encoding step is based on entropy encoding, it is important to note that the complexity of the encoder increases as the size of the samples increases. A single bit increase in the number of bits used to represent the sample effectively doubles the total number of codes that must be calculated. The DPCM implementation discussed above calculates the difference between successive samples, so its output will produce both positive and negative values, whereas the input was strictly positive. Since having both positive and negative values in the output would require an additional bit to represent all the possible values, the number of symbols the entropy encoder would have to handle would double, and would therefore significantly increase the complexity of the entropy encoder. In order to eliminate this problem, the quantizer can be implemented so

that the entire range of output values is mapped to strictly positive values, thereby eliminating the need for the additional (sign) bit.

This mapping of a range of positive and negative values to all positive values can be accomplished by exploiting the fact that we know that every input into each encoder will be a set size. Examining how the DPCM is calculated, and then reconstructed shows how we can take advantage of this fact. The following is a simple example, using 4-bit inputs, of how the DPCM would normally be calculated. This DPCM operation results in a 5-bit output, but when the signal is reconstructed back to the original data when the sign bit is always zero and is no longer needed. In the following example, the DPCM is calculated according to the implementation given in Figure 3.1(a). The difference value based on each input sample is calculated by taking the two's complement of the previous input value, and then adding it to the current input value. The resulting DPCM encoded value needs 5 bits to represent it since taking the difference between the two samples can result in negative values. To undo the DPCM encoding, the decoder simply starts with an initial value of zero, and keeps a running sum of all the DPCM values. Since some of the DPCM values are negative, this running sum is really composed of both addition and subtraction operations. This will restore the data back to its original values, and the sign bit can then be dropped since we know that all the original input values were strictly positive. Table 3.3 shows the DPCM encoding and decoding of an example set of data.

Sample #	Input Sample Values	DPCM Encoded Value: w/ sign bit	Undo DPCM	Restored Data: sign bit no longer needed
Initial	0000			
1	0001	00001	0000 + 00001 = 00001	0001
2	0011	00010	0001 + 00010 = 00011	0011
3	1100	01001	0011 + 01001 = 01100	1100
4	0110	11010	0100 + 11010 = 00110	0110
5	0010	11100	0110 + 11100 = 00010	0010
6	0011	00001	0010 + 00001 = 00011	0011
7	0011	00000	0011 + 00000 = 00011	0011
8	0000	11101	0011 + 11101 = 00000	0000
9	1110	01110	0000 + 01110 = 01110	1110
10	1000	11010	1110 + 11010 = 01000	1000

Table 3.3: Simple DPCM example with 4-bit inputs and 5-bit DPCM outputs

The above example is a valid way of calculating the DPCM values of a signal, and then restoring the signal back to its original form. However, as discussed previously, having this extra bit is not desirable since the DPCM values will eventually be encoded again using an entropy encoder. Including the sign bit will double the number of codes the entropy encoder would need to calculate and will increase the overhead cost associated with transmitting the codes to the decoder. The need to keep the sign bit can be eliminated by remembering that the inputs to the encoder will all be represented using a set number of bits. Since we are dropping the sign bit of the DPCM values, after a negative sign bit is dropped, any values that were negative (sign bit of 1) are simply the two's complement of the magnitude of the negative value. For example, if the DPCM gave an output of -2 for a sample value, then the 2's complement of 2, which is 14 for 4-bit values, is used instead. When reversing the DPCM, adding 14 would achieve the same result as subtracting 2, since the data size is constrained to 4-bits and the value would roll over to zero when needed so as to achieve the same result. This only works

due to the fact that we are requiring all the inputs of the DPCM encoder to be a single size. To make this process a little more clear, the same input sequence from the previous example is taken and the DPCM process is repeated, with the exception that this time we will drop the sign bit. The DPCM is calculated the same way, but the outputs of the DPCM are held to 4 bits, the sign bit on any negative values is simply dropped. The original values are restored from the DPCM data, the values are simply added together and any overflow is ignored. This is shown in Table 3.4 and the results are the same as when the sign bit was kept.

Sample #	Input Sample Values	DPCM Encoded Value: sign bit dropped	Undo DPCM	Restored Data: Overflow – sign bit dropped
Initial	0000			
1	0001	0001	0000 + 0001 = 0001	0001
2	0011	0010	0001 + 0010 = 0011	0011
3	1100	1001	0011 + 1001 = 1100	1100
4	0110	1010	1100 + 1010 = 10110	0110
5	0010	1100	0110 + 1100 = 10010	0010
6	0011	0001	0010 + 0001 = 0011	0011
7	0011	0000	0011 + 0000 = 0011	0011
8	0000	1101	0011 + 1101 = 10000	0000
9	1110	1110	0000 + 1110 = 1110	1110
10	1000	1010	1110 + 1010 = 11000	1000

Table 3.4: Simple DPCM example with 4-bit inputs and 4-bit DPCM outputs with the sign bits Being dropped

3.1 Run-Length Encoding

Run-Length Encoding (RLE) is an effective tool to use in cases where there are segments of data that all have the same value. In cases where there is a sequence of N samples that all have the same value, rather than encoding all N values it is much more efficient to recognize this as a run-length, and encode the sample value of the sequence, along with the number of samples contained in the run length. If RLE is performed on the output of

a DPCM operation, then there are two ways that run-lengths can be created. The first is in the case stated above, where a segment of the input signal's values are all the same; in this case the DPCM operation will output all zeros for this segment since the difference between equal valued samples is zero. The second case where run-lengths could be generated is if the input signal's values are increasing or decreasing at a steady rate. If for example you have a string of ten values where each value is one greater than the last, then the DPCM will output a string of DPCM values that all have a value of "0001" for the 4-bit case. While ECG signals do not normally include long sequences of samples that are all the exact same value, they do include segments that are steadily increasing/decreasing by constant amounts. Plus, if a 12 or 16-bit ECG signal is split into smaller bit sizes, then the higher bits may very well contain strings of identical values since only the lower bits may be required to represent the full value of the sample. Therefore, implementing a RLE operation so that the input to the RLE block comes from the output of the DPCM transformation can provide additional compression performance.

When a valid run-length is identified, the first sample in the run length is encoded normally. This first sample serves as the identifier for what the remaining values in the run length are. Following this first sample, a run-length flag is encoded so that the decoder knows to treat the following sample as a run-length count value and not as normal sample. When the decoder encounters a run-length flag, it simply retrieves the previously decoded sample, and restores as many of that value as is indicated by the run-length count. A diagram of the run-length encoding structure is shown in Figure 3.2.

Figure 3.2: Order of encoded values used to represent a run-length sequence

3.2 Segmenting Samples for Entropy Coding

One of the challenges when encoding ECG data is that there are several numbers of bits that are commonly used to represent the digitized signals. 11, 12, or 16 bit representations are the most common, but there are also 8 bit representations that are used as well. For an 8 bit representation there are 256 different possible values, for 12 bit representations this balloons to 4,096 possible values, and 65,536 possible values for the 16 bit representation. This presents a problem since calculating and storing 65,536 different codes would be enormously time consuming, as well as require an exorbitant amount of resources. In addition, the codes will have to be transmitted along with the compressed data, so it is desirable to reduce the total number of codes needed so that the overhead cost associated with transmitting the codes is reduced.

This problem can be addressed by recognizing that as the number of bits used to represent the inputs increases, the number of different possible input values increases exponentially. We can exploit this fact by taking the bits used to represent each input, and breaking it up into several smaller bit segments. These separate segments can then be encoded and decoded individually. Once the segments have been individually decoded, they can then be concatenated back together to restore the original full length input values. This will effectively reduce the number of individual codes that need to be calculated. For example, as discussed above the 16 bit representation would require the

calculation of 65,536 individual codes, however, if each 16 bit value was encoded using two 8-bit encoders the total number of codes that would need to be calculated would be 2*256 = 512. This is an enormous reduction in the number of codes that have to be calculated and stored. Breaking up the input bits into combinations of 4, 6, and 8 bit segments is investigated, since these can be easily combined to reach the 12 and 16 bit ECG representations. Segments longer than 8 bits are not considered, since anything above 8 bits requires too many codes, and the time needed to compute the large number of codes would not be acceptable.

3.3 Histogram Calculation

Since the implementation being proposed is based on entropy encoding, it is necessary to generate histograms to provide the statistics which will be used in the calculation of the final codes used to encode the ECG signals. In order to generate codes with which to encode the signal samples that have gone through the DPCM and RLE blocks, it is necessary to have values for the probabilities of all the possible symbols of the input signal. Since the statistics of the input signal are not known in advance, an initial histogram based on ECG signals from the PhysioBank Physiologic Signal Archives [6] are used to generate codes that will be used to encode the initial part of the signal. As the signal data is processed a new histogram will need to be calculated from the current samples of the signal being encoded. When enough of the signal has been calculated to form a suitable histogram based on the new signal, then new codes would be calculated. Once the new codes have been calculated, then the old codes would be replaced with the new codes, and the new codes transmitted so that the decoder has the codes to use for decoding and restoring the data.

3.4 Huffman Coding

Once a histogram, based on a fixed number of input samples, has been calculated, Huffman codes are then generated based on the algorithm described in [43]. It should be noted that the implementation used here does not actually calculate the probabilities of the symbols in order to simplify the hardware and to avoid having to use floating point representation. Instead, calculation of the Huffman codes is done using the actual number of times each symbol occurred instead of using the probabilities. This has the same affect and creates the same code table that using actual probabilities would have created.

In order to implement the Huffman algorithm, two queue structures are used to hold the leaf nodes that are used to create the binary tree. The Huffman codes are calculated using the following steps:

1. The histogram values are sorted in decreasing order so that the value that is most likely to occur will be accessed last, and the value that is least likely to occur will be accessed first.

2. Examine the next accessible nodes in each (non-empty) queue and take the two nodes that have the lowest values. This may mean taking two nodes from the same queue, or one from each queue. The two nodes representing the two values least likely to occur are taken and their values added together to form a new node, with the original two nodes as children. If there is a tie when choosing which queue to take nodes from, always choose the first queue, this

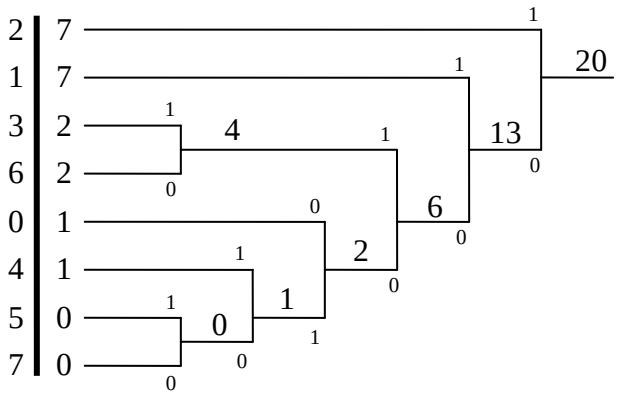
will maintain the mathematical optimality of the Huffman coding while minimizing both the variance and the length of the longest code.

3. Each time two nodes are added together, the first node selected from the queues is assigned a ‘0’, and the second node selected is assigned a ‘1’. This new node is then stored in the second queue structure.

4. If there is still more than one node in the two queues combined, go back to step 2. Repeat this process until all the nodes from the first queue have been chosen and all the nodes from the second queue have been combined until there is only 1 node left.

5. The one remaining node is the root node of the binary tree and valid Huffman codes for all possible symbols have been calculated. The codes are then obtained from the node tree generated by the above steps by starting at the root node and traversing down to the leaf node that represents the desired value. The values, ‘1’ or ‘0’, of each node passed are collected in order and these form the code.

The following figure shows an example of the binary tree and resulting code table generated from a set of 8 possible values and their histogram (number of times each value occurs). The small numbers on the binary tree represent the value (0, 1) assigned to a particular branch, while the large values on the tree gives the total number of value occurrences that can be reached from that point in the binary tree.



Value	Assigned Code
0	0001
1	01
2	1
3	0011
4	00011
5	000101
6	00010
7	000100

Figure 3.3: Example of Huffman code table calculation using 8 values and 20 total occurrences of those values

Once the Huffman codes have been calculated, they are then transformed into Reversible-Variable-Length Codes (RVLC) in order to provide error resiliency since RVLC codes are symmetrical and can be decoded both frontwards and backwards. These RVLC codes are the final codes used to encode the input data after it has gone through the DPCM and RLE transformations.

3.5 Reversible Variable Length Coding

RVLC codes are generally calculated by taking a previously calculated table of variable length codes, and using the characteristics of the variable length codes to provide metrics from which the RVLCs are then calculated. Takishima *et al*. in [44] and Tsai *et al*. in [45] provide algorithms for calculating RVLCs from Huffman codes. However, the algorithms they describe calculate symmetrical codeword's on a full binary Huffman tree and pre-calculate the number of available codeword's at each level before generating the symmetrical RVLC. The overall design procedure used in these algorithms requires high complexity when adapting pre-calculated values at each level to the given Huffman code

table. There also are restrictions imposed during this adaption process which can lead to valid symmetrical codeword's being missed at some levels.

Jeong and Ho present a new algorithm in [46] which simplifies the adaption process to the given Huffman code table, and also often reduces the average code length since this algorithm does not miss any symmetrical codewords. This algorithm generates RVLCs from the Huffman table, but it is able to provide a simpler and faster implementation by taking advantage of some basic characteristics of RVLCs. When Huffman codes are generated, the codes must satisfy the prefix condition, i.e. the first portion of a code can not be identical to any of the shorter codes. RVLCs on the other hand must satisfy both the prefix and the suffix condition in order to be decoded in both the forward and backward directions. However, when considering the symmetrical RVLC, the prefix condition automatically leads to the suffix condition due to the symmetrical bit patterns of RVLC codes. Therefore, if all the bits of a chosen symmetrical codeword which satisfies the prefix and suffix conditions are inverted, a second symmetric codeword is obtained which also satisfies the prefix and suffix conditions. This means that if we have *S* Huffman codes that we need to calculate RVLCs for, then we only need to calculate codes for one side of the Huffman tree which will be *S*/2 (round up if not an integer value) codeword's. This gives us half of the needed codeword's, and the other half can be easily generated be simply inverting the bits of the *S*/2 calculated codeword's. Essentially, if you consider a binary tree and look at the two branches off the main node, this algorithm finds all the RVLC codes on the '0' branch (it finds S/2 codes on the '0' side). Since these S/2 codes meet all the conditions of a RVLC code, then their complements must also meet the conditions as well. Since

we only need to check the prefix and symmetric properties when calculating codes for one side of the Huffman tree, and not for both sides, the time required to calculate the RVLC codes and the number of values searched through to find the symmetric values is dramatically reduced.

When calculating the initial RVLCs for the left side of the Huffman tree, the table of RVLCs will contain only one code that consists of all '0' bits or else the prefix condition would not be satisfied. Therefore, all the codes that are longer than this *zero-code* (Z_L), where L is the level at which the Z_L code appears, must start with less than L zeros in order for the prefix condition to be satisfied. This choice will affect the number of RVLC's that are found at each level, and will therefore affect average code length of the generated RVLCs. In order to reduce the average code length as much as possible, the characteristics of the probabilities of the Huffman codes are taken into account, and the best fit of the RVLC to the probability distribution of the Huffman code is obtained by setting L equal to the shortest Huffman code $L_{H\_\min}$. Table 3.5 shows an example of how a set of RVLC codes is generated from a set of Huffman codes. The first column shows the values from the left side of a binary tree, while the next three columns show whether each value from the binary tree meets the necessary conditions of a RVLC code. If a value does not meet any of the four conditions it is immediately discarded, if it meets all the conditions, then it is selected as an RVLC code and replaces the next most common Huffman code.

Values Selected from Left Side of Binary Tree	Symm-etric?	Symmetric. Code > 0 or length (Zero_Code) = Z_L ?	Prefix Condition Met?	Valid RVLC Codes Found	Find Remainder of RVLC Codes by Inverting Previous Column
000	✓	×			
001	×				
010	✓	✓	✓	010	101
011	×				
0000	✓	✓	✓	0000	1111
0001	×				
0010	×				
0011	×				
0100	×				
0101	×				
0110	✓	✓	✓	0110	1001
0111	×				
00000	✓	×			
…	×				
00100	✓	✓	✓	00100	11011
…	×				
01010	✓	✓	×		
…	×				
01110	✓	✓	✓	01110	10001
…	×				
000000	✓	×			
…	×				
001100	✓	✓	✓	001100	110011
…	×				
010010	✓	✓	✓	010010	101101
…	×				
011110	✓	✓	✓	011110	100001

Table 3.5: Example of the process used to select RVLC codes with $Z_L = 4$

The algorithm for generating RVLC's according to the algorithm described above is shown here.

1. The bit length of Z_L for the left half region of the binary Huffman tree is selected to be the same length as the shortest Huffman code, $L_{H\_\min}$.

2. Until all the necessary number of symmetric codes from the left half region of the binary tree have been selected (*S*/2 codes in all), all available symmetrical codeword's are chosen from the highest remaining level (i.e. shortest symmetrical codewords are evaluated first). These symmetrical codeword's are further weeded out by eliminating all codeword's that violate the prefix condition (i.e. symmetrical codes that starts with a previous smaller code are not selected), and any all zero codes whose lengths do not equal Z_L.

3. Combining the selected codeword's with their bit-inversed codeword's gives a full RVLC table of symmetrical codes that can be decoded from both directions.

4. The shortest Huffman code is replaced with the shortest RVLC codeword. This is repeated with all remaining codes until all Huffman codes have been replaced. This maintains the goal of assigning the shortest codeword's to the most frequently seen symbols.

3.6 ECG Signal Database

The ECG signals used to evaluate the best DPCM implementation to use, as well as to evaluate the final compression algorithm, were taken from the PhysioBank Physiologic Signal Archives [6]. These archives contain a comprehensive collection of ECG records available to the biomedical research community. This database of ECG records is often used as a standard database for determining the effectiveness of encoders geared towards ECG signals. Two collections of ECG records were chosen from the PhysioBank database from which individual signals (leads) were chosen to test the compression system. The 12-bit signals contain 30 minutes worth of ECG data, while the 16-bit signals contain 35 minutes worth. About one third of each signal, 150,000 data samples or about 10 minutes worth will be used in the testing of the encoding systems presented here.

The MIT-BIH Malignant Ventricular Arrhythmia Database was chosen to select 12-bit signals from because it contained 22 half-hour plus ECG recordings of subjects who experiences sustained episodes of ECG irregularities. The records contain ECG signals with a variety of different irregularities. The individual signals were selected by taking the first 16 records, and by taking the first ECG lead from each of the selected records. This provided a broad range of signals to be used for testing the ECG encoder and decoder. All signal values were shifted into the range of all positive numbers. A more detailed description of these records is given in Table 3.6.

Signal Num	Record	Record Lead	Number of Abnormal Episodes	Total Time Affected
1	418	ECG	61	4:06
2	419	ECG	33	3:29
3	420	ECG	3	4:42
4	421	ECG	50	5:46
5	422	ECG	5	11:55
6	423	ECG	5	8:12
7	424	ECG	9	8:41
8	425	ECG	3	5:55
9	426	ECG	10	13:04
10	427	ECG	2	24:12
11	428	ECG	8	34:48
12	429	ECG	17	31:50
13	430	ECG	32	34:19
14	602	ECG	5	6:44
15	605	ECG	1	0:29
16	607	ECG	9	33:32

Table 3.6: Detailed description of selected 12-bit ECG signals chosen from the MIT-BIH Malignant Ventricular Arrhythmia Database. Each signal from this database contains 525,000 samples, and is sampled at a rate of 250 Hz for a total length (Time) of 35 minutes.

The St. Petersburg INCART 12-lead Arrhythmia Database was chosen to select 16-bit signals from because it contained 75 recordings extracted from 32 Holter records, with each record being 30 minutes in length. The records contain ECG signals where the patients were undergoing coronary artery disease, and exhibiting a range of different ECG irregularities. A more detailed description of these signals is given in Table 3.7.

Signal #	Record	Record Lead	Number of Abnormal Episodes
1	I01	V1	344
2	I01	V2	
3	I11	V1	28
4	I11	V2	
5	I20	V1	290
6	I20	V2	
7	I32	V1	57
8	I32	V2	
9	I46	V1	426
10	I46	V2	
11	I57	V1	29
12	I57	V2	
13	I69	V1	169
14	I69	V2	
15	I75	V1	620
16	I75	V2	

Table 3.7: Description of representative set of 16-bit ECG signals chosen from the St. Petersburg INCART 12-lead Arrhythmia Database. Each signal from this database contains 462,600 samples, and is sampled at a rate of 257 Hz for a total length (Time) of 30 minutes.

Chapter 4

Implementation Methodology

4.0 Design Considerations for 12-bit and 16-bit ECG Encoding Systems

The compression system presented here was designed to be able to adapt to changing characteristics within the signal itself, and to be able to handle different signal types using a single compression system. When considering ECG signals in particular, there will be variance between signals from different patients, but unless an irregularity occurs, the ECG signal should be relatively periodic for long stretches of time. The characteristics of the signal may change if the patient increases or decreases physical activity, for example, but it should then remain periodic for a number of samples before the next change. This inherent periodicity in ECG signals means that having the ability for the compression system to periodically recalculate its codes may not be needed in some cases. The following two encoding systems were designed to investigate the effect that the periodicity of the ECG signals has, and which type of implementation is would be better suited for ECG compression in terms of both performance and required resources.

The first system (system A) was designed to periodically update its code tables based on the current characteristics of the signal, while the second system (System B) uses a single set of pre-loaded code tables to encode the entire signal thereby requiring significantly less hardware resources. These two systems were designed to meet the goal of requiring few hardware resources, low power usage, and to provide error resiliency to

limit data during signal transmission. The selected ECG signals will be run through both configurations, and the compression results along with the hardware requirements for each configuration can be used to determine which is more beneficial.

ECG signals are commonly quantized into 12 and 16 bits. Separate systems were set up for each case. Since DPCM is performed on the signals before they are encoded, the values of the resulting individual samples after DPCM in both the 12 and 16 bit representations should be small. This means that the large portion of the upper bits in both the 12 and 16 bit representations should not be needed to represent the value to be encoded, and should therefore be '0'. It would be expected then that the 16 bit representation would have a greater percentage of zeros in the high bits than the 12 bit representation would. This higher number of zero bits gives the best chance of allowing the highest bits to be encoded using run length techniques which would allow for the best compression possible. Therefore, we would expect that the compression system presented here would achieve a better overall compression ratio on the ECG signals that use 16 bits to represent the data then those that use 12 bits.

4.1 System Configuration

A simplified block diagram of the type of implementation presented is this thesis is shown in Figure 4.1. The input signal samples will be split into a chosen number and size of segments, with each segment being encoded independently of the others. The encoded segments will then be packed together and transmitted over a communications channel to a desired location where they can later be decoded and analyzed as needed. Table 4.1 shows a comparison of the number of codes required for the different possible

segment configurations. While the 4-4-4-4 segment encoding configuration is listed as a possibility in the table, it was not implemented in this design since having four different segments to package and transmit posed large problems with the transmission structure being targeted. Additionally, the single segment 12 and 16 bit configurations were not used either, since the total number of individual codes would require a large amount of time and resources to generate. In our research of previous work done in the area of lossy and lossless ECG compression, presented in section 2.4, we did not find any other instances of implementations that split the incoming signal, encoded the segments separately, and then recombined the segments upon decoding to restore the original signal.

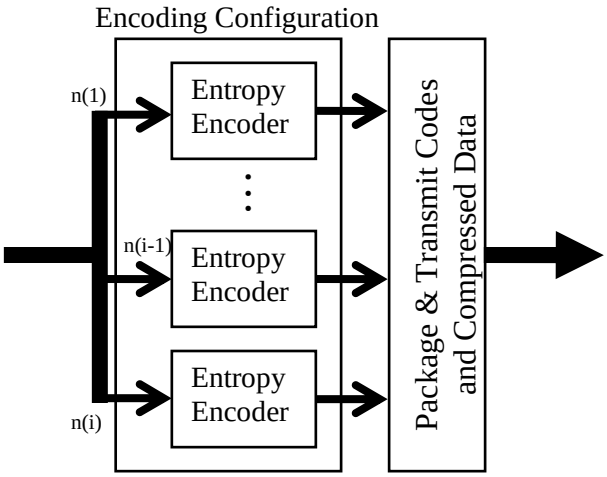


Figure 4.1: Top level diagram of signal encoder where n(0) through n(i) are the bit widths of the data going to the segment encoders, and the sum from n(1) through n(i) equals the bit width of the original ECG signal

Input Bit Size	Possible Coding Configurations	Number of Codes Required	Bytes of BRAM Required (System A)	Bytes of BRAM Required (System B)
12	**4-4-4**	**48**	324	120
	4-8, 8-4	**272**	2684	1048
	6-6	***128***	1030	374
	12	4,096	-	-
16	4-4-4-4	64	-	-
	4-6-6, 6-4-6, 6-6-4	**144**	1138	414
	4-4-8, 4-8-4, 8-4-4	**288**	2792	1088
	8-8	***512***	5152	2016
	16	65,536	-	-

Table 4.1: Comparison of segmented encoding configurations, the number of codes required for each configuration, and the number of BRAM bytes to implement each encoder configuration. Bolded configurations are implemented and tested. Italicized configuration for both the 12 and 16-bit implementations provided the best results. BRAM requirements only given for implementations that were demonstrated

The top level diagram of the proposed entropy coding implementation is shown in Figure 4.2. The final coding scheme is based on an entropy coder using Huffman codes which are then transformed into Reversible Variable Length Codes (RVLC). The RVLC codes are symmetrical, and are therefore frontwards and backwards decodable. In addition, they satisfy the prefix condition, meaning that a smaller code is not identical to the beginning of any longer code. Because RVLC codes are symmetrical, this means that if the prefix condition is met, the suffix condition will automatically be met as well. These conditions ensure that each RVLC code chosen is not only decodable from both directions, but that each code is uniquely decodable as well. If the prefix/suffix conditions were not met, it could be possible to have a code whose value could be contained in the beginning of a longer code. If this was the case the beginning portion of the longer code could be wrongly decoded as the shorter code, which would wreck the whole decoding process and result in incorrect data restoration. Incorporating RVLC

codes allows the decoder to decode backwards from a synchronization marker when an error occurs, thereby minimizing the data loss due to transmission errors. In order to maximize the compression results from the entropy coder, a Differential Pulse Code Modulation (DPCM) operation is first performed on the data signal, followed by run length coding (RLC). Due to the inherent correlation between neighboring samples in ECG waveforms, the combination of DPCM followed by RLC significantly reduces the entropy of the signal and results in more effective entropy coding.

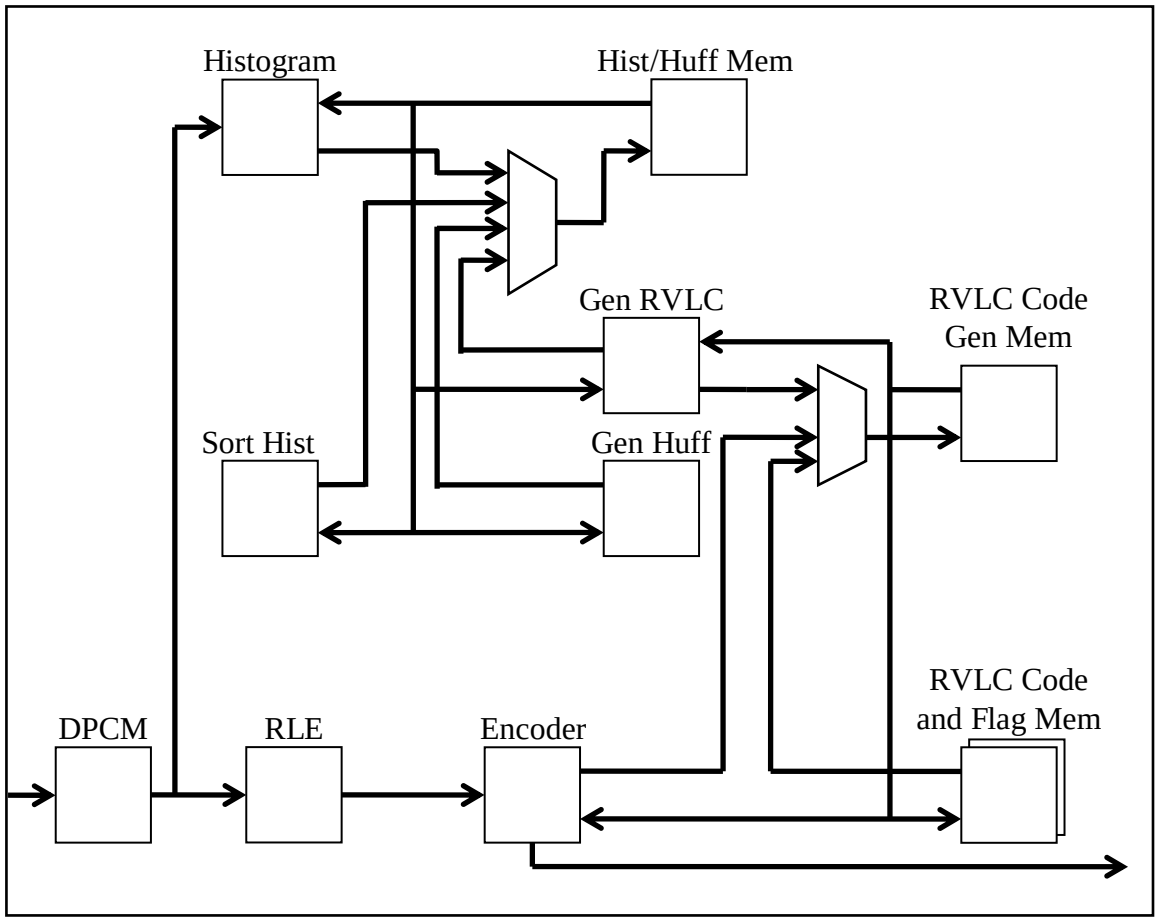


Figure 4.2: Entropy coding system top level view for a 4, 6, or 8-bit segment encoder with the ability update the RVLC tables

	4-Bit	6-Bit	8-Bit
Histogram Storage & Huffman Calculation Memory	16x22-bits	64x26-bits	256x30-bits
RVLC Generation Code Memory	16x12-bits	64x15-bits	256x19-bits
RVLC Final Code Storage Memory	16x16-bits	64x22-bits	256x31-bits
RVLC Final Flag Storage Memory	4x16-bits	4x22-bits	4x31-bits
Total Memory Usage	40 Bytes	187 Bytes	1008 Bytes

Table 4.2: Memory size requirements for 4, 6, and 8 segment encoders

The entire set of VHDL files for the encoder, and the Java software files for the decoder can be accessed at http://ivpcl.org. This is done to allow others to continue work on all or part of the implementation proposed in this thesis, and to allow for exact reproducibility of the results presented in this paper.

4.2 Component Implementation

4.2.1 Run-Length Coder

In order to be able to encode the data in a way that allows for error resiliency, the values that are sent for any run lengths must be encoded in the entropy coder since RVLC codes are used to allow for recovery from errors. The number of different values that the entropy coder can encode is based on the number of bits used to represent the input and is given as: $2^{input\_bit\_size}$. Since there is a limit on the largest value that the entropy coder can encode, the maximum length that can be sent for the run length must be able to be represented in the same number of bits used to represent the input values. This means that the longest run length that can be sent is: $2^{input\_bit\_size} - 1$.

As the ECG signal samples are fed into the encoding system, each sample is compared against the previous sample to determine if they are identical. If the current and previous samples are identical, then the current value is not transmitted, and a run length counter is incremented. This process is repeated until either the run length has reached its maximum allowed length, or a data sample is presented that is not identical to the previous sample. This means that for a sequence of identical values, the first value in the sequence will be encoded normally and will not be included in the run length since the run length count doesn't start until a second identical value is seen. Therefore the decoder only needs the number of values in the run length, and not the actual value of the run length samples, to be able to decode it since the value immediately preceding the run length is the same as the values in the run length.

When the end of the run length has been identified, the run length counter is compared against a specified minimum run length value to ensure that transmitting the run length value would provide better compression results then if the individual signal samples had been transmitted instead. This is a concern because when a run length occurs, a flag has to be encoded along with the run length so that the decoder can differentiate between a normally encoded sample, and an encoded run length. Since this flag adds extra bits onto the data that is transmitted for a run length, a run length must be long enough to make including the extra bits for the flag worthwhile. If a run length of only two or three samples is detected, it will often be more efficient to encode those samples individually rather than as a run length. The known length of the flag used to indicate a run length is compared to the most common shortest codes for each encoder

segment, to determine what the minimum run length will be. If this minimum run length is not met then the short run length is simply encoded as individual samples.

4.2.2 Histogram Calculation

The generation of the histogram is performed based on the ECG data samples that are currently being sent into the system. The goal of periodically recalculating the codes is to be able to identify changes in the properties of the signal and to provide updated codes that provide the best compression possible for the changing signal. Therefore, when new codes are recalculated, the histogram must provide a histogram of the most recent data seen. The system monitors the incoming signal and will recalculate the code tables at constant predefined intervals. The system provides an input to the histogram calculator long enough prior to the system needing to recalculate the codes, so that the there is enough time to calculate a full histogram of the current signal. A predefined constant value can be preset in the system to identify how many samples constitute a full histogram. This value can be changed to provide the ability to control how many samples in the current signal will be used to build up the histogram.

A block of memory is used to store the histogram values as it is being calculated. The index's of the memory range between $[\ 0,\ 2^{input\_bit\_size} - 1\ ]$, where the input bit size is either four, six, or eight. The index's of the memory are used to represent the different possible sample values, while the value stored in the memory at each index represents the number of times that value has been seen. When a sample is sent into the system, it is sent through the DPCM transform to reduce the entropy, the resulting value is the sent

into the run length/entropy encoder. At the same time, if the system has indicated that histogram is being calculated, then the value stored in the histogram memory is accessed using the DPCM value as the index, and the histogram value is incremented and then stored back to the same index location.

Since a block of memory with a predefined size is used for storing the histogram, it is necessary to limit the maximum number of samples that can be used to generate a single histogram table. The number of values used must be limited to the maximum value that one histogram memory location can store. Otherwise, if the histogram is dominated by a couple of values, it would be possible for the histogram value for those signals to exceed the number of bits used to store it, causing that value to overflow and roll back to zero. This would create an inaccurate histogram which would adversely affect the compression of the signal. Limiting the number of values used to calculate the histogram is acceptable, since it is desired to have the histogram only be based on a select portion of the ECG signal so that the code tables that are generated accurately reflect the characteristics of the current signal.

4.2.3 Sorting the Histogram

The basis of the entropy coder is that the shortest codes will be mapped to the most commonly seen sample values after they have been transformed in the DPCM process. In order to perform this mapping, the histogram has to be sorted so that the histogram values are easily accessible in descending order from the largest histogram value to the smallest. Once the histogram has finished being calculated, the values are read out of the histogram memory and stored in the Huffman code memory. Each memory index of the

Huffman memory is wide enough to store both the histogram value, and the index location representing the sample value that the histogram count represents. The histogram/index value pairs are then sorted and stored in the Huffman code memory from largest to smallest histogram count value.

Since the Huffman encoding assigns the shortest codes to the most frequently seen signal values, the same memory can be used to store both the sorted histogram values and the generated Huffman codes. The shortest Huffman codes are calculated first; therefore, as they are calculated they are stored in the memory location of the largest remaining histogram value and take on the sample value stored at the same histogram location.

4.2.4 Huffman Coder

Once a valid histogram has been calculated, the Huffman codes based on that histogram are calculated using the traditional Huffman algorithm described in section 3.4. However, the actual hardware implementation used was slightly different. Rather than use two queues as described previously, a single memory block was used to generate the Huffman codes. The histogram codes were sorted, from least to most likely to occur, according to their frequency of occurrence, so as histogram values are removed and added together, these intermediate values that would normally be stored in the second queue, are instead stored back in the next available vacated histogram memory location. This effectively reuses the vacated histogram memory locations to store the combined histogram values. It just requires a little additional logic to keep pointers of the next original histogram values and recombined histogram values to be evaluated.

4.2.5 Reversible Variable Length Codes

Once the full Huffman table has been calculated, the RVLC codes based on that histogram are calculated using the approach described in section 3.5. The length of the shortest Huffman code is used to set the parameters of the RVLC generation algorithm. Enough RVLC codes are generated to replace all the Huffman codes, and then four additional RVLC codes are generated to be used as flags that can be used during the packaging of the codes for transmission. One of the flags is used to indicate that a run length sequence was encoded, and the others are left for the any other uses that may be desired or required in the packaging of the codes.

4.2.6 Packaging Encoded Data for Transmission

Once the encoding of the data has been completed, it must then be packaged and transmitted to a chosen location where it can be decoded and analyzed at a later time. Since one of the major goals of this implementation is to minimize the amount of signal data that is lost in the event of data loss or corruption during transmission, there are specific things that can be done when packaging the data that would assist the encoder in restoring as much of the original signal as possible. This requires adding in a small amount of additional information into the data before it is transmitted. This will affect the overall compression performance of the system, but the payoff is that the decoder then has the ability to restore a greater amount of the encoded data.

One of the first steps often taken to minimize the amount of data loss or corruption during transmission, is to include synchronization markers at various points in

the encoded data so that even if data is lost or corrupted, the synchronization markers provide a point of reference from which to continue decoding. The synchronization marker will consist of signal sample values that have not been DPCM encoded, that will be placed at specific points in the encoded string. This is important in the implementation presented here because of the fact that we are using the DPCM transform to reduce the entropy of the signal. Reversing the DPCM transform requires keeping a running sum of all the decoded values in order to restore the original signal. If even one of the values is lost, then it is impossible to reverse the DPCM transform beyond that point until the next synchronization marker is reached.

While synchronization markers provide a point from which to restart the process of reversing the DPCM transform, all the data from the point where the data was lost to the next synchronization marker will be lost unless there is a way to start at the next synchronization marker, and decode the data backwards until the point where the error occurred is reached. This is why the DPCM encoded data was also RVLC encoded. RVLC codes are symmetric, i.e. the value that is obtained is the same whether it is observed starting at the left of the value and going right (frontwards), or is observed starting at the right of the value and going left (backwards). Therefore, encoding the data with these RVLC values will allow for the data on both sides of the corrupted portion of the signal to be recovered, at which point interpolation methods can be used to approximate the values that were not able to be recovered.

For the purposes of the research done for the implementations being described here, the medium used to transmit the data is a packet based bus with a variable length payload that has a maximum length of 256 bytes. The type of bus and the specific

properties that it contains are important in determining the best way to package and transmit the encoded data. Each packet that is sent contains both header data as well as the payload data. The header data contains a one byte address specifying the destination of the packet, a two byte object address that can be used to specify what type of data is in a given packet, and a one byte payload length value that is one less than the actual length of the payload (i.e. a one byte payload will have a payload length of zero, and a 256 byte payload will have a header payload length of 255). The packet format described above is shown in Figure 4.3.

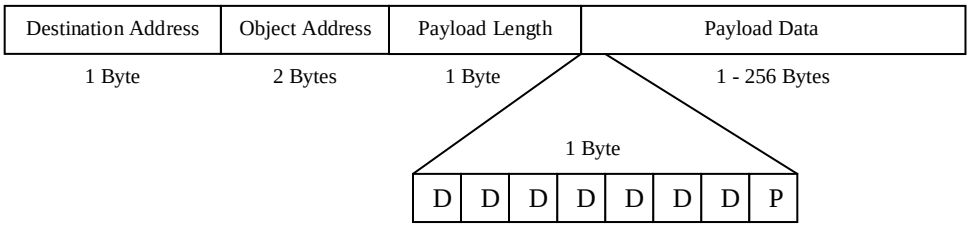
Destination Address	Object Address	Payload Length*	Payload Data
1 Byte	2 Bytes	1 Byte	1 - 256 Bytes

* - In order to be able to indicate lengths of up to 256 using a single byte, the payload length sent in the header is the actual payload length - 1. A payload length of zero would mean there was one byte of payload data, and a payload length of 255 would mean there were 256 bytes of data in the payload.

Figure 4.3: Packet format of transmission medium

The first thing to take into account when considering how to package the data is how the decoder will be able to identify where an error may have occurred. While the RVLC encoding will allow decoding of the data in both directions, it can not guard against a bit flip in the data throwing the decoder off. Suppose there was a RVLC code that had a value of “111” as well as one that had a value of “101101”, and a bit was flipped in the second value during transmission causing it to become “111101”. If the decoder was decoding this from left to right, it would read the first three bits and think that it found a valid RVLC code of “111”, when in reality the actual RVLC code was something else entirely. This type of bit flip would cause the decoder to wrongly decode several values before it got stuck and not be able to decode the remaining encoded data.

To address this problem, the last bit in every byte of the transmission packet payload will be reserved as a parity bit and will be calculated based on the seven previous bits, as seen in Figure 4.4. When the decoder receives the packet, the first step it will take will be to go through the payload data and double check that the parity bit in each byte still has the correct value. If the parity is wrong then the decoder knows that the data in that particular byte was altered, and will know that any code or portion of a code that was stored in that byte can not be accepted as a correct code.



D – One bit of encoded data
P – Parity bit calculated from previous 7 bits of encoded data

Figure 4.4: Structure of each byte of packet payload data

Another problem with reconstructing the original sample values from the DPCM encoding is seen when a bit flip occurs in a packet, thereby interrupting the process of the decoder keeping a running sum of the DPCM encoded values to reverse the DPCM encoding. The data was encoded in such a way as to allow for as much of the encoded data to be recovered as possible when an error like this occurs, but losing even one sample is enough to disrupt the ability to recover the original data if the decoder can only undo the DPCM encoding from one direction. The solution to this problem is to use the synchronization markers that are already embedded in the transmitted data. This value can simply be decoded by undoing the RVLC encoding, and does not rely on the DPCM encoding at all. This provides a value directly from the original signal, providing a basis from which the DPCM values recovered after an error occurs can be decoded, as well as

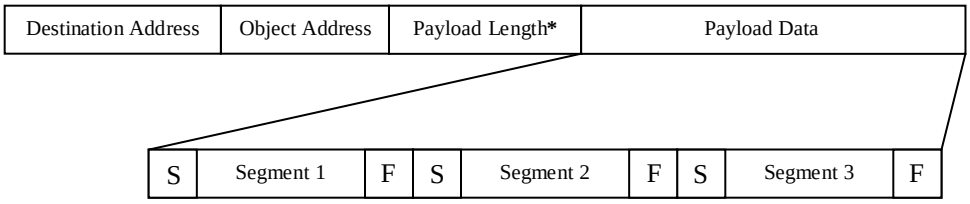
a providing a point from which to restart the DPCM decoding in the next packet. This value is placed in every single packet that is transmitted, and will be the very first value packed into the payload portion of the packet for each block of data from each encoding segment.

As the data is encoded, it is packaged into packets of the form previously described. Since the codes used to encode the signal data are different lengths, when packaged into the payload portion of the transmission packet the codes will often be packaged with portions of the code in sequential bytes. This is acceptable since the decoder first removes the parity bits, and then reformats the data into a single string. This single string is then decoded by reversing the RVLC and DPCM encoding. After the last byte of data for each segment in the encoder is packaged, there will often be a byte that is only partially filled. Values must be provided for all the bits in the byte, but the decoder must also be able to know that these extra bits do not contain any encoded data. In order to meet these requirements, a single bit with a value of '1' is always added after last encoded data has been packaged for each encoder segment. Then, if there are any bits left undefined in the last byte, they are all filled with '0's. This will occasionally result in an extra entire bit being added when the encoded data fills up the last byte completely. In this case when the extra '1' bit is added, it is placed in an additional byte whose remaining bits are filled with '0's except for the last one which is the parity bit. This occasional extra byte is necessary to allow the decoder to know what bits are extra data, and which actually contain encoded signal data. After the decoder removes all the parity bits from the encoded data, it looks at the end of the string and ignores all the '0' bits

until it sees the first ‘1’ bit. This ‘1’ bit is also ignored since it is the first extra bit that was added in, and the remaining data is then known to contain the encoded signal values.

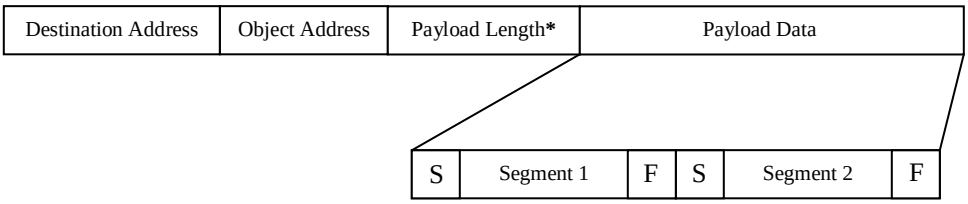
The encoding of the full ECG signal requires two or three encoding segments depending on which configuration is selected (8-4, 4-6-4, etc.). This means that data from all the segments will have to be encoded in each transmission packet. The easiest way to do this would be to encode all the segments from the first sample together, then all from the second sample together, and so on. This would work if it were not for the fact that run length encoding is being utilized. Since each signal sample is broken up into either two or three segments and each segment is encoded separately, a signal sample may have one segment be encoded by itself, while another segment is part of a run length encoding. This means that it is not always possible to encode all the segments from a single sample together. In addition this could result in different amounts of signal values being encoded in each transmission packet. If an entire packet were to get lost there would be no way to know exactly how many sample portions from each encoding segment were lost.

In order to ensure that the decoder can stay synchronized as it decodes the data, the encoded data will be packaged so that each encoder segment has its data packaged together. An example of this is shown in Figure 4.5 for a three segment encoder and in Figure 4.6 for a two segment encoder. Since the entire first segment is packaged before the next segment(s) are packaged, it is necessary to know how many encoded signal samples are going to be packaged in each transmission packet. The number of encoded signal samples encoded in each transmission packet must be kept to a value that allows all the data from all the segments to fit in a single transmission packet.



S – Synchronization value placed at start of encoded segment data
F – Filler data to fill out the remaining bits of last byte of each segment

Figure 4.5: Payload packaging configuration for 3-segment encoder



S – Synchronization value placed at start of encoded segment data
F – Filler data to fill out the remaining bits of last byte of each segment

Figure 4.6: Payload packaging configuration for 2-segment encoder

4.3 Decoding Packets with Errors

The ability of the decoder to recover from errors depends on several contributing factors. The structure of the packet plays a roll in how much data is able to be recovered, but the largest contributing factor is the number of bit errors present in a single packet. The two and three segment packet structures shown in Figure 5 (Methodology) and Figure 6 (Methodology), encode each segment data in such a way as to be independently decodable from the other segments in the packet. Each segment contains a synchronization marker, and the data from neighboring segments never share space in a single byte of data. However, it is important to keep in mind that portions of the data from a single sample of the original ECG signal are spread over all the segments in a packet. If one segment contains an error, then the sample that particular portion of corrupted data belongs too will be corrupted.

The total number of bit errors that are present in a packet affects the decoder's ability to recover from errors the most. Figure 4.7 shows a two segment encoder that has a single bit error in the packet. In this case the decoder begins at the beginning of segment 1, and decodes until it reaches the location of the error. Each byte contains a single parity bit, so the error can be traced to the specific byte that it occurs in, but not to the exact bit. Once the decoder has reached the error, then it goes to the next packet, takes the synchronization marker from the same segment number, and decodes the segment with the error in the backwards direction. With this ability to recover data in both directions, a single bit error will only result in a few samples being corrupted, rather than having every sample from the bit error to the next resynchronization marker corrupted.

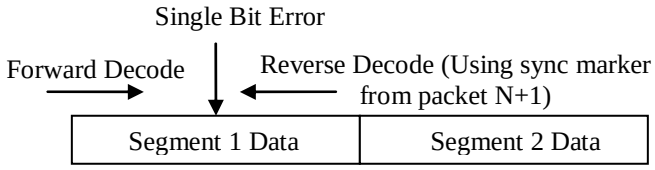


Figure 4.7: Payload packaging configuration for 2-segment encoder with single bit error

The next consideration would be in the event that there were two errors in a single packet. In this case the amount of data able to be recovered depends on the location of the errors. Figure 4.8 shows the case where one error occurs in each segment of data in the two segment encoder. This case is very similar to the single bit error case since each segment is independently decodable. Each segment is decoded in the forward direction up to the error, the resync markers are obtained from the next packet and the segments are then decoded in the reverse direction until they again encounter the error. This again allows the recovery of all but a small number of samples affected by the bit errors.

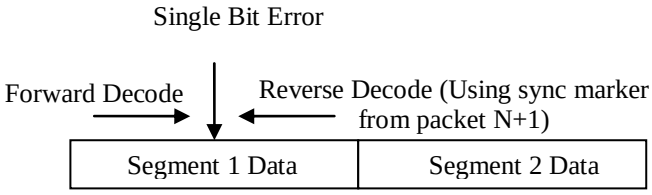


Figure 4.8: Payload packaging configuration for 2-segment encoder with two bit errors in separate segments

The other option for the two bit error case is if both errors occur in the same segment, as shown in Figure 4.9. This is the case that would cause the most data loss since the decoder has no way to decode that data that lies between the two corrupted bits. The decoder follows the same method to recover data, first from the forward direction, and then from the reverse direction, but all the data between the errors is not able to be recovered.

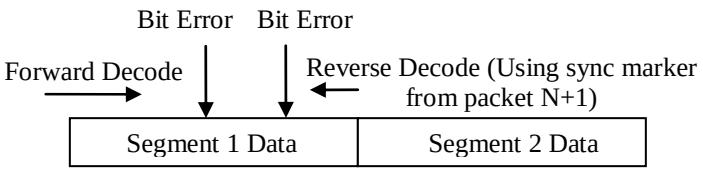


Figure 4.9: Payload packaging configuration for 2-segment encoder with two bit errors in the same segment

For the case where there are three or more bit errors in a single packet, the analysis is the same, but with more errors you guarantee that the distance between the to furthest apart errors in a single segment will be larger, resulting in a greater amount of data being lost.

4.4 Estimation of Unrecoverable Corrupted Data Samples

While the encoding algorithm presented in this paper allows for lossless encoding and exact decoding of the original data this is only truly possible in a perfect environment where there are no external influences on the encoded data. In the situation where the signals are encoded, and then transmitted over some medium to another location, there is the very real possibility of having errors injected into the packet data during transmission.

If the transmission is affected in such a way that causes the packet to never reach its destination, there is no way the encoding/decoding algorithm can recover the data from the lost packet. The best that could be done strictly from the decoder side would be to have a method to estimate the lost data through either linear interpolation, or though some method of estimating the data based on previous similar periodic signal data. The transmission scheme could incorporate resend logic that would resend a packet that was

not successfully delivered, but this would depend on the transmission technology, and not the encoder/decoder implementation presented here.

Since the signal packet structure, discussed in Section 4.2.6, has synchronization markers in every packet, losing an entire packet would not affect the decoder's ability to recover the data from previous or following packets. If a packet is lost completely, the decoder will identify and flag the samples which were not successfully recovered, but will continue decoding the remaining packets normally.

The other case to consider is if a packet is not completely lost, but instead has portions of its encoded data corrupted. The decoder will attempt to decode as much of the data as possible, and will then perform a simple linear interpolation to estimate the values of the unrecovered samples. Since DPCM is performed on the ECG signals and the difference between the signal values is encoded rather than the signal values themselves, it is very likely that a large amount of the upper bits will be all zeros since the difference between successive samples is generally pretty small. This means that if an error occurs in the upper bits, even though it may corrupt a number of samples, the interpolator will look at the values before and after the error and will perform linear interpolation to estimate the values of the corrupted samples. When the error occurs in the upper 'zero' bits of the samples, the interpolator will often estimate the 'zero' value correctly and will result in those samples being recovered exactly.

4.5 Identifying Accuracy of Estimated Data Samples

In the discussion in sections 2.2 and 2.3 on previous work done in the area of lossy ECG compression, the standard metric used to determine how close the compressed signal is to the original is the percent root mean-square difference (PRD) between the two signals. The PRD is defined as

$$PRD = \sqrt{\frac{\sum_{i=1}^{L}[X(i) - \hat{X}(i)]^2}{\sum_{i=1}^{L}[X(i)]^2}} \times 100\% \qquad \text{Equation (3)}$$

where X and $\hat{X}$ denote the original and reconstructed signals respectively, and L denotes the total number of samples.

In 1990 the American Heart Association recommended a goal PRD of 5% for routine visual readings of compressed and reconstructed ECG signals [47]. This gives us a benchmark with which to compare our results.

Chapter 5

Results

5.0 Hardware Requirements for the Encoder System Configurations

Though the achieved compression ratios of the individual encoder system configurations are important in determining which system configuration is the optimal solution, the hardware requirements of each configuration must also be taken into account. In some cases it may be necessary to sacrifice a small amount of compression performance if it meant that the hardware requirements would be significantly lower, or vice versa. The hardware requirements for each system configuration was obtained both for the implementation that had the ability to recalculate the code tables during compression, and for the implementation that used a single code table to compress the entire signal.

The encoding system was targeted for implementation in the Xilinx Spartan-3e (XC3S1200E) FPGA device [48]. This device contains 60 CLB rows by 46 CLB columns, for a total of 2,168 CLB's. Each CLB in this family of devices contains four slices, and each slice contains two look-up tables to implement logic, and two dedicated storage elements that can be used as flip-flops or latches. This equates to a total of 17,344 LUTs/Flip-Flops or 19,512 equivalent logic cells. The device contains 8,762 bits of RAM16/SRL16, and 138,752 distributed RAM bits. Table 5.1 and Table 5.2 list the hardware requirements for each segment configuration of the 12-bit encoders. Table 5.3 and Table 5.4 lists the hardware requirements for each segment configuration of the 16-bit encoders.

Configuration	Flip-Flops	(%)	Slice	(%)	LUT	(%)	Pwr (50 MHz)	Pwr (1 MHz)
4-8	1425	8%	3113	35%	5426	31%	42.78 mW	23.82 mW
8-4	1426	8%	3171	36%	5503	31%	43.11 mW	23.99 mW
6-6	1409	8%	3720	37%	5774	33%	41.96 mW	23.57 mW
4-4-4	1664	9%	2791	32%	4809	27%	54.47 mW	29.19 mW

Table 5.1: Resource and power utilization for 12-bit ECG signal encoder configurations using System A segment encoders

Configuration	Flip-Flops	(%)	Slice	(%)	LUT	(%)	Pwr (50 MHz)	Pwr (1 MHz)
4-8	582	3%	971	11%	1705	9%	18.17 mW	8.70 mW
8-4	583	3%	969	11%	1700	9%	19.31 mW	8.76 mW
6-6	578	3%	769	8%	1302	7%	17.47 mW	8.51 mW
4-4-4	715	4%	899	10%	1500	8%	19.01 mW	9.13 mW

Table 5.2: Resource and power utilization for 12-bit ECG signal encoder configurations using System B segment encoders

Configuration	Flip-Flops	(%)	Slice	(%)	LUT	(%)	Pwr (50 MHz)	Pwr (1 MHz)
8-8	1662	9%	4169	48%	7295	42%	48.17 mW	26.18 mW
4-4-8	1922	11%	3929	45%	6813	39%	51.50 mW	29.70 mW
4-8-4	1922	11%	3930	45%	6815	39%	54.75 mW	30.49 mW
8-4-4	1922	11%	3925	45%	6807	39%	51.74 mW	29.69 mW
4-6-6	1904	10%	3970	45%	6935	39%	54.44 mW	30.41 mW
6-4-6	1901	10%	3973	45%	6933	39%	54.65 mW	30.77 mW
6-6-4	1898	10%	4050	46%	7083	40%	54.26 mW	30.63 mW

Table 5.3: Resource and power utilization for 16-bit ECG signal encoder configurations using System A segment encoders

Configuration	Flip-Flops	(%)	Slice	(%)	LUT	(%)	Pwr (50 MHz)	Pwr (1 MHz)
8-8	614	3%	1011	11%	1720	9%	20.70 mW	8.76 mW
4-4-8	777	4%	1160	13%	1945	11%	24.69 mW	9.02 mW
4-8-4	777	4%	1159	13%	1945	11%	21.79 mW	9.03 mW
8-4-4	777	4%	1161	13%	1951	11%	20.62 mW	9.01 mW
4-6-6	766	4%	984	11%	1623	9%	20.00 mW	8.88 mW
6-4-6	768	4%	985	11%	1615	9%	19.74 mW	8.83 mW
6-6-4	770	4%	984	11%	1615	9%	19.09 mW	8.84 mW

Table 5.4: Resource and power utilization for 16-bit ECG signal encoder configurations using System B segment encoders

The systems that used code table recalculations obviously required significantly higher hardware resources since there was considerably more logic involved to generate the histograms, perform the sorting of the histograms, and to perform the Huffman and RVLC encoding. On the other hand, among all the hardware configurations that

implemented RVLC table recalculations, there was no significant variation in the required hardware resources.

The systems that did not perform RVLC code table recalculations required significantly fewer hardware resources. Furthermore, we did not find significant variations among different configurations. This indicates that the size of the hardware should not be the deciding factor in choosing what configuration to use. Instead, the choice can be made simply by determining what configuration provides the best compression performance.

5.1 Results for System A and System B Encoding Configurations

Compression of each of the sixteen signals selected for the 12 and 16-bit ECG representations was performed and the results analyzed to determine which encoding configuration provided the best results, and whether having periodic code table recalculations was beneficial for these types of signals. Each of the ECG signals was pre-processed and code tables for each signal and system configuration combination were generated based on the first 10,000 samples (~10 minutes). When each signal was encoded in each of the individual segment configurations, the pre-processed code tables generated from that signal for that configuration was loaded into the memory for the encoder to use until the system could recalculate the code tables on its own. Here, we report results first from use of the optimum code tables (the code tables generated from the signal being compressed), and then from the use of any set of code tables (from the 12-bit signals) to encode any of the 12-bit signals and the same for the 16-bit signals.

5.1.1 Encoding of 12-Bit ECG Signals with Optimal RVLC Code Tables

Figure 5.1 shows a comparison of the compression ratios for each of the 12-bit ECG signals when compressed using each of the different encoding segment configurations.

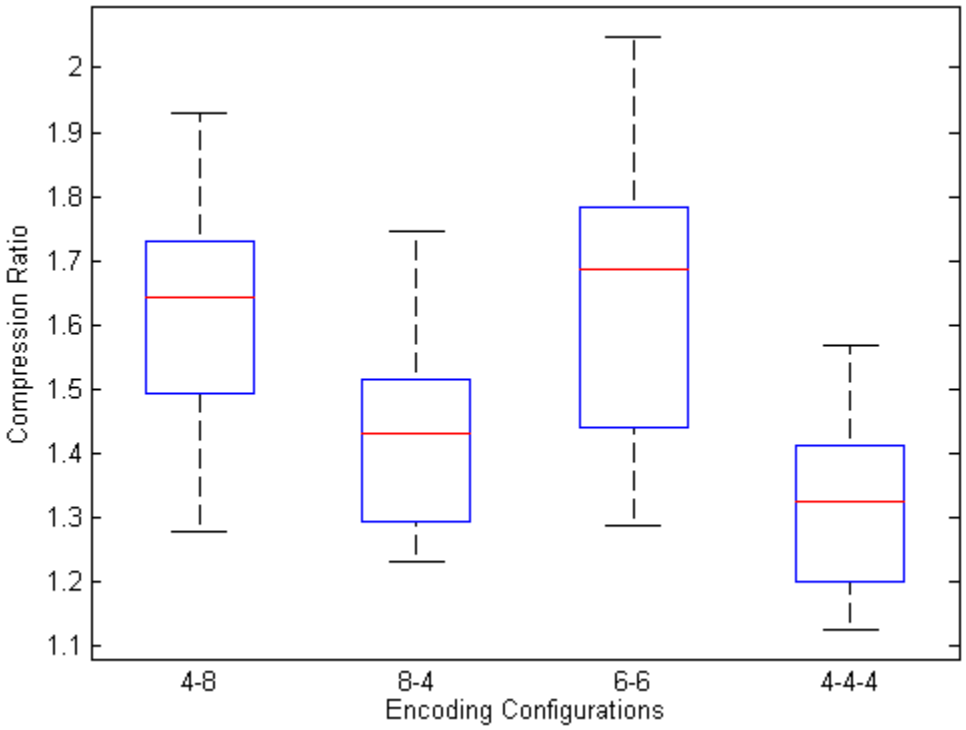


Figure 5.1: Compression ratio results for 12-bit encoding with optimal Huffman codes. For each encoding configuration, a boxplot is used to summarize the variation. Here, each box indicates the 25th percentile, the median, and the 75th percentile. The whiskers extend to the extreme values. Based on the combination of independent encoders, we have four possibilities. For example 4-8 implies the use of a 4-bit encoder for the most significant bits and an 8-bit encoder for the least-significant bits.

When using optimal RVLC code tables computed for each ECG signal, the 6-6 encoder configuration provided the best overall compression results. To see if there are significant differences when using the same RVLC table to encode the entire signal, we present results for the 6-6 encoder configurations in Figure 5.2. From Figure 5.2, it is clear that there is no significant gain from re-calculating optimal RVLC tables for each

ECG signal. Instead, it makes sense to compute optimal RVLC tables once, and then use the same optimal code tables for all other ECG signals.

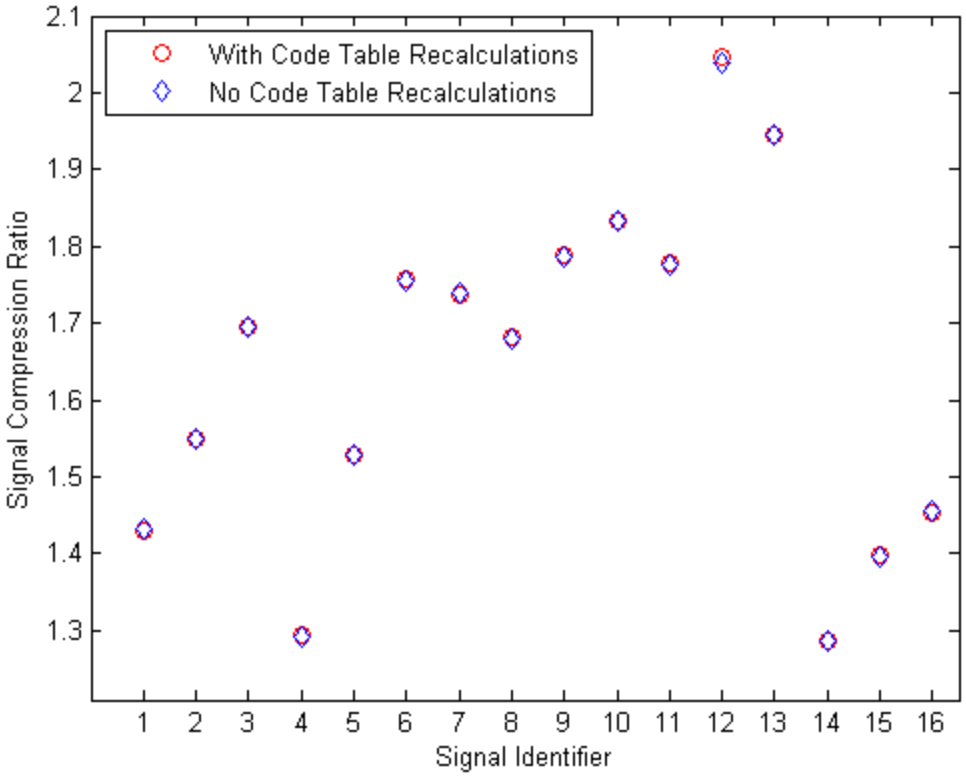


Figure 5.2: Compression ratios of 12-bit signals both with (System A) and without (System B) RVLC code table recalculations.

5.1.2 Encoding of 12-Bit ECG Signals without Code Table Recalculations

From Figure 5.2, we saw that when the signals were encoded using their own pre-calculated code tables and had no code table recalculations that the performance was almost identical to when code table recalculations were used. However, what if we didn't want to have to calculate a specific code table for each individual signal? It may be possible that from a subset of similar signals, a universal code table could be generated that would provide close to the same compression results. To investigate this possibility,

we will take each signal and compress it 16 times using its own code table, and the code tables from each of the other 12-bit signals.

The results from encoding each of the 16 signals with code tables from each of the 16 signals are displayed in Figure 5.3. The line in the center of the box indicates where the 50th percentile, and the top and bottom of each box indicate where the 75th and 25th percentile is respectively. Significant outliers are shown in the red plus sign. For additional reference, the black diamonds show the compression ratio for each signal when its own code table was used.

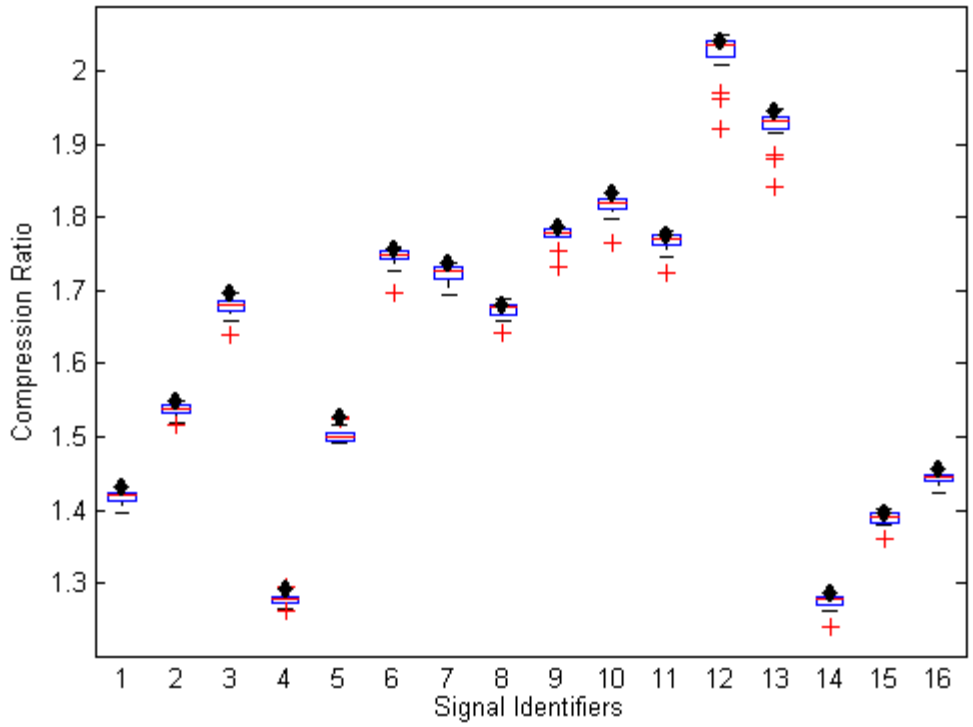


Figure 5.3: Range of compression ratios for 6-6 segment encoding configuration when code tables from each of the other 12-bit signals are used, compared against CR's when the optimal code tables were used (black diamonds)

When we inspect the results shown in Figure 5.3, we see that while each signal generally achieves close to its best compression performance is when its own code table is used. However, the results from when other code tables are used are only fractionally less, with a very small standard deviation. This indicates that the code table from any other similar signal may be used to encode a particular signal without the fear of having the overall compression results suffer significantly. Or, if given a subset of similar signals, a single code table could be calculated from this set of signals that would achieve very satisfactory results on the entire set of signals.

5.1.3 Encoding of 16-Bit ECG Signals with Code Table Recalculations

Similar to the 12-bit case, compression of each of the sixteen signals selected for the 16 bit ECG signals was encoded with periodic code table recalculations and a comparison of the results are displayed in Figure 5.4.

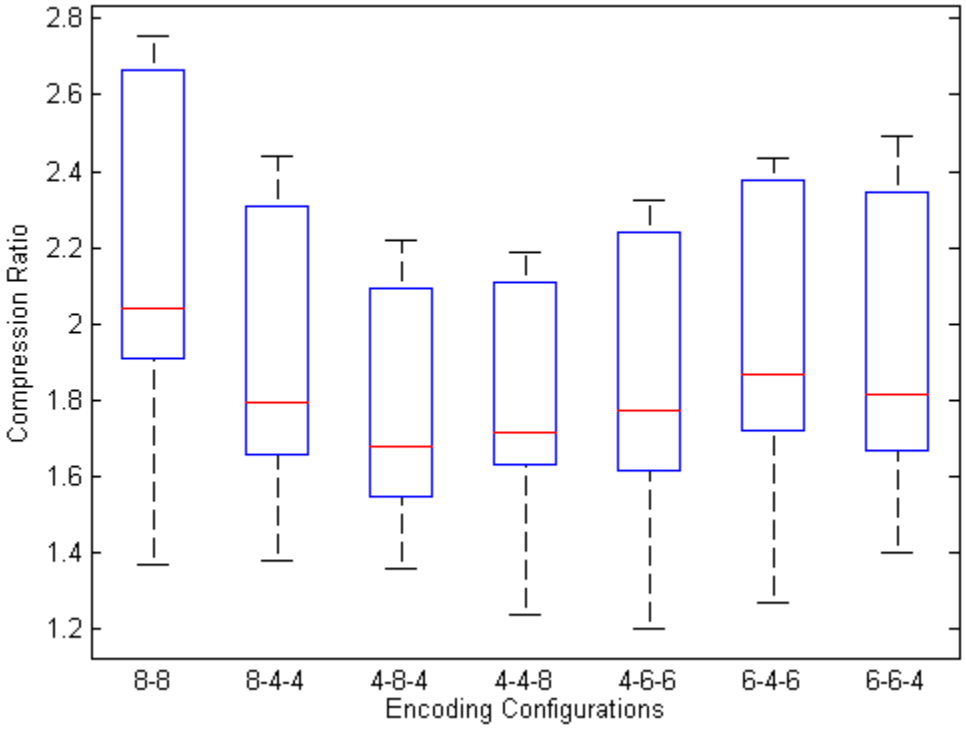


Figure 5.4: Compression ratio results for 16-bit encoding with optimal Huffman codes. For each encoding configuration, a boxplot is used to summarize the variation. Here, each box indicates the 25th percentile, the median, and the 75th percentile. The whiskers extend to the extreme values. Based on the combination of independent encoders, we have four possibilities. For example 8-4-4 implies the use of a 8-bit encoder for the most significant eight bits and an two 4-bit encoders for the least-significant eight bits.

As seen from Figure 5.4, for the 16 bit ECG signals, the compression system configured in the 8-8 bit segment configuration provided the best average compression results. But once again this does not tell us what value, if any, performing periodic code table recalculations provides. In order to determine if the code table recalculations provide any value, the same set of 16-Bit signals will again be compressed without including the hardware to recalculate the codes. This will allow for a comparison of the compression results and hardware usage of both versions of the encoding system to determine which one provides the best combination of compression results and hardware usage.

Figure 5.5 shows the compression ratios of each of the 16-bit signals encoded twice using the 8-8 segment configuration. The first set of results was plotted with the signals having their pre-calculated code tables pre-loaded in memory, and then their code tables recalculated every 10,000 samples (~39 seconds apart). Secondly, the same set of signals was again encoded using their pre-calculated code tables, but this time the single, pre-loaded, set of code tables was used to encode the entire signal. Looking at these results shows that there was virtually no benefit to recalculating the code tables. Only signals had a noticeably higher compression ratio when the code tables were recalculated, and a couple actually achieved better compression results with no code table recalculation. Overall, the difference between the two sets of results is not significant enough to warrant including the extra hardware required to perform the code table recalculations.

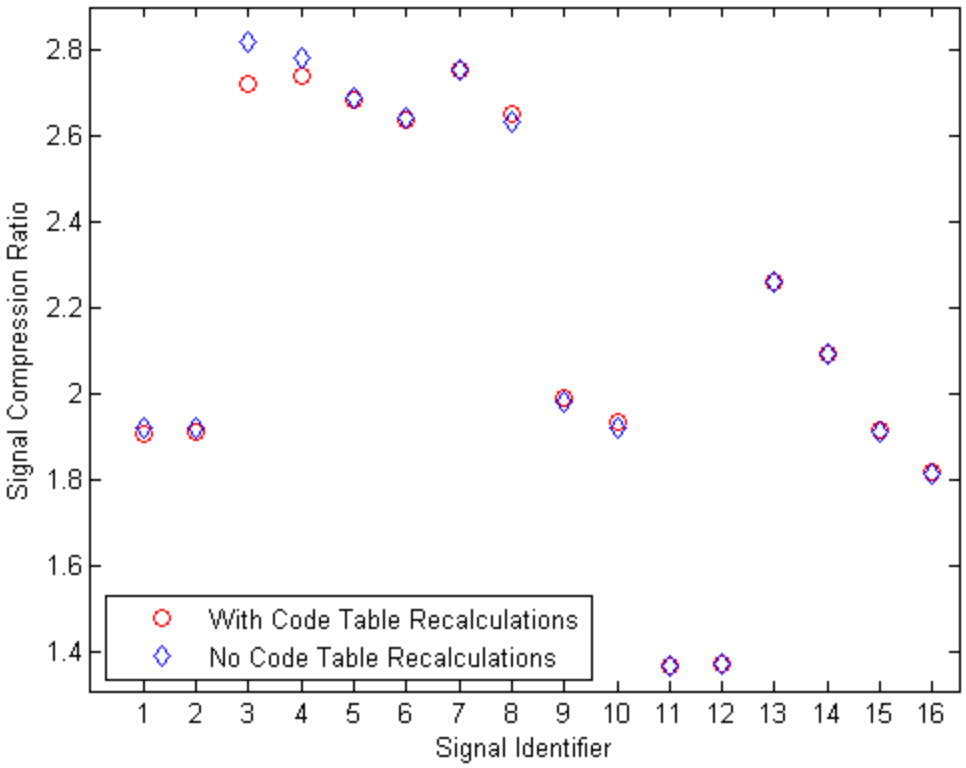


Figure 5.5: Compression ratios of 16-bit signals both with (System A) and without (System B) RVLC code table recalculations.

5.1.4 Encoding of 16-Bit ECG Signals without Code Table Recalculations

From Figure 5.5, we saw that when the signals were encoded using their own pre-calculated code tables with no code table recalculations that the performance was almost identical to when code table recalculations were used. Similar to the 12-bit case, it would be useful to determine if a single code table could be generated from a subset of similar signals, and be used on a larger set of signals while still achieving close to the same compression results. To investigate this possibility, we will take each signal and compress it 16 times using its own code table, and the code tables from each of the other 16-bit signals.

The results from encoding each of the 16 signals using code tables from each of the 16 signals are displayed in Figure 5.6. The line in the center of the box indicates where the 50th percentile, and the top and bottom of each box indicate where the 75th and 25th percentile is respectively. Significant outliers are shown in the red plus sign. For additional reference, the black diamonds show the compression ratio for each signal when its own code table was used.

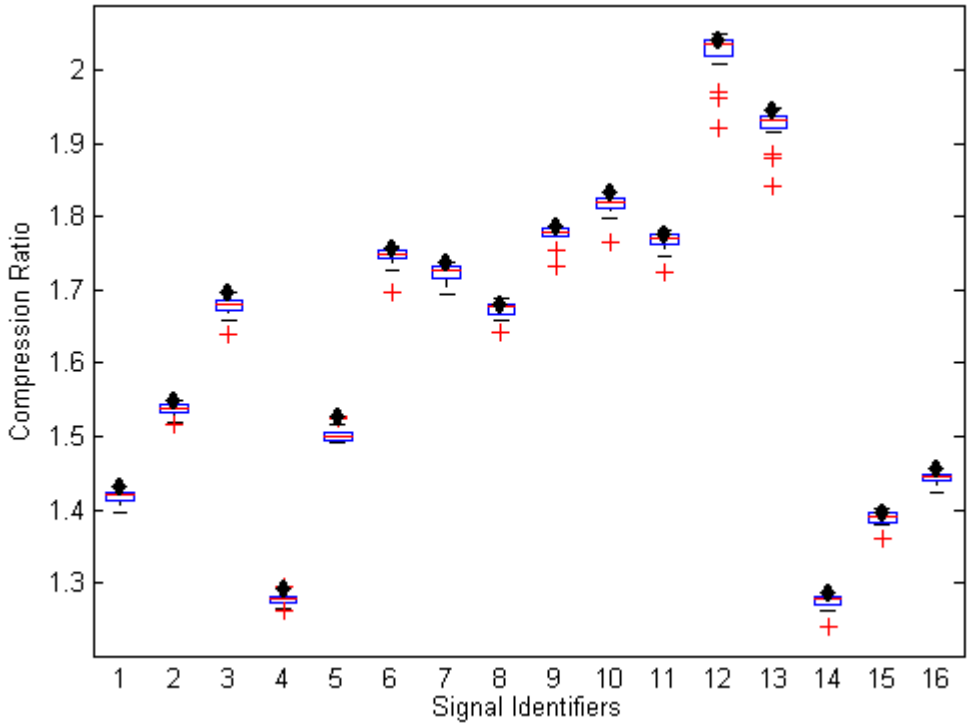


Figure 5.6: Range of compression ratios for 88 segment encoding configuration when code tables from each of the other 16-bit signals are used, compared against CR's when the optimal code tables were used (black diamonds)

When we inspect the results shown in Figure 5.6, we see that while each signal generally achieves close to its best compression performance is when its own code table is used. However, the results from when other code tables are used are normally only fractionally less, with a very small standard deviation. This indicates that the code table

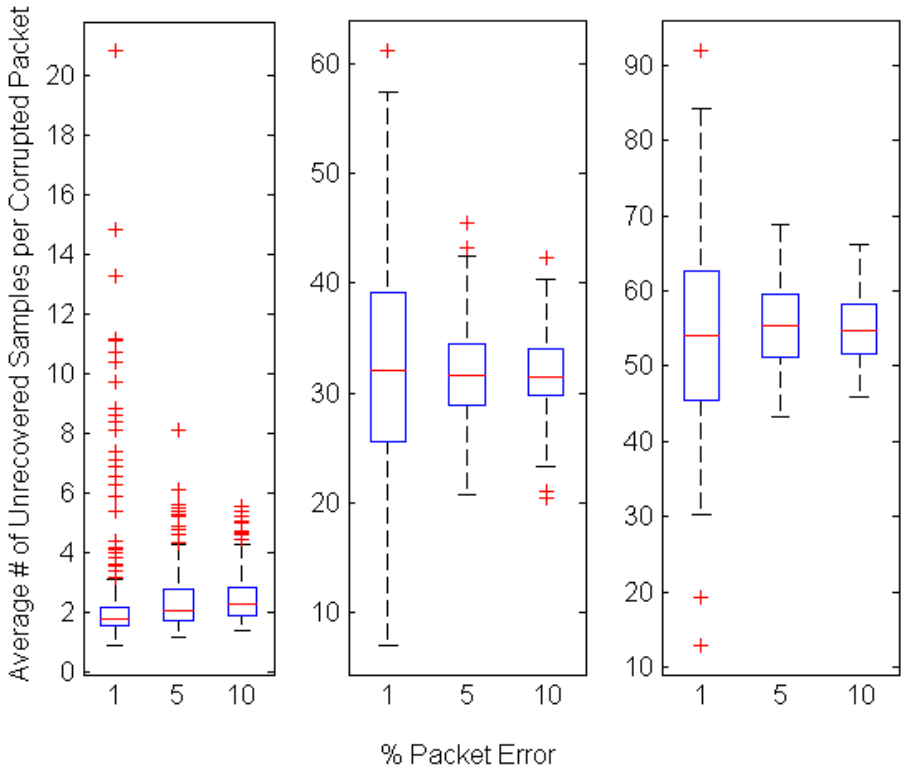
from any other similar signal may be used to encode a particular signal without the fear of having the overall compression results suffer significantly. Or, if given a subset of similar signals, a single code table could be calculated from this set of signals that would achieve very satisfactory results on the entire set of signals.

5.2 Evaluation of Error Recovery Capabilities of ECG Signal Decoder

In order to evaluate the performance of the decoder when one or more bit errors are present in a packet, we performed a series of simulations where the decoder decoded the signals with different amounts of errors injected into the encoded packets. In section 4.3 a discussion was provided on how the sample loss rate would be affected by different amounts of bit errors. In this evaluation we will consider packet corruption rates of 1%, 5%, and 10%. For each packet corruption rate we will first insert single bit errors, and then insert two bit errors per corrupted packet and compare the difference in the number of unrecovered samples. While the number of unrecovered samples is an important metric, we will also be looking at the PRD values of the estimated samples to see how close the estimation was to the original sample values. This will allow us to evaluate not only how many samples were unrecovered, but also what the likelihood is that the estimated samples will fall within an acceptable range that will still allow for accurate diagnosis of the ECG signal.

5.2.1 Decoding Analysis of 12-Bit ECG Signals with Injected Errors

Each encoded 12-Bit signal was injected with errors and then run through the decoder, and the total amount of unrecovered samples was recorded. This was done 15 times per signal for each of the different packet corruption rates. The first two plots in Figure 5.7 show the results for both one and two bit errors per corrupted packet. The third plot shows the single bit case again, but this time the decoder performed only frontwards decoding, but did not perform backwards decoding. This means that the decoder recovered data up to where the error occurred, but could not recover any data from the packet after the error occurred since the reverse decoding was disabled. This shows the improvement in the amount of recovered data that is attributed to being able to decode the packet from both directions.



(a) (b) (c)

Figure 5.7: Average number of unrecovered samples at different error rates for 12-bit signals. (a) Results with 1 bit error per corrupted packet. (b) Results with 2 bit errors per corrupted packet. (c) Results with 1 bit error per corrupted packet and only decoded in the frontward direction.

For the case where a single bit error was injected into each corrupted packet, the average number of samples exactly decoded samples from the entire signal was above 99% for each packet error percentage. For the case where two bit errors were injected into each corrupted packet, the amount of exactly decoded samples was 96% or higher for all selected packet corruption rates.(a) (b) (c)

Figure 5.8 shows a series of boxplots of the PRDs of the error injected signals after they have been restored, with any unrecoverable samples being estimated using linear interpolation. This shows that for the given cases, the average PRD remained below 5% while the percentage of corrupted packets was 10% or below.

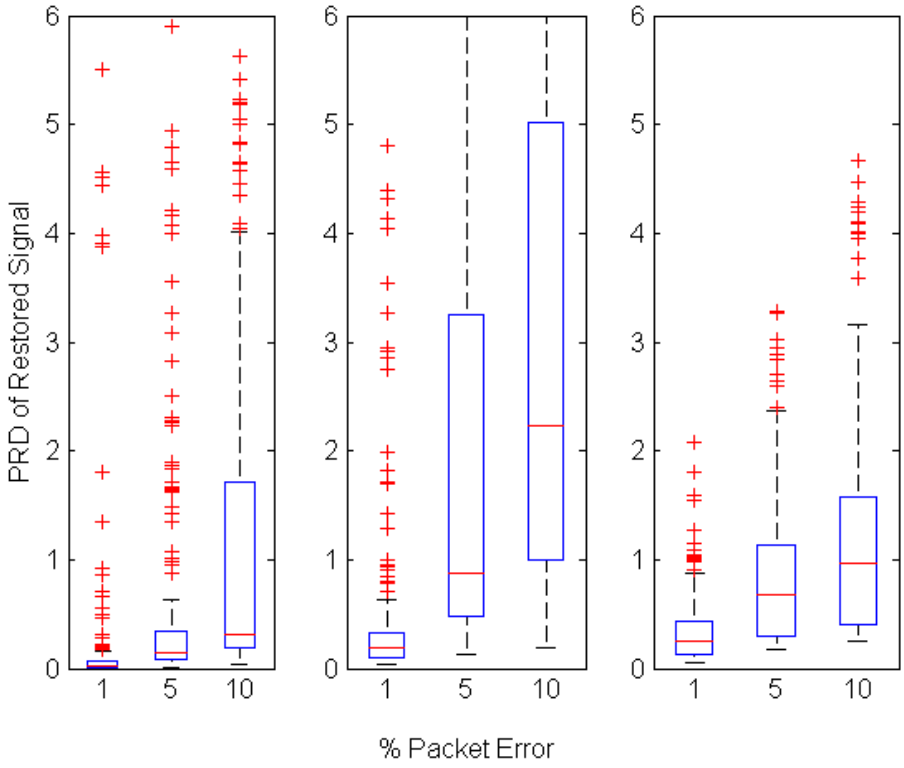


Figure 5.8: Average PRD of restored 12-bit signals with errors introduced. (a) PRD results with 1 bit error per corrupted packet. (b) PRD results with 2 bit errors per corrupted packet. (c) PRD results with 1 bit error per corrupted packet and only decoded in the frontward direction.

5.2.2 Decoding Analysis of 16-Bit ECG Signals with Injected Errors

Each encoded 16-Bit signal was injected with errors and then run through the decoder, and the total amount of unrecovered samples was recorded. This was done 15 times per signal for each of the different packet corruption rates. The first two plots in Figure 5.9 show the results for both one and two bit errors per corrupted packet. The third plot shows the single bit case again, but this time the decoder performed only frontwards decoding, but did not perform backwards decoding. This shows the improvement in the amount of recovered data that is attributed to being able to decode the packet from both directions.

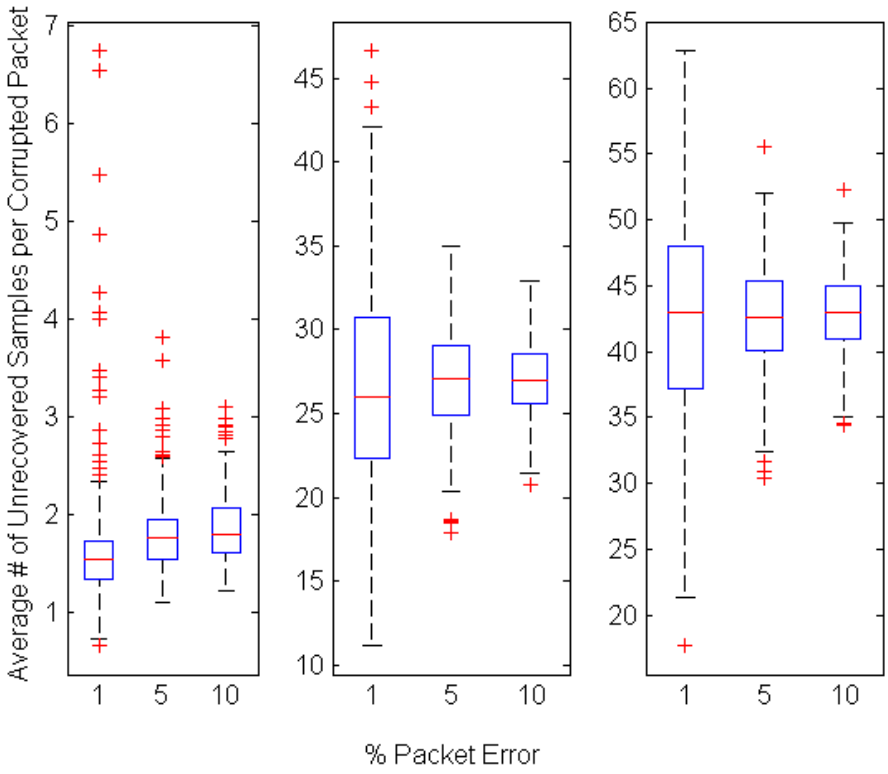


Figure 5.9: Average number of unrecovered samples at different error rates for 16-bit signals. (a) Results with 1 bit error per corrupted packet. (b) Results with 2 bit errors per corrupted packet. (c) Results with 1 bit error per corrupted packet and only decoded in the frontward direction.

Again we calculate the percentage of the signal that is exactly recovered to give a different view of our results. For the case where a single bit error was injected into each corrupted packet, the average number of samples exactly decoded from the entire signal was above 99% for each packet error percentage. For the case where two bit errors were injected into each corrupted packet, the amount of exactly decoded samples from the entire signal was above 97% or higher for all packet corruptions rates.

Figure 5.10 shows a series of boxplots of the PRDs of the error injected signals after they have been restored, with any unrecoverable samples being estimated using

linear interpolation. This shows that for the given cases, the average PRD remained below 5% while the percentage of corrupted packets was below 10% or lower.

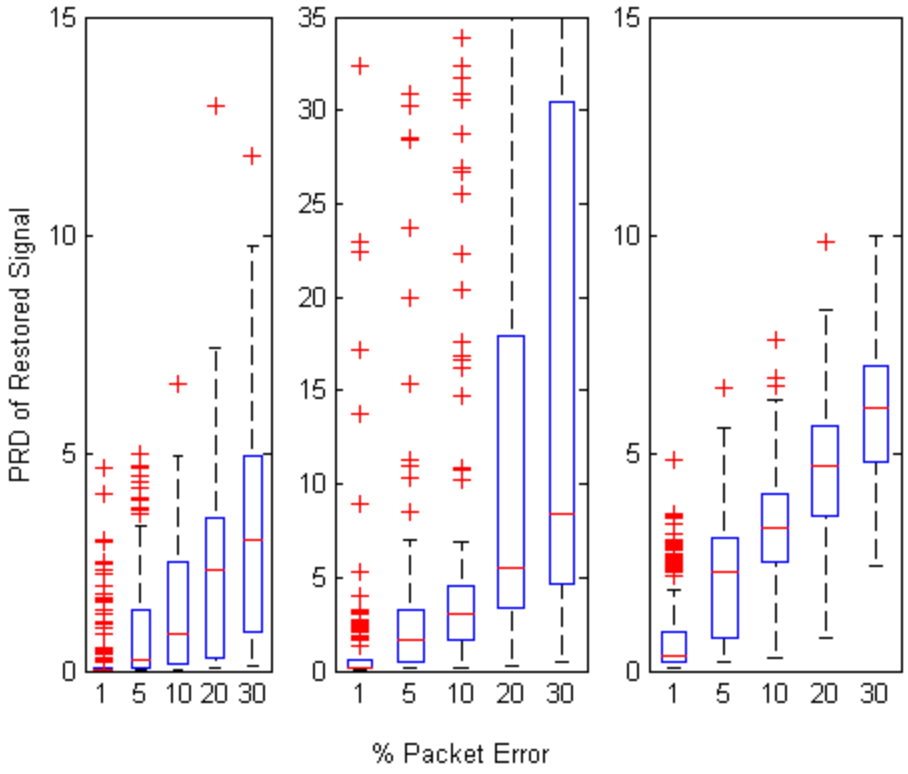


Figure 5.10: Average PRD of restored 16-bit signals with errors introduced. (a) PRD results with 1 bit error per corrupted packet. (b) PRD results with 2 bit errors per corrupted packet. (c) PRD results with 1 bit error per corrupted packet and only decoded in the frontward direction.

Conclusions

In this thesis, the objective of designing and implementing a low-energy, error resilient, scalable, lossless ECG encoder was attained. Two systems (System A and System B) were investigated with the first having the ability to recalculate the code tables at various points during the encoding of the ECG signal, and the second using a pre-defined set of code tables to encode the entire signal. The results from both systems showed that the system that used a single set of code tables for the entire signal (System B), and therefore required significantly less resources, performed as well as System A.

In addition, corruption of the data packets during transmission was simulated and the ability of the decoder to recover as much data as possible was tested. The samples that were not able to be exactly recovered were estimated using linear interpolation to provide values as close to the originals as possible. For single bit errors the average number of unrecovered samples per corrupted packet was less than three samples, and the average PRD was within a few percentage points of the 5% PRD recommended for recovery of an entire signal when the packet corruption rates were 10% or less. This held true for both the single and double bit error cases. Overall, when a single bit error was injected per corrupted packet, the decoder was able to exactly recover over 99% of the samples at any packet corruption rate, and for the case where two bit errors were injected into each corrupted packet the recovery rate of the samples from the entire signal was above 96% for both the 12 and 16-bit representations..

Both versions of the ECG encoding system were implemented in VHDL and targeted towards the Xilinx Spartan-3e (XC3S1200E) FPGA device. The implementation

of the encoding systems in this technology is easily transferrable to future generations of FPGAs.

Future Work

- Investigate more advanced methods of interpolating the corrupted samples to further improve signal reconstruction

- Investigate scrambling the encoded data prior to transmission in order to minimize the effect of burst errors

- Port design to a VLSI hardware platform to further reduce power requirements, and integrate with existing portable telemedicine systems, especially wearable ECG monitoring systems for detecting abnormal heart beats

- Explore implementation into other similar non-ECG systems

References

[1] E. Kyriacou, *et. al.*, "Integrated platform for continuous monitoring of children with suspected cardiac arrhythmias," *9th International Conference on Information Technology and Applications in Biomedicine*, pp.1-4, 5-7 Nov. 2009.

[2] K. M. Chang and S. H. Liu; "Portable sleep monitoring by ECG and accelerator and Bluetooth transmission," *2nd International Conference on Biomedical Engineering and Informatics.*, pp.1-4, Oct. 2009.

[3] J. Dong, S. Zhang, and X. Jia, "A portable intelligent ECG monitor based on wireless internet and embedded system technology," *International Conference on Biomedical Engineering and Informatics*, vol.2, pp. 553-556, May 2008.

[4] D. Lucani, G. Cataldo, J. Cruz, G. Villegas, and S. Wong, "A portable ECG monitoring device with Bluetooth and Holter capabilities for telemedicine applications," *28th IEEE Annual International Conference on Engineering in Medicine and Biology Society*, pp. 5244-5247, Aug.-Sept. 2006.

[5] "The McGill Physiology Virtual Lab - Cardiovascular Laboratory: Electrocardiogram: Introduction," http://www.medicine.mcgill.ca/physio/vlab/cardio/introECG.htm, accessed May 2011.

[6] A. L. Goldberger, *et. al.*, "PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals," *Circulation*, vol. 100, no. 23, pp. e215-e220 [Circulation Electronic Pages; http://circ.ahajournals.org/cgi/content/full/101/23/e215], June 13 2000.

[7] S. M. S. Jalaleddine, C. G. Hutchens, R. D. Strattan, and W. A. Coberly, "ECG data compression techniques-a unified approach," *IEEE Transactions on Biomedical Engineering*, vol. 37, no. 4, pp. 329-343, April 1990.

[8] M. Blanco-Velasco, F. Cruz-Roldan, F. Lopez-Ferreras, A. Bravo-Santos, and D. Martinez-Munoz, "A low computational complexity algorithm for ECG signal compression," *Medical Engineering and Physics*, vol. 26, no. 7, pp. 553-568, Sep. 2004.

[9] E. Alexandre, A. Pena, and M. Sobreira, "On the use of 2-D coding techniques for ECG signals," *IEEE Transactions on Biomedical Engineering*, vol. 10, no. 4, pp. 809-811, Oct. 2006.

[10] B. Huang, Y. Wang, and J. Chen, "2-D compression of ECG signals using ROI mask and conditional entropy coding," *IEEE Transactions on Biomedical Engineering*, vol. 56, no. 4, pp. 1261-1263, Apr. 2009.

[11] S.C. Tai, C. C. Sun, and W. C. Yan, "A 2-D ECG compression method based on wavelet transform and modified SPIHT," *IEEE Transactions on Biomedical Engineering*, vol. 52, no. 6, pp. 999-1008, June 2005.

[12] E. B. de Lima Filho, N. M. M. Rodrigues, E. A. B. Silva, S. M. M. de Faria, V. M. M. da Silva, and M. B. de Carvalho., "ECG signal compression based on Dc equalization and complexity sorting," *IEEE Transactions on Biomedical Engineering*, vol. 55, no. 7, pp. 1923-1926, July 2008.

[13] H. H. Chou, Y. J. Chen, Y. C. Shiau, and T. S. Kuo. "An effective compression algorithm for ECG signals with irregular periods," *IEEE Transactions on Biomedical Engineering*, vol. 53, no. 6, pp. 1198-1205, June 2006.

[14] C. T. Ku, K. C. Hung, T. C. Wu, and H. S. Wang, "Wavelet-based ECG data compression system with linear quality control scheme," *IEEE Transactions on Biomedical Engineering*, vol. 57, no. 6, pp.1399-1409, June 2010.

[15] M. Blanco-Velasco, F. Cruz-Roldan, J. I. Godine-Llorente, and K. E. Barner, "ECG compression with retrieved quality guaranteed," *Electronics Letters*, vol. 42, no. 23, pp. 1466-1467, Nov. 2004.

[16] S. G. Miaou and C. L. Lin, "A quality-on-demand algorithm for wavelet-based compression of electrocardiogram signals," *IEEE Transactions on Biomedical Engineering*, vol. 49, no. 3, pp. 233-239, March 2002.

[17] S. G. Miaou and S. N. Chao, "Wavelet-based lossy-to-lossless ECG compression in a unified vector quantization framework," *IEEE Transactions on Biomedical Engineering*, vol. 52, no. 3, pp. 539-543, Jan. 2005.

[18] A. Alesanco, and J. Garcia, "Automatic real-time ECG coding methodology guaranteeing signal interpretation quality," *IEEE Transactions on Biomedical Engineering*, vol. 55, no. 11, pp. 2519-2527, Nov. 2008.

[19] L. Brechet, "Compression of biomedical signals with mother wavelet optimization and best-basis wavelet packet selection," *IEEE Transactions on Biomedical Engineering*, vol. 54, no. 12, pp. 2186-2192, Dec. 2007.

[20] S. G. Miaou, H. L. Yen, and C. L. Lin, "Wavelet-based ECG compression using dynamic vector quantization with tree codevectors in a single codebook," *IEEE Transactions on Biomedical Engineering*, vol. 49, no. 7, pp. 671-680, July 2002.

[21] B. S. Kim, S. K. Yoo, and M. H. Lee, "Wavelet-based low delay ECG compression algorithm for continuous ECG transmission," *IEEE Transactions on Information Technology in Biomedicine*, vol. 10, no. 1, pp. 77-83, Jan. 2006.

[22] A. Alesanco, S. Olmos, R. S. H Istepanian, and J. Garcia, "Enhanced real-time ECG coder for packetized telecardiology applications," *IEEE Transactions on Information Technology in Biomedicine*, vol. 10, no. 2, pp. 229-236, April 2006.

[23] A. Bendifallah, R. Benzid, and M. Boulemden, “Improved ECG compression method using discrete cosine transform, “ *Electronics Letters*, vol. 47, no. 2, pp. 87-89, Jan. 2011.

[24] R. Nygaard, G. Melnikov, and A. K. Katsaggelos, “A rate distortion optimal ECG coding algorithm,” *IEEE Transactions on Biomedical Engineering*, vol. 48, no. 1, pp. 28-40, Jan. 2001.

[25 H. Kim, R. F. Yazicioglu, P. Merken, C. Van Hoof and H. J. Yoo, “ECG signal compression and classification algorithm with quad level vector for ECG Holter system,” *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 1, pp. 93-100, Jan. 2010.

[26] E. B. L. Filho, N. M. M. Rodrigues, E. A. B. da Silva, M. B. de Carvalho, S. M. M. de Faria, and V. M. M. da Silva, “On ECG signal compression with 1-D multiscale recurrent patterns allied to preprocessing techniques,” *IEEE Transactions on Biomedical Engineering*, vol. 56, no. 3, pp. 896-900, March 2009.

[27] O. Sayadi and M. B. Shamsollahi, “ECG denoising and compression using a modified extended Kalman filter structure,” *IEEE Transactions on Biomedical Engineering*, vol. 55, no. 9, pp. 2240-2248, April 2008.

[28] C. M. Fira and L. Goras, “An ECG signals compression method and its validation using NN’s,” *IEEE Transactions on Biomedical Engineering*, vol. 55, no. 4, pp. 1319-1326, April 2008.

[29] C. C. Sun and S. C. Tai, “Beat-based ECG compression using gain-shape vector quantization,” *IEEE Transactions on Biomedical Engineering*, vol. 52, no. 11, pp. 1882-1888, Nov. 2005.

[30] S. G. Miaou and H. L. Lin, “Multichannel ECG compression using multichannel adaptive vector quantization,” *IEEE Transactions on Biomedical Engineering*, vol. 48, no. 10, pp. 1203-1207, Oct. 2001.

[31] E. B. de Lima Filho, E. A. B. da Silva, M. B. de Carvalho, W. S. da Silva Junior, and J. Koiller, “Electrocardiographis signal compression using multiscale recurrent patterns,” *IEEE Trans. On Circuits and Systems I: Regular Papers*, vol. 52, no. 12, pp. 2739-2753, Dec. 2005.

[32] A. Chatterjee, A. Nait-Ali, and P. Siarry, “An input-delay neural-network-based approach for piecewise ECG signal compression," *IEEE Transactions on Biomedical Engineering*, vol.52, no.5, pp. 945-947, May 2005.

[33] J. J. Wei, C. J. Chang, N. K. Chou, and G. J. Jan, “ECG data compression using truncated singular value decomposition,” *IEEE Transactions on Information Technology in Biomedicine*, vol. 5, no. 4, pp. 290-299, Dec. 2001.

[34] E. Chua, C. C. Fu, and W. C. Fang, "VLSI implementation of a mixed bio-signal lossless data compressor for portable brain-heart monitoring systems," *IEEE International Conference on Consumer Electronics*, pp. 557-558, Jan. 2011.

[35] Y. Kamamoto, et. al., "An efficient lossless compression of multichannel time-series signals by MPEG-4 ALS," *IEEE International Symposium on Consumer Electronics*, pp. 159-163, May 2009.

[36] Q. Zhou, "Study on ECG data lossless compression algorithm based on K-means cluster," *International Conference on Future Computer and Communication*, pp. 91-93, June 2009.

[37] Z. Arnavut, "ECG signal compression based on Burrows-Wheeler transformation and inversion ranks of linear prediction," *IEEE Transactions on Biomedical Engineering*, vol. 54, no. 3, pp. 410-418, March 2007.

[38] A. Ottley and R. Boltan, "Hardware-assisted lossless ECG coder," *Canadian Conference on Electrical and Computer Engineering*, pp. 362-365, May 2005.

[39] C. D. Giurcâneanu, I. Tabus, and S. Mereuta, "Using contexts and R-R interval estimation in lossless ECG compression," *Computer Methods and Programs in Biomedicine*, vol. 67, no. 3, pp. 177–186, Mar. 2002.

[40] K, Duda, P. Turcza, and T. P. Zieliński, "Lossless ECG compression with lifting wavelet transform," *Proc. IEEE Instrumentation and Measurement Technology Congress.*, vol. 1 pp. 640-644, May 2001.

[41] Z. Arnavut, "Lossless and near-lossless compression of ECG signals," *23rd Annual Conference of the IEEE Enginering in Medicine and Biology Society*, vol. 3, pp. 2146-2149, Oct. 2001.

[42] A. Koski, "Lossless ECG encoding," *Compuer. Methods and Programs in Biomedicine*, vol. 52, no. 1, pp. 23–33, Jan. 1997.

[43] D. A. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the Institute of Radio Engineers.*, vol. 40, no. 9, 1952, pp.1098-1101.

[44] Y. Takishima, M. Wada, and H. Murakami, "Reversible variable length codes," *IEEE Transactions on communications*, vol. 43, no. 2/3/4, pp. 158-162, April 1995.

[45] C. W. Tsai and J. L. Wu, "On constructing the Huffman-code-based reversible variable-length codes," *IEEE Transactions on Communications*, vol. 49, no. 9, Sept. 2001

[46] W. H. Jeong, Y. S. Yoon, and Y. S. Ho, "Design of reversible variable length codes using properties of the Huffman code and average length function," *2004 International Conference on Image Processing*, Oct. 2004, pp. 817-820.

[47] J. J. Bailey, *et. al.*, "Recommendations for standardization and specifications in automated electrocardiography—bandwidth and digital signal processing," *Circulation*, vol. 81, pp. 730–739, Feb. 1990.

[48] Xilinx. (2009, Aug.) www.Xilinx.com. [Online]. HYPERLINK http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

[49] Y. Takishima, M. Wada and H. Murakami, "Reversible variable length codes," *IEEE Transactions on Communications*, vol. 43, no. 2/3/4, pp. 158-162, April 1995.