ECG Compression for Holter Monitoring

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Electrical Engineering
University of Saskatchewan
Saskatoon

By
Adam Ottley

©Adam Ottley, April 2007. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Electrical Engineering
3B48.1 Engineering Building
57 Campus Drive
Saskatoon, Saskatchewan
Canada
S7N 5A9

Abstract

Cardiologists can gain useful insight into a patient's condition when they are able to correlate the patent's symptoms and activities. For this purpose, a Holter Monitor is often used — a portable electrocardiogram (ECG) recorder worn by the patient for a period of 24–72 hours. Preferably, the monitor is not cumbersome to the patient and thus it should be designed to be as small and light as possible; however, the storage requirements for such a long signal are very large and can significantly increase the recorder's size and cost, and so signal compression is often employed. At the same time, the decompressed signal must contain enough detail for the cardiologist to be able to identify irregularities. "Lossy" compressors may obscure such details, where a "lossless" compressor preserves the signal exactly as captured.

The objective of this thesis is to develop and implement a low-complexity lossless ECG encoding algorithm capable of at least a 2:1 compression ratio in an embedded system for use in a Holter Monitor.

Different lossless compression techniques were evaluated in terms of coding efficiency as well as suitability for ECG waveform application, random access within the signal and complexity of the decoding operation. For the reduction of the physical circuit size, a System On a Programmable Chip (SOPC) design was utilized.

A coder based on a library of linear predictors and Rice coding was chosen and found to give a compression ratio of at least 2:1 and as high as 3:1 on real-world signals tested while having a low decoder complexity and fast random access to arbitrary parts of the signal. In the hardware-assisted implementation, the speed of encoding was a factor of between four and five faster than a software encoder running on the same CPU while allowing the CPU to perform other tasks during the encoding process.

Acknowledgements

Thanks go to my supervisor, Dr. Ron Bolton, for his guidance and financial support in the development of this thesis, to the Department of Electrical Engineering as well for financial assistance, and most importantly to my Father, Richard Ottley, for the moral support given as I worked toward this goal. Additional thanks go to the Canadian Microelectronics Corporation for providing the Altera Nios development kit.

This thesis is dedicated to the memory of Jerry Huff.

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents v

List of Tables vii

List of Figures viii

List of Abbreviations ix

1 Introduction 1
- 1.1 Medical . . . 1
  - 1.1.1 Electrocardiography . . . 1
  - 1.1.2 Heart Monitoring . . . 4
- 1.2 Digital Data Compression . . . 5
  - 1.2.1 Theory . . . 6
  - 1.2.2 Lossy Compression . . . 6
  - 1.2.3 Lossless Compression . . . 8
- 1.3 Objectives . . . 16

2 Altera Development Platform 17
- 2.1 System On a Chip . . . 17
  - 2.1.1 System On a Programmable Chip . . . 19
- 2.2 Altera SOPC platform . . . 20
  - 2.2.1 Nios CPU . . . 20
  - 2.2.2 Avalon Bus . . . 23
  - 2.2.3 SOPC Builder . . . 25
- 2.3 Base System . . . 26
  - 2.3.1 CompactFlash Host . . . 28

3 Encoding Algorithm Design 35
- 3.1 Waveform Coding . . . 35
  - 3.1.1 Partitioning . . . 36
  - 3.1.2 Linear Predictive Coding . . . 37
  - 3.1.3 Transform/Hybrid Coding . . . 46
  - 3.1.4 ECG Characteristics . . . 47
  - 3.1.5 Platform Concerns . . . 48

3.2 Encoding Algorithm . . . 49
3.2.1 Objectives and Requirements . . . 49
3.2.2 Interchannel Decorrelation . . . 50
3.2.3 Intrachannel Decorrelation . . . 51
3.2.4 Entropy Coding . . . 56

4 Encoder Implementation **59**
4.1 Interface . . . 59
4.1.1 Analysis . . . 60
4.1.2 Coder . . . 61
4.2 Coder Implementation Design . . . 61
4.2.1 Analysis . . . 61
4.2.2 Coder . . . 63

5 Results **68**
5.1 Compression Ratio . . . 69
5.2 Speed . . . 71

6 Discussion and Conclusions **73**
6.1 Objectives . . . 73
6.2 Performance Analysis . . . 74
6.2.1 Compression . . . 74
6.2.2 Speed . . . 75
6.3 Future Work . . . 76
6.3.1 Hardware Implementation . . . 76
6.3.2 Codec Design . . . 77

References **82**

A Verilog Code **83**
A.1 Instruction (cinst_lpcrice.v) . . . 83
A.2 Analysis (analyze.v) . . . 90
A.3 Rice Coder (rice_coder.v) . . . 95

B C Code **100**
B.1 CompactFlash Host Driver (cfdrv.c) . . . 100
B.2 FAT Filesystem Driver (fat.c) . . . 105
B.3 FLAC Library (flac.c) . . . 131
B.4 FLAC Data Types (flac.h) . . . 145
B.5 FAT Filesystem Data Types (fat.h) . . . 145

C List of MIT-BIH Database Records Tested **148**

List of Tables

2.1 Nios Register Groups 21
2.2 Nios Custom Instruction Input Ports 23
2.3 CompactFlash ATA registers 29
2.4 Avalon signals used in the CompactFlash peripheral 31
2.5 Important Identify Device results for a 16MB CF storage card 32

3.1 Numbers 0–4, Rice coded using parameters 0–3 45
3.2 Block count for each predictor order 56

4.1 List of custom instruction prefix values & functions 60

5.1 Compression Ratio of encoded ECG files 70
5.2 Cycles required to encode MIT-BIH ECG signals using hardware and software 72

List of Figures

1.1 Sample ECG signal . . . 3
1.2 The Components of an ECG Signal . . . 3
1.3 Block Diagram of a Lossy Encoder/Decoder System . . . 7
1.4 An ECG and a Piecewise Linear Approximation . . . 9
1.5 Block Diagram of a Lossless Encoder/Decoder System . . . 9
1.6 Example Code Tree . . . 13

2.1 A System-On-a-Chip using an AMBA-like bus . . . 19
2.2 A Custom Instruction as part of the Nios CPU . . . 22
2.3 Nios Development Board Block Diagram (From [13]) . . . 27
2.4 CompactFlash Waveform Timing Configuration . . . 33

3.1 General Structure of a Linear Predictive Waveform Coder . . . 37
3.2 General Structure of a Linear Predictive Decorrelator . . . 40
3.3 General Structure of a Linear Predictive Reconstructor . . . 42
3.4 Structure of a Lossless Transform Coder . . . 47
3.5 Close-up of an ECG waveform, showing individual samples . . . 52
3.6 Example Frequency Spectra of ECG signals . . . 53
3.7 Frequency Response of Predictors . . . 54
3.8 Example Probability Density Functions of ECG Signals and their Residuals . . . 57
3.9 Optimal Rice Parameters versus Residual Totals . . . 58

4.1 Block diagram of the analysis function . . . 62
4.2 Pseudocode for the analysis process . . . 62
4.3 Block Diagram of the Instruction . . . 64
4.4 Block Diagram of the Rice Coder . . . 65
4.5 Pseudocode for the encoding process . . . 67

List of Abbreviations

ADC	Analog to Digital Converter
ALAC	Apple Lossless Audio Codec
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application-Specific Integrated Circuit
ATA	AT Attachment
BWT	Burrows-Wheeler Transform
CHS	Cylinder/Head/Sector
CIS	Card Information Structure
CODEC	Coder/Decoder
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DPCM	Differential Pulse Code Modulation
DRQ	Data Request
DSP	Digital Signal Processing/Processor
DVD	Digital Versatile Disc
ECG	Electrocardiogram
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIR	Finite Impulse Response
FLAC	Free Lossless Audio Codec
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
HDL	Hardware Description Language
IDE	Internal Drive Electronics
IIR	Infinite Impulse Response
I/O	Input/Output
IRQ	Interrupt Request
JPEG	Joint Photographic Experts Group
JTAG	Joint Test Action Group
LBA	Logical Block Addressing
LED	Light Emitting Diode
LPC	Linear Predictive Coding
LPCM	Linear Pulse Code Modulation
LTAC	Lossless Transform Audio Coding
MAC	Media Access Control
MIT-BIH	Massachusetts Institute of Technology-Beth Israel Hospital
MPEG	Moving Picture Experts Group
MSTEP	Multiply Step
MUL	Multiply

PCB	Printed Circuit Board
PCMCIA	Personal Computer Memory Card Industry Association
PDA	Portable Digital Assistant
PHY	Physical Layer
PRD	Percent Root Mean Square Distortion
PWL	Piecewise Linear
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random-Access Memory
SOC	System On a Chip
SOPC	System On a Programmable Chip
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

Chapter 1

Introduction

1.1 Medical

As the centre of the human circulatory system, the heart has long been studied by medical professionals. Its importance in the operation of the human body has led to the development of an entire branch of medicine dedicated to studying the heart and the diseases related to it, called *cardiology*.

1.1.1 Electrocardiography

The heart functions by pacemaker cells generating a pulse of electricity at the top of the heart, causing that part to contract. The pulse travels through the different parts of the heart muscle, causing them to contract in a sequence that causes the circulation of blood throughout the human body.

Although the heart is the main target of the electrical pulses, the pulses do spread outward through the rest of the body. However, by the time it reaches the skin the pulse is very weak due to attenuation by body tissue and bone. In the late 1800s, successful attempts were made to measure the electrical activity of a frog's heart by direct measurements on the heart itself, but doctors wanted to find a non-invasive method. It was in 1903 that Willem Einthoven solved the issue by improving the sensitivity of the sensors using a string galvanometer. For the development of the first working electrocardiograph, Einthoven was awarded the Nobel Prize in Medicine

in 1924 [1].

An electrocardiogram (ECG) is a graph that show the electrical activity of the heart muscle. Cardiologists use the ECG as a tool for several uses including:

- detecting abnormalities in the heart's activity (cardiac arrhythmia),
- detecting electrolyte disturbances,
- providing information on the physical condition of the heart (e.g., hypertrophy, improper valve operation, trauma), and
- screening for Ischaemic heart disease (reduced blood supply)

The heart's electrical activity is measured by electrodes that are placed on the skin. Initially, sensors were placed at points on the left arm, right arm and left leg that are electrically equidistant from the heart. Each pair of points from the three are standard measuring locations, called I, II and III. Since then, nine additional standard measurements have been added using a central point V near the heart as the reference along with the three limb leads aVL, aVR, and aVF as well as six precordial (chest) leads V1–V6. Each lead reflects different portions of the heart and gives clues as to which part of the heart muscle is affected by the disorder [2]. The graphs generated from each lead vary in appearance due to the differing electrical pulse strengths and polarities at each measuring point.

A typical ECG signal as recorded from lead II is seen in Figure 1.1. The typical resting heart rate varies by gender, age, and physical condition, but for adults in general, it is about 70–75 beats per minute, with 50–100 being considered the normal range.

Each beat in an ECG signal consists of several waves, named P through T, with several intervals (illustrated in Figure 1.2) of the beat named after the waves that comprise them. The characteristics of these components can indicate the type of disorder affecting the patient. Characteristics such as the length of an interval,

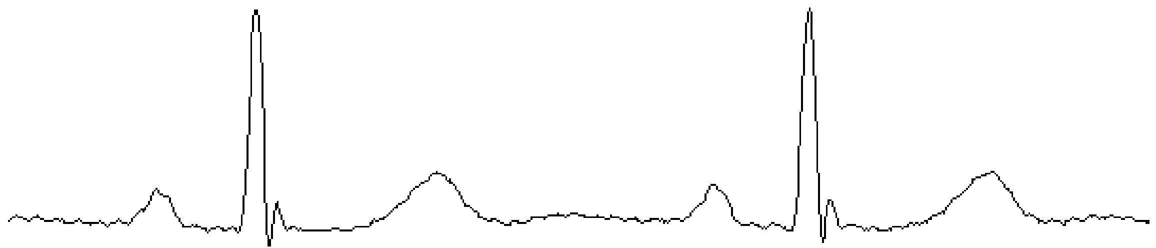


Figure 1.1: Sample ECG signal

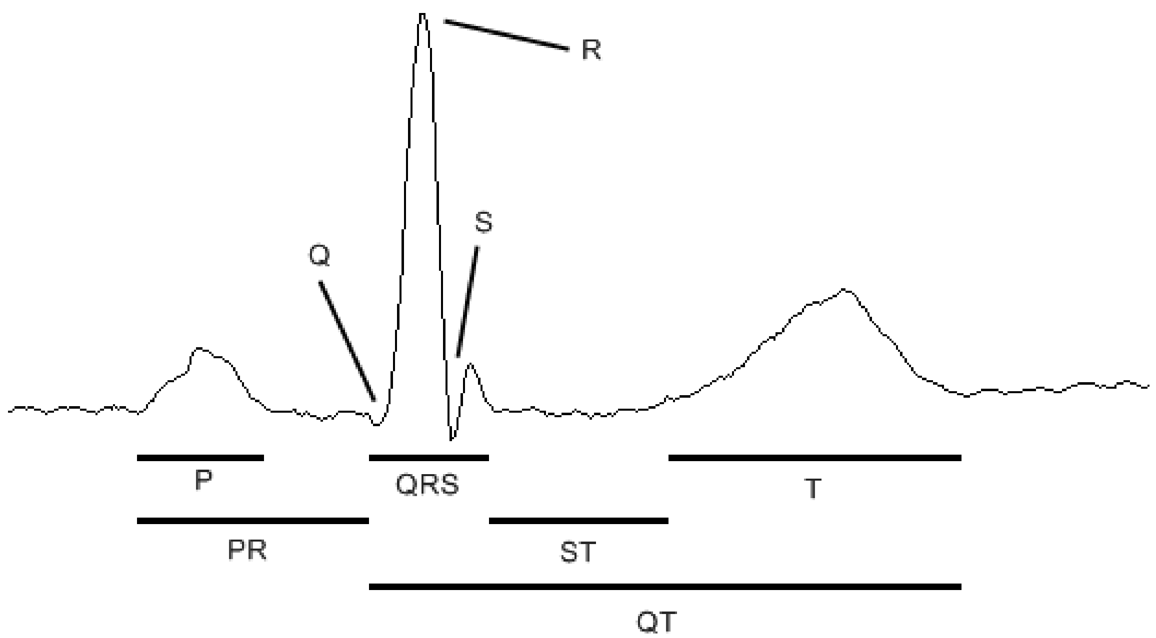


Figure 1.2: The Components of an ECG Signal

or the polarity, amplitude or elevation/depression of a beat component are taken into account by a specialist when diagnosing a patient. For example, the leads I, II and aVF are used to determine the 'electrical axis' of the heart, appearing as a vector from the origin in an X-Y graph. The quadrant in which the axis lies can be indicative of afflictions such as right ventricular hypertrophy, or something as innocuous as transposed electrodes.

1.1.2 Heart Monitoring

Normally, an electrocardiogram is recorded at a hospital or a medical laboratory using a large nonportable recorder with the patient at rest or doing a limited set of activities such as running on a treadmill. As a result, the circumstances under which a patient's symptoms arise may not be possible under such monitoring.

A **Holter monitor**, or **ambulatory electrocardiography device**, is a portable ECG monitor for recording a patient's heart activity for 24 hours or longer, normally taking from two to seven measurements on the chest rather than the full 12. The long recording period makes it more likely that an intermittent irregularity will be recorded, and its portability permits recording during the patient's normal activity, where some symptoms are most likely to occur. Patients are also often asked by the doctor to keep a diary of activities and perceived symptoms, which the doctor uses to correlate with irregularities found in the recording. An **Event monitor** is a similar device that is worn for longer periods and records only when the patient or caregiver activates it manually.

Analog Holter monitors record low-speed onto magnetic tape, usually standard 60 or 90-minute cassette tapes. While it is possible for digital Holter monitors to record to tape, more often a digital memory such as flash memory is used.

The potential of a digital Holter/event monitor goes beyond simple continuous recording. The device could be programmed to continuously acquire but may not

store the entire length of signal, instead only committing the signal to storage when an unexpected deviation in the incoming signal occurs, or when prompted by the wearer. In such cases, a chosen length of signal prior to the current acquisition can be stored in memory and committed to storage along with the incoming signal so that the heart activity leading up to the decision to start recording can be stored as well.

Miniaturization is a primary design goal of a Holter monitor. It is very important that the device not be cumbersome to the patient so that it does not interfere with the wearer's activity. As such, the minimum-sized components necessary must be used. However, recording for such a long period of time can require a great amount of storage, especially when multiple sensors are used. A three-lead recording at a sampling rate of 400Hz and a sample size of eight bits for 24 hours stored in linear pulse code modulated (LPCM) format would require approximately 99MB of storage. While compression is often employed to reduce the size of the stored data, most methods in use lower the quality of the signal by discarding details considered to be insignificant by the encoder. With the removal of information comes the possibility that details that are of importance to the cardiologist may be lost.

When not using compression, reducing the storage requirements can necessitate lowering the recording parameters such as sample rate and sample size, which reduces the precision of the recorded signal.

1.2 Digital Data Compression

The objective of compressors is to reduce the size of a piece of data. The use of the term *coder* is preferred as it is a more accurate description of the function of the system. As we will see, in some cases a reduction in data size cannot be guaranteed, so the term *compressor* isn't necessarily accurate.

1.2.1 Theory

The **entropy** of a signal is the measure of the amount of information (randomness) in that signal [3]. Compression happens when data that correlates with other portions of the signal is replaced by more concise representation, e.g. a reference to the data at that location, or an efficiently encoded difference between it and a previous piece of data. When all correlation has been removed, the signal is completely random and is at its smallest possible size, which is the size of the signal's entropy. By passing the encoded signal through the decoder, which is the inverse system of the encoder, the original signal can be reconstructed.

Another way to view correlation is as *statistical redundancy.* In a piece of English text, the five vowel characters comprise approximately 40% of the letters in a typical message, despite being less than 20% of the alphabet. Also, letter combinations can be common or rare; the letter S is often followed by the letter H, while a D followed by a W is comparatively infrequent.

1.2.2 Lossy Compression

In a lossy compression system as in Figure 1.3, the encoder discards information that it deems to be insignificant in order to reduce the data size below the entropy size. The decoded signal $\tilde{x}[n]$ is an approximation to the original signal $x[n]$, usually in the domain (time, transform) most pertinent to the application. In some applications, a lossy compressor is entirely unacceptable as any change in the message may have unwanted consequences, such as when compressing a computer executable program, or in compressing text where the loss of diacritical marks or a change in whitespace may result in misinterpretation by the reader — for example, the French word où (where) becoming ou (or).

The decision of what type of lossy encoder to use for a given application is

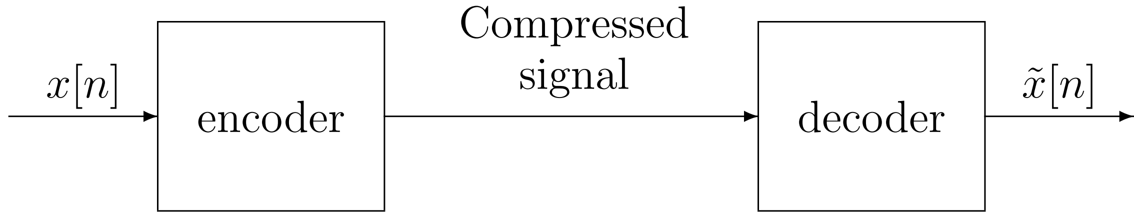


Figure 1.3: Block Diagram of a Lossy Encoder/Decoder System

determined by the characteristics of the signal and the way in which the signal will be viewed/perceived by the observer.

Lossy waveform encoders use one of two types of approximation:

Time domain The goal of time domain approximation is to assemble a waveform with a minimum of mathematical difference between it and the original. This is the type desired in ECG applications, as the cardiologist analyzes the signal by viewing the waveform on a screen.

Transform domain In common use in perceptual codecs used for compressing images, audio or video. Examples of transforms used include the Fast Fourier Transform and the (Modified) Discrete Cosine Transform, the latter of which is used extensively for its energy compaction properties in audio/video codecs such as MP3 audio and MPEG-2 video.

As mentioned, lossy ECG compression is typically done in the time domain as it is viewed by the cardiologist as a waveform on a screen or piece of paper. There are several types of time domain approximation, two well-known types of which are:

Differential Pulse Code Modulation (DPCM) The difference between successive samples is calculated and stored using fewer bits than would be used for the raw samples themselves. Loss is incurred when the difference between two samples is greater than can be represented using the selected number of bits. The net effect is a limit on the slope of the stored signal, and as a result DPCM

is best suited to signals that do not rapidly change amplitude.

Piecewise Linear (PWL) Approximation The signal is analyzed and the encoder decides on a series of linear segments that best approximate the acquired signal. For each segment only one or two pieces of data need be stored: the beginning and end points (in some systems the starting point of a segment is not assumed to be the end point of the previous segment). Figure 1.4 shows an ECG waveform and a PWL approximation of it.

1.2.3 Lossless Compression

A lossless compressor is one where the output of the decoder is identical to the input to the encoder, as shown in Figure 1.5.

The term *compressor* is not entirely accurate when applied to a lossless system. Let us assume that we have a compressor that transforms each input message into a distinct, smaller message. Consider the set of messages that are at most N bits long. The total number of messages in the set is

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1, \tag{1.1}$$

assuming we include the zero-length message. Now consider the set of messages that are fewer than N bits long, which has

$$\sum_{i=0}^{N-1} 2^i = 2^N - 1 \tag{1.2}$$

members. The larger set represents the range of inputs to the compressor, and the smaller set represents the range of possible outputs, so long as the assumption holds true. However, the fact that the input set is larger disproves the possibility of such a compressor existing; one cannot map each member of the input set uniquely to a member of the output set. Also, if an input message of k bits long maps to an output message of $j < k$ bits, then at least one input message of no more than j bits

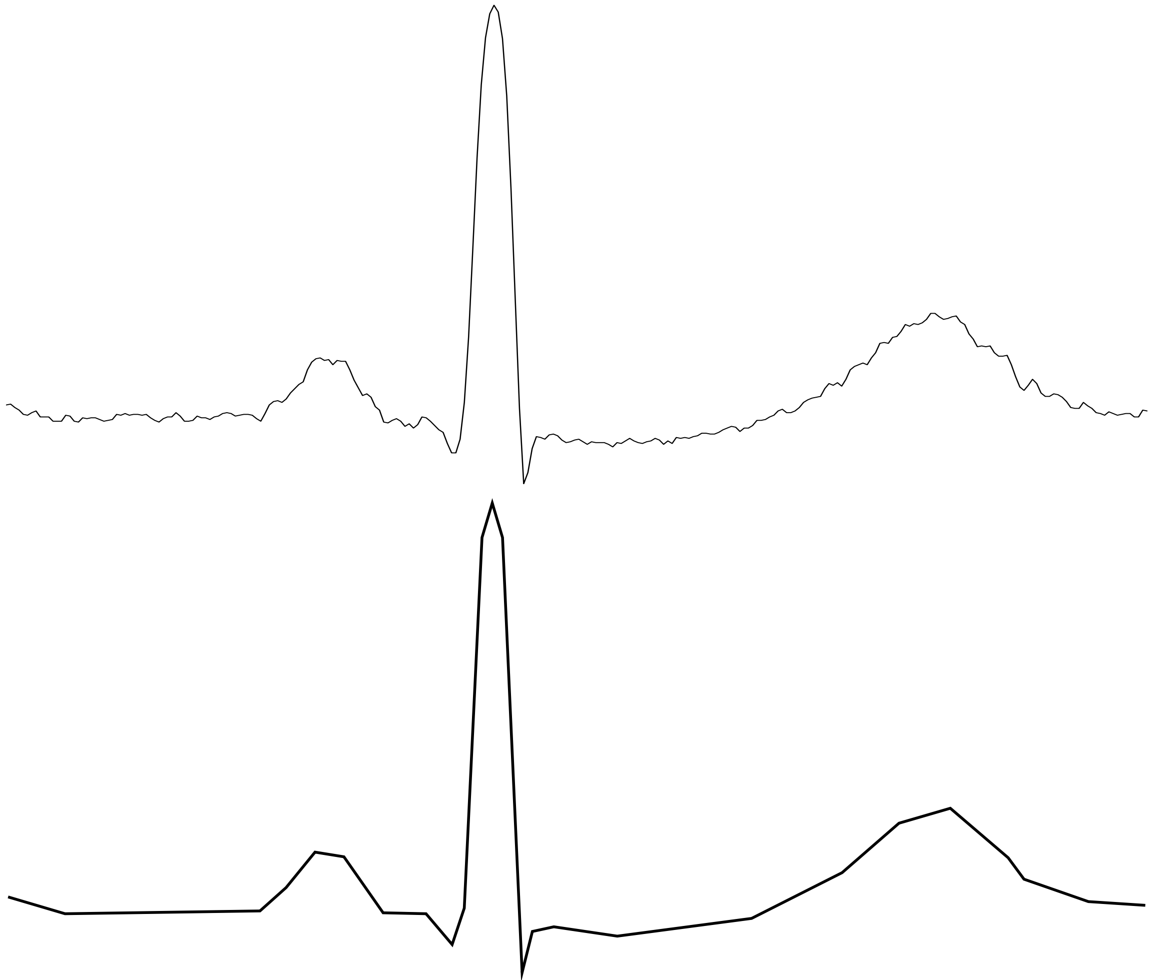


Figure 1.4: An ECG and a Piecewise Linear Approximation

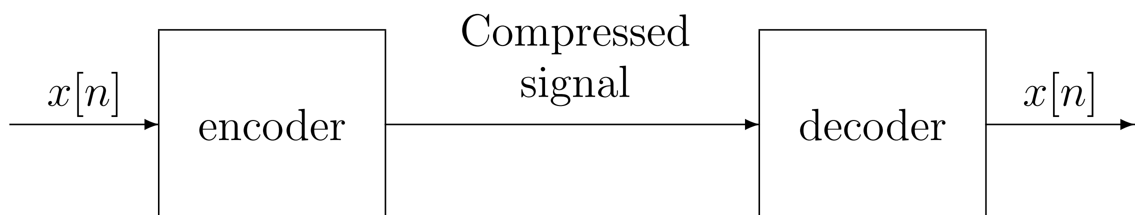


Figure 1.5: Block Diagram of a Lossless Encoder/Decoder System

long must map to an output message that is longer than j bits. Thus any lossless compressor that decreases the size of some messages must necessarily make some other messages longer. It is for this reason that for a lossless system, the term *coder* is more accurate and will be used in this paper from here on.

Because the length of the output may vary for different but equal-length input signals, lossless codecs are said to be *variable bit rate* by nature. This characteristic is important to design for particularly in random-access streams. With a *constant bit rate* encoder, the stream position of a given encoded portion of the signal is implied by the position of its decoded counterpart in the original signal.

1.2.3.1 Techniques

Lossless encoders are typically designed to be most effective on a specific type of data. While most are general-purpose in that they are capable of accepting any binary data as input, different types of data may have correlation appear in subtly different ways which may not be properly exploited by an encoder due to assumptions in its design, resulting in suboptimal size reduction.

Text data is an excellent example. Each character is stored in an integer number of bytes, and certain characters and combinations thereof are more common than others. Therefore text-oriented encoders are meant to work on byte streams and exploit repetitions of byte combinations which appear in the text as letter combinations, words and phrases. The **LZ77** method of encoding maintains a sliding window of a certain number of past characters, with the position being operated on being at the end of the window. If the encoder sees that the most recent input already appears in the window, it is replaced in the output stream by a pointer — a position and length pair that occupies less space than the phrase it references.

Video is a sequence of images that can be exploited because most of the time one frame is very similar to the next. The differences between frames are therefore small

and can be stored more efficiently than the frames themselves. Scene changes are detected by the encoder as consecutive frames with little correlation between them, and at those points the first frame of the new scene is encoded to a “keyframe” rather than storing the large difference between it and the previous image.

It is often beneficial to perform a transformation on the data to make redundancy easier to identify and remove. An example of this is the Burrows-Wheeler Transform (BWT) used in the **bzip** family of encoders. While operations such as grouping all instances of each character together can create a far more compressible file, the BWT is interesting in that it frequently makes a file more compressible and is also reversible, so that the original file can be reconstructed perfectly.

Waveforms, especially those acquired from an analog to digital converter, are difficult to compress without the use of a transformation. Two portions of the waveform that look identical to an observer viewing them on a screen may actually be very subtly different due to noise or inexact sampling instants, preventing exact matches.

1.2.3.2 Entropy Coding

Any message can be disassembled into the set of symbols from which it is made, for instance turning a text message into letters, whitespace, and punctuation marks, with the full set of symbols comprising the message being called the alphabet. Entropy coding assigns variable-length codes to the symbols in an alphabet based on their probabilities. The most frequently-seen symbols are given the shortest codes in order to minimize the size of the encoded stream. The least frequent symbols are given longer codes. A message with a compact probability density function (that is, a message with many high-probability symbols) is more susceptible to entropy coding and will have a smaller encoded size than a message that has many symbols with near-equal probabilities.

A common form of entropy code is a “prefix code”, also known as a prefix-free code or a comma-free code. Since entropy codes are variable length, the receiver/decoder must have some way of knowing where one code ends and another begins. The lack of a separating marker in a prefix code is what leads to the term comma-free code. Suppose that the letter A is sent as the code 001, the letter B is sent as 1, and the letter C is sent as 0011. In a non-comma-free code, the word “CAB” would be sent as 0011,001,1. However, most machines and communication systems represent everything in binary, and adding a third state would be expensive and cumbersome.

So a separating marker cannot be used, and some other method of telling one code from another must be found. The easiest method is to make all codes equal length. However, this completely negates the benefits of entropy coding, and a reduction in data size would only happen if the set of symbols that appear in the message is small enough to reduce the number of bits per symbol, which is a rare situation. If the variable-length codes are used without a comma, the word “CAB” becomes 00110011, which is ambiguous. The decoder does not have enough information to decide if the message is 001 1 001 1 (ABAB), 001 1 0011 (ABC), 0011 001 1 (CAB), or 0011 0011 (CC).

So to remedy the situation a rule must be instituted: No code can be split into two pieces and have the first piece be identical to another code in full. In the given example, the code for the letter C can be split into 001 1. Since 001 is the code for the letter A, the example code violates the rule. With the rule in place, the receiver/decoder is always able to know where a code ends so long as the code’s start point is known.

The prefix-free attribute allows the depiction of the code assignments as a tree. Consider an alphabet containing the letters A through F with the code assignments A = 01, B = 0010, C = 000, D = 10, E = 11, and F = 0011. The code tree for this assignment is seen in Figure 1.6. That symbols only appear at the leaves of the tree

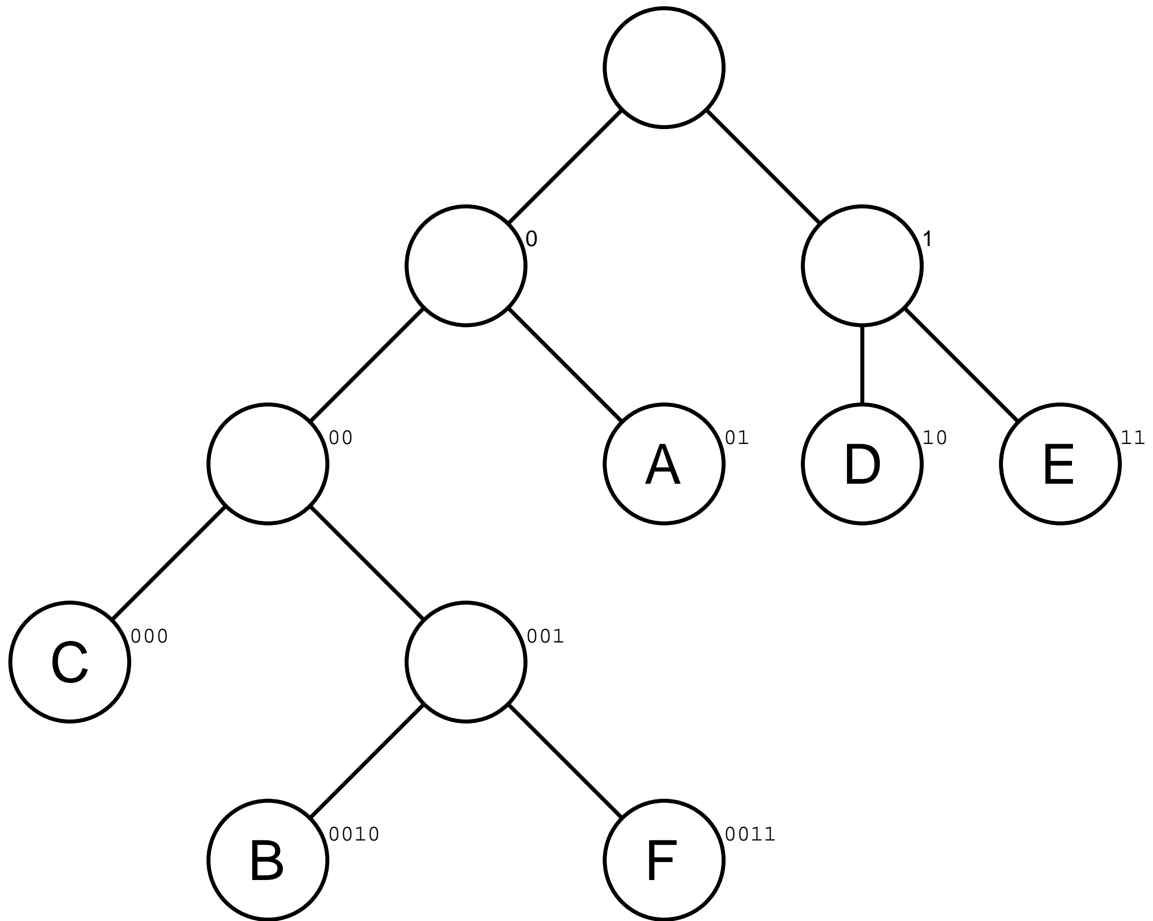


Figure 1.6: Example Code Tree

and not at any junctions makes this a prefix-free code.

The optimal code length for a given symbol is $-log_b P$, where b is the number of symbols in the alphabet and P is the probability of that particular symbol.

A symbol is an arbitrary piece of the message. The set of symbols is decided based on the nature of the message — for example, for a text message the set of symbols can be the set of characters that appear in the message. However, if a certain combination of characters frequently appears in the message, it may be beneficial to define that combination as a symbol in addition to the characters that comprise the combination.

The main difference between entropy coding methods is in how the codes are assigned to the symbols. The optimal method in terms of encoding efficiency is to

calculate the probabilities of each symbol and assign codes based on that. There are two well-known algorithms for doing this:

Shannon-Fano coding A set of all symbols and their probabilities is constructed. The set is then split into two subsets whose probabilities are as close to equal as possible, with one subset being assigned codes that begin with 0 and the other given codes beginning with 1. The process is repeated with each subset to determine subsequent bits of the codes, until all remaining subsets consist of only one symbol. This can be seen as building a code tree from the top down, since the process begins at the root of all branches.

Huffman coding Huffman coding [4] is similar to Shannon-Fano coding except that the tree is constructed from the bottom up rather than the top down. Each symbol is a trivial binary tree with the symbol as the only node. The trees are sorted by frequency in the message and the two least frequent trees are joined as leaves of a new root node into a single tree. The sorting and combining is repeated until a single tree is constructed. Less frequent symbols have more branches between their leaves and the root and are encoded with a greater number of bits. The term **Huffman code** is sometimes used to refer to any prefix-free code, even when Huffman's algorithm was not used to construct the code tree.

Other encoding schemes use a fixed table of symbol-code pairings, also called a codebook. Such mappings are created to provide optimal encoding efficiency for alphabets with given probability distributions and in some cases allow a code and its corresponding symbol to be derived from each other without the use of a lookup table.

The simplest form of a fixed-codebook mapping is **Unary Coding**, which encodes a positive integer n with $n - 1$ binary ones terminated by a zero. Thus, the number 4 would be represented as 1110.

Other fixed codes include:

Elias-Gamma Coding The number to be encoded k is written out in truncated binary (i.e., using n bits, where $n = (\log_2 k + 1)$, rounded down). One is subtracted from the number of bits written and that many zeros are prepended to indicate the code length. The result is a mapping of $1 \rightarrow 1$, $2 \rightarrow 010$, $3 \rightarrow 011$, $4 \rightarrow 00100$, etc.

Fibonacci Coding The largest number in the Fibonacci sequence (1, 1, 2, 3, 5, 8...) equal to or less than the number to be encoded n is subtracted from n, and the result is kept. The Fibonacci number subtracted is identified as the kth unique Fibonacci number and the kth bit of the code is set to a 1. The process is repeated on the result until the result is 0, and a 1 is appended to the code. A Fibonacci code has a mapping of $1 \rightarrow 11$, $2 \rightarrow 011$, $3 \rightarrow 0011$, $4 \rightarrow 1011$, etc. Fibonacci coding is notably resilient to errors. In most other codes, a single bit error will result in the stream following the error being read incorrectly. Since, in a Fibonacci code, two consecutive one bits always appear at the end of a code and nowhere else, reading in a zero bit will always stop the propagation of an error.

Golomb Coding In Golomb Coding [5], a parameter p is chosen based on the variance of the alphabet's probability distribution. To obtain the code corresponding to a number, the parameter is used as a divisor on the number, giving a quotient and a remainder. The quotient is stored in a form of unary coding where the number n is represented by n ones followed by a zero, allowing the representation of the number zero. The remainder is Huffman coded and appended to the unary-coded quotient, giving the completed code. Golomb codes are the optimal mappings for exponentially-decaying alphabets of positive integers [6].

Note that while the examples given can only represent positive integers, it is possible to use these codes with the set of all integers by mapping the set of all numbers to the set of positive numbers, which is possible since both sets are infinite.

Many Audio/Video codecs define codebooks that are efficient in most situations to allow for simpler encoders used in embedded recorders that do not need to calculate symbol probabilities and assemble a code tree. Joint Photographic Experts Group (JPEG) images are an example that allows this.

1.3 Objectives

Given the need for Holter Monitors to be as portable as possible, an integrated system using as few physical devices as necessary is clearly advantageous to use. At the same time, the large amount of data collected by a Holter Monitor can become cumbersome to store and transmit. The objectives of this thesis are to:

- develop a lossless encoding algorithm that is:
    - low-complexity, and
    - capable of reducing the data size of ECG signals by a ratio of at least 2:1,
- implement the algorithm in a CPU custom instruction for an embedded system on a programmable chip, and
- accelerate the encoding process compared to a software-only encoder.

Chapter 2

Altera Development Platform

This chapter describes the hardware platform upon which the coder was designed, starting with the concepts of the type of computer system used and its benefits for this project.

2.1 System On a Chip

Normally, a computer system consists of multiple discrete chips connected on a Printed Circuit Board (PCB). Normal functions include a processor, input/output (I/O) handling, storage and communication. A System On a Chip (SOC) is a computer system where several functions are integrated onto a single silicon die. The use of such a design has several benefits for embedded systems in particular.

Embedded systems are small computer systems that are designed and programmed to handle specific tasks, unlike general-purpose computers. Embedded systems normally have fixed hardware specifications chosen for the tasks they are designed to perform. Examples of embedded systems are a digital wristwatch, a Digital Versatile Disc (DVD) video player, or a digital Holter monitor. They may also be components of a larger system. Modern automobiles have several embedded systems controlling various components such as traction control, dashboard display, and anti-lock brakes.

There are several design goals that are common to most embedded systems for which SOC designs are attractive. By needing to manufacture a single device in place of several, the chip production costs can be lowered significantly. The PCB cost can

also be reduced as less surface area is used and the inter-device transmission lines are moved on-chip, possibly reducing the number of circuit board layers needed. The reduced space requirements also have the benefit of making the device as a whole smaller and less bulky. As well, assembly costs are lower for a single-chip design.

Being non-modular, it is difficult to replace a failed component in an embedded system. With fewer chips and solder points, a SOC has fewer potential points of failure. Having the components close together on the same chip also avoids the performance penalties of routing signals off-chip through PCB traces. Single-chip systems also consume less power.

Typical components in a SOC are:

- Microprocessor, microcontroller or digital signal processor (DSP)
- Memory (ROM, RAM, EEPROM)
- Controllers for external mass storage (hard disk, removable flash)
- Communication controllers (serial, parallel, ethernet)
- Direct Memory Access (DMA) controller
- Timers
- Analog interfaces e.g., A/D and D/A Converters
- Power Management circuitry

The design of a SOC is very modular. The components are connected to a common on-chip bus, the most popular of which is the Advanced Microcontroller Bus Architecture [7]. As such, a lot of attention is paid to interfaces to make the system components technology-independent and interoperable. This allows the designer to select only the functions needed for the system and combine them with a minimum of effort. A SOC system is illustrated in Figure 2.1.

Having so many components of a system integrated into a single chip has its tradeoffs. Each component takes space on the die, and in order to keep the overall chip size down, the components tend to not be as complex and sophisticated as

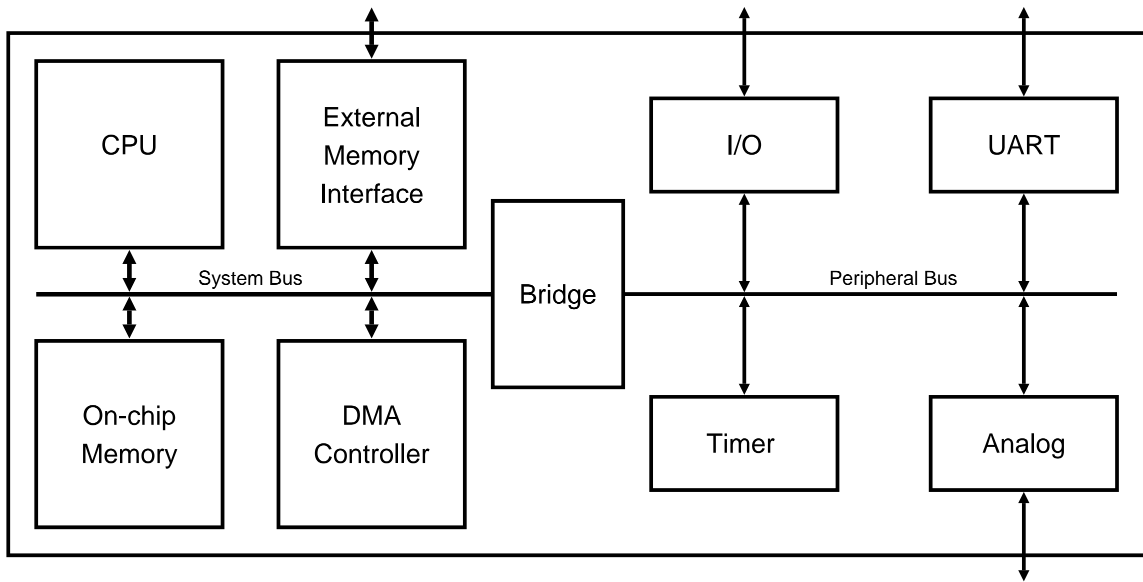


Figure 2.1: A System-On-a-Chip using an AMBA-like bus

those in a general-purpose computer. However, this is not a concern for embedded systems, which typically focus on a limited set of tasks and are not required to be versatile.

2.1.1 System On a Programmable Chip

A System On a Programmable Chip (SOPC) is a SOC instantiated in a programmable device such as a Field Programmable Gate Array (FPGA).

The use of a programmable device brings flexibility to the design process. Fixed Application-Specific Integrated Circuit (ASIC) designs need to be brought to manufacturing at some point, and should a need to make changes to the system arise after that point, the penalties in terms of time and money are large. By contrast, programmable devices can be easily reconfigured when the system is modified, and they allow for system-level testing earlier in the design process. For low-volume products, using programmable devices can be significantly cheaper than designing and manufacturing an ASIC.

One notable difference between SOCs and SOPCs is that SOPCs do not tend to have as much on-chip memory as SOCs. Memory in Programmable devices is much less dense than can be achieved with a custom design, and so SOPCs devote on-chip resources mostly to logic and use external devices for memory.

2.2 Altera SOPC platform

The SOPC platform used in this project is from Altera. It uses the Nios CPU [8] and communication between components is handled via the Avalon bus [9]. Development of the system is done through the SOPC Builder portion of the Altera Quartus II software package.

2.2.1 Nios CPU

The cornerstone of the Altera SOPC platform is the Nios 3.0[1] CPU. It is a soft-core CPU in that it is written in Hardware Description Language (HDL) that is synthesized at system build time for the target FPGA using parameters given by the system designer.

The Nios CPU is a pipelined Reduced Instruction Set Computer (RISC) processor that is targeted toward embedded applications. It is available as a 16-bit or a 32-bit CPU, and can be configured at system build time for different levels of performance, capability, and FPGA resource usage, although several configuration options are only available for the 32-bit Nios or on certain families of FPGAs. Optional separate instruction and data cache memories of configurable sizes can be included using on-chip resources. The 32-bit CPU also has an integer multiply function that can be 1) implemented as shifts and adds done entirely in software, 2) assisted by a single multiply step (MSTEP) hardware instruction, or 3) a hardware multiply (MUL)

[1] This refers to revision 3.0 of the Nios CPU, which is not to be confused with the Nios II CPU

instruction that requires at most three cycles to perform a 16 $\times$ 16-bit $\rightarrow$ 32-bit multiply. When an FPGA with Digital Signal Processing (DSP) blocks such as the Stratix family is used, the MUL instruction requires only one cycle to perform a multiply. The Nios is a little endian CPU, meaning that for multibyte datatypes, the least significant byte is transmitted or stored first.

The CPU has a register file of 128, 256, or 512 general-purpose registers, 32 of which are accessible at any point; 24 via a sliding window and 8 global registers. The window slides with 16-register granularity to create an overlap of 8 between adjacent window positions. When the window is moved forward in the file, the upper 8 registers (the "In" registers) become the lower 8 registers (the "Out" registers) of the new position. The middle 8 registers of any window are only accessible in that window position and are called the "Local" registers. Table 2.1 lists the register groups and their programming names.

Group	Names
In	`%i0`–`%i7` or `%r24`–`%r31`
Local	`%L0`–`%L7` or `%r16`–`%r23`
Out	`%o0`–`%o7` or `%r8`–`%r15`
Global	`%g0`–`%g7` or `%r0`–`%r7`

Table 2.1: Nios Register Groups

2.2.1.1 Custom Instructions

The Nios CPU can be customized with user logic in the form of custom instructions [10]. Up to five such instructions (named USR0–USR4) may be defined to implement functions necessary to the application, and are written in hardware description languages such as VHDL or Verilog or defined through Quartus block diagrams. The instructions are instantiated into the CPU's arithmetic logic unit (ALU) as illustrated in Figure 2.2.

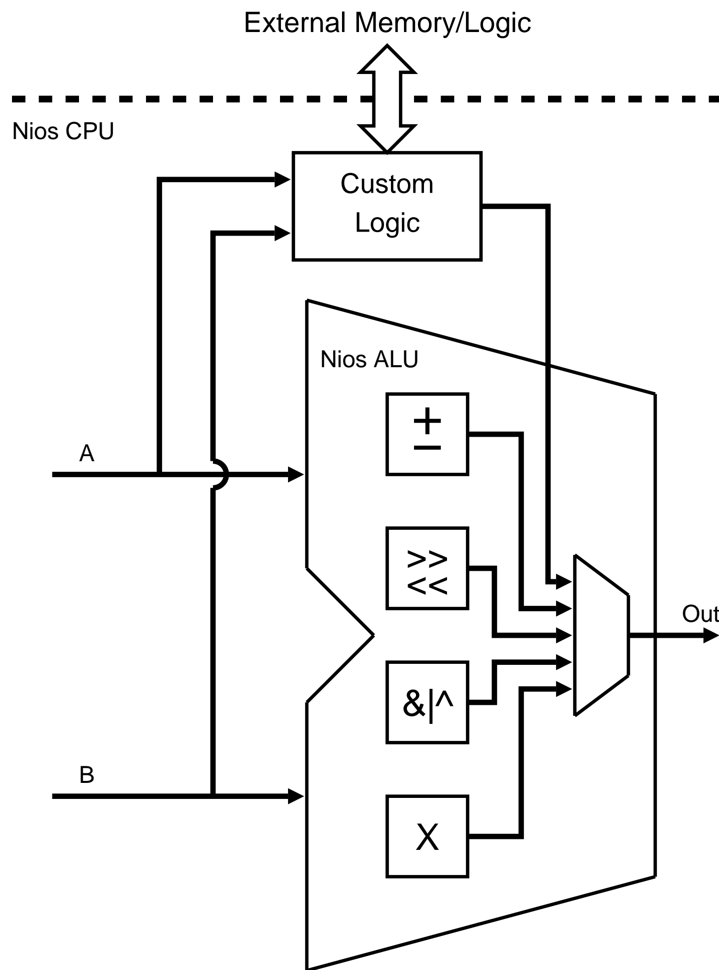


Figure 2.2: A Custom Instruction as part of the Nios CPU

Custom instructions use a standard set of interface signals consisting of up to two input data ports of the same width of the CPU, with one output port of the same width. Additional input signals are available for use in instructions that use sequential logic. The full list of available input signals is shown in Table 2.2. In addition, other ports in the instruction can be exported from the system for connection to logic external to the CPU or FPGA. Of note is the fact that only the first custom instruction (USR0) may use any general purpose register as the source for both input data ports. Instructions USR1–USR4 must use register `%r0` as the source for the `datab` port.

Name	Width (bits)	Description
`dataa`	CPU width	First general data port
`datab`	CPU width	Second general data port
`clk`	1	CPU clock
`clk_en`	1	CPU global clock enable
`reset`	1	CPU reset signal
`start`	1	Indicates that the data on the input ports are valid
`prefix`	11	Selects from different functions of the instruction

Table 2.2: Nios Custom Instruction Input Ports

The `prefix` port is an 11-bit input port included to select different functions of a single instruction. Its use requires an extra clock cycle to load the value into the prefix register before the instruction is issued. The value to load into the prefix register must be immediate rather than any general-purpose register.

The `clk`, `clk_en`, `reset` and `start` ports are required for sequential logic instructions that return their results after more than one cycle. Custom instructions that are entirely combinational return after one cycle and need not use those ports. The number of cycles before return for a sequential instruction may be configured to suit the instruction's design, but is fixed for all uses of the instruction; the different functions selected using the `prefix` port must all return after the same number of cycles.

2.2.2 Avalon Bus

The subsystem through which the peripherals in an Altera SOPC are connected is the Avalon Bus. It is designed to provide a simple bus protocol with low use of FPGA resources.

Each Avalon peripheral has one or more ports connected to the bus. The two types of ports, Master and Slave, are distinguished in that only Master ports may initiate data transfers. Examples of a device that would use a master port are a

CPU or a DMA controller. Slave peripherals can, however, be configured with an Interrupt Request (IRQ) line that is asserted when the peripheral requests a transfer to occur.

From the perspective of the programmer, devices on the Avalon Bus appear as 32-bit memory locations in a 32-bit address space. Each peripheral may have an arbitrary number of input address lines regardless of where it resides in the memory map; the Avalon bus handles address translation and only asserts the lines for the requested address minus the peripheral's base address. Multiple instantiations of the same peripheral are distinguished by names assigned by the designer. A peripheral's data port may be 8 or 16 bits wide rather than 32 should such widths be more appropriate (e.g., a controller for a memory with a 16-bit data port width). The peripheral designer has the option of having the Avalon bus convert a single 32-bit transfer into two 16-bit or four 8-bit transfers to/from the peripheral, for example converting a 32-bit read to two 16-bit reads from a controller for a 16-bit wide memory. This is called "Dynamic Bus Sizing". In general, this is only useful for memories, while most other peripherals use "Native Address Alignment", where for example one read from a 16-bit peripheral results in the upper 16 bits of the returned data being undefined.

Several basic peripherals (e.g., RS-232 UART, programmed I/O, timer) are included with the Altera SOPC Builder software. More advanced devices such as external bus interfaces, multi-channel DMA controllers and Universal Serial Bus (USB) controllers are available from Altera or from third-party vendors. Custom peripherals can be designed in HDL and added to the system using the Interface to User Logic device [11].

2.2.3 SOPC Builder

SOPC Builder is the software through which a designer configures the system. In it, the designer selects the components that comprise the system as well as configuration options for the individual components. For each device, a unique base Avalon address and name are assigned, as well as an IRQ line if applicable. Each type of peripheral also has its own set of configuration options accessible through SOPC builder (e.g., configuring a timer as a simple periodic interrupt or as a watchdog timer). This is where the designer imports the description files for custom instructions and adds them to the CPU.

After the system is assembled with all components configured and all assignments made, the system build process generates the hardware description and the software development kit (SDK) for the system based on the configuration options. The system can then be compiled for the target FPGA using Quartus II.

2.2.3.1 Software Development Kit

The software development kit created at system build time assists in the creation of software programs for the system. The memory map, the IRQ line & peripheral name assignments for each component, and custom instruction macros are defined in the generated header file so that the programmer may address the components and instructions by the names assigned by the system designer. Peripherals may also add functions to the SDK. An example of this is the included timer peripheral, which adds a function that installs an interrupt service routine upon the first call to it and on subsequent calls returns the number of milliseconds passed since the last call. For software compilation and debugging, SOPC Builder provides the Nios SDK shell, a command-line interface based on the Cygwin UNIX-like environment for Windows [12]. The SDK shell includes the GNU compiler collection (GCC) and binary utilities (binutils) as its standard toolchain for compiling and linking

programs along with several wrapper programs to simplify the command line usage by automating the build process into one command.

2.3 Base System

This section describes the hardware and SOPC system used in this project. The development hardware and software were provided by the Canadian Microelectronics Corporation.

The base hardware into which the system was instantiated is the Altera Nios Development Kit, Stratix Professional Edition. The development board [13] included in the kit is centered around a Stratix EP1S40F780 FPGA with 41,250 logic elements.

Other features of the board are:

- 1MiB[2] SRAM (2 × IDT71416)
- 16MiB SDRAM (Micron MT48LC4M32B2)
- 8MiB Flash memory (AMD AM29LV065D)
- MAX EPM7128AE configuration control logic
- Type 1 CompactFlash header
- RJ-45 Ethernet connector with physical layer and media access control (PHY/-MAC) (SMSC LAN91C111-NE)
- Two RS-232 DB9 serial ports
- Two JTAG connectors
- 50MHz socketed clock crystal
- External clock input
- Eight LEDs
- Dual-digit seven segment LED display
- Four push-button switches
- Two 41-pin expansion prototype connectors

[2] Using the IEC binary prefixes [14] to denote power-of-two multiples

- Mictor debugging connector

The development board's block diagram is shown in Figure 2.3 [13]. The Ethernet PHY/MAC, Flash memory and SRAM are connected to a common bus external to the FPGA. The CompactFlash connector and the first expansion prototype connector share several pins on the FPGA and therefore cannot be used simultaneously. The on-board flash memory may be used for general-purpose storage or to hold configuration data for the configuration controller to load into the FPGA upon power-up.

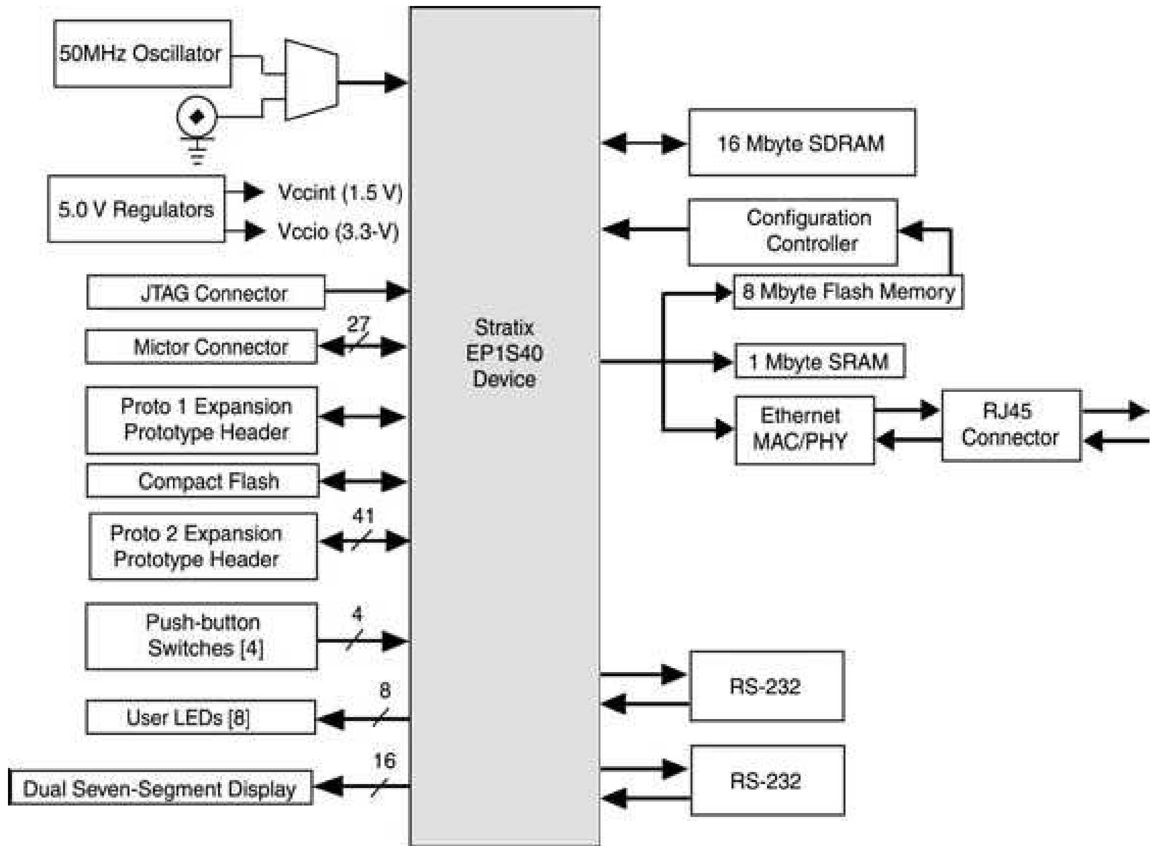


Figure 2.3: Nios Development Board Block Diagram (From [13])

The Nios processor was configured as a 32-bit CPU. The MUL instruction was enabled, and the presence of DSP blocks on the Stratix device mean that the instruction requires only one cycle to perform a 16-bit by 16-bit multiply. Instruction and data caches were enabled and had sizes of 2KiB and 4KiB respectively.

The system was configured with the following components:

- Nios CPU
- Boot ROM (on-chip memory)
- UART
- Timer
- Controllers for on-board memories:
  - SRAM
  - SDRAM
  - Flash
- Parallel I/O for buttons & seven segment displays
- CompactFlash host

All components except the CompactFlash host were supplied by Altera with the SOPC Builder software.

2.3.1 CompactFlash Host

The ideal method of transferring ECG files to the development board for testing is to use a general-purpose computer to write the files to a removable storage medium that can be accessed by the development board. CompactFlash [15] media is the obvious solution, as the development board has the required hardware connector. However, no CompactFlash host peripheral is supplied to enable access to the card through the Avalon bus. As a result, a host needed to be designed.

CompactFlash devices have a 50-pin electrical interface that is compatible with the PC Card [16] interface in PCMCIA I/O mode and the ATA-4 spec in True IDE mode. A third mode of operation called Memory mode is available and is the default. The bi-directional data port width is 16 bits, with the option of 8-bit access for 8-bit hosts via two card enable input pins. CompactFlash storage cards are accessed using

Address	Name	Notes
`000`	Data register	16-bit
`001`	Error register	8-bit, Read-only
`010`	Sector count register	8-bit
`011`	LBA bits 7–0	8-bit
`100`	LBA bits 15–8	8-bit
`101`	LBA bits 23–16	8-bit
`110`	LBA bits 27–24, drive/mode select	8-bit
`111`	Status register	8-bit, Read-only
`111`	Command register	8-bit, Write-only

Table 2.3: CompactFlash ATA registers

a subset of the ATA command set; several commands in the ATA command set do not apply to solid-state storage and are therefore unused.

The address on an ATA device selects from a set of registers to be read or written. There are registers for specifying the start sector of read/write operations (in cylinder/head/sector format or logical block addressing), for specifying the number of sectors requested, a register where the stored data is written or read, as well as command, status, and error registers.

ATA Commands are issued to the card by writing the command's parameters to the proper registers, then writing the desired command's code to the command register. Operations that involve transferring sector data to or from the card are followed by several sequential read or write accesses to the data register. For example, the procedure for reading a sector of data from the card is:

1. Write the number of sectors desired (1) to the Sector Count register
2. Write the first desired sector's address to the CHS/LBA registers
3. Write the command code for Read Sectors (0x20) to the Command register
4. Poll the Status register until bit 6 (Data Request or DRQ) of it is set, indicating that data needs to be transferred from the card

5. Perform 256 16-bit or 512 8-bit reads from the data register

CompactFlash cards have two memory spaces in memory mode: common memory and attribute memory. Common memory is the space through which the ATA registers are accessed and most operations are performed, while attribute memory contains card metadata and status & configuration registers. The PCMCIA Card Information Structure (CIS) data also resides in this space. Asserting the active-low $\overline{\text{REG}}$ pin on the card selects the common memory space, while holding it low selects attribute memory.

The primary function of the host is to translate Avalon bus read and write requests to CompactFlash accesses. It is designed to be as simple as possible, using only memory mode with 16-bit wide access. The host contains a delay counter that is triggered upon system reset or card insertion and holds the read and write enable signals inactive for a certain period in order to give the card time to complete its initialization cycle. Configuration registers internal to the host control the $\overline{\text{REG}}$ and card enable pins on the card.

Routines to translate requests to read and write sectors into the component ATA reads and writes as well as read card-specific info are implemented in a software driver that is linked into the main program.

The CompactFlash host is a 16-bit Avalon slave peripheral with five bits of address space. The peripheral's address space maps to addresses on the card as well as the host's internal configuration. Accesses to addresses 00h–0Fh on the host are passed to the card using the same address, while accesses to 10h–1Fh select the host's configuration registers and do not result in accesses to the card. The full set of Avalon signals used in the host is listed in Table 2.4.

Only four address lines of the 11 on the card are used due to simplification of the host, as the standard ATA register set only requires four bits of address space. It is assumed that only storage cards will be used with the host, and that there will

Signal Name	Width (bits)	Direction	Description
clk	1	in	Avalon clock
reset_n	1	in	Global reset
chipsel	1	in	Avalon chip select
addr	5	in	Address (Translated)
read_n	1	in	Read enable
write_n	1	in	Write enable
data_in	16	in	Input data
data_out	16	out	Output data

Table 2.4: Avalon signals used in the CompactFlash peripheral

be few if any accesses to attribute memory. The information about the size of the storage card necessary for a filesystem driver can be obtained in common memory through ATA commands. A listing of pertinent Identify Device results for the card used in this project is shown in Table 2.5. All omitted words are reserved, obsolete, or zero. Section 6.2.1.6 of the CompactFlash specification gives full details about the expected results of the Identify Device command.

The host contains a set of internal registers that control certain other signals between the host and the card. The two card enable lines are among these so that they can be both set high to put the card in standby mode. The CompactFlash $\overline{\text{REG}}$ signal is available so that the programmer may set it low to access the attribute memory space on the card. There are also soft reset and enable signals for the host itself. The development board connects the CompactFlash RESET pin to the board's global RESET signal and not a general I/O pin on the FPGA, so a host reset cannot be passed to the card as well. The host configuration registers are accessed through address 10h on the host.

Rather than generating the CompactFlash access waveforms in the host, that task is left to the Avalon bus, with the signals being passed to the card. The slowest timing mode (250ns) is used for compatibility with all existing CompactFlash cards, but faster modes may be hard-coded in SOPC Builder by altering the setup, wait,

Word #	Value	Information
0	0x848A	CF storage card signature
1	0x003D	Default number of cylinders = 61
3	0x0010	Default number of heads = 16
6	0x0020	Default number of sectors/track = 32
7–8	0x00007A00	Number of sectors/card = 31,232
10–19	“3071800096”	Serial number
22	0x0004	ECC bytes/sector on R/W Long commands
23–26	“Rev 2.06”	Firmware revision in ASCII
27–46	“CF 16MB”	Model number in ASCII (space-padded)
47	0x0001	Maximum sectors/block on R/W Multiple command
49	0x0200	Supports LBA mode addressing
51	0x0100	Supports PIO mode 1 access timing
53	0x0001	Words 54–58 are valid
54	0x003D	Current number of cylinders = 61
55	0x0010	Current number of heads = 16
56	0x0020	Current number of sectors/track = 32
57–58	0x00007A00	Current number of sectors = 31,232
59	0x0100	Multiple sector setting is valid
60–61	0x00007A00	Number of LBA-addressable sectors = 31,232

Table 2.5: Important Identify Device results for a 16MB CF storage card

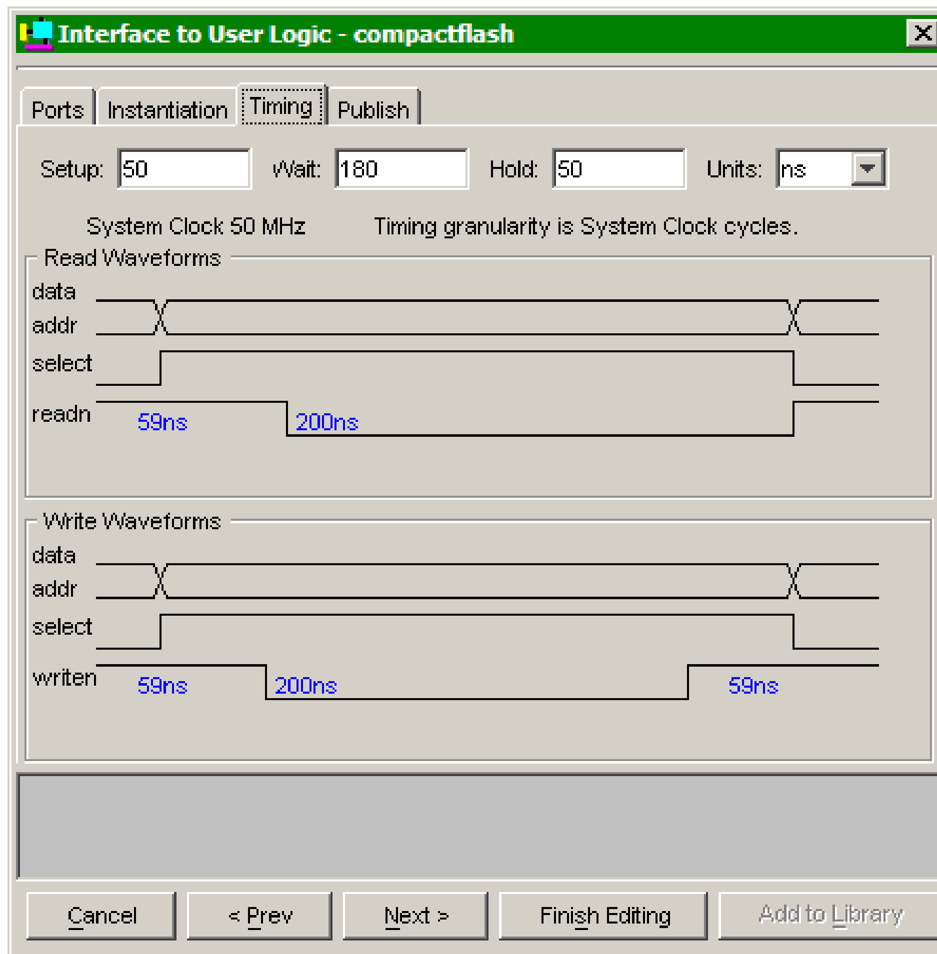


Figure 2.4: CompactFlash Waveform Timing Configuration

and hold times for the host peripheral if it can be assumed that the card supports that mode. The interface for this, showing the timing waveforms, is shown in Figure 2.4.

In order to support a range of transfer timings, the complexity of the host would have to increase significantly. The Avalon read/write waveforms, which are fixed at system build time, would no longer be used as the card access signals. The host would need to be designed to generate different signal timings based on its internal configuration, defaulting to a mode that is mandatory for all CompactFlash cards. The host's software driver would then determine the modes supported by the card by examining the data returned by an ATA Identify Device command and configure the

host accordingly. Since the goal of the host is merely to allow data to be transferred to the development board from a personal computer with little regard for speed, the host was not designed to be so complex.

Chapter 3
Encoding Algorithm Design

General data encoders are meant for use on the most common data types such as text and executable programs. Electrocardiograms, however, are waveform data, and waveforms have their own characteristics that reduce the effectiveness of general data compressors when used on this class of data. The properties of waveform data as well as the manner in which a waveform is read by the observer necessitate a different encoder design. This chapter begins by describing the traditional techniques employed in a waveform-specific encoder, then explains the specific encoding method used in this project for use on electrocardiogram signals.

3.1 Waveform Coding

Digital LPCM waveforms are created by sampling the amplitude of an analog waveform and deriving a quantized value for that amplitude. Dictionary encoders rely on repeated strings of identical symbols to reduce the data size. In a waveform, periods of exactly-matching passages are infrequent except in some cases of simple, computer-generated signals such as tones. Even in signals that are noticeably periodic to the human eye such as an ECG, it is difficult to find matching sequences of samples as the exact amplitudes in each period vary due to noise and differing sampling instants. The minor variations from one period to the next are enough to cause mismatch from a dictionary encoder's point of view. Also, dictionary encoders conventionally use bytes as the symbols from which phrases are comprised. In the

case where the number of bits used to store a waveform sample is not a multiple of eight (common in ECG signals), the encoder's efficiency can be further reduced as the assumption of bytes being the optimal symbol size is inaccurate. Therefore, correlation must be removed using a method more suited specifically to waveform data.

3.1.1 Partitioning

Common to a large number of waveform encoders is a partitioning operation that divides the signal into sequential blocks, to be encoded independently of each other, primarily to facilitate random access. The first benefit of this is that arbitrary parts of the signal are more quickly and easily accessible as less of the signal needs to be decoded in order to read a given part of the signal. This is in contrast to most text and data compressors, which require the packed file to be decoded from the beginning until the region of interest. This is not a concern for the files that general compressors are meant to be used on, but fast random access in waveforms is desired for operations such as seeking and editing.

Another benefit of partitioning is that it allows the encoder to adapt to the signal characteristics as they vary over time. Waveforms with a high rate of variability will not be packed efficiently using a fixed set of parameters across the entire signal. By encoding independent blocks into frames, each block can be encoded using a different set of parameters suited to that portion of the signal's properties. These parameters are stored in the header of the frame they are used on, and are the result of analysis performed on the block's data. Such headers can also contain synchronization information for streaming purposes.

The third benefit is resilience against corruption of the packed signal. Generally, an error in an encoded file will result in everything following the error being corrupted upon decoding. Errors can cause the misinterpretation of variable-length

code endpoints and cause pointers to refer to incorrect data, and so the error propagates forward. Partitioning the signal stops error propagation at the end of each frame, limiting the effect of errors to the frames they occur in and leaving other signal portions unaffected. This is important for streaming signals over a communications medium so that the receiver may recover from transmission errors.

There is a tradeoff when choosing the number of samples in a block. If it is too short, the frame headers contribute a significant amount of overhead and negate the gains of compression. If it is too long, the signal characteristics may vary too much within the block for any fixed set of encoding parameters to be effective at reducing the size of the data.

The term **block** is used here to refer to a group of inter-channel samples following partitioning. The term **frame** is used to refer to an encoded block.

3.1.2 Linear Predictive Coding

A typical linear predictive lossless waveform encoder has the topology shown in Figure 3.1. It includes operations for removing correlation between channels as well as within channels. The final part is a statistical entropy coder as described in the introduction. The buffer block stores an entire block of samples.

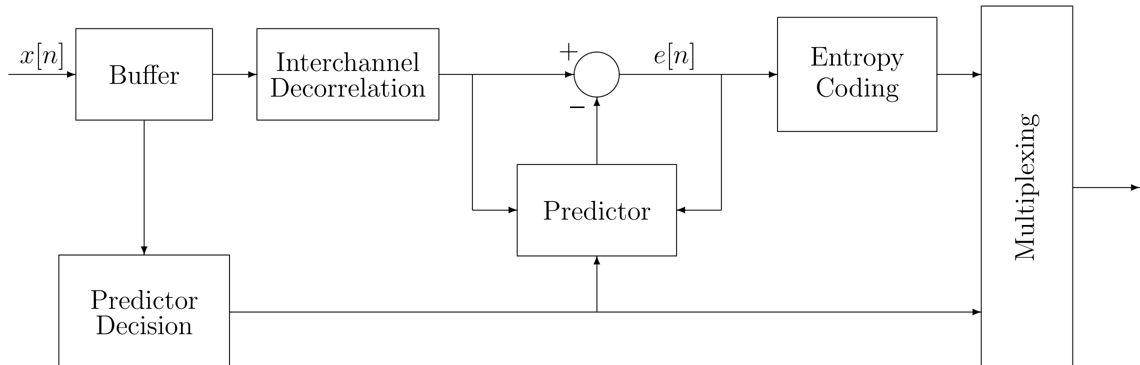


Figure 3.1: General Structure of a Linear Predictive Waveform Coder

3.1.2.1 Inter-Channel Decorrelation

Waveforms are often packed several at a time, each representing a different channel. In audio applications, each speaker in a system may be driven by a different channel waveform. In an ECG, the signals from each lead may be stored as separate channels. Correlation between these channels may be removed by deriving new channels which are each functions of the original signals.

For example, if two channels L and R very closely match in the time domain, the encoder may define the "middle" channel M as the average of the two signals $(L+R)/2$ and the "side" channel S as the difference between the two signals, divided by two $(L-R)/2$. This is called *mid/side stereo*. The middle channel would match both channels closely and have most of the information from both. The side channel would be small in amplitude and would contain much less information. The original signals could then be reconstructed by $L = M + S$ and $R = M - S$. Similarly, if two channels were very nearly the inverses of each other, using mid/side stereo would result in the side channel having the most information and the middle channel having very little. In both cases, the redundancy of having much of the same information present in both channels is removed. Another type of channel assignment is to keep one channel as it is, and define the side channel as the difference between the two originals, referred to in audio applications as *left/side* or *right/side stereo*, depending upon which channel is the reference.

3.1.2.2 Intra-Channel Decorrelation

If one were to view a waveform closely enough so that they could see the individual samples, chances are that they would notice that consecutive samples are rather close in amplitude. The idea that, in many cases, successive samples have small differences between them is exploited by storing the change in the signal by some encoding schemes in the field of communication. Differential Pulse Code Modulation

(DPCM) stores the difference between samples using a smaller number of bits than would be used for the original signal. This is a lossy compression scheme, as the number of bits used imposes restrictions on the magnitude of the difference stored, and should the original signal have a larger difference, noise is introduced. Delta Modulation is a subset of DPCM where one bit is used to denote if the next sample is greater or smaller than the previous one.

This idea is potentially useful for a lossless waveform encoder, but to apply generally to all waveforms, it must be viewed in a more broad sense. In the case of storing differences, correlation is removed by reasoning that successive samples will have similar magnitudes, and thus there is a high probability that the difference between them will be small. The idea of there being correlation from one sample to the next is retained, but in other signals it may be less subtle than one can tell simply by looking at the signal. The more general model is that the next sample can be predicted to be a linear combination of past samples and the error signal, which is called Linear Prediction.

3.1.2.3 Linear Prediction

In linear prediction, the short-term characteristics of the signal $x[n]$ are used to obtain an estimate $\hat{x}[n]$ of the next sample based on the values of a certain number of past samples $x[n-1]$, $x[n-2]$, and so on. If the prediction is accurate, the prediction error $e[n] = x[n] - \hat{x}[n]$ will be small and will require fewer bits to represent in the encoded output. The prediction error signal $e[n]$ is referred to as the residual. The generalized equation for the error computation is given in Equation 3.1.

$$e[n] = x[n] - Q\left\{\sum_{k=1}^{M} \hat{a}_k x[n-k] - \sum_{k=1}^{N} \hat{b}_k e[n-k]\right\} \tag{3.1}$$

Equation 3.1 can be viewed to represent a digital filter with both feedforward ($\hat{a}_k$) and feedback ($\hat{b}_k$) terms. The $Q\{\}$ indicates quantization to the same precision as the signal $x[n]$. The general equation has the structure seen in Figure 3.2. The

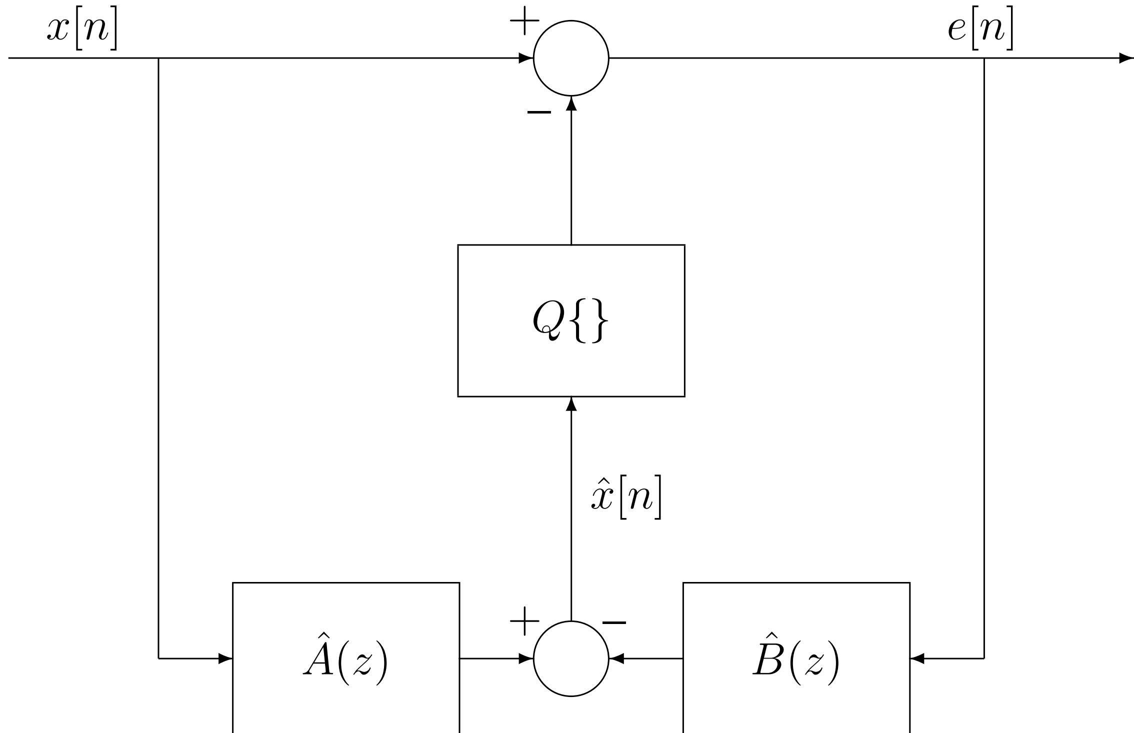


Figure 3.2: General Structure of a Linear Predictive Decorrelator

z-transform polynomials $\hat{A}(z)$ and $\hat{B}(z)$ represent the feedforward and feedback coefficients respectively. A predictor where the feedback coefficients $\hat{b}_k$ are zero has the equation

$$e[n] = x[n] - Q\left\{\sum_{k=1}^{M} \hat{a}_k x[n-k]\right\} \tag{3.2}$$

and is called a Finite Impulse Response (FIR) predictor. If the feedback coefficients are nonzero, the predictor is called an Infinite Impulse Response (IIR) predictor, regardless of the value of the feedforward coefficients.

An optimal predictor will output a residual that is completely uncorrelated, i.e., it will have a flat (white) frequency spectrum [17]. In this case, the predictor coefficients represent the redundant information in the signal. Note that an optimal predictor will not result in an all-zero residual unless the signal's entropy is small enough to be stored in the predictor coefficients, e.g., the signal being a tone at a frequency of

π with a predictor of $\hat{x}[n] = -x[n-1]$. Instead, the prediction errors represent the information in the signal.

IIR predictors are capable of matching a much greater range of spectral shapes than FIR predictors and can therefore be used to make more optimal predictors with the same number or fewer coefficients. However, IIR filter optimization is much more complicated than FIR optimization, and only moderate improvements in encoding efficiency have been demonstrated thus far [18, 19].

The first samples of the signal to be encoded may be of any value, and if the system initializes the filter with zeros and feeds the signal in from its very beginning, the output may have a transient passage at its beginning that will be inefficient to encode. Instead, what is typically done in a practical system is to initialize the feedforward chain of length M with the samples $0...(M-1)$ of the signal and pass the signal through starting with sample M. This avoids the transient in the output, but necessitates the storage of the first M samples in the stream. These "warm-up" samples can be stored in the encoded stream in their original format.

The original signal can then be reconstructed from the residual, the predictor coefficients, and the warm-up samples using the inverse system of the decorrelator, which is described in Equation 3.3 and shown in Figure 3.3.

$$x[n] = e[n] + Q\left\{\sum_{k=1}^{M} \hat{a}_k x[n-k] - \sum_{k=1}^{N} \hat{b}_k e[n-k]\right\} \tag{3.3}$$

The reconstruction filter will be an IIR filter unless the decorrelator had no feedforward ($\hat{A}(z) = 0$). It is possible that a reconstruction filter will be unstable, but this does not matter as its input is the output of its inverse, and so the reconstructed signal will always be bounded.

It should be noted that floating-point operations are unsuitable for use in the decorrelator. Rounding errors and architecture-specific idiosyncrasies with floating-point operations can result in inexact reconstruction of the signal between platforms. When the original signal is in floating-point format, the samples must be represented

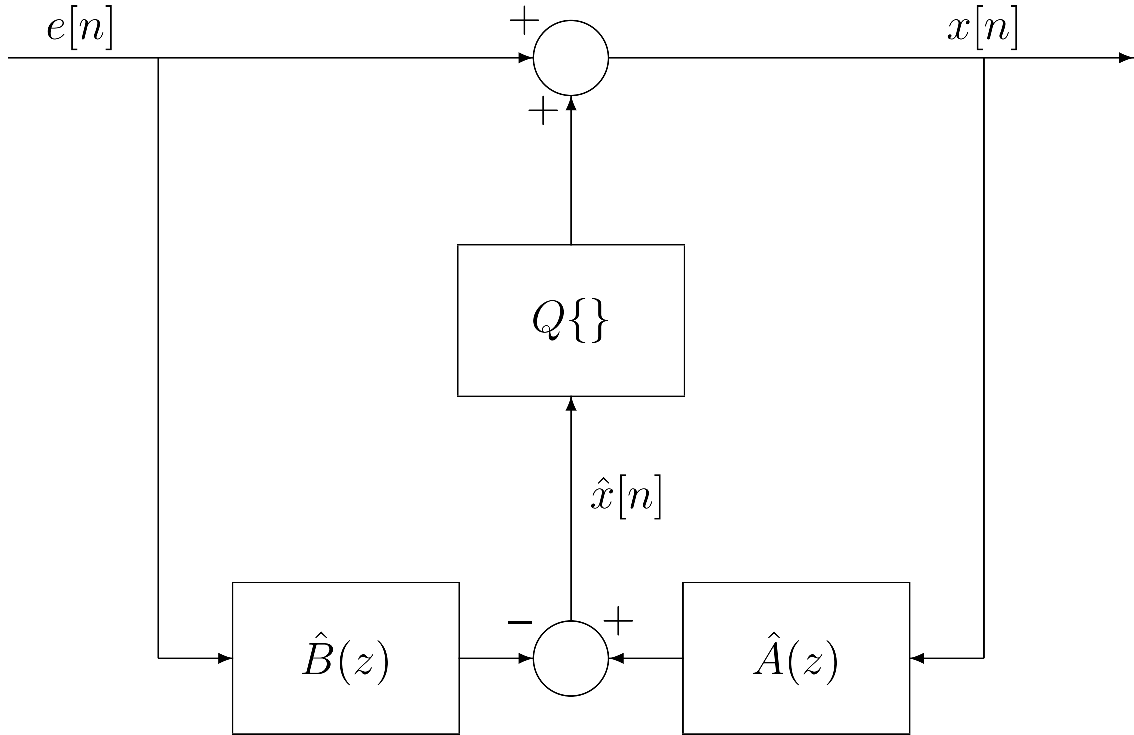


Figure 3.3: General Structure of a Linear Predictive Reconstructor

as integers in the encoder so that no loss is introduced. This restriction does not apply to any analysis process that decides upon the predictor coefficients; as long as the resulting coefficients are quantized to integers or fractions, floating-point math is allowable in that portion of the encoder.

There are several methods of deciding on the predictor to be used. The simplest of them is for the encoder to have a library of fixed predictors. Fixed predictor libraries may be comprised of filters constructed to be effective on different types of waveforms, e.g., different styles of music, or the predictors may have similar characteristics, but have varying degrees of filter response, for use on signals that are assumed to have certain characteristics. This may save space in the stream as an index to the predictor used can be stored in the stream rather than all of the coefficients. Some analysis is performed on the signal to determine the best predictors to use from the library and that predictor is used to encode the signal. The decision process may involve, for

example, analysis of the frequency spectrum or comparing the residuals after using each of the predictors on a portion the entirety of the block.

Another method is to calculate an optimal set of predictor coefficients from the signal data based on criteria such as minimum mean square error. Typically this will be done by the Covariance method [20] or calculating the autocorrelation of the signal and using the Levinson recursion [21] algorithm or one of its faster derivatives. The Levinson algorithm has been proven to give the optimal FIR predictor coefficients, and also allows the encoder to iteratively calculate the coefficients for each order of filter. It also allows for calculation of the residual variance, from which the encoded residual size can be estimated. As the order increases, the residual size will decrease as the predictor better suits the signal but the storage required for the predictor coefficients will increase. The iterations end where the total size does not decrease and that order is selected. Once the predictor is calculated, the decorrelation is performed exactly as in the case of a fixed predictor library.

The decoder complexity when using a fixed predictor is independent of the process of choosing the predictor. Only the order of the predictor (the number of coefficients in the filter) affects the number of operations required to reconstruct a signal. Increasing the thoroughness of the search for the optimal predictor may increase the encode complexity, but has little effect on the decode process, and so there is said to be *asymmetry* between the encoder and decoder.

An adaptive predictor monitors the residual and alters one or more factors based on how predictable the signal is judged to be. The factor changes as the signal properties change, and with a good adaptation decision process, the predictor becomes well-suited to the entire block.

Consider this example of a simple adaptive FIR predictor. The factor a represents the predictability of the signal. The most recent samples of the signal are

$$x[53] = 4$$

$$
\begin{aligned}
x[54] &= 8 \\
x[55] &= 26
\end{aligned}
$$

The predictor is $\hat{x}[n] = 2x[n-1] - x[n-2]$. By this estimation,

$$\hat{x}[56] = (2 \times 26) - 8 = 44.$$

The residual sample $e[n]$ is defined as $x[n] - (a/a_{max})\hat{x}[n]$. If $a = 128$, $a_{max} = 256$, and $x[56] = 30$, then

$$e[56] = 30 - \left(\frac{128}{256}\right) 44 = 30 - 22 = 8.$$

In this case, the factor a would be increased, as a larger value would have been more appropriate.

Unlike with a fixed predictor, this feedback provides temporal adaptivity within blocks, and as a result an adaptive predictor is capable of more efficient encoding. The tradeoff is that it requires an increase in the complexity of the decoder. The code path that adapts the factors must be present in the reconstruction filter so that the changes occur in the decoder as they do in the encoder. As a result, codecs based on adaptive prediction are more *symmetric* in that as the encoder adaptation complexity increases, so does the decoder complexity.

3.1.2.4 Entropy Coding

Entropy coding is applied to the residual signal in order to represent it in as few bits as necessary. Some algorithms use Huffman coding, while others use a static code such as one of those listed in Section 1.2.3.2. When used by itself on a waveform, entropy coding is at best moderately effective. It is much more efficient when it follows decorrelation, as a decorrelated signal has a probability density function that is more compact than that of the original signal. This is similar to how lossy audio encoders work, where a normal or modified Discrete Cosine Transform of the signal

results in a compacted energy spectrum, which is then psychoacoustically shaped and entropy coded.

The output of an effective predictor will be a stream of small, uncorrelated numbers centred at zero. This includes negative values. While several entropy coding schemes are incapable of representing negative values, that can be worked around by mapping the infinite set of all numbers onto the infinite set of positive numbers, normally by mapping positive values to even numbers and negative values to odd numbers. This is functionally identical to using a separate sign bit and accounting for the use of the "negative zero" symbol.

One algorithm that is particularly common in lossless encoders is Rice coding [22]. Rice coding is a special case of Golomb coding where the tunable parameter p is a power of two, i.e., $p = 2^k$. In such a case, the Rice parameter k is equal to the number of bits used to store the remainder. This case has properties that make it the easiest for two's complement machines to calculate. Table 3.1 shows a sample of the relationship between the Rice parameter k, some Rice codes and the numbers they represent. Note that a Rice codes with a parameter of 0 are identical to the Unary codes for the same numbers.

In the general case of Golomb coding, the quotient and remainder calculation is performed by an integer divide and a modulo operation. Rice coding allows these calculations to be performed more easily and quickly in two's complement machines.

Number	**Parameter**			
	0	1	2	3
0	1	10	100	1000
1	01	11	101	1001
2	001	010	110	1010
3	0001	011	111	1011
4	00001	0010	0100	1100

Table 3.1: Numbers 0–4, Rice coded using parameters 0–3

In such machines, the number to be encoded can be split on bit boundaries to obtain the quotient and remainder. For example, consider the encoding of the number 26 with the Rice parameter $k = 3$, which is the same as Golomb parameter $p = 8$. 26 divided by eight is three with a remainder of two. 26 in binary is 11010, which when split at the boundary three bits from the right, the results are 11 (three) and 010 (two). Three in unary is 0001, and so the code for 26 with $k = 3$ is 0001010. In a two's complement machine, division by 2^k can be achieved by an arithmetic shift right by k bits. Calculation of the remainder is simply a bitwise AND of the number with $(1 << k) - 1$.

The other aspect of Rice coding that distinguishes it from Golomb coding is the mapping performed prior to the encoding. This is done so that it may properly represent both negative and positive integers. The mapping is as follows:

$$M(e[n]) = \begin{cases} 2e[n], & e[n] \geq 0 \\ 2|e[n]| - 1, & e[n] < 0 \end{cases}$$

3.1.2.5 Multiplexing

The multiplexing operation combines the residual and the information necessary to decode the waveform into the completed frame. The encoded subblocks, or subframes, are stored sequentially in the frame. The finished frames out of the multiplexer are cascaded to form the bitstream. The additional information in each frame includes the predictor coefficients, residual encoding parameters, and block metadata. The encoder may also include cyclic redundancy check (CRC) checksums for the purposes of verifying the data or avoiding false stream synchronization.

3.1.3 Transform/Hybrid Coding

Another method of lossless waveform encoding is to first obtain a time-domain approximation of the signal using lossy compression. The lossy approximation as well

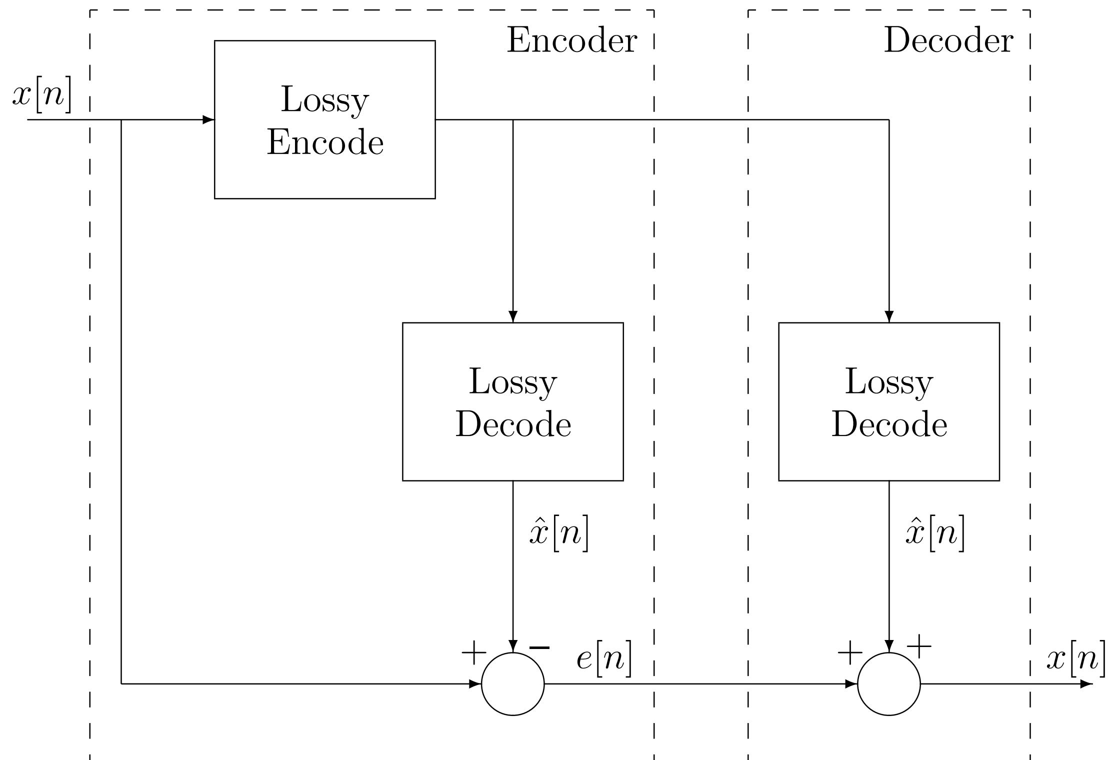


Figure 3.4: Structure of a Lossless Transform Coder

as the difference between it and the original signal are then entropy coded and transmitted. The block diagram of such an encoder is shown in Figure 3.4.

One audio codec, called Lossless Transform Audio Coding (LTAC), was designed by Purat et al [23] at the Technical University of Berlin in the late 1990s. It is the only major known attempt at such a codec, and has since been judged as slower and less efficient than codecs based on linear prediction. Development on LTAC has been discontinued by its developers in favour of prediction-based coding.

3.1.4 ECG Characteristics

The facts that an electrocardiogram signal has expected beat components and that a normal ECG is quite periodic introduce areas that an ECG encoder may exploit in order to obtain better compression.

Piecewise linear approximation is extensively used in lossy encoders because most

components, in particular the QRS interval, can be approximated using a relatively small number of lines. The periodicity is sometimes exploited using techniques similar to the Ziv-Lempel (LZ) algorithms [24] involving references to past portions of the signal in a sliding window or with a dictionary of beat components. These techniques may be accompanied by a gain factor to allow the matching of various signal strengths.

3.1.4.1 Lossless Concerns

Lossy algorithms, so long as they are designed to decode to the same result on all architectures, can be made lossless by simply adding the residual or error signal, which is the difference between the lossy reconstruction and the original signal. The error signal then contains the information lost in the lossy step and will comprise the majority of the data stored.

Digital ECG signals are acquired at varying sample rates and bit depths, which has certain effects on the compressibility of the signal. It has been shown [25] that the compression ratio is positively correlated to the sampling rate (i.e., the ratio increases as the sampling rate increases) and is negatively correlated to the bit depth. This is because increasing the sampling rate gives a smoother, move predictable signal, whereas reducing the bit depth eliminates the fine detail that hinders compression.

3.1.5 Platform Concerns

The algorithm to be implemented needs to be designed around the characteristics of the target platform in order to be efficient. The target is to create the encoder as a custom CPU instruction in an FPGA-based SOPC.

Custom instructions are instantiated as a part of the CPU's arithmetic logic unit and receive information via the same interface as the other function of the ALU. Custom instructions cannot also be Avalon peripherals and thus cannot access

system memory on their own, Custom instructions may contain any custom logic including memory, but the expense of using memory in terms of FPGA resources discourages this. Plus, all transfer of data between the instruction and memory must be handled by the CPU in small (64-bit input, 32-bit output) pieces, which introduces a bottleneck into the system that discourages the use of techniques that involve frequent memory access and high-order analysis of the signal.

3.2 Encoding Algorithm

This section describes a lossless encoder suitable for electrocardiogram signals using the concepts introduced in Section 3.1.

3.2.1 Objectives and Requirements

There were several factors taken into consideration in the design of the encoding algorithm.

The platform into which the encoder will be built is an embedded System on a Programmable Chip. The system will be used as a Holter Monitor, but there is no assumption that the monitor will only continuously encode and store. For example, instead of recording for the entire time it is operating, it may be designed to record starting when the wearer presses a button on the device, be it because he is about to begin a strenuous exercise, or because he is experiencing significant discomfort. The monitor may also continuously read but not store the heart's activity, watch for deviations from the regular heart activity, and decide to store the waveform should it judge that the wearer is experiencing a sudden condition.

To allow such duties to be performed, the encoder should not obstruct other tasks by requiring a significant fraction of the CPU's cycles. The use of a SOPC allows the inclusion of custom logic designed to perform specialized functions, so the task

of encoding the captured ECG signals can be offloaded from the main CPU.

The decoder is most likely to reside on a computer in a hospital or medical lab, but there is the possibility that other devices may access the stored signals. The specialist may use a PDA (Portable Digital Assistant) to read the ECG simply because it is more convenient than using a non-mobile computer. Ambulances could be equipped with portable computers that, when connected to the Holter Monitor, could download the most recent ECG capture and annotations and inform the paramedics of the most likely cause of an incident.

Due to the possibility of platforms with low computing capability being used to read the signals, it is a good idea to keep the decoder complexity low. The encoder may have an arbitrary level of complexity, but the algorithm must be asymmetric to keep the decoder simple.

The encoder is designed to closely follow the structure of a traditional Linear Predictive waveform encoder as described in Section 3.1.2. The main reasons for doing so are that a prediction-based algorithm results in a simpler and faster decoding algorithm than a transform-based method, as well as familiarity — the implementation may be designed so that the encoded bitstream is compatible with a format that is recognizable among software developers and already has available software libraries that can decode the waveform.

3.2.2 Interchannel Decorrelation

Interchannel decorrelation was considered, but is unlikely to have enough benefits to justify the extra effort. A Holter Monitor acquires measurements from different locations on the patient's chest for the purpose of obtaining as much information as possible. The different leads reflecting different areas of the heart are not likely to contain redundant information that interchannel decorrelation would remove. Time delays between signals can also greatly reduce its effectiveness. The addition of mid-

side and left/right-side channel mappings only improves compression of two-channel ECG signals by a few percent, even when the channels appear very similar to the eye.

Instead, interchannel decorrelation is most useful in the same way as having the option of run-length encoding a block — to improve compression for special cases of a signal, e.g., when a monaural signal has been split across multiple channels.

3.2.3 Intrachannel Decorrelation

The characteristics of the signals to be encoded strongly influence the decision on how to choose a predictor. A fixed library is simpler, but may not be efficient for encoding signals that vary greatly from one to another. It is possible to match a wider range of spectral shapes by calculating the predictor using the Levinson method, with the tradeoff of requiring more calculations for analysis. Non-adaptive predictors are used because of the desired asymmetry between the encoder and decoder.

ECG signals are periodic beats with expected components (see Figure 1.1) and approximate period time. There is variation in the magnitude, polarity, and elevation of the beat components between people, measurement locations, and arrhythmia as well as some variation in the period time, but that is the major extent to which different ECGs differ. As a result, it is clear that ECGs are a very narrow class of signals, and so a fixed library of predictors appears to be a good candidate.

When observing an ECG signal closely as in Figure 3.5 one will notice that there is very little rapid change from one sample to the next with the exception of the QRS interval. This hints that most of the signal energy is at low frequencies.

Figure 3.6 shows that indeed the majority of the energy is in the low frequencies. An optimal predictor 'whitens' the signal by having a frequency response that is the inverse of the signal's spectrum. Therefore to decorrelate a signal with a spectrum similar to those shown, a highpass filter is called for. In fact, the spectra have

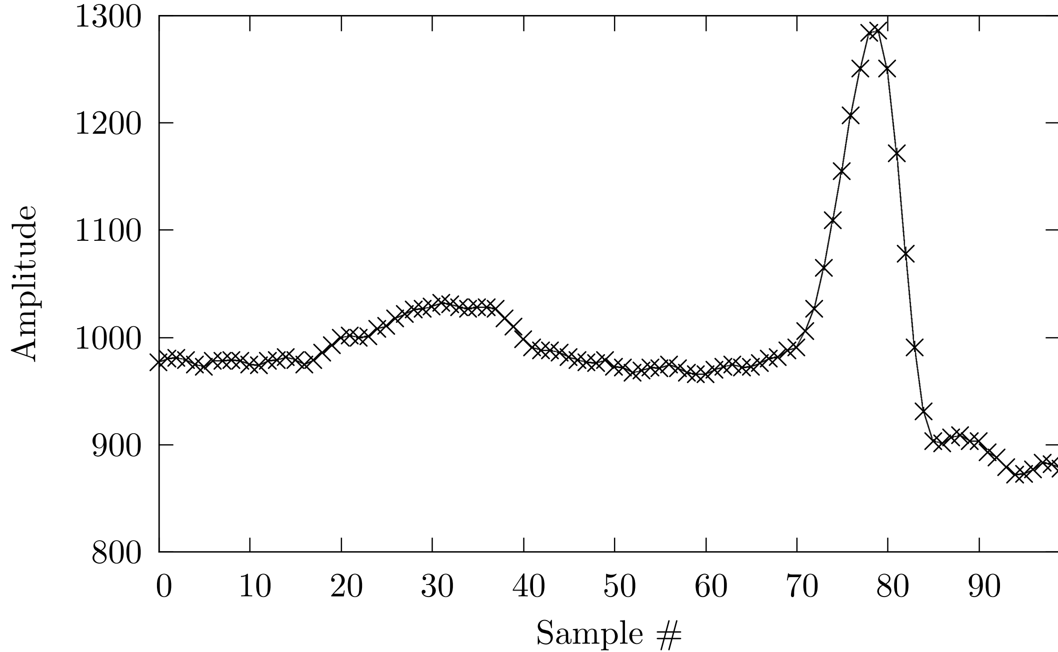


Figure 3.5: Close-up of an ECG waveform, showing individual samples

a general trend of decreasing as the frequency increases. The inverse of such a spectrum will increase with frequency, and a filter that implements it will be easier to create than one with a more detailed response. The simplest filter with an increasing response is a differentiator.

The slope of the frequency spectra varies from signal to signal. The different slopes will require differing degrees of response to whiten, which can be achieved via multiple passes through the predictor.

The predictor library used in this encoder consists of differentiators of first through third order:

$$e_1[n] = x[n] - x[n-1] \tag{3.4}$$

$$e_2[n] = x[n] - 2x[n-1] + x[n-2] \tag{3.5}$$

$$e_3[n] = x[n] - 3x[n-1] + 3x[n-2] - x[n-3] \tag{3.6}$$

The frequency responses of these predictors can be seen in Figure 3.7.

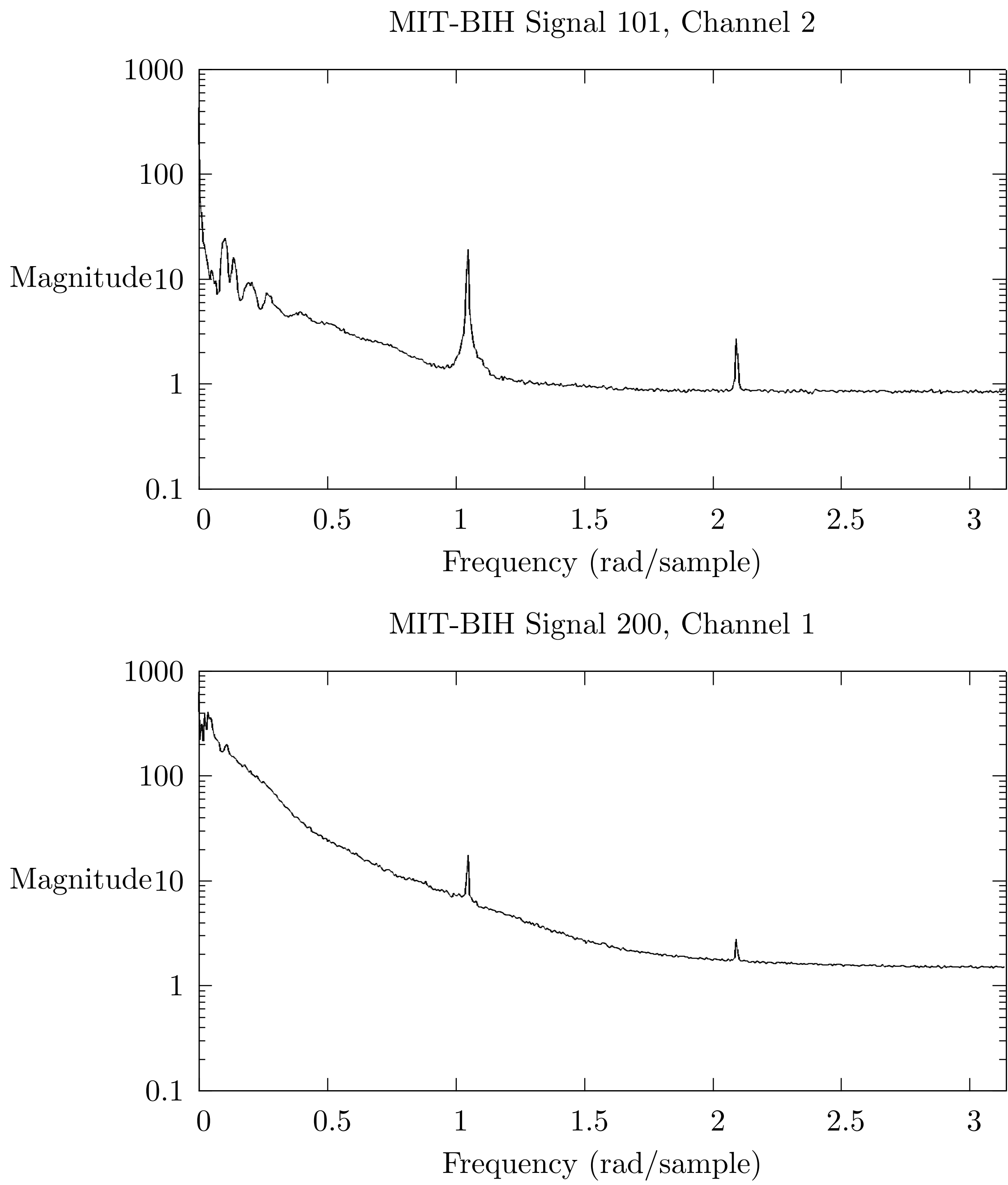


Figure 3.6: Example Frequency Spectra of ECG signals

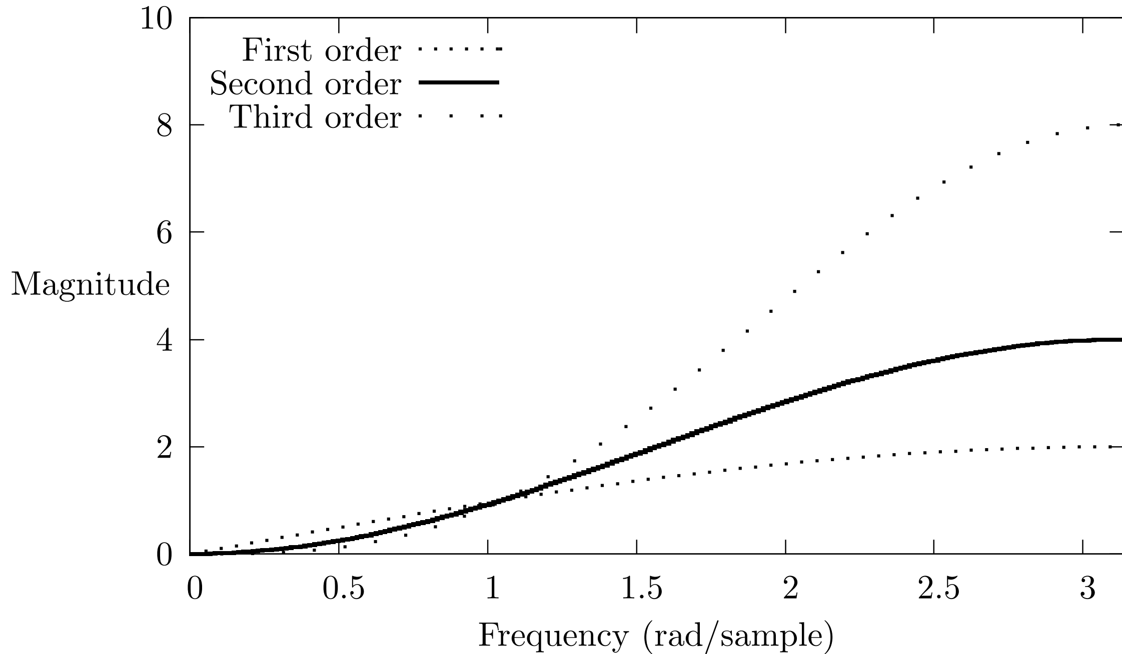


Figure 3.7: Frequency Response of Predictors

The frequency response graphs show that each of them has a unity gain at a frequency of $\pi/3$. Above that frequency, the predictors amplify rather than attenuate. The response of these predictors are a suitable match for typical ECG signals as the majority of an ECG signal's energy is in the lower frequency range that will be attenuated.

These predictors offer benefits that simplify the analysis and reconstruction of the signal. The choice of which order to use on a block is made simpler by the fact that each order's error signal can be computed from the previous order's error recursively:

$$e_1[n] = x[n] - x[n-1] \tag{3.7}$$

$$e_2[n] = e_1[n] - e_1[n-1] \tag{3.8}$$

$$e_3[n] = e_2[n] - e_2[n-1] \tag{3.9}$$

Calculation of each predictor's error for analysis purposes can therefore be paral-

lelized to obtain each residual simultaneously.

Since the optimal predictor results in the lowest-amplitude error signal, this encoder estimates the best predictor order by summing the absolute value of each error sample for each order and choosing the order with the lowest total. A signal with a constant value will result in the error totals being zero, which will be detected by the encoder so that it may be run-length coded instead of predicted. It can also be flagged as a device malfunction, as such a signal is unlikely to occur from an electrode attached to a person.

Calculating the original signal from the residual can be performed without the need for multiplication by implementing the reconstruction filters using only shift and add operations. The second and third order reconstructors, which use non-unity factors, use the following equations:

$$x_2[n] = e[n] + (x[n-1] << 1) - x[n-2] \tag{3.10}$$

$$\begin{aligned} x_3[n] = {} & e[n] + (x[n-1] << 1) + x[n-1] - (x[n-2] << 1) \\ & - x[n-2] + x[n-3] \end{aligned} \tag{3.11}$$

where $<<$ is the arithmetic left shift operator.

Table 3.2 shows the aggregate number of times each predictor was chosen when encoding ECG signals from the MIT-BIH [26, 27], European ST-T [28], and the MIT Compression Test [29] databases, using a block size of 1024 samples. Also included in the table are totals for zero-order ($\hat{x}_0[n] = 0$) and fourth order ($\hat{x}_4[n] = 4x[n-1] - 6x[n-2] + 4x[n-3] - x[n-4]$) predictors, illustrating that their use in this encoder is unnecessary.

The benefit of having decorrelation before entropy coding is shown in Figure 3.8, which depicts two signals' probability density functions and those of their residuals as calculated using the method outlined above. The residual pdfs are much taller and narrower and are therefore more susceptible to entropy coding due to a shift

Library	Predictor Order					Total
	0	1	2	3	4	Total
MIT-BIH	0	4920	26753	77	0	31750
European ST-T	0	18445	33970	325	0	52740
MIT Compression Test	2	226	1419	33	0	1680
Total	2	23591	62142	435	0	86170

Table 3.2: Block count for each predictor order

in probability toward the lowest-magnitude symbols, which require fewer bits to represent.

3.2.4 Entropy Coding

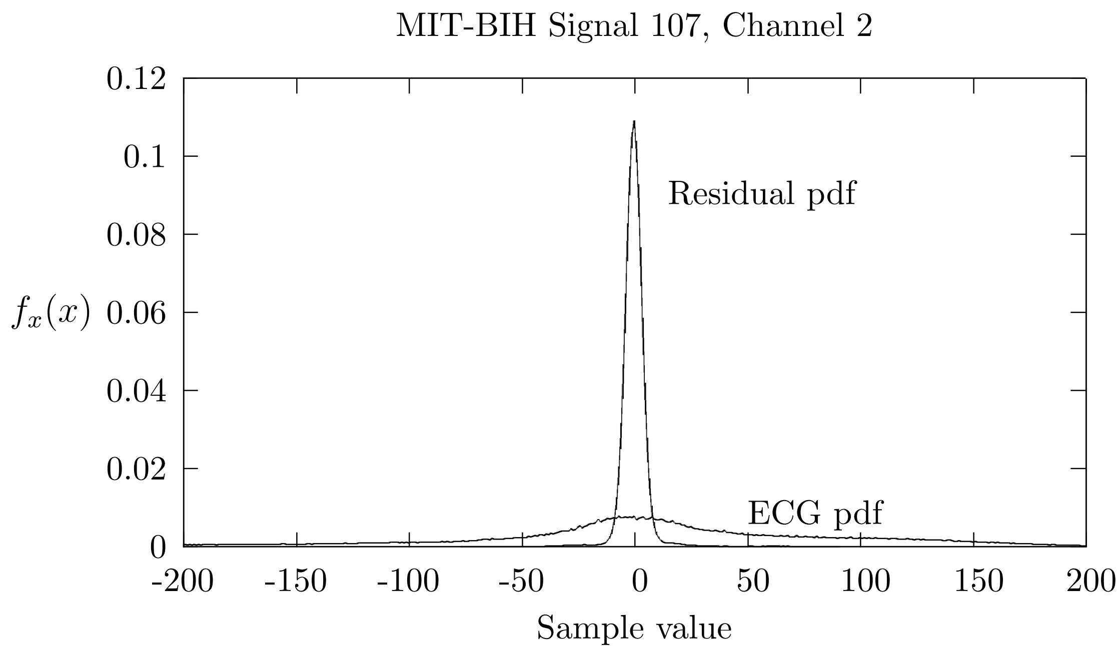
The encoder uses Rice Coding to represent the residual signal, as it provides the best balance between encoding efficiency and ease of decoding.

Rice codes represent the optimal code mappings for a residual with a Laplacian probability density function:

$$f(x) = \frac{\lambda}{2} e^{-\lambda |x|} \tag{3.12}$$

The Laplacian distribution is considered a good approximation for the distribution of a residual following linear predictive decorrelation [23, 30].

The entire residual of a block is encoded with a single Rice parameter. In estimating the optimal parameter with which to encode the residual, the total absolute error calculated when choosing the predictor order is of great help. Figure 3.9 shows the absolute residual totals of 1024-sample ECG signal blocks after decorrelation and the calculated optimal rice parameters for each of them. The vertical lines at 1024×2^n are the approximate points where the optimal Rice parameter increases. This leads to a simple and accurate method of estimating the optimal Rice parameter based on the block size N and total absolute error E, which is to use the lowest



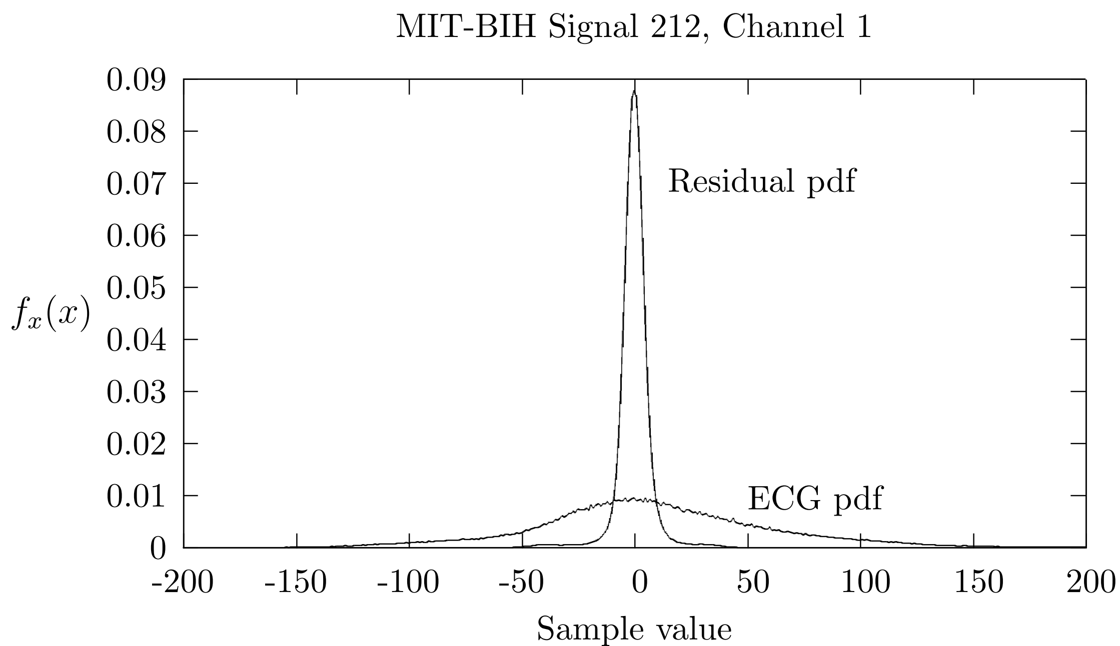


Figure 3.8: Example Probability Density Functions of ECG Signals and their Residuals

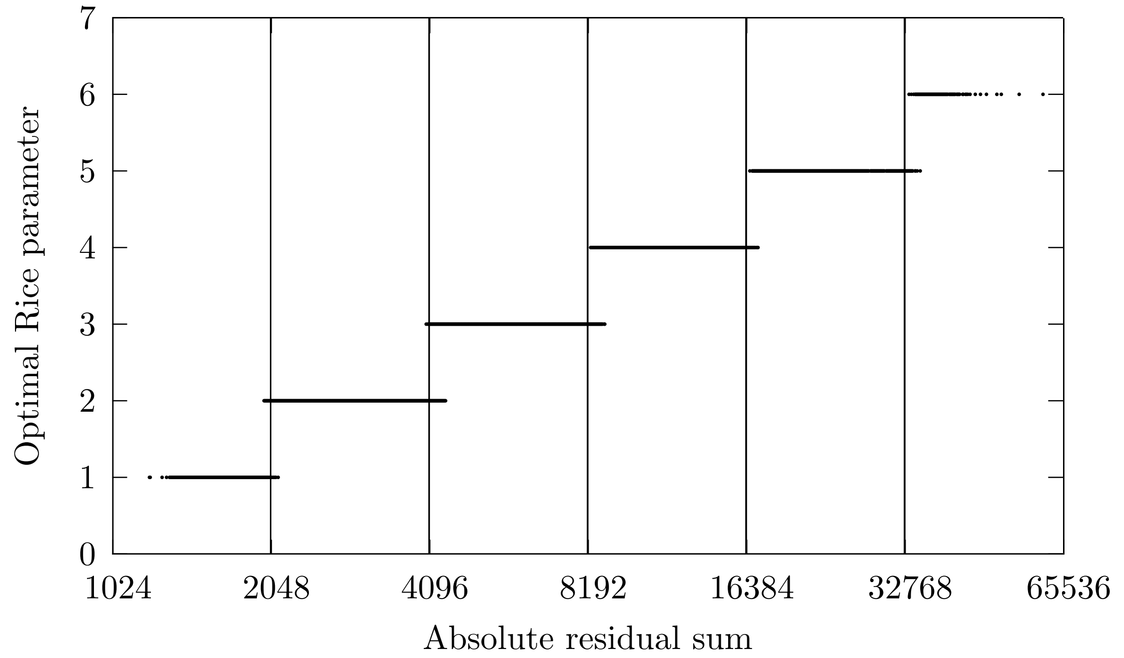


Figure 3.9: Optimal Rice Parameters versus Residual Totals

k such that

$$(N \times 2^k) > E.$$

Chapter 4
Encoder Implementation

This chapter discusses the custom instruction implementation of the encoding algorithm described in Chapter 3.

4.1 Interface

The instruction's analysis and encoding operations are addressed separately. The instruction therefore uses the prefix port to select from the two operations, which are further divided into component functions, as listed in Table 4.1. These functions vary in the number of cycles needed to complete their respective tasks, but the instruction must return after the same number of cycles for all prefix functions. To minimize wasted cycles, the instruction is thus designed to return after one cycle while all operations that require longer than that will continue after the instruction call until completed. By having the calculations performed in the background, the CPU may address other tasks while the instruction is processing rather than idly waiting for it to complete.

The instruction's signal interface uses the full set of signals listed in Table 2.2. When supplying the instruction with ECG data, the two data ports are used to transmit up to four samples per call, with each sample being in two's complement format with 16 bits of precision.

The Query Status function returns the operational status of the instruction, indicating that it is idle, busy, or has an encoded result available. The instruction

Prefix[3:0]	Function	Inputs	Return Value
0000	Begin New Block	Parameters, warmup	busy/success
0001	Query Status	unused	status code
0010	Get Encoded Result	unused	Result register
0011	Get # of valid bits	unused	Valid bits
01XX	Encode Samples XX = (sample count − 1)	Samples	busy/success
10XX	Analyze Samples XX = (sample count − 1)	Samples	busy/success
1100	Process Analysis Result	unused	busy/success
1101	Get Analysis Result	unused	Analysis result
1111	Reset Analysis	unused	busy/success

Table 4.1: List of custom instruction prefix values & functions

is considered busy if either the analysis or the encoder has yet to finish processing samples that it has been given.

The following are more detailed descriptions of the other prefix functions:

4.1.1 Analysis

Analyze Samples Supplies the analysis block with up to four samples of ECG data.

Process Analysis Result After all samples of a block have been supplied to the instruction, this function tells the instruction to perform the shifts on the lowest error to determine the optimal Rice parameter (see Section 3.2.4).

Get Analysis Result Returns the optimal predictor order and Rice parameter.

Reset Analysis Clears the state of the analysis block so that a new block may be started.

4.1.2 Coder

Begin New Block Clears the state of the encoder block and accepts the encoding parameters and warm-up samples for a new block as input.

Get Encoded Result Reads the contents of the result register.

Get # of valid bits Returns the number of bits shifted into the result register.

Code Samples Supplies the encoder with up to four samples of ECG data.

4.2 Coder Implementation Design

This section discusses the internal design of the instruction.

4.2.1 Analysis

The analysis block is shown in Figure 4.1. The initial buffer stage is the immediate destination of the samples presented in a Analyze Samples call. This is followed by the chain of subtractors that computes the error signal for each order. Since the first residual sample for the Nth order cannot be properly calculated until the $(N+1)^{\text{th}}$ sample has been received, care was taken to ensure that earlier outputs from the subtractors are not considered valid. Such transients may have a large enough magnitude to interfere with the results of the analysis, particularly with signals that are not centred at zero.

After a set of samples has been passed to the instruction, it is considered busy until all of the samples have been processed. When the instruction becomes ready again, the next samples may be given immediately. This process is repeated until the host software would like to obtain the results for the block of samples given.

Each order has an accumulator into which the absolute value of each valid residual sample is added. These totals are the inputs to the comparator, which selects the

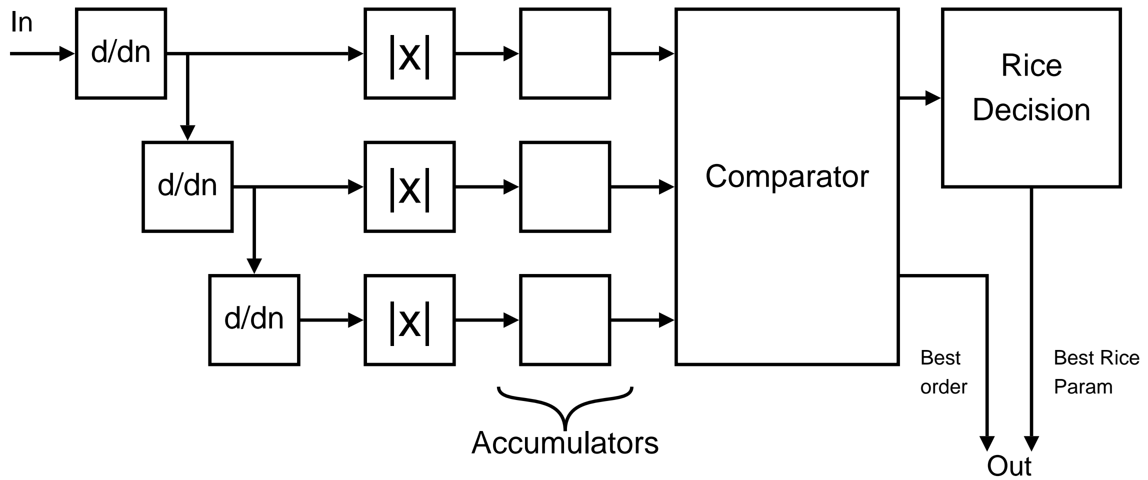


Figure 4.1: Block diagram of the analysis function

```
function wait_until_ready()
{
   while true:
      if Query_Status() == 0:
         return
}

function Analyze_block(samples)
{
   Reset_Analysis()
   while samples_remaining is greater than 3:
      wait_until_ready()
      Analyze_Samples(4, samples)
      samples_remaining -= 4
   wait_until_ready()
   Analyze_Samples(samples_remaining, samples)
   wait_until_ready()
   Process_Analysis_Result()
   wait_until_ready()
   return Get_Analysis_Result()
}
```

Figure 4.2: Pseudocode for the analysis process

lowest error total and has that total and the corresponding predictor order as its outputs.

When a Process Analysis Result command is issued, the Rice calculation block loads the total number of samples in the block into a shift register and performs successive left shifts until the contents are greater than the given error total. Upon completion of that step, the analysis is finished and the recommended parameters may be read by the host software using a Get Analysis Result command.

Requesting the analysis results does not clear the internal state of the instruction. Instead, the Reset Analysis command is used to clear the state so that a new block of samples may be analyzed. This is so that the software program may get the assessment of a part of the block and the whole, perhaps in the process of deciding upon an optimal block size. For example, if the analysis results for the first half of a block differ from those for the entire block, the program may decide to divide the one block into two as they would be more efficiently encoded with different parameters.

4.2.2 Coder

The encoding portion of the instruction is designed to be given a block of the ECG signal as input and return the Rice coded residual as output. Only the residual is returned so that the software may choose any stream format it prefers rather than having one coded directly in the instruction. The encoded residual may be of arbitrary length, and so the instruction must return portions of the output as they are completed. The encoded result is dependent on the correlation between successive samples and thus there is no one-to-one relationship between a sample of the signal and the output. Further, the variable bit rate nature of lossless encoding means that a group of samples sent to the encoder may result in any number of bits in the output.

Consequently, the encoder is designed to construct the result into a 32-bit shift

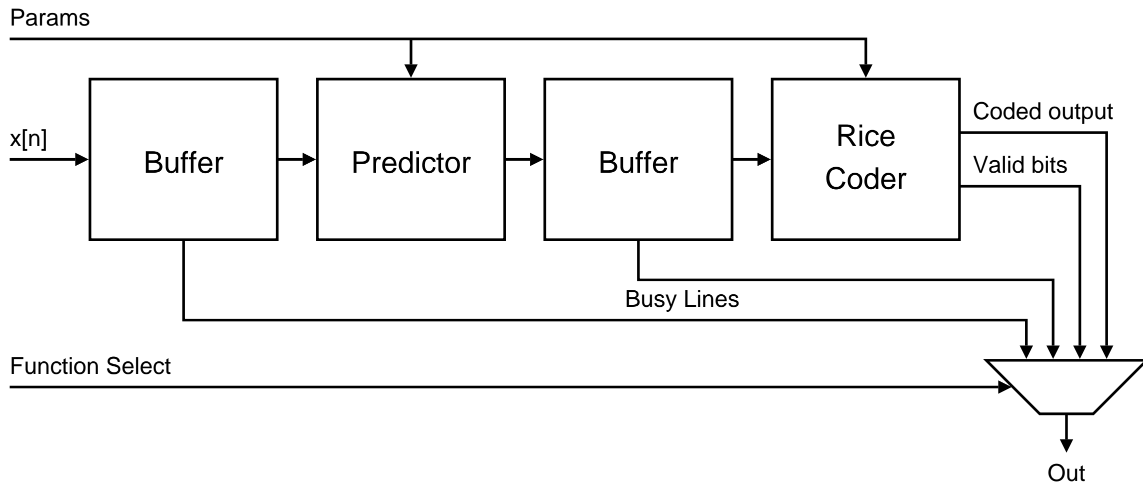


Figure 4.3: Block Diagram of the Instruction

register whose contents are returned upon a Get Encoded Result instruction call. When the full 32-bit word has been clocked in, the encoding process pauses and the instruction returns a Result Ready code to Query Status calls. When the encoded result has been read by the host software, the encoding process resumes until the result is again full or the samples sent to the encoder have been exhausted.

The result register is 32 bits wide to match the width of a custom instruction result port. The programmer must take care to not directly store the pieces of the result sequentially as the Nios CPU is little endian and will cause byte order issues.

The encoder (Figure 4.3) is identical to the core of Figure 3.1 save for implementation specifics. The FIFO buffer between the predictor and the Rice coder holds the error samples as they wait to be encoded and is necessary because Rice coding a residual sample may take many more cycles than predicting a sample. The predictor is identical to Figure 3.2 and uses non-vendor-specific multiply logic that completes a multiply in four cycles. A predictor using Altera multiply megafunctions would use the Stratix chip's DSP blocks and would therefore be capable of single-cycle multiplication.

The Rice coder (Figure 4.4) operates in stages. The mapped residual sample is loaded into a parallel input shift register/down-counter. In the first stage, the lower

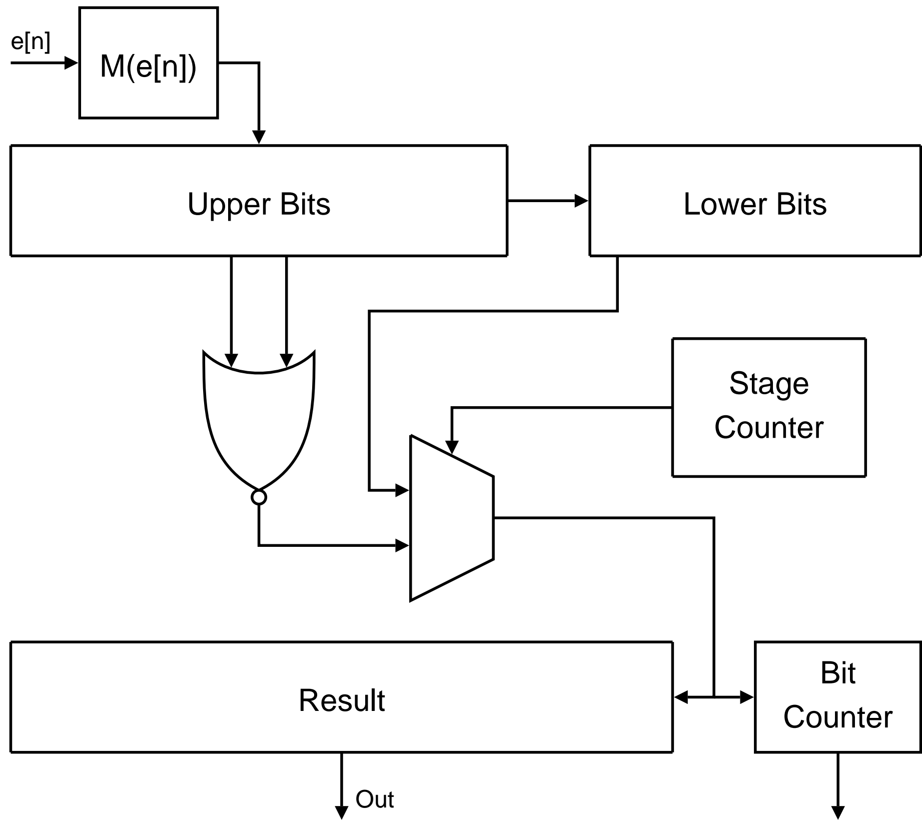


Figure 4.4: Block Diagram of the Rice Coder

k bits are shifted out of the register and into a 1-bit wide stack. Next, the shift register/counter (now containing only the upper bits of the sample) counts down by one each cycle as the chain of zeros forming the unary portion of the Rice code is clocked into the result register. When the counter reaches zero, the terminating one is clocked into the result. Finally, the lower bits of the sample are popped out of the stack and into the result register, forming the completed code.

When all samples of a block have been passed to the instruction and the encoder is awaiting input, the result shift register may be partially filled. These final bits are returned upon a result request call, but the contents give no indication as to how many of the 32 bits are part of the encoded bitstream. Therefore an extra function of the instruction returns the number of bits shifted into the register since the last

result read so that the host software can append the correct set of bits to the stream. Unlike when a full 32-bit result is read, reading a partial result does not affect the state of the instruction.

The return value of a call to the Get Encoded Result function is the result register contents when the instruction is idle and 0x00000001 when it is busy. However, any 32-bit value including 0x00000001 is a valid encoding result and thus that return value does not necessarily indicate failure. Instead, a Get Encoded Result call should only be performed after a Query Status call returns Ready or Result Available.

Listings of important Verilog code for the instruction are in Appendix A.

```
function check_ready()
{
   while true:
      status := Query_Status()
      if status == 0:
         return false
      else if status == 2
         return true
}

function check_output()
{
   while true:
      if check_ready()
         Get_Encoded_Result()
         store encoded result
      else
         return
}

function Encode_block(sampledata, order, rparam)
{
   warmup := first order samples in sampledata
   Begin_New_Block(warmup, order, rparam)
   while samples_remaining is greater than 3:
      Encode_Samples(4, sampledata)
      samples_remaining := samples_remaining - 4
      check_output()
   Encode_Samples(samples_remaining, sampledata)
   check_output()
   Get_Encoded_Result()
   Get_Valid_Bits()
   return
}
```

Figure 4.5: Pseudocode for the encoding process

Chapter 5

Results

This chapter shows the results of testing the custom instruction encoder on the criteria of compression level and encoding speed. Also included are encoding speed results for an implementation of the encoding algorithm chosen using only the base Nios instruction set.

Evaluation of lossy ECG encoders uses measurements related to the amplitude difference between the original and the reconstructed signal. The standard quantitative measurement is the percent root mean square difference (PRD), which is given by:

$$\mathrm{PRD} = \sqrt{\frac{\sum_{n=1}^{N}(x(n) - \tilde{x}(n))^2}{\sum_{n=1}^{N} x^2(n)}} \times 100 \tag{5.1}$$

where a lower PRD value indicates that the reconstruction approximates the original more closely and is therefore better. However, qualitative evaluations are almost invariably used due to humans being better judges of which details of the signal are important.

Lossless algorithms introduce no distortion or noise by definition and are therefore evaluated using only quantitative criteria.

The Free Lossless Audio Codec (FLAC) stream format [31] was chosen as the format into which the encoded ECG signals were stored. It was selected as opposed to creating an ad hoc stream format because of its suitability for storing ECG signal parameters (e.g., very low sampling rates, arbitrary bits per sample) and availability of the format specification and tools to encode, decode, and test the integrity

of FLAC streams. Also, the format has a defined block type for fixed N^{th} order differentiators as used in this encoder where only the order is needed to denote the predictor rather than storing all coefficients of the predictor.

Several software programs and libraries were written for the Nios platform to facilitate the testing of the instruction. The libraries provided functions for interpreting the RIFF WAVE format [32] with LPCM encoding in which the uncompressed ECG signals were stored, and for writing the encoded streams in FLAC format. The FLAC encoder library contains two sets of analysis and encoder functions, one utilizing the custom instruction and the other using only the base Nios instruction set. The hardware-assisted path uses the `flac__analyze_frame_cinst()` and `flac__code_frame_cinst()` functions for analysis and encoding, respectively and the software path uses `flac__analyze_frame_soft()` and `flac__code_frame_fixed_soft()`. The source code for the FLAC encoder library is listed in Appendix B.3.

5.1 Compression Ratio

Twenty-five records from the MIT-BIH Arrhythmia database [26, 27] were obtained from Physionet.org [33] and used as test signals for the instruction. Each channel in each record was compressed independently. The list of records used is in Appendix C. The signals were all 11-bit, 360Hz LPCM signals at 650,000 samples in length. For the purposes of comparison, the uncompressed file size used in the calculations is 893,750 bytes, which is the size of the raw signal data. Using this file size leads the compression ratios given in the results to be somewhat conservative, as no metadata size is taken into account, nor is the inefficiency of the format in which the signals were obtained, where two samples were packed into three-byte, effectively making the signals 12 bits per sample with one bit always zero. The compressed filesizes used in the calculations are the full FLAC stream sizes including all metadata and

header information needed to properly parse and decode the stream.

The compression ratio is a measure of the amount of data size reduction achieved and is calculated by:

$$\text{CR} = \frac{\text{Uncompressed size}}{\text{Compressed size}} : 1. \tag{5.2}$$

More efficient encoding results in smaller compressed files and thus a higher compression ratio. Selected compression results are in Table 5.1. There is notable variation of the ratios of different signals, the reasons for which will be discussed in Section 6.2.1. On average, the algorithm compressed the signals at a ratio of 2.55:1 with a standard deviation of 0.17.

Record	**Channel**	**Ratio**
101	1	2.58:1
101	2	3.15:1
103	2	2.48:1
107	1	2.41:1
200	1	2.49:1
200	2	2.75:1
201	1	2.79:1
201	2	2.75:1
203	2	2.44:1
205	2	2.91:1
212	1	2.28:1
217	1	2.43:1
217	2	2.76:1

Table 5.1: Compression Ratio of encoded ECG files

The lossless nature of the algorithm and encoder was verified by decoding the encoded signals using the reference FLAC tools and comparing the resulting sample data with that of the original unencoded file.

Discussion of the results and how they relate to the characteristics of each signal is in Section 6.2.

5.2 Speed

Comparing the encoding speed of the custom instruction to other algorithms cannot be easily preformed as the necessary software is not usable on the Nios platform. Thus the speed comparisons are limited to the custom instruction versus a software version of the same algorithm.

A prefix function of the instruction assists in measuring the number of cycles taken to perform tasks. The function contains a 32-bit counter that increments each cycle. When that prefix function is called, the counter's contents are returned by the instruction and then cleared. Thus the function returns the number of cycles elapsed since the previous call to it. The maximum measurable interval is $2^{32} - 1$ cycles, which at the development board's clock speed of 50MHz is approximately 86 seconds. By inspection during testing, none of the tasks performed required such a long time to complete, and thus it can be trusted that the cycle counts returned were not rendered inaccurate due to counter overflow.

Table 5.2 contains the number of cycles needed to complete the encoding of several ECG signals from the MIT database as well as the ratio of the software to hardware results, which is the factor by which the custom instruction increases the speed of encoding. The cycle counts listed do not include the slow process of reading the source ECG files from the CompactFlash card, nor the writing of the encoded files to storage. These operations were excluded in order to isolate the operations involving the two implementations so that the measurements are more reflective of their performances. Since the slowest CompactFlash Memory Mode timing was in use, including the time taken during card access would obscure the speed differences between the software and custom instruction encoders. Instead, the cycle counts shown represent the accumulation of the times taken to encode each block of a signal, measured between the point after the unencoded samples have been loaded

Record	Channel	Cycle count		Ratio
		Custom Instruction	Software	
101	1	61,669,060	287,677,561	4.66
101	2	55,472,228	280,186,046	5.05
103	2	62,995,648	288,389,304	4.58
107	1	64,021,296	288,547,608	4.51
200	1	63,078,187	288,529,902	4.57
200	2	59,633,752	284,361,979	4.77
201	1	58,793,517	284,094,026	4.83
201	2	59,575,108	286,037,647	4.80
203	2	64,013,198	288,994,769	4.51
205	2	57,553,743	281,242,281	4.89
212	1	66,336,147	290,072,014	4.37
217	1	63,790,234	288,293,535	4.52
217	2	59,287,305	285,388,904	4.81

Table 5.2: Cycles required to encode MIT-BIH ECG signals using hardware and software

into memory and the point after the encoded frame has been completed. Discussion of these results is in Section 6.2.2.

Chapter 6
Discussion and Conclusions

The benefits of a hardware assisted encoder are mainly the offloading of the encoding burden from the main CPU and the improvements in encoding speed from parallelization and available operations. General-purpose CPUs are limited by the range of operations it is capable of executing and by the number of available execution pipelines, which is almost always 1 for embedded devices. Custom logic can be tailored to perform desired types of calculations and can have any number of pipelines to parallelize any operations than can be. This chapter makes an assessment of the algorithm and the implemented encoder to determine its success in those areas and offers suggestions on how that performance may be improved.

6.1 Objectives

The objectives of this thesis were to:

- develop a lossless encoding algorithm that is:
    - low-complexity, and
    - capable of reducing the data size of ECG signals by a ratio of at least 2:1,
- implement the algorithm in a CPU custom instruction for an embedded system on a programmable chip, and
- accelerate the encoding process compared to a software-only encoder.

The algorithm and custom instruction encoder met these objectives:

- the lossless nature of the encoder was verified in Section 5.1,
- the low complexity of the decoding algorithm was explained in Section 3.2,
- the compression ratio target was met in all tested cases as seen in Table 5.1, and
- the speed was increased over software as seen in Table 5.2.

6.2 Performance Analysis

6.2.1 Compression

The algorithm implemented was selected on the basis of suitability to the class of signals to be compressed (ECGs) as well as the complexity of decoding. The encoding process is generally low-complexity, since ECG signals are such a narrow class that all tend to exhibit energy spectra that the chosen predictors work well on. As well, these predictors lend themselves to particularly simple methods of deciding upon the predictor order and the Rice coding parameter, both of which can be estimated using results from a single analysis process.

However, there is nothing in the algorithm that is designed to take advantage of the normal structure of ECG signals. Silence between beats is treated no differently than the beat components, even though they have greatly different properties. The highly periodic nature of ECG signals is also not exploited in any way.

Even then, the algorithm achieved a reduction in data size of at least 2:1 on all tested signals as specified in the objectives, with some consistent results on what signal properties lead to better compression.

The two main influences on the compressibility of a signal are the strength of the signal and the noise level, in particular in the high frequency range. Stronger signals

and ones with greater high-frequency noise tend to be less compressible. The impact of noise intuitively makes sense as the predictor response shown in Figure 3.7 shows gain rather than attenuation at frequencies above one-sixth of the sampling rate.

Note the compression ratios for both channels of record 201, which are higher than average. The signals in that record are not particularly weak, but contain very little noise. The channels of record 217 are noticeably different in amplitude, with channel 2 being weaker and as a result more compressible. Channel 2 of record 200 contains severe noise (though not high-frequency) in some regions of the signal, but is otherwise a weak signal. Channel 1 of record 212, which has the lowest compression ratio, is a relatively strong signal with a few periods of high-frequency noise. The greatest discrepancy between compressibility of channels is in record 101, where the highly compressible channel 2 is characterized by an extremely low signal level, to the point that it is considered unusable by cardiologists.

6.2.2 Speed

The encoding algorithm chosen is very basic and straightforward. The use of only integer arithmetic guarantees that all calculations used are lossless. The complexity of operations is low as specified in the objectives, and the algorithm as a whole is quite conducive to a software implementation. This is what leads to the rather low factor by which the custom instruction increases the encoding speed.

The main areas in which parallelization is leveraged are the calculation of the error for each order in the Analysis block and the encoder's predictor filter, with little else being aided significantly by hardware.

The factor by which the custom instruction decreases the encode time ranged mostly between 4.5 and 5.0. This is a lower difference than one may typically expect from a hardware-assisted implementation — it is likely that the low complexity of the encoding process is quite conducive to a software version. It is interesting to note that

there is a correlation between the compressibility of a signal and the speed difference factor — signals with a higher compressibility had lower factors than signals with low compressibility. This indicates that the custom instruction is faster at packing the variable-length Rice codes than the software is.

Taking into account the fact that the custom instruction does not block the CPU from addressing other tasks during the encoding process, the software implementation does not perform so slowly that a hardware encoder would make a noticeable difference in a real-world situation. The hardware encoder used less than six seconds of CPU time to encode a 30-minute ECG signal — an encode rate of 300 times real time. Even with sampling rates approaching 1,000Hz and CPU clock rates at half or less of that of the development board, the software version of the algorithm would still be capable of encoding the signal using only a fraction of the available cycles. A more complex encoding algorithm that compresses more efficiently and is more susceptible to parallelization is necessary to warrant a hardware-assisted implementation over software-only.

6.3 Future Work

6.3.1 Hardware Implementation

The encoder, despite being designed to operate without blocking the CPU, still requires the CPU to move data between the instruction and memory. The interface between the CPU and the instruction is a bottleneck as it frequently requires the attention of the CPU to make small (one or two-word) memory accesses. To remove this limit, the encoder may be implemented as an Avalon peripheral rather than as a CPU instruction.

The custom instruction could be fitted with an interface external to the CPU that communicates with an Avalon peripheral, but to do so would be give little to

no benefit over creating the encoder solely as an Avalon device and would still occupy one of the five custom instruction slots.

As an Avalon implementation, the encoding process would begin with the CPU providing the addresses and sizes of the source and destination blocks of memory to the encoder. At the end of the encoding process, an IRQ line would be asserted to alert the CPU of the completion. Status registers would contain information about the encoded residual, e.g., size and parameters used, as well as indicate the success or failure (e.g., the encoded result being larger than the unencoded samples) of the encoding. Such a design would require no CPU attention between the acquisition of the samples and the end of the encoding process. A slave-only peripheral could be used in conjunction with a Direct Memory Access (DMA) controller for memory access. Alternatively, the encoder may be designed with two Avalon ports: one slave to receive commands from the CPU, and a master to perform memory reads and writes.

6.3.2 Codec Design

Although it does not pertain directly to the use of a custom instruction, the design of the encoder can be improved within the limits of the design goals. Changes to the algorithm would need to avoid a significant increase in decoder complexity to remain consistent with the goals of the thesis.

6.3.2.1 Short-Term Prediction

Additional hardware resources can be allocated to solving for more optimal static predictors. This may be an area where the use of custom logic would bring much greater advantage over software. Calculating the autocorrelation of the signal and subsequently, the predictor coefficients could benefit greatly from parallelization in hardware and therefore be much faster than it would be in software. Such predictors

would not impact the decode complexity significantly, retaining the suitability for use with embedded signal readers. Research [34, 25] indicates that a maximum order of 5 would be approximately optimal as larger orders would be minimally beneficial due to the small autocorrelation function of ECG signals using longer lags.

6.3.2.2 Long-Term (Inter-beat) Prediction

In most cases, there will be a great deal of similarity between one beat and the next. The short-term prediction in this encoder does not take any advantage of this, even though a lot of redundant information between multiple beats could potentially be removed. The delay between consecutive beats (the R-R interval) may be detected using a low-complexity QRS detection algorithm such as the one given in [35] so that a delayed copy of the signal may be subtracted from the original. Thus the first beat remains intact, but subsequent beats appear as the difference between it and the previous one. This technique would become more effective with a longer block size, but not one so long that the ECG characteristics change significantly over the block's length. The delay from the previous block can be used as a starting point for the search for the delay in the next block, speeding up the search process. One lossless ECG codec [36] implements this technique using delay and gain parameters.

6.3.2.3 Entropy Coding

While Rice coding is very close to optimal for this type of data in terms of prefix codes, the limitation that each residual block is encoded with a single Rice parameter prevents adaptivity to the different contexts (e.g., beat components) of an ECG.

Context Modeling is the general term for partitioning the signal into regions to be encoded independently. A typical block will contain at least one full beat of the waveform, likely several, and applying the idea of partitioning the signal for temporal adaptivity may bring benefits by allowing components such as the QRS complex to

be encoded using a larger Rice parameter as the residual values are likely to be greater in that area of the beat.

One basic method is to partition the residual into a certain number of equal-length parts. To make the idea more effective on ECGs, the components may be detected and codes may be inserted prior to the residual to denote the positions, lengths, and encoding parameters of the different regions. Information from QRS detection in long-term prediction may be reused for this purpose.

Another option is to take the idea of adaptive prediction and apply it to Rice coding by having the Rice parameter adapt to the residual based on the size of the output codes. This would have the same effect on the decoding algorithm as an adaptive predictor would, requiring the adaptation algorithm to be included in the decoder, increasing its complexity and symmetry. This idea is present in the Apple Lossless Audio Codec (ALAC) from Apple Computers [37].

6.3.2.4 Framing

The FLAC framing format used to store the results uses indices to denote the fixed predictor used rather than storing the full set of coefficients, and so little space is used. Should general predictors be used, though, the space used to store the coefficients becomes a concern. Storing them with linear quantization as FLAC does is not very efficient, and so other codecs such as MPEG-4 ALS [38, 39] have developed other techniques involving entropy coding and representation as reflection coefficients rather than using linear quantization.

References

[1] The Electrocardiogram — Looking at the heart of electricity. NobelPrize.org. Accessed on 2005-10-26. [Online]. Available: http://nobelprize.org/medicine/educational/ecg/ecg-readmore.html

[2] L. Schamroth, *An introduction to Electrocardiography*, 6th ed. Oxford: Blackwell Scientific Publications, 1982.

[3] C. Shannon, "A Mathematical Theory of Communication," *Bell Systems Technical Journal*, vol. 27, pp. 379–423, 623–656, Jul./Oct. 1948.

[4] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proc. IRE*, vol. 40, pp. 1098–1101, Sep. 1952.

[5] S. Golomb, "Run-Length Encodings," *IEEE Trans. Inf. Theory*, vol. 12, no. 3, pp. 399–401, Jul. 1966.

[6] R. Gallager and D. Van Voorhis, "Optimal Source Codes for Geometrically Distributed Integer Alphabets," *IEEE Trans. Inf. Theory*, vol. 21, pp. 228–230, Mar. 1975.

[7] *ARM AMBA Specification*, ARM Ltd. Std., Rev. 2.0, May 1999.

[8] *Nios 3.0 CPU Data Sheet*, DS-NIOSCPU-2.1, Altera Corp., Mar. 2003, version 2.1.

[9] *Avalon Bus Specification*, MNL-AVABUSREF-2.3, Altera Corp., Jul. 2003, version 2.3.

[10] *Custom Instructions for the Nios Embedded Processor*, AN-188-1.2, Altera Corp., Sep. 2002, version 1.2.

[11] *Developing Peripherals for SOPC Builder*, AN333-1.0, Altera Corp., March 2004, version 1.0.

[12] Cygwin. [Online]. Available: http://www.cygwin.com/

[13] *Nios Development Board Reference Manual, Stratix Professional Edition*, MNL-NIOSSTXPRO-1.1, Altera Corp., Jul. 2003, version 1.1.

[14] *IEC 60027-2: Letter symbols to be used in electrical technology — Part 2: Telecommunications and electronics*, 2nd ed., International Electrotechnical Commission, Nov. 2000.

[15] *CF+ and CompactFlash Specification*, CompactFlash Association Std., Rev. 3.0, Dec. 2004.

[16] *PC Card Standard*, Personal Computer Memory Card International Association Std., Rev. 8.0, Apr. 2001.

[17] J. I. Makhoul, "Linear Prediction: A Tutorial Review," *Proc. IEEE*, vol. 63, pp. 561–580, Apr. 1975.

[18] P. Craven and M. Gerzon, "Lossless Coding for Audio Discs," *J. AES*, vol. 44, no. 9, pp. 706–720, Sep. 1996.

[19] P. Craven, M. Law, and R. Stuart, "Lossless Compression Using IIR Prediction Filters," presented at the 102nd AES Convention, Munich, Mar. 22–25, 1997, preprint 4415.

[20] N. S. Javant and P. Noll, *Digital Coding of Waveforms.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

[21] N. Levinson, "The Wiener RMS (root mean square) Error Criterion in Filter Design and Prediction," *J. Math. Phys.*, vol. 25, pp. 261–278, 1947.

[22] R. Rice, "Some Practical Universal Noiseless Coding Techniques," Jet Propulsion Laboratory, Pasadena, California, Tech. Rep., 1979.

[23] M. Purat, T. Liebchen, and P. Noll, "Lossless Transform Coding of Audio Signals," presented at the 102nd AES Convention, Munich, Mar. 22–25, 1997, preprint 4414.

[24] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[25] A. Koski, "Lossless ECG encoding," *Comput. Meth. Prog. Biomed.*, vol. 52, pp. 23–33, 1997.

[26] G. B. Moody. MIT-BIH Arrhythmia Database, 3rd ed. Massachusetts Institute of Technology, Cambridge MA. [CD-ROM].

[27] G. B. Moody and R. G. Mark, "The Impact of the MIT-BIH Arrhythmia Database," *IEEE Eng. Med. Biol. Mag.*, vol. 20, no. 3, pp. 45–50, May/Jun. 2001.

[28] European ST-T Database. European Society of Cardiology. [Online]. Available: http://www.physionet.org/physiobank/database/edb/

[29] G. B. Moody, R. G. Mark, and A. L. Goldberger, "Evaluation of the TRIM ECG Data Compressor," *Proc. Comput. Cardiol.*, vol. 15, pp. 167–170, 1988.

[30] A. Bruekers, A. Oomen, and R. van der Vleuten, "Lossless Coding for DVD audio," presented at the 101st AES Convention, Los Angeles, Nov. 8–11, 1996, preprint 4358.

[31] J. Coalson. (2003, Jan.) FLAC — Format. [Online]. Available: http://flac.sourceforge.net/format.html

[32] *Multimedia Programming Interface and Data Specifications*, IBM Corp./Microsoft Corp. Std., Rev. 1.0, Aug. 1991.

[33] G. B. Moody, R. G. Mark, and A. L. Goldberger, "PhysioNet: A Web-Based Resource for the Study of Physiologic Signals," *IEEE Eng. Med. Biol. Mag.*, vol. 20, no. 3, pp. 70–75, May/Jun. 2001.

[34] G. Nave and A. Cohen, "ECG Compression Using Long-Term Prediction," *IEEE Trans. Biomed. Eng.*, vol. 40, no. 9, pp. 877–885, Sep. 1993.

[35] I. S. N. Murthy and M. R. Rangaraj, "New Concepts for PVC Detection," *IEEE Trans. Biomed. Eng.*, vol. 26, no. 7, pp. 409–416, Jul. 1979.

[36] C. D. Giurcăneanu, I. Tăbuş, and Ş. Mereuţă, "Using contexts and R-R interval estimation in lossless ECG compression," *Comput. Meth. Prog. Biomed.*, vol. 67, pp. 177–186, 2002.

[37] D. Hammerton. (2005, Mar.) Reverse Engineered ALAC Decoder. [Online]. Available: http://craz.net/programs/itunes/alac.html

[38] T. Liebchen, T. Moriya, N. Harada, Y. Kamamoto, and Y. A. Reznik, "The MPEG-4 Audio Lossless Coding (ALS) Standard — Technology and Applications," presented at the 119th AES Convention, New York, Oct. 7–10, 2005, preprint 6589.

[39] T. Liebchen. (2006, Apr.) MPEG-4 Audio Lossless Coding (ALS). [Online]. Available: http://www.nue.tu-berlin.de/forschung/projekte/lossless/mp4als.html

Appendix A

Verilog Code

A.1 Instruction (cinst_lpcrice.v)

```
1  module cinst_lpcrice(clk,
2  reset,
3  clk_en,
4  start,
5  prefix,
6  dataa,
7  datab,
8  result);
9
10 // CPU interface
11 input clk;
12 input clk_en;
13 input start;
14 input reset;
15 input [10:0] prefix;
16 input [31:0] dataa;
17 input [31:0] datab;
18
19 output [31:0] result;
20
21
22
23 `define INCLUDE_TIMER
24
25 reg past_ready_n;                //
26 reg filter_load_warmup;          // signal
27 reg rice_reading_riced_output;   //
28 reg rice_clear;                  // clear signal to the
       rice coder
```

```
29                                      // used when loading
                                           parameters
30 reg [1:0] lpc_order;                 // stored LPC order
31 reg [2:0] rice_param;                // stored rice parameter
32 reg [2:0] numfiltout;                // number of unencoded
      samples from the filter
33 reg [2:0] numsamples;                // number of samples
      available to the filter
34
35 reg [15:0] sample_1;                 // First (oldest) sample
       available
36 reg [15:0] sample_2;                 //
37 reg [15:0] sample_3;                 //
38 reg [15:0] sample_4;                 //
39
40 reg [15:0] filter_wu_1;              // Filter warmup samples
41 reg [15:0] filter_wu_2;              //
42 reg [15:0] filter_wu_3;              //
43
44 reg [18:0] filt_out_buf_1;           // filter output samples
45 reg [18:0] filt_out_buf_2;           //
46 reg [18:0] filt_out_buf_3;           //
47 reg [18:0] filt_out_buf_4;           //
48
49
50 `ifdef INCLUDE_TIMER
51 reg [31:0] timer_count;              // Timer cycle count
52 `endif
53
54 reg [31:0] result;                   //
55
56 wire filter_input_valid;             // Flag to tell the
      filter to read next sample
57 wire filter_ready;                   // Flag from the filter
      saying it's ready for a sample
58 wire busy;                           // Global busy flag.
59 wire rice_datain_avail;              // Flag telling rice
      coder that a sample is available
60 wire rice_busy;                      // Rice coder busy
      signal
61 wire rice_output_full;               // Rice coder's output
      buffer full flag
62 wire analyze_start;                  // Start signal for
      analysis
```

```
63 wire [4:0] rice_valid_bits;      // Number of filled bits
     in rice coder buffer
64 wire [15:0] filter_input;        // Alias for sample_1
65 wire [18:0] filter_output;       // Output from filter
66 wire [18:0] rice_datain;         //
67 wire [31:0] rice_out;            //
68 wire [31:0] analyze_result;     //
69
70 wire analyze_busy;
71
72 assign busy = ((rice_busy || ~filter_ready || (numsamples
     > 3'd0) || (numfiltout > 3'd0)) && ~rice_output_full)
    || analyze_busy;
73 assign filter_input = sample_1;
74 assign rice_datain = filt_out_buf_1;
75 assign rice_datain_avail = (~rice_busy && (numfiltout >
    3'd0) && ~rice_output_full);
76 assign filter_input_valid = (filter_ready && (numsamples
    > 3'd0));
77 assign analyze_start = (start && prefix[4] && ~prefix[3])
    ;
78
79
80 always @(posedge clk or posedge reset) begin
81    if (reset) begin
82       sample_1 <= 16'd0;
83       sample_2 <= 16'd0;
84       sample_3 <= 16'd0;
85       sample_4 <= 16'd0;
86       numsamples <= 3'd0;
87       rice_param <= 3'd0;
88       lpc_order <= 2'd0;
89       past_ready_n <= 1'b0;
90       rice_clear <= 1'b0;
91       filt_out_buf_1 <= 19'd0;
92       filt_out_buf_2 <= 19'd0;
93       filt_out_buf_3 <= 19'd0;
94       filt_out_buf_4 <= 19'd0;
95       filter_wu_1 <= 16'd0;
96       filter_wu_2 <= 16'd0;
97       filter_wu_3 <= 16'd0;
98       numfiltout <= 3'd0;
99       filter_load_warmup <= 1'b0;
100      rice_reading_riced_output <= 1'b0;
101      result <= 32'd0;
```

```
102 `ifdef INCLUDE_TIMER
103       timer_count <= 32'd0;
104 `endif
105    end //reset
106
107    else if (clk_en) begin
108       //Instruction handling
109       if (start) begin
110          casex(prefix)
111          11'b00000000000: begin  //Send params
112             if (busy) begin
113                result <= 32'd1;
114             end //busy
115             else begin
116                filter_load_warmup <= 1'b1;
117                rice_clear <= 1'b1;
118                sample_1 <= 16'd0;
119                sample_2 <= 16'd0;
120                sample_3 <= 16'd0;
121                sample_4 <= 16'd0;
122                numsamples <= 3'd0;
123                past_ready_n <= 1'b0;
124                lpc_order <= dataa[20:19];
125                rice_param <= dataa[18:16];
126                filter_wu_1 <= dataa[15:0];
127                filter_wu_2 <= datab[31:16];
128                filter_wu_3 <= datab[15:0];
129                filt_out_buf_1 <= 19'd0;
130                filt_out_buf_2 <= 19'd0;
131                filt_out_buf_3 <= 19'd0;
132                filt_out_buf_4 <= 19'd0;
133                numfiltout <= 3'd0;
134                rice_reading_riced_output <= 1'b0;
135                result <= 32'd0;
136             end
137          end // 11'b00000000000
138
139          11'b00000000001: begin // Query Status
140             result <= {30'd0, rice_output_full, busy};
141          end // 11'b00000000001
142
143          11'b00000000010: begin // get output value
144             if (busy)
145                result <= 32'd1;
146             else begin
```

```
147                 rice_reading_riced_output <= 1'b1;
148                 result <= rice_out;
149              end
150           end // 11'b00000000010
151
152           11'b00000000011: begin // Get numvalidbits
153              if (busy)
154                 result <= 32'd1;
155              else
156                 result <= {27'd0, rice_valid_bits};
157           end // 11'b00000000011
158
159           11'b00000001xxx: begin // Send samples
160              if (busy)
161                 result <= 32'd1;
162              else begin
163                 sample_1 <= dataa[31:16];
164                 sample_2 <= dataa[15:0];
165                 sample_3 <= datab[31:16];
166                 sample_4 <= datab[15:0];
167                 numsamples <= prefix[2:0];
168                 result <= 32'd0;
169              end // ~busy
170           end // 11'b00000001xxx
171           11'b00000010xxx: begin  // Analysis
172              result <= analyze_result;
173           end
174 `ifdef INCLUDE_TIMER
175           11'b10000000000: begin // timer control
176              timer_count <= 32'd0;
177              result <= timer_count;
178           end
179 `endif
180
181           endcase
182        end // start
183
184        past_ready_n <= ~filter_ready;
185        if (filter_load_warmup) //warmup load to filter
              only needs one cycle
186           filter_load_warmup <= 1'b0;
187        if (rice_reading_riced_output)
188           rice_reading_riced_output <= 1'b0;
189        if (rice_clear)
190           rice_clear <= 1'b0;
```

```
191          if (rice_datain_avail) begin
192             case (numfiltout)
193             3'd1: begin  //no 3'b000 since that would mean
                   rice_datain_avail = 0
194                if (filter_ready && past_ready_n) begin
195                   filt_out_buf_1 <= filter_output;
196                end // filter_ready && past_ready_n
197                else begin
198                   numfiltout <= numfiltout - 1'b1;
199                   filt_out_buf_1 <= 19'd0;
200                end
201             end // 3'd1
202             3'd2: begin
203                filt_out_buf_1 <= filt_out_buf_2;
204                if (filter_ready && past_ready_n) begin
205                   filt_out_buf_2 <= filter_output;
206                end // filter_ready && past_ready_n
207                else begin
208                   numfiltout <= numfiltout - 1'b1;
209                   filt_out_buf_2 <= 19'd0;
210                end
211             end // 3'd2
212             3'd3: begin
213                filt_out_buf_1 <= filt_out_buf_2;
214                filt_out_buf_2 <= filt_out_buf_3;
215                if (filter_ready && past_ready_n) begin
216                   filt_out_buf_3 <= filter_output;
217                end // filter_ready && past_ready_n
218                else begin
219                   numfiltout <= numfiltout - 1'b1;
220                   filt_out_buf_3 <= 19'd0;
221                end
222             end // 3'd3
223             3'd4: begin
224                filt_out_buf_1 <= filt_out_buf_2;
225                filt_out_buf_2 <= filt_out_buf_3;
226                filt_out_buf_3 <= filt_out_buf_4;
227                filt_out_buf_4 <= 19'd0;
228                numfiltout <= numfiltout - 1'b1;
229             end // 3'd4
230             endcase
231          end // rice_datain_avail
232          if (filter_ready) begin
233             if (numsamples > 3'd0) begin // sample available
                    to send to filter
```

```
234                 sample_1 <= sample_2;
235                 sample_2 <= sample_3;
236                 sample_3 <= sample_4;
237                 sample_4 <= 16'bXXXXXXXXXXXXXXXX;
238                 numsamples <= numsamples - 1'b1;
239                 //note: filter_input_valid is not asserted
                       here; it is combinational.
240              end // (numsamples > 3'd0)
241
242
243              if (past_ready_n && ~rice_datain_avail) begin //
                    transition from filter_ready == 0 to 1 -> new
                    filter output
244                 case(numfiltout)
245                 3'b000: begin
246                    filt_out_buf_1 <= filter_output;
247                    numfiltout <= numfiltout + 1'b1;
248                 end
249                 3'b001: begin
250                    filt_out_buf_2 <= filter_output;
251                    numfiltout <= numfiltout + 1'b1;
252                 end
253                 3'b010: begin
254                    filt_out_buf_3 <= filter_output;
255                    numfiltout <= numfiltout + 1'b1;
256                 end
257                 3'b011: begin
258                    filt_out_buf_4 <= filter_output;
259                    numfiltout <= numfiltout + 1'b1;
260                 end
261                 endcase
262              end // past_ready_n && ~rice_datain_avail
263           end // filter_ready
264  `ifdef INCLUDE_TIMER
265           timer_count <= (timer_count + 1'b1);
266  `endif
267
268        end //clk_en
269  end // posedge clk or posedge reset
270
271
272
273  lpc_coder filter1(.clk(clk),
274                 .clk_en(clk_en),
275                 .reset(reset),
```

```
276             .datain_avail(filter_input_valid),
277             .indata(filter_input),
278             .result(filter_output),
279             .wu_1(filter_wu_1),
280             .wu_2(filter_wu_2),
281             .wu_3(filter_wu_3),
282             .ready(filter_ready),
283             .load_warmup(filter_load_warmup),
284             .order(lpc_order)
285             );
286
287 rice_coder rice1(.clk(clk),
288             .clk_en(clk_en),
289             .reset(reset),
290             .value_in(rice_datain),
291             .valuein_avail(rice_datain_avail),
292             .output_full(rice_output_full),
293             .rice_param(rice_param),
294             .rice_clear(rice_clear),
295             .busy(rice_busy),
296             .reading_output(rice_reading_riced_output),
297             .bits_shiftin(rice_valid_bits),
298             .value_out(rice_out)
299             );
300
301 analyze analyze1(.clk(clk),
302             .clk_en(clk_en),
303             .ext_reset(reset),
304             .dataa(dataa),
305             .datab(datab),
306             .start(analyze_start),
307             .operation(prefix[2:0]),
308             .eval_busy(analyze_busy),
309             .result(analyze_result)
310             );
311
312 endmodule
```

A.2 Analysis (analyze.v)

```
1 module analyze(clk,
2 clk_en,
3 dataa,
4 datab,
5 start,
```

```
6  ext_reset,
7  operation,
8  eval_busy,
9  result);
10
11
12 input [31:0] dataa, datab;
13 input clk, clk_en, start, ext_reset;
14 input [2:0] operation;
15 output eval_busy;
16 output [26:0] total_error_1, total_error_2, total_error_3
      ;*/
17 output [31:0] result;
18
19
20 wire ord_1_lt_2, ord_1_lt_3, ord_2_lt_3, eval_busy;
21 //wire param_0, param_1, param_2, param_3, param_4,
      param_5, param_6, param_7;
22 wire [18:0] error_1, error_2, error_3, abs_error_1,
      abs_error_2, abs_error_3;
23
24 reg process_result;
25 reg [1:0] recommended_order;
26 reg [2:0] numevalsamples, recommended_rparam;
27 reg [15:0] sample_eval_1, sample_eval_2, sample_eval_3,
      sample_eval_4, last_eval_sample, blocksize;
28 reg [18:0] last_error1_sample, last_error2_sample;
29 reg [26:0] total_error_1, total_error_2, total_error_3,
      error_of_best_order;
30 reg [23:0] temp_blksize;
31 wire [31:0] result;
32
33 assign ord_1_lt_2 = total_error_1 <= total_error_2;
34 assign ord_1_lt_3 = total_error_1 <= total_error_3;
35 assign ord_2_lt_3 = total_error_2 <= total_error_3;
36 /*assign param_0 = blocksize >= error_of_best_order;
37 assign param_1 = {blocksize, 1'b0} >= error_of_best_order
      ;
38 assign param_2 = {blocksize, 2'b00} >=
      error_of_best_order;
39 assign param_3 = {blocksize, 3'b000} >=
      error_of_best_order;
40 assign param_4 = {blocksize, 4'b0000} >=
      error_of_best_order;
```

```
41 assign param_5 = {blocksize, 5'b00000} >=
      error_of_best_order;
42 assign param_6 = {blocksize, 6'b000000} >=
      error_of_best_order;
43 assign param_7 = {blocksize, 7'b0000000} >=
      error_of_best_order;*/
44 assign error_1 = sample_eval_1 - last_eval_sample;
45 assign error_2 = error_1 - last_error1_sample;
46 assign error_3 = error_2 - last_error2_sample;
47 assign abs_error_1 = error_1[18] ? ~error_1 : error_1;
48 assign abs_error_2 = error_2[18] ? ~error_2 : error_2;
49 assign abs_error_3 = error_3[18] ? ~error_3 : error_3;
50 assign eval_busy = (process_result || (numevalsamples !=
      0));
51 //assign reset = (ext_reset | (&operation));
52 assign result = {error_of_best_order, recommended_order,
      recommended_rparam};
53
54 always @(posedge clk or posedge ext_reset) begin
55    if (ext_reset)
56    begin
57       blocksize <= 16'd0;
58       sample_eval_1 <= 16'dX;
59       sample_eval_2 <= 16'dX;
60       sample_eval_3 <= 16'dX;
61       sample_eval_4 <= 16'dX;
62       last_eval_sample <= 16'dX;
63       last_error1_sample <= 19'dX;
64       last_error2_sample <= 19'dX;
65       total_error_1 <= 27'd0;
66       total_error_2 <= 27'd0;
67       total_error_3 <= 27'd0;
68       numevalsamples <= 3'd0;
69       //result <= 32'd0;
70    end
71    else if (clk_en)
72    begin
73       if (start)
74       begin
75          if (!eval_busy)
76          begin
77             if (operation == 3'b111)
78             begin
79                blocksize <= 16'd0;
80                sample_eval_1 <= 16'dX;
```

```
81                sample_eval_2 <= 16'dX;
82                sample_eval_3 <= 16'dX;
83                sample_eval_4 <= 16'dX;
84                last_eval_sample <= 16'dX;
85                last_error1_sample <= 19'dX;
86                last_error2_sample <= 19'dX;
87                total_error_1 <= 27'd0;
88                total_error_2 <= 27'd0;
89                total_error_3 <= 27'd0;
90                numevalsamples <= 3'd0;
91             end
92             else if (operation[2] == 1'b0) //load samples
93             begin
94                sample_eval_1 <= dataa[31:16];
95                sample_eval_2 <= dataa[15:0];
96                sample_eval_3 <= datab[31:16];
97                sample_eval_4 <= datab[15:0];
98                numevalsamples <= operation[1:0] + 1'b1;
99             //   result <= 32'd0;
100            end
101            else if (operation == 3'b100) //process
                  result
102            begin
103               process_result <= 1'b1;
104               recommended_rparam <= 3'd0;
105               temp_blksize <= {8'd0, blocksize};
106            end
107         end // !eval_busy
108      end // start
109      if (numevalsamples > 0)
110      begin
111         numevalsamples <= numevalsamples - 1;
112         if (blocksize >= 1)
113            total_error_1 <= total_error_1 + abs_error_1;
114         if (blocksize >= 2)
115            total_error_2 <= total_error_2 + abs_error_2;
116         if (blocksize >= 3)
117            total_error_3 <= total_error_3 + abs_error_3;
118         last_eval_sample <= sample_eval_1;
119         sample_eval_1 <= sample_eval_2;
120         sample_eval_2 <= sample_eval_3;
121         sample_eval_3 <= sample_eval_4;
122         sample_eval_4 <= 16'dX;
123         last_error1_sample <= error_1;
124         last_error2_sample <= error_2;
```

```
125            blocksize <= blocksize + 1;
126         end
127         else if (process_result == 1'b1)
128         begin
129            if (temp_blksize >= error_of_best_order)
130            begin
131               process_result <= 1'b0;
132            end
133            else
134            begin
135               temp_blksize <= {temp_blksize[22:0], 1'b0};
136               recommended_rparam <= recommended_rparam + 1'
                     b1;
137            end
138         end  //(process_result == 1'b1)
139
140      end
141   end
142
143
144
145   always
146   begin
147      if ((ord_1_lt_2 == 1'b1) && (ord_1_lt_3 == 1'b1))
148      begin
149         recommended_order <= 2'd1;
150         error_of_best_order <= total_error_1;
151      end
152      else if ((ord_2_lt_3 == 1'b1) && (ord_1_lt_2 == 1'b0))
153      begin
154         recommended_order <= 2'd2;
155         error_of_best_order <= total_error_2;
156      end
157      else if ((ord_2_lt_3 == 1'b0) && (ord_1_lt_3 == 1'b0))
158      begin
159         recommended_order <= 2'd3;
160         error_of_best_order <= total_error_3;
161      end
162      else
163      begin
164         recommended_order <= 2'bXX;
165         error_of_best_order <= 27'hXXXXXXX;
166      end
167   /*    if (param_0 == 1'b1)
168         recommended_rparam <= 3'd0;
```

```
169     else if (param_1 == 1'b1)
170        recommended_rparam <= 3'd1;
171     else if (param_2 == 1'b1)
172        recommended_rparam <= 3'd2;
173     else if (param_3 == 1'b1)
174        recommended_rparam <= 3'd3;
175     else if (param_4 == 1'b1)
176        recommended_rparam <= 3'd4;
177     else if (param_5 == 1'b1)
178        recommended_rparam <= 3'd5;
179     else if (param_6 == 1'b1)
180        recommended_rparam <= 3'd6;
181     else
182        recommended_rparam <= 3'd7;
183     if (eval_busy)
184     begin
185        result <= 32'd1;
186     end
187     else if (operation == 3'b101)
188     begin
189        result <= {error_of_best_order, recommended_order,
              recommended_rparam};
190     end
191     else
192     begin
193        result <= 32'dX;
194     end*/
195
196  end
197
198  endmodule
```

A.3 Rice Coder (rice_coder.v)

```
1  module rice_coder(clk,
2  value_in,       // input number (19-bit 2s complement)
3  valuein_avail, // line indicating inpuit value ready to
      be loaded
4  output_full,    // line indicating output available
5  rice_param,     // rice parameter
6  rice_clear,     // clear used when loading params
7  clk_en,         // main clock enable from CPU
8  reset,          // main reset from CPU
9  busy,           // line indicating work being done
10 reading_output,// asserted when output is being read
```

```
11 bits_shiftin,
12 value_out      // output 32-bit word
13 );
14
15
16 input clk, clk_en, valuein_avail, reset, reading_output,
     rice_clear;
17 input [2:0] rice_param;
18 input [18:0] value_in;
19
20 output busy, output_full;
21 output [4:0] bits_shiftin;
22 output [31:0] value_out;
23
24 // Actual registers
25 reg [31:0] value_out;
26 reg [18:0] value, upper_bits;
27 reg [7:0] lower_bits;
28 reg [4:0] bits_shiftin;
29 reg [1:0] param_temp, stage;
30 reg input_sign, /*output_read,*/ output_full, work_to_do;
31
32 // Combinational
33 reg [18:0] abs_val;
34 reg next_bit;
35
36 wire busy;
37
38 assign busy = (work_to_do && ~output_full);
39
40 // Synchronous events
41
42 always @(posedge clk or posedge reset)
43 begin
44    if (reset) begin
45       value_out <= 32'd0;
46       upper_bits <= 19'd0;
47       lower_bits <= 8'd0;
48       value <= 19'd0;
49       bits_shiftin <= 5'd0;
50       param_temp <= 2'd0;
51       stage <= 2'd0;
52       //output_read <= 1'b1;
53       output_full <= 1'b0;
54       work_to_do <= 1'b0;
```

```
55          input_sign <= 1'b0;
56       end //if (reset)
57
58       else if (clk_en) begin
59
60          if (busy) begin
61
62             if (stage[1]) begin // nothing is shifted to the
                   output in stage 1 or 2
63                value_out <= {value_out[30:0], next_bit};
64                bits_shiftin <= bits_shiftin + 1;
65                //output_read <= 1'b0;
66                output_full <= &bits_shiftin;
67             end // if (stage != 2'd1)
68
69
70
71             // Stage-specific synchronous events
72             case(stage)
73                2'd0: begin
74                   param_temp <= rice_param;
75                   upper_bits <= abs_val; // not val_1 since
                        it may be negative
76                   if (rice_param == 3'd0) // rice param = 0,
                         no need for stage 2
77                      stage <= stage + 2;
78                   else
79                      stage <= stage + 1;
80                end // stage == 2'd0
81                2'd1: begin
82                   param_temp <= param_temp - 1;
83                   upper_bits <= upper_bits >> 1;
84                   lower_bits <= {upper_bits[0], lower_bits
                        [7:1]};
85                   if (param_temp == 3'd1)
86                      stage <= stage + 1;
87                end // stage == 2'd1
88                2'd2: begin
89                   upper_bits <= upper_bits - 1;
90                   if (~|upper_bits) begin // upper_bits ==
                        all zeros, meaning in last cycle of
                        stage
91                      param_temp <= rice_param;
92                      if (rice_param == 3'd0) begin // rice
                           param = 0, no need for stage 4
```

```
93                      stage <= stage + 2;
94                      work_to_do <= 1'b0;
95                   end // rice_param == 3'd0
96                   else
97                      stage <= stage + 1;
98                end // ~|upper_bits
99             end // stage == 2'd2
100            2'd3: begin
101               lower_bits <= lower_bits << 1;
102               param_temp <= param_temp - 1;
103               if (param_temp == 3'd1) begin
104                  stage <= stage + 1;
105                  work_to_do <= 1'b0;
106               end // param_temp == 3'd1
107            end // stage == 2'd2
108         endcase
109      end // busy
110
111      if (rice_clear) begin
112         value_out <= 32'd0;
113         upper_bits <= 19'd0;
114         lower_bits <= 8'd0;
115         value <= 19'd0;
116         bits_shiftin <= 5'd0;
117         param_temp <= 2'd0;
118         stage <= 2'd0;
119         //output_read <= 1'b1;
120         output_full <= 1'b0;
121         work_to_do <= 1'b0;
122         input_sign <= 1'b0;
123      end // rice_clear
124
125      // Load new value into temp registers
126      else if (valuein_avail) begin // also ~busy
127         value <= value_in;
128         work_to_do <= 1'b1;
129      end // valuein_avail && ~busy
130
131      if (reading_output) begin
132         //output_read <= 1'b1;
133         output_full <= 1'b0;
134         value_out <= 32'd0;
135      end // reading_output
136   end // clk_en
137
```

```
138
139
140 end
141
142
143 // Asynchronous stuff
144 always
145 begin
146
147    //---------------------------------------
148    //mux connected to shift register input
149    case(stage)
150    2'b00: //stage 1
151       next_bit <= 1'bX; //value[18];
152    // No shifting in stage 2
153    2'b01:
154       next_bit <= 1'bX;
155    2'b10: //stage 3
156       next_bit <= ~|upper_bits;
157    2'b11: //stage 4
158       next_bit <= lower_bits[7]; //assuming the rice
            param register is 8 bits
159    endcase
160
161    //---------------------------------------
162    //Conditional negation of number
163
164    if(value[18])
165       abs_val <= {~value[17:0], 1'b1}; // + 1'b1;
166    else
167       abs_val <= {value[17:0], 1'b0};
168    /* By just negating, We avoid wasting the "minus zero"
        code.
169    This improves the coding efficiency by a few percent.
       */
170 end
171
172 endmodule
```

Appendix B

C Code

B.1 CompactFlash Host Driver (cfdrv.c)

```
1  #include <stdio.h>
2  #include "nios.h"
3  #include "cf_host_defines.h"
4  #include "formats.h"
5
6  // These apparently need to be ints to work properly
7  // Volatile keyword bypasses the data cache
8
9  volatile int32 * cf_errp = (int32 *) CF_HOST_ERROR_REG;
10 volatile int32 * cf_datap = (int32 *) CF_HOST_DATA_REG;
11 volatile int32 * cf_statusp = (int32 *)
      CF_HOST_STATUS_REG;
12 volatile int32 * cf_cmdp = (int32 *) CF_HOST_COMMAND_REG;
13 volatile int32 * cf_lba_27_24p = (int32 *)
      CF_HOST_LBA_27_24_REG;
14 volatile int32 * cf_lba_23_16p = (int32 *)
      CF_HOST_LBA_23_16_REG;
15 volatile int32 * cf_lba_15_8p = (int32 *)
      CF_HOST_LBA_15_8_REG;
16 volatile int32 * cf_lba_7_0p = (int32 *)
      CF_HOST_LBA_7_0_REG;
17 volatile int32 * cf_sec_ctp = (int32 *)
      CF_HOST_SECTOR_COUNT_REG;
18 volatile int32 * cf_ctlp = (int32 *)
      CF_CONTROL_STATUS_REG;
19
20
21
22 boolean cf__check_cf_present(void)
23 {
```

```
24     if (*cf_ctlp & CF_CTL_CD_N)
25        return 0;
26     else
27        return 1;
28  }
29
30
31  void cf__wait_ready(void)
32  {
33     int16 temp_status;
34     temp_status = (int16) *cf_statusp;
35     while ((temp_status & 0xF000) != 0x5000)
36     {
37        #ifdef CFDRV_DEBUG
38        printf("Wait_ready: temp_status = 0x%04X\n", (
             temp_status & 0x0000FFFF));
39        #endif
40        temp_status = (int16) *cf_statusp;
41     }
42  }
43
44
45
46  void cf__wait_DRQ(void)
47  {
48     int16 temp_status;
49     temp_status = (int16) *cf_statusp;
50     while ((temp_status & 0xF800) != 0x5800)
51     {
52        #ifdef CFDRV_DEBUG
53        printf("Wait_DRQ: temp_status = 0x%04X\n", (
             temp_status & 0x0000FFFF));
54        #endif
55        temp_status = (int16) *cf_statusp;
56        if ((temp_status & 0x0100) == 0x0100)
57        {
58           printf("Wait_DRQ:  Error occurred: error
                register = 0x%04X\n", *cf_errp);
59           exit(1);
60        }
61     }
62  }
63
64  void cf__read_256(int16 *dp)
65  {
```

```
66      int16 word_count = 0x0000;
67      while (word_count < 0x0100)
68      {
69         *dp = (int16) *cf_datap;
70         word_count++;
71         dp++;
72      }
73   }
74
75   void cf__write_256(int16 *dp)
76   {
77      int16 word_count = 0x0000;
78      while (word_count < 0x0100)
79      {
80         *cf_datap = (int32) *dp;
81         word_count++;
82         dp++;
83      }
84   }
85
86
87
88
89   void cf__check_status(void)
90   {
91      int16 temp_status;
92      temp_status = (int16) *cf_statusp;
93      printf("Check_Status: temp status = 0x%04X\n",
           temp_status);
94   }
95
96
97
98   void cf__read_sectors(byte *dp, uint32 sector_start,
        uint8 sector_count)
99   {
100     int16 lba_27_24 = ((0xE0000000 | (sector_start & 0
           x0F000000)) >> 24);
101     int16 lba_23_16 = ((sector_start & 0x00FF0000) >> 16);
102     int16 lba_15_8 = ((sector_start & 0x0000FF00) >> 8);
103     int16 lba_7_0 = (sector_start & 0x000000FF);
104     int16 i = 0;
105     int16 *sdp = (int16 *) dp;
106
107
```

```
108     cf__wait_ready();
109     *cf_sec_ctp = (int32) ((lba_7_0 << 8) | sector_count);
110     *cf_lba_15_8p = (int32) ((lba_23_16 << 8) | lba_15_8);
111     *cf_lba_27_24p = (int32) ((CF_COMMAND_READ_SECTORS <<
           8) | lba_27_24);
112
113     while (i < sector_count)
114     {
115        cf__wait_DRQ();
116        cf__read_256(sdp);
117        i++;
118        sdp += 256;
119     }
120  }
121
122
123
124  void cf__write_sectors(byte *dp, uint32 sector_start,
        uint8 sector_count)
125  {
126     int16 lba_27_24 = ((0xE0000000 | (sector_start & 0
           x0F000000)) >> 24);
127     int16 lba_23_16 = ((sector_start & 0x00FF0000) >> 16);
128     int16 lba_15_8 = ((sector_start & 0x0000FF00) >> 8);
129     int16 lba_7_0 = (sector_start & 0x000000FF);
130     int16 i = 0;
131     int16 *sdp = (int16 *) dp;
132
133     cf__wait_ready();
134
135     *cf_sec_ctp = (int32) ((lba_7_0 << 8) | sector_count);
136     *cf_lba_15_8p = (int32) ((lba_23_16 << 8) | lba_15_8);
137     *cf_lba_27_24p = (int32) ((CF_COMMAND_WRITE_SECTORS <<
            8) | lba_27_24);
138
139     while (i < sector_count)
140     {
141        cf__wait_DRQ();
142        cf__write_256(sdp);
143        i++;
144        sdp += 256;
145     }
146  }
147
148
```

```
149
150 boolean cf__write_verify_sectors(byte *dp, uint32
       sector_start, uint8 sector_count)
151 {
152     int cmp_result;
153     byte *tdp;
154     tdp = (byte *) malloc(512 * sector_count);
155
156     cf__wait_ready();
157
158     cf__write_sectors(dp, sector_start, sector_count);
159
160     cf__wait_ready();
161
162     cf__read_sectors(tdp, sector_start, sector_count);
163
164     cmp_result = memcmp(dp, tdp, 512 * sector_count);
165
166     if (cmp_result == 0)
167         return 0;
168     else
169         return 1;
170 }
171
172
173
174
175 void cf__erase_sectors(int32 sector_start, int8
       sector_count)
176 {
177     int16 lba_27_24 = ((0xE0000000 | (sector_start & 0
           x0F000000)) >> 24);
178     int16 lba_23_16 = ((sector_start & 0x00FF0000) >> 16);
179     int16 lba_15_8 = ((sector_start & 0x0000FF00) >> 8);
180     int16 lba_7_0 = (sector_start & 0x000000FF);
181
182     cf__wait_ready();
183
184     *cf_sec_ctp = (int32) ((lba_7_0 << 8) | sector_count);
185     *cf_lba_15_8p = (int32) ((lba_23_16 << 8) | lba_15_8);
186     *cf_lba_27_24p = (int32) ((CF_COMMAND_ERASE_SECTORS <<
            8) | lba_27_24);
187
188
189 }
```

```
190
191
192
193
194 cf__cardinfo *cf__get_cardinfo(void)
195 {
196    int16 *dp = (int16 *) malloc(512);
197    cf__cardinfo *cfinfo = (cf__cardinfo *) malloc(sizeof(
         cf__cardinfo));
198    *cf_cmdp = (CF_COMMAND_IDENTIFY_DEVICE << 8);
199    cf__wait_DRQ();
200    cf__read_256(dp);
201    cfinfo->cylinders = *(dp + 54);
202    cfinfo->heads = *(dp + 55);
203    cfinfo->sectors_per_track = *(dp + 56);
204    cfinfo->sectors_per_card = (*(dp + 7) << 16) + *(dp +
         8);
205    cfinfo->max_multisect = *(dp + 47);
206    cfinfo->sectors_total = (*(dp + 58) << 16) + *(dp +
         57);
207    cfinfo->sectors_LBA = (*(dp + 61) << 16) + *(dp + 60);
208    free(dp);
209    return cfinfo;
210 }
```

B.2 FAT Filesystem Driver (fat.c)

```
1  /*********************************************
2  fat.c
3  Routines for accessing FAT12/16 filesystems
4  *********************************************/
5
6  #include "fat.h"
7  #include "cf_host_defines.h"
8  #include <stdio.h>
9  #include "partition.h"
10
11 uint32 fat__cluster_to_sector(uint32 cluster, fat__ptinfo
       *fsinfo)
12 {
13     uint32 sectornum;
14     /* Starting cluster of the FS seems to be 2 for some
          reason
15        Possibly because of the . and .. entries? */
```

```
16      sectornum = ((cluster - 2) * fsinfo->
           sectors_per_cluster) + fsinfo->data_location_sector
           ;
17      if (sectornum > (fsinfo->sector_start + fsinfo->
           total_sectors_low))
18      {
19          return 0;
20      }
21      else
22      {
23          return sectornum;
24      }
25  }
26
27  void fat__print_fatinfo(fat__ptinfo *fsinfo)
28  {
29      printf("
           ----------------------------------------------\n")
           ;
30      printf("FAT Filesystem info:\n");
31      printf("Starting sector: %d\n", fsinfo->sector_start)
           ;
32      printf("%d bytes per sector\n", fsinfo->
           bytes_per_sector);
33      printf("%d sectors per cluster\n", fsinfo->
           sectors_per_cluster);
34      printf("%d reserved sectors\n", fsinfo->
           reserved_sectors);
35      printf("%d file allocation tables\n", fsinfo->
           FAT_count);
36      printf("%d root directory entries\n", fsinfo->
           root_dir_entries);
37      printf("Sectors low: 0x%08X\n", fsinfo->
           total_sectors_low);
38      printf("Sectors high: 0x%08X\n", fsinfo->
           total_sectors_high);
39      printf("%d sectors per FAT\n", fsinfo->
           sectors_per_FAT);
40      printf("%d hidden sectors\n", fsinfo->hidden_sectors)
           ;
41      printf("filesystem type: FAT%d\n", fsinfo->fstype);
42      printf("FAT starting sector: %d\n", fsinfo->
           FAT_location_sector);
43      printf("Root directory starting sector: %d\n", fsinfo
           ->rootdir_location_sector);
```

```
44      printf("Data starting sector: %d\n", fsinfo->
           data_location_sector);
45      printf("Data start byte offset: %d\n", fsinfo->
           data_location_byte_offset);
46      printf("
           -----------------------------------------------\n")
           ;
47  }
48
49  void fat__print_fileobj(fat__fileobj *file)
50  {
51      printf("
           -----------------------------------------------\n")
           ;
52      printf("File Object info:\n");
53      printf("Filename = %s\n", file->stats->filename);
54      printf("Extension = %s\n", file->stats->extension);
55      printf("Attributes = 0x%02X\n", file->stats->attr);
56      printf("Creation Time = 0x%08X\n", file->stats->
           creation_time);
57      printf("Last Access Time = 0x%04X\n", file->stats->
           last_access_time);
58      printf("Last Modified Time = 0x%08X\n", file->stats->
           last_modified_time);
59      printf("First Cluster = %d\n", file->stats->
           first_cluster);
60      printf("File Size = %d bytes\n", file->stats->
           file_size);
61      printf("Current Cluster: %d\n", file->current_cluster
           );
62      printf("Next Cluster: %d\n", file->next_cluster);
63      printf("Current byte offset: %d\n", file->byte_offset
           );
64      printf("
           -----------------------------------------------\n")
           ;
65  }
66
67  void fat__print_fat_raw(fat__ptinfo *fsinfo)
68  {
69      int32 *ifatp;
70      uint16 i, j, k;
71      ifatp = (int32 *) fsinfo->fatp;
72      k = 0;
```

```
73      printf("
           ------------------------------------------------\n")
           ;
74      printf("FAT Content:\n");
75      for (i = 0; i < fsinfo->sectors_per_FAT; i++)
76      {
77          for (j = 0; j < 128; j++)
78          {
79              printf("word[%d] = 0x%08X\n", k, *ifatp);
80              ifatp++;
81              k++;
82          }
83      }
84      printf("
           ------------------------------------------------\n")
           ;
85  }
86
87  void fat__print_rootdir(fat__ptinfo *fsinfo)
88  {
89      int32 *irdp;
90      uint16 i, j;
91      irdp = (int32 *) fsinfo->rdp;
92      printf("
           ------------------------------------------------\n")
           ;
93      printf("Root Directory Content:\n");
94      for (i = 0; i < fsinfo->root_dir_entries; i++)
95      {
96          for (j = 0; j < 8; j++)
97          {
98              printf("Entry %d, word %d = 0x%08X\n", i, j,
                   *irdp);
99              irdp++;
100         }
101     }
102     printf("
           ------------------------------------------------\n")
           ;
103 }
104
105 void fat__print_rdentry(fat__filestats *stats)
106 {
107     printf("
           ------------------------------------------------\n")
```

```
        ;
108     printf("File Stats info:\n");
109     printf("Filename = %s", stats->filename);
110     if ((stats->filename[0] == 0x05) || (stats->filename
           [0] == 0xE5))
111         printf(" (DELETED)");
112     printf("\n");
113     printf("Extension = %s\n", stats->extension);
114     printf("Attributes = 0x%02X\n", stats->attr);
115     printf("Creation Time = 0x%08X\n", stats->
           creation_time);
116     printf("Last Access Time = 0x%04X\n", stats->
           last_access_time);
117     printf("Last Modified Time = 0x%08X\n", stats->
           last_modified_time);
118     printf("First Cluster = %d\n", stats->first_cluster);
119     printf("File Size = %d bytes\n", stats->file_size);
120     printf("
           ------------------------------------------------\n")
           ;
121 }
122
123 int32 fat__parse_bootsector(byte *dp, fat__ptinfo *fsinfo
    )
124 {
125     if ((*(dp + 54) != 0x46) || (*(dp + 55) != 0x41) ||
           (*(dp + 56) != 0x54))
126     {
127         printf("assertion failed: invalid FS type\n");
128         return 1;
129     }
130     if ((*(dp + 57) == 0x31) && (*(dp + 58) == 0x0032))
131     {
132         fsinfo->fstype = 12;
133     }
134     else if ((*(dp + 57) == 0x31) && (*(dp + 58) == 0
           x0036))
135     {
136         fsinfo->fstype = 16;
137     }
138     else if ((*(dp + 57) == 0x33) && (*(dp + 58) == 0
           x0032))
139     {
140         printf("Error: FAT32 not supported by this driver
               \n");
```

```
141             return 2;
142         }
143         else
144         {
145             printf("assertion failed: invalid FS type\n");
146             return 1;
147         }
148         fsinfo->bytes_per_sector = (*(dp + 11) | (*(dp + 12)
               << 8));
149         fsinfo->sectors_per_cluster = *(dp + 13);
150         fsinfo->reserved_sectors = (*(dp + 14) | (*(dp + 15)
               << 8));
151         fsinfo->FAT_count = *(dp + 16);
152         fsinfo->root_dir_entries = (*(dp + 17) | (*(dp + 18)
               << 8));
153         fsinfo->total_sectors_low = ((*(dp + 19) & 0x000000FF
               ) | ((*(dp + 20) << 8) & 0x0000FF00) | \
154     ((*(dp + 32) << 16) & 0x00FF0000) | ((*(dp + 33) << 24) &
          0xFF000000));
155         fsinfo->sectors_per_FAT = (*(dp + 22) | (*(dp + 23)
               << 8));
156         fsinfo->hidden_sectors = (*(dp + 28) | (*(dp + 29) <<
                8) | (*(dp + 30) << 16) | (*(dp + 31) << 24));
157         fsinfo->total_sectors_high = ((*(dp + 34) & 0
               x000000FF) | ((*(dp + 35) << 8) & 0x0000FF00));
158         return 0;
159     }
160
161     int32 fat__read_fat_fsinfo(uint32 sector_start, byte *
           sector, uint32 size_sectors, fat__ptinfo *fsinfo)
162     {
163         int32 parse_result;
164         #ifdef FAT_DEBUG
165         printf("Reading FAT Filesystem info, sector %d\n",
               sector_start);
166         #endif
167         parse_result = fat__parse_bootsector(sector, fsinfo);
168         if (parse_result != 0) // error
169         {
170             return parse_result;
171         }
172         else
173         {
174             fsinfo->sector_start = sector_start;
```

```
175         fsinfo->FAT_location_sector = sector_start +
               fsinfo->reserved_sectors;
176         fsinfo->rootdir_location_sector = fsinfo->
               FAT_location_sector + (fsinfo->FAT_count *
               fsinfo->sectors_per_FAT);
177         fsinfo->data_location_sector = fsinfo->
               rootdir_location_sector + ((fsinfo->
               root_dir_entries * 32) >> 9);
178         fsinfo->data_location_byte_offset = ((fsinfo->
               root_dir_entries * 32) & 0x1FF);
179     }
180     return 0;
181 }
182
183 byte *fat__read_root_dir(fat__ptinfo *fsinfo)
184 {
185     byte *dp;
186     uint8 rd_sector_count = (fsinfo->root_dir_entries >>
           4);
187
188     dp = (byte *) malloc(rd_sector_count << 9);
189     cf__read_sectors(dp, fsinfo->rootdir_location_sector,
             rd_sector_count);
190     return dp;
191 }
192
193 void fat__read_fat_entry(byte *rdp, uint16 entrynum,
       fat__filestats *stats)
194 {
195     int32 i;
196     uint16 start_byte = entrynum << 5;
197     for (i = 0; i < 8; i++)
198         stats->filename[i] = *(rdp + start_byte + i);
199     stats->filename[8] = 0; //null termination
200     for (i = 0; i < 3; i++)
201         stats->extension[i] = *(rdp + start_byte + 8 + i)
               ;
202     stats->extension[3] = 0;
203     stats->attr = (uint8) *(rdp + start_byte + 11);
204     stats->fine_res_timestamp = (uint8) *(rdp +
           start_byte + 13);
205     stats->creation_time = 0;
206     for (i = 0; i < 4; i++)
207         stats->creation_time |= ((((int32) *(rdp +
               start_byte + 14 + i)) & 0x000000FF) << (8 * i))
```

```
            ;
208     stats->last_access_time = *((uint16 *) (rdp +
           start_byte + 18));
209     stats->last_modified_time = 0;
210     for (i = 0; i < 4; i++)
211         stats->last_modified_time |= ((((int32) *(rdp +
               start_byte + 22 + i)) & 0x000000FF) << (8 * i))
               ;
212     stats->first_cluster = *((uint16 *) (rdp + start_byte
            + 26));
213     stats->file_size = *((uint32 *) (rdp + start_byte +
           28));
214     /*#ifdef FAT_DEBUG
215     print_buffer(rdp + start_byte, 32);
216     #endif*/
217 }
218
219 void fat__print_rootdir_entries(fat__ptinfo *fsinfo)
220 {
221     fat__filestats *stats;
222     uint16 i = 0;
223     stats = (fat__filestats *) malloc(sizeof(
           fat__filestats));
224
225     do
226     {
227         fat__read_fat_entry(fsinfo->rdp, i, stats);
228         if ((stats->filename[0] != 0x00) && (stats->
               filename[0] != 0x05) && (stats->filename[0] !=
               0xE5))
229         {
230             fat__print_rdentry(stats);
231         }
232         i++;
233     } while (stats->filename[0] != 0x00);
234     free(stats);
235 }
236
237 void fat__print_all_rootdir_entries(fat__ptinfo *fsinfo)
238 {
239     fat__filestats *stats;
240     uint16 i = 0;
241     stats = (fat__filestats *) malloc(sizeof(
           fat__filestats));
242
```

```
243     do
244     {
245         fat__read_fat_entry(fsinfo->rdp, i, stats);
246         if (stats->filename[0] != 0x00)
247         {
248             fat__print_rdentry(stats);
249         }
250         i++;
251     } while (stats->filename[0] != 0x00);
252     free(stats);
253 }
254
255 void fat__read_occupied_fat_entry(byte *rdp, uint16
       entrynum, fat__filestats *stats)
256 {
257     int32 i = 0;
258     int32 j = 0;
259     do
260     {
261         #ifdef FAT_DEBUG
262         printf("read_occupied_fat_entry: Reading entry
               num %d\n", j);
263         #endif
264         fat__read_fat_entry(rdp, j, stats);
265         if ((stats->filename[0] != 0x05) && (stats->
               filename[0] != 0xE5) && (stats->filename[0] !=
               0x00))
266         {
267             #ifdef FAT_DEBUG
268             printf("occupied entry found at %d\n", i);
269             #endif
270             i++;
271         }
272         j++;
273     } while (i <= entrynum);
274 }
275
276 boolean fat__check_filename_exists(byte *filename, byte *
       extension, fat__ptinfo *fsinfo)
277 {
278     fat__filestats *stats;
279     uint16 i;
280     int32 j, filecount;
281     boolean found;
```

```
282     stats = (fat__filestats *) malloc(sizeof(
           fat__filestats));
283     filecount = fat__get_file_count(fsinfo->rdp);
284     found = 0;
285     for (i = 0; i < filecount; i++)
286     {
287         fat__read_occupied_fat_entry(fsinfo->rdp, i,
               stats);
288         found = 1;
289         for (j = 0; j < 8; j++)
290         {
291             if (stats->filename[i] != *(filename + i))
292                 found = 0;
293         }
294         for (j = 0; j < 3; j++)
295         {
296             if (stats->extension[i] != *(extension + i))
297                 found = 0;
298         }
299         if (found == 1)
300         {
301             free(stats);
302             return found;
303         }
304     }
305     free(stats);
306     return found;
307 }
308
309 uint16 fat__get_first_unoccupied_entry(byte *rdp)
310 {
311     uint16 i = 0;
312     fat__filestats stats;
313     while (1)
314     {
315         fat__read_fat_entry(rdp, i, &stats);
316         if ((stats.filename[0] == 0x05) || (stats.
               filename[0] == 0xE5) || (stats.filename[0] == 0
               x00))
317              return i;
318         i++;
319     }
320 }
321
322 int32 fat__get_file_count(byte *rdp)
```

```
323 {
324     fat__filestats stats;
325     int32 i, entry;
326     entry = 0;
327     i = 0;
328     do
329     {
330         fat__read_fat_entry(rdp, entry, &stats);
331         if ((stats.filename[0] != 0x00) && (stats.
               filename[0] != 0x05) && (stats.filename[0] != 0
               xE5) && \
332 (stats.filename[0] != 0x2E))
333             i++;
334         entry++;
335     } while (stats.filename[0] != 0x00);
336     #ifdef FAT_DEBUG
337     printf("File count = %d\n", i);
338     #endif
339     return i;
340 }
341 
342 int32 fat__get_used_rd_entry_count(byte *rdp)
343 {
344     fat__filestats stats;
345     int32 i, entry;
346     entry = 0;
347     i = 0;
348     do
349     {
350         fat__read_fat_entry(rdp, entry, &stats);
351         if (stats.filename[0] != 0x00)
352             i++;
353         entry++;
354     } while (stats.filename[0] != 0x00);
355     #ifdef FAT_DEBUG
356     printf("File count = %d\n", i);
357     #endif
358     return i;
359 }
360 
361 void fat__print_rootdir_raw(fat__ptinfo *fsinfo)
362 {
363     int32 *irdp;
364     uint16 i, j;
365     int32 rd_entry_count;
```

```
366     irdp = (int32 *) fsinfo->rdp;
367     rd_entry_count = fat__get_used_rd_entry_count(fsinfo
           ->rdp);
368     printf("
           ------------------------------------------------\n")
           ;
369     printf("Root Directory Content:\n");
370     for (i = 0; i < rd_entry_count; i++)
371     {
372         for (j = 0; j < 8; j++)
373         {
374             printf("Entry %d, word %d = 0x%08X\n", i, j,
                   *irdp);
375             irdp++;
376         }
377     }
378     printf("
           ------------------------------------------------\n")
           ;
379 }
380
381 /*int32 fat__compare_fat(byte *fat1, byte *fat2, uint16
       fat_size)
382 {
383     uint16 i;
384     boolean different = 0;
385     for (i = 0; i < fat_size; i++)
386     {
387         if (*(fat1 + i) != *(fat2 + i))
388         {
389             printf("Offset %d:  fat1 = 0x%02X, fat2 = 0x
                   %02X\n", i, (*(fat1 + i) & 0x000000FF), (*(
                   fat2 + i) & 0x000000FF));
390             different = 1;
391         }
392     }
393     return different
394 }*/
395
396 byte *fat__read_fat(fat__ptinfo *fsinfo)
397 {
398     byte *fatp;
399     byte *fat2p;
400     int32 cmp_result;
401     uint32 fat_size = fsinfo->sectors_per_FAT * 512;
```

```
402     fatp = (byte *) malloc(fat_size);
403     fat2p = (byte *) malloc(fat_size);
404
405
406     cf__read_sectors(fatp, fsinfo->FAT_location_sector,
            fsinfo->sectors_per_FAT);
407
408     cf__read_sectors(fat2p, fsinfo->FAT_location_sector +
             fsinfo->sectors_per_FAT, fsinfo->sectors_per_FAT);
409     cmp_result = memcmp(fatp, fat2p, fat_size);
410     //cmp_result = fat__compare_fat(fatp, fat2p, fat_size
            );
411     if (cmp_result != 0)
412     {
413         printf("Copies of the FAT are different\n\004");
414         exit(1);
415     }
416     //printf("Copies of the FAT are identical\n");
417     free(fat2p);
418     return fatp;
419 }
420
421 fat__ptinfo *fat__mount_fat_filesystem(byte *mbrp,
        pt__partitioninfo *ptinfo, uint8 partnum)
422 {
423     byte *dp;
424     fat__ptinfo *fat_fsinfo;
425     fat_fsinfo = (fat__ptinfo *) malloc(sizeof(
            fat__ptinfo));
426     dp = (byte *) malloc(512);
427
428     if ((ptinfo->fstype != 0x01) && (ptinfo->fstype != 0
            x06))
429         return NULL;
430
431     cf__read_sectors(dp, ptinfo->lba_sector_start, 1);
432     fat__read_fat_fsinfo(ptinfo->lba_sector_start, dp,
            ptinfo->lba_numsectors, fat_fsinfo);
433     fat_fsinfo->rdp = (byte *) fat__read_root_dir(
            fat_fsinfo);
434     fat_fsinfo->fatp = (byte *) fat__read_fat(fat_fsinfo)
            ;
435     #ifdef FAT_DEBUG
436     printf("Filesystem mounted.\n");
437     #endif
```

```
438     return fat_fsinfo;
439 }
440
441 uint32 fat__get_next_file_cluster_fat12(uint16 clusternum
       , byte *fatp)
442 {
443     uint16 bytenum, next_cluster;
444     bytenum = (clusternum + (clusternum >> 1));
445     if ((clusternum & 0x0001) != 0x0001)
446         next_cluster = (((uint16) *(fatp + bytenum)) & 0
               x00FF) | (((uint16) (*(fatp + bytenum + 1) & 0
               x0F)) << 8);
447     else
448         next_cluster = ((*(fatp + bytenum) & 0x00F0) >>
               4) | ((((uint16) *(fatp + bytenum + 1)) & 0
               x00FF) << 4);
449     return (((uint32) next_cluster) & 0x0000FFF);
450 }
451
452 uint32 fat__get_next_file_cluster_fat16(uint16 clusternum
       , byte *fatp)
453 {
454     uint16 *sfatp = (uint16 *) fatp;
455     return (((uint32) *(sfatp + clusternum)) & 0x0000FFFF
           );
456 }
457
458 uint32 fat__get_next_file_cluster(uint16 clusternum,
       fat__ptinfo *fat_fsinfo)
459 {
460     if (fat_fsinfo->fstype == 12)
461         return fat__get_next_file_cluster_fat12(
               clusternum, fat_fsinfo->fatp);
462     else if (fat_fsinfo->fstype == 16)
463         return fat__get_next_file_cluster_fat16(
               clusternum, fat_fsinfo->fatp);
464     else
465         return 0;
466 }
467
468 void fat__print_cluster_fat12(uint32 num)
469 {
470     if (num == 0x00000000)
471         printf("Free\n");
472     else if (num == 0x00000001)
```

```
473         printf("Reserved\n");
474     else if ((num >= 0x00000002) && (num <= 0x00000FEF))
475         printf("%d\n", num);
476     else if ((num >= 0x00000FF0) && (num <= 0x00000FF6))
477         printf("Reserved Value (0x%03X)\n", num);
478     else if (num == 0x00000FF7)
479         printf("Bad\n");
480     else if ((num >= 0x00000FF8) && (num <= 0x00000FFF))
481         printf("EOF (0x%03X)\n", num);
482     else
483         printf("Bad Value: 0x%03X\n", num);
484 }
485 
486 void fat__print_cluster_fat16(uint32 num)
487 {
488     if (num == 0x00000000)
489         printf("Free\n");
490     else if (num == 0x00000001)
491         printf("Reserved\n");
492     else if ((num >= 0x00000002) && (num <= 0x0000FFEF))
493         printf("%d\n", num);
494     else if ((num >= 0x0000FFF0) && (num <= 0x0000FFF6))
495         printf("Reserved Value (0x%04X)\n", num);
496     else if (num == 0x0000FFF7)
497         printf("Bad\n");
498     else if ((num >= 0x0000FFF8) && (num <= 0x0000FFFF))
499         printf("EOF (0x%04X)\n", num);
500     else
501         printf("Bad Value: 0x%04X\n", num);
502 }
503 
504 void fat__print_cluster(uint32 num, uint8 type)
505 {
506     if (type == 12)
507         fat__print_cluster_fat12(num);
508     else if (type == 16)
509         fat__print_cluster_fat16(num);
510 }
511 
512 void fat__print_fat(fat__ptinfo *fsinfo)
513 {
514     int32 entrynum;
515     uint32 result;
516     int32 numentries = (fsinfo->sectors_per_FAT << 12) /
        fsinfo->fstype;
```

```
517     for (entrynum = 0; entrynum < numentries; entrynum++)
518     {
519         result = fat__get_next_file_cluster(entrynum,
               fsinfo);
520         printf("Cluster %d: ", entrynum);
521         fat__print_cluster(result, fsinfo->fstype);
522     }
523 }
524
525
526
527
528 uint32 fat__get_next_avail_cluster_fat12(uint16
       clusternum, byte *fatp)
529 {
530     uint16 cluster = clusternum;
531     uint16 clusterdata, bytenum;
532     do
533     {
534         cluster++;
535         bytenum = (cluster + (cluster >> 1));
536         if ((cluster & 0x0001) != 0x0001)
537             clusterdata = *(fatp + bytenum) | (((uint16)
                   (*(fatp + bytenum + 1) & 0x0F)) << 8);
538         else
539             clusterdata = ((*(fatp + bytenum) & 0xF0) >>
                   4) | (((uint16) *(fatp + bytenum + 1)) <<
                   4);
540     } while (clusterdata != 0);
541     return (uint32) cluster;
542 }
543
544 uint32 fat__get_next_avail_cluster_fat16(uint16
       clusternum, byte *fatp)
545 {
546     uint16 *sfatp = (uint16 *) fatp;
547     uint16 cluster = clusternum;
548     do
549     {
550         cluster++;
551     } while (*(sfatp + cluster) != 0);
552     return (uint32) cluster;
553 }
554
```

```
555 uint32 fat__get_next_avail_cluster(uint16 clusternum,
       fat__ptinfo *fat_fsinfo)
556 {
557     if (fat_fsinfo->fstype == 12)
558         return fat__get_next_avail_cluster_fat12(
               clusternum, fat_fsinfo->fatp);
559     else if (fat_fsinfo->fstype == 16)
560         return fat__get_next_avail_cluster_fat16(
               clusternum, fat_fsinfo->fatp);
561     else
562         return 0;
563 }
564
565 fat__fileobj *fat__open_read(uint16 entrynum, fat__ptinfo
       *fat_fsinfo)
566 {
567     fat__fileobj *file;
568     fat__filestats *stats;
569     uint32 sector_num;
570     file = (fat__fileobj *) malloc(sizeof(fat__fileobj));
571     stats = (fat__filestats *) malloc(sizeof(
           fat__filestats));
572     file->stats = stats;
573     fat__read_occupied_fat_entry(fat_fsinfo->rdp,
           entrynum, file->stats);
574     file->current_cluster = file->stats->first_cluster;
575     file->next_cluster = fat__get_next_file_cluster(file
           ->current_cluster, fat_fsinfo);
576     file->byte_offset = 0;
577     file->fsinfo = fat_fsinfo;
578     file->cluster = (byte *) malloc(fat_fsinfo->
           sectors_per_cluster * fat_fsinfo->bytes_per_sector)
           ;
579     sector_num = fat__cluster_to_sector(file->
           current_cluster, file->fsinfo);
580     cf__read_sectors(file->cluster, sector_num, file->
           fsinfo->sectors_per_cluster);
581     #ifdef FAT_DEBUG
582     fat__print_fileobj(file);
583     #endif
584     file->type = 'r';
585     file->rd_entrynum = entrynum;
586     return file;
587 }
588
```

```
589 fat__fileobj *fat__open_write(fat__ptinfo *fsinfo,
       fat__filestats *stats)
590 {
591     fat__fileobj *file;
592     file = (fat__fileobj *) malloc(sizeof(fat__fileobj));
593     file->stats = stats;
594     file->stats->first_cluster =
           fat__get_next_avail_cluster(0, fsinfo);
595     file->stats->file_size = 0;
596     file->prev_cluster = 0;
597     file->current_cluster = file->stats->first_cluster;
598     file->byte_offset = 0;
599     file->fsinfo = fsinfo;
600     file->cluster = (byte *) malloc(fsinfo->
           sectors_per_cluster * fsinfo->bytes_per_sector);
601     file->type = 'w';
602     file->rd_entrynum = fat__get_first_unoccupied_entry(
           fsinfo->rdp);
603     #ifdef FAT_DEBUG
604     fat__print_fileobj(file);
605     #endif
606     return file;
607 }
608
609 int32 fat__read_next_cluster(fat__fileobj *file)
610 {
611     uint32 sector_num;
612     if (file->byte_offset >= file->stats->file_size)
613         return 1;
614     file->current_cluster = file->next_cluster;
615     file->next_cluster = fat__get_next_file_cluster(file
           ->current_cluster, file->fsinfo);
616     sector_num = fat__cluster_to_sector(file->
           current_cluster, file->fsinfo);
617     //cf__wait_ready();
618     cf__read_sectors(file->cluster, sector_num, file->
           fsinfo->sectors_per_cluster);
619     return 0;
620 }
621
622 int32 fat__read(fat__fileobj *file, uint32 bytes, byte *
       dp)
623 {
624     int32 bytes_read, bytes_left, a;
625     uint32 mask;
```

```
626
627     if (file->type == 'w')
628         return 0;
629     mask = (file->fsinfo->sectors_per_cluster * file->
           fsinfo->bytes_per_sector) - 1;
630     if ((file->byte_offset + bytes) > file->stats->
           file_size)
631         bytes_read = file->stats->file_size - file->
               byte_offset;
632     else
633         bytes_read = bytes;
634     bytes_left = bytes_read;
635     #ifdef FAT_DEBUG
636     printf("Reading %d bytes...\n", bytes_read);
637     #endif
638     while (bytes_left > 0)
639     {
640         if (((file->byte_offset & mask) + bytes_left) >
               mask)
641         {
642             memcpy(dp + (bytes_read - bytes_left), file->
                   cluster + (file->byte_offset & mask), (mask
                    + 1) - \
643 (file->byte_offset & mask));
644             bytes_left -= ((mask + 1) - (file->
                   byte_offset & mask));
645             file->byte_offset += ((mask + 1) - (file->
                   byte_offset & mask));
646             a = fat__read_next_cluster(file);
647             if (a != 0)
648                 return -1;
649         }
650         else
651         {
652             memcpy(dp + (bytes_read - bytes_left), file->
                   cluster + (file->byte_offset & mask),
                   bytes_left);
653             file->byte_offset += bytes_left;
654             bytes_left = 0;
655         }
656     }
657     return bytes_read;
658 }
659
660 void fat__write_fat(fat__ptinfo *fsinfo)
```

```
661 {
662     uint8 i;
663     #ifdef FAT_DEBUG
664     boolean write_result;
665     #endif
666     //There are multiple copies of the FAT for redundancy
        , make sure we write all of them
667     for (i = 0; i < fsinfo->FAT_count; i++)
668     {
669         #ifndef DEBUG
670         cf__write_sectors(fsinfo->fatp, fsinfo->
            FAT_location_sector + (i * fsinfo->
            sectors_per_FAT),\
671 fsinfo->sectors_per_FAT);
672         #else
673         write_result = cf__write_verify_sectors(fsinfo->
            fatp, fsinfo->FAT_location_sector + \
674 (i * fsinfo->sectors_per_FAT), fsinfo->sectors_per_FAT);
675         if (write_result == 0)
676             printf("FAT copy %d written successfully\n",
                i + 1);
677         else
678             printf("FAT copy %d write failed\n", i + 1);
679         #endif
680     }
681 }
682
683 void fat__write_rootdir(fat__fileobj *file)
684 {
685     byte *ep;
686     int8 i;
687     uint16 start_byte;
688     uint8 rd_sector_count;
689     #ifdef FAT_DEBUG
690     boolean write_result;
691     #endif
692     ep = (byte *) malloc(32);
693     for (i = 0; i < 8; i++)
694     {
695         *(ep + i) = file->stats->filename[i];
696     }
697     for (i = 0; i < 3; i++)
698     {
699         *(ep + 8 + i) = file->stats->extension[i];
700     }
```

```
701     *(ep + 11) = file->stats->attr;
702     *(ep + 13) = file->stats->fine_res_timestamp;
703     for (i = 0; i < 4; i++)
704         *(ep + 14 + i) = (byte) ((file->stats->
            creation_time >> (8 * i)) & 0x000000FF);
705     //*((uint32 *) (ep + 14)) = file->stats->
        creation_time;
706     *((uint16 *) (ep + 18)) = file->stats->
        last_access_time;
707     for (i = 0; i < 4; i++)
708         *(ep + 22 + i) = (byte) ((file->stats->
            last_modified_time >> (8 * i)) & 0x000000FF);
709     //*((uint32 *) (ep + 22)) = file->stats->
        last_modified_time;
710     *((uint16 *) (ep + 26)) = file->stats->first_cluster;
711     *((uint32 *) (ep + 28)) = file->stats->file_size;
712 /*    #ifdef FAT_DEBUG
713     print_buffer(ep, 32);
714     #endif*/
715     start_byte = file->rd_entrynum << 5;
716     memcpy(file->fsinfo->rdp + start_byte, ep, 32);
717     rd_sector_count = (file->fsinfo->root_dir_entries >>
        4);
718     #ifndef DEBUG
719     cf__write_sectors(file->fsinfo->rdp, file->fsinfo->
        rootdir_location_sector, rd_sector_count);
720     #else
721     write_result = cf__write_verify_sectors(file->fsinfo
        ->rdp, file->fsinfo->rootdir_location_sector,
        rd_sector_count);
722     if (write_result == 0)
723         printf("Root directory written successfully\n");
724     else
725         printf("Root directory write failed\n");
726     #endif
727     free(ep);
728 }
729
730
731
732 void fat__fat_update_fat12(byte *fatp, uint32 clusternum,
    uint32 next)
733 {
734     uint16 bytenum;
735     bytenum = (clusternum + (clusternum >> 1));
```

```
736     if (next == FAT_EOF)
737     {
738         if ((clusternum & 0x0001) != 0x0001)
739         {
740             #ifdef FAT_DEBUG
741             printf("Cluster number is even\n");
742             #endif
743             *(fatp + bytenum) = 0xFF;
744             *(fatp + bytenum + 1) = (*(fatp + bytenum +
                   1) & 0xF0) | 0x0F;
745         }
746         else
747         {
748             #ifdef FAT_DEBUG
749             printf("Cluster number is odd\n");
750             #endif
751             *(fatp + bytenum) = (*(fatp + bytenum) & 0x0F
                   ) | 0xF0;
752             *(fatp + bytenum + 1) = 0xFF;
753         }
754     }
755     else
756     {
757         if ((clusternum & 0x0001) != 0x0001)
758         {
759             #ifdef FAT_DEBUG
760             printf("Cluster number is even\n");
761             #endif
762             *(fatp + bytenum) = (byte) (next & 0x000000FF
                   );
763             *(fatp + bytenum + 1) = (*(fatp + bytenum +
                   1) & 0x000000F0) | ((next >> 8) & 0
                   x0000000F);
764         }
765         else
766         {
767             #ifdef FAT_DEBUG
768             printf("Cluster number is odd\n");
769             #endif
770             *(fatp + bytenum) = (*(fatp + bytenum) & 0
                   x0000000F) | ((next << 4) & 0x000000F0);
771             *(fatp + bytenum + 1) = (byte) ((next >> 4) &
                    0x000000FF);
772         }
773     }
```

```
774 }
775
776 void fat__fat_update_fat16(byte *fatp, uint32 clusternum,
       uint32 next)
777 {
778     uint16 *sfatp = (uint16 *) fatp;
779     if (next == FAT_EOF)
780         *(sfatp + clusternum) = 0xFFFF;
781     else
782         *(sfatp + clusternum) = (uint16) next;
783 }
784
785 void fat__fat_update(fat__ptinfo *fsinfo, uint32
      clusternum, uint32 next)
786 {
787     if (fsinfo->fstype == 12)
788         fat__fat_update_fat12(fsinfo->fatp, clusternum,
               next);
789     else if (fsinfo->fstype == 16)
790         fat__fat_update_fat16(fsinfo->fatp, clusternum,
               next);
791 }
792
793 void fat__flush_write(fat__fileobj *file, boolean
      write_meta)
794 {
795     #ifdef FAT_DEBUG
796     boolean write_result;
797     #endif
798     uint32 sector_num = fat__cluster_to_sector(file->
           current_cluster, file->fsinfo);
799     //write file data
800     #ifndef DEBUG
801     cf__write_sectors(file->cluster, sector_num, file->
           fsinfo->sectors_per_cluster);
802     #else
803     write_result = cf__write_verify_sectors(file->cluster
           , sector_num, file->fsinfo->sectors_per_cluster);
804     if (write_result == 0)
805         printf("Cluster written successfully\n");
806     else
807         printf("Cluster write failed\n");
808     #endif
809
810     if (file->prev_cluster != 0)
```

```
811         fat__fat_update(file->fsinfo, file->prev_cluster,
                file->current_cluster);
812     fat__fat_update(file->fsinfo, file->current_cluster,
            FAT_EOF);
813
814     if (write_meta == 1)
815     {
816         fat__write_fat(file->fsinfo);
817         fat__write_rootdir(file);
818     }
819 }
820
821
822 void fat__flush_write_test(fat__fileobj *file)
823 {
824     uint32 sector_num = fat__cluster_to_sector(file->
            current_cluster, file->fsinfo);
825     //write file data
826
827     if (file->prev_cluster != 0)
828     {
829         printf("Updating FAT cluster %d to point to %d\n"
                , file->prev_cluster, file->current_cluster);
830         fat__fat_update(file->fsinfo, file->prev_cluster,
                file->current_cluster);
831     }
832     printf("Updating FAT cluster %d to point to EOF\n",
            file->current_cluster);
833     fat__fat_update(file->fsinfo, file->current_cluster,
            FAT_EOF);
834
835 }
836
837 uint32 fat__write(fat__fileobj *file, uint32 bytes, byte
        *dp)
838 {
839     #ifndef NO_FAT_WRITE
840     int32 bytes_left, bytes_written;
841     uint32 mask;
842     uint32 written;
843
844     if (file->type == 'r')
845         return 0;
846
847     #ifdef FAT_DEBUG
```

```
848        printf("fat__write: Writing %d bytes to file\n",
              bytes);
849        #endif
850        mask = (file->fsinfo->sectors_per_cluster * file->
              fsinfo->bytes_per_sector) - 1;
851        bytes_left = bytes;
852        while (bytes_left > 0)
853        {
854            if (((file->byte_offset & mask) + bytes_left) >
                  mask) // Write spans multiple clusters
855            {
856                #ifdef FAT_DEBUG
857                printf("Write reaches end of cluster\n");
858                printf("Writing %d bytes\n", (mask + 1) - (
                      file->byte_offset & mask));
859                #endif
860                memcpy(file->cluster + (file->byte_offset &
                      mask), dp + (bytes - bytes_left), (mask +
                      1) - (file->byte_offset & mask));
861                written = ((mask + 1) - (file->byte_offset &
                      mask));
862                bytes_left -= written;
863                #ifdef FAT_DEBUG
864                printf("Old file size = %d\n", file->stats->
                      file_size);
865                #endif
866                file->byte_offset += written;
867                file->stats->file_size += written;
868                #ifdef FAT_DEBUG
869                printf("New file size = %d\n", file->stats->
                      file_size);
870                #endif
871                fat__flush_write(file, 0);
872                file->prev_cluster = file->current_cluster;
873                file->current_cluster =
                      fat__get_next_avail_cluster(file->
                      current_cluster, file->fsinfo);
874            }
875            else
876            {
877                #ifdef FAT_DEBUG
878                printf("Write within single cluster\n");
879                printf("Writing %d bytes\n", bytes_left);
880                #endif
```

```
881             memcpy(file->cluster + (file->byte_offset &
                   mask), dp + (bytes - bytes_left),
                   bytes_left);
882             file->byte_offset += bytes_left;
883             #ifdef FAT_DEBUG
884             printf("Old file size = %d\n", file->stats->
                   file_size);
885             #endif
886             file->stats->file_size += bytes_left;
887             #ifdef FAT_DEBUG
888             printf("New file size = %d\n", file->stats->
                   file_size);
889             #endif
890             bytes_left = 0;
891         }
892     }
893     #endif // NO_FAT_WRITE
894     return bytes;
895 }
896 
897 int32 fat__write_test(fat__fileobj *file, uint32 bytes,
       byte *dp)
898 {
899     int32 bytes_left, bytes_written;
900     uint32 mask;
901 
902     if (file->type == 'r')
903         return 0;
904 
905     mask = (file->fsinfo->sectors_per_cluster * file->
           fsinfo->bytes_per_sector) - 1;
906     bytes_left = bytes;
907     while (bytes_left > 0)
908     {
909         if (((file->byte_offset & mask) + bytes_left) >
               mask) // Write spans multiple clusters
910         {
911             memcpy(file->cluster + (file->byte_offset &
                   mask), dp + (bytes - bytes_left), (mask +
                   1) - (file->byte_offset & mask));
912             bytes_left -= ((mask + 1) - (file->
                   byte_offset & mask));
913             file->byte_offset += ((mask + 1) - (file->
                   byte_offset & mask));
```

```
914             file->stats->file_size += ((mask + 1) - (file
                   ->byte_offset & mask));
915             fat__flush_write_test(file);
916             file->prev_cluster = file->current_cluster;
917             file->current_cluster =
                   fat__get_next_avail_cluster(file->
                   current_cluster, file->fsinfo);
918         }
919         else
920         {
921             memcpy(file->cluster + (file->byte_offset &
                   mask), dp + (bytes - bytes_left),
                   bytes_left);
922             file->byte_offset += bytes_left;
923             file->stats->file_size += bytes_left;
924             bytes_left = 0;
925         }
926     }
927     return bytes;
928 }
929
930 uint32 fat__close(fat__fileobj *file)
931 {
932     #ifndef NO_FAT_WRITE
933     if (file->type == 'w')
934         fat__flush_write(file, 1);
935     #endif
936     free(file->cluster);
937     free(file->stats);
938     free(file);
939     return 0;
940 }
```

B.3 FLAC Library (flac.c)

```
1  /**********************************************
2  flac.c
3  Library of FLAC enoder & format functions
4  **********************************************/
5
6  #ifndef __SOFTWARE_CODER
7  #include "cinst_lpc.h"
8  #endif
9  #include "flac.h"
10 #include "bitbuffer.h"
```

```
11
12 void flac__append_utf8_num(bitbuffer *buffer, uint32 num)
13 {
14     if (num <= 0x7F)
15         bitbuffer__write_int(buffer, num, 8);
16     else if (num <= 0x7FF)
17     {
18         bitbuffer__write_int(buffer, (0x0C0 | (num >> 6))
              , 8);
19         bitbuffer__write_int(buffer, (0x080 | (num & 0x3F
              )), 8);
20     }
21     else if (num <= 0xFFFF)
22     {
23         bitbuffer__write_int(buffer, (0x0E0 | (num >> 12)
              ), 8);
24         bitbuffer__write_int(buffer, (0x080 | ((num >> 6)
               & 0x3F)), 8);
25         bitbuffer__write_int(buffer, (0x080 | (num & 0x3F
              )), 8);
26     }
27     else if (num <= 0x1FFFFF)
28     {
29         bitbuffer__write_int(buffer, (0x0F0 | (num >> 18)
              ), 8);
30         bitbuffer__write_int(buffer, (0x080 | ((num >>
              12) & 0x3F)), 8);
31         bitbuffer__write_int(buffer, (0x080 | ((num >> 6)
               & 0x3F)), 8);
32         bitbuffer__write_int(buffer, (0x080 | (num & 0x3F
              )), 8);
33     }
34     else if (num <= 0x3FFFFFF)
35     {
36         bitbuffer__write_int(buffer, (0x0F8 | (num >> 24)
              ), 8);
37         bitbuffer__write_int(buffer, (0x080 | ((num >>
              18) & 0x3F)), 8);
38         bitbuffer__write_int(buffer, (0x080 | ((num >>
              12) & 0x3F)), 8);
39         bitbuffer__write_int(buffer, (0x080 | ((num >> 6)
               & 0x3F)), 8);
40         bitbuffer__write_int(buffer, (0x080 | (num & 0x3F
              )), 8);
41     }
```

```
42      else if (num <= 0x7FFFFFFF)
43      {
44          bitbuffer__write_int(buffer, (0x0FC | (num >> 30)
              ), 8);
45          bitbuffer__write_int(buffer, (0x080 | ((num >>
              24) & 0x3F)), 8);
46          bitbuffer__write_int(buffer, (0x080 | ((num >>
              18) & 0x3F)), 8);
47          bitbuffer__write_int(buffer, (0x080 | ((num >>
              12) & 0x3F)), 8);
48          bitbuffer__write_int(buffer, (0x080 | ((num >> 6)
               & 0x3F)), 8);
49          bitbuffer__write_int(buffer, (0x080 | (num & 0x3F
              )), 8);
50      }
51      else
52      {
53          bitbuffer__write_int(buffer, 0x0FE, 8);
54          bitbuffer__write_int(buffer, (0x080 | ((num >>
              30) & 0x3F)), 8);
55          bitbuffer__write_int(buffer, (0x080 | ((num >>
              24) & 0x3F)), 8);
56          bitbuffer__write_int(buffer, (0x080 | ((num >>
              18) & 0x3F)), 8);
57          bitbuffer__write_int(buffer, (0x080 | ((num >>
              12) & 0x3F)), 8);
58          bitbuffer__write_int(buffer, (0x080 | ((num >> 6)
               & 0x3F)), 8);
59          bitbuffer__write_int(buffer, (0x080 | (num & 0x3F
              )), 8);
60      }
61  }
62
63  byte flac__encode_blksize(uint16 blksize)
64  {
65      if (blksize == 192)
66          return 0x01;
67      else if (blksize == 576)
68          return 0x02;
69      else if (blksize == 1152)
70          return 0x03;
71      else if (blksize == 2304)
72          return 0x04;
73      else if (blksize == 4608)
74          return 0x05;
```

```
75      else if (blksize == 256)
76          return 0x08;
77      else if (blksize == 512)
78          return 0x09;
79      else if (blksize == 1024)
80          return 0x0A;
81      else if (blksize == 2048)
82          return 0x0B;
83      else if (blksize == 4096)
84          return 0x0C;
85      else if (blksize == 8192)
86          return 0x0D;
87      else if (blksize == 16384)
88          return 0x0E;
89      else if (blksize == 32768)
90          return 0x0F;
91      else if (blksize <= 256)
92          return 0x06;
93      else
94          return 0x07;
95  }
96
97  byte flac__encode_srate(uint32 srate)
98  {
99      if (srate == 8000)
100         return 0x04;
101     else if (srate == 16000)
102         return 0x05;
103     else if (srate == 22050)
104         return 0x06;
105     else if (srate == 24000)
106         return 0x07;
107     else if (srate == 32000)
108         return 0x08;
109     else if (srate == 44100)
110         return 0x09;
111     else if (srate == 48000)
112         return 0x0A;
113     else if (srate == 96000)
114         return 0x0B;
115     else if (((srate % 1000) == 0) && ((srate / 1000) <=
           255))
116         return 0x0C;
117     else if (srate < 65536)
118         return 0x0D;
```

```
119     else if (((srate % 10) == 0) && (srate <= 655350))
120         return 0x0E;
121     else
122         return 0x00;
123 }
124
125 byte flac__encode_chan_sampsize(uint32 nchan, uint32
       sampwidth)
126 {
127     byte chan_sampsize = 0x00;
128     chan_sampsize |= (byte) ((nchan - 1) << 3);
129     if (sampwidth == 8)
130         chan_sampsize |= 0x01;
131     else if (sampwidth == 12)
132         chan_sampsize |= 0x02;
133     else if (sampwidth == 16)
134         chan_sampsize |= 0x04;
135     else if (sampwidth == 20)
136         chan_sampsize |= 0x05;
137     else if (sampwidth == 24)
138         chan_sampsize |= 0x06;
139     return chan_sampsize;
140 }
141
142 byte *flac__make_streaminfo(boolean last,
       flac__streamstats *stats)
143 {
144     int32 i;
145     byte *temp;
146     bitbuffer *buffer;
147     buffer = (bitbuffer *) bitbuffer__allocate(38); //38
           = 4-byte header plus 34-byte block
148     if (last == 1)
149         bitbuffer__write_int(buffer, 1, 1);
150     else
151         bitbuffer__write_zeros(buffer, 1);
152     bitbuffer__write_zeros(buffer, 7); // block type 0 =
           STREAMINFO
153     bitbuffer__write_int(buffer, 34, 24); // STREAMINFO
           blocks are always 34 bytes long
154     bitbuffer__write_int(buffer, stats->min_blksize, 16);
155     bitbuffer__write_int(buffer, stats->max_blksize, 16);
156     bitbuffer__write_int(buffer, stats->min_frmsize, 24);
157     bitbuffer__write_int(buffer, stats->max_frmsize, 24);
158     bitbuffer__write_int(buffer, stats->srate, 20);
```

```
159     bitbuffer__write_int(buffer, (stats->nchan - 1), 3);
160     bitbuffer__write_int(buffer, (stats->bitspersamp - 1)
           , 5);
161     bitbuffer__write_int(buffer, (int32) (stats->numsamp
           >> 32), 4);
162     bitbuffer__write_int(buffer, (int32) (stats->numsamp
           & 0x00000000FFFFFFFF), 32);
163     bitbuffer__write_zeros(buffer, 128); //md5sum
           placeholder
164     bitbuffer__flush_buffer(buffer);
165     temp = buffer->data;
166     free(buffer);
167     return temp;
168 }
169
170 void flac__make_frame_header(bitbuffer *buffer,
       flac__framestats *frmstats, boolean vbs)
171 {
172     byte encblksize, encsrate, temp, crc;
173     // set the maximum size for the bitbuffer to be the
           size of an unencoded frame plus headers
174     bitbuffer__write_int(buffer, 0x3FFE, 14); //sync code
175     bitbuffer__write_zeros(buffer, 2); //mandatory value
176     encblksize = flac__encode_blksize(frmstats->blocksize
           );
177     bitbuffer__write_int(buffer, encblksize, 4);
178     encsrate = flac__encode_srate(frmstats->srate);
179     bitbuffer__write_int(buffer, encsrate, 4);
180     temp = flac__encode_chan_sampsize(frmstats->nchan,
           frmstats->bitspersamp);
181     bitbuffer__write_int(buffer, temp, 7);
182     bitbuffer__write_zeros(buffer, 1);
183     if (vbs)
184         flac__append_utf8_num(buffer, frmstats->sampnum);
185     else
186         flac__append_utf8_num(buffer, frmstats->blocknum)
               ;
187     if (encblksize == 0x07) //"get 16-bit blocksize from
           end of header"
188         bitbuffer__write_int(buffer, (frmstats->blocksize
               - 1), 16);
189     else if (encblksize == 0x06) //"get 8-bit blocksize
           from end of header"
190         bitbuffer__write_int(buffer, (frmstats->blocksize
               - 1), 8);
```

```
191     if (encsrate == 0x0C) //"get 8-bit sample rate in kHz
            from end of header"
192         bitbuffer__write_int(buffer, (frmstats->srate /
                1000), 8);
193     else if (encsrate == 0x0D) //"get 16-bit sample rate
            in Hz from end of header"
194         bitbuffer__write_int(buffer, frmstats->srate, 16)
                ;
195     else if (encsrate == 0x0E) //"get 16-bit sample rate
            in tens of Hz from end of header"
196         bitbuffer__write_int(buffer, (frmstats->srate /
                10), 16);
197     bitbuffer__flush_buffer(buffer);
198     crc = crc8(buffer->data, buffer->size);
199     bitbuffer__write_int(buffer, crc, 8);
200 }
201 
202 #ifndef __SOFTWARE_CODER
203 
204 uint32 flac__analyze_frame_cinst(int32 *wavdata, uint16
        numsamples)
205 {
206     uint32 dataa, datab, status, result;
207     nm_lpc_reset_analysis(0, 0);
208     while (numsamples >= 4)
209     {
210         status = cinst_lpc__wait_busy();
211         dataa = (((*wavdata & 0x0000FFFF) << 16) | (*(
                wavdata + 1) & 0x0000FFFF));
212         datab = (((*(wavdata + 2) & 0x0000FFFF) << 16) |
                (*(wavdata + 3) & 0x0000FFFF));
213         wavdata += 4;
214         nm_lpc_analyze_samples_4(dataa, datab);
215         numsamples -= 4;
216     }
217     status = cinst_lpc__wait_busy();
218     if (numsamples == 3)
219     {
220         dataa = (((*(wavdata) & 0x0000FFFF) << 16) | (*(
                wavdata + 1) & 0x0000FFFF));
221         datab = (((*(wavdata + 2) & 0x0000FFFF) << 16));
222         nm_lpc_analyze_samples_3(dataa, datab);
223     }
224     else if (numsamples == 2)
225     {
```

```
226         dataa = (((*(wavdata) & 0x0000FFFF) << 16) | (*(
               wavdata + 1) & 0x0000FFFF));
227         nm_lpc_analyze_samples_2(dataa, 0);
228     }
229     else if (numsamples == 1)
230     {
231         dataa = (((*(wavdata) & 0x0000FFFF) << 16));
232         nm_lpc_analyze_samples_1(dataa, 0);
233     }
234     cinst_lpc__wait_busy();
235     nm_lpc_process_analysis_result(0, 0);
236     cinst_lpc__wait_busy();
237     result = nm_lpc_get_analysis_result(0, 0);
238     return result;
239 }
240
241 void flac__code_frame_cinst(bitbuffer *buffer, int32 *
       wavdata, uint16 numsamples, uint8 order, uint8 rparam)
242 {
243     int32 i, j, samplesleft;
244
245     uint32 dataa, datab, result, status;
246     samplesleft = numsamples - order;
247
248     dataa = ((order << 19) | (rparam << 16) | (*(wavdata
           + order - 1) & 0x0000FFFF));
249     datab = 0;
250     for (i = 2; i <= order; i++)
251     {
252         /* Construct datab.  The order of the warmup
253         samples is important: If used, datab holds the
254         first one or two warmup samples. */
255         datab |= ((*(wavdata + order - i) & 65535) << ((3
               - i) * 16));
256     }
257     /* send parameters */
258     status = cinst_lpc__wait_busy();
259     nm_lpc_begin_block(dataa, datab);
260     wavdata += order;
261
262     while (samplesleft >= 4)
263     {
264         status = cinst_lpc__wait_busy();
265         if (status == 0)
266         {
```

```
267             //printf("Sending four samples...\n");
268             dataa = (((*(wavdata) & 0x0000FFFF) << 16) |
                   (*(wavdata + 1) & 0x0000FFFF));
269             datab = (((*(wavdata + 2) & 0x0000FFFF) <<
                   16) | (*(wavdata + 3) & 0x0000FFFF));
270             #ifdef FLAC_DEBUG
271             printf("Sending samples [%d] = 0x%04X, [%d] =
                    0x%04X, [%d] = 0x%04X, [%d] = 0x%04X\n",
                   numsamples - samplesleft, \
272 *wavdata & 0x0000FFFF, numsamples - samplesleft + 1, *(
        wavdata + 1) & 0x0000FFFF, numsamples - samplesleft +
        2, \
273 *(wavdata + 2) & 0x0000FFFF, numsamples - samplesleft +
        3, *(wavdata + 3) & 0x0000FFFF);
274             #endif
275             wavdata += 4;
276             nm_lpc_code_samples_4(dataa, datab);
277             samplesleft -= 4;
278         }
279         else if (status == 2)
280         {
281             while (status == 2)
282             {
283                 result = nm_lpc_get_coded_result(0, 0);
284                 #ifdef FLAC_DEBUG
285                 printf("flac__code_frame_cinst: got 0x%08
                       X from instruction\n", result);
286                 #endif
287                 bitbuffer__write_int(buffer, result, 32);
288                 status = cinst_lpc__wait_busy();
289             }
290         }
291     }
292     while (samplesleft >= 1)
293     {
294         status = cinst_lpc__wait_busy();
295         if (status == 0)
296         {
297             dataa = ((*(wavdata) & 0x0000FFFF) << 16);
298             wavdata++;
299             nm_lpc_code_samples_1(dataa, 0);
300             samplesleft--;
301         }
302         else if (status == 2)
303         {
```

```
304             while (status == 2)
305             {
306                 result = nm_lpc_get_coded_result(0, 0);
307                 #ifdef FLAC_DEBUG
308                 printf("flac__code_frame_cinst: got 0x%08
                       X from instruction\n", result);
309                 #endif
310                 bitbuffer__write_int(buffer, result, 32);
311                 status = cinst_lpc__wait_busy();
312             }
313         }
314     }
315     status = cinst_lpc__wait_busy();
316 
317     while (status == 2)
318     {
319         result = nm_lpc_get_coded_result(0, 0);
320         #ifdef FLAC_DEBUG
321         printf("flac__code_frame_cinst: got 0x%08X from
               instruction\n", result);
322         #endif
323         bitbuffer__write_int(buffer, result, 32);
324         status = cinst_lpc__wait_busy();
325     }
326     j = nm_lpc_get_remaining_bits(0, 0);
327     #ifdef FLAC_DEBUG
328     printf("flac__code_frame_cinst: %d bits remain\n", j)
           ;
329     #endif
330     result = nm_lpc_get_coded_result(0, 0);
331     #ifdef FLAC_DEBUG
332     printf("flac__code_frame_cinst: got 0x%08X from
           instruction (last in block)\n", result);
333     #endif
334     bitbuffer__write_int(buffer, result, j);
335 }
336 #endif // ndef __SOFTWARE_CODER
337 
338 uint8 flac__estimate_best_rice(uint16 blksize, uint32
       total_error)
339 {
340     uint8 k = 0;
341     uint32 n = blksize;
342     while (n < total_error)
343     {
```

```
344         k++;
345         n <<= 1;
346     }
347     return k;
348 }
349
350 uint8 flac__estimate_best_order(int32 *wavdata, uint32
       sample_count, uint32 *error)
351 {
352     int32 temp;
353     uint32 error1 = 0;
354     uint32 error2 = 0;
355     uint32 error3 = 0;
356     int32 *error1_array;
357     int32 *error2_array;
358     uint16 k;
359     error1_array = (int32 *) calloc(sample_count, sizeof(
           int32));
360     error2_array = (int32 *) calloc(sample_count, sizeof(
           int32));
361     for (k = 1; k < sample_count; k++)
362     {
363         temp = *(wavdata + k) - *(wavdata + k - 1);
364         *(error1_array + k - 1) = temp;
365         if (temp < 0)
366             error1 -= temp;
367         else
368             error1 += temp;
369     }
370     for (k = 1; k < (sample_count - 1); k++)
371     {
372         temp = *(error1_array + k) - *(error1_array + k -
               1);
373         *(error2_array + k - 1) = temp;
374         if (temp < 0)
375             error2 -= temp;
376         else
377             error2 += temp;
378     }
379     for (k = 1; k < (sample_count - 2); k++)
380     {
381         temp = *(error2_array + k) - *(error2_array + k -
               1);
382         if (temp < 0)
383             error3 -= temp;
```

```
384         else
385             error3 += temp;
386     }
387     free(error1_array);
388     free(error2_array);
389     if ((error1 <= error2) && (error1 <= error3))
390     {
391         *error = error1;
392         return 1;
393     }
394     else if ((error2 < error1) && (error2 <= error3))
395     {
396         *error = error2;
397         return 2;
398     }
399     else
400     {
401         *error = error3;
402         return 3;
403     }
404 }
405
406 uint32 flac__analyze_frame_soft(int32 *wavdata, uint32
       sample_count)
407 {
408     uint32 total_error, packed_order_param;
409     uint8 order, rparam;
410     order = flac__estimate_best_order(wavdata,
           sample_count, &total_error);
411     rparam = flac__estimate_best_rice(sample_count,
           total_error);
412     packed_order_param = (uint32) (rparam | (order << 3))
           ;
413     return packed_order_param;
414 }
415
416 void flac__calc_residual_fixed(int32 *wavdata, uint16
       numsamples, uint8 order, int32 *rp)
417 {
418     int32 i;
419     wavdata += order;
420     if (order == 3)
421     {
422         for (i = 3; i < numsamples; i++)
423         {
```

```
424             *(rp++) = *wavdata - (3 * (*(wavdata - 1))) +
                    (3 * (*(wavdata - 2))) - *(wavdata - 3);
425             wavdata++;
426         }
427     }
428     else if (order == 2)
429     {
430         for (i = 2; i < numsamples; i++)
431         {
432             *(rp++) = *wavdata - (2 * (*(wavdata - 1))) +
                    *(wavdata - 2);
433             wavdata++;
434         }
435     }
436     else if (order == 1)
437     {
438         for (i = 1; i < numsamples; i++)
439         {
440             *(rp++) = *wavdata - *(wavdata - 1);
441             wavdata++;
442         }
443     }
444 }
445 
446 void flac__code_frame_fixed_soft(bitbuffer *buffer, int32
      *wavdata, uint16 numsamples, uint8 order, uint8 rparam
    )
447 {
448     int32 *rp;
449     int32 i;
450     rp = (int32 *) calloc(numsamples - order, sizeof(
           int32));
451     flac__calc_residual_fixed(wavdata, numsamples, order,
             rp);
452     for (i = 0; i < (numsamples - order); i++)
453         bitbuffer__write_rice(buffer, *(rp + i), rparam);
454     free(rp);
455 }
456 
457 void flac__make_subframe_fixed(bitbuffer *buffer, int32 *
    wavdata, flac__framestats *frmstats)
458 {
459     uint32 result;
460     uint8 rparam, order;
461     int32 subfrm_header;
```

```
462     uint8 i;
463     #ifndef __SOFTWARE_CODER
464     result = flac__analyze_frame_cinst(wavdata, frmstats
           ->blocksize);
465     #else
466     result = flac__analyze_frame_soft(wavdata, frmstats->
           blocksize);
467     #endif
468     rparam = (uint8) (result & 0x00000007);
469     order = (uint8) ((result >> 3) & 0x00000003);
470     subfrm_header = 0x08 | order;
471     bitbuffer__write_int(buffer, subfrm_header, 7);
472     bitbuffer__write_zeros(buffer, 1); //to be replaced
           if "wasted bits" is checked and implemented
473     //write warmup samples
474     for (i = 0; i < order; i++)
475         bitbuffer__write_int(buffer, *(wavdata + i),
               frmstats->bitspersamp);
476     bitbuffer__write_zeros(buffer, 2); //partitioned rice
            coding
477     bitbuffer__write_zeros(buffer, 4); //partition order
           = 0
478     bitbuffer__write_int(buffer, rparam, 4);
479     #ifndef __SOFTWARE_CODER
480     flac__code_frame_cinst(buffer, wavdata, frmstats->
           blocksize, order, rparam);
481     #else
482     flac__code_frame_fixed_soft(buffer, wavdata, frmstats
           ->blocksize, order, rparam);
483     #endif
484 }
485
486 void flac__make_frame(bitbuffer *buffer, int32 *wavdata,
       flac__framestats *stats, boolean vbs)
487 {
488     uint16 frame_footer;
489     flac__make_frame_header(buffer, stats, vbs);
490     flac__make_subframe_fixed(buffer, wavdata, stats);
491     bitbuffer__flush_buffer(buffer);
492     frame_footer = crc16(buffer->data, buffer->size);
493     bitbuffer__write_int(buffer, frame_footer, 16);
494     //bitbuffer__flush_buffer(buffer);
495 }
```

B.4 FLAC Data Types (flac.h)

```
1  /*
     ************************************************************
2  flac.h
3
4  Defines types for FLAC format
5
6  ***********************************************************
     */
7  #ifndef __FLAC_H
8  #define __FLAC_H
9
10 #include "formats.h"
11
12 typedef struct
13 {
14    uint16 min_blksize;
15    uint16 max_blksize;
16    uint32 min_frmsize;
17    uint32 max_frmsize;
18    uint32 srate;
19    uint8 nchan;
20    uint8 bitspersamp;
21    uint64 numsamp;
22 } flac__streamstats;
23
24 typedef struct
25 {
26    uint16 blocksize;
27    uint8 nchan;
28    uint8 bitspersamp;
29    uint32 blocknum;
30    uint64 sampnum;
31    uint32 srate;
32 } flac__framestats;
33
34
35 #endif
```

B.5 FAT Filesystem Data Types (fat.h)

```
1 /*
   *******************************************************
2 fat.h
3
4 Defines types for FAT filesystem driver
5
6 *******************************************************
   */
7 #ifndef __FAT_H
8 #define __FAT_H
9
10 #include "formats.h"
11
12 #define FAT_EOF    0xFFFFFFFF
13
14
15 typedef struct
16 {
17    uint32 sector_start;
18    uint16 bytes_per_sector;
19    uint8 sectors_per_cluster;
20    uint16 reserved_sectors;
21    uint8 FAT_count;
22    uint16 root_dir_entries;
23    uint32 total_sectors_low;
24    uint32 total_sectors_high;
25    uint16 sectors_per_FAT;
26    uint32 hidden_sectors;
27    uint8 fstype;
28    uint32 FAT_location_sector;
29    uint32 rootdir_location_sector;
30    uint32 data_location_sector;
31    uint32 data_location_byte_offset;
32    byte *rdp;
33    byte *fatp;
34 } fat__ptinfo;
35
36 typedef struct
37 {
38    uint8 filename[9];
39    uint8 extension[4];
40    uint8 attr;
41    uint8 fine_res_timestamp;
42    uint32 creation_time;
```

```
43    uint16 last_access_time;
44    uint32 last_modified_time;
45    uint16 first_cluster;
46    uint32 file_size;
47 } fat__filestats;
48
49 typedef struct
50 {
51    fat__filestats *stats;
52    uint32 current_cluster;
53    uint32 next_cluster;
54    uint32 prev_cluster;
55    uint32 byte_offset;
56    byte *cluster;
57    fat__ptinfo *fsinfo;
58    uint8 type;
59    uint16 rd_entrynum;
60 } fat__fileobj;
61
62 #endif // __FAT_H
```

Appendix C

List of MIT-BIH Database Records Tested

100
101
102
103
104
105
106
107
118
119
200
201
202
203
205
207
208
209
210
212
213
214
215
217
219