Doctoral Thesis
Sundsvall 2005

Alternating Coding and its Decoder Architectures for Unary-Prefixed Codes

Shang Xue

Supervisors: Associate Professor Bengt Oelmann
Associate Professor Mattias O'Nils

Electronics Design Division, in the
Department of Information Technology and Media
Mid Sweden University, SE-851 70 Sundsvall, Sweden

ISSN 1652-893X
Mid Sweden University Doctoral Thesis 1

ISBN 91-85317-08-X

Mittuniversitetet
MID SWEDEN UNIVERSITY

A dissertation submitted to the Mid Sweden University, Sweden, in partial fulfillment of the requirements for the degree of Doctor of Technology.

Alternating Coding and its Decoder Architectures for Unary-Prefixed Codes

Shang Xue

Electronics Design Division, in the
Department of Information Technology and Media
Mid Sweden University, SE-851 70 Sundsvall
Sweden

Telephone: +46 (0)60 148600

Printed by Kopieringen Mittuniversitetet, Sweden, 2005

To my husband Nan, my father Mr. Peiding Xue and my mother Ms. Wannan Wang

ABSTRACT

The entropy coding of high peak, heavy-tailed probability distributions such as the Laplacian, Cauchy, and generalized Gaussian have been a topic of interest because they are able to provide good models for data in many coding systems, especially in image and video coding systems. This thesis studies the entropy coding of such high peak, heavy-tailed probability distributions. By summarizing the encoding of such distributions under the concept "Unary Prefixed Codes" (UPC), the thesis depicts the encoding via a different approach. By extending the concept of UPC, the thesis proposes a universally applicable coding algorithm "Unary Prefixed Huffman" (UPH) that could be applied to both finite and infinite sources. The code set resulting from the UPH algorithm has a coding efficiency which is upper-bounded by entropy + 2 given that the entropy is finite, and is able to provide sub-optimal encoding of the sources studied in the thesis. The thesis also proposes several different variations of UPCs that are simple in structure yet efficient for use for several variations of the high peak, heavy-tailed distributions that are commonly found in image and video coding systems.

By applying the concept of the UPC, the thesis further proposes a coding method named the "Alternating Coding" (ALT) method. The ALT coding provides a coding pattern that is different from the conventional method which enables the extraction of special properties of the UPCs. Using the extraction of the special property of the UPCs, decoding could be greatly simplified and parallel decoding could be a possibility. Moreover, for the highly structured UPCs that are widely used in image and video coding systems, the ALT coding enables an error resiliency mechanism to be applied, which helps to improve the error tolerance of these UPC packets to a significant extent. Simulations and actual application results of the ALT coding are discussed in the thesis.

By applying the ALT coding, the hardware architecture of the decoder changes accordingly. The ALT decoder is different to the conventional variable length decoders that have been applied in the decoding of UPCs, as it is able to utilize the special properties of the UPCs and thus simplify the decoder architecture. As shown in the thesis, the ALT decoders are smaller in size, faster in speed and consume much less power compared to the conventional decoders. This is particularly true for those highly structured UPCs that are commonly used in image and video coding systems. Actual realizations of several ALT decoders are discussed in the thesis, and comparisons are made to the conventional decoders. The improvements are shown to be very evident.

ACKNOWLEDGEMENTS

I want to say that, studying in Sweden was a pleasant journey. I feel blessed to have this opportunity to experience this beautiful country and its amiable people while at the same being able to complete my Ph.D. study. The years spent in Sweden would definitely be a sparkling memory of my life. I will definitely come back to this peaceful land again when I have the chance.

First of all, I would like to express my gratitude toward Docent Bengt Oelmann in the first place. It was him who helped me make my way through out my Ph.D. study at Mid-Sweden University. For a non-Swedish student like me, Dr. Oelmann, as my supervisor, not only provided me with a lot of guidance and various opportunities in my study and research, but also helped me adapting to the life in Sweden as a foreigner. I would not have completed my Ph.D. study without his support and considerations. I would also like to thank Professor Youzhi Xu for introducing me to this opportunity to start my Ph.D. at Mid-Sweden University, and for his wise advices at the beginning of my Ph.D. study. I am also very grateful to Professor Hans-Erik Nilsson and Docent Mattias O'Nils for their support and help. Also many thanks shall be given to the people in our department: Cao, Jon, Henrik, Mats, Krister, Munir and many more. Thank you for your kindness and friendliness and thank you for the happy parties.

I also want to thank all my Chinese fellows whom I met in Sundsvall: Cris Ding and Xiaoou Song, Guangjiong Dong and Juanwen, Tao Feng and Yan Song, Lixin Ning and Xiaoli Hou. Life is much easier and more fun with all your help and accompany. It is really lucky to get to know all of you.

The Mid-Sweden University and the KK-foundation are greatly acknowledged for their financial support.

Most of all, I want to share this thesis with my dear husband Nan Gu, my father Mr. Peiding Xue and my mother Ms. Wannan Wang back in China. Without a supportive and caring family, life would have been much tougher for me, especially during those lonely, homesick days far far away from my beautiful homeland.

Sundsvall, April 2005

Shang Xue

TABLE OF CONTENTS

ABBREVIATIONS AND ACRONYMS

GENERAL

AC…...……….	Alternating Current
ALT..……….	Alternating Coding
ASIC…………	Application-Specific Integrated Circuit
BDL…………	Boundary Detection Logic
BER………….	Bit Error Rate
BSC………….	Binary Symmetric Channel
CABAC…….	Context-Based Adaptive Binary Arithmetic Coding
CAVLC…….	Context-Based Adaptive Variable Length Coding
CDL…………	Codeword Disabling Logic
CMOS……….	Complementary Metal Oxide Semiconductor
CODEC………	Encoder and DECoder
CR……………	Correct Ratio
DC……………	Direct Current
DCT…………	Discrete Cosine Transform
EG…………..	Exponential Golomb Code
EIB………….	Even-Indexed Bits
EOB………….	End Of Block
ES……………	Error Speculation
FGS………….	Fine Granular Scalability
FIFO…………	First In First Out
FPGA………..	Field-Programmable Gate Array
FSM…………	Finite State Machine
GG……………	Generalized Gaussian
GR…………	Golomb Rice Code
HG…………	Hybrid Golomb Code
HVS………….	Human Visual Systems
IDCT…………	Inverse Discrete Cosine Transform
LB……………	Length Buffer
JPEG…………	Joint Photographic Experts Group
KLT………….	Karhunen-Loeve Transform
LE……………	Length Extraction unit
LUT………….	Look-Up Table
MC…………..	Motion Compensation
ME……………	Motion Estimation
MPEG………..	Motion Pictures Experts Group
OIB………….	Odd-Indexed Bits
PCLE…………	Parallel Codeword Length Extractor

pdf……………	Probability Density Function
PISO…………	Parallel-Input Serial Output
PLA…………..	Programmable Logic Array
PLS…………	Fast variable length decoder using Plane Separation
PSNR…………	Peak-Signal-to-Noise Ratio
RLD………….	Remaining Length Detector
RT-level	Register Transfer Level
RVLC………..	Reversible Variable Length Codes
UPC………….	Unary-Prefixed Code
UPH………….	Unary-Prefixed Huffman Code
UVLC………...	Universal Variable Length Code
VHDL………..	Very High Speed Integrated Circuit Hardware Description Language
VLC…………	Variable Length Code
XOR................	exclusive OR

LIST OF FIGURES

LIST OF PAPERS

This thesis is mainly based on the following ten papers, herein referred to by their Roman numerals:

Paper I	**Unary Prefixed Huffman Coding for a Group of Quantized Generalized Gaussian Sources** Shang Xue and Bengt Oelmann, Submitted to IEEE Transaction on Communications
Paper II	**Unary-Prefixed Encoding of the Lengths of Consecutive Zeros in a Bit Vector** Shang Xue and Bengt Oelmann, IEE Electronics Letters, vol.41, no.6, pp.346-347, 2005
Paper III	**Efficient Decoding of Variable Length Encoded Image Data on the Nios II Soft-Core Processor** Peter Mårtensson, Jens Persson, Shang Xue, and Bengt Oelmann, In the proceedings of the International Workshop on Applied Reconfigurable Computing, Algarve, Portugal, February 2005
Paper IV	**Efficient VLSI Implementation of a VLC Decoder for Golomb-Rice Code using Alternating Coding** Shang Xue and Bengt Oelmann, In the proceedings of the IEEE Norchip'03, Riga, Latvia, November, 2003
Paper V	**Parallel Variable-Length Decoder Architecture for Alternated Coded GR-Codes** Shang Xue and Bengt Oelmann, In the proceedings of the IEEE Norchip'03, Riga, Latvia, November, 2003
Paper VI	**Error Resilient coding of DCT coefficients using alternating coding of UVLC** Shang Xue and Bengt Oelmann, In the proceedings of Norsig, Bergen, Norway, October, 2003
Paper VII	**A Coding Method for UVLC Targeting Efficient Decoder Architecture** Shang Xue and Bengt Oelmann,

In the proceedings of the 3rd IEEE International Symposium on Image and Signal Processing and Analysis, Rome, Italy, September, 2003

Paper VIII **Alternating Coding for Universal Variable Length Code**
Shang Xue and Bengt Oelmann,
In the proceedings of the IEEE International Conference on Image Processing, Barcelona, Spain, September, 2003

Paper IX **Efficient VLSI Implementation of a VLC Decoder for Universal Variable Length Code using Alternating Coding**
Shang Xue and Bengt Oelmann,
In the proceedings of IEEE Annual Symposium on VLSI, Tampa, Florida, USA, February, 2003

Paper X **Hybrid Golomb Codes for a Group of Quantized GG Sources**
Shang Xue, Youshi Xu and Bengt Oelmann,
IEE Proceedings -- Vision, Image and Signal Processing, vol.150, no. 4, pp. 256-260, August, 2003

1 INTRODUCTION

This chapter is an introduction of the entire thesis work, which includes the background and motivation associated with the thesis work, and a brief description of the thesis study.

1.1 BACKGROUND

The work in this thesis originated from a study of the entropy coding of some image and video data. The encoding and decoding of image and video data, especially video data, requires an entire complex system which is an integration of many different functional parts. To convert image/video into electronic signals that are suitable for physical transmission is no easy task. Especially for image/video, the high bit rates that result from the various types of digital video make their transmission through their intended channels very difficult. Compression coding bridges a crucial gap between the user's demands (high-quality still and moving images, delivered quickly at a reasonable cost) and the limited capabilities of transmission networks and storage devices. For example [43], a "television quality" digital video signal requires 216 Mbits of storage or transmission capacity for one second of video. Transmission of this type of signal in real time is beyond the capabilities of most present-day communications networks. A two-hour movie (uncompressed) requires over 194 Gbytes of storage, equivalent to 42 DVDs or 304 CD-ROMs. In order for digital video to become a plausible alternative to its analogue predecessors (such as the analogue television), it is necessary to develop methods to reduce or compress this prohibitively high bit-rate signal.

The drive to solve this problem has taken decades and massive efforts in research, development and standardization. Significant gains in storage, transmission, and processor technology have been achieved in recent years, and it is primarily the reduction of the amount of data that needs to be stored, transmitted, and processed that has made widespread use of digital video a possibility.

Modern image/video coding standards have adopted comprehensive compression methods to remove the redundancy in image and video data and thus compress the amount of data to be stored and transmitted. Compression could be performed at the encoder for transmission and then decompressed at the decoder to restore the original signals. The decompressed signal may be identical to the original signal (lossless compression) or it may be distorted and degraded (lossy compression). Compression of image and video signals is based on the fact that there are always spatial, temporal or statistical redundancies that could be removed. For instance, neighboring pixels in an image or a video frame tend to be highly correlated and so there is significant spatial redundancy. Neighboring regions within successive video frames also tend to be highly correlated and thus significant temporal redundancy exists. These statistical redundancies could be modeled by using proper source models. A good source model then attempts to exploit the properties of video or image data and to represent it in a form that can be readily compressed by an entropy encoder. A source model may also take advantage of subjective redundancy, exploiting the sensitivity of the human visual system (HVS) to various characteristics of image and video. For example, the HVS is much more sensitive to low rather than to high frequencies and so it is possible to compress an image by eliminating certain subjectively redundant components of the information. Although the decoded image is no longer identical to the original, the information loss is hardly perceived by the human viewer.

There are many different techniques of compression in the image and video coding systems. In an image coding system, there are three basic parts of compression: transform coding, quantization and entropy coding. In a video coding system, frame differencing and motion-compensated prediction are also applied to further reduce the temporal redundancies.

Figure 1-1 shows an example of the block diagram of the image enCOder and DECoder (CODEC).

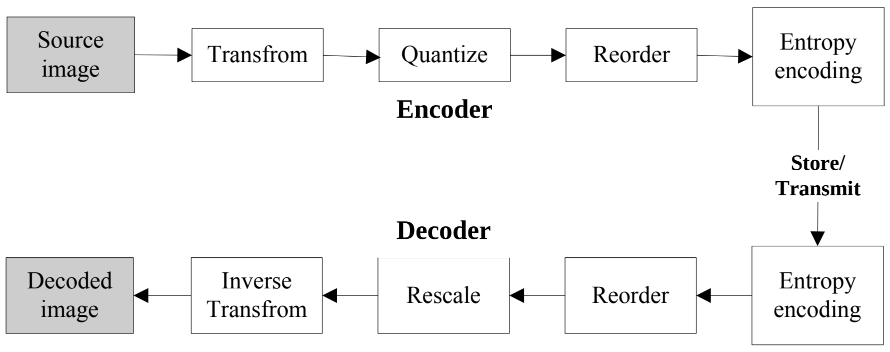


Figure 1-1 Image CODEC

In an image CODEC, the transform coding stage transforms the image from the spatial domain into another domain in order to make it more amenable to compression. The transform may be applied to discrete blocks in an image (block transform) or to the entire image. In a video coding system, a block transform is usually applied. The Karhumen-Loeve transform (KLT) has the "best" performance of any block-based image transform. The coefficients produced by the KLT are decorrelated and the energy is packed into a minimal number of coefficients. However, KLT is very computationally complex and is impractical for use. The discrete cosine transform (DCT) performs nearly as well as the KLT and is much more computationally efficient and therefore DCT is usually applied. The DCT are usually applied as block-base transforms. Figure 1-2 shows an example of a block-based DCT. In the original block, it can be seen that the energy is distributed across all the samples but after the DCT, the energy is concentrated into a few significant coefficients (at the top left). Other types of transforms such as the wavelet transform are also commonly found in the image coding systems.

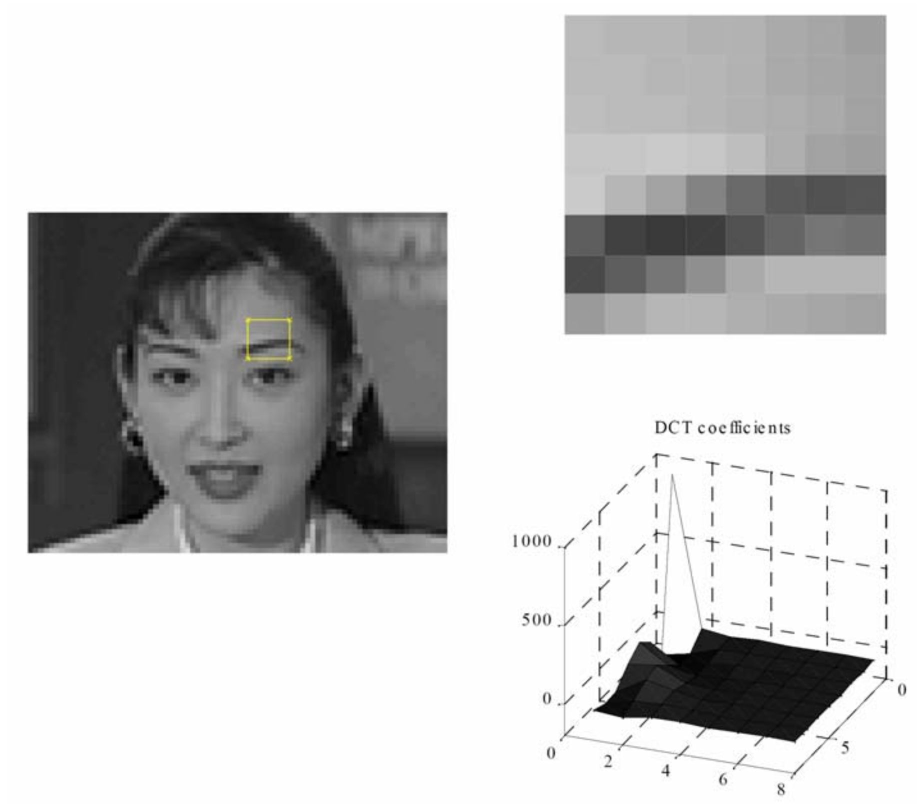
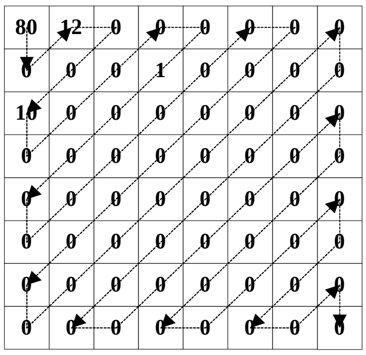


Figure 1-2 Block based DCT

The quantization stage in an image encoder removes those components of the transformed data unimportant to the visual appearance of the image but retains the visually important components. This is typically done by dividing each transformed coefficient by an integer and then discarding the remainder.

80	**12**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**1**	**0**	**0**	**0**	**0**
10	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**

8x8 quantized DCT coefficeints



Zigzag reordering

Figure 1-3 Zigzag reordering

After the image is transformed and quantized, the quantized coefficients are reordered so that the non-zero values can be grouped together in sequence. The non-zero quantized coefficients are usually clustered around the "top-left" corner containing mainly the low frequency coefficients and thus by means of a zigzag scan, the non-zero coefficients can be grouped together. Figure 1-3 illustrates the zigzag ordering of the quantized transformed coefficients. The reordered coefficient array usually consists of a group of non-zero coefficients followed by mostly zeros. For the example in Figure 1-3, the zigzag scanned DCT coefficients appear as follows:

80, 0, 12, 0, 0, 10, 0, 0, 0, 0, 0, 1, 0, 0, ..., 0.

Such a pattern is usually coded using the run length coding where the length of zeros between non-zero values and the non-zero value are coded as a (run, level) pair instead of coding every single repeating zero in the array. So for the example in Figure 1-3, the (run, level) pair appears as:

80, (1, 12), (2, 10), (5, 1), EOB (End Of Block).

Statistical models are then applied to the run length coded data and entropy coding of the statistical models is performed. The entropy coding of these data involves different statistical models and different coding algorithms. The statistical models are usually source distributions with high peaks, heavy tails, and coding algorithms involving variable length encoding and arithmetic coding. Variable length encoding is a common technique used in coding any discrete source, which assigns shorter codewords to frequent symbols and longer codewords to infrequent symbols in order to reduce the average code length. Arithmetic coding achieves variable length encoding by mapping a series of symbols to a fractional number which is then converted into a binary number. It has proved to be very efficient, and the match to the actual statistical model can be very accurate, but the algorithm is in general computationally complex.

The output of the entropy encoder is a sequence of binary codes that represent the original image in compressed form. To recreate the image, decoding of the compressed image is performed. The inverse procedure is taken step by step as Figure 1-1 shows.

The video coding system is even more complicated than the image coding system with the image encoder being a mere part of the video encoder. Figure 1-4 shows the block diagram of a video encoder.

A video signal consists of a sequence of individual picture frames in which each frame may be compressed individually using an image encoder (intra-frame coding). However, consecutive frames usually have strong temporal correlations and therefore could be further compressed by predicting and compensating for the current frame using previous frame references (inter-frame coding). The main difference between the video and image CODEC lies here. Predicting the current frame using those previously transmitted is called frame differencing. A residual frame is produced by subtracting the previous frame from the current frame in a video sequence, and the residual frame is compressed and transmitted instead of the current frame itself. This is the simplest predictor in a video coding system. Frame differencing enables good compression to be achieved when successive frames are similar. But when there is a significant change between the previous and current frames, significantly better predictions could be achieved by estimating the movement and compensating for it. Motion estimation and compensation assist in achieving these goals.

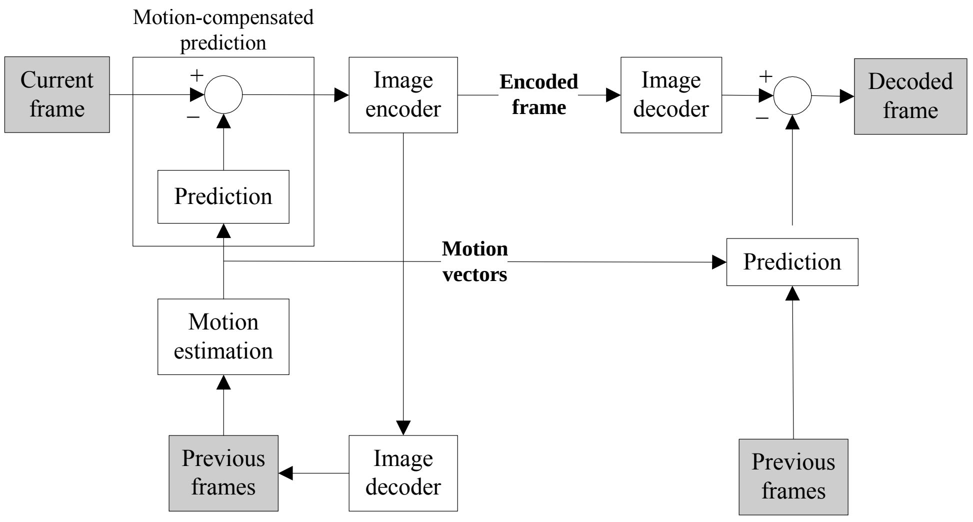


Figure 1-4 Video encoder

The entropy coding in the video coding system involves more types of data in comparison to the image encoders. In the video encoder, an image transform is applied to the residual frame and the coefficients are quantized, reordered and run-length coded. The result of the run-length coding is entropy coded as in an image encoder. However, the statistical models are generally different for intra- and inter-coded frames. Moreover, if motion compensated prediction is to be followed through, motion vector information must also be sent in addition to the run-length coded data. Therefore the motion vectors must also be entropy coded. There are also other data types such as quantizer parameter, headers and parameters etc, which all need to be entropy coded to remove the statistical redundancy. For different data types, variable length coding of proper statistical models as well as arithmetic coding could both be applied. For instance, in H.264 [44][45][49], entropy coding could be performed using fixed- or variable length codes, or context-based adaptive arithmetic coding (CABAC) [46][47][48] (which is a low-complexity adaptive binary arithmetic coding technique with context modeling), and context-based adaptive variable length coding (CAVLC) [50] and exp-Golomb codes.

From the above we see that, entropy coding is one of the key parts involved in image/video compression. Proper statistical models need to be applied to perform entropy coding efficiently.

With reference to the implementation of the video CODEC, there are many issues requiring to be taken into consideration. Video compression and decompression are known to be computationally intensive tasks that require special hardware or very powerful general-purpose processors. It is possible to implement the video coding mostly in hardware and use a micro controller to implement high-level control functions in software. However, it is also possible to implement the codec completely in software and use a high-end, high-performance micro controller or digital signal processor (or both) [58]. A special hardware solution is always better from a performance, area and power point of view as the architecture can be designed to implement a specific algorithm. A software-based solution, on the other hand, is often considered more appealing as it is flexible and easier to develop. The availability of low-cost and low-power hardware with sufficiently high performance is essential for the popularization of image and video coding applications. Thus, efficient hardware implementations in VLSI are of vital importance. However, image and video coding algorithms are characterized by very high computational complexity. Real-time processing of multi-dimensional image and video signal involves operating continuous data streams of huge volumes. Such critical demands cannot be fulfilled by conventional hardware architectures without specific adaptation [66]. Therefore any tradeoff between the software and hardware solutions should be studied carefully before the system architecture is designed.

In [59], an MPEG-4 video codec is designed using a combination of RISC and dedicated hardware engines in order to satisfy the requirements for both low power and programmability. This is because dedicated hardware is much better from power- and area-efficient standpoints and software programmability whereas an embedded reduced instruction set computer processor is preferable in order to cope with the MPEG standardization. The dedicated engines in [59] are adopted for computationally intensive functions in MPEG4, such as DCT, inverse DCT (IDCT), Motion Estimation (ME), Motion Compensation (MC), and the Variable Length Code (VLC) CODEC, while the embedded RISC processor is included to provide flexibility for other tasks. By doing so, together with several levels of low-power techniques, such as parallel operation, clock gating, etc, the design in [59] achieved 70% power saving when compared to a conventional design. In their design, it was shown that the power dissipated by the VLC decoder alone consisted of approximately 9% of the total power dissipation even using a dedicated hardware design. The DCT and IDCT module are also energy consuming components which between them consume respectively 6% and 13% of the total power dissipation. In [67], the computational load of MPEG decoder was analyzed and it was shown that the VLC decoding and inverse quantization utilize up to 24% of the total computational load, the IDCT approximately 28% of the computation, and the MC 48%. This also shows that the VLC decoding is one of the performance limiting components and requires careful consideration. It is commonly accepted that the DCT/IDCT, ME/MC, quantization and VLC decoding are the performance limiting modules in a video CODEC or multimedia system [68] [69] [70]. Almost all MPEG-4 CODEC designs [60] [61] [62] [63] [64] [65] [67] adopt dedicated module architectures for the computationally intensive ME/MC, DCT/IDCT, and the VLC CODECs. In [63], dedicated module architectures are even adopted for all coding tasks including CODEC control.

From the above we have seen that the VLC CODEC part in a video CODEC is usually designed using dedicated modules that are able to work independently, as it is one of the most computational intensive parts of the video CODEC. Therefore an efficient VLC decoder plays an important role in a video CODEC. The simplification of the VLC decoder dedicated to video systems then becomes an interesting topic to study.

1.1.1 The statistical models of some image/video data

To efficiently perform entropy coding in image and video coding systems, an accurate model of the image and video data need is a necessity regardless of which entropy coding algorithm is to be applied. The modeling of the different types of image/video data is a massive subject and has involved a great deal of effort by many researchers. The work in this thesis does not involve the modeling of image/video data. Our emphasis is to study and improve the entropy coding of some specific probability models that are often encountered in image/video encodings.

Many different types of image/video data could be modeled with probability distributions having high peaks and heavy tails. For instance, several studies on the statistical distribution of the AC coefficients have been proposed, in which the AC coefficients were conjectured to have Gaussian [34] [35], Laplacian [36] [37], or more complex distributions [38][39]. The work in [40] also indicates that the AC coefficients can be suitably modeled using Cauchy distribution. It is generally believed that the distribution of the luminance components of a transformed image block is also Laplacian [52][53]. [51]confirmed the Laplacian distribution for both the luminance and chrominance channels of DCT encoded images and video sequences. Gaussian and Laplacian distributions are the most popular statistical models used for DCT coefficients [54][55] and DCT residuals [56]. A mixed Laplacian model was proposed in [57] as an accurate statistical model for DCT residuals for the MPEG4 FGS (Fine Granular Scalability) enhancement layer. In [12], scalar quantized, run-length-coded image sub-bands are modeled using a generalized Gaussian (GG) distribution and it has proved to be a more flexible model. In [15], another discrete distribution has been designed for the length of each run of zeros in a uniformly quantized sub-band of a wavelet transformed image.

The shapes of all of these probability distributions used in the modeling of image/video data contain high peaks and heavy tails. They provide accurate models for some of the image/video data and therefore provide a reasonable model for the entropy coding of these image/video data. Figure 1-5 [51] shows an example of the distribution of some image data. Its high peak, heavy-tailed shape is very obvious.

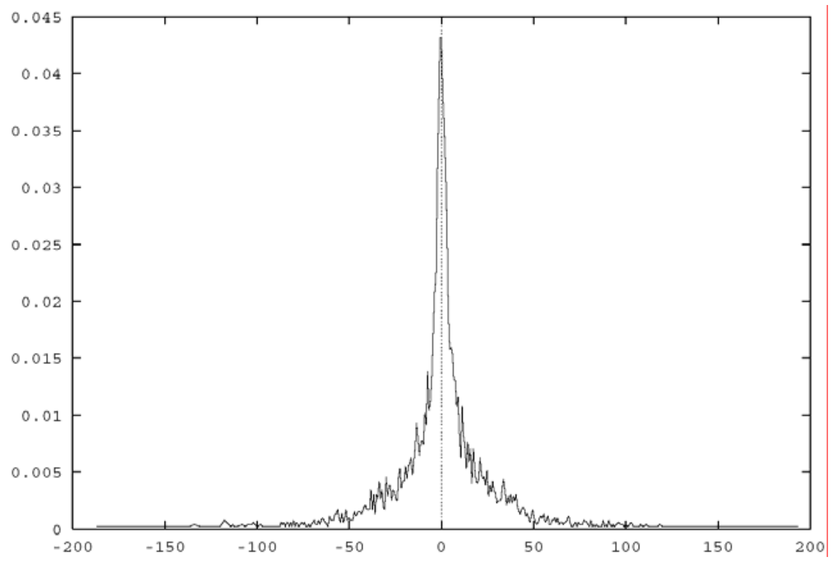


Figure 1-5 Histogram of a certain image data

1.1.2 The architecture of the variable length decoder

VLC are codes with variable code lengths. The basic concept of the entropy coding is to assign shorter codewords to symbols with higher appearance frequencies and longer codewords to symbols with lower appearance frequencies, thus reducing the average length of the codes. To encode and decode VLCs efficiently, different types of VLC encoders and decoders have been developed.

The design of VLC encoders is straightforward. We can simply describe VLC encoders using block diagrams as are shown in Figure 1-6. The input symbol is fed into a look up table and then the corresponding codeword is read out from the table. With an output buffer, codewords with variable lengths can be output at a constant rate.

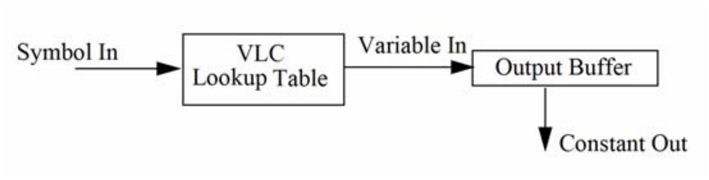


Figure 1-6 Block diagram of a VLC encoder

Decoding of the VLCs is in much more difficult since the variable lengths make the codewords difficult to separate. The codeword boundary cannot be determined until previous codewords have been decoded. This recursive dependence results in an upper bound on the iteration speed and limits the decode throughput.

The most straightforward means of implementing a VLC decoder is to use a "tree-based architecture" as shown in Figure 1-7 .

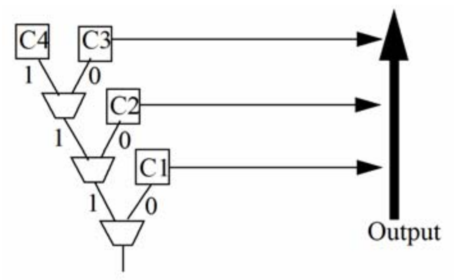


Figure 1-7 The tree-based architecture

Such a tree-based structure is based on the fact that the decoding process actually is a traversal along the directed path of the code tree. One can map the code tree directly as shown in Figure 1-7 . The branching function at each internal node can be modeled as a 1-to-2 demultiplexer. Obviously, this structure has an output of one bit per cycle.

Pipelining can increase the throughput of the tree-based decoder, as discussed by Shih-Fu Chang and David G. Messerschmitt in [41]. The most straightforward method is to partition the decoder into pipeline stages where each one includes one level of the code tree. Then the decoder can be implemented by simply cascading several ROMs, where the number of ROMs is equal to the depth of the code tree.

Although pipelining could be achieved, this direct implementation using a tree-based architecture is obviously inefficient. Many other different methods and concepts have been proposed in VLC decoder implementations. Different types of VLC decoders are developed according to the different ways in which the code word boundaries are determined. Figure 1-8, Figure 1-9 and Figure 1-10 show block diagrams of three types of decoders.

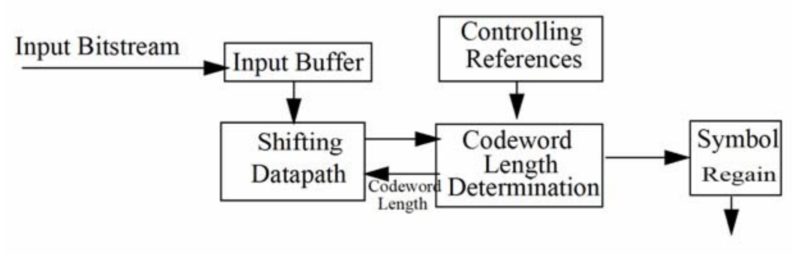


Figure 1-8 VLC decoder type one

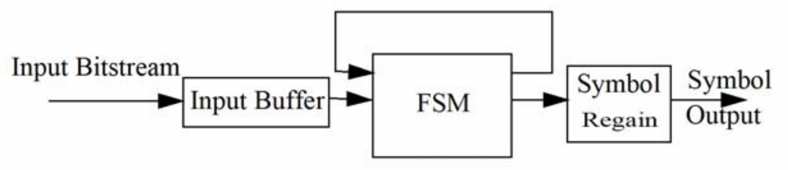


Figure 1-9 VLC decoder type two

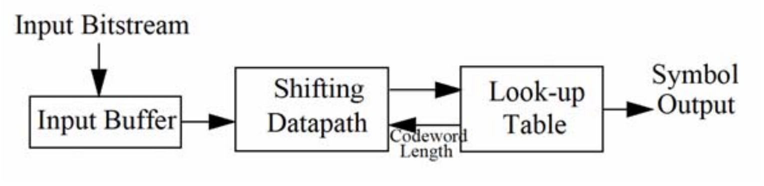


Figure 1-10 VLC decoder type three

The VLC decoder in Figure 1-10 is the most commonly used VLC decoder architecture. It is a general VLC decoder structure that could be used for any VLC. It involves the input buffer, a shifting scheme and Look-Up Tables (LUT) that provide references for the codeword lengths as well as the decoding of the actual data. It is possible to decode one codeword per clock cycle.

The bottleneck of the decoding throughput of VLC decoders is caused by the sequential dependencies of the codewords. Therefore, to break the dependency to attempt to achieve concurrency is of great importance in increasing the decoding throughput. To balance the tradeoff between throughput and complexity, the papers by H. D. Lin and D. G. Messerchmitt [42] introduced several general methods for parallel decoding processes. However, a general VLC architecture will always suffer for complexity as it is necessary to consider all the possible cases which could happen in the VLC. Such complexity leads to large, slow and power consuming designs.

1.2 MOTIVATION BEHIND THE STUDY

The motivations behind the study of this thesis are based on the following two considerations:

1. To improve the entropy coding of those probability distributions that are used to model image/video data;
2. A simplification of the VLC decoder for these image/video codes

1.2.1 Improvement in the entropy coding

As was described in section 1.1.1, there are several different probability distributions that are used to model some of the image/video data. Even for one type of image/video data, such as the DCT coefficients, there are different probability models used to model them. The entropy coding for each probability model, is usually at least slightly different. Therefore different entropy codes have been developed for these different probability distributions and have been applied to the coding of some image/video data. Considering these distribution and code variations, it is sometimes difficult to select an optimal match or indeed a sub-optimal one. For instance, optimal entropy codes exist for the Laplacian distributions, yet for the GG distributions, no optimal codes could be constructed. Therefore, to efficiently model and encode the image/video data source, it is necessary to not only match the data to a good statistical model, but also alter the entropy encoding of these statistical models.

It is well known that the Huffman encoding algorithm [1] has proved to be optimal for any finite source. Therefore, it might be considered possible to apply the Huffman encoding algorithm to the different statistical models thus avoiding the need to select another efficient entropy code. However, the distributions of these image/video data are all modeled using infinite sources which are not applicable to the Huffman algorithm. The reason behind this is that the Huffman algorithm requires the encoding to be initiated through the merger of the two symbols with the least probability values, whereas for infinite sources, there are no “least” probability values.

In order to tackle these infinite sources while at the same time still being flexible in order to adapt to the change caused by using different statistical models in the encoding procedure, in this thesis we have attempted to study and improve the entropy coding of these high-peaked, heavy-tailed probability distributions and have proposed new codes as well as coding algorithms.

Moreover, the resulting entropy codes are, in the majority of cases VLCs. The VLC has the disadvantage of being vulnerable to transmission errors, as will be demonstrated in Chapter 3. The work in this thesis also attempts to improve the error-resiliency of the entropy codes for the probability distributions used to model some of the image/video data.

1.2.2 Simplification of the decoder architecture

As we have mentioned in the previous section, the most commonly used, and most efficient VLC decoder structure involves buffering, shifting and table-look-up in its architecture. The shifting scheme and the LUTs are usually large, slow and power consuming and these all limit the performance of parts of the VLC decoder.

The key point in a VLC decoder is the determination of the variable code lengths, which is necessary in order to proceed with the decoding. For a common VLC decoder, determining the lengths of the decoders is only possible by searching the LUT, matching the codewords and reading out the code lengths of these codewords. With the decoded code length, the shifting scheme would be able to shift out the decoded codewords and immediately restart decoding. However, there are certain VLCs where the very structure of the codes provides additional information concerning the lengths of the code lengths. For the widely used image/video entropy codes, it is worthwhile studying the code structure and attempting to extract useful information from it. The other part of the work in this thesis is devoted to the study of the code structures of the image/video entropy codes, involving an attempt to extract useful code length information and thus simplify the decoder architecture for these entropy decoders.

1.3 THESIS OUTLINE

There are five chapters in this thesis. The first chapter consists of an introduction and provides the background and motivation behind the thesis. The last chapter consists of a brief summary of the entire work. The main work of this thesis is described in chapters two, three and four, respectively.

In chapter two, we focus on the efficient entropy encoding of particular sources that are commonly found in modeling image and video data. In this chapter, we introduce a general concept which summarizes one type of image/video entropy codes, and then different variations of this concept are introduced and discussed.

Chapter three introduces a coding method developed on the basis of the coding concept introduced in chapter two. Some applications of the coding method are then shown and its advantages and disadvantages are discussed.

Chapter four of this thesis focuses on the decoder architecture built on the coding method introduced in chapter three. The variations of the decoders in accommodating different image/video entropy code sets are discussed and applications of such decoders are also shown. The advantages and disadvantages of such decoders are also discussed in the chapter.

Chapter five is a brief summary of the thesis and suggestions are also made concerning several possible future continuations of the thesis work.

2 UNARY-PREFIXED CODES

The starting point for the study of the entropy coding of the typical sources in image/video coding systems is with the existing codes used in the coding of these sources. As mentioned in the introduction, these source probability distributions, such as Laplacian, generalized Gaussian, Cauchy etc., are all of similar shapes, i.e., all with high peaks and heavy tails. Therefore the optimal or nearly optimal entropy codes for these sources, also share some common properties. In this chapter, we study the optimal and nearly optimal codes of some typical probability distributions and summarize the entropy codes of these sources under the common name: "Unary-prefixed Codes" (UPC). Based on the study of previous work, we propose a new type of UPC as well as an adaptive coding algorithm for these sources, the resulting codes from the adaptive algorithm could also belong to the UPC family.

In this chapter, we first introduce the existing UPCs. Then we introduce the new UPC and the adaptive coding algorithm proposed. While introducing the adaptive algorithm, several possible coding strategies are discussed, which result in code sets with different properties. Finally, we present the applications of different UPCs.

2.1 THE EXISTING UPCS

2.1.1 Run-Length Encodings

Consider repeatedly performing a success-failure experiment having a probability of success $1-\theta$, $(0<\theta<1)$ until the first success appears. For example, flipping a coin (with the probability of getting head to be $1-\theta$) until you get a head, or receiving a binary sequence bit by bit (with probability of getting "1" to be $1-\theta$) till you get a "1". Let random variable *X* denote the number of failures until a success appears, then the probability distribution of *X* can be given by:

$$P(X=k)=\theta^k(1-\theta), \quad k=0,1,2,3,4\cdots \tag{2.1}$$

Such a discrete probability distribution is called a geometric distribution and the random variable *X* here has an infinite positive integer sample space: $\{0,1,2,3,4,\cdots\cdots\}$.

Now let us consider the entropy coding of an integer source with the geometric probability distribution given in Eq.(2.1). It is well known that by applying the Huffman coding algorithm, we are able to encode the letters of a finite source alphabet into Huffman codes [1], which are uniquely decipherable codes with minimum expected codeword length. However, for an integer source of the geometric distribution, the alphabet is infinite and the Huffman algorithm cannot be applied directly. This is due to the fact that the Huffman algorithm requires the encoding to start by 'merging' the least probable letters in the alphabet.

S.W. Golomb initiated the early work [2] in coding infinite alphabets of non-negative integer sources, which follow the geometric distribution in Eq.(2.1), into optimal codes. He named the random variable *X* as "the *run lengths* between successive unfavorable events" and studied the case when θ satisfies $\theta^m = \frac{1}{2}$, where *m* is some positive integer. Under such conditions, θ could only take values in the set: $\{\sqrt[1]{\frac{1}{2}},\ \sqrt[2]{\frac{1}{2}},\ \sqrt[3]{\frac{1}{2}},\ \sqrt[4]{\frac{1}{2}} \cdots\}$.

Since we have $\theta^m = \frac{1}{2}$, then the probability of the run length $n+m$ is:

$$\begin{aligned} P(X = n+m) &= \theta^{n+m}(1-\theta) \\ &= \tfrac{1}{2}\theta^n(1-\theta) \\ &= \tfrac{1}{2}P(X = n) \end{aligned} \tag{2.2}$$

This means that a run length $n+m$ occurs with a probability of exactly one half of run length *n*. Suppose a run length *n* is coded using a binary code of *l*-bit, then it is obviously very reasonable to encode a run length $n+m$ using a binary code of length ($l+1$). Intuitively, every *m* codeword, apart from the initial few, should have the same code length. Golomb has pointed out that, this argument, though not rigorous, leads to the correct conclusion that for geometric distributions with $\theta^m = \frac{1}{2}$, the optimal code set should include *m* codewords of each possible code length, except for the shortest code lengths, which are not used at all if $m > 1$, and possibly one transitional code length, which is used fewer than *m* times. This argument, as also indicated by Golomb, could easily be verified by mathematical induction.

In general, let *k* be the smallest integer satisfying $2^k \ge 2m$, then we have exactly *m* codes for each code length longer than *k*. There are $2^{k-1} - m$ codewords for code length $k-1$.

A quick proof of this argument would be as follows. According to the Kraft inequality [3], for prefix codes, codewords with length *n* occupy $1/2^n$ of the total leaves of the binary code tree. Therefore for the above allocation of the code lengths, all codewords with length longer than *k* bits occupy $m/2^{k-1}$ of the total leaves. This is because:

$$\frac{m}{2^k} + \frac{m}{2^{k+1}} + \frac{m}{2^{k+2}} + \frac{m}{2^{k+3}} + \cdots = \frac{m}{2^{k-1}}$$

Therefore, the rest of the codes must be occupying proportionally:

$$1 - \frac{m}{2^{k-1}} = \frac{2^{k-1} - m}{2^{k-1}}$$

of the total leaves. Thus, it follows that, the number of codewords with length $k-1$ must be $2^{k-1} - m$.

When m is a power of 2, i.e., $m = 2^{k-1}$, we have $2^{k-1} - m = 0$. Thus there are no codewords with length $k-1$ and every code length will have exactly m codewords.

For instance, if we have $m = 4$, then $\theta^4 = \frac{1}{2}$, the run length codes will appear as shown below:

N	$\theta^n(1-\theta)$	**Run Length Codes**
0	0.151	000
1	0.128	001
2	0.109	010
3	0.092	011
4	0.078	1000
5	0.066	1001
6	0.056	1010
7	0.048	1011
8	0.040	11000
9	0.034	11001
10	0.029	11010

Table 2-1(a) Run length codes with $m = 4$

However for $m = 3$, i.e., $\theta^3 = \frac{1}{2}$, the run length code will be:

N	$\theta^n(1-\theta)$	**Run Length Codes**
0	0.206	00
1	0.164	010
2	0.130	011
3	0.103	100
4	0.081	1010
5	0.064	1011
6	0.051	1100
7	0.041	11010
8	0.032	11011
9	0.026	11100
10	0.021	111010
…	…	…

Table 2-1(b) Run length codes with $m = 3$

Note that in Table 2-1(a), the shortest code length has four codewords, which is equal to *m*; whereas in Table 2-1(b), the shortest code length has one codeword, which is not equal to *m*.

Now we have discussed the case when $m = -\log 2/\log\theta$ is an integer. However, in most cases, $-\log 2/\log\theta$ is not an integer. Under such circumstances, the number of codewords having the same code lengths will then oscillate between $\lfloor m \rfloor$ and $\lfloor m \rfloor + 1$. Golomb pointed out that, when *m* is very big, θ approaches 1, and it would be possible to choose an integer closest to *m* and still perform run length encoding; which will not lead to a bad result.

If we look closely at the run length codes, it is not difficult to find out that, starting from the very first codeword; every *m* codewords in the run length code set contain exactly the same leading bits. For instance, in Table 2-1(b), when $m = 3$, the first three codewords have the same leading bit “0”, the second three codewords have the same leading bits “10”, the third three codewords have the same leading bits “110” and so on. In fact, every codeword in a run length code set can be expressed as the concatenation of the common leading bits in an *m*-codeword group and some binary codes.

Let us now investigate this interesting property from another approach by looking at the case when $m = 1$. The following table shows the run length code when $m = 1$.

n	$\theta^n(1-\theta)$	**Run Length Codes**
0	1/2	0
1	1/4	10
2	1/8	110
3	1/16	1110
4	1/32	11110
5	1/64	111110
6	1/128	1111110
7	1/256	11111110
8	1/512	111111110
9	1/1024	1111111110
10	1/2048	11111111110

Table 2-2 The run length code when $m = 1$

When $\theta^k = \frac{1}{2}$, the sum of every *k*-codeword group will have a probability distribution as shown in Table 2-2. This is easily verifiable since the sum of the first *k* probabilities is:

$$\sum_{i=0}^{k-1} \theta^i (1-\theta) = 1 - \theta^k = \frac{1}{2} \tag{2.3}$$

And therefore the sum of the *j*-th group of *k* probabilities is $1/2^j$

For the distribution in Table 2-2, we can see that, every codeword is a unary code of the integer *n* plus a “0”. We can simply call it a unary prefix since the bit “0” exists for every codeword. This unary prefix is exactly the common leading bits we have talked about. Then it is obvious that for $\theta^m = \frac{1}{2}$, the run length code can be expressed as a unary prefix plus a $\lfloor \log_2 m \rfloor$-bit or $\lfloor \log_2 m \rfloor + 1$-bit suffix.

2.1.2 The Golomb Rice codes

Until now, in the run length encodings, we have been discussing the situation when $\theta^m = \frac{1}{2}$, where *m* is an integer. Under such conditions, Golomb has proved that the run length codes are optimal for the geometric distribution in Eq.(2.1). Golomb has indicated that in most cases, θ cannot satisfy this condition, but run length coding strategy could still be used. Gallager and Van Voorhis [4] generalized Golomb's idea to the entire interval when $0 < \theta < 1$ and proved that optimal code exists for any probability distribution with $0 < \theta < 1$.

Gallager and Voorhis pointed out that, the run length codes are not only optimal for $\theta^m = \frac{1}{2}$, but also optimal for any θ that satisfies:

$$\theta^m + \theta^{m+1} \le 1 \le \theta^m + \theta^{m-1} \tag{2.4}$$

It is obvious that, for any θ satisfying $0 < \theta < 1$, there exists a unique *m* such that the inequality (2.4) is satisfied. Therefore, Gallager and Voorhis's result indicates that for $0 < \theta < 1$, optimal codes can be constructed using Golomb's run length encoding algorithm.

Now let us look at a particular θ such that $0 < \theta < 1$. From inequality(2.4), we could find out the corresponding integer *m*. For this specific θ and *m*, we define a discrete source that has $n+m+1$ symbols, and has a probability distribution given by:

$$P_n(k) = \begin{cases} (1-\theta)\theta^k, & 0 \le k \le n \\ \dfrac{(1-\theta)\theta^k}{1-\theta^m}, & n < k \le n+m. \end{cases} \tag{2.5}$$

Here *n* can be any integer. In fact, the last *m* probability values in such a discrete source can be considered to be the sum of all probability values in Eq.(2.1) with $k > m$. That is:

$$\frac{(1-\theta)\theta^k}{1-\theta^m} = \sum_{j=0}^{\infty}(1-\theta)\theta^{k+jm}. \tag{2.6}$$

Now let us consider the optimal coding of this discrete source with $n+m+1$ symbols. The first $n+1$ symbols of this discrete source have probability values that decrease as *n* increases; similarly, the last *m* symbols also have decreasing probability values. Therefore we know that, the ($n+m$)-th probability value is smaller or equal to the ($n-1$)-th probability value:

$$\frac{(1-\theta)\theta^{n+m}}{1-\theta^m} \le (1-\theta)\theta^{n-1}. \tag{2.7}$$

Whereas the ($n+m-1$)-th probability value is bigger than the *n*-th probability value:

$$\frac{(1-\theta)\theta^{n+m-1}}{1-\theta^m} > (1-\theta)\theta^n. \tag{2.8}$$

Eq.(2.7) can be derived from the left hand side of Eq.(2.4), and Eq.(2.8) can be derived from the right hand side of Eq.(2.4). Thus we can conclude that the ($n+m$)-th probability value and the *n*-th probability value are the two smallest probability values in the probability sequence. As we know that the Huffman coding algorithm is initiated by merging the two smallest probability values, therefore the ($n+m$)-th symbol and the *n*-th symbol will be merged first, and the probability value after merging will be $(1-\theta)\theta^n / 1-\theta^m$. Now we assign "1" to the ($n+m$)-th symbol and "0" to the *n*-th symbol. The resulting probability distribution becomes a discrete source in the form of Eq.(2.5), with now *n* becomes $n-1$. Following the above steps, we can continue our encoding until $n=0$. Finally the discrete source becomes:

$$P_{-1}(k) = \frac{(1-\theta)\theta^{n+m-1}}{1-\theta^m}, \qquad 0 \le k \le m-1 \tag{2.9}$$

Now from Eq.(2.4), we know that in the probability distribution defined by Eq.(2.9), the sum of the two smallest probability values exceeds the biggest probability value. Therefore the optimal code for such distribution can vary by only one bit in length. Then for $k < 2^{\lfloor \log_2 m \rfloor + 1} - m$ in Eq.(2.9), the code length would be $\lfloor \log_2 m \rfloor$, and the rest of code would be of length $\lfloor \log_2 m \rfloor + 1$. Now for every $k \le n$, the optimal code could be considered to be the optimal code of *k* mod *m* concatenated with the unary code of $\lfloor k/m \rfloor$. And as *n* can be any integer, we can conclude that this is the optimal encoding for the geometric distribution.

Thereupon, we can summarize the above encoding algorithm as follows. Express the source integer *k* of a geometric distribution using a quotient *j* and reminder *r*:

$$k = mj + r \tag{2.10}$$

where *m* satisfies Eq.(2.4), then the optimal code for the geometric distribution can be constructed using the unary expression of *j* plus the Huffman code of *r*, and the length of the Huffman code is $\lfloor \log_2 m \rfloor$ or $\lfloor \log_2 m \rfloor + 1$.

By studying some special but representative cases, Rice [5] proposed one type of sub-optimal codes for the geometric distribution in Eq.(2.1). This type of code, which was latterly referred to as the Golomb Rice (GR) code, is highly structured and has found a variety of applications in many coding systems such as the coding of Laplacian distributed prediction errors in lossless image coding algorithms [6].

The special case studied by Rice involved *m* being a power of 2, i.e. $m = 2^k$. Under this condition, the run length code becomes a unary code for *j* plus a fixed *k*-

bit length code. The *k*-bit suffix of the codeword represents one of the reminders in the interval $[0, 2^k - 1]$. For instance, when $k = 2$, the integer 9 will be coded as 11001. From Gallager and Voorhis's analysis, it is obvious that the GR codes works optimally only when $\theta^{2^k} = \frac{1}{2}$ and if we are to apply the GR codes for any $0 < \theta < 1$, it will not always be possible to achieve optimality. However, the GR codes are able to perform almost optimally for all $0 < \theta < 1$. Its advantage is its simplicity of structure which makes it easy to construct and decode.

Table 2-3 gives an example of the GR codes.

n	**Unary Prefix**	**Suffix**	**Length**
0	0	0	2
1	0	1	2
2	10	0	3
3	10	1	3
4	110	0	4
5	110	1	4
6	1110	0	5
7	1110	1	5
8	11110	0	6
9	11110	1	6
10	111110	0	7
11	111110	1	7
12	1111110	0	8
…	…	…	…

Table 2-3 GR code (*k*=1)

The GR code can also be shown in a code tree format, as Figure 2-1 demonstrates. Figure 2-1 shows a GR code tree with suffix length one, which is an exact set of unary codes.

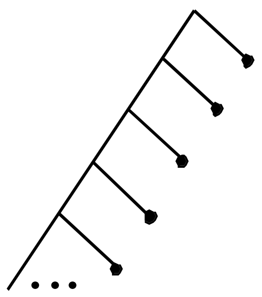


Figure 2-1 GR code (k=1)

2.1.3 The Exponential-Golomb codes

Although it is not possible for the GR codes to achieve optimality in most cases, they have been shown to be applicable in the coding of geometric distributions and have been found to be nearly optimal for geometric distributions and sources associated with the Laplacian distributions. For the GR code, every code length has exactly 2^k codewords. This matches the geometric distribution or Laplacian distributions reasonably well because the geometric distribution "decays" at some constant exponential rate. In many real-world coding systems, however, the probability distributions with higher peaks and heavier tails are usually found to better fit empirical data models. For instance, the Generalized Gaussian family with given source parameters, the Cauchy distributions, and so on, are all shapes with higher peaks and heavier tails. Such distributions and the sources associated with them no longer have constant "decay" rates, on the other hand, the "decay" rate of the distribution functions are usually steep for bigger density values, and flat for smaller density values. Thus to encode such sources, it is more reasonable to consider codes that have fewer codewords of shorter code lengths and more codewords of longer code lengths.

Bearing such concerns in mind, Teuhola [7] proposed another type of code, attempting to provide better matches for these high peak and heavy tail distributions. The code is called an Exponential-Golomb (EG) code. The EG code, in contrast to the GR codes, has an exponentially increasing number of codewords for each code length.

The EG codes could also be viewed as a unary prefix concatenated with a fixed length suffix, where only the length of the suffix is no longer fixed for all prefix lengths. In contrast to the GR codes, the EG codes have longer suffix lengths for longer prefix lengths, shorter suffix lengths for shorter prefixes. Such suffix structures enable more codewords for longer code lengths. The suffix of the EG code could be further separated into two parts, one part associated with the unary prefix where its length is fixed once the prefix length is fixed and the other part is of arbitrary length *k*. The group of codes that have the same prefix are of the same code length, so we could group these codes using an index *j*, where *j* equals 0, 1, 2, and so on. For the *j*-th code group, the prefix length is actually $j+1$, the part of the suffix that is associated with the prefix is one bit shorter than the prefix and therefore it is *j* bits; the arbitrary part of the suffix is *k* bit long. Therefore for an EG code, the prefix is of $j+1$ bits, the suffix is of $j+k$ bits, and each EG code yields a length of $2j+k+1$ bits. Table 2-4 shows the EG code with $k=0$.

n	EG code	Length
0	0	1
1	100	3
2	101	3
3	11000	5
4	11001	5
5	11010	5
6	11011	5
7	1110000	7
8	1110001	7
9	1110010	7
10	1110011	7
11	1110100	7
12	1110101	7
…	…	…

Table 2-4 EG code with *k*=0

From Table 2-4, we can see that, the *j*-th "group" of codes has the same code length $2j+k+1$, and there are in total 2^{j+k} codewords in the group. It is obvious that the number of the codewords in the group increases exponentially with the group number *j*.

In a similar manner to the GR codes, we can also represent the EG codes by a code tree.

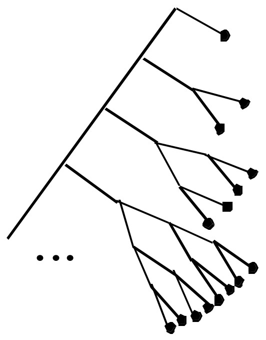


Figure 2-2 EG code (k=0)

The EG code is actually a special case of the Elias code [8]. As Teuhola mentioned, the EG code cannot perform optimally for any distribution, however, it works reasonably well for almost all exponential distribution, in general, by carefully selecting the suffix length. Moreover, the construction of the EG codes is also very simple, which makes it very practical for applications.

2.2 THE HYBRID GOLOMB CODE

On the basis of the GR codes and the EG codes, we proposed another type of code which we called Hybrid Golomb codes in [9]. The HG codes are hybrids of the GR codes and EG codes, which also perform sub-optimally for exponentially distributed sources.

From the previous sections we know that for the GR codes with *k*-bit suffixes, every code length group has 2^k codewords; whereas for the EG codes with *k*-bit arbitrary suffixes, every code length has 2^{j+k} (where j is the group number). Both the GR and EG codes are constructed using a unary prefix and a fixed length suffix. For the GR codes, with a fixed number of codewords for each code length group, the codeword length increases linearly as the probability value decreases; for the EG codes, on the other hand, the codeword length increases exponentially as the probability value decreases. Such properties of GR and EG codes make them suitable for application to slightly different sources. As we have discussed in previous sections, GR codes are more suitable for application in those sources with a fixed exponential decay rate, and EG codes are more suitable for sources with higher peaks and heavier tails. However, in many practical application situations, the sources are usually not fixed but vary under different situations. For example, in image and video coding, the image and video data may change significantly due to the nature of the image or video. Therefore, it may be more practical to design one type of code that generally works in a satisfactory manner. The HG codes are designed with such concerns in mind.

In constructing the HG codes, we also assign a unary prefix (or a group number) to each codeword, however the suffix will be a hybrid of the GR suffix and the EG suffix. For the codewords with lengths $2j+k$, $j>1$, the number of codewords increases exponentially for the same length, which is $2^{j+k-1}-1$ for codeword group *j*; however for the codewords with lengths $2j+k+1$, $j>2$, the number of codewords remain the same as for 2^{k+1}. For the initial two codeword groups ($j=0$ and $j=1$), the number of codewords are kept at 2^k. Thus, the code length integrated the properties of both the GR code and the EG code. Table 2-5 gives a comparison of the three types of codes with $k=0$.

n	GR	Length	EG	Length	HG	Length
0	0	1	0	1	0	1
1	10	2	100	3	10	2
2	110	3	101	3	1100	4
3	1110	4	11000	5	11010	5
4	11110	5	11001	5	11011	5
5	111110	6	11010	5	111000	6
6	1111110	7	11011	5	111001	6
7	1111110	8	1110000	7	111010	6
8	11111110	9	1110001	7	1110110	7
9	111111110	10	1110010	7	1110111	7
10	1111111110	11	1110011	7	11110000	8
…	…	…	…	…	…	…

Table 2-5 Comparison of GR, EG and HG codes

Again, we can also express the HG codes by a code tree.

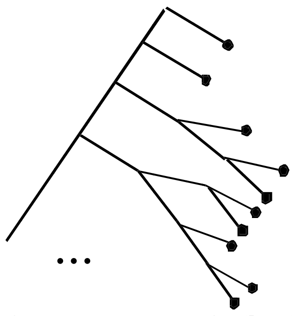


Figure 2-3 HG code (*k*=0)

With integrated properties from GR and EG, the HG codes works efficiently for a wide variety of sources with high peak and heavy tail distributions and the performances are comparable to both GR and EG codes.

We applied the HG codes to the quantized generalized Gaussian sources with given source parameters (υ and δ / σ) suitable for use in the modeling of image and video data. The generalized Gaussian sources are found to provide good models for image and video data. We will discuss these sources in detail in later sections. A comparison study is performed for HG codes, GR codes and EG codes.

The comparison results in a wide range of source parameters are shown in Figure 2-4 through to Figure 2-8. We see here that the HG codes perform robustly well for these different sources. Although it is noticeable that the HG codes are more comparable to the EG codes with *k* equal to 0, in fact they outperform the EG codes for the source parameters we studied.

Figure 2-9 shows the efficiency difference between HG codes and EG codes with *k*=0 since the performance of these two sets of codes are very close to each other and both appear to be the most robust codes. The efficiency difference is calculated as:

$$D(\upsilon)=\frac{\int_a^b \eta_{HG}(\upsilon,\delta/\sigma)d(\delta/\sigma)-\int_a^b \eta_{EG}(\upsilon,\delta/\sigma)d(\delta/\sigma)}{\int_a^b \eta_{EG}(\upsilon,\delta/\sigma)d(\delta/\sigma)} \tag{2.11}$$

where $D(\upsilon)$ is the efficiency difference, which is a function of the source parameter υ, $\eta_{HG}(\upsilon,\delta/\sigma)$ and $\eta_{EG}(\upsilon,\delta/\sigma)$ are the efficiencies of the sets of HG codes and EG codes that are both functions of source parameters υ and δ/σ and (a,b) is the range of the integration. We see here that the HG codes perform better than EG codes for the source parameters we have studied.

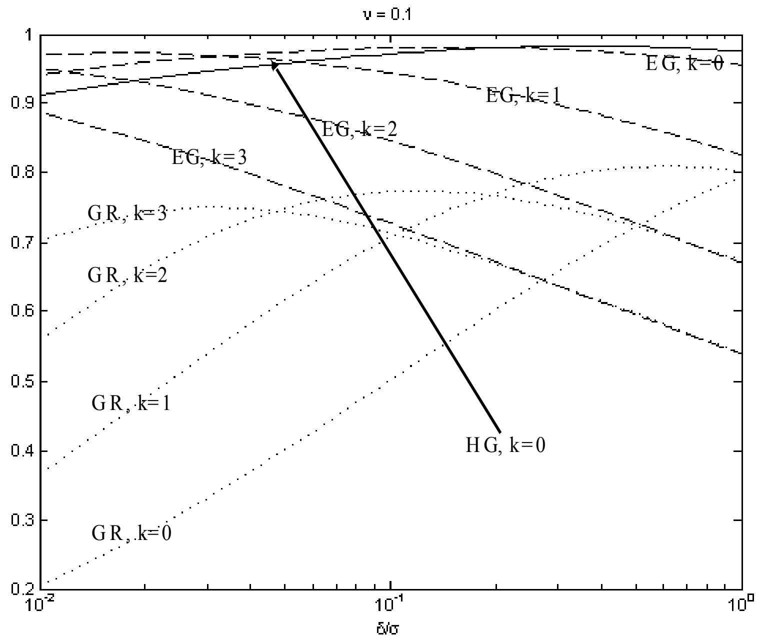


Figure 2-4 Comparison of coding efficiencies of HG, GR and EG codes for quantized GG sources with $\upsilon = 0.1$

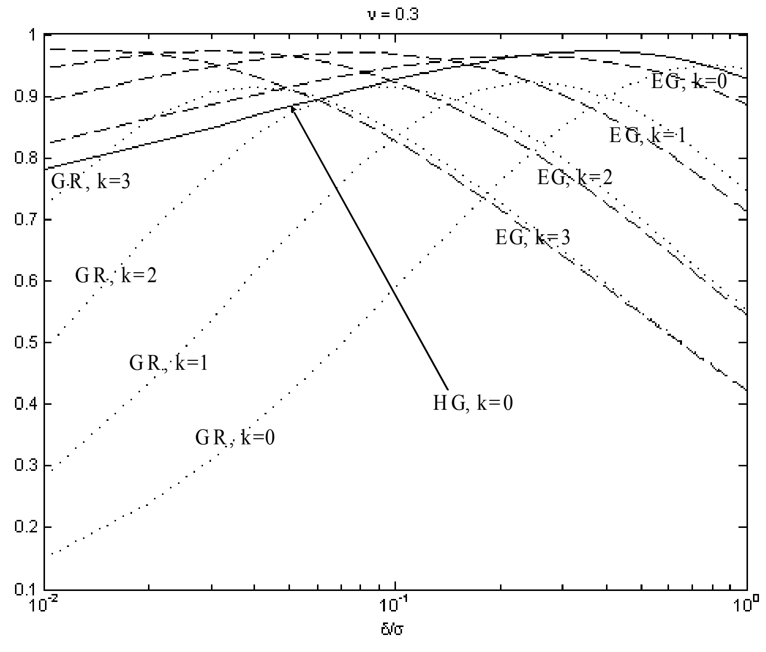


Figure 2-5 Comparison of coding efficiencies of HG, GR and EG codes for quantized GG sources with $\upsilon = 0.3$

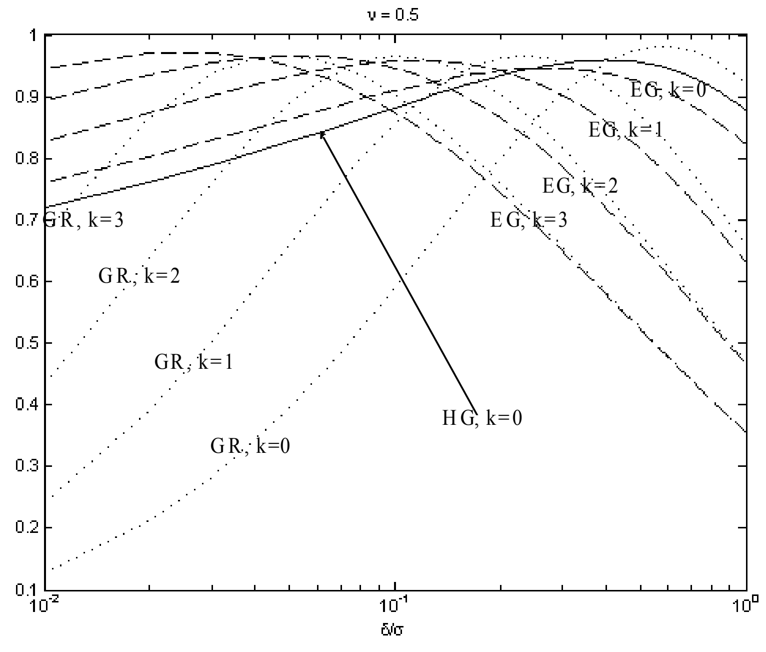


Figure 2-6 Comparison of coding efficiencies of HG, GR and EG codes for quantized GG sources with $\upsilon = 0.5$

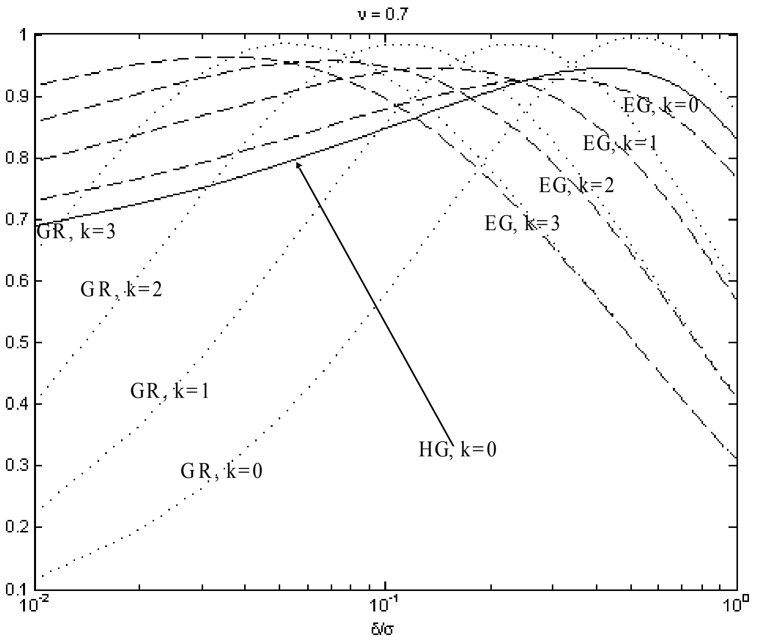


Figure 2-7 Comparison of coding efficiencies of HG, GR and EG codes for quantized GG sources with $\upsilon = 0.7$

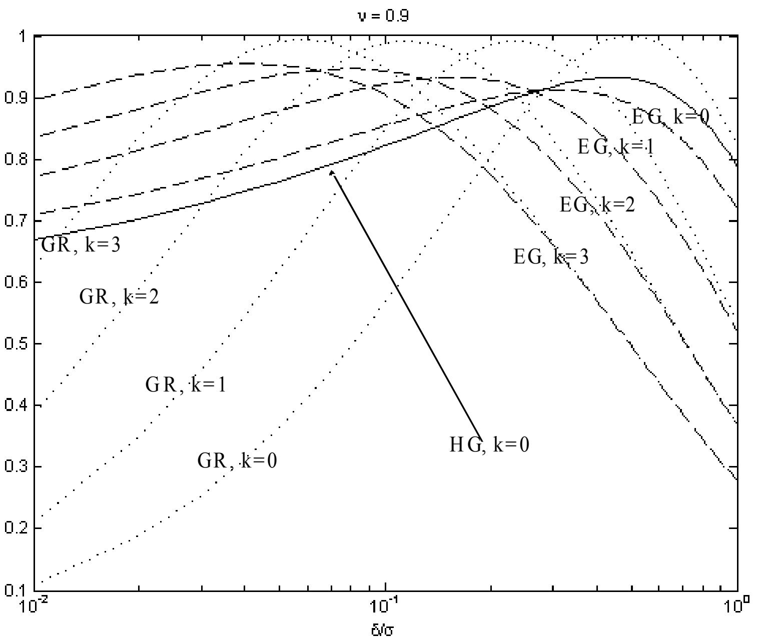


Figure 2-8 Comparison of coding efficiencies of HG, GR and EG codes for quantized GG sources with $\upsilon = 0.9$

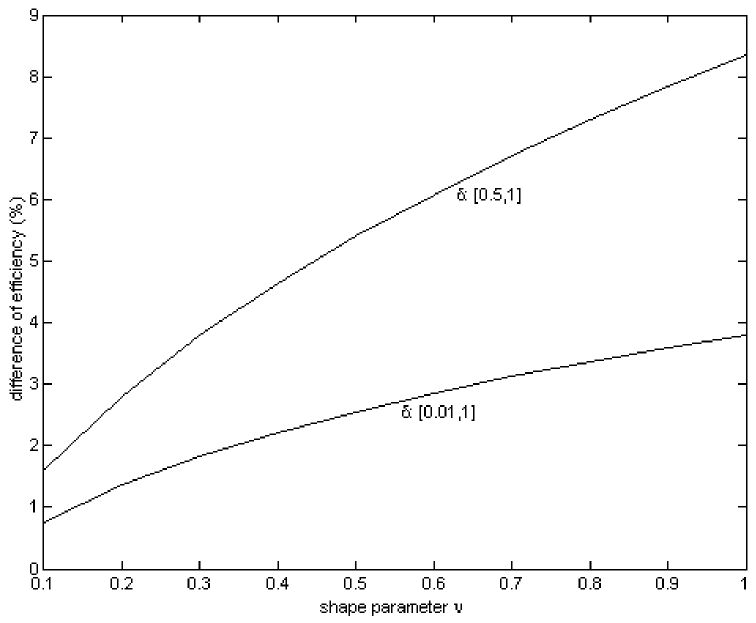


Figure 2-9 Efficiency difference between HG codes and EG codes (*k*=0)

2.3 THE CONCEPT OF UPC

2.3.1 General concept

In the previous sections, we have discussed the GR codes, EG codes and HG codes. These codes are all optimal or nearly optimal codes for sources with high peaks and heavy tails. We have also seen that, the constructions of these codes all involve concatenations of unary prefixes and suffixes. Each suffix can be of a fixed or variable length. It is obvious that these codes are of similar structures. We therefore give a common name to these codes: the Unary-Prefixed Codes.

By the term Unary-Prefixed Codes (UPC), we mean any code that is constructed by the concatenation of a unary prefix and a suffix. The suffix can be a fixed-length code, or can be any form of variable length code. As we will show in the following sections, the GR, EG and the HG codes are not the only codes which belong to this category as many different codes could be designed with such a structure. These differ from the Huffman coding algorithm as all UPCs could be constructed using a top-down approach, which does not require coding to start from the least probably source symbols and therefore the UPCs are suitable for application to sources with infinite alphabets. Moreover, the unary prefixes of UPCs provide good matches for exponentially shaped distributions (high peak and heavy tail distributions). Therefore in the coding of infinite sources with geometric distributions, or sources associated with the Laplacian, generalized Gaussian, Cauchy distributions etc, applying the UPCs is reasonable and natural.

To fully reveal the reason why the UPCs are efficient and suitable for use with these high peak and heavy-tailed distributions, some analysis must be conducted.

2.3.2 The optimality of the unary prefixes

As we have described in the previous sections, each UPC consists of a unary prefix and a binary suffix. To understand why the UPCs are able to provide a reasonably efficient compression, we firstly need understand the reason why and how well the unary prefixes are able to match an infinite discrete probability distribution. This section will show that, for any infinite discrete source, it is possible to segment the discrete probability distribution associated with it and that the result of such segmentation is able to be optimally encoded using unary codes.

Suppose we have a probability distribution $\{p_k\}_{k=1}^{\infty}$. Let us always assume that $\{p_k\}_{k=1}^{\infty}$ is a decreasing sequence, which it usually is for any reasonable applications. A segmentation of this probability distribution results in subsets of probabilities $\{P_k\}_{k=1}^{\infty}$, where

$$P_k = \{p_{s_{k-1}+1}, p_{s_{k-1}+2}, \dots, p_{s_k}\}.$$

Let us denote the summations of each subset by $\{S_k\}_{k=1}^{\infty}$, where

$$S_k = \sum\nolimits_{i=s_{k-1}+1}^{s_k} p_i$$

No matter how we segment this probability distribution, the result of these summations $\{S_k\}_{k=1}^{\infty}$ is a new probability distribution, since the summation of all S_k is also one. Now let us perform the segmentation by finding s_k in the following manner:

1) Start with $k = 0$ and $s_0 = 0$
2) For the current k, let

$$S_{-k} = \sum_{i=s_k+1}^{\infty} p_i$$

3) Find s_{k+1} such that:

$$\left| \sum_{i=s_k+1}^{s_{k+1}} \frac{p_i}{S_{-k}} - \frac{1}{2} \right|$$

is minimized

4) Let $k = k + 1$ and repeat from step 2).

This iterated process could also be described as the following. Suppose that we have already obtained the first *k* segments. In finding the next segment, we first normalize the set of probabilities that are left:

$$\overline{P}_k = \{\overline{p}_{k,1} = \frac{p_{s_k+1}}{S_{-k}}, \overline{p}_{k,2} = \frac{p_{s_k+2}}{S_{-k}}, \cdots \overline{p}_{k,j} = \frac{p_{s_k+j}}{S_{-k}}, \cdots\}$$

Then we find the index j such that the summation from $\overline{p}_{k,1}$ to $\overline{p}_{k,j}$ will be closest to $1/2$, and we let $s_{k+1} = s_k + j$. In another words, each time we are attempting to achieve a subset of the remaining probabilities, such that the summation of the probabilities in the subset will be as close to one half of the remaining total as possible.

It is easy to see that this segmentation process results in a new probability distribution $\{S_k\}_{k=1}^{\infty}$ that is very close to $\{1/2^k\}_{k=1}^{\infty}$. It is obvious that for the probability distribution $\{1/2^k\}_{k=1}^{\infty}$, unary codes are optimal. So now the question is, are the unary codes also optimal for the new probability distribution $\{S_k\}_{k=1}^{\infty}$? To verify this, we have the following lemma and theorem.

Lemma 1: For any infinite source with probability distribution given by $Q=\{q_1,q_2,q_3,\ldots\}$ with $H(Q)<\infty$, let $S_n^Q=\sum_{k=n}^{\infty} q_k$. Unary codes are optimal for sources that satisfy the following condition:

$$q_{n-2} > S_n^Q \quad \text{for any} \quad n \geq 3 \tag{2.12}$$

Proof: It is obvious that, under the above hypothesis, for any truncated probability distribution

$$Q_n=\{q_1,q_2,\ldots,q_{n-1},S_n^Q\}, \quad n\geq 3;$$

The Huffman codes are equivalent to the unary codes. Then from the result in [12], we can conclude that the optimal codes for the probability distribution satisfying (2.12) converge to unary codes.□

Theorem 1: For the segmentation $\{P_k\}_{k=1}^{\infty}$ of a probability distribution $P=\{p_k\}_{k=1}^{\infty}$ with $H(P)<\infty$ described above, unary codes are optimal for the probability distribution given by the summations $S=\{S_k\}_{k=1}^{\infty}$.

Proof: Let $H(S)=-\sum_{k=1}^{\infty} S_k \log S_k$, we will have:

$$\begin{aligned}
H(S) &= -\sum_{k=1}^{\infty} S_k \log S_k \\
&= -\sum_{k=1}^{\infty} (p_{s_{k-1}+1}+p_{s_{k-1}+2}+\cdots+p_{s_k})\log S_k \\
&< -\sum_{k=1}^{\infty} (p_{s_{k-1}+1}\log p_{s_{k-1}+1}+p_{s_{k-1}+2}\log p_{s_{k-1}+2}+\cdots+p_{s_k}\log p_{s_k}) \\
&= H(P)<\infty
\end{aligned}$$

As described in the segmentation process, s_k is decided by minimizing $\left|S_k/S_{-(k-1)}-\frac{1}{2}\right|$. If $S_k/S_{-(k-1)}\geq\frac{1}{2}$, we have:

$$\frac{S_k}{S_{-(k-1)}} \geq \frac{S_{k+1}}{S_{-(k-1)}} + \sum_{j=k+2}^{\infty}\frac{S_j}{S_{-(k-1)}}$$

$$\frac{S_k}{S_{-(k-1)}} > \sum_{j=k+2}^{\infty}\frac{S_j}{S_{-(k-1)}}$$

$$S_k > \sum_{j=k+2}^{\infty} S_j$$

If $S_k / S_{-(k-1)} < \frac{1}{2}$, we must have $(S_k + p_{s_k+1}) / S_{-(k-1)} > \frac{1}{2}$. Let $p_{s_k+1} = a + b$, such that:

$$\frac{S_k + a}{S_{-(k-1)}} = \frac{1}{2} \quad \text{and} \quad \frac{b + \sum_{j=k+1}^{\infty} S_j}{S_{-(k-1)}} = \frac{1}{2}$$

It is clear that $a \le b$, for otherwise $\left| S_k / S_{-(k-1)} - \frac{1}{2} \right|$ is not minimized. Therefore we have:

$$\frac{S_k}{S_{-(k-1)}} = \frac{1}{2} - \frac{a}{S_{-(k-1)}} \ge \frac{1}{2} - \frac{b}{S_{-(k-1)}} = \sum_{j=k+1}^{\infty} \frac{S_j}{S_{-(k-1)}}$$

$$\frac{S_k}{S_{-(k-1)}} \ge \sum_{j=k+2}^{\infty} \frac{S_j}{S_{-(k-1)}}$$

$$S_k \ge \sum_{j=k+2}^{\infty} S_j$$

Hence condition (2.12) is satisfied by $S = \{S_k\}_{k=1}^{\infty}$. Now we can apply Lemma 1 to S to show that the unary codes are indeed optimal.□

The above theorem shows that, by segmenting a countably infinite discrete probability distribution into proper probability subsets, it is possible for the summations of the subsets to be optimally coded by the unary codes. Therefore, properly assigning the unary prefixes to one probability distribution may lead us towards an efficient coding. If we are able to achieve an optimal suffix coding for each prefix group, optimal coding may be achieved. GR, EG and HG codes, although not grouping the probability distribution exactly as given for the five steps, actually results in a similar segmentation for the high-peaked and heavy-tailed distributions. This partially reveals the general reason for the efficiency of GR, EG and HG codes.

Now we have seen that for any countably infinite discrete distribution, we are able to assign optimal unary prefixes, the question remaining is how to make the suffix as efficient as possible?

2.3.3 The Unary-Prefixed Huffman coding algorithm

In this section, we propose a coding algorithm named the "Unary-Prefixed Huffman Coding" (UPH). The UPH is designed on the basis of the optimal prefix assignment discussed in the previous sub-section. We will show that the UPH algorithm guarantees an overall efficient encoding. Also, for geometric distributions, the UPH code becomes the codes described in [4], and is therefore optimal.

The basic idea of the UPH, is to firstly attempt to segment a probability distribution $\{p_k\}_{k=1}^{\infty}$ into subsets $\{P_k\}_{k=1}^{\infty}$, $P_k = \{p_{s_{k-1}+1}, p_{s_{k-1}+2}, \dots, p_{s_k}\}$, where the summation of each subset $\{S_k = \sum_{i=s_{k-1}+1}^{s_k} p_i\}_{k=1}^{\infty}$ should be as close to $\{1/2^k\}_{k=1}^{\infty}$ as possible. The segmentation process is exactly as described in the previous sub-section. Then within each probability subset P_k, a normalization process is applied, and the Huffman coding is performed. For each codeword within the subset P_k, the UPH code is expressed as the concatenation of a unary prefix for length *k* and the corresponding Huffman suffix within P_k.

By extending the four-step process described in the previous sub-section, the UPH algorithm is fully described by the following steps:

1) Start with $k = 0$, let $s_0 = 0$.
2) For the current value k, let

$$S_{-k} = \sum_{i=s_k+1}^{\infty} p_i \tag{2.13}$$

3) Find s_{k+1} such that the difference

$$\left| \sum_{i=s_k+1}^{s_{k+1}} \frac{p_i}{S_{-k}} - \frac{1}{2} \right| \tag{2.14}$$

is minimized

4) Let:

$$P_{k+1} = \{p_{s_k+1}, p_{s_k+2}, \cdots p_{s_{k+1}}\}$$
$$S_{k+1} = p_{s_k+1} + p_{s_k+2} + \cdots + p_{s_{k+1}}$$
$$\hat{P}_{k+1} = \{\frac{p_{s_k+1}}{S_{k+1}}, \frac{p_{s_k+2}}{S_{k+1}}, \dots, \frac{p_{s_{k+1}}}{S_{k+1}}\}$$

Note that now $\hat{P}_{k+1}$ is obtained by normalizing P_{k+1} into a discrete probability distribution. Performing a Huffman coding to the distribution given by $\hat{P}_{k+1}$; we obtain $s_{k+1} - s_k$ Huffman codes. In the future when we refer to the Huffman codes within P_{k+1}, we mean those codes obtained from $\hat{P}_{k+1}$. For each

Huffman code within P_{k+1}, we attach a unary prefix $\underbrace{111\cdots1}_{k}0$ (or equivalently $\underbrace{000\cdots0}_{k}1$) to it.

5) Let $k = k+1$ and repeat from step 2).

Let us take a simple example. Suppose we have an infinite probability distribution: $\{p_n = 1/(3\cdot 2^{\lceil n/3 \rceil})\}_{n=1}^{\infty}$, which looks like:

$$\{\tfrac{1}{6},\tfrac{1}{6},\tfrac{1}{6},\tfrac{1}{12},\tfrac{1}{12},\tfrac{1}{12},\tfrac{1}{24},\tfrac{1}{24},\tfrac{1}{24},\cdots\} \tag{2.15}$$

where each set of three probabilities can be summed to a value that is a power of $\frac{1}{2}$. Thus we have $\{S_k = 1/2^k\}_{k=1}^{\infty}$, and $\{P_k = \{1/(3\cdot 2^k), 1/(3\cdot 2^k), 1/(3\cdot 2^k)\}\}_{k=1}^{\infty}$. For this example, each P_k has three equivalent probability values. Therefore the Huffman codes within each P_k are: $\{1, 00, 01\}$ or $\{0, 10, 11\}$. The UPH code is then a concatenation of a unary code of k and one of the codes within $\{1, 00, 01\}$ or $\{0, 10, 11\}$.

The UPH algorithm is actually performing the encoding in two optimal steps, it can be proved that the code sets constructed by UPH have average code lengths upper bounded by $H(P)+2$.

Theorem 2: The UPH codes have average code lengths upper bounded by $H(P)+2$, given that the entropy $H(P)$ is finite.

Proof: Let us look at the result of UPH coding for the first *n* segmentations. We have *n* probability subsets $P_1, P_2, \cdots, P_n$. Let

$$P^n = P_1 \cup P_2 \cup \cdots \cup P_n$$

We normalize these *n* subsets $P_1, P_2, \cdots, P_n$:

$$\hat{P}_1 = \{\frac{p_1}{S_1}, \frac{p_2}{S_1}, \dots, \frac{p_{s_1}}{S_1}\}$$

$$\hat{P}_2 = \{\frac{p_{s_1+1}}{S_2}, \frac{p_{s_1+2}}{S_2}, \dots, \frac{p_{s_2}}{S_2}\}$$

$$\vdots$$

$$\hat{P}_n = \{\frac{p_{s_{n-1}+1}}{S_n}, \frac{p_{s_{n-1}+2}}{S_n}, \cdots, \frac{p_{s_n}}{S_n}\}$$

For each $P_k, 1 \le k \le n$, the partial entropy $H(P_k)$ is defined as:

$$H(P_k) = -\sum_{i=s_{k-1}+1}^{s_k} p_i \log p_i$$

Then for each $\hat{P}_k$, we have:

$$H\left(\hat{P}_k\right) = -\sum_{i=s_{k-1}+1}^{s_k} \frac{p_i}{S_k} \log \frac{p_i}{S_k}$$

$$= -\sum_{i=s_{k-1}+1}^{s_k} \frac{p_i}{S_k} \log p_i + \sum_{i=s_{k-1}+1}^{s_k} \frac{p_i}{S_k} \log S_k$$

$$= \frac{H(P_k)}{S_k} + \log S_k$$

The UPH performs Huffman coding on every probability segment P_k, $1 \le k \le n$. As Huffman coding is optimal and is upper-bounded by entropy+1, for each normalized probability set $\hat{P}_k$, we have:

$$\hat{L}_k \le H(\hat{P}_k) + 1 = \frac{H(P_k)}{S_k} + \log S_k + 1,$$

where $\hat{L}_k$ is the average code length of the Huffman code for each $\hat{P}_k$. For each set of Huffman codes in $\hat{P}_k$ we assign a *k*-bit prefix, therefore the average code length of UPH for the truncated probability set P is satisfied by:

$$\bar{L}^n_{UPH} = \sum_{k=1}^{n} S_k(\hat{L}_k + k) \le \sum_{k=1}^{n} S_k(\frac{H(P_k)}{S_k} + \log S_k + k) + \sum_{k=1}^{n} S_k$$

i.e.:

$$\bar{L}^n_{UPH} \le \sum_{k=1}^{n} \{H(P_k) + S_k(k + \log S_k)\} + (1 - S_n) \qquad (2.16)$$

From the formula given in Eq.(2.13), we know that the probabilities in P^n are summed as $\sum_{k=1}^{n} S_k = 1 - S_{-n}$. So we can normalize P^n as:

$$\hat{P}^n = \{\frac{p_1}{1-S_{-n}}, \frac{p_2}{1-S_{-n}}, \ldots, \frac{p_{s_n}}{1-S_{-n}}\}$$

We then have:

$$\begin{aligned} H(\hat{P}^n) &= -\sum_{i=1}^{s_n} \frac{p_i}{1-S_{-n}} \log \frac{p_i}{1-S_{-n}} \\ &= -\sum_{i=1}^{s_n} \frac{p_i}{1-S_{-n}} \log p_i - \sum_{i=1}^{s_n} \frac{p_i}{1-S_{-n}} \log(1-S_{-n}) \\ &= \sum_{k=1}^{n} \frac{H(P_k)}{1-S_{-n}} + \log(1-S_{-n}) \end{aligned}$$

Hence:

$$\sum_{k=1}^{n} H(P_k) = (1-S_{-n})\{H(\hat{P}^n) - \log(1-S_{-n})\}$$

Now we can rewrite the inequality (2.16) as:

$$\bar{L}_{UPH}^n \le (1-S_{-n})\{H(\hat{P}^n) - \log(1-S_{-n}) + 1\} + \sum_{k=1}^{n} S_k (k + \log S_k) \qquad (2.17)$$

As $n \to \infty$, $S_{-n} \to 0$, $H(\hat{P}^n) \to H(P)$, and $\bar{L}_{UPH}^n \to \bar{L}_{UPH}$, the average code length of UPH. It should also be recalled that $H(S)$ denotes the entropy of the source $\{S_k\}_{k=1}^{\infty}$, therefore as $n \to \infty$, the inequality (2.17) becomes:

$$\begin{aligned} \bar{L}_{UPH} &\le H(P) + 1 + \sum_{k=1}^{\infty} S_k (k + \log S_k) \\ &= H(P) + 1 + \{\sum_{k=1}^{\infty} k S_k - H(S)\} \\ &\le H(P) + 2 \end{aligned}$$

The last inequality is attempted because the unary codes are optimal for $\{S_k\}_{k=1}^{\infty}$ and $H(S) < \infty$. Therefore, when $H(P) < \infty$, the average code length of the UPH is bounded by $H(P) + 2$. The theorem is proved.□

Here we see that when we concatenate a unary code and a Huffman code, no matter to what source our codes are applied, the coding efficiency will never fall below $H(P) + 2$. It can be shown that, for geometric distribution, the UPH codes will become the codes proposed by Gallager and Voorhis [4], which are optimal.

Theorem 3: For the geometric distribution, the code set resulting from the UPH algorithm is equivalent to Gallager's code set in [4], which is optimal for geometric distributions.

Proof: Let us first recall that in [4], Gallager and Voorhis showed that, the run length codes are optimal for any geometric distribution with parameter θ that satisfies:

$$\theta^m + \theta^{m+1} \le 1 \le \theta^m + \theta^{m-1} \qquad (2.18)$$

It is obvious that, for any θ which satisfies $0<\theta<1$, there exists a unique m such that the inequality is satisfied. Therefore, their result indicates that for $0<\theta<1$, optimal codes can be constructed using Golomb's run length encoding algorithm.

Now to prove this theorem, we only need to show that for all k, $s_{k+1}-s_k=m$, where m is the integer satisfying the inequality (2.18). Since for every probability segment, both the proposed algorithm and Gallager's algorithm perform a Huffman coding on the segment. Hence if the segmentations of the two codes are the same, the two code sets are equivalent.

For consistency, let us shift the index of geometric distributions by 1, namely, let $\{p_i=\theta^{i-1}(1-\theta)\}_{i=1}^{\infty}$ be our geometric distribution with parameter θ. For $0\le k<\infty$, we have:

$$S_{-k}=\sum_{i=s_k+1}^{\infty}p_i=1-\sum_{i=1}^{s_k}p_i=\theta^{s_k} \tag{2.19}$$

and hence:

$$\begin{aligned}\overline{P}_k&=\{\frac{p_{s_k+1}}{\theta^{s_k}},\frac{p_{s_k+2}}{\theta^{s_k}},\frac{p_{s_k+3}}{\theta^{s_k}},\frac{p_{s_k+4}}{\theta^{s_k}},\cdots\}\\&=\{(1-\theta),(1-\theta)\theta,(1-\theta)\theta^2,(1-\theta)\theta^3,\cdots\}\\&=\{p_k\}_{k=1}^{\infty}\end{aligned}$$

This means that we have exactly the same pattern of data to work with for each of the iterations. Hence it is sufficient to study the case $k=0$, and show $s_1=m$. Now let $k=0$, from formula(2.14), we must have:

$$\left|\frac{1}{2}-(1-\theta^{s_1})\right|=\left|\frac{1}{2}-\sum_{i=1}^{s_1}p_i\right|\le\left|\frac{1}{2}-\sum_{i=1}^{s_1-1}p_i\right|=\left|\frac{1}{2}-(1-\theta^{s_1-1})\right| \tag{2.20}$$

and

$$\left|\frac{1}{2}-(1-\theta^{s_1})\right|=\left|\frac{1}{2}-\sum_{i=1}^{s_1}p_i\right|\le\left|\frac{1}{2}-\sum_{i=1}^{s_1+1}p_i\right|=\left|\frac{1}{2}-(1-\theta^{s_1+1})\right| \tag{2.21}$$

From(2.20), we have:

$$\begin{aligned}&(\theta^{s_1}-\tfrac{1}{2})^2\le(\theta^{s_1-1}-\tfrac{1}{2})^2\\&\theta^{s_1-1}-\theta^{s_1}\le\theta^{2s_1-2}-\theta^{2s_1}\\&\theta^{s_1}(\frac{1}{\theta}-1)\le\theta^{2s_1}(\frac{1}{\theta}-1)(\frac{1}{\theta}+1)\end{aligned}$$

As $0 < \theta < 1$, then $1 \le \theta^{s_1 - 1} + \theta^{s_1}$. Similarly, from (2.21) we find that $\theta^{s_1} + \theta^{s_1+1} \le 1$. Now there is a unique integer m satisfying the condition(2.18), thus $s_1 = m$. □

Until now, we have shown that the UPH algorithm is able to perform a two-step optimal encoding locally for any countably infinite discrete source. Although local optimality does not necessarily lead to global optimality, the UPH codes are able to achieve high efficiency despite source variations. In particular, the UPH codes could reach optimality for geometric distributions.

2.3.4 Modifying the UPH codes into codes with simpler structures

In the previous sub-section, we have seen that the construction of UPH codes could be summarized in two steps. The first involves dividing the probability distribution into subsets that could be optimally coded using a set of unary codes, and then secondly a Huffman coding is performed on each of these subsets. It is obvious that, unlike the GR, EG and HG codes, which all have code structures in closed forms, to construct the UPH codes requires a great deal of computation.

To simplify the UPH codes, we could relax the second step and use a set of pseudo-fixed length codes as suffixes for each unary prefix. By compromising the optimality of Huffman codes in the second step, we are able to greatly simplify the encoding procedure and still keep the optimality achieved from the first step. In the second coding step, we replace the Huffman coding by the pseudo-fixed length codes, which are constructed using the Huffman algorithm and by assuming that the probabilities in the probability subsets are equal. It is obvious that this is not true in most cases; however, since the Huffman codes for equal probability values are almost fixed length codes, this will result in a much simpler code structure.

The modified UPH algorithm could be described using similar steps as the UPH, by merely modifying the fourth step:

1) Start with $k = 0$, let $s_0 = 0$.
2) For the current value k, let

$$S_{-k} = \sum_{i=s_k+1}^{\infty} p_i$$

3) Find s_{k+1} such that the difference

$$\left| \sum_{i=s_k+1}^{s_k+1} \frac{p_i}{S_{-k}} - \frac{1}{2} \right|$$

is minimized

4) Let:

$$P_{k+1} = \{p_{s_k+1}, p_{s_k+2}, \cdots p_{s_{k+1}}\}$$

$$S_{k+1} = p_{s_k+1} + p_{s_k+2} + \cdots + p_{s_{k+1}}$$

For the probability set P_{k+1}, there are $n_k = s_{k+1} - s_k$ probability values. We assume that these n_k probabilities are equal to each other and then perform Huffman coding. The resulting codes will be binary codes either of length $\lfloor \log_2 n_k \rfloor$ or $\lfloor \log_2 n_k \rfloor + 1$. We then attach a common unary prefix of length $k+1$ to all these binary codes to complete the encoding of this segment.

5) Let $k = k+1$ and repeat from step 2).

In this modified UPH code, the suffix length differs by at most one bit, thus it is called the pseudo fixed length codes. While performing Huffman codes for n_k equal probabilities values, we only need assign fixed length codes with length $\lfloor \log_2 n_k \rfloor$ to the first $2^{\lfloor \log_2 n_k \rfloor + 1} - n_k$ probability values and fixed length codes with length $\lfloor \log_2 n_k \rfloor + 1$ to the remainder of the probability values, as previously mentioned. No actual Huffman encoding algorithm is required. Thus the coding process could be greatly simplified.

The proof for Theorem 3 will also show that, for geometric distributions, the modified UPH codes also becomes the optimal codes proposed by Gallager in [4]. The modified UPH coding process is exactly the same as that of Gallager's codes within each segment, while the segmentation of the modified UPH is the same as the original UPH algorithm.

2.4 THE APPLICATIONS OF THE UPCS

In the previous section, we introduced the UPCs. We have seen that, the UPCs are actually one type of general methods of coding. It generalizes the GR, EG and HG codes into a much wider concept. Moreover, by extending this concept, we are able to find a more general algorithm – the UPH algorithm. The UPH has been proven to be efficient in coding the countably infinite discrete sources. With proper modification of the UPH algorithm, we are also able to simplify the code construction. However, as has been mentioned previously, the UPCs are designed for sources with high peaks and heavy tails. Although we have found an upper bound of the coding efficiency for the UPH codes, this upper bound is not strong enough for us to conclude that the UPH algorithm is indeed always efficient for any infinite source. Now the focus shifts to the high-peaked, heavy-tailed distributions that are commonly seen in image/video coding systems in order to show that the UPCs in general provide good compression for such distributions.

In this section, the UPCs including the GR, EG, HG and the UPH codes are all applied to the quantized Generalized Gaussian (GG) distributions and comparisons of their coding efficiencies are made.

The GG probability density function (GG pdf) can be expressed using the following expression and parameters.

$$f_x(x) = c_1 \exp(-c_2 |x|^{\upsilon}) \tag{2.22}$$

where,

$$c_1 = \frac{\upsilon\eta(\sigma,\upsilon)}{2\Gamma(1/\upsilon)},\ c_2 = [\eta(\sigma,\upsilon)]^{\upsilon},\ \text{and}\ \eta(\sigma,\upsilon) = \frac{1}{\sigma}\left[\frac{\Gamma(3/\upsilon)}{\Gamma(1/\upsilon)}\right]^{1/2} \tag{2.23}$$

In Eq.(2.23), Γ is the gamma function.

The GG pdf is a function of υ and σ. Parameter υ is called the "shape parameter", and σ is the standard deviation. When $\upsilon = 1$, the generalized GG becomes a Laplacian distribution; when $\upsilon = 2$, the pdf becomes a Gaussian distribution and as $\upsilon \to \infty$, the distribution becomes a uniform distribution. Here we see that, the GG family includes a large variety of distributions when the shape parameter varies. All, however, have high peaks and heavy tails. For $\upsilon > 1$, the distribution "decays" more rapidly and thus has a thinner tail. For $\upsilon = 1$, the distribution "decays" at a constant exponential rate, and for $0 < \upsilon < 1$, the distribution "decays" more slowly and thus has a thicker, or heavier tail.

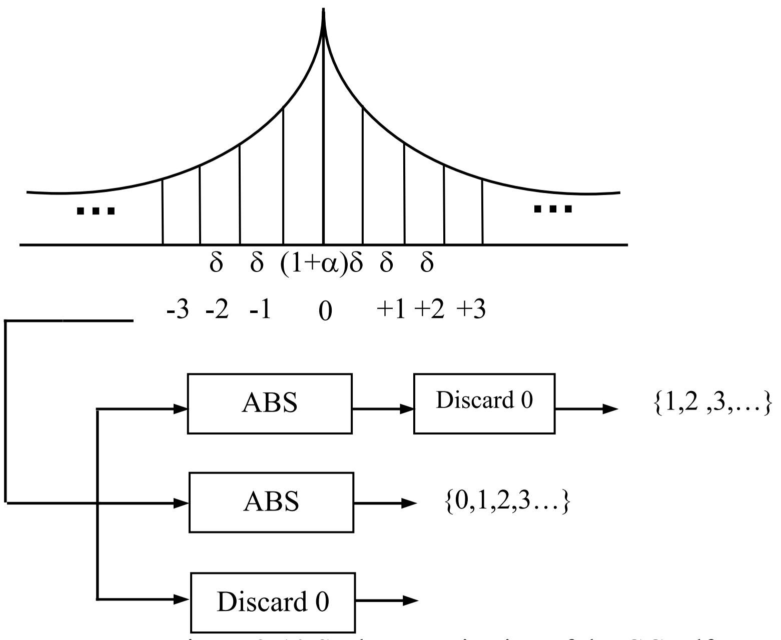


Figure 2-10 Scalar quantization of the GG pdf

The GG distributions are continuous distributions; therefore to map the GG pdf into a practical integer source, it is necessary to apply quantization methods. Different quantization methods have been developed in associating a continuous pdf to a discrete source [10] [11]. In this thesis, we use a uniform scalar quantizer with a deadzone at the origin as shown in Figure 2-10 [12]. The quantization step size is δ and the width of the deadzone at the origin is $(1+\alpha)\delta, \alpha \geq 0$. Such uniform scalar quantizers with a deadzone are common in many coding systems. As also shown in Figure 2-11, there are three different types of mappings following the quantization results: positive, non-negative, and two-sided non-zero discrete sources, which provide appropriate matches for different sources. For instance, "in many wavelet, JPEG, and MPEG image and video coding applications, transformed, scalar quantized image data is raster scanned to generate a description using pairs of the form (RUN, LEVEL) where "run" is the number of zero-valued coefficients encountered before the next significant coefficient, and "level" is the magnitude and sign of the integer representing the significant coefficient. The run values are restricted to the non-negative integers, or, if runs of zero are not encoded, to the positive integers. The "levels", on the other hand, are two-sided, nonzero integers [12]. While studying the theoretical properties of the UPCs, in this thesis, we use the positive mapping. It is obvious that no generality is lost.

A quantized GG source is specified by the shape parameter υ, the normalized quantizer step size δ/σ, the deadzone parameter α and the mapping.

From Figure 2-10, and Eq.(2.22), we formalize the quantization to be:

$$P(k)=\frac{2}{1-P(0)}\int_{(2k-1+\alpha)\frac{\delta}{2}}^{(2k+1+\alpha)\frac{\delta}{2}} c_1\exp(-c_2|x|^{\upsilon})dx, \quad k=1,2,3,\ldots \tag{2.24}$$

And:

$$P(0)=2\int_0^{(1+\alpha)\delta/2} c_1\exp(-c_2|x|^{\upsilon})dx \tag{2.25}$$

The quantized GG sources have been shown to be able to provide efficient and accurate models for many different types of image and video data. For instance, recent works in subband image coding have resulted in the high-peaked, heavy-tailed distributions such as GG and some others previously mentioned in [10] [11] [13] [14]. It was pointed out that wavelet transformed image data can be modelled using GG sources with a shape parameter within the range $0<\upsilon<1$. When $\upsilon=1$, we know that the GG pdf becomes Laplacian, which has been shown to provide good models for many image video systems, such as in the modelling of the prediction errors in lossless image coding algorithms. Therefore, in this thesis we will focus on the study of GG sources with the shape parameter within the interval $(0,1]$.

When $\upsilon=1$, the GG pdf becomes Laplacian, the quantization, then becomes:

$$\frac{P(k+1)}{P(k)}=\frac{\int_{(2k+1+\alpha)\frac{\delta}{2}}^{(2k+3+\alpha)\frac{\delta}{2}} c_1\exp(-c_2x^{\upsilon})dx}{\int_{(2k-1+\alpha)\frac{\delta}{2}}^{(2k+1+\alpha)\frac{\delta}{2}} c_1\exp(-c_2x^{\upsilon})dx}=\exp(-c_2\delta) \tag{2.26}$$

Here $\exp(-c_2\delta)$ is a constant that is smaller than 1 but larger than 0 and we can see that after the quantization, the GG pdf becomes a geometric distribution with $\theta=\exp(-c_2\delta)$. We have shown and proved in the previous section that, the codes by Gallager in [4], UPH codes, and the modified UPH codes are all optimal in the coding of geometrically distributed discrete sources, therefore, for the quantized GG source with $\upsilon=1$, optimal coding could be achieved.

Now let us look, in a similar manner, at the GG pdf with $0<\upsilon<1$. Firstly we simplify the expression in Eq.(2.24) as:

$$P(k)=c_0\int_{a_k}^{a_k+\delta} c_1\exp(-c_2x^{\upsilon})dx, \quad k=1,2,3,\ldots\ldots \tag{2.27}$$

where $a_k=(2k-1+\alpha)\delta/2$. Then, we have:

$$R_k = \frac{P(k)}{P(k+1)} = \frac{c_0 \int_{a_k}^{a_k+\delta} c_1 \exp(-c_2 x^{\upsilon}) dx}{c_0 \int_{a_k+\delta}^{a_k+2\delta} c_1 \exp(-c_2 x^{\upsilon}) dx} = \frac{\int_{a_k}^{a_k+\delta} \exp(-c_2 x^{\upsilon}) dx}{\int_{a_k+\delta}^{a_k+2\delta} \exp(-c_2 x^{\upsilon}) dx}$$
$$= \frac{\int_{a_k}^{a_k+\delta} \exp(-c_2 x^{\upsilon}) dx}{\int_{a_k}^{a_k+\delta} \exp(-c_2 (x+\delta)^{\upsilon}) dx} \tag{2.28}$$

It can be shown that $R_k < \exp(\upsilon \cdot \delta \cdot a^{\upsilon-1})$. Therefore for $0 < \upsilon < 1$, the probability values in the quantized GG source decrease at a higher rate for probability values near zero, and decrease at a lower rate when the probability value is far from zero. Consequently, this results in a quantized GG source with higher peaks and heavier tails.

We would like to apply the different types of UPCs to the quantized GG sources with a shape parameter in the range $(0,1]$ and compare their performances. It must be noted, however, that the quantized GG sources have finite entropy, according to the following theorem:

Theorem 4: Let $P = \{P(k)\}_{k=1}^{\infty}$ be the quantized GG source, then $H(P) < \infty$.

Note that according to this theorem, the finite entropy hypothesis is satisfied by the quantized GG source; hence we can directly apply our previous theoretical results concerning the code efficiency to this source. We will return to this at a later stage.

Proof: We want to show that

$$H(P) = -\sum_{k=1}^{\infty} P(k) \log P(k) < \infty \tag{2.29}$$

where $P(k)$ is defined as in Eq.(2.27).

It is easy to check that multiplying $P(k)$ by a constant will not change the finiteness of $H(P)$, thus it can be assumed that $c_0 = 1$. As $\exp(-c_2 x^{\upsilon})$ is a decreasing function, we have:

$$\delta e^{-c_2 (\alpha_k + \delta)^{\upsilon}} < P(k) < \delta e^{-c_2 \alpha_k^{\upsilon}} \tag{2.30}$$

Let $Q = \{q_k = P(k)/\delta\}_{k=1}^{\infty}$, then

$$H(Q) = -\sum_{k=1}^{\infty} q_k \log q_k = \frac{1}{\delta} H(P) + \frac{1}{\delta} \log \frac{1}{\delta} \tag{2.31}$$

It is obvious that if $H(Q)$ is finite, then $H(P)$ must be finite. Now since $a_k = (2k - 1 + \alpha)\delta/2$, we can find fixed integers n, m, such that:

$$(k - n)\delta < a_k < a_{k+1} = a_k + \delta < (k + m)\delta. \tag{2.32}$$

We then have:

$$0 < -q_k \log q_k < [(k+m)\delta]^{\upsilon} e^{-c_2[(k-n)\delta]^{\upsilon}} = C(k+m)^{\upsilon} e^{-D(k-n)^{\upsilon}} \tag{2.33}$$

where C, D are constants. However, it is a simple matter to check that the infinite series

$$\sum_{k=1}^{\infty} (k+m)^{\upsilon} e^{-D(k-n)^{\upsilon}} \quad \text{and} \quad \sum_{k=1}^{\infty} k^{\upsilon} e^{-Dk^{\upsilon}}$$

has the same convergence. Hence to prove the theorem, it suffices to show that $\sum_{k=1}^{\infty} k^{\upsilon} e^{-Dk^{\upsilon}}$ is a convergent series.

Now since the function $x^{\upsilon} e^{-Dx^{\upsilon}}$ is eventually decreasing, we can use the integral test for this infinite series. We obtain:

$$\int_1^{\infty} x^{\upsilon} e^{-Dx^{\upsilon}} dx = \int_1^{\infty} \frac{-x}{D\upsilon} d(e^{-Dx^{\upsilon}}) = \frac{e^{-D}}{D\upsilon} + \frac{1}{D\upsilon} \int_1^{\infty} e^{-Dx^{\upsilon}} dx < \infty. \tag{2.34}$$

The last term is finite because $e^{-Dx^{\upsilon}}$ is essentially a probability density function. Hence it has been shown that $\sum_{k=1}^{\infty} k^{\upsilon} e^{-Dk^{\upsilon}}$ is convergent.□

Now we are ready to apply the UPCs to the quantized GG sources. In comparing the performances of the several types of UPCs, we used the coding efficiency defined as:

$$\eta = \frac{h}{L_{av}} \tag{2.35}$$

Where h is the entropy of the quantized GG sources:

$$h = -\sum_{k=1}^{\infty} P(k) \log_2 P(k) \tag{2.36}$$

And the average code length is:

$$L_{av} = \sum_{k=1}^{\infty} P(k) l(k) \tag{2.37}$$

The comparison results are given in Figures 2-11, 2-12, 2-13, 2-14, 2-15, and 2-16. For each υ, the standardized stepsize δ/σ is chosen within the range $[10^2, 10^0]$, which is adequate for modelling image and video data [12].

Figure 2-11 shows the performances of the UPCs for quantized GG source with $\upsilon = 0.1$. Here, it can be seen that the EG code with $k = 0$, HG codes and the UPH codes, all perform very efficiently for these sources. The GR codes and EG codes with larger k values are comparatively inefficient. The UPH codes, in this case, achieve coding efficiencies very close to the entropy and are obviously superior to the remaining codes.

Figure 2-12 and 2-13 show the performances of the UPCs for quantized GG source with $\upsilon = 0.3$ and $\upsilon = 0.5$. The comparison results are similar to those in Figure 2-11. However, the GR codes are more efficient, when comparing the larger shape parameters. The UPH codes still show comparatively better performances for these quantized GG sources.

Figure 2-14 and 2-15 shows the performances of the UPCs for quantized GG source with $\upsilon = 0.7$ and $\upsilon = 0.9$. Here it can be seen that, as opposed to the sources with smaller shape parameters, for these quantized GG sources, the GR codes take over. In the figures, we see that GR codes produce efficiency peaks that are much higher than those for EG and HG codes. This is because the "decay" rate becomes constant. However, the UPH codes, as we expected, still outperform all other type of UPCs. In the figures, we also see that, for the quantized GG sources with increased δ/σ values, all UPCs appear to have lower coding efficiency.

Figure 2-16 shows the performances of the UPCs for quantized GG source with $\upsilon = 1.0$. Now the quantized GG become geometric distributions, and as has been proven, the UPH codes now become the optimal codes. GR codes are now simply special cases of the UPH codes. Therefore, the efficiency curve of the UPH code becomes the envelope for those of the GR codes. Between the efficiency peaks of the GR codes, the UPH codes are able to provide constant coding efficiency.

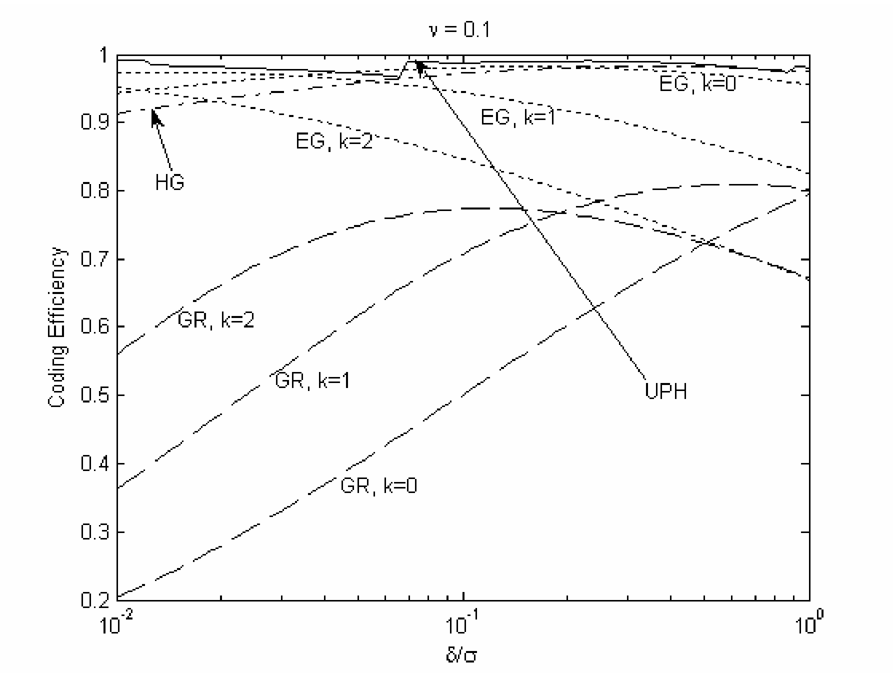


Figure 2-11 Comparison of coding efficiencies of different UPCs for quantized GG sources with $\upsilon = 0.1$

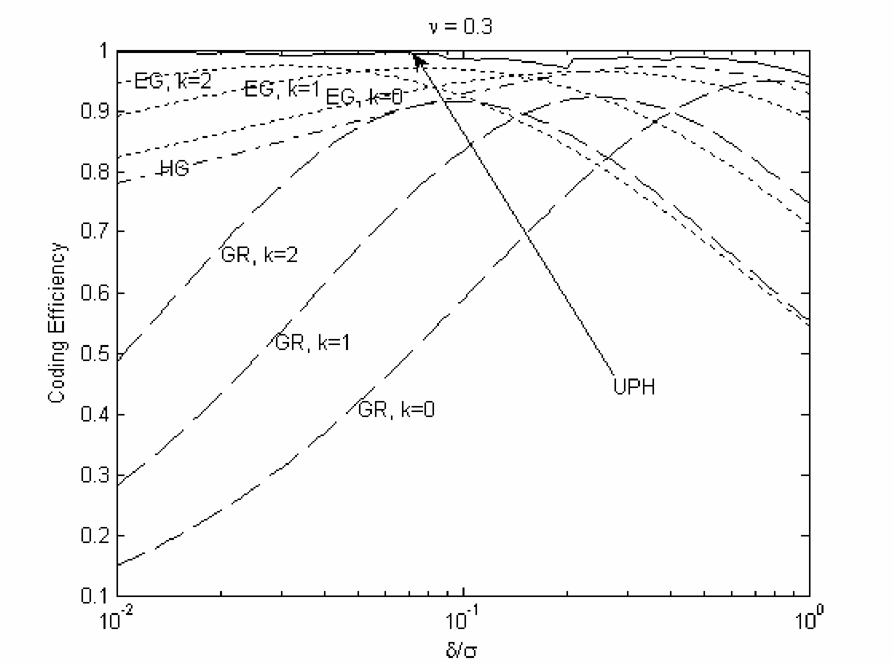


Figure 2-12 Comparison of coding efficiencies of different UPCs for quantized GG sources with $\upsilon = 0.3$

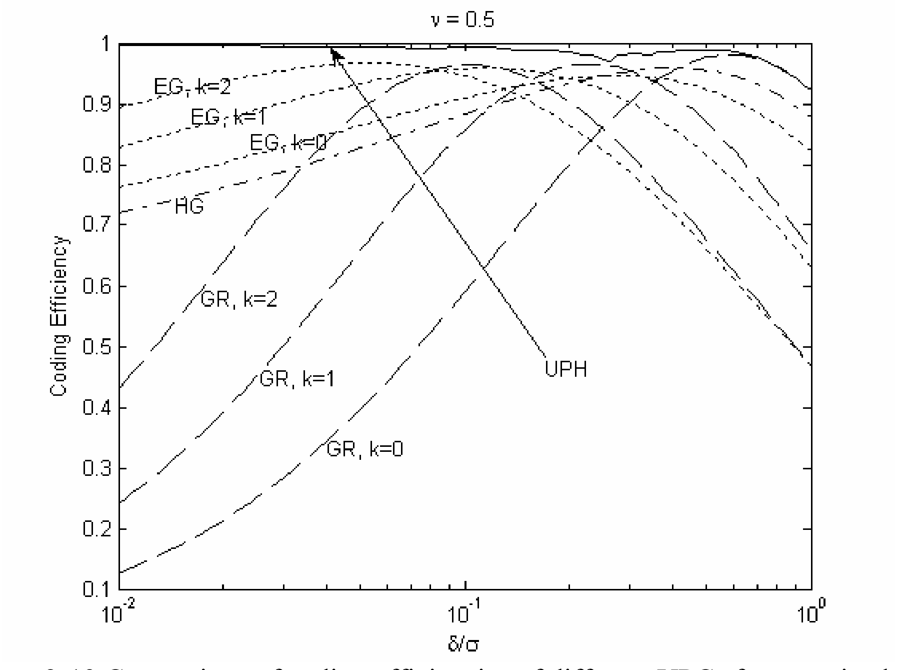


Figure 2-13 Comparison of coding efficiencies of different UPCs for quantized GG sources with $\upsilon = 0.5$

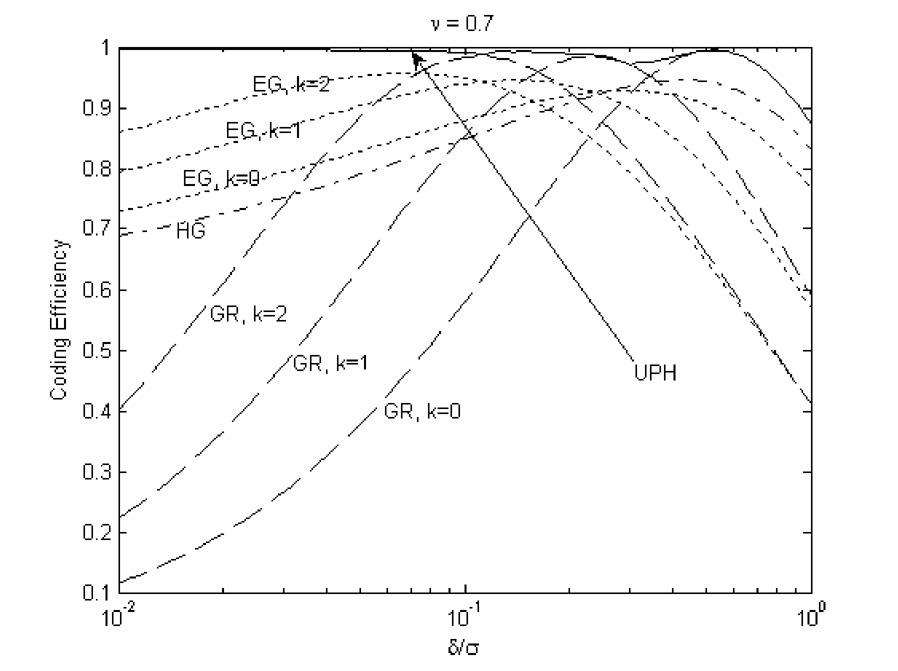


Figure 2-14 Comparison of coding efficiencies of different UPCs for quantized GG sources with $\upsilon = 0.7$

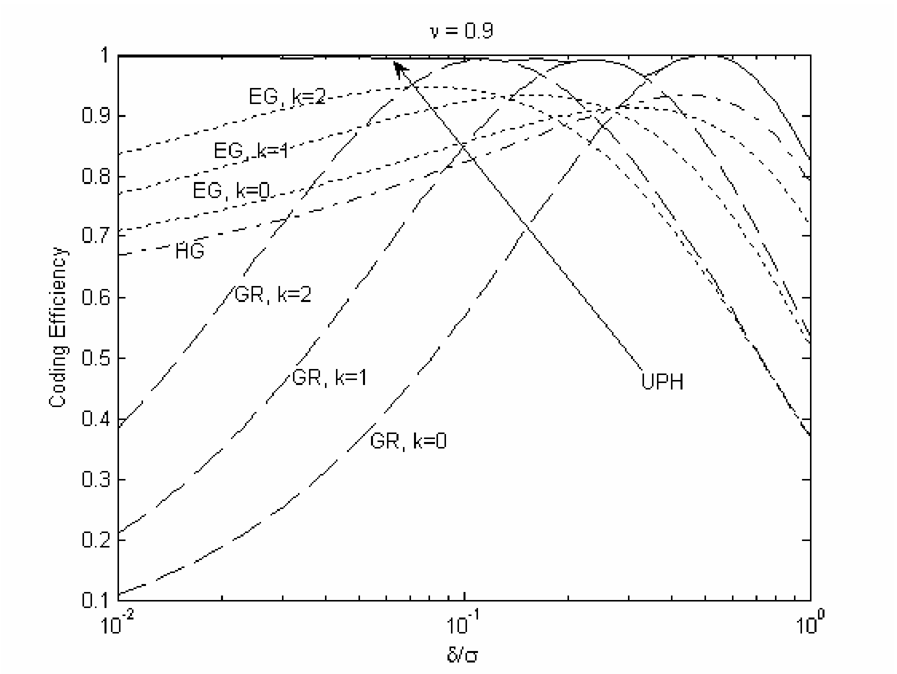


Figure 2-15 Comparison of coding efficiencies of different UPCs for quantized GG sources with $\upsilon = 0.9$

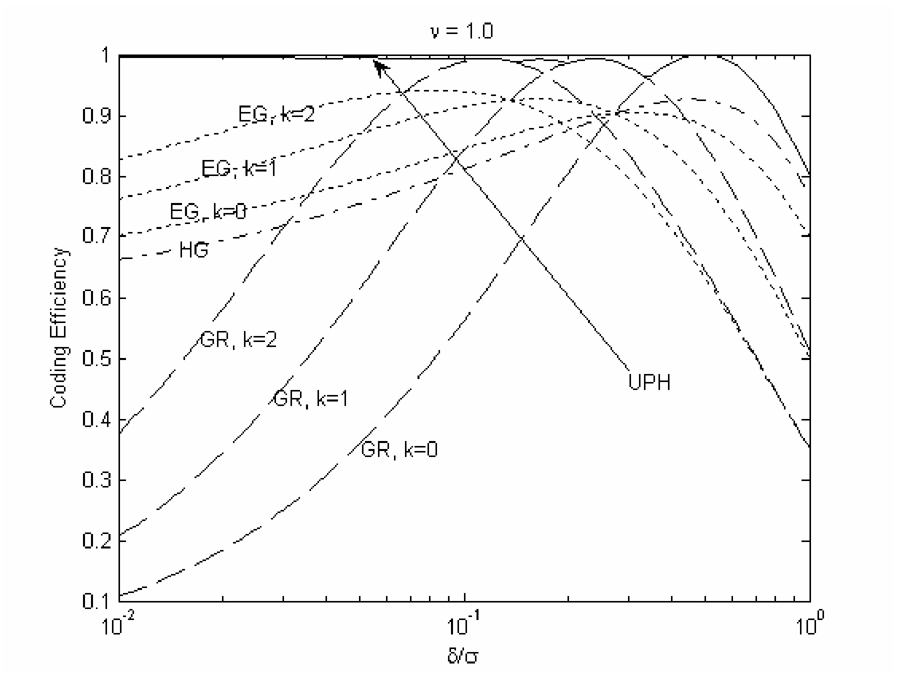


Figure 2-16 Comparison of coding efficiencies of different UPCs for quantized GG sources with $\upsilon = 1.0$

We can also see from these figures that the UPH codes perform exactly as the GR, EG codes for some υ and δ/σ. This means that, for some source parameters, the GR and EG codes are actually special cases of the UPH codes, whereas, because of the flexibility of the UPH algorithm, the UPH codes are able to achieve better matches to the GG sources and thus perform consistently well for all source parameters.

From these results, we see that these different types of UPCs are all highly efficient in coding the high-peaked and heavy-tailed distributions such as the quantized GG. Due to the unary prefixes, the UPCs are able to match the exponential decrease of the distribution reasonably well and hence are in general efficient.

The GR, EG and HG codes are highly structured and thus simple in construction. However, they do suffer from compromised performances for some source parameters. The UPH codes, on the other hand, are robust in performance. The UPH codes, however, are unable to provide a closed form code structure and are therefore more difficult to build.

In section 2.3.4, we introduced the modified UPH codes. The modified UPH codes, with only one optimization step, cannot provide an improvement on the coding efficiency of the UPH codes. The modified UPH codes, by applying the pseudo fixed length codes for each unary prefix, yield similar code structures to those of the GR codes and the codes in [4] for many circumstances. However, the modified UPH codes are still able to provide better coding efficiency when compared to the GR and, in particular, the EG codes.

It has been proved that there does not exist discrete sources that could be optimally coded using the EG codes. However, the authors in [15] designed a class of pdfs that are well matched to the EG codes and they also showed that these pdfs are good probability models for empirically observed integer sources, such as in the coding of the quantised subband of wavelet-transformed images [4]. These integer sources can be expressed using the discrete pdf:

$$P_\alpha(k) = \frac{1}{\psi'(\alpha)}(\alpha + k)^{-2}, \quad k = 1, 2, 3, \ldots\ldots \tag{2.38}$$

where $\alpha > 0$, ψ' is the first derivative of the digamma function $\psi(y) = \Gamma'(y)/\Gamma(y)$, and $\Gamma(y)$ is the Euler gamma function.

A random variable, whose probability distribution is given by Eq.(2.38) ., has infinite mean and entropy and thus, in relation to the performances of the EG codes and the modified UPH codes, we compare the estimated coding redundancies of these two different of UPCs.

Figure 2-17 shows a comparison of the estimated redundancies of the modified UPH codes and the EG codes with different k values in coding the pdfs in Eq.(2.38) for a wide range of different α values. From the figure, it is obvious that the modified UPH codes are better, with reference to compression, when compared to the EG codes. Moreover, since the UPE algorithm works more adaptively according to different pdfs with different parameters than do the EG codes, it is unnecessary to make selections of k to achieve a better performance, which is the case for the EG codes.

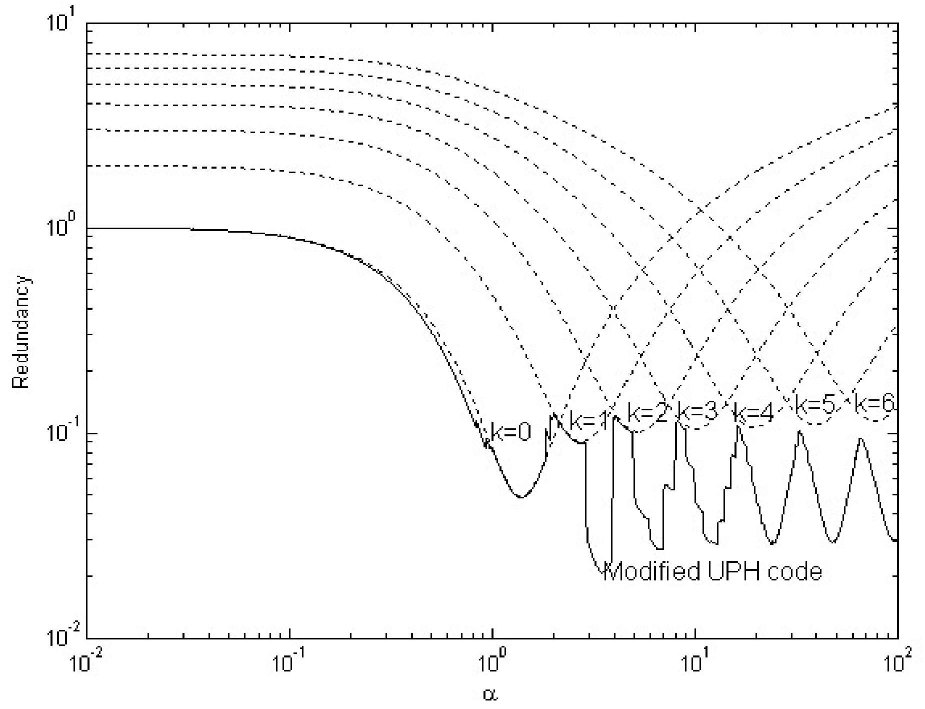


Figure 2-17 Comparison of the redundancies of the EG codes and the modified UPH codes

2.5 THE WEAK LOWER BOUND OF THE UPH CODES

In section 2.3.3 we have shown that the coding efficiency of the UPH code is lower bounded by entropy + 2. In the last section of this chapter, we want to demonstrate that this bound is fairly weak given the outstanding performances of the UPH codes. This is particularly true for the high-peaked, heavy-tailed sources studied in this thesis.

Here in this section, we still use the quantized GG sources with shape parameter in the range (0,1] and δ/σ is chosen within the range $[10^{-2}, 10^{0}]$. Figures 2-18, 2-19, 2-20, 2-21, 2-22 and 2-23 show the coding efficiency of the UPH codes and the corresponding lower bound for quantized GG sources with different shape parameters. It can be seen that it is possible for the lower bound to be as low as 40% or even worse, when the coding efficiency of the UPH codes is still near the entropy. This shows the weakness of the lower bound when we are studying such high-peaked, heavy-tailed sources. It seems that one could develop still better bounds and this is an obvious extension area for the work in this thesis.

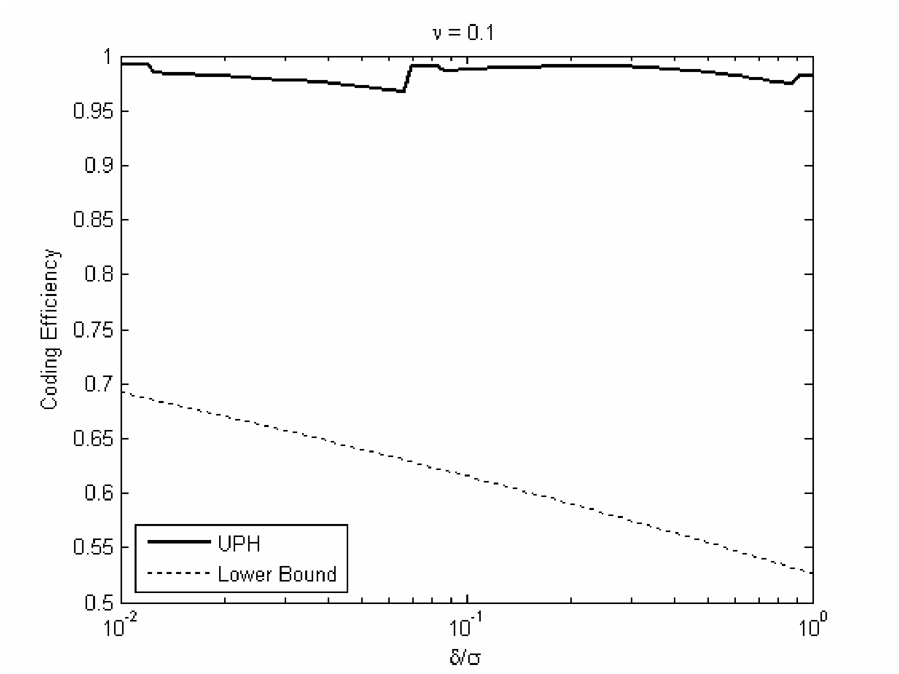


Figure 2-18 Lower bound of UPH code for quantized GG with shape parameter 0.1

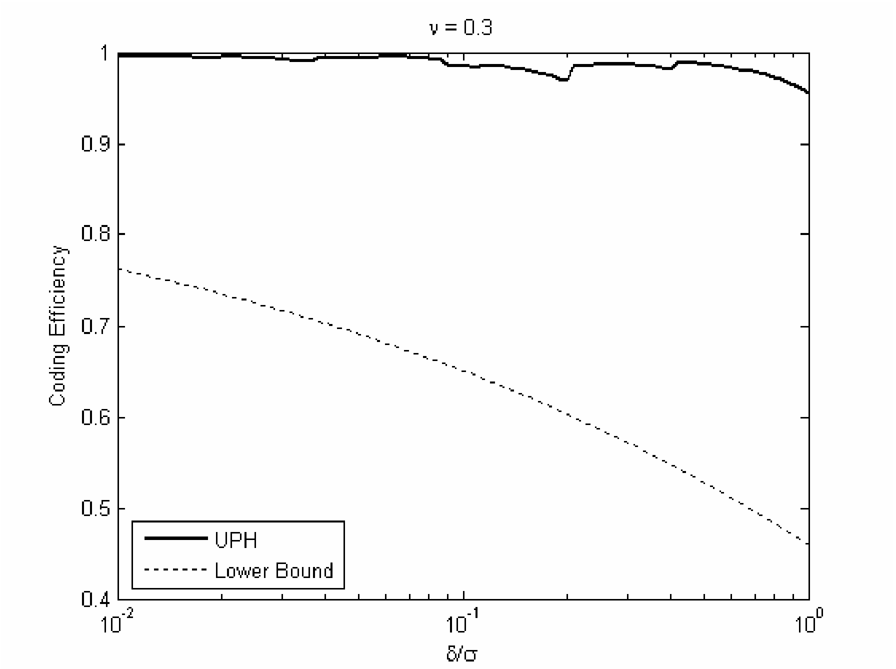


Figure 2-19 Lower bound of UPH code for quantized GG with shape parameter 0.3

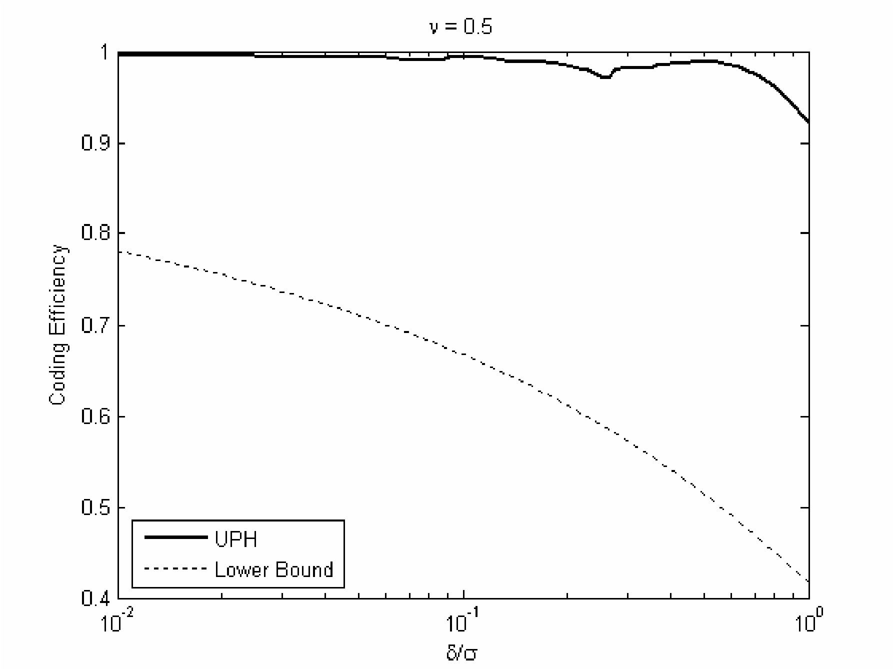


Figure 2-20 Lower bound of UPH code for quantized GG with shape parameter 0.5

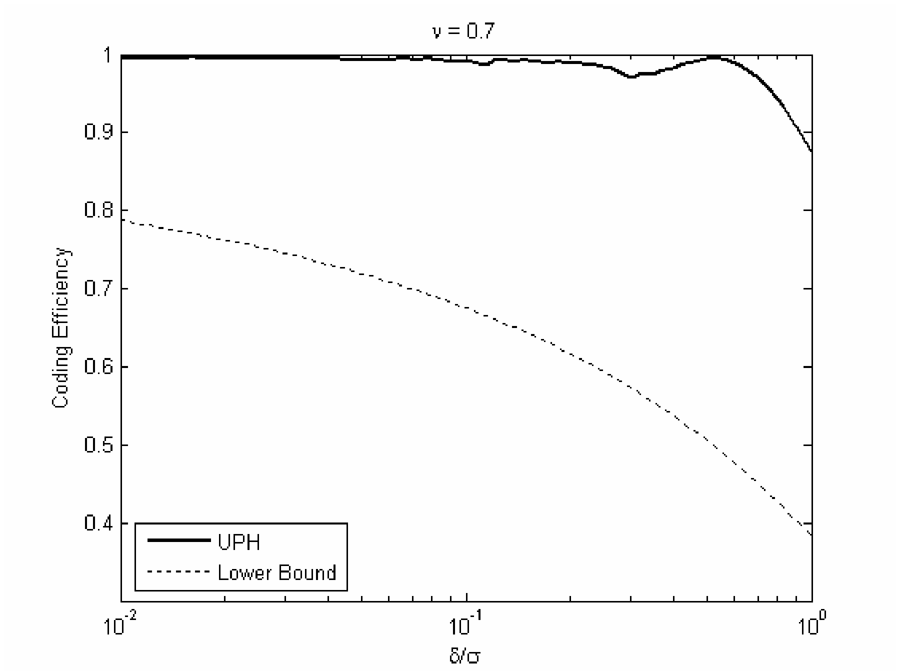


Figure 2-21 Lower bound of UPH code for quantized GG with shape parameter 0.7

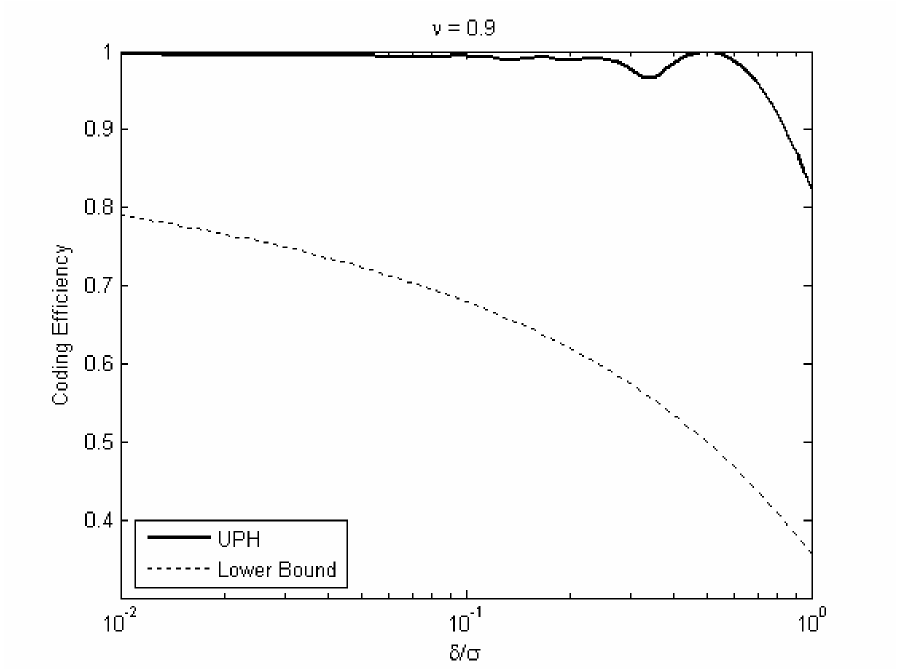


Figure 2-22 Lower bound of UPH code for quantized GG with shape parameter 0.9

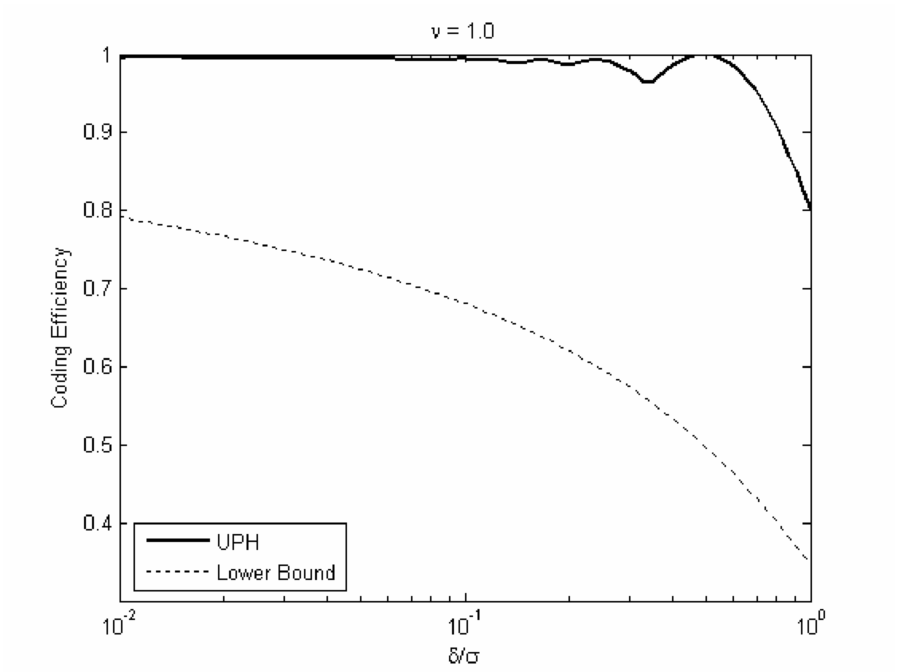


Figure 2-23 Lower bound of UPH code for quantized GG with shape parameter 1.0

3 ALTERNATING CODING

In this chapter, we introduce a coding method called "Alternating Coding" (ALT). The ALT is built for the UPCs and on the basis of the UPCs. The UPCs are variable length codes (VLC). In the encoding and decoding procedures, the UPCs are usually treated the same as any other type of VLCs. The VLCs in general, are difficult to decode because of their variable lengths. However, we have seen in the previous chapter that, the UPCs have specific structures and, in taking advantages of these, it may assist in relaxing the constraints of the decoding procedure and thus be beneficial. The ALT coding method is thus designed with such concerns in mind. The ALT, by extracting the unary properties of the prefixes of the UPCs, provides a different approach to the encoding and decoding and thus enables a simpler means of decoding as well as a possible mechanism for error resiliency. In this chapter, we introduce the ALT coding method and discuss its applications.

3.1 THE ALT CODING IN GENERAL

From the previous chapter, we have seen that, a UPC consists of a unary prefix and a variable length suffix. Any UPC code, no matter whether it is GR, EG, HG, UPH or modified UPH, is in such a form. Although the code lengths of UPCs vary, the unary prefixes provide a natural grouping of the codes and thus each unary prefix conveys certain information about the codes. For instance, for GR codes with suffix length equal to three, every prefix group includes exactly 8 codewords. Therefore, when a codeword is given, by checking the unary prefix, we will be able to locate the codeword within only 8 codes. The ALT coding, then, attempts to extract such information, which is conveyed by the unary prefixes. The basic concept of the ALT coding is to code the unary part of the UPC and the variable length part of the UPC separately. Such separation should provide convenience in the extraction of the information in the unary prefixes. However on the other hand, such separation should not break the dependencies of the prefixes and the suffixes; neither should it complicate the encoding and decoding to any extent. The ALT, does indeed attempt to take care of all these aspects.

Now let us look at how ALT actually works. By applying the ALT coding, a UPC sequence is split into two sub-sequences: a unary prefix sub-sequence and a variable length suffix sub-sequence as illustrated in Figure 3-1. The unary prefix sub-sequence contains only the prefixes, and the variable length suffix sub-sequence contains only the variable length suffixes. The order of the codewords is kept intact. Now in the unary prefix sub-sequence, the prefixes are easily separated by extracting the zeros. For instance, if we have a sequence of GR codes with suffix length k equal to 2, the prefix sub-sequence may appear as follows:

$$\{10, 110, 111110, 0, 111110, 110, 10, 0, 0, 11110\}.$$

We know that every prefix is of the form $\underbrace{11....1}_{n}0$, where *n* can be 0,1,2,3, etc. Therefore each zero in the prefix indicates the last bit of one prefix. Such information makes it is easy to decode the unary prefixes. The unary prefixes, as we have seen, relate to the suffixes of the UPCs in particular ways and once the unary prefixes have been successfully separated, it is possible to use such relations between the prefix and the suffix of each UPC to direct us to further separations of the suffixes. Once the prefixes and the suffixes are both successfully separated, complete UPCs are then successfully separated. In general, the unary prefixes provide an index to a group of suffixes, which naturally provides a more rapid location of the UPCs. However, different UPCs have different prefix-suffix relations, therefore the ALT coding must be modified slightly to adapt to these differences. This is particularly true for the highly structured UPCs such as the GR codes and the EG codes, where the prefix-suffix relation is very special and therefore enables great freedom in the simplification of the decoding. Moreover, for these highly structured codes, we are able to implement a comparatively more efficient error handling mechanism.

By separating the prefixes and suffixes, we have two separate sequences, prefix sub-sequence and suffix sub-sequence instead of one. We have shown that in the prefix sub-sequence, the direct concatenation of the unary prefixes already offers us simple prefix boundary detection. However, we would like to change such direct concatenation into a different, yet equivalent form. The reason for this will become clear in subsequent discussions. We will show that such modification in the prefix sub-sequence allows for the possibility for equipping the error handling mechanism. Instead of the direct concatenation, we change the unary prefixes in the prefix sub-sequence into two sets of codes, one set containing all-one codes and the other all the zero codes. The unary prefixes in the unary prefix sub-sequence are then coded in an alternating manner using the all-one codes and all-zero codes. As the prefix is a unary code, the code length uniquely identifies the unary prefix itself and thus we can use any code with the same length to represent the unary codes. By alternating the all-zero and all-one codes, the codeword boundaries are indicated by changes from one to zero or zero to one. Thus the code boundary detection is still as simple as in the original form.

Let us look at a simple example of alternating the coding of the unary prefixes using the all-one codes and all-zero codes. This example is illustrated by the GR codes. Table 3-1 gives the code table of the GR code with suffix of length 2.

n	GR	Unary prefix	Prefix Length	All-one/All-zero codes	Suffix	Suffix Length
0	000	0	1	1/0	00	2
1	001	0	1	1/0	01	2
2	010	0	1	1/0	10	2
3	011	0	1	1/0	11	2
4	1000	10	2	11/00	00	2
5	1001	10	2	11/00	01	2
6	1010	10	2	11/00	10	2
7	1011	10	2	11/00	11	2
8	11000	110	3	111/000	00	2
9	11001	110	3	111/000	01	2
10	11010	110	3	111/000	10	2
11	11011	110	3	111/000	11	2
12	111000	1110	4	1111/0000	00	2
13	111001	1110	4	1111/0000	01	2
14	111010	1110	4	1111/0000	10	2
15	111011	1110	4	1111/0000	11	2
…	…	…	…	…	…	…

Table 3-1 GR code with suffix length 2

Suppose we are to code an integer sequence consisting of 11 integers {5, 6, 3, 1, 0, 1, 2, 0, 11, 0, 15} using the GR codes, then we obtain a GR sequence:

{1001, 1010, 011, 001, 000, 001, 001, 000, 11011, 000, 111011}.

For this GR sequence, the unary prefixes are:

{10, 10, 0, 0, 0, 0, 0, 0, 110, 0, 1110}.

Now, by alternating the all-zero codes and the all-one codes from the unary prefixes it forms the unary prefix sub-sequence:

{11, 00, 1, 0, 1, 0, 1, 0, 111, 0, 1111}.

For the suffixes, no modification is applied, and they are simply concatenated to form a suffix sub-sequence. Then these two sub-sequences are concatenated to form an ALT packet and the decoding of an ALT coded UPC sequence is based on this entire ALT packet.

By using this GR code example and, as has been previously mentioned, the suffixes for the GR codes are fixed length codes, thus in this case, the "variable length" suffixes are merely a set of 2-bit codes {01, 10, 11, 01, 00, 01, 01, 00, 11, 00, 11}. For the ALT coding, these suffixes are kept intact and concatenated to the prefix sub-sequence. Therefore the ALT packet of this GR code example becomes:

{$\underline{\text{11, 00, 1, 0, 1, 0, 1, 0, 111, 0, 1111}}$, 01, 10, 11, 01, 00, 01, 01, 00, 11, 00, 11}.

In this example the prefix sub-sequence is underlined.

As is indicated by the name "Alternating Coding", the key part of this coding method relies on the separation of the two sub-sequences and the alternating coded prefix sub-sequence.

3.1.1 The ALT encoding

From the simple example above, we have seen how a UPC sequence is separated into two sub-sequences to form an ALT packet. The example uses the GR code set and the encoding is performed by simply concatenating the two sub-sequences. However, this varies for different types of GR codes. In this section we describe how an ALT packet is completely encoded from any UPC sequence packet and indicate the differences for different types of UPCs.

Before discussing the ALT encoding, the UPCs must be separated into two categories. For some UPC codes, the suffixes are of variable length, such as the HG codes, the UPH codes and the modified UPH codes. For others, the suffixes are of fixed length or the suffix lengths are fixed for each prefix, such as in the GR codes and the EG codes. The encoding of a UPC sequence is slightly different for these two types of UPCs.

For UPCs with fixed suffix lengths, the code length information is entirely conveyed by the unary prefixes. However, "fixed suffix length", does not necessarily mean that the suffix must be of fixed length, but means that the suffix contains completely redundant information regarding the code length. For instance, the length of a GR code is the length of the unary prefix plus the *k*-bit suffix. Here the suffix length is indeed fixed. But the suffix length of an EG code is not fixed, yet by knowing either the suffix or prefix, the code length is known. Suppose the prefix length is *j*, the fixed part of the suffix is *k*, and then the code length is $2j+k-1$. Therefore although the suffix length varies, we are able to figure out the length of the entire UPC codeword without knowing the length of the suffix given that we have the prefix.

It is possible to illustrate the encoding of a UPC sequence with fixed length suffixes by Figure 3-1. This figure shows a UPC sequence enclosed by two synchronization markers. The synchronization markers are commonly found in the coding of VLCs. The synchronization markers are used to resynchronize the decoding at intervals where there are concerns about errors. The encoding is simply separating the prefix and suffix of each code, collecting the prefixes and suffixes to form a prefix sub-sequence and a suffix sub-sequence, respectively. The prefix sub-sequence is then placed in front of the suffix sub-sequence and the synchronization markers are kept intact. Some ALT packet information may need to be added to the encoded ALT packet, indicating how many codewords there are in the entire packet, which is very important if parallelization and error resiliency are required in the decoding procedure. In many real cases where UPCs are applied, such information is already included in the header, therefore it is only necessary to resort the prefixes and suffixes.

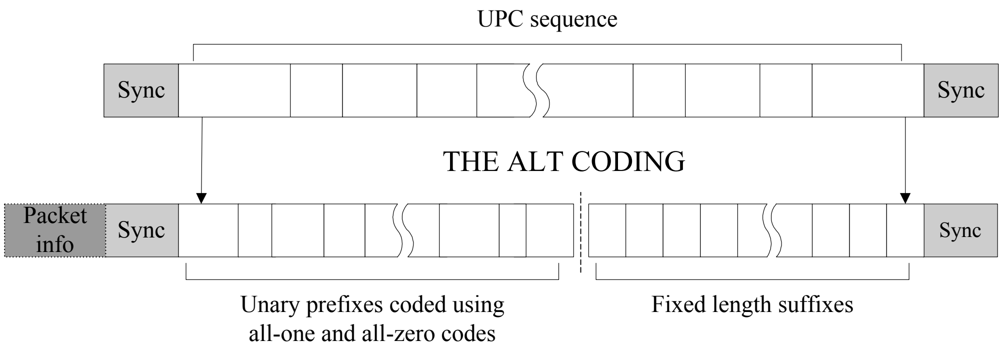


Figure 3-1 The ALT coding for fixed-length-suffix UPCs

Figure 3-2 shows the GR code example from the previous section. The prefixes are underlined. For the GR codes, the suffixes are completely fixed length codes, so only the prefix sub-sequence is a variable length sequence.

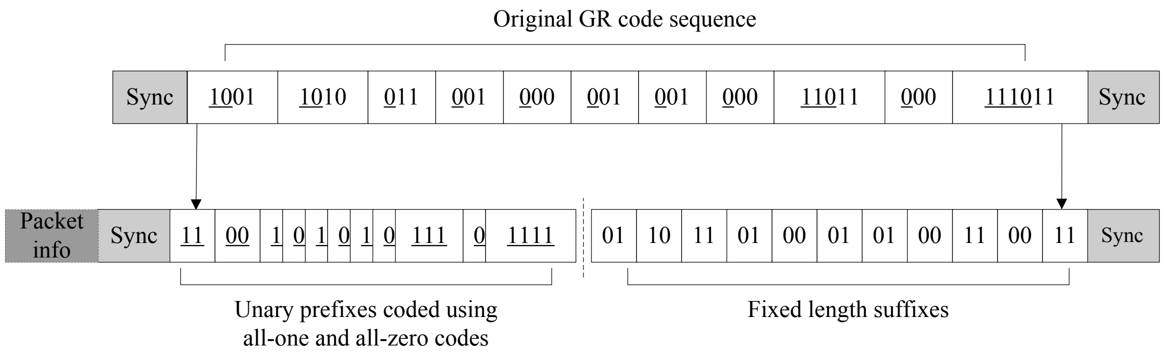


Figure 3-2 The GR code example

For the EG codes, the suffix length varies with the prefix. So both the prefix and the suffix sub-sequences are variable length codes. Table 3-2 shows a set of EG codes.

n	EG	Unary prefix	Prefix Length	Suffix	Suffix Length
0	0	0	1	-	0
1	100	10	2	0	1
2	101	10	2	1	1
3	11000	110	3	00	2
4	11001	110	3	01	2
5	11010	110	3	10	2
6	11011	110	3	11	2
7	1110000	1110	4	000	3
8	1110001	1110	4	001	3
9	1110010	1110	4	010	3
10	1110011	1110	4	011	3
11	1110100	1110	4	100	3
12	1110101	1110	4	101	3
13	1110110	1110	4	110	3
14	1110111	1110	4	111	3
…	…	…	…	…	…

Table 3-2 EG code with parameter *k*=0

It is again obvious from this table that although the suffix length of the EG codes are not exactly fixed; it is linearly related to the length of the prefixes. Therefore the suffix length could still be deemed to be fixed.

Figure 3-3 shows the EG code example using the code table above. The prefixes are underlined.

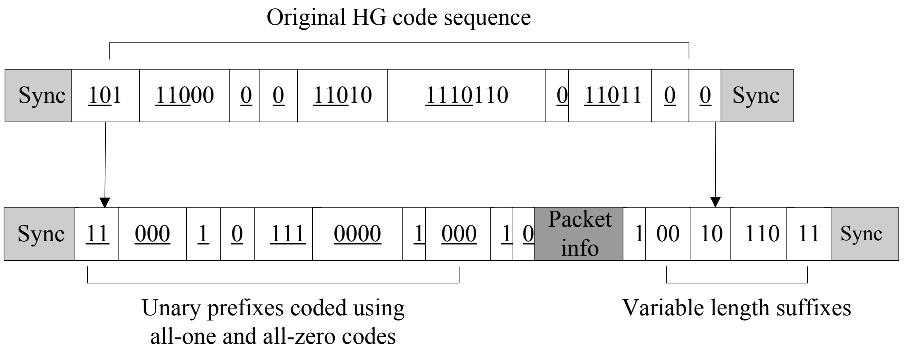


Figure 3-3 The EG code example

We see that the resulting ALT packet for this example is:
{<u>11, 000, 1, 0, 111, 0000, 1, 000, 1, 0</u>, 1, 00, 10, 110,11}.

For the UPCs with variable suffix lengths, the role of the prefix is no more than a code group index, so the encoding of a UPC sequence with variable length suffixes can be illustrated as shown in Figure 3-4. The only difference in the ALT coding for such codes is that the packet information is placed in between the prefix sub-sequence and the suffix sub-sequence. In this case the packet serves as a separation between the two sub-sequences, which is crucial in the decoding procedure; therefore it is no longer optional.

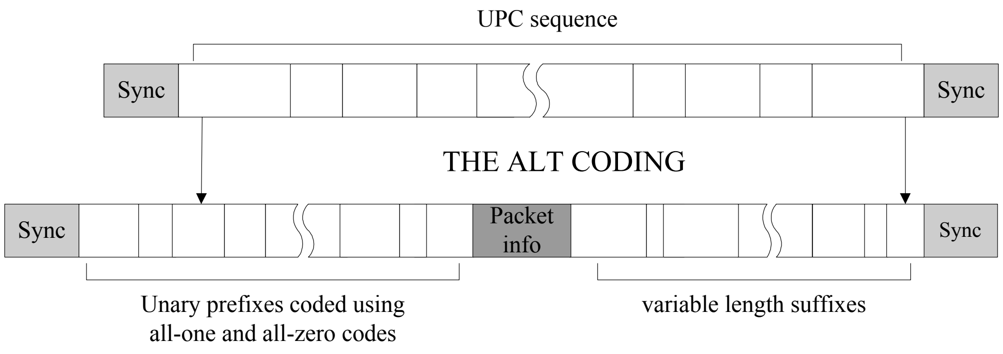


Figure 3-4 The ALT coding for variable-length-suffix UPCs

For this type of UPC, an HG code sequence is taken as an example. The set of HG codes are shown in Table 3-3.

Suppose an HG sequence {10, 1100, 0, 10, 1100, 1110110, 0, 11010, 10, 0} is encoded using the ALT method. This encoding is illustrated in Figure 3-5 and the unary prefixes are underlined. The resulting ALT packet for this example is:
{<u>11, 000, 1, 00, 111, 0000, 1, 000, 11, 0</u>, 0, 0, 10, 10}.

n	HG	Unary prefix	Prefix Length	All-one/All-zero codes	Suffix	Suffix Length
0	0	0	1	1/0	-	0
1	10	10	2	11/00	-	0
2	1100	110	3	111/000	0	1
3	11010	110	3	111/000	00	2
4	11011	110	3	111/000	01	2
5	111000	1110	4	1111/0000	00	2
6	111001	1110	4	1111/0000	01	2
7	111010	1110	4	1111/0000	10	2
8	1110110	1110	4	1111/0000	110	3
9	1110111	1110	4	1111/0000	111	3
10	11110000	11110	5	11111/00000	000	3
…	…	…	…	…	…	…

Table 3-3 HG code with parameter *k*=0

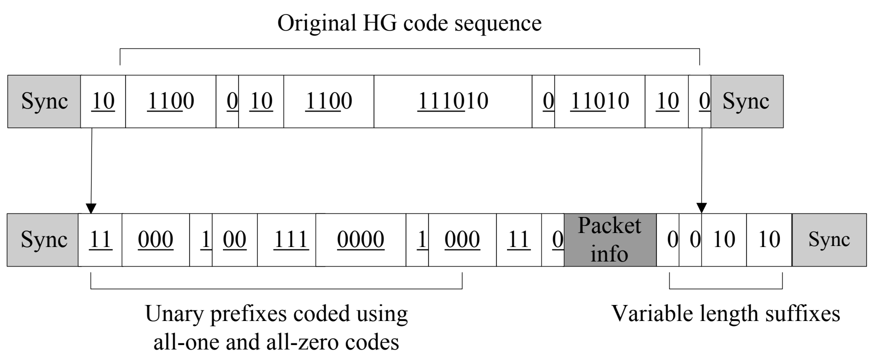


Figure 3-5 ALT encoding of the HG code sequence (*k*=0)

3.1.2 The ALT decoding

All UPCs are variable length codes and as such are usually decoded using general VLC decoding methods. Decoding of the VLCs is, in general, inefficient because the variable code lengths make it difficult to detect the codeword boundaries. In a VLC sequence, decoding of the current codeword depends on the completion of the decoding for the previous codewords because the end of a codeword cannot be found until it is decoded. This makes the decoding of the VLC packet a serial procedure and it is thus usually very difficult to parallelize. Moreover, to detect the codeword boundaries of VLCs, we must also introduce codeword tables, or look up tables (LUT) in the decoding procedure, which include all possible codewords in the VLC sequence, in order to match the codeword and enable decoding to continue. The codeword tables are usually very large and thus to search for and match a codeword with one specific code in the table makes the decoding computationally complex.

However, we have seen that the UPCs are of a particular pattern and the ALT coding may help to simplify the decoding procedure. This is especially true for UPCs with fixed suffix lengths. Thanks to their highly structured code pattern, the decoding procedure can be greatly simplified when comparing it to the traditional VLC decoding.

In the previous section we have seen that the ALT encoding is a packet-based procedure; therefore, decoding is also based on packets. Since the encoding procedures are slightly different for UPCs with fixed suffix lengths to those of variable suffix lengths, the decoding procedures for these two types of UPCs are also slightly different. A description of the decoding of these two types of UPCs now follows.

For UPCs with fixed suffix length such as the GR codes and EG codes, the ALT coding enables a very simple decoding structure that allows parallelization, and the LUTs could be eliminated completely. To decode the ALT coded GR or EG packets, buffering is required to store at least one entire packet. But since buffering is almost a given in a VLC decoder, no extra functional component is added to the ALT decoder.

The decoding of an ALT packet can be illustrated by Figure 3-6. It should be remembered that packet information provides us with the number of codewords in a packet. Since the suffix lengths of the UPCs are fixed and with the information of the number of codewords in the packet, it is easy to determine how many bits in the packet belong to the prefix sub-sequence and how many to the suffix sub-sequence. It is then a simple matter to separate the prefix and suffix sub-sequences. We have also indicated that the packet information is not a necessity for the UPCs with fixed suffix lengths. When the packet information is not provided, it is necessary for the decoding of the ALT coded UPC packet to be performed from both ends and as a serial procedure, similar to that for the general VLCs. Such cases are not dealt with here as, under these circumstances, the advantages of the ALT method are not evident.

The prefix sub-sequence consists of all-one codes and all-zero codes and the detection of the boundary is then easily achieved via a row of exclusive OR (XOR) logics. This greatly simplifies the decoding and also makes it possible to parallelize the decoding procedure because the xor logics does not depend recursively upon previously decoded codewords and can be done simultaneously for the entire packet. When the boundaries of the prefixes are detected, the length of the suffixes can be easily calculated using the previous knowledge concerning the suffix lengths or the linear relationship between the prefix and the suffix. With the readily separated prefixes and suffixes, the UPC codewords can then be restored. In the previous chapter it was shown that UPCs are designed for discrete sources and, indeed, they are generally used to encode integers. For these UPCs with fixed length suffixes, the inverse mapping from the UPCs back to the integers can easily be done using algebraic calculations. Therefore the LUTs are no longer required.

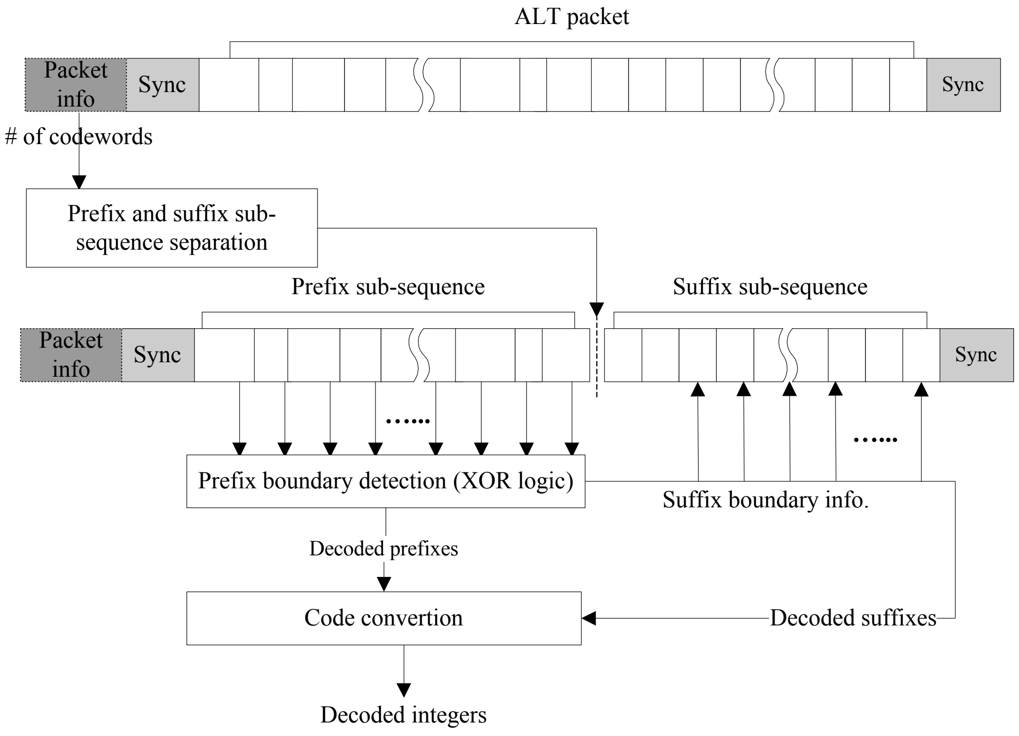


Figure 3-6 ALT decoding for UPCs with fixed suffix length

Two examples are now investigated, one from the GR codes and one from the EG codes, respectively.

Suppose we have an ALT coded GR sequence with suffix length k equal to 2, and the packet information is provided, so it known that there are $n = 8$ codewords in the packet (the original GR code table is given in Table 3-4):

{11, 0, 11, 0, 1, 00, 111, 0000, 00, 10, 10, 11, 00, 11, 01, 10}

Coded Integer	GR	Prefix Length
0	000	1
1	001	1
2	010	1
3	011	1
4	1000	2
5	1001	2
6	1010	2
7	1011	2
8	11000	3
9	11001	3
10	11010	3
11	11011	3
…	…	…

Table 3-4 GR code with *k*=2

The decoding could be characterized using the following steps:

1) Upon receiving the packet, it is known that there are $k \cdot n = 2 \cdot 8 = 16$ bits belong to the suffix sub-sequence, thus the two sub-sequences can be separated as follows:
Prefix sub-sequence: {11, 0, 11, 0, 1, 00, 111, 0000}
Suffix sub-sequence: {00, 10, 10, 11, 00, 11, 01, 10}

2) Perform the xor operations to the prefix sub-sequence and obtain the prefix boundaries and decode the prefix lengths simultaneously: {2, 1, 2, 1, 1, 2, 3, 4}

3) Separate the fixed-length suffixes and map them back to integer values: {0, 2, 2, 3, 0, 3, 1, 2}

4) Decode the encoded integer by using:

$$\text{coded integer} = (\text{prefix length } -1) \cdot 2^{k} + \text{suffix}$$

For instance, the first integer is coded as: $(2-1) \cdot 2^{2} + 0 = 4$. The packet is then decoded as: {4, 2, 4, 3, 0, 7, 9, 18}. Thus the decoding is complete.

Now, another example of the decoding of the EG codes will be looked at. Suppose we have an ALT coded EG sequence with parameter *k* equals to 0, and the packet information is provided, so it is known that there are $n = 8$ codewords in the packet (the EG code table is given in Table 3-2):

{11, 0, 11, 0, 1, 00, 111, 0000, 0, 1, 1, 11, 010}

The decoding could be characterized using the following steps:

1) Upon receiving the packet its length is counted first, which in this example is 24 bits. We know that for EG codes $k = 0$, the suffix length equals the prefix length -1. Thus, with the information concerning the number of codewords in the packet, it is possible to calculate the length of the suffix sub-sequence. Suppose the length of the suffix sub-sequence is l_s, the total length of the ALT packet is then $2 \cdot l_s + n = 2 \cdot l_s + 8 = 24$, thus $2 \cdot l_s = 24 - 8 = 16$, and then $l_s = 8$. With the length of the suffix sub-sequence, we are then ready to separate the prefix sub-sequence and the suffix sub-sequence:

Prefix sub-sequence: {11, 0, 11, 0, 1, 00, 111, 0000}

Suffix sub-sequence: {0, 1, 1, 11, 010}

2) Perform the xor operations to the prefix sub-sequence and obtain the prefix boundaries and simultaneously decode the prefix lengths: {2, 1, 2, 1, 1, 2, 3, 4}

3) Now that the length of each prefix is known, it is possible to easily determine the length of each corresponding suffix. Then the suffixes could be separated and mapped back to integer values: {0, 1, 1, 3, 2}

4) Decode the encoded integer by using:

$$\text{coded integer} = 2^{\text{suffix length}} + \text{suffix} - 1.$$

For instance, the first integer is coded as: $2^1 + 0 - 1 = 1$ and the packet is then decoded as: $\{1, 0, 2, 0, 0, 2, 6, 9\}$. Thus the decoding is complete.

The decoding of ALT packets of UPCs with variable length suffixes is somewhat more complicated and the LUTs cannot be eliminated because of the variable suffixes. The decoding can be illustrated by Figure 3-7. When the suffix lengths of the UPCs are not fixed, the separation of the prefix and suffix sub-sequences must rely on separation codes. By checking the separation code, the ALT packet between the two synchronization markers can be separated into a prefix sub-sequence and a suffix sub-sequence. The prefix sub-sequence still consists of alternating coded all-one codes and all-zero codes, so the detection of the prefixes is the same as for the UPCs with fixed length suffixes. Once the prefixes are detected, it is no longer possible to automatically separate the suffixes by associating the prefix lengths to the suffix lengths because the variation in the suffix lengths cannot be related to the lengths of the prefixes. Under such circumstances, the LUTs are still required in order to decode the suffixes. However, now the LUT is only required for the suffixes, thus the LUT table size can be greatly reduced. As the LUT is reduced in size, the search for the codeword can be accelerated. Table 3-5 is the LUT of a set of HG codes (k=0), and Table 3-6 shows the reduced LUT after ALT coding is applied.

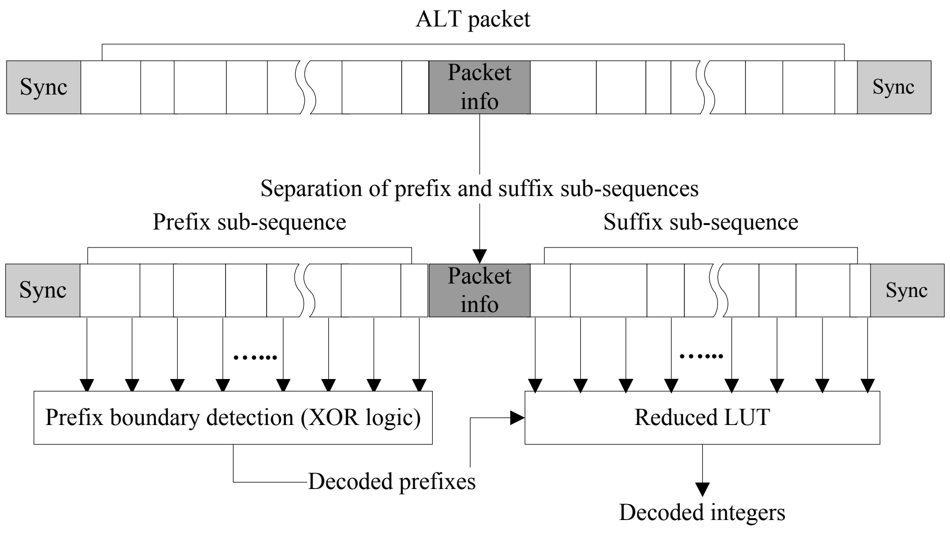


Figure 3-7 ALT decoding for UPCs with variable suffix length

Coded Integer	LUT
0	0
1	10
2	1100
3	11010
4	11011
5	111000
6	111001
7	111010
8	1110110
9	1110111
10	11110000

Table 3-5 HG code with parameter k=0

Coded Integer	Prefix Length	LUT
0	1	-
1	2	-
2	3	0
3	3	00
4	3	01
5	4	00
6	4	01
7	4	10
8	4	110
9	4	111
10	5	000

Table 3-6 HG code with parameter *k*=0

The next example concerns the decoding of the HG codes, using the code table in Table 3-5 and 3-6. Suppose the ALT coded HG sequence appears as follows: {11, 0, 11, 0, 1, 00, 111, 0000, 0, 1, 1, 11, 010}

The decoding could be characterized using the following steps.

1) Upon receipt of the packet plus information, the prefix and suffix sub-sequences can be immediately separated.

Prefix sub-sequence: {11, 0, 11, 0, 1, 00, 111, 0000}

Suffix sub-sequence: {01, 01}

2) Perform the xor operations to the prefix sub-sequence and obtain the prefix boundaries and simultaneously decode the prefix lengths: {2, 1, 2, 1, 1, 2, 3, 4}

With the length of each prefix, it is possible to match the associated suffix to the reduced LUT. For instance, the first prefix has length two, from Table 3-6, it can be seen that for this prefix, there is no suffix, and the integer represented by this codeword is “1”. The last prefix has length four, thus the five possible suffixes associated with this prefix are searched and it is found that suffix “01” with prefix length four represents “7”. By this means, the entire ALT packet is decoded: {1, 0, 1, 0, 0, 1, 4, 6}.

3.2 THE ERROR RESILIENCY OF THE ALT CODING

It has been mentioned in previous sections that one advantage of the ALT coding is the possibility of implementing an error resiliency mechanism for the code packets.

A UPC sequence, belonging to the VLC family, is very vulnerable to transmission over a noisy channel because of synchronization losses. Even one bit error may cause loss of synchronization for the entire code packet. This is because the bit error may cause the failure of decoding and as the decoding of the VLC sequence is a serial procedure, failure to decode one codeword terminates the entire decoding procedure. This is shown in Figure 3-8. Therefore, when an error is detected in a VLC sequence, the entire decoded packet can no longer be trusted and has to be retransmitted. One way to increase the error resiliency of the VLCs is to replace them using fixed length codes. However fixed length codes are not typically as efficient as traditional entropy codes, which are almost always VLCs, in complexity and memory-constrained environments. Particularly in image and video coding systems because of the development of image and video communications, there is strong incentive to preserve the basic VLC coding framework used in the standards while at the same time attempting to improve the error resiliency.

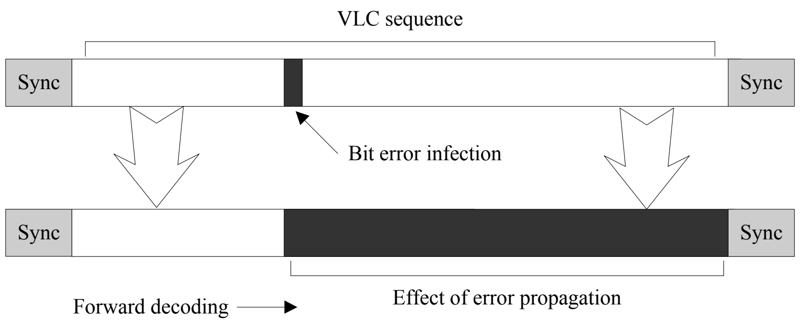


Figure 3-8 Bit error propagation of a VLC sequence

3.2.1 Bi-directional decodability

The considerations regarding the error resiliencies of the VLCs have led to a growing level of interest in the error resilient coding of VLCs, such as the Reversible Variable Length Code (RVLC). The RVLC was first proposed by Takishima, Wada and Murakami in [16]. The idea behind the RVLCs is that decoding can be performed by processing the received code sequence either forwards or backwards. This is based on the fact that the RVLCs are VLCs which can be uniquely decoded from both directions. By using the RVLC instead of the

VLC, the error resiliency of a code sequence can be improved by decoding both from the beginning and the end of the code sequence. For example, a decoder can begin by processing the received code sequence in the forward direction, and upon detection of an error, proceed immediately to the end of the bit stream and decode in the reverse direction. By this means, the error may be located in a much smaller section of the code sequence and more codewords could be recovered from an error infected sequence. Figure 3-9 shows how RVLC could retrieve codewords unable to be retrieved by common one-way decodable VLCs.

In [16], Takishi *et al.* they studied the conditions for the existence of RVLCs and proposed algorithms for their construction. Toshiba and Ericsson [17] proposed two different schemes for constructing RVLCs with systematic structures enabling easy coding and decoding. The RVLCs, must satisfy the suffix condition for instantaneous backward decoding as well as the prefix condition for instantaneous forward decoding. The suffix condition is that each codeword does not coincide with the suffixes of longer codewords; while the prefix condition expresses that there is no coincidence with the prefixes of longer codewords. It is then obvious that, if a set of VLCs are of symmetric structures and they satisfy the prefix condition, they will automatically satisfy the suffix condition and therefore are automatically RVLCs. Not all VLCs could be modified into RVLC without lengthening some of the original VLC codewords. However, by extending the code length of necessary codewords, we are always able to modify VLCs into RVLCs.

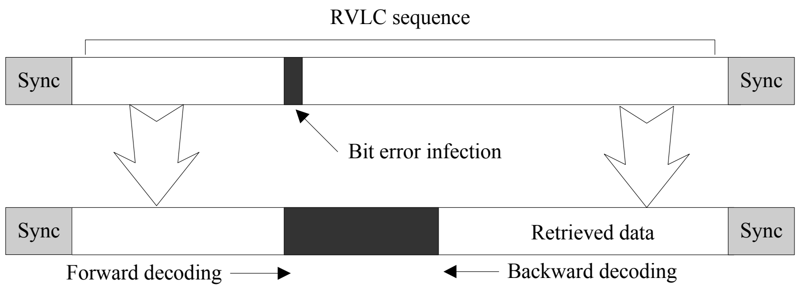


Figure 3-9 Bit error propagation of a VLC sequence

The authors in [18] proposed a class of parameterized RVLC that have length distributions identical to those of GR codes and EG codes. Since it has been proved that GR codes and EG codes are able to correspond to pdfs well matched to the statistics of image and video data, such as Laplacian and GG pdfs, it follows that the RVLC counterpart of GR and EG enables an increase in robustness in order to channel errors while incurring no penalty in coding efficiency. Table 3-7(a) and 3-7(b) shows the examples of RVLCs for GR and EG codes.

	$k=1$				$k=2$			
	GR		RVLC		GR		RVLC	
	Prefix	Suffix	Prefix	Suffix	Prefix	Suffix	Prefix	Suffix
0	0	0	0	0	0	00	0	00
1	0	1	0	1	0	01	0	01
2	10	0	11	0	0	10	0	10
3	10	1	101	1	0	11	0	11
4	110	0	101	0	10	00	11	00
5	110	1	1001	1	10	01	11	01
6	1110	0	1001	0	10	10	11	10
7	1110	1	1001	1	10	11	11	11
…	…	…	…	…	…	…	…	…

Table 3-7(a) GR code and the RVLC counterpart

	EG	RVLC
0	00	00
1	01	01
2	1000	1010
3	1001	1011
4	1010	1110
5	1011	1111
6	110000	100010
7	110001	100011
8	110010	100110
9	110011	100111
10	110100	110010
11	110101	110011
…	…	…

Table 3-7(b) EG code and the RVLC counterpart ($k=1$)

In the code table it can be seen that the RVLCs for GR and EG satisfy the prefix and suffix condition simultaneously, and the code length distributions are exactly the same as the original GR codes and EG codes. The authors in [18] showed that by using the RVLCs, the bi-directional decoding gave an image domain Peak-Signal-to-Noise Ratio (PSNR) that was on average 2.2 dB superior to the PSNR obtained using forward decoding only, when the Bit Error Rates (BER) was 10^{-4}. The average PSNR improvement at a 10^{-3} BER was 0.9 dB.

No changes were required to be made to an ALT packet in order to make the UPCs with fixed suffix lengths, namely GR and EG codes, bi-directionally decodable as the code structure for these UPCs is completely symmetric. Therefore, there are greater increases in error resiliency for UPCs when they are coded and decoded using the ALT method.

However, for UPCs with variable suffix lengths, such as HG codes, UPH codes and modified UPH codes, although applying the ALT coding still keeps their prefixes in a symmetric structure, modifications are still required to make the variable length suffixes into RVLC in order for the code sequence to be bi-directionally decodable. This may require extensions of suffix lengths. Therefore not all UPCs can be converted to RVLCs without sacrificing the coding efficiency. However for those UPCs with fixed suffix lengths, the ALT coding offers the bi-directional decodablity as a free product without compromising the coding efficiency.

3.2.2 Error Speculation

We have seen that, by applying the ALT coding, for UPCs with fixed length suffixes, bi-directional decodablity could automatically be obtained without penalty. But what is more appealing is that, by introducing an "Error Speculation" (ES) mechanism, we are able to achieve even better error resiliency for the UPCs with fixed length suffixes.

The ES mechanism is developed based on the fact that in an ALT packet, bit errors in the suffix sub-sequence will not propagate. Taking GR and EG as examples, we know that for GR and EG codes, the prefixes are unary codes, so any single bit error within the prefix will cause loss of synchronization and therefore cause the error to propagate. Whereas the suffixes of the GR and EG codes are simply fixed or pseudo fixed length codes, with every bit in the suffix being either "1" or "0", thus a bit error in the suffixes is not detectable and will not propagate. For instance, for a set of GR codes with $k = 2$, suppose we are to encode integers $\{5, 3, 0, 2, 0\}$, the GR sequence would be $\{1001, 011, 000, 010, 000\}$, if a bit error occurs in the prefix, say in the prefix of the first codeword, and the code sequence becomes: $\{1\boldsymbol{1}01, 011, 000, 010, 000\}$. Now the sequence would be parsed as: $\{1\boldsymbol{1}010, 11000, 010, 000\}$ and then decoded as $\{10, 8, 2, 0\}$. Here we see that, this one bit error in the prefix has caused the incorrect parsing of three codewords. On the other hand, if, instead, there is one bit error in the suffix, then if the error infected sequence appears as $\{100\boldsymbol{0}, 011, 000, 010, 000\}$, parsing of the codewords will not change, and the code sequence will be decoded as $\{4, 3, 0, 2, 0\}$. We see that here only the codeword infected by the bit error was incorrectly decoded and the other codewords were not affected.

Now in an ALT coded packet, the prefix sub-sequence consists of all-one and all-zero codes, the simple code pattern could help us to identify bit errors. A bit error in the prefix sub-sequence of an ALT packet will result in, at most, four incorrect decodings of the prefixes.

To simplify the analysis, we assume that only one bit error occurs in a prefix sub-sequence. The bit error will have four types of influences on the prefix sub-sequence. Suppose there are *N* prefixes in the ALT packet.

1) An error occurring on the boundary of the prefix sub-sequence causes an insertion or a deletion of one codeword. For example, the first codeword 1111 becomes 0111 or the first two codewords 0111 become 1111.

2) An error infects the shortest ALT coded prefix (i.e. one-bit prefix) which sits in between two codewords. This results in a deletion of two codewords. For example, 1110111 becomes 1111111. Then three prefixes become only one.

3) An error occurs in the middle of a prefix whose length is greater than two bits. This results in the insertion of two codewords. For example, 1111111 becomes 1110111. Then one prefix becomes three.

4) An error occurs on the boundary of two prefixes. This is a non-propagating error. For example, 1110000 becomes 1111000. This will not influence synchronization.

When cases 1, 2 or 3 occur, the number of prefixes detected will not be equal to *N*. When one of these cases is detected, we speculate where the error bit occurs by the "error speculation".

If the number of prefixes is *N*-1 or *N*+1, then case 1 has occurred. The error is then speculated to have occurred on the first or the last bit.

If the number of prefixes is *N*-2, then case 2 has occurred. If there exists a prefixes that has a length longer than the longest possible prefix length, this prefix must have been infected by a bit error. In this case, the error can be located precisely. Otherwise, it is speculated that the location of the error is within the longest prefix (as longer prefixes have less probability of occurrence.) in the prefix sub-sequence, and randomly change the value of one bit in this prefix. By doing so, resynchronization is achieved and many correct codewords can be resumed.

If the number of prefixes is *N*+2, then case 3 has occurred. It is assumed that a one-bit prefix in between the two shortest prefixes is the error bit (this is reasonable as the shorter the codeword is, the more probable it occurs in a sequence and hence more probable to be infected by an error). Again, resynchronization and error recovery can both be achieved.

Actually, no more than 4 codewords will be ruined even in the worst speculation when only one bit error exists in the sequence. This makes the ALT packet perform robustly when subject to a bit error.

Simulations were conducted to study the actual effect of bit errors with or without the implementation of the ALT coding and the ES mechanism. In the simulation, we chose the Universal Variable Length Codes (UVLC). UVLC is a reversible version of EG code with $k = 0$, and will be discussed in greater detail in later sections. For the original UVLC packet, two-way decoding is applied to increase the error resiliency, and the ALT coded UVLC packet only involves the ES and no two-way decoding is applied.

Figure 3-10 shows the comparison of performance in terms of Correct Ratio (CR) between the ALT coded packets with ES implemented and the UVLC. In this simulation, one bit error is inserted in code packet of different sizes.

The Correct Ratio is defined as:

$$\text{CR}=\frac{\text{Number correctly decoded codeword}}{\text{Total number of codewords}}$$

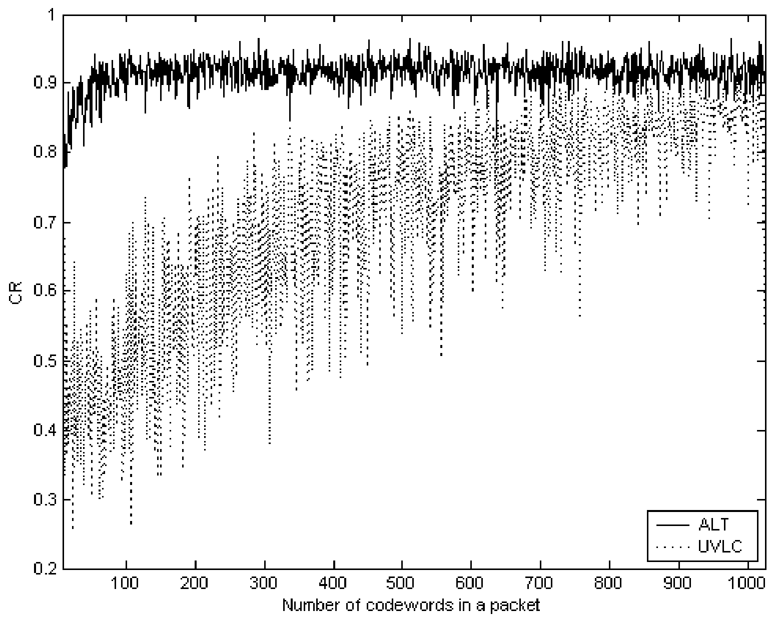


Figure 3-10 Comparison of CR

From the results it can be seen that when the ES mechanism is applied, the ALT coded decoding always achieves a CR of approximately 90%, yet that of the original EG packet can even fall below 30%. Moreover, the CR of the ALT packet shows much smaller variance than that of the UVLC packet and is therefore much more robust.

3.2.3 Combining bi-directional decoding and Error Speculation

In the previous sections, we have talked about the bi-directional decoding and the ES mechanism. The ES mechanism assumes only one bit error in an ALT packet and deals with cases that could only happen under such circumstances. In [19], the authors showed that for a Binary Symmetric Channel (BSC), the crossover probability is below 10^{-3}, the possibility of having more than one bit errors in a packet is thus very small. Therefore, the error speculation is able to efficiently improve the error resiliency and robustness of an ALT packet. However, in practice, we could not guarantee that only one bit error will occur in a code packet. To deal with more than one bit error in a packet, yet still being able to take advantage of the ES mechanism, would involve combining the ES with the bi-directional decoding.

In an ALT packet for UPCs with fixed suffix lengths, to combine these two error resilient methods, we first perform error speculation as described in the previous section. However, the ES mechanism only works when there is one bit error in an ALT packet. When more than one bit error occurs in a code packet, the ES may or may not work since the ES is only able to handle four different error patterns. When the error speculation fails, we perform a two-way decoding.

It is possible for two-way decoding and ES to be performed at the same time, when ES fails; the result of the two-way decoding could then be used instead. The result of the two-way decoding is based on the detection of the error in the packet. Error detection is achieved by examining one of the following:

1) Upon decoding, the last bit of a packet does not coincide with the end of a codeword.

2) When the number of decoded codewords is greater than the number of codewords in the packet.

3) When an illegal codeword is detected.

When decoding is interrupted by one of the above, errors must have occurred in one or all of the previously decoded codewords, so the previously decoded codewords should not be trusted. Backward decoding results in a similar outcome. The codewords able to be trusted then fall in the intersection of the forward and backward decoding.

For UPCs with fixed length suffixes, the two-way decoding could be further simplified thanks to the regular code structure and fixed suffix length. Since only errors in the prefixes would cause error propagation, we only need to look at the prefix sub-sequences to perform the two-way decoding. And as the prefixes are coded using all-zero and all-one codes, there are much simpler ways to detect errors.

Let us take the EG code with $k=0$ as an example. Suppose an ALT coded EG packet consists of N codewords of L bits in length. The lengths of the prefixes are denoted as $l_1,\ l_2,\ l_3, \ldots\ldots\ l_M$. The lengths of the corresponding suffix lengths are then $l_1-1,\ l_2-1,\ l_3-1, \ldots\ldots\ l_M-1$. M is the number of codewords detected. Errors will be detected when one or more of the following cases are encountered:

1) $M<N$.

Let f and b satisfy:

$$l_1-1+l_2-1+\ldots\ldots+l_f-1 \geq \frac{L-N}{2}$$

$$l_N-1+l_{N-1}-1+\ldots\ldots+l_b-1 \geq \frac{L-N}{2}$$

Then let set A be the set: $A=\{x \mid x\in(l_1,\ l_2,\ldots\ldots,l_{b-1})\cup(l_{f+1},\ldots\ldots,l_N)\}$

2) $M>N$.

Let B be the set: $B=\{x \mid x\in(l_1,\ldots\ldots,l_{N-M-1})\cup(l_{N+2},\ldots\ldots,l_M)\}$

3) Prefixes longer than the longest possible prefix are detected.

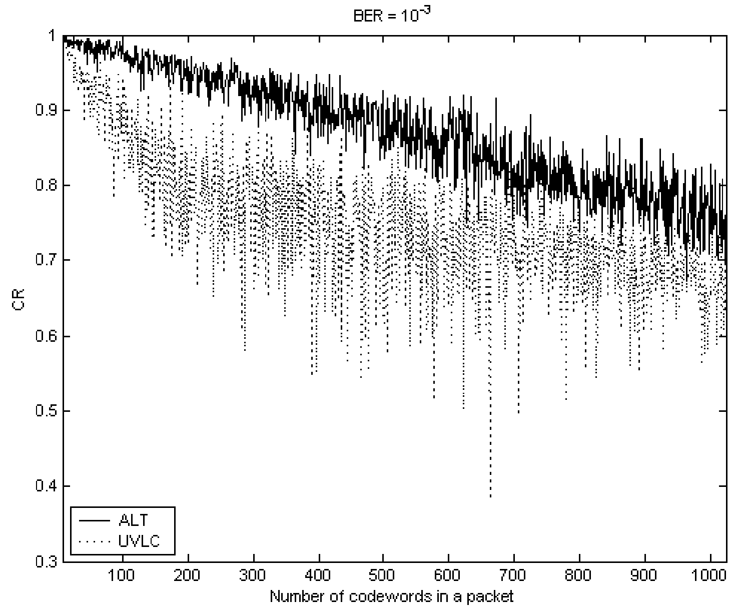
Assume that the prefixes which exceed the longest prefix are prefixes numbered $x_1,\ x_2,\ldots\ldots,x_k$. Then let set C be:

$C=\{x \mid x\in(l_1,\ldots\ldots,l_{x_1-1})\cup(l_{x_k+1},\ldots\ldots,l_N)\}$.

Then the decoded prefixes will be:

$$\text{Decoded prefixes} = \begin{cases} A, & \text{case 1} \\ A\cap C, & \text{case 1 and case 3} \\ B, & \text{case 2} \\ B\cap C, & \text{case 2 and case 3} \\ C, & \text{case 3} \end{cases}$$

Simulations are conducted to study the effect of the ES and the two-way decoding. Figure 3-11 shows a comparison of the CR between an ALT coded UVLC packet and an original UVLC packet. Both packets are subjected to corruption using a BSC with a bit error rate (BER) of 10^{-4} and 10^{-3}. The number of codewords in a packet varies from 8 to 1024.



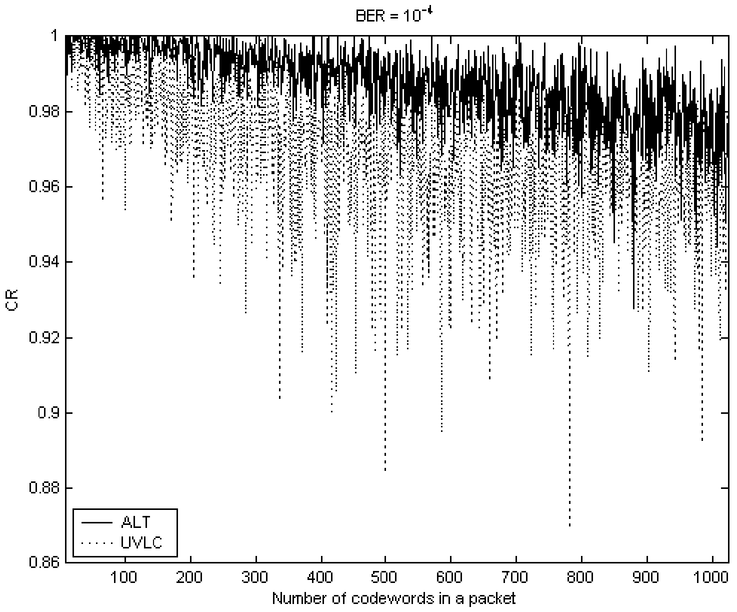


Figure 3-11 Comparison of CR of ALT coded EG and EG under different BERs

From Figure 3-11 it can be seen that the ALT packet always outperforms the original EG packet. The CR of the ALT packet is almost always exceeds 80% and has a much smaller variance. However the CR of the original EG packet falls below 60% for large packet sizes and bigger BER and its variance is much bigger. In addition, the advantage of the ALT packet is more evident when the packet is subjected to a higher BER. It can be seen that the CR of ALT packet under 10^{-3} BER is comparable to that of the EG under 10^{-4} BER when the packet size is less than 500.

3.3 APPLICATIONS OF THE ALT CODING

To demonstrate the error resiliency provided by the ALT coding, we applied the ALT coding to specific UPCs and studied the results. Since the UPCs, which are efficient entropy codes for sources with high-peaked, heavy-tailed probability distributions very often found in image/video data, we applied the ALT coding to the UVLC, which is commonly encountered in image/video systems. As we have mentioned, the UVLC is actually a reversible version of the EG codes [20]. Table 3-8 gives an example of the UVLC. "x" represents a bit that can be either one or zero.

Class	UVLC	Length	Value to be expressed
1	1	1	1
2	$0x_00$	3	$'x_0'+ 2[2:3]$
3	$0x_11x_00$	5	$'x_1x_0'+ 4[4:7]$
4	$0x_21x_11x_00$	7	$'x_2x_1x_0'+ 8[8:15]$
5	$0x_31x_21x_11x_00$	9	$'x_3x_2x_1x_0'+ 16[16:31]$
…	… …	…	… …

Table 3-8 An example of UVLC

The UVLC is used in H.26L to perform entropy coding. In [21], UVLC is suggested to be used in the coding of DCT coefficients for H.26L. It is claimed to be able to provide good performances in terms of coding efficiency, configurability to various applications, and error resiliency. In H.26L, the UVLC uses one infinite-extent codeword set rather than designing a different code for each element of the H.26L syntax, only the mapping to the single UVLC code table is customized to the probabilistic behavior of the data. However, extra bits need to be added to indicate the signs of each LEVEL. To apply the ALT coding to DCT coefficients, we make further separation of the UVLC coded "LEVELs", which keeps the codeword of RUNs and LEVELs in accordance. This also helps to simplify the decoding scheme. Table 3-9 and Table 3-10 give examples of RUN UVLC and LEVEL UVLC [21]. We see that the code tables of RUNs and LEVELs are actually identical except for the sign bits (marked by "s" in the tables) at the end of each codeword in the LEVEL table.

Codeword	Length	Value of RUN
1	1	0
$0x_00$	3	if x_0=0, EOB
		if x_0=1, RUN=1
$0x_11x_00$	5	'x_1x_0'+ 2[2:5]
… …	…	… …
$0x_50x_40x_31x_21x_11x_00$	13	'$x_3x_2x_1x_0$'+ 62[62:125]

Table 3-9 RUN UVLC

Codeword	Length	Absolute value of LEVEL
1s	2	1
000	3	EOB
010s	4	2
$0x_11x_00$s	6	'x_1x_0' + 3[3:6]
… …	…	… …
$0x_50x_40x_31x_21x_11x_00$s	14	'$x_5x_4x_3x_2x_1x_0$' + 63[63:126]

Table 3-10 LEVEL UVLC

To apply ALT coding to DCT coefficients, we further separate each packet into one of ALT coded UVLCs and one of sign bits as the code tables of RUNs and LEVELs are identical except for the sign bits. Figure 3-12 shows the separation. By doing such a separation, the codewords in the "ALT coded UVLC packet" are then kept in accordance and therefore can be decoded as described in previous sections.

After the ALT coded UVLC packet is decoded, the sign bits can then be imposed upon the LEVELs as the position of each LEVEL is then known.

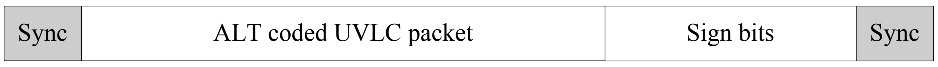


Figure 3-12 Further separation of ALT packet in DCT coding

For the DCT coefficients, RUNs and LEVELs appear pairwisely, so the number of codewords between two suffixes must be even. However, the error speculation as well as ALT decoding itself, may result in an incorrect partition of the code packet and therefore the number of codewords between two suffixes may be odd. When the number of codewords between suffixes is detected to be odd, we always discard one codeword to make it even. This results in the absence of some high frequency components, which influence only the details of the block.

In DCT coding, the suffix plays a very important role as an error in the suffix results in error propagation to the next block. The number of suffixes in the image is also a key factor in reconstructing the image.

Assume there are *X* suffixes in a packet, and *Y* suffixes detected. We perform the following to guarantee the reconstruction of the image.

1) $X < Y$. Discard the extra ones at the end of the packet.

2) $X > Y$. Put zeros at the end of the packet to fill up the absent suffixes.

After the above are performed, the sign bits will then be matched to the decoded codewords. Due to the error speculation, we may have inserted or deleted some LEVELs in the packet; therefore, there may be too many or too few sign bits. For simplicity, if there are too many sign bits, we simply discard the extra bits; if there are too few sign bits, we deem the remaining LEVELs to be positive.

Now that we have made the necessary modifications to the UVLCs and several images have been transformed using 8×8 DCT, zig-zag scanned and then run-length coded, the RUNs and LEVELs are then coded using the original UVLC and the ALT.

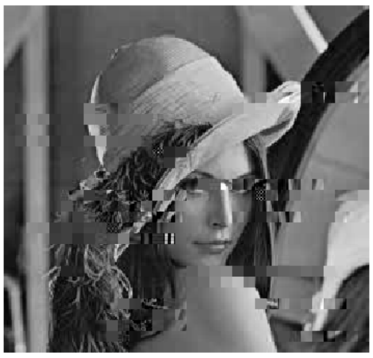
These coded images are subjected to a BSC with a BER of 10^{-3}. The PSNR of the reconstructed images are then compared in Table 3-11.

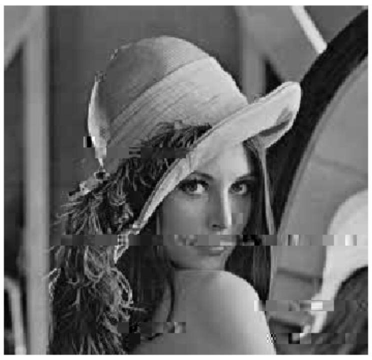
Image	PSNR of UVLC coded image (dB)	PSNR of ALT coded image (dB)
Lena	21.92	27.50
Cameraman	24.23	26.06
Monkey	17.81	22.38
House	27.67	30.07

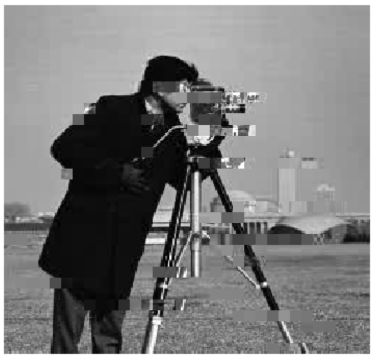
Table 3-11 Comparison of PSNR

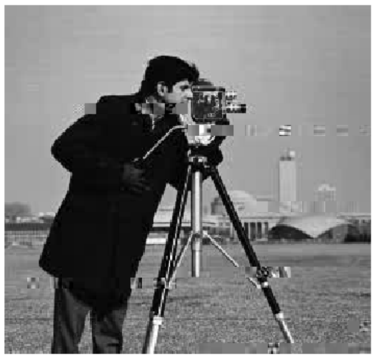
From Table 3-11 we see that the ALT coded images are always better than the original UVLC coded ones. The PSNR increases by approximately 2 ~ 5 dB.

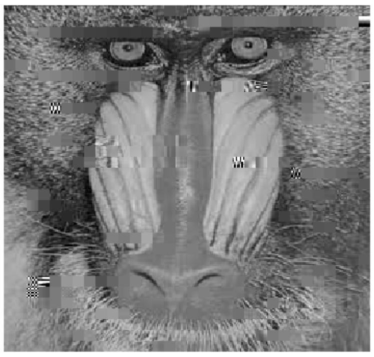
Figure 3-13 shows the comparison of the visual qualities of the images in Table 3-11. The qualities of the ALT coded images are evidently better.

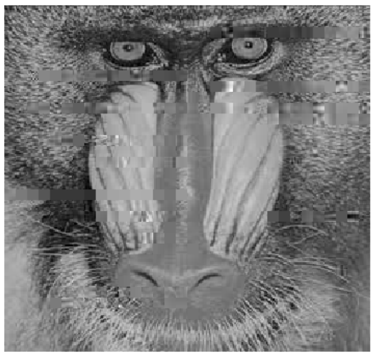

(a). The reconstructed Lena Using UVLC

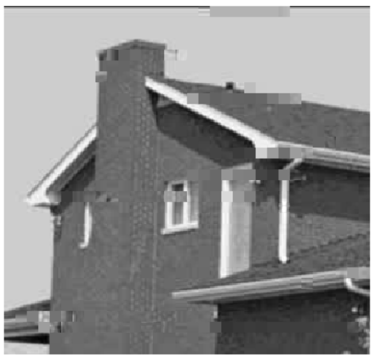

(b). The reconstructed Lena using ALT


(c). The reconstructed Cameraman using UVLC


(d). The reconstructed Cameraman using ALT


(e). The reconstructed Monkey using UVLC


(f). The reconstructed Monkey using ALT


(g). The reconstructed House using UVLC

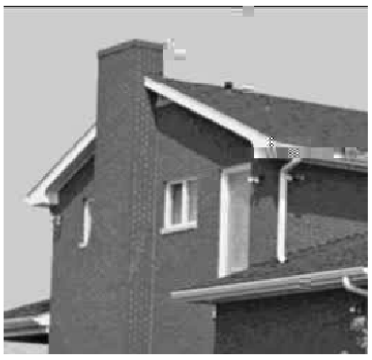

(h). The reconstructed House using ALT

Figure 3-13 Comparison of the visual quality of reconstructed images

From this application it can be seen that the ALT coding is able to increase the error resiliency of UPC codes. However, this is for only those UPCs that are of fixed suffix length. For UPCs with variable suffix length, improvement of error resiliency is still possible yet not very evident, and it is done at a cost of a lowered coding efficiency. This is because we need to make the UPC suffixes bi-directionally decodable and that usually requires the lengthening of the suffixes.

Although for these UPCs the error resiliency is not a large advantage, we will see in the next chapter that the decoder architecture and decoding speed could be greatly improved by using the ALT coding method.

To demonstrate the simplification in the decoding procedure enabled by ALT coding, we use the ALT approach to accelerate the variable length decoding of the run length coded image data in the JPEG standard in the Nios II embedded processor for Altera FPGA implementation, by using customized instructions [71]. We replace the VLC code table in the JPEG standard with the ALT coded data and implement customized instructions to accelerate the decoding. The ALT coding is again applied to the reversible EG codes UVLC and the UVLC is used to replace the VLC in the JPEG standard. The results show that software accelerated implementations enabled a speed-up by a factor in excess of 3.5 compared to the standard JPEG software implementation. Table 3-12 below shows the decoding performance for software implementations of the standard and the ALT coded JPEG. The acceleration in decoding is very obvious.

VLC decoder	Cycles/codeword	Cycles/coefficient
Standard JPEG	783	56.0
ALT coded JPEG	212	15.9

Table 3-12 VLC decoding throughput

3.4 THE PROS AND CONS OF ALT CODING

In the chapter we have introduced the ALT coding method and its applications. To conclude the chapter, there follows a discussion concerning the pros and cons of the ALT coding method.

The ALT coding method is designed on the basis of the UPC codes. It takes advantage of the simple structure of the unary prefixes of the UPC codes. By separating the prefixes and suffixes, and assigning alternating coded all-one and all-zero codes to the prefixes, codeword boundaries are very easily detected using xor logics. This breaks the bottleneck in the decoding of a common VLC, and enables parallel decoding structure to be implemented. The ALT method collects the alternating coded prefixes in a prefix sub-sequence. This sub-sequence is held in a very regular structure, which enables Error Speculation to be implemented for UPCs with fixed suffix length, thus assisting in the improvement of the error resiliency.

However, on the other hand, ALT coding performs a further separation of a code packet. This demands the decoding to be done in the unit of packets which requires buffering. Although this is not a problem in most cases since buffering is always necessary, buffering in the unit of packets is still required. For each packet, packet information about how many codewords are contained in an ALT packet is usually required, which adds additional bit costs. Error Speculation mechanism may help to increase the error resiliency, but on the other hand, it could also increase the computational complexity.

In general, the ALT method's biggest advantage is the simplification in the decoding and decoder structures thanks to the simple structures of the unary prefixes. In the next chapter, we will introduce the ALT decoders which are built on the basis of the ALT coding method.

4 ALT DECODER

In the previous chapter, we discussed how the ALT encoding is able to increase the error resiliency. We also mentioned that the ALT coded prefix sub-sequence enables a simple means of detecting codeword boundaries. By extracting code length information from the prefixes of the UPCs, the ALT coding method helps to break the dependencies between the variable length codewords and thus enables simplifications in the decoding procedure. We have discussed the decoding of the ALT packet in algorithms; however, the real advantage brought by the ALT coding is the great improvement in the hardware architecture of the UPC decoders.

In this chapter, we first introduce the ALT decoder architecture. Then the discussion is followed by comparisons of the ALT decoders to the general VLC decoder. The advantages and disadvantages of the ALT decoders will then be studied.

4.1 THE VLC DECODER STRUCTURES

Since the UPCs are all variable length codes, it is natural to use a general VLC decoder to decode a UPC code sequence. Therefore before starting to discuss the ALT decoder structure, we must firstly look at the general VLC decoders.

In the introductory part of this thesis, we have briefly discussed the basic VLC decoder architecture. We know that the VLC decoding requires sequential performance and therefore it is difficult for the VLC decoder to be pipelined or parallelized. Various decoding methods using parallel or pipelined architecture have been developed to reduce the decoding time. One early work on the VLC decoding was the tree-based searching algorithm of MARVLE [22], [23] which was able to decode the input code sequence serially at a speed of one bit per cycle. Therefore, the decoding time depended on the code length, i.e., longer codewords required longer decoding time. Sun and Lei [24] developed a bit-parallel decoder which could decode each code word in one clock cycle by parallel matching the current code sequence with all possible code words in a LUT. Sun and Lei [26], [27] then further improved the bit-parallel decoder by excluding an accumulator from the feedback path of the bit-parallel decoder. The works in [25] analyzed the PLA-based pipelined tree-based architecture, which combined several technologies such as flexible operation in the decoding process and the high-level optimization based on the Sun and Lei's architecture. [33] proposed two methods to create concurrency and to improve the decoder throughput:

1) The concurrent finite state machine (FSM), which extended the tree-based searching algorithm to the FSM.
2) The bit-positioning method, which divided the code sequence into blocks with overlapping windows.

The divided code sequences were decoded concurrently using Sun and Lei's decoder as a basic decoding unit, and the decoded data merged during the final stage. Such division makes the parallelization possible but the computational complexity is very high.

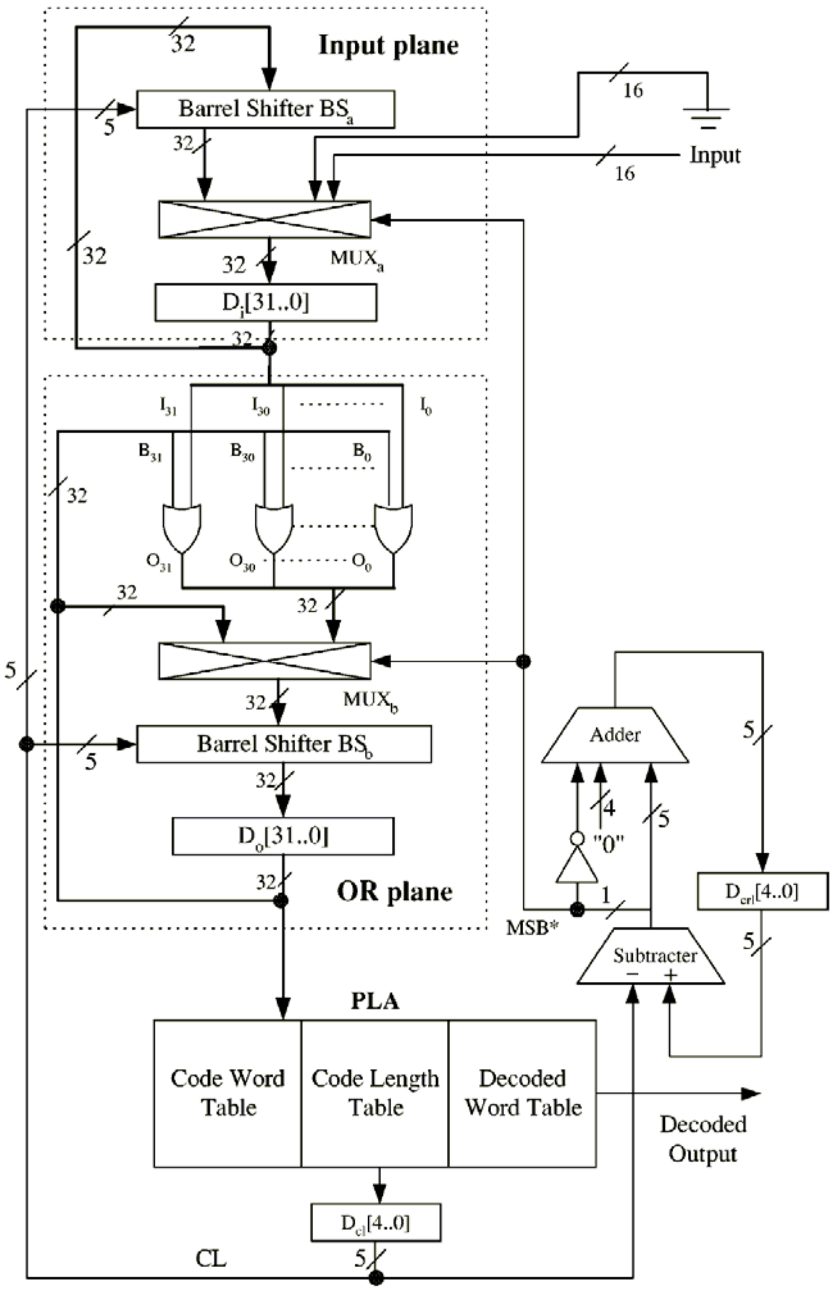


Figure 4-1 The PLS decoder

In [28], a fast VLC decoder using plane separation (PLS) was proposed. The architecture of the PLS decoder was based on the separation of an input-plane and an or-plane. By doing so, the decoder could perform input and decoding concurrently. Figure 4-1 shows the block diagram of the PLS decoder.

The decoder consists of two separate planes of an input plane. Each plane consists of a barrel shifter, a 32-bit 2:1 multiplexer, and a 32-bit output latch (BS_a, MUX_a, and D_i for input plane, and BS_b, MUX_b and D_o for the or-plane). The output data (D_o) from the or-plane (BS_b) is matched with all possible codewords in the codeword table. A match symbol and the corresponding latched code length in D_{cl} are obtained from the matching process. After the matching process, the input plane rotates the data in D_i at the BS_a and the or-plane shifts the data in D_o at BS_b both to the left direction by the amount of the latched code length. Bits shifted out to the left side of the or-plane are lost, while those of the input plane are attached to the least significant bits of the input plane. At the same time, the length of the remaining data in the or-plane is calculated. If the remaining bit length is smaller than the required code length (which is the maximum possible code length) for the next matching, it can only be performed after updating the or-plane by loading the next input code sequence. This can be simply performed by the bitwise or-operations of D_i and D_o, as shown in Figure 4-1. If the remaining bit length is larger than or equal to the required bit length, the next matching process can be repeated without the or operations

By decoding the sample images in MPEG-2 video sequences, the authors claimed that the PLS decoder reduces the required total processing time by approximately 30% compared to those of Sun and Lei's decoder and their modified decoder.

The VLC decoders discussed above were mainly aimed at high throughput and power dissipation was not a focus.

Various levels of design techniques have been employed in designing low power VLC decoders. The most effective approach to lowering the power consumption is by reducing the supply voltage. However, as the supply voltage is reduced, the propagation delay of the circuit increases, limiting the amount of voltage scaling under a particular throughput constraint. From this voltage scaling point, the parallel method is preferred to the tree search method since the parallel approach processes multiple bits per cycle. In other words, the parallel architecture can be run at a slower clock frequency and lower voltage than the serial method for a given throughput. At a higher level, the parallel VLC decoder can be decomposed into two components: the VLC detector which involves the shifting of data, and the LUT. The VLC detector receives the input VLC sequence and generates an address for the LUT. To reduce additional circuit overheads, address generation is simply an alignment of the VLC's at a fixed position so that the LUT uses the VLC itself as the address. The LUT receives the address from the VLC detector and produces the corresponding output codeword and length. The length is stored in an accumulator, which informs the shifter part how many bits have been used. As we have seen in the previous VLC decoder examples, to achieve a low power operation, both parts of VLC decoder need to be optimized.

A popular method for increasing the efficiency of the LUT is called the "prefix-predecoding". In cases where the number of codewords in the table is large, codewords always exist with common prefixes in the code table. By exploiting these common prefixes, the size of the LUT could be reduced. Several approaches have exploited for this prefix-predecoding method to efficiently decode the VLCs [29], [30], [31]. The basic idea of prefix predecoding is to group the VLCs by their common prefixes. In such decoders, the LUTs are separated into several block LUTs. The common prefixes and the short codewords are stored in one block LUT and the grouped suffixes with common prefixes are stored in several different block LUTs. The VLC sequence is decoded using the following steps: Firstly, the VLC is fed to the block LUT where prefixes and short codewords are stored. If the VLC is a short code without any prefix, then the output codeword is produced from this LUT and the next VLC is ready to be decoded. If the input VLC is one of the long codes with a prefix, only the prefix is decoded in this LUT. The remainder of the VLC is decoded in one of the subsequent blocks where the grouped suffixes are stored. With this prefix- predecoding method, the size of the LUT is reduced because the prefixes are no longer redundant in the LUT. The authors in [32] pointed out that, for MPEG-2 DCT AC coefficients, the majority of VLCs can be clustered by their prefixes and a greater than 50% area reduction can be achieved compared to a single table. In addition, such prefix-predecoding methods could also help to reduce the power consumption by up to a factor of two since the switched capacitance is also reduced [32]. Often, the LUTs are not only the most power intensive blocks but also the most area occupying component blocks of a VLC decoder. According to [32], in an MPEG-2 VLC decoder system, approximately 80% of the area is consumed by the LUT. Therefore, reducing the LUT size, not only helps to reduce power consumption, but also helps to make the decoder compact in size.

4.2 THE GENERAL ALT DECODER STRUCTURE

The ALT decoder, designed specifically for the ALT coded UPCs, do not follow the general concept of the VLC decoders. As has been described in the previous chapter, the ALT coding resorts the prefixes and the suffixes of an entire UPC packet, collects the prefixes and replaces the prefixes with alternating coded all-one and all-zero codes. The advantage of these all-one and all-zero codes is that the codeword boundaries are easily detected by a row of xor logics. The xor logic can be seen as the key part of the ALT decoding. This follows directly from the alternating coded all-one codes and all-zero codes. Figure 4-2 shows an example of detecting prefix boundary by a row of xor operations.

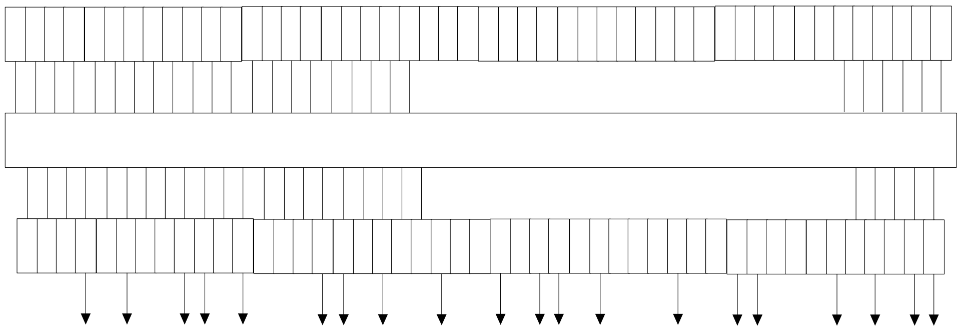


Figure 4-2 Detecting prefixes by a row of xor operations

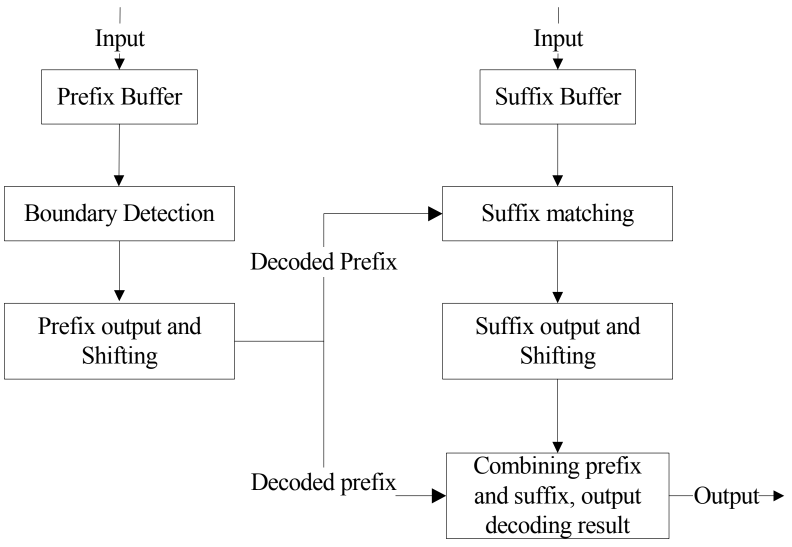
In Figure 4-2, every bit of the prefix sub-sequence is connected to a xor gate, only when a prefix boundary occurs, will the output of the xor gate yield a "1". Therefore, via the row of the xor gates, we are able to detect the boundaries of the prefixes fairly easily. When the boundaries of the prefixes are marked, we are able to calculate the lengths of each prefix, and since the prefixes are simply some unary codes, its length uniquely identifies each prefix. Thus the decoding of the prefixes is complete. With the decoded prefixes, the associated suffixes, no matter whether they are fixed length codes or variable length codes, are then able to be decoded by some simple calculation or by searching and matching the LUTs.

For UPCs with fixed length suffixes, the suffix decoding is basically just a suffix sub-sequence buffer plus a shifting unit used to output the decoded suffixes according to each suffix length. For UPCs with variable suffix lengths, the suffix decoding involves an ordinary VLC decoding procedure, which includes LUT searching as well as the shifting mechanism.

After both the prefixes and the suffixes are decoded, we still need to combine the two decoding results to decode the original symbol. For the UPCs, as we have seen, they are usually applied to the integer sources, and therefore the coded symbols are usually some integers. For UPCs with fixed length suffixes, combining the prefixes and suffixes to decode the original integer could be easily done by simple arithmetic calculations, as we have already seen in chapter two. For UPCs with variable suffix length, it again involves LUT searching; however, the prefixes already provide the LUT searching with a certain index and thus the searching is more efficient. Moreover, for some UPCs, there are common LUTs that could be used by different groups of codewords that are indexed by the prefixes. Such common LUTs also help to reduce the size of the LUT and in turn help to reduce the size of the decoder.

We could see here that, for UPCs with fixed length suffixes, the LUTs are completely eliminated, which is a big boost for the performance of the decoder, whereas the UPCs with variable suffix length are not as greatly improved. In this chapter, emphasis will be placed on the ALT decoders designed for fixed suffix length UPCs, namely, GR and EG codes; however a common structure for an ALT decoder will be given.

For any ALT coded UPC packet, the ALT decoder could be separated into two sub-decoders, corresponding to the two sub-sequences in an ALT packet. Moreover, another Figure 4-3 shows the function diagrams of the ALT decoder.



Input
Prefix Buffer
Boundary Detection
Prefix output and Shifting
Input
Suffix Buffer
Suffix matching
Decoded Prefix
Suffix output and Shifting
Decoded prefix
Combining prefix and suffix, output decoding result
Output

Figure 4-3 Function diagram of an ALT decoder

4.2.1 The prefix sub-decoder

For all ALT decoders, the prefix sub-decoders are primarily of the same structure. The prefix sub-decoder is the key part of the ALT decoder. It is the part where the ALT decoder differs from traditional VLC decoders. The performance boost of the ALT decoder comparing it to the traditional VLC decoders is mainly due to the simplification in the decoding of the prefixes. The decoder architecture of the prefix sub-decoder is illustrated in Figure 4-4. Here we have discarded the peripheral structures of the decoder, such as the packet buffers. We also assume the maximum code length of the prefix to be 16 bit. Customizing the prefix sub-decoder with different maximum prefix lengths could be simply done by scaling the devices.

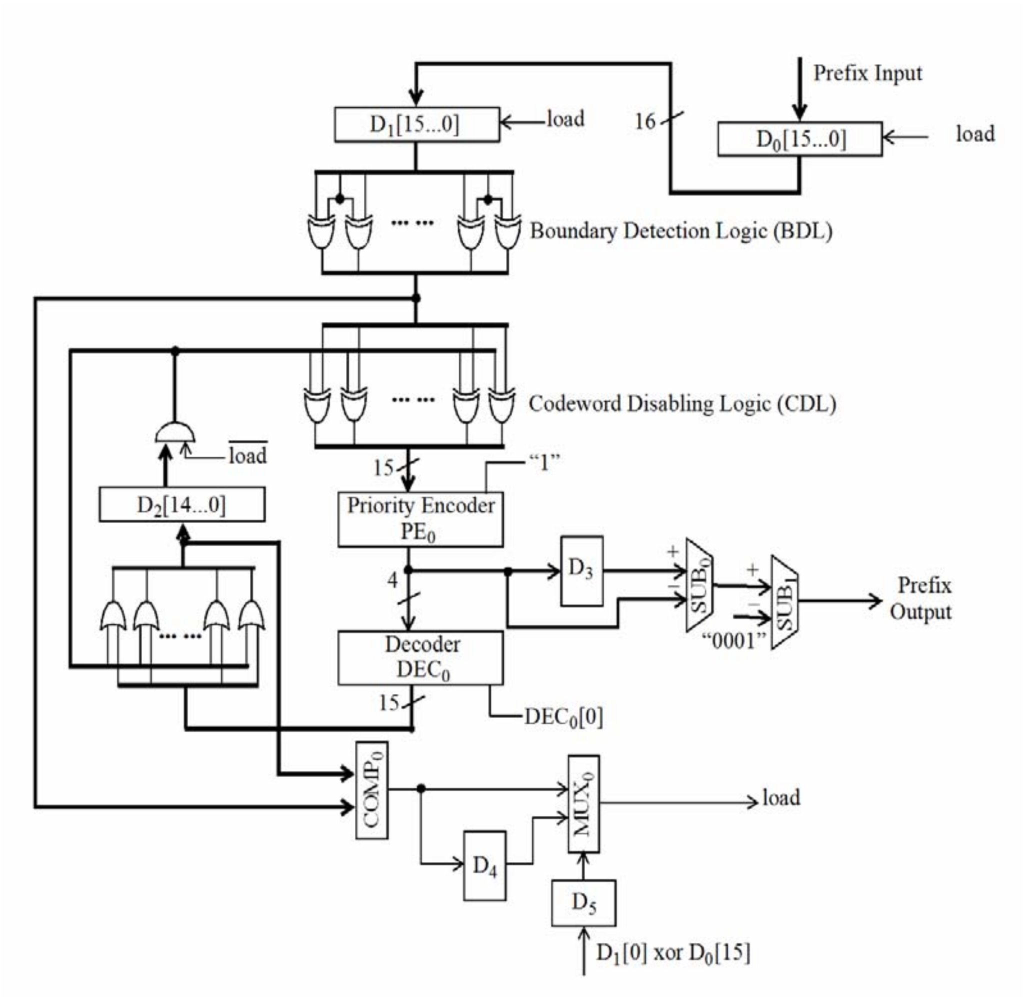


Figure 4-4 General architecture of an ALT prefix sub-decoder

The prefix sub-decoder could be divided into several functional blocks as indicated in Figure 4-4. They are: the input buffers, the Boundary Detection Logic (BDL) which simply consists of a row of xor-gates, the Codeword Disabling Logic (CDL) which is used to disable the decoded prefix boundaries in each clock cycle, the prefix decoding logic, which decodes the prefix lengths representing each of the unary prefixes. The sub-decoder is a very simple and small design; there are neither LUTs nor shifting scheme included in the entire sub-decoder.

The function of the prefix sub-decoder is to generate the length of each prefix and to provide it as a reference point in decoding the suffixes. Bearing in mind that the maximum prefix length is 16 bits, to represent the lengths of the prefixes, we need 4-bits. Therefore, the decoder consists of one 16-to-4 priority encoder (PE_0), one 4-to-16 decoder (DEC_0), two 16-bit buffers (D_0 and D_1), one 15-bit register D_2, one 4-bit register D_3, one 15-bit comparator ($COMP_0$), two 4-bit subtractors (SUB_0 and SUB_1), one 1-bit 2:1 multiplexer (MUX_0), and two 1-bit registers (D_4 and D_5). Since each xor gate has two bits input, the 16 bit buffer thus results in a 15 bit output after the row of xor gates, and thus there are 15-bit registers and comparators.

Now let us look at how the sub-decoder functions. The prefix sub-sequence of the prefix sub-decoder is put into the two buffers D_0 and D_1, the first two bytes in D_1 and the second two bytes in D_0. The first two-bytes of the prefixes are then fed into the xor-gates in the "Boundary Detection Logic" (BDL) where two consecutive bits are xored with each other. As the prefixes are now denoted in alternating all-one and all-zero codes, only at each prefix boundary will a "1" be generated by the xor operations as indicated in Figure 4-2. Each "1" then indicates a prefix boundary. The output after the BDL is then fed into the priority encoder PE_0 in order to generate the position of the first OIB boundary. Register D_3 is originally loaded with the number 16 (that is "0000" in a 4-bit binary code). The length of the first prefix is then calculated by SUB_0 and at the same time D_3 is updated with the position of the first prefix boundary. The 4-to-16 bit decoder DEC_0 generates the position of the first prefix boundary and disables the first "1" of the input of the priority encoder by using the or-gates and the "Codeword Disabling Logic" (CDL). In the next clock cycle, the second prefix boundary is encoded by PE_0. Again the second prefix boundary is placed in D_3 and its position is decoded by DEC_0. The same operations are then repeated. Thus the length and the prefixes are generated as well as the output for decoding and further reference.

Let us look at a simple example to see exactly how each functional block works. Assume we have a prefix sub-sequence:

{00, 1111, 0, 1, 0, 111, 0, 11111, 0000, 1, 0, 1, 0, 1, 00, 1111, ...}

Table 4-1 illustrates how the prefix decoder decodes the prefix sub-sequence. Suppose the prefix sub-decoder was initialized to all zeros before D_1 is loaded with data. When "load" is set to high, D_1 and D_0 are loaded with

"0011110101110111" and "1100001010100111" respectively. Then $D_1[0]$ xor $D_0[15]$ is set to low, which indicates the last codeword in D_1 continues in D_0.

Clock cycle 0	D_1_out	0011110101110111	PE_0_out	1110
	BDL_out	010001111001100	D_3_out	0000
	CDL_out	010001111001100	SUB_0_out	0010
	DEC_0_out	0100000000000000	OIB_out	0001
	D_2_in	010000000000000	$COMP_0$_out	0
	D_2_out	000000000000000	load	0
Clock cycle 1	D_1_out	0011110101110111	PE_0_out	1010
	BDL_out	010001111001100	D_3_out	1110
	CDL_out	000001111001100	SUB_0_out	0100
	DEC_0_out	0000010000000000	OIB_out	0011
	D_2_in	010001000000000	$COMP_0$_out	0
	D_2_out	010000000000000	load	0
Clock cycle 2	D_1_out	0011110101110111	PE_0_out	1001
	BDL_out	010001111001100	D_3_out	1010
	CDL_out	000000111001100	SUB_0_out	0001
	DEC_0_out	0000001000000000	OIB_out	0000
	D_2_in	010001100000000	$COMP_0$_out	0
	D_2_out	010001000000000	load	0
...	**... ...**			
Clock cycle 6	D_1_out	0011110101110111	PE_0_out	0011
	BDL_out	010001111001100	D_3_out	0100
	CDL_out	000000000000100	SUB_0_out	0001
	DEC_0_out	0000000000001000	OIB_out	0000
	D_2_in	010001111001100	$COMP_0$_out	1
	D_2_out	010001111001000	load	1
Clock cycle 7	D_1_out	1100001010100111	PE_0_out	1110
	BDL_out	010001111110100	D_3_out	0011
	CDL_out	010001111110100	SUB_0_out	0101
	DEC_0_out	0100000000000000	OIB_out	0100
	D_2_in	010000000000000	$COMP_0$_out	0
	D_2_out	000000000000000	load	0
...	Decoding continues			

Table 4-1 Example of the decoding procedure of prefix sub-decoder

From the table we see that, the first seven prefixes are decoded within the first seven clock cycles, after the seventh prefix is decoded, the first two bytes in D_1 are decoded and new data is required to be loaded from D_0. The loading is done by setting the "load" signal and those bits left in the first two bytes but not yet decoded could be converted directly to the prefix length calculation. Thus, on receiving the new data for the prefixes, the remainder of the last prefix in the first two bytes could be added directly to the left over value and therefore will not cost an extra clock cycle. Therefore, we could guarantee a constant decoding rate of one prefix per clock cycle for this prefix sub-decoder.

As mentioned previously, the prefix sub-decoder takes advantage of the alternately coded all-one and all-zero codes, thus greatly simplifying the detection of the codeword boundaries with no table look-up necessary in the prefix sub-decoder and thus also no sequential operations. This makes it very easy to parallelize the decoding procedure and thus accelerate the decoding speed. However, parallelization of the prefix sub-decoder needs to work in accordance with the suffix sub-decoder in order to complete the decoding, so parallelization could not be achieved for all UPCs. This is only possible for those highly structured UPCs such as the GR codes, which have fixed suffix length independent of the prefixes. In other UPCs, the lengths of the suffixes are variable or are decided by the length of the prefixes, therefore the suffix sub-decoder works as a traditional VLC decoder or works in a similar way, and parallelization is difficult.

4.2.2 The suffix sub-decoder and decoding of the entire UPC

Since the suffixes of different types of UPCs are different in nature, the suffix sub-decoder needs to be able to accommodate these. The differences between these suffix sub-decoders include more than the two categories of UPCs, namely, UPCs with fixed length suffixes and UPCs with variable length suffixes, which have been discussed until now. But in general, the suffix sub-decoders can be divided into three categories.

1) Suffix sub-decoder for UPCs with fixed suffix lengths which are not associated with the prefixes. For instance, the GR codes with arbitrary suffix length where the prefix conveys no information about the suffix.
2) Suffix sub-decoders for UPCs with fixed suffix length completely associated with the prefixes. These codes include the EG codes and all UPCs whose suffixes have linear relationships with the prefixes.
3) Suffix sub-decoders for UPCs with variable length suffixes. These include HG codes and UPH codes and all UPCs with variable lengths.

These suffix sub-decoders will now be discussed separately.

For the suffix sub-decoder in the first category, decoding the suffixes is very simple because they are simply fixed-length codes and their lengths are known. Also since the suffixes are actually independently fixed-length codes, we do not need the prefixes to serve as references. Therefore the decoding of suffixes is simply a fixed-length decoder. What is even simpler is that, since UPCs usually encode integers, the decoding of suffixes is actually unnecessary, as the requirement is to output the suffixes and use them later in the arithmetic operations when the encoded integers are to be decoded. Table 4-2 gives an example of how to generate the encoded integer using only prefix lengths and the suffix of a set of GR codes. As shown in chapter two, when the prefix length *k* is given, the encoded integer value could be calculated using 2-x. Thus we are able to recover the encoded integer value using only simple multiplications and additions.

Suppose the length of each suffix is *k*-bits, all that would be in the suffix sub-decoder would be a *k*-bit register. The *k*-bit register outputs a *k*-bit suffix every clock cycle and by combining this *k*-bit suffix with the decoded prefix in each clock cycle, it is possible to further decode the entire UPC.

Valu e	GR	Unary prefix	Prefix Length *k*	Suffix	Suffix Length	Value=(*k*-1)*2^2+Suffix
0	000	0	1	00	2	(1-1)*4+0=0
1	001	0	1	01	2	(1-1)*4+1=1
2	010	0	1	10	2	(1-1)*4+2=2
3	011	0	1	11	2	(1-1)*4+3=3
4	1000	10	2	00	2	(2-1)*4+0=4
5	1001	10	2	01	2	(2-1)*4+1=5
6	1010	10	2	10	2	(2-1)*4+2=6
7	1011	10	2	11	2	(2-1)*4+3=7
8	11000	110	3	00	2	(3-1)*4+0=8
9	11001	110	3	01	2	(3-1)*4+1=9
10	11010	110	3	10	2	(3-1)*4+2=10
11	11011	110	3	11	2	(3-1)*4+3=11
12	111000	1110	4	00	2	(4-1)*4+0=12
13	111001	1110	4	01	2	(4-1)*4+1=13
14	111010	1110	4	10	2	(4-1)*4+2=14
15	111011	1110	4	11	2	(4-1)*4+3=15
…	…	…	…	…	…	…

Table 4-2 GR code with suffix length 2

For the suffix sub-decoders in the second category, decoding becomes more complex because the suffix lengths are now linearly associated with the prefixes. The length of each suffix varies with the length of the corresponding prefix. Therefore, although we consider such UPCs to be fixed suffix codes, in fact their suffix length is variable, but in a way that is completely linked to the length of the prefix. Thus to decode a suffix, we need the decoded prefix to act as a reference. For these UPCs the length of each suffix could be easily be generated from the corresponding prefix and the LUTs are not required in order to perform the decoding. What must be added, when comparing this to the suffix sub-decoders in the first category, is a shifting scheme to remove the suffixes already decoded. The fixed *k*-bit register is no longer able to perform this task because now the suffix lengths vary with the prefixes. Figure 4-5 shows an example of the suffix sub-decoder for EG codes with *k*=0. Here we assume that the maximum suffix length is 15 bits.

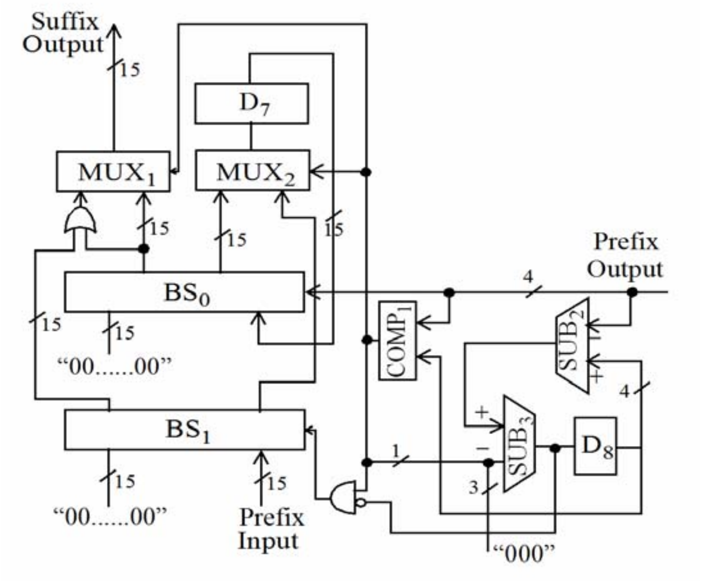


Figure 4-5 Example of EG suffix sub-decoder (*k*=0)

The suffix sub-decoder functions as follows.

The sub-decoder in Figure 4-5 contains one 15-bit register (D_7), one 4-bit register D_8, two 15-bit 2:1 multiplexers (MUX_1 and MUX_2), two 30-bit barrel shifters (BS_0 and BS_1), two 4-bit subtractors (SUB_1 and SUB_2), and a 4-bit greater than & equality comparator ($COMP_1$).

The suffixes are first loaded in the lower half of the two barrel shifters, the first 15 bits in BS_0 and the following 15 bits in BS_1. The upper half of the barrel shifters are both loaded with 15 bit zeros. D_8 is originally loaded with 15 ("1111" in binary), which is the maximum suffix length. BS_0 shifts the suffix series to its upper half according to the first suffix length generated from the prefix sub-decoder. The first suffix is then generated from the upper half of BS_0. At the same time, SUB_2 outputs the length of the rest of the suffix sub-sequence after the first suffix has been shifted out. This length is stored in D_8, to be used for the decoding of the next suffix. In the next clock cycle, the lower half of BS_0 is loaded with the shifted suffix series and the upper half is cleared into all zeros. Therefore the decoding of the next suffix can be performed. The same operations are then repeated. When suffix decoding is performed until the end of the first 15 bits is reached, the length of the suffix sub-sequence left in BS_0 will be equal to or smaller than the length of the next suffix. This will make the output of $COMP_1$ become "1". A new 15-bit suffix sub-sequence will then be loaded to the suffix input. The contents of BS_0 and BS_1 are both shifted according to the length of the next suffix. The two separated parts of the last suffix in BS_0 can be merged by the or-gates and MUX_1 so that the complete suffix can be generated. MUX_2 is used to load new data into BS_0. Decoding can then be performed continuously.

Now we use an example to demonstrate how this sub-decoder works. Suppose we have a suffix sub-sequence {10010010101110100......} to be decoded. Table 4-3 shows the performance of the decoding in the suffix sub-decoder.

Clock cycle 1	BS_1_out	000000000000000$\mathbf{X_{14}X_{13}}$ … … $\mathbf{X_0}$	Prefix_out	0001
	BS_0_out	0000000000000001001001010111010	$COMP_1$_out	0
	D_7_out	000000000000000	SUB_2_out	1110
	Suffix_out	000000000000001	SUB_3_out	1110
Clock cycle 2	BS_1_out	000000000000000$\mathbf{X_{14}X_{13}}$ … … $\mathbf{X_0}$	Prefix_out	0011
	BS_0_out	00000000000000100101011101 0000	$COMP_1$_out	0
	D_7_out	001001010111010	SUB_2_out	1100
	Suffix_out	000000000000001	SUB_3_out	1011
Clock cycle 3	BS_1_out	000000000000000$\mathbf{X_{14}X_{13}}$ … … $\mathbf{X_0}$	Prefix_out	0000
	BS_0_out	0000000000000000010101110100000	$COMP_1$_out	0
	D_7_out	001010111010000	SUB_2_out	1100
	Suffix_out	000000000000000	SUB_3_out	1011
…	… …			
Clock cycle 15	BS_1_out	000000000000000$\mathbf{X_{14}X_{13}}$ … … $\mathbf{X_0}$	Prefix_out	0001
	BS_0_out	000000000000000100000000000000	$COMP_1$_out	0
	D_7_out	010000000000000	SUB_2_out	0010
	Suffix_out	000000000000000	SUB_3_out	0001
Clock cycle 16	BS_1_out	0000000000$\mathbf{X_{14}X_{13}}$ … … $\mathbf{X_0}$00	Prefix_out	0011
	BS_0_out	000000000000100$\mathbf{X_{12}X_{11}}$ … … $\mathbf{X_0}$00	$COMP_1$_out	1
	D_7_out	100000000000000	SUB_2_out	1100
	Suffix_out	000000000001$X_{14}X_{13}$	SUB_3_out	0010
…	Decoding continues … …			

Table 4-3 Example of a suffix decoding process

In Table 4-3, $X_{14}...X_0$ indicates the new data arriving after the first 15 bits. In our example, $X_{14}X_{13}$ are actually "00".

The suffix sub-decoder outputs the suffixes as they are, the actual integer to be decoded still need to be handled. For the UPCs in the second category, decoding of the integer is similar to that in the first category. No LUTs are needed because the suffixes are completely associated with the prefixes and therefore the decoding could also be done by manipulating the prefixes and the suffixes arithmetically. Hence, the decoding of the integers could also be simplified and accelerated at this level.

Table 4-4 gives an example of how to decode the coded integer value using arithmetic operations on the prefixes and the suffixes. The example is a set of EG codes with *k*=0. As shown in this table, the decoding of the actual integer could be performed using a set of arithmetic calculations

Value	EG	Unary prefix	Prefix Length m	Suffix	Suffix Length m-1	Value=2^{m-1}-1+Suffix
0	0	0	1	-	0	2^0-1+0=0
1	100	10	2	0	1	2^1-1+0=1
2	101	10	2	1	1	2^1-1+1=2
3	11000	110	3	00	2	2^2-1+0=3
4	11001	110	3	01	2	2^2-1+1=4
5	11010	110	3	10	2	2^2-1+2=5
6	11011	110	3	11	2	2^2-1+3=6
7	1110000	1110	4	000	3	2^3-1+0=7
8	1110001	1110	4	001	3	2^3-1+1=8
9	1110010	1110	4	010	3	2^3-1+2=9
10	1110011	1110	4	011	3	2^3-1+3=10
11	1110100	1110	4	100	3	2^3-1+4=11
12	1110101	1110	4	101	3	2^3-1+5=12
13	1110110	1110	4	110	3	2^3-1+6=13
14	1110111	1110	4	111	3	2^3-1+7=14
…	…	…	…	…	…	…

Table 4-4 EG code with parameter *k*=0

In later sections we will see applications of the ALT decoders where these calculations are implemented using code converters.

For the suffix sub-decoders in the third category, decoding becomes even more complex because the suffix lengths are not only variable in length, but also have no evident association with the prefixes. This forces the decoding of the suffixes to follow the manner of a VLC decoding procedure. However, this suffix VLC decoder is still greatly reduced in size and is much more efficient in LUT searching because part of the searching reference has already been decoded from the prefix sub-decoder. The LUT needed in the suffix sub-decoder is greatly reduced in size because only one prefix per prefix group needs to be stored. This actually yields a natural prefix-predecoding of the UPC packet.

Table 4-5 gives the LUT of a set of HG codes. Table 4-6 gives the LUT in the suffix sub-decoder in the ALT decoder.

n	LUT	
	Prefx	Suffix
0	0	-
1	10	-
2		0
3	110	10
4		11
5		00
6		01
7	1110	10
8		110
9		111
10	11110	000

Table 4-5 LUT of HG codes

Prefix length	LUT Suffix							
	-	0	00	01	10	110	111	000
1	0							
2	1							
3		2	3	4				
4			5	6	7	8	9	
5								10

Table 4-6 Reduced LUT in the suffix sub-decoder

In the original LUT, one HG codeword matches one integer, so the table increases with the number of codewords. In the reduced LUT, however, the shaded cells represent the cells that are absolutely necessary. We see here that a group of suffixes share one prefix and thus the prefixes do not need to be repeatedly stored. Hence the LUTs will be greatly reduced in size. The LUTs are used to search for the encoded integers, in the LUT in Table 4-5, the search must be accomplished by looking through all the codewords and matching them up with the buffered code sequence, which may lead to long delays because there is no previous knowledge concerning the whereabouts of the codeword in the LUT . Take the HG code in Table 4-5 and 4-6 as an example, suppose codeword 1110111 is to be decoded, in Table 4-5, we must search 10 rows for this code and then decode it as integer 8. However 1110110 has a prefix length of 4, and the suffix is 111, when we know that the prefix is of 4 bits, only the fourth row of the LUT needs to be searched and there are only 4 cells to be searched until it is possible to decode the integer 8. And as mentioned in section 4.1, a full codeword table also makes the decoder power intensive.

The average energy consumption per codeword in the LUT can be modeled by the following equation:

$$\overline{E}_{LUT} = \sum_{i=1}^{n} P_i \cdot E_i \tag{4.1}$$

Where $\overline{E}_{LUT}$ represents the average energy consumption per codeword in the LUT, n is the number of codewords in the LUT, P_i is the probability of the codeword i and E_i is the energy required to decode codeword i.

In conventional approaches, the energy required to decode a VLC in a LUT does not greatly vary over the codeword probability. (i.e., $E_i \approx$ constant, $\forall i$). This is because the VLC table is implemented in a single LUT and the whole table has to be charged and discharged every cycle. For the UPCs, however, the single LUT method does not exploit the fact that decoding the unary prefixes requires only a few xor operations and searching for the prefixes is unnecessary. Besides, such a single LUT method also neglects to take into account the frequent occurrence of short codeword. Therefore, the average energy in (4.1) is dominated by codewords which have high probability of occurrence.

For the ALT coded packet, to analyze the energy consumption of the LUT searching, we need to modify equation (4.1) into:

$$\overline{E}^{a}_{LUT} = \sum_{i=1}^{n} P_i (E_i^p + E_i^s) \tag{4.2}$$

where $\overline{E}^{a}_{LUT}$ stands for the energy consumed by the ALT decoder in the LUT searching, E_i^p represents the energy consumed by decoding the prefix, and E_i^s represents the energy consumed by decoding the suffix. In the ALT decoder, the prefixes are decoded simply by a row of xor operations, and the LUT does not require to be searched for prefixes, thus we can regard the energy consumed by decoding the prefixes to be almost zero (i.e., $E_i^p \approx 0, \forall i$). It is obvious that

$\overline{E}^{a}_{LUT} < \overline{E}_{LUT}$ since part of the LUT search is almost completely eliminated by the separation of the prefixes and suffixes.

On the other hand, the ALT decoder also provides a low power approach that exploits the variable length codeword statistics, making the energy to decode a codeword dependent on the codeword probability. This is achieved naturally by also decoding the prefix and the suffix separately. Due to the nature of the high peak, heavy tail sources that are suitable for the UPCs, the UPCs with variable length codes, such as the HG code, UPH codes and the modified UPH codes, shorter prefixes are usually associated with shorter suffixes and thus fewer codewords. The HG code set in Table 4-5 is an example. Thus, by extracting the prefixes, we make a natural partition of the original single LUT into several variable size tables with respect to their energy consumption and frequency of occurrence. By having such variable size tables, the energy required to decode a codeword will vary according to the size of the partitioned table. Low power can be achieved if the dominant term in (4.1) is made small. For the UPC case here, shorter prefixes are associated to shorter suffixes and the numbers involved in this case are small. Therefore for the short codewords which have higher occurrence probabilities, the searching range is smaller, thus the dominant term in (4.1) could be reduced. Taking into consideration that the energy consumed by searching the prefix is almost 0, the energy consumption of an ALT decoder could then be expressed as:

$$\overline{E}^{a}_{LUT} = \sum_{i=1}^{n} P_i \cdot E_i^s = P_1^p \cdot E_{(1)}^s + P_2^p \cdot E_{(2)}^s + \ldots + P_r^p \cdot E_{(r)}^s \tag{4.3}$$

Here P_i^p, $1 \le i \le r$ is the occurrence probability of the prefixes, and r is the number of different prefixes in the code set. $E_{(i)}^s$ $1 \le i \le r$ is the energy required to search for a suffix in prefix *i*, and we know that for codewords with higher occurrence probabilities, $E_{(i)}^s$ is reduced by having a smaller search range, which obviously assists in the reduction of power consumption.

It can be seen from the above analysis that, even though for the UPCs in the third category, decoding in the suffix sub-decoder is more complex, we are still able to achieve faster decoding, smaller decoder size and less power dissipation. To decode the suffixes, as previously mentioned, involves a regular VLC decoding procedure, but the VLC decoding is greatly reduced in scale thanks to the separated prefixes.

As a matter of fact, any parallelization and optimization possible for a general VLC decoder could still be applied to the suffix sub-decoder. For instance, the rapid PLS VLC decoder mentioned in previous sections [28] could be modified and applied to the suffix sub-decoder.

In general, we see that, for any type of UPC, the ALT decoding could greatly assist in the reduction of the decoding complexity and efficiency. The resulting ALT decoders are also smaller, faster and less power consuming.

4.3 APPLICATIONS OF THE ALT DECODER

In the previous section we have discussed the architecture of the ALT decoder and have seen that by exploiting the ALT coding method, the decoder built on this basis could be greatly simplified leading to a reduction in size, power consumption and increase in decoding speed. In this section, we give examples of applications of the ALT decoding in the decoding of GR codes and one type of EG codes. A parallel decoding structure of the GR decoder is also proposed, which enables the decoding of multi-codewords per clock cycle.

4.3.1 An ALT decoder for GR codes

GR codes are very often encountered in image/video data, thus developing a special decoder for the GR codes will lead to improvements in the overall performance of the image/video decoder. As described in chapter one, there are different sets of GR codes depending on the suffix lengths. In order to examine the improvement of the ALT decoder, three designs were made for GR codes with 0 bit suffix, 1 bit suffix and 2 bit suffix. The designs are shown in Figure 4-6. The maximum prefix length is assumed to be 16 bit, which is sufficiently long for the image/video codes.

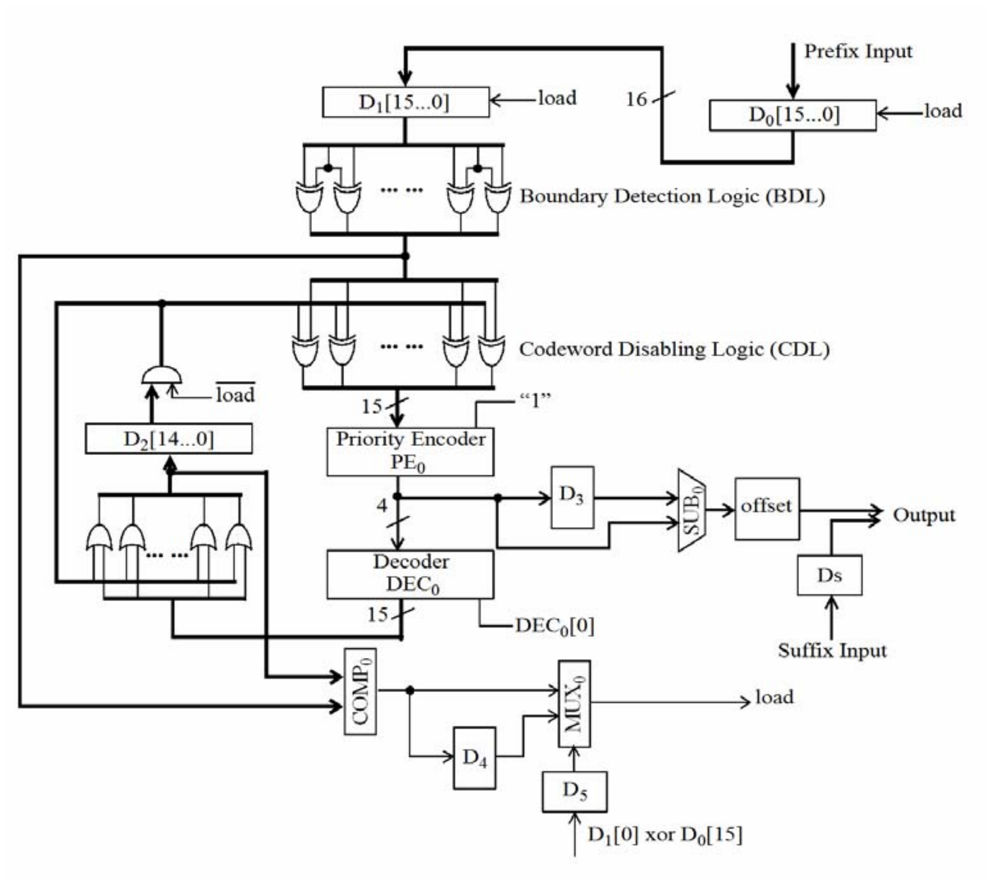


Figure 4-6 ALT decoder for GR codes

The ALT decoder has two inputs for the separated prefix and suffix sub-sequences. One is the prefix input and the other is the suffix input. The decoder consists of one 16-to-4 priority encoder (PE_0), one 4-to-16 decoder (DEC_0), two 16-bit buffers (D_0 and D_1), one 15-bit register D_2, one 4-bit register D_3, one 15-bit comparator ($COMP_0$), one 4-bit subtractor (SUB_0), one 1-bit 2:1 multiplexer (MUX_0), one n-bit register D_s (n is the length of the suffix) and two 1-bit registers (D_4 and D_5).

The prefix input of the decoder is put into the two buffers D_0 and D_1, the first two bytes in D_1 and the second two bytes in D_0. The first two-byte prefix series is then fed to the xor-gates in the "Boundary Detection Logic" (BDL) where two consecutive bits are xored with each other. As the prefixes are now denoted in alternating all-one and all-zero codes, only at each prefix boundary will a "1" be generated by the xor operations. Therefore, each "1" indicates a prefix boundary. The output after the BDL is then fed into the priority encoder PE_0 in order to generate the position of the first codeword boundary. Register D_3 is originally loaded with the number 16 (that is "0000" in a 4-bit binary code). The length of the first prefix is then calculated by SUB_0 and at the same time D_3 is updated with the position of the first codeword boundary. The 4-to-16 bit decoder DEC_0 generates the position of the first codeword boundary and disables the first "1" of the input of the priority encoder by using the or-gates and the "Codeword Disabling Logic" (CDL). In the next clock cycle, the second codeword boundary is encoded into PE_0. Again the second codeword boundary is put to D_3 and its position is decoded by DEC_0. The same operations are then repeated.

As discussed in chapter one, the prefix of a GR code is the unary expression of a quotient, which can itself be easily generated by offsetting the integer which represents the prefix length. Therefore, by offsetting the output of SUB_0, the value of the quotient can be generated. The suffix of a GR code is already a binary expression, so the actual integer a GR code represents can be generated simply by concatenating the suffix and the decoded prefix.

When decoding is performed until the end of D_1, the output of D_2 will then be accumulated to make it the same as the output of BDL, and the output of $COMP_0$ is set to high. The operation $D_1[0]$ xor $D_0[15]$ is used to find out whether the prefix in D_1 still continues in D_0. If the prefix continues, the "load" signal is generated immediately and new data are loaded into the buffers. If the end of D_1 is the end of a prefix, then the load signal needs to be delayed to the next clock cycle. A multiplexer MUX_0 and a 1-bit register D_4 are used to complete this.

To handle different lengths of suffixes, what requires to be changed is the size of the register D_s, which when handling GR codes without any suffixes, is completely eliminated.

This ALT decoder belongs to the first category discussed in the last section. The design in Figure 4-6 is almost exactly the same as for a general prefix sub-decoder. Neither look-up tables nor shifting scheme are necessary, and it is capable of decoding one codeword per clock cycle.

In order to determine the performance of this decoder, its performance is compared to the PLS decoder developed by Jae Ho Jeon et al [28]. Their decoder is scaled for the GR codes that are studied. Figure 4-7 shows the modified PLS decoder.

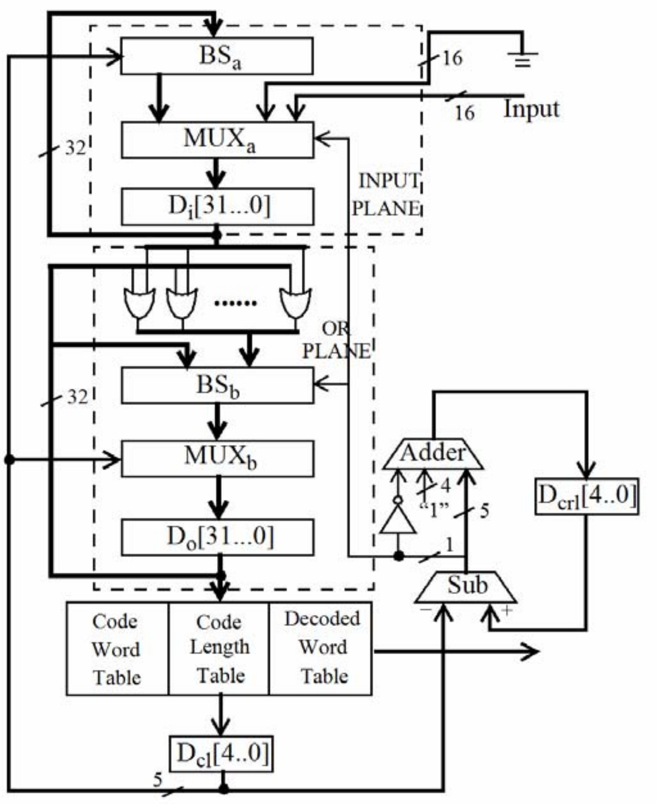
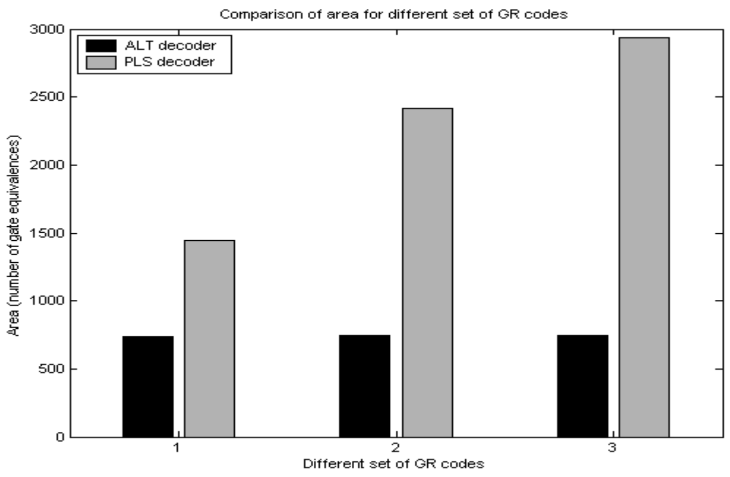
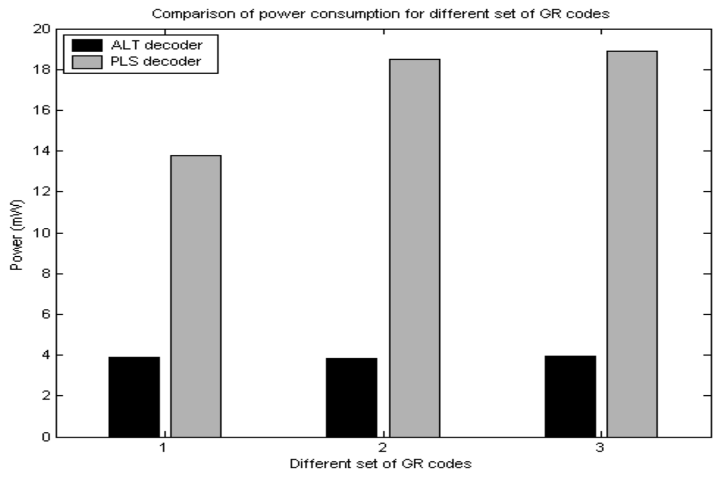


Figure 4-7 The PLS decoder

For a set of GR codes with maximum codeword length of 16 bits, the decoder consists of two separate planes. Each plane consists of a barrel shifter, a 32-bit 2:1 multiplexer, and a 32-bit output register. The codeword table in this case is loaded with a GR codeword table and so is the code length table. This decoder is capable of decoding one codeword per clock cycle and the design makes the coding process parallel by using an "or plane". However, feeding the codeword length from the look-up tables back to the barrel shifters still limits the decoding throughput. All the possible codewords, codeword lengths and decoded integers need to be implemented in the look-up tables, and two types of barrel shifters are included. These all limit the efficiency of the PLS decoder. According to our synthesis results, look-up tables and barrel shifters utilize as much as a minimum of 67% of the total area of the PLS decoder.

We compare the delay, area and power consumption of the ALT decoder to those of the PLS decoder. Both of the decoder types have been implemented in synthesizable VHDL and their performance has been estimated according to the synthesis results. For each type, three decoders for GR codes have been implemented: without a suffix, with 1-bit suffix and 2-bit suffix. The maximum prefix length is kept constant at 16 bits. The results are shown in Figure 4-8. Both types of decoders are implemented in VHDL and synthesized using Design Compiler from Synopsys. The delay has been obtained from static timing analysis and the figures for power consumption from Synopsys' Power Compiler. A standard cell library in a 0.5 μm CMOS process has been used.





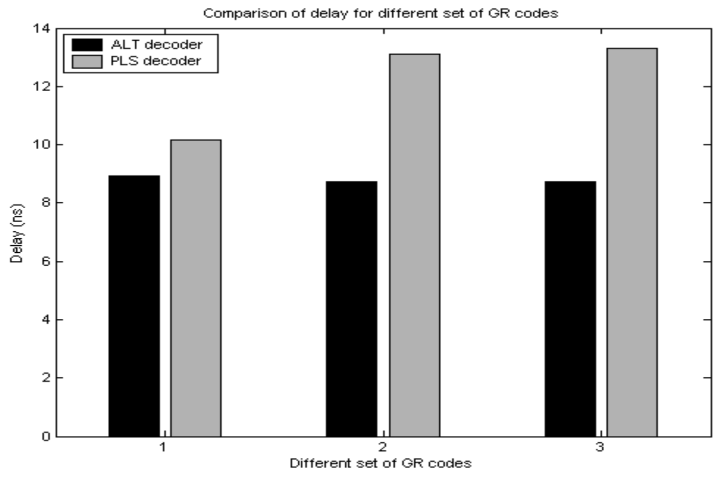


Figure 4-8 Comparison of performances of PLS and ALT decoder

In Figure 4-8, the numbers 1, 2 and 3 on the x-axis represent three different sets of GR codes, 1 stands for GR codes without a suffix, 2 for GR codes with 1-bit suffix, and 3 for GR codes with 2-bit suffix. From these graphs it is obvious that the ALT decoder performs much better than the PLS decoder with regards to area, power and delay. The improvements are dramatic for area and power. For GR codes without a suffix, the ALT decoder receives only 87% of the delay, uses 51% of the area and 28% of the power consumption compared to those of the PLS decoder. For GR codes with 2-bit suffix, the related performances are as good as 65% of the delay, 25% of the area and 20% of the power consumption compared to that of the PLS decoder. Moreover, the performances are constant for different sets of GR codes, whereas the performance of the PLS decoder degrades quite rapidly as the suffix length grows. When the maximum codeword length increases from 16 bits to more than 16 bits yet less than 32 bits, the barrel shifters in the PLS decoder require 5 bits instead of 4 bits to count the number of bits needing to be shifted. Therefore, when 1-bit suffix is added to the prefix that has the maximum prefix length of 16 bits, there are abrupt increases in delay, power and area in the PLS decoder, and this makes the ALT decoder comparatively better.

4.3.2 An ALT decoder for EG codes

For this design, we designed a decoder for the UVLC, which as mentioned in chapter two, is a reversible version of the EG code with *k*=0. For the UVLC, the prefix of the EG code becomes the Odd Indexed Bits (OIB), and the suffix of the EG which for *k*=0 is one bit shorter than the prefix, becomes the Even Indexed Bits (EIB). The design of the entire decoder is based on the architecture discussed in the last section for the UPCs with variable lengths that are linearly related to the length of prefix. The complete design could be described by Figure 4-9 below. In the design, the maximum UVLC length is 31 bit.

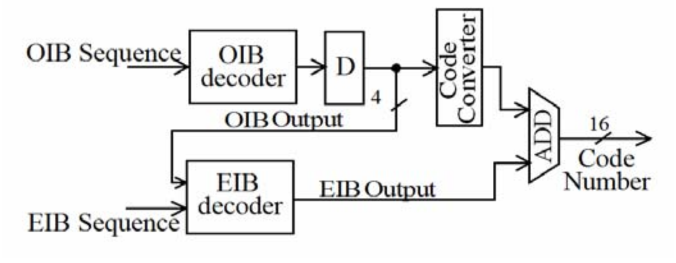


Figure 4-9 ALT decoder for UVLC

The OIB decoder functions exactly like the example in Figure 4-4 and the EIB decoder functions exactly like the example in Figure 4-5. The EIB output is equivalent to the suffix output and the OIB output is equivalent to the prefix output. The code converter applies the relationship in Table 4-4 in order to decode the encoded integer. Table 4-7 shows the details of the truth table of the code converter. The output of the code converter is added to the output of the decoded EIB using the 16-bit adder (ADD) to generate the actual code number.

Input (4 bit)	Output (16 bit)	Output in decimal
0000	0000000000000000	2^0
0001	0000000000000001	2^1
0010	0000000000000011	2^2
0011	0000000000000111	2^3
0100	0000000000001111	2^4
0101	0000000000011111	2^5
0110	0000000000111111	2^6
0111	0000000001111111	2^7
1000	0000000011111111	2^8
1001	0000000111111111	2^9
1010	0000001111111111	2^{10}
1011	0000011111111111	2^{11}
1100	0000111111111111	2^{12}
1101	0001111111111111	2^{13}
1110	0011111111111111	2^{14}
1111	0111111111111111	2^{15}

Table 4-7 Truth table of the code converter

This ALT decoder is also compared to the PLS decoder. However now it requires reconfiguration to a UVLC whose maximum codeword length is 31 bits. The reconfigured PLS decoder is shown in Figure 4-10.

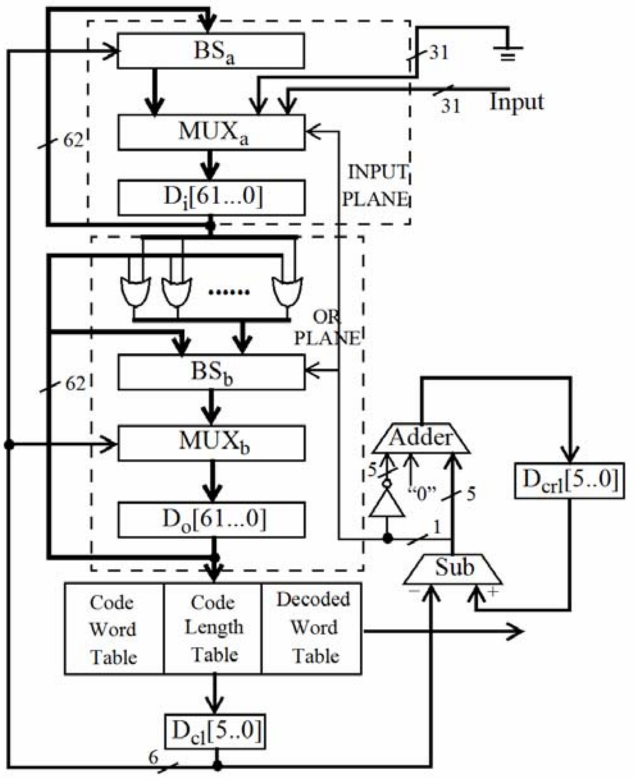


Figure 4-10 The reconfigured PLS decoder

The decoder consists of two separate planes. For UVLC, each plane consists of a 62-bit barrel shifter, a 62-bit 2:1 multiplexer, and a 62-bit output register. The codeword table in this case is loaded with a UVLC codeword table and so is the code length table. This decoder is capable of decoding one codeword per clock cycle and the design makes the coding process parallel by using an "or-plane". All the possible codewords, codeword lengths and decoded code numbers are implemented in the look-up tables, and two types of barrel shifters. These all limit the efficiency of the PLS decoder. According to our synthesis results, look-up tables and barrel shifters use as much as 75% of the total area of the PLS decoder.

We compare the delay, area and power consumption of the ALT decoder to those of the PLS decoder. Both types of decoders are implemented in VHDL and synthesized using Design Compiler from Synopsys. The delay has been obtained from static timing analysis and the figures for power consumption from Synopsys' Power Compiler. A standard cell library in a 0.5 μm CMOS process has been used. The results are shown in Table 4-8.

	ALT	PLS	Ratio (ALT/PLS)
Delay (ns)	8.96	12.0	75%
Area (gates)	1855	3146	59%
Power (mW)	6.74	15.0	45%

Table 4-8 Comparison of performances

It can be seen that the ALT decoder outperforms the PLS decoder for all factors: speed, area and power. The reduction of size and power consumption of the ALT decoder is due to the elimination of huge codeword tables and code length tables and the reduction of the size of the shifting scheme in the conventional VLC decoders. This is also part of the reason why the ALT decoder increases in speed. Another factor for the speed increase of the ALT decoder is because the coding procedure is parellelized by separating the decoding of OIBs and EIBs.

4.3.3 Parallel ALT decoder for GR codes

We have seen from the previous sections that for the GR codes, the LUTs and the shifting scheme could be completely eliminated. The GR decoder involves simply a prefix sub-decoder and a buffer. This structure could easily be expanded to a high-level of parallelization by decoding. If we treat the ALT GR decoder as a one functional unit, many of these units may result in an extension of the decoder structure that can help to parallelize the decoding. In this parallel ALT decoder, the design enables multi-codewords per clock cycle decoding.

Here we target a decoder architecture with constant input rate and variable output rate. It decodes all codewords in an arbitrarily large input buffer in parallel. The output of the symbols is delivered serially at a variable rate and can be fed to an external FIFO buffer. The overall architecture is shown in Figure 4-11.

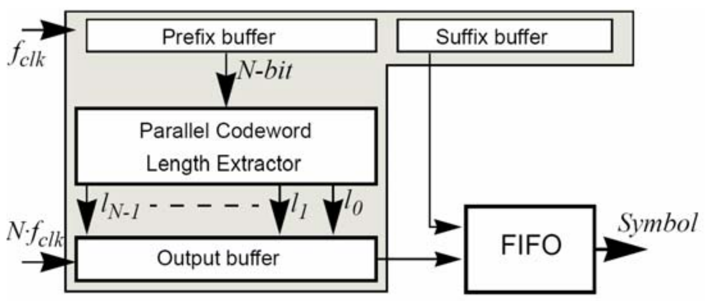


Figure 4-11 Overall decoder architecture

Even though it can decode an arbitrarily large input buffer with constant delay, the throughput has an upper limit which is defined by the maximum speed that the output buffer can deliver a serial sequence of symbols. The decoder is implemented in RT-level VHDL code and from the synthesis results, the maximum throughputs are more than 300M Symbols/s and 800M Symbols/s for FPGA and ASIC implementations respectively.

For ALT-coded GR-codes the decoding is reduced to length extraction of the prefixes. For each clock cycle of f_{clk} the decoder takes in a new set of codewords of *N* bits to the Prefix Buffer. The lengths of the prefixes are extracted in parallel by the Parallel Codeword Length Extractor. At most, when all the codewords are of a minimum length of one bit, *N* codewords are decoded. The output buffer is therefore designed to receive *N* codeword lengths (l_{N-1} to l_0). For normal image data the codeword lengths are distributed within the range of one to a maximum prefix codeword length of *M*. This means that normally, not all positions in the output buffer will be occupied. An empty-indicator, called e_i, is generated and shows whether a buffer position is empty or occupied. The decoded codeword lengths are serially placed in the outputs of the Output Buffer which is a Parallel-Input Serial Output (PISO) register. The maximum output rate is when the prefix codeword lengths are all of one bit which makes it necessary to clock the Output Buffer at $N \cdot f_{clk}$. The empty-indicator from the Output Buffer is used to indicate the existence of data taken out from the Output Buffer. The Suffix Buffer is a PISO where the shifting of *k* steps takes place when e_i is true.

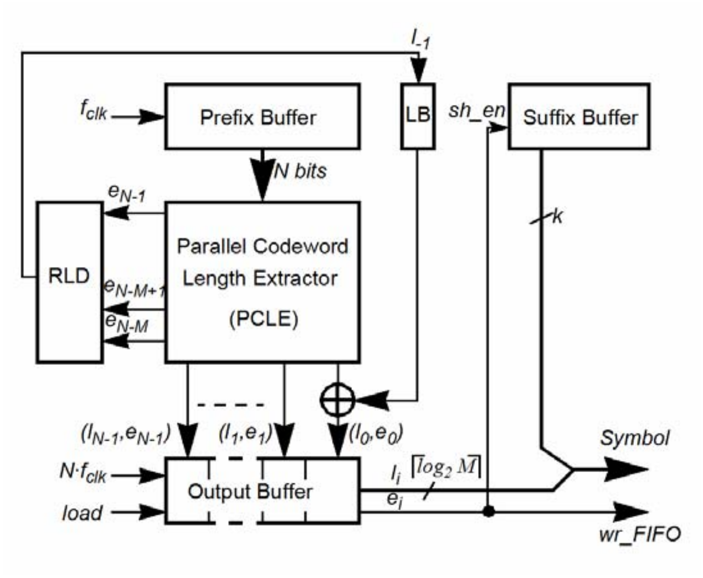
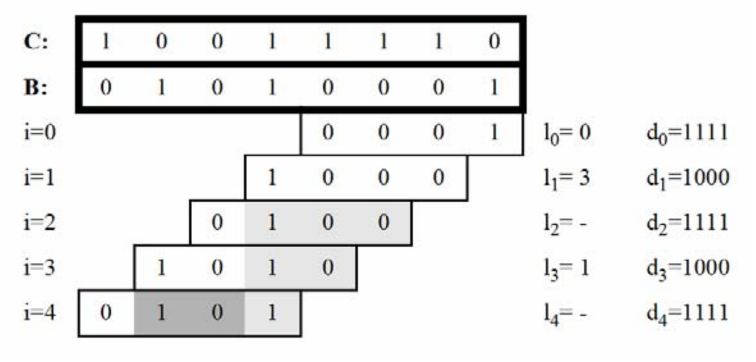


Figure 4-12 Detailed decoder architecture

Under the condition that the codeword length extraction can be parallelized, the critical timing path in this architecture is in the Output Buffer: $t_{critical} = t_{mux} + t_{DFF}$, where t_{mux} is the delay in a 2-1 multiplexer and t_{DFF} is the delay in a D-flip/flop.

Before detailing how the proposed Parallel Codeword Length Extractor (PCLE) is designed, an example is presented showing the working principle in Table 4-9. The input buffer contains the alt-coded prefixes, of maximum length four bits ($M = 4$), in the vector *C*. The rightmost bit in the buffer is considered to be the first bit. From *C* the boundary vector *B* is computed where a '1' indicates the position of the last bit in a prefix code. The length extraction is segmented to windows of *M* bits. Based on the first four bits ($i = 0$) in the *B* vector, the first occurrence of a boundary, i.e. a '1' at position 0 gives us the length $l_0 = 0$. It is guaranteed that the shortest prefix, which is one bit long, is extracted from this window. The next window can therefore be positioned one bit to the left of the previous window. In general, there will be *N M*-bit windows for a prefix buffer of *N* bits. In the next window ($i = 1$) a boundary is found at position 3 ($l_1 = 3$). This boundary is also found in the windows $i = 2, 3,$ and 4. These types of boundaries must be disabled by providing an offset for the above decoder architecture, the key part of which is the "Parallel Codeword Length Extractor" (PCLE) which is used to extract the GR prefixes in a parallel manner.



C:	1	0	0	1	1	1	1	0		
B:	0	1	0	1	0	0	0	1		
i=0					0	0	0	1	l_0= 0	d_0=1111
i=1				1	0	0	0		l_1= 3	d_1=1000
i=2			0	1	0	0			l_2= -	d_2=1111
i=3		1	0	1	0				l_3= 1	d_3=1000
i=4	0	1	0	1					l_4= -	d_4=1111

Table 4-9 Example of parallel length extraction

In order to achieve parallel length extraction, each window has a Length Extraction (LE) unit. It contains three functions:

1) A length extraction function providing the prefix length (l_i);
2) Computation of the disable mask (d_i) that is fed to the following LE-units;
3) Computation of the empty-indicator e_i.

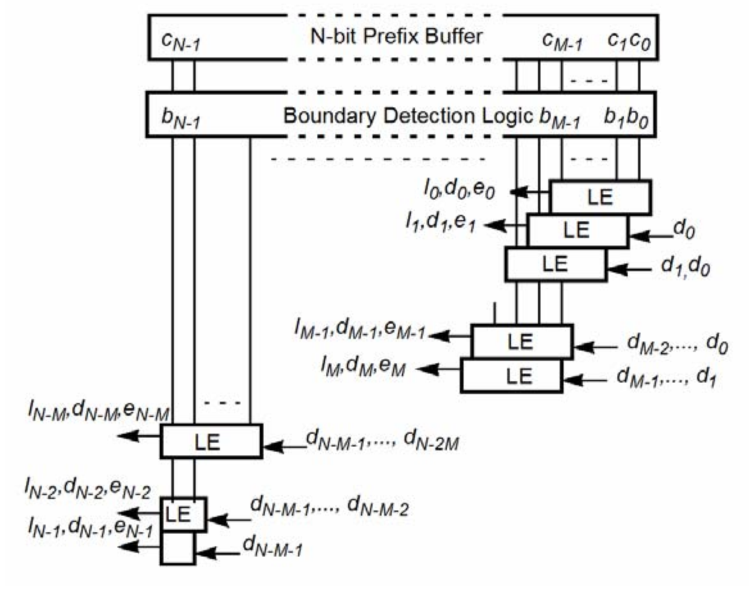
The offset is computed exclusively on the basis on *B*. The length is based on *B* and the offsets from the *M-1* previous LE-units. The block diagram of the PLCE is shown in Figure 4-13. The delay in a parallel architecture cannot be dependent on the size of the prefix input buffer (*N*). For the proposed architecture the delay is dependent on the maximum codeword length (*M*) and not on *N* which allows unlimited parallelization.

In the LE-unit the offset D_i , based on the disable masks from the *M-1* previous LE-units, is computed as:

$$D_i = \bigwedge_{l-M+1}^{i-1} shr_j(d_j) \tag{4.4}$$

where the functions $shr_j(d_j)$ shifts d_j positions to the right with '1' shifted in from the left. When implemented, this is done by wiring. The prefix codeword length is computed as: $l_i = length(D_i \wedge C_i) - length(D_i)$.

In this case C_i is the prefix code for the *i*-th window and the length function is returning the position of the first occurrence of a '1' from the right in the vector.



c_{N-1}
N-bit Prefix Buffer
c_{M-1} $c_1 c_0$
b_{N-1}
Boundary Detection Logic b_{M-1} $b_1 b_0$
l_0, d_0, e_0
LE
l_1, d_1, e_1
LE
d_0
LE
d_1, d_0
$l_{M-1}, d_{M-1}, e_{M-1}$
LE
$d_{M-2}, \ldots, d_0$
l_M, d_M, e_M
LE
$d_{M-1}, \ldots, d_1$
$l_{N-M}, d_{N-M}, e_{N-M}$
LE
$d_{N-M-1}, \ldots, d_{N-2M}$
$l_{N-2}, d_{N-2}, e_{N-2}$
LE
$d_{N-M-1}, \ldots, d_{N-M-2}$
$l_{N-1}, d_{N-1}, e_{N-1}$
d_{N-M-1}

Figure 4-13 Parallel codeword length extraction

The empty-indicator is computed as: $e_i = \neg(D_i \wedge C_i)$

The Critical timing path comes from computing the length l_i and is implemented as shown in Figure 4-14.

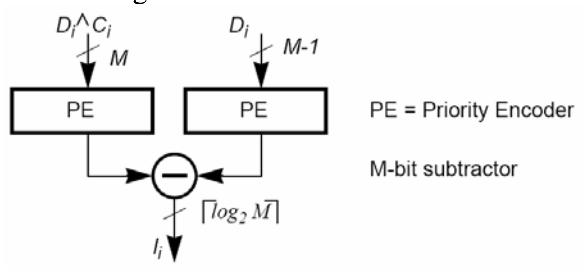


Figure 4-14 Codeword length detection unit

It is possible that only the first part of the last prefix code resides in the Prefix Buffer with the rest being loaded during the next clock cycle. The function Remaining Length Detector (RLD), shown in Figure 4-11, decodes the length of the partial code from the *M-1* empty-indicators and it is stored in the Length Buffer (LB) to be used for the next set of data loaded in the Prefix Buffer.

The alternating coding enables simple logic for length extraction. This is important, even though the architecture can be parallelized without any limitations; it will affect the required clock frequency of the output buffer and the area-cost for the ASIC or FPGA implementation.

The computational logic for the PLCE and the RLD are implemented in the RT-level VHDL. Logic synthesis using Synopsys' Design Compiler for the ASIC implementation in a 0.5 μm CMOS technology and WebPack for the FPGA implementation in Xilinx's Spartan IIe device. The delays have been obtained from pre-layout timing analysis with wire-load models from the silicon vendors.

When designing a decoder with maximum throughput, the minimum size of the Prefix Buffer is determined by the maximum clock frequency of the Output Buffer. This will determine the number of LE-units that is equal to the number of bits in the Prefix Buffer. The number of LE-units required for maximum decoding throughput is:

$$N_{LE} = \left\lceil \frac{t_{LE} + t_{RLD}}{1/f_{outbuff}} \right\rceil = N$$

where t_{LE} is the delay of one LE-unit and t_{RLD} is the delay of the RLD. The delays t_{LE} and t_{RLD} are dependent on the maximum prefix codeword length *M*. Decoders for maximum codeword lengths of 4, 8, 12, 16, 24 and 32 have been designed and evaluated. The suffix length (*k*) does not affect the computational logic and different values of *k* are therefore not investigated. In Figure 4-15 the number of parallel LE-units required for maximum throughput is shown. Maximum throughput for the ASIC and FPGA implementations are 810 MSymbols/s and 340 MSymbols/s respectively. For large values of *M*, the FPGA requires more LE-units compared to the ASIC implementation to reach a maximum throughput. The main reason for this is the larger increase in wire delays for the FPGA.

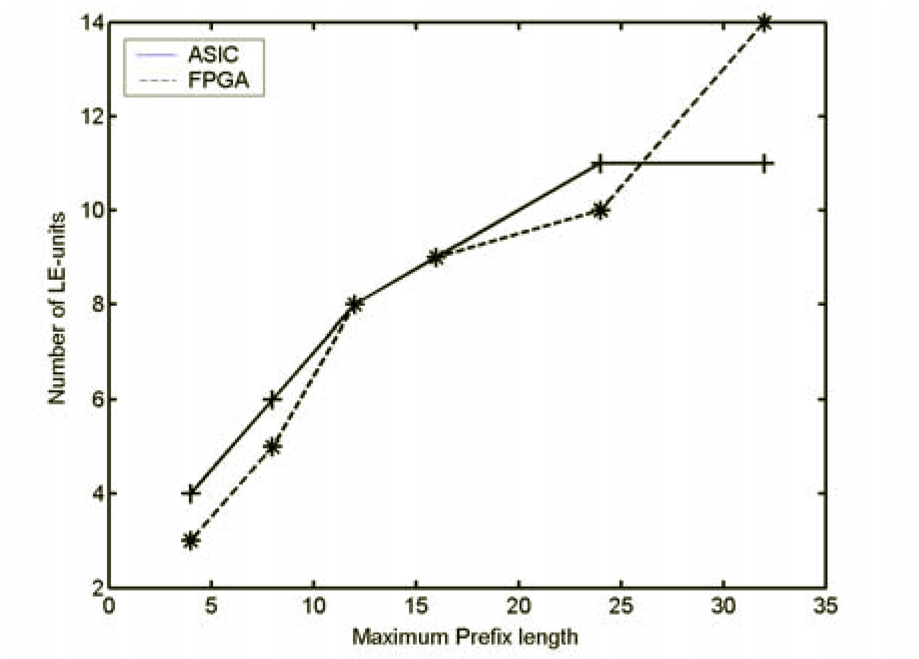


Figure 4-15 Number of parallel LEs for maximum throughput

The areas required for both ASIC and FPGA implementations of the decoder, for different values of *M*, are shown in Figure 4-16. The area grows linearly for the ASIC implementation. However, for the FPGA implementation the area increases rapidly for $M = 32$ because the delay of the LE-unit is large and must be compensated for by increasing the parallelism that requires more LE-units.

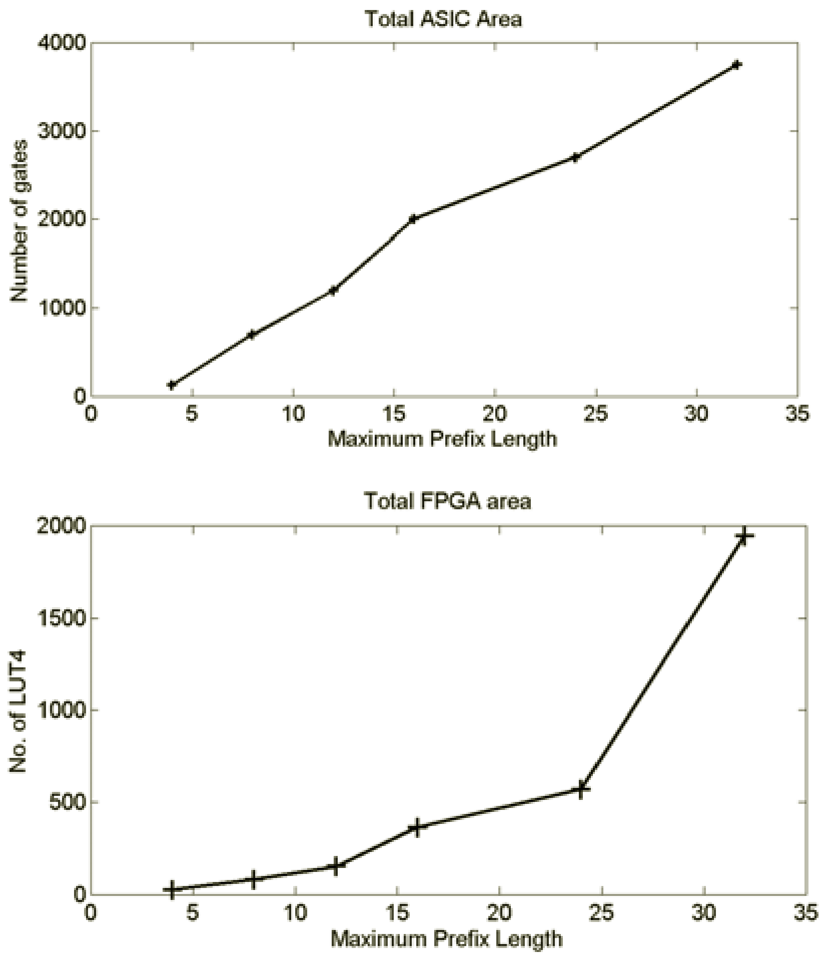


Figure 4-16 Area for computational logic

4.4 THE PROS AND CONS OF THE ALT DECODER

The ALT decoders are built on the basis of the ALT coding method and as shown in the previous sections, the ALT coding is able to extract the structured pattern in the UPC codes and thus leave greater freedom to simplify the decoder architecture to boost its performance. However, these improvements are offset by a cost of loss of generality and flexibility. It has both advantages and drawbacks.

The advantages of the ALT decoders have been discussed throughout this chapter and are now summarized. By separating the decoder into two sub-decoders – the prefix sub-decoder and the suffix sub-decoder, it is possible to eliminate both the LUTs and the shifting scheme (usually barrel shifters) in the prefix sub-decoders; the LUTs in the suffix sub-decoder are also able to be eliminated or reduced. The LUTs and the shifting unit are actually the parts causing the greatest limitations on the performance. For the highly structured UPCs where the suffixes are of fixed lengths and are independent of the prefixes, immediate multi-codeword decoding could be a possibility because of the ALT coded prefixes.

The ALT decoders are designed for the ALT coded UPC codes. All the performance improvements in the ALT decoder are essentially the result of the separation of the unary prefix from the suffixes. However, from chapters two and three, it has been shown that not all VLCs are able to be efficiently converted to UPCs. Thus the ALT decoder is not necessarily a good choice for those VLCs. Moreover, although the advantages of the ALT decoder have been discussed throughout this chapter, the ALT coding does require the encoding to be done in a particular way, so the performance improvement of the decoder has a premise that the UPCs are coded by the ALT coding method and sent to the decoder as the ALT packet. This actually requires a more complex encoder because, in the ALT, decoding requires a separation of prefix and suffix sub-sequences. This means extra buffering and shifting. However, on the other hand, encoding does not curtail the process because we always have a priori knowledge of the codes and the codeword packet whilst encoding them.

In general, it is possible to conclude that for the image/video data which is able to be efficiently encoded by UPCs and ALT coding method, the ALT decoder serves as high-efficiency tool to perform the decoding of the ALT packet.

5 THESIS SUMMARY

In this thesis, the efficient entropy coding of some high-peaked, heavy-tailed source distributions which are often used in the modeling of image and video signals have been studied and a different coding method, the ALT coding, has been proposed, and a different decoder architecture which can be built on the basis of ALT. In this chapter, we briefly summarize the main contribution of the thesis work.

5.1 UPCS

Image video data are often modeled using high-peaked, heavy-tailed probability distributions such as Laplacian, Cauchy and the GG family. Sources with such distributions have an infinite alphabet and therefore the well-known optimal coding algorithm Huffman coding algorithm cannot be applied. These infinite sources all have exponential “decay” rates. Such a property makes the UPCs suitable for use in coding these sources. It has been proved in this thesis that, from all the different UPCs, the UPH codes are able to provide the highest coding efficiency. The coding efficiency of UPH is lower bounded by entropy + 2. In applications, the UPH codes could actually provide coding efficiency much higher than entropy + 2. However, the construction of UPH codes involves serious computation and therefore the UPH codes are not as convenient in applications as the other highly structured UPCs such as the GR, EG, HG codes.

5.2 ALT CODING

The ALT coding method is designed on the basis of the concept of the UPC. The UPCs are all in the form of unary prefixes concatenated with some sort of suffixes which could easily be resorted into a unary prefix sub-sequence and a suffix sub-sequence. The code length information conveyed by the unary prefixes could then be utilized in the decoding of the UPCs and thanks to the simple structure of the unary codes, error resiliency could be improved by two-way decoding and the ES mechanism.

Simulations testing the effect of the ALT decoding as well as practical applications of the ALT coding using UPCs both show that it is possible to significantly improve the error resiliency.

5.3 ALT DECODERS

One of the appealing advantages of the ALT coding is that the unary sub-sequence in the ALT packet could be easily decoded and as this contains the code length information it could greatly reduce the complexity of the entire decoding procedure. Such an advantage is especially beneficial when it comes to the hardware decoder architectures. The unary prefixes partially breaks the sequential dependencies of the variable length UPCs and therefore greatly simplifies the decoding of the entire ALT coded UPC packet. For different types of UPCs, different ALT decoder architectures have been proposed. By comparing these ALT decoders to the general VLC decoders, we have shown that the ALT decoders are fast, small and energy saving. This is especially true for the highly structured UPCs such as the GR and EG codes. These codes are very widely used in practice, and are not merely restricted in the image and video coding system.

5.4 FUTURE WORK

The lower bound of the coding efficiency of the UPH codes has been found to be entropy + 2 in this thesis. We have discussed in the thesis that this lower bound is comparatively weak, especially in the coding of the high-peaked, heavy-tailed probability distributions. Stronger bound should be able to be found by further careful study of the coding procedures.

The ALT coding method has proved itself able to provide better error resiliency, particularly for the highly structured UPCs with fixed length suffixes such as the GR and EG. This is because these codes, with fixed length suffixes, are easily able to be converted into two-decodable codes and bi-directional decoding could be applied with ease. For those UPCs with variable length suffixes, the two-way decoding is difficult because to make them decodable from both directions it is necessary to convert the suffixes into reversible codes, which may result in a lower coding efficiency. By further extending these UPCs into reversible codes and studying the error resiliency obtained by making them two-way decodable should also prove to be interesting and useful.

Much work has been done in this thesis in designing and comparing different ALT decoders with a general VLC decoder. These decoders have been isolated from the peripheries of the entire coding system in order to study their performances clearly and fairly. However, putting them back in to the big coding system and studying its compactness as well as compatibility on a grander scale is also a possible extension of the work in the thesis.

6 REFERENCES

[1] D. Huffman, "A method for the construction of minimum redundancy codes," *Proc. Inst. Radio Eng.,* September, 1952, vol. 40, pp. 1098-1101.

[2] S.W. Golomb, "Run-length encodings," *IEEE Transactions on Information Theory*, vol. 12, pp. 399-401, July 1966

[3] Raymond W. Yeung, *A First Course in Information Theory*, Kluwer Academic/Plenum Publishers, 2002

[4] R. G. Gallager and D.C. Van Voorhis, "Optimal source codes for geometrically distributed integer alphabets," *IEEE Trans Inform. Theory*, vol. 21, pp. 228-230, March 1975

[5] R. F. Rice, "Some practical universal noiseless coding techniques," Tech. Rep. JPL-79-22, Jet Propulsion Laboratory, Pasadena, CA, March 1979

[6] M. J. Weinberger, G. Seroussi, and G. Sapiro, "LOCL-I: A low complexity, context-based lossless image compression algorithm," *Proc. Of the 1996 IEEE Data Compression Conference*, Snowbird, UT, pp. 140-149, April, 1996.

[7] J. Teuhola, "A Compression Method for Clustered Bit-Vectors," *Information Processing Letters*, vol. 7, pp. 308-311, October 1978

[8] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. On Info. Theory*, vol. 21, pp. 194-203, March 1975

[9] S. Xue and B. Oelmann, "Hybrid Golomb codes for a group of quantized GG sources," *IEE Proc.-Vis. Image Signal Process*, vol. 150, pp. 256-260, August, 2003

[10] R. Laroia and N. Farvardin, "A structured fixed-rate vector quantizer derived from length scalar quantizer – Part I: Memoryless sources," *IEEE Trans. On Information Theory*, vol. 39, pp. 851-867, May, 1993

[11] G. Calvagno, C. Ghirardi, G. A. Mian, and R. Rinaldo, "Modeling of subband image data for buffer contral," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 7, pp. 402-408, April, 1997

[12] J. Wen and J. D. Villasenor, "Structured prefix codes for quantized low-shape-parameter generalized guassian sources," *IEEE Trans. on Information Theory*, vol. 45, pp. 1307-1314, May, 1999

[13] Scott M. LoPresto, Kannan. Ramchandran, and Michael. T. Orchard, "Image Coding Based on Mixture Modeling of Wavelet Coefficients and a Fast Estimation-Quantization Framework," *1977 IEEE Data Compression Conference*, Snowbird, UT, pp. 221-230, March 1997

[14] K. A. Birney and T. R. Fischer, "On the modeling of DCT and subband image data for compression," *IEEE Trans. Image Processing*, vol. 4, pp. 186-193, Feb, 1995

[15] A. Kiely and M. Klimesh, “Generalized golomb codes and adaptive coding of wavelet-transformed image subbands,” IPN PR 42-154, pp. 1-14, August 15, 2003

[16] Y. Takishima, M. Wada, H. Murakami, “Reversible Variable Length Codes,” *IEEE Trans. Comm.*, vol. 43, No.2/3/4, pp. 158-162, 1995

[17] ISO/IEC JTC1/SC29/WG11 N1383, “Description of Error Resilient Core Experiments,” Nov. 1996

[18] J. Wen and J. D. Villasenor, “A Class of Reversible Variable Length Codes for Robust Image and Video Coding,” *Proc. Int. Conf. Image Processing*, vol. 2, pp. 65-68, Oct. 1997

[19] M. Rahman and S. Misbahuddin, “Effects of a binary symmetric channel on the synchronization recovery of variable length code,” *Computer J.*, vol. 32, pp. 246-251, 1989

[20] Y. Itoh, “Bi-directional motion vector coding using universal VLC,” *Signal Processing: Image Communication*, vol. 14, pp. 541-557, May 1999

[21] Y. Itoh, Ngai-Man Cheung, “Universal variable length code for DCT coding,” *International Conference on Image Processing*, vol. 1, pp. 940-943, 2000

[22] A. Mukherjee, N. Ranganathan, and M. Bassiouni, “Efficient VLSI design for data transformation of tree-based codes,” *IEEE Trans. Circuits Syst.*, vol. 38, pp. 306–314, Mar. 1991

[23] A. Mukherjee, N. Ranganathan, J. W. Flieder, and T. Acharya, “MARVLE: A VLSI chip for data compression using tree-based codes,” *IEEE Trans. VLSI Syst.*, vol. 1, pp. 203–214, June 1993

[24] S. M. Lei and M. T. Sun, “An entropy coding system for digital HDTV applications,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 2, pp. 147–154, Mar. 1991

[25] S. F. Chang and D. G. Messerschmitt, “Designing high-throughput VLC decoder Part I—Concurrent VLSI architecture,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 2, pp. 187–196, June 1992

[26] P. Pirsch, *VLSI Implementations for Image Communication.* Amsterdam, The Netherlands: Elsevier, pp. 345–364, 1993

[27] M. T. Sun and S. M. Lei, “A high-speed entropy decoder for HDTV,” *Proc. IEEE 1992 Custom Integrated Circuits Conf.*, pp. 26.3.1–26.3.4, 1992

[28] Jae Ho Jeon et al, ”A fast variable-length decoder using plane separation,” *IEEE. Trans. Circuits Syst. Video Technol.*, vol. 10, pp. 806-812, Aug. 2000.

[29] S. B. Choi and M. H. Lee, “High speed pattern matching for a fast Huffman decoder,” *IEEE Trans. Consumer Electron.*, vol. 41, pp. 97–103, Feb. 1995

[30] S. F. Chang and D. G. Messerschmitt, “Designing high-throughput VLC decoder Part I—Concurrent VLSI architectures,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 2, pp. 187–196, June 1992

[31] R. Hashemian, “Design and hardware construction of a high speed and memory efficient Huffman decoding,” *IEEE Int. Conf. Consumer Electron.*, pp. 74–75, 1994

[32] S. H. Cho, T. Xanthopoulos and A. P. Chandrakasan, “A Low Power Variable Length Decoder for MPEG-2 Based on Nonuniform Fine-Grain Table Partitioning,” *IEEE Trans. VLSI Systems*, vol. 7, no. 2, pp. 249-257, June 1999

[33] H. D. Lin and D. G. Messerschmitt, “Designing high-throughput VLC decoder Part II—Parallel decoding methods,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 2, pp. 197–206, June 1992

[34] W. K. Pratt, *Digital Image Processing*, New York: Wiley-Interscience, chap. 10, 1978

[35] A. N. Netravali and J. O. Limb, “Picture coding: A review,” *Proc. IEEE*, vol. 68, pp. 7-12, Mar 1960

[36] R. C. Reininger and J. D. Gibson, “Distributions of the twodimensional DCT coefficients for images,” *IEEE Trans. on Commun.*, vol. COM-31, pp 835-839, June 1983.

[37] S. R. Smooth and R. A. Lowe, “Study of DCT coefficients distributions,” *Proc. SPIE*, pp. 403-311, Jan. 1996.

[38] E. Y. Lam and J. W. Goodman, “A mathematical analysis of the DCT coefficient distributions for images,” *IEEE Trans. on Image Proc.*, vol. 9, no. 10, pp. 1661-1666, Oct. 2000,

[39] T. Eude, R. Grisel, H. Cherifi, and R. Debrie, “On the distribution of the DCT coefficients,” *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing*, vol. 5, pp. 365-368, Apr. 1994

[40] Y. Altunbasak and N. Kamaci, “An analysis of the DCT coefficient distribution with the H.264 video coder,” *IEEE Int. Conf. on Acoustics Speech and Signal Processing*, Montreal, Canada, May 2004

[41] S. F. Chang and D. G. Messerschmitt, “Designing a high-throughput VLC decoder. I. Concurrent VLSI architectures,” *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 2, Issue 2, pp. 187-196, June 1992

[42] H. D. Lin and D. G. Messerschmitt, “Designing a high-throughput VLC decoder. II. Parallel decoding methods,” *IEEE Trans on Circuits and Systems for Video Technology*, vol. 2, Issue 2, pp. 197-206, June 1992

[43] Iain E. G. Richardson, *Video Codec Design – Developing Image and video Compression Systems*, John Wiley & Sons Ltd., 2002

[44] ISO/IEC 14496-10 and ITU-T Rec. H.264, Advanced Video Coding, 2003

[45] T. Wiegand, G. Sullivan, G.Bjontegaard and A. Luthra, “Overview of the H.264/AVC Video Coding Standard,” *IEEE Trans. on Circuits and Systems for Video Tech.*, vol. 13, Issue 7, pp. 560-576, July 2003

[46] D. Marpe, G. Blattermann and T. Wiegand, Adaptive codes fro H.26L, ITU-T SG16/6 document VCEG-L13, Eibsee, Germany, Janurary 2001

[47] H. Schwarz, D. Marpe and T. Wiegand, CABAC and slices, JVT document JVT-D020, Klagenfurt, Austria, July 2002

[48] D. Marpe, H. Schwarz and T. Wiegand, “Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard,” *IEEE Trans. On Circuits and Systems for Video Tech.*, vol. 13, Issue 7, pp. 620-636, July 2003

[49] Iain E. G. Richardson, H.264 and MPEG-4 Video Compression – Video Coding for Next-generation Multimedia, John Wiley & Sons Ltd., 2003

[50] G. Bjontegaard and K. Lillevold, Context-adaptive VLC coding of coefficients, JVT document JVT-C028, Fairfax, May 2002

[51] S. R. Smoot and L. A. Rowe, "Study of DCT coefficient distributions," *SPIE Symposium on Electronic Imaging*, vol. 2657, San Jose, USA, January 1996

[52] A. N. Netravali and B. G. Haskell, Digital Pictures: Representation and Compression. Applications of Communications Theory (Series Editor R. W. Lucky), Plenum Press, NY, NY, 1988

[53] R. C. Reininger and J. D. Gibson, "Distrubutions of the two-dimensional DCT coefficients for images," *IEEE Transactions on Communications*, vol.31, Issue 6, pp.835-839, June 1983

[54] T. Chiang and Y. Q. Zhang, "A New Rate Control Scheme Using Quadratic Distortion Mode," IEEE Trans. CSVT, vol.7, February 1997

[55] N. Jayant and P. Noll, "Digital coding of Waveforms," Englewood Cliffs, NJ: Prentice Hall, 1984

[56] Q. Wang, Z. Xiong, F. Wu and S. Li, "Optimal Rate Allocation for Progressive Fine Granularity Scalable Video Coding," *IEEE Signal Processing Letter*, vol. 9, February 2002

[57] M. Dai, D. Loguinov, and H. Radha, "Statistical Analysis and Distortion Modeling of MPEG-4 FGS," *IEEE International Conference on Image Processing (ICIP),* September 2003

[58] "Video Codec Selection for Wireless Multimedia Terminals", Hntro Products, [online]. Available: http://www.hantro.com/pdf/codec.pdf

[59] M. Takahashi et al., "A 60 mW MPEG4 video codec using clustered voltage scaling with variable supply-voltage scheme," *IEEE Journal of Solid-State Circuits*, vol. 33, No.11, November, 1998

[60] T. Nishikawa et al., "A 60 MHz 240 mW MPEG-4 video-phone LSI with 16 Mb embedded DRAM," *IEEE Journal of Solid-State Circuits*, vol. 35, No.11, November, 2000

[61] T. Hashimoto et al., "A 90 mW MPEG4 video codec LSI with the capability for core profile," in *Digest of Technical Papers of IEEE International Solid-State Circuits Conference*, 2001

[62] M. Ohashi et al., "A 27 MHz 11.1 mW MPEG-4 video decoder LSI for mobile application," *IEEE Journal of Solid-State Circuits*, vol. 37, No.11, November, 2002

[63] M. Nakayama et al., "An MPEG-4 video LSI with an error-resilient codec core based on a fast motion estimation algorithm," in *Digest of Technical Papers of IEEE International Solid-State Circuits Conference*, 2002

[64] H. Arakida et al., "A 160mW, 80nA standby, MPEG-4 audiovisual LSI with 16mb embedded DRAM and a 5GOPS adaptive post filter," in *Digest of Technical Papers of IEEE International Solid-State Circuits Conference*, 2003

[65] H. J. Stolberg et al., “An SoC with Two Multimedia DSPs and a RISC Core for Video Compression and Surveillance,” in *Digest of Technical Papers of IEEE International Solid-State Circuits Conference*, 2004

[66] P. C. Tseng and L.G. Chen, “Hardware Architecture Design for Visual Processing: Present and Future,” *IEEE AP-ASIC2004*, pp. 6-9, August, 2004

[67] “Choosing a Platform Architecture for Cost Effective MPEG-2 Video Playback”, Platform Architecture Labs/Platform Technical Marketing Desktop Products Group, Intel Corporation, April, 1996

[68] S. Sriram and Ching-Yu Hung, “MPEG-2 Video Decoding on the TMS320C6X DSP Architecture”, DSPS R&D Center, Texas Instruments, Inc.

[69] P. Pirsch and H. J. Stolberg, “VLSI architectures for multimedia”, *1998 ICECS*, vol. 1, pp.3-10, September, 1998

[70] B. Furht, “Processor Architectures for Multimedia: A Survey (Invited paper)”, *Multimedia Modeling Conf.*, pp. 89-109, November, 1997

[71] P. Mårtensson, J. Persson, Shang Xue, and Bengt Oelmann, “Efficient Decoding of Variable Length Encoded Image Data on the Nios II Soft-Core Processor”, *In the proceedings of the International Workshop on Applied Reconfigurable Computing*, Algarve, Portugal, February 2005