Lossless video compression in an FPGA for reducing DDR memory bandwidth usage

Fredrik Jacobsen Fagerheim
Stian Røed Hafskjold

Master of Science in Electronics

Submission date:	June 2013
Supervisor:	Kjetil Svarstad, IET
Co-supervisor:	Jørgen Linnerud, Cisco Systems

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

PROBLEM STATEMENT

It would be interesting to investigate if the memory bandwidth usage could be significantly decreased using lossless, or close to lossless, compression algorithms and find out if these are possible to implement in an FPGA and be used for real-time video in telepresence/video-conferencing products.

The master thesis will focus on writing HDL components for the algorithms and implement them in an FPGA with low FPGA resource usage in mind and benchmark them. For the implementation it is important to find best and worst cases scenarios as well as average use of memory bandwidth. The student would have to benchmark these implementations using real-life video from telepresence/video-conferencing scenarios as well as corner-case video patterns to prove the qualities of the different implementations. It is required that the project results in a clear table comparing the different algorithms, or set of algorithms, used for the implementations.

PREFACE

This Master's thesis was completed at the Norwegian University of Science and Technology (NTNU), under the department of Electronics and Telecommunications. This thesis marks the end of a five year long study, to achieve a Master of Science degree in the field of Electronics with Digital System Design as the main profile.

This thesis is a continuation of a preliminary project which was completed during the autumn of 2012. The problem description is given by Cisco Systems Norway, who develops products that are focused on Telepresence technology and services.

We would like to thank our supervisors at NTNU, Professor Kjetil Svarstad, and our external supervisor at Cisco, Jørgen Linnerud, for meeting with us on a regular basis, providing us with helpful feedback and guidance.

ABSTRACT

We show that a hardware implementation of a lossless image compression scheme can be used as means for lowering DDR memory bandwidth usage from a video stream. A prediction scheme based on LOCO-I is used to reduce correlative redundancy between sequential pixels, before the data is encoded by Golomb coding. The data packages after source coding contain a continuous stream of prefix codes, in order to eliminate the header data imposed by more advanced packing schemes. This in turn results in a higher demand on the decoding side in terms of resource usage, because of the need for high parallelism when a new prefix code is counted and decoded each clock cycle.

The test images are reduced in size by 49-84%, depending on their inherent complexity. Resource consumption for this design amounts to 10100 Logic Elements (synthesized for an Altera Cyclon III FPGA - EP3C80F484C6), with a operating frequency of 152,86 MHz for a throughput of 458 MB/s. These numbers can be improved by reducing the algorithm complexities. LE usage is reduced to 4695, while accomplishing a image size reduction of 34-59%.

We present a way to increase decompression throughput by adding parallel decoder modules. Those changes will increase throughput to a multiple of 458 MB/s while worsening the compression somewhat. LE cost increases depending on the level of parallelism.

SAMMENDRAG

Vi viser at en hardware-implementasjon av tapsfri bildekompresjon kan bidra til å redusere bruken av minnebåndbredden fra en videostrøm. En prediksjonsmodell basert på LOCO-I brukes til å redusere korrelativ redundands mellom sekvensielle piksler, før dataen kodes med Golomb-koding. Den komprimerte dataen pakkes tett i en kontinuerlig strøm med prefikskodede ord for å redusere overflødig data som innføres av mer kompliserte pakkemetoder. Den tette pakkingen resulterer i noe høyere ressursbruk i dekodingen, fordi ett kodeord må separeres bit-vis parallellt per klokkeperiode i den kontinuerlige strømmen.

Testbildene reduseres i størrelse med 49-84%, avhengig av deres iboende kompleksitet. Ressursforbruk for dette designet ligger på 10100 Logiske Elementer (Logic Elements) (syntetisert for en Altera Cyclon III FPGA - EP3C80F484C6), med en operasjonsfrekvens på 152,86 MHz for en databehandlingsrate på 458 MB/s. Disse tallene kan forbedres ved å redusere kompleksiteten i algoritmene som benyttes. LE-forbruk kan da reduseres til 4695, med en reduksjon i bildestørrelse på 34-59%.

Vi foreslår å øke databehandlingsraten på dekomprimeringssteget ved å kjøre flere dekodings-moduler i parallell. Disse endringene øker databehandlingsraten til et multiplum av 458 MB/s, mot at kompresjonseffekten reduseres noe. LE-forbruket øker avhengig av antall parallelle moduler.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

ONE

INTRODUCTION

In recent years, the primary channels for communication and distribution of video content has begun to change. Faster broadband speeds allow for services to shift from proprietary and rigorous distribution channels, to the more dynamic and widely available global network that is the Internet. As the entertainment industry pushes the boundaries for what can be achieved in terms of video quality on preloaded storage mediums such as Blu-ray, higher quality has also come to be expected from online services, where the source material is transferred to its destination via a network connection. For this transfer to be possible on a wide range of network connection speeds, the video information must be compressed before leaving the host. If the source is something pre-recorded, like a movie, computationally demanding compression algorithms can be applied in advance in order to have the optimally encoded video ready when some client requests the content. A live recording, on the other hand, must be encoded in real time if it is going to appear live for the person viewing it at the other end. When the time variable is pushed like that, the system performance necessary to achieve a high-quality low-bitrate encoding rises drastically. With a set system performance, the trade-off between size and video quality becomes apparent. The size must be reduced to a so that the data it is easily transferred to its destination, while the video quality should not degrade too far from the original.

Communication via live high definition video is almost changing what is considered to be face to face interactions. The more immersive experience is brought forth by the continuing development of better video capturing devices as well as consumer grade high resolution monitors with high-quality color reproduction. These improvements to the capturing and displaying require more of the technology responsible for compressing and encoding the video output. Higher resolution means that more pixels must be processed per unit of time, while the modern monitors allow us to see exactly where the image lost information in the encoding process. Running better algorithms during encoding can make sure more relevant image information is kept, while reducing the noise and other components that are either considered unimportant or simply unnoticeable to the human eye.

These image processing algorithms demand high throughput of internal data in order to process high definition video streams in real time. The algorithms need to cache large amounts of data in memory to allow for encoding across several video frames. The resource demand and cost introduced if high speed registers are used, imposed by the significant data size, results in the need for higher level memory, such as DDR SDRAM. This comes at the cost of lower internal throughput because of the lower memory bandwidth.

1.1 The thesis and our interpretations

The thesis asks whether lossless or close to lossless compression algorithms can be implemented in an FPGA in order to reduce the memory bandwidth usage on real-time video telepresence/video-conferencing products. This was interpreted to mean that the raw video stream is to be compressed before it is stored in memory, then decompressed when it is processed further. No other details or restrictions are given other except for the real-time requirement, which is interpreted to mean a frame processing rate equal to the defined frame rate at a given resolution.

The thesis text does not specify if or how the solution should co-exist with an existing system. Certain implications of our solution have therefore manifested at rather late stages of the work process. Our solution's usefulness can therefore prove to be non-existent in a system with specific requirements. These are mainly requirements for how the existing system expects to access the compressed information in memory. The report discusses these problems and tries to suggest changes to our design that may improve its usefulness.

The goal of performing tests on an FPGA was put on hold because it would have required considerable more time. Access to a full fledged test system necessitates a travel to Cisco Norway, and would have tied down their resources as well. Instead we prioritized finding alternatives to our primary design that hopefully would broaden our conclusion. Further, we focused on the core of the compression-decompression modules, meaning that system control and general memory packing modules have not been implemented. The operation and functionality of these are considered to be so tightly connected to the adopting system, that they in any case must be built with a specific system in mind. Instead the overall control possibilities are discussed in chapter 7.

1.2 Contribution

This thesis has continued the process that began with the preliminary work presented in the project report [1]. Where the preliminary work focused on measuring the performance of a number of image compression techniques, this work uses the findings to implement some of the algorithms in VHDL. The implementations will then form the core of a system that is able to reduce the data size of an RGB video stream on-the-fly through lossless compression. On-the-fly means that the video content is compressed as it arrives, without touching DDR memory first. This in turn means that the encoder must handle a throughput close to 374 MB/s for a 1080p video stream at 60 frames per second. The lossless compression is meant to lower DDR memory usage, lending more headroom to the lossy video encoding processes. This report does not contain a complete system, ready to be run on an FPGA. Instead it presents a compression module and a decompression module.

A compression scheme was chosen based on the arguments presented later in chapter 2.1. LOCO-I with context modeling and Golomb coding showed the best results in the preliminary work, and also was also considered to be among the most viable for a hardware implementation.

The compression module takes one 8 bit color component per clock cycle, and produces compressed irregularly timed 16 bits of packed data. The decompression module decodes those 16 bits of packed data and produces one 8 bit color component per clock cycle. Additional functionality that is necessary for a system to be able to operate is discussed, but not implemented in VHDL, because of uncertainties surrounding the integrating (existing) system.

Requirements and guidelines

- Real-time operation $((1920 \times 1080)\frac{pixel}{frame} \times 3\frac{Byte}{pixel} \times 60\frac{frame}{second} = 374\ MB/s)$
- System frequency of 148,5 MHz for 1080p resolution
- Aim for using few Logic Elements
- Limit use of caching and block ram

1.3 Outline

Chapter 3 describes a system level model of the design. It outlines the functionality necessary for the compression scheme to operate on a hardware architecture.

Chapter 4 presents the VHDL implementation. Here the hardware-level considerations that went in to designing synthesizable modules are described.

Chapter 5 describes the test methodology and test data used to obtain the results.

Chapter 6 presents the compression results and relevant figures with different system configurations and algorithm complexities.

Chapter 7 discusses the adjustments to algorithms used and reasons for making changes. Results of the VHDL implementation are discussed in terms of throughput and resource consumption, a possible limited implementation is explored. Considerations for a system that is integrating the compression scheme are discussed, as well as limitations and possible solutions that can bypass those limitations. Finally the primary and simplified implementation are compared.

Chapter 8 delivers the conclusion where results and findings discussed in chapter 7 are summarized. Points to continue working on are also listed.

CHAPTER

TWO

BACKGROUND

2.1 Preliminary work

In the preliminary work [1] we presented several compression schemes that can be used for reducing the storage size of an image. Different contenders for the two stages of redundancy reduction and entropy coding were implemented in C++ and tested in order to evaluate their performance in terms of compression. The implementations also served as great tools to better understand the computational steps of the different algorithms. A decision on which scheme to continue working on had to be made by looking at compression statistics and estimating the complexity of a hardware implementation, especially considering the throughput and real time requirements.

Basis for choosing our algorithm

The compression comparisons ([1] Ch. 5.6, fig. 21-22) showed that LOCO-I + Golomb performed the best on images with higher entropy, while arithmetic coding performed the best on images with lower entropy. Both schemes were among the top three for both image types. FELICS placed right behind LOCO-I + Golomb on both image types, while LZSS performed well on low entropy images.

Some of the coding algorithms has characteristics that make them less suitable for hardware implementation than others. Both of the Lempel-Ziv variants try to represent sequences of recurring symbols with dictionary look-up values. The creation of these values require a number of searches, and the number of comparisons necessary in order to parallelize a search is quite large (16.264 - ref [1] Ch. 6.4). The encoding is also causal, making it unfeasible to split searches into multiple stages.

Golomb coding is fairly well suited for hardware implementation, with a more hardware specific variant known as Golomb-Rice. GR uses divisors that are powers of 2, so that divisions can be implemented as right shifting and multiplications as left shifting. The algorithm requires no caching or storage, but it depends on optimal k-values to achieve optimal coding. Decoding is causal in the sense that the codewords are of variable lengths, and one code must be separated from the rest before the next can be found. Calculation of the binary values can be performed over several pipelined steps if necessary to increase speed. An implementation using Golomb coding with Differential-Differential Pulse Code Modulation presented in [2] achieved a high throughput of 2720 MB/s due to high parallelism.

Huffman coding in its simplest form require a pre-calculated code table. Even if the compression is acceptable, it places last among the examined algorithms. Compression can be improved by using an adaptive code table, although this requires an adaptive statistical model of the alphabet for each frame in order to optimize the code table. Decoding involves the same causal variable length code word separation as Golomb. This fact makes Golomb a better candidate because it has the same technical challenge, but does not require caching of a table or statistical modeling during runtime.

Arithmetic coding was never fully implemented in C++, and a Java encoder [13] was used instead, so the algorithm was not as well understood as the other alternatives. It also required a statistical model similar to adaptive Huffman in order to optimize encoding. An hardware architecture based on context-based modeling and arithmetic coding is presented in [5]. It uses 1123 slices for a throughput of 15,37 MB/s.

FELICS is considered to be more complicated than the LOCO-I + Golomb alternative, because it uses a second source coding technique in addition to Golomb. This, paired with the fact that it showed worse compression results means that LOCO-I + Golomb is a better choice. The implementation in [4] achieved a throughput of 546 MB/s.

The existing implementations

Some specifications from papers presenting hardware implementations of similar compression schemes are summarized in table 2.1.

It can be seen that only DDPCM+GR and FELICS cover our throughput requirement of 374 MB/s (3 Gb/s), but both are VLSI-oriented ASICs, so it may be unlikely that they would have achieved anything close to those frequencies on common FPGAs. The high parallelism of DDPCM+GR means it could still be able to reach the required throughput even if its operating frequency takes a hit. It should

Table 2.1: *Performance comparison between encoder implementations*

Algorithm	DDPCM+GR	JPEG-LS	FELICS	Arithmetic Code
Technology	0.15µm	0.18µm	0.13µm	Vertex 4
Operating freq	170 MHz	40 MHz	273 MHz	123 MHz
Throughput	2720 MB/s	9.9 MB/s	546 MB/s	15,37 MB/s
Compression	5.26 $^b/_s$	-	3.40 $^b/_s$	4.55 $^b/_s$
Parallelism	16	1	2	1
Memory	Not used	2.25KB	1.9KB	7.7KB
Reference	[2]	[3]	[4]	[5]

also be noted that the test images used for obtaining the compression rates differ between the papers. Their efficiencies are therefore difficult to gauge, and it is more relevant to note that i.e. [2] has a best case rate of 4.75 $^b/_s$ (bits per 8 bit sample).

Decision

Based on these considerations, the choice fell on implementing the LOCO-I context modeling with adaptive Golomb encoding. It had the highest potential for compression, the prediction and context modelling had high potential for pipelining, while the Golomb scheme was well understood with tunable properties.

The most apparent obstacle before implementation had begun was the issue of decoding tightly packed Golomb code words of variable lengths. One code must be separated from the rest before the next can be separated. The unary coded quotient q must be counted, as its length is unknown at the beginning of a new code word decoding. The length of the residual r is equal to the k-value, and is known at the start of a code word decoding.

It must also be attempted to minimize the amount of overhead data required to locate, recover and decode the frame data from memory. The three color components are expected to be processed by separate parallel encoder instances, and a packing scheme must be in place which keeps track track of the colors in the memory.

2.2 Image representation and compression theory

This section provides the theory needed in order to understand the basis behind compression algorithms and the choices made during the initial work.

Since most of the theory regarding image representation and compression were examined during the preliminary project, some of the sections will be repeated here in this section and its subsections. More specifically, section 3 'Image compression theory' in [1] and some parts of section 2 'Image representation' are repeated.

An uncompressed image is represented by an array of pixels, where each pixel is given a numerical value describing the color and luminance information of that pixel. This value is encoded with a fixed amount of bits, given by the alphabet size. In an RGB24 representation, the luminance of each color component red, green and blue, is encoded with an alphabet size of 256 per pixel, resulting in a total of 8+8+8=24 bits per pixel. The total information stored in a picture, however, does usually not require as many bits, because some of the information is redundant (i.e repeated or irrelevant).

Image compression is the process of identifying this redundant information, and reducing the amount of data needed to represent the total information stored in an image, by removing as much redundant information from each pixel as possible. The objective is to represent the image with fewer bits than the original representation, and the amount of compression obtained is the ratio between the total number of bits before and after compression.

Statistical redundancy in an image can be divided into three main categories, coding, spatial and spectral redundancy, and will be presented in the following sections.

Coding redundancy

In an uncompressed image each pixel is encoded using a fixed size bit representation. In the case of RGB24, as mentioned in the previous section, the pixel is coded with 3 separate color components, red, green and blue, each component consisting of 8 bits, allowing for an alphabet size of 256 characters. This would lead to a fixed image size relative to the number of total pixels in the image. However, the value represented by each pixel is not completely random (unless the image consists entirely of white noise), and thus it would be useful to represent values occurring more often with shorter bit lengths than values that occur rarely. This will in all cases result in a shorter (or equal) total bit length than the original representation.

Values that occur more often than others, i.e. has a higher probability, is said to contain less information. For instance, if the probability of an outcome equals 1, the information content would be zero, because it would be entirely predictable. The relationship between information content and probability of an outcome x_i, can be written as,

$$I(x_i) = \log \frac{1}{P(x_i)} = -\log P(x_i), \tag{2.1}$$

and gives the number of digits needed to represent x_i. Using the base 2 logarithm would give us the result in bits. So by generating a probability distribution of all the pixel values, $P(x)$, we can get a measure on the information content stored in each pixel. More precisely this is called entropy and is defined as

$$H = \sum_i P(x_i)I(x_i) = -\sum_i P(x_i)\log_2 P(x_i) \tag{2.2}$$

and gives the average number of bits per pixel needed to represent all information. Or in other words, the entropy serves as a lower bound on the average number of bits needed. So by multiplying the average number by the total number of pixels, we get a lower bound on the compressibility of the data. Thus, the entropy of a data set provides us with very useful information regarding the efficiency of the chosen data compression scheme, by comparing the size of the compressed data with the entropy.

If we would like to code the image with shorter bit length at pixels that occur often, and vice versa, we would need a coding scheme supporting variable bit length. Coding the image using variable bit length based on the probability of the value to be coded, is called entropy coding. Examples of entropy coding algorithms includes Huffman coding, arithmetic coding and Golomb coding.

Spacial redundancy

Specifically, equation 2.2 is called the first order entropy, and only takes into account the first order data correlation between pixels. That is the correlation between the actual pixel values through out the image frame. The n-th order entropy would also consider higher order correlation, like the difference between two subsequent pixels, which would give us the second order correlation. The data redundancy governing higher order correlation is called *spacial redundancy*.

Spectral redundancy

Color information can be represented in different ways. In an *RGB* representation, the luminance of each color component is described separately. In another representation *R-G,G,B-G*, the luminance of the color green is given a value, while the other colors, blue and red, is represented by how much they differ from green. In realistic images the luminance information between red, green and blue usually correlate to a certain degree. Thus, describing the image according to the *R-G,G,B-G* representation would in most cases result in a lower entropy [6].

Image compression

Lossless image compression schemes are usually broken into several processing steps, where each step perform calculations sample by sample (pixel) in a predefined order. Because of the limitations induced by the requirements, only schemes that scan through the samples line by line are considered (This is called raster scan). They are also limited to one pass only. A template of the current sample to be processed,*x*, with regard to its neighboring samples is illustrated in figure 2.1. This representation as well as the naming of the samples, will be used throughout this paper. Notice that samples *a* through *e* are causal and known to both the encoder and the decoder.

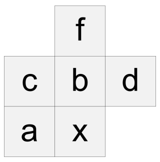


Figure 2.1: *Causal template of current sample 'x'*

The schemes are usually divided into two main components, modeling and source coding. In the modeling part of the scheme, the predicted value of the current sample based on samples *a* through *e* is computed. The source coding part then encodes the error residual of the predicted value. [11]

The actual compression happens during source coding, since this is the stage where the output code is produced. However, source coding of an image without decorrelation through the modeling stage, is usually futile because the uncorrelated data is more or less random to the source coding unit. And likewise, the modeling stage without a proper source coding scheme is also futile (i.e. encode the decorrelated image using regular binary coding), because the actual bit length would not be reduced.

Modeling

The purpose of modeling is to identify spatial redundancy within the image frame, caused by correlation between samples. This correlation indicates a predictive relationship, and if the data is predictive there is no need to store this data for further use by the decoder. So by representing each sample as a predicting function of a subset of previous samples, and only store the error of the relationship between the function value (predicted value) and the actual value, the information stored for each sample is said to be decorrelated with respect to this subset. This representation will be referred to as the prediction residual,

$$\varepsilon = \hat{x} - x, \tag{2.3}$$

where $\hat{x}$ is the predicted value, and x is the actual value.

When considering the statistics of the prediction residuals throughout the image frame, ε can be thought of as a stochastic variable. It is an accepted observation that the probability density function of the prediction residual, $\mathrm{P}(\varepsilon)$, has a two-sided geometric distribution. If the prediction is based solely on the previous sample subset *a* - *e*, the expected value of ε implicitly equals zero, and it has an unknown variance.

The subset in which the decorrelation is performed has to be limited to a few previous samples (a through e), considering the complexity of the computation, so not all spatial information is decorrelated. In order to compensate for this, more parameters can be introduced when calculating the prediction residual. Since ε can be modelled as a stochastic variable, it would be useful if we could produce a better model for the expected value as well as the variance. So, by assigning each sample a context based on the gradients of the surrounding samples, we can assume that all samples within the same context will have similar statistical properties. This would lead to the possibility to implement additional parameters in the computation of ε, since the prediction errors from previous samples, would give an idea on how well the prediction itself performs within the given context. The accumulated prediction errors of samples within the same context can produce an offset parameter for the expected value of the prediction residual as well as a parameter for the variance.

The stochastic variable ε can now be expressed as a function of the subset a through e as well as a function of previous samples within the same context,

$$\varepsilon = \hat{x}(a...e) - x + C(\varepsilon'), \tag{2.4}$$

where C is the offset parameter of the expected value computed on the basis of previous values of the prediction residual, ε'.

In order to make a general description of the modeling process, relevant to a large number of available compression schemes, the main steps involved can be summarized as following [11]:

1 A prediction step, in which the predicted value of sample x is computed based on a subset of previous samples.

2 Determination of a context where x occurs.

3 Finding a probabilistic model for the prediction residual based on the context from the previous step.

4 Output the result from step 1 as well as the parameters of the model in step 3 to the coder.

Some of the parameters output from step 3, like the offset parameter of the expected value can be added directly to the prediction residual (eq. 2.4) and be utilized by all coders. Others are used as optimization parameters of the coding process itself. This includes the variance of ε, where the distribution of the code lengths of the individual characters, can be adapted to this parameter (this will be explained in the next section). If the coder is not adaptive, there is no need to compute these parameters. Also, in a number of less complex compression algorithms, step 2 and 3 are omitted completely, and the prediction residual coded is based on the result from step 1 only.

Entropy coding

The purpose of entropy coding is to produce shortest possible bit lengths based on the probability distribution of the samples to be coded. The main objective is to obtain a average bit length per sample as close to the entropy (eq. 2.2) as possible. The two most common techniques are *Huffman coding* and *arithmetic coding*. Both of these are based on obtaining a probability table before the actual coding process. If the statistics of the probability distribution is known in advance, universal codes like *Golomb codes* may be used directly without computing a probability table first.

Golomb codes As stated by [7], it is a widely accepted observation that the global statistics of residuals from a fixed predictor in continuous-tone images are well-modeled by a two-sided geometric distribution (TSGD) centered at zero. They further give a complete characterization of optimal prefix codes for this type of distribution. The family of optimal codes is an extension of the Golomb codes [8], which are optimal for one-sided geometric distributions.

Golomb codes has a tunable parameter m and in order to illustrate the effect of this, a list of Golomb codes for different values of m, is given in table 2.2. It is clear that choosing codes with parameter $m = 1$ is only optimal if it is highly likely that values are close to zero, e.g. coding value 255 with $m = 1$ would result in a code length of 256 bits. This can be interpreted as the variance of the prediction residual, and an m with a higher value would be preferable if the variance is high.

Table 2.2: *Golomb codes for different values of m*

$M(\varepsilon)$	$m = 1$ $(k = 0)$	$m = 2$ $(k = 1)$	$m = 4$ $(k = 2)$	$m = 8$ $(k = 3)$
	$q{\cdot}r$	$q{\cdot}r$	$q{\cdot}r$	$q{\cdot}r$
0	0·	0·0	0·00	0·000
1	10·	0·1	0·01	0·001
2	110·	10·0	0·10	0·010
3	1110·	10·1	0·11	0·011
4	11110·	110·0	10·00	0·100
5	111110·	110·1	10·01	0·101
6	1111110·	1110·0	10·10	0·110
7	11111110·	1110·1	10·11	0·111
8	111111110·	11110·0	110·00	10·000
9	1111111110·	11110·1	110·01	10·001

The codes can be further limited to a minimal complexity sub-family of the Golomb codes, called *Golomb-Rice codes*, in which the tunable parameter m is a power of 2 ($m = 2^k$). This makes Golomb-Rice codes more convenient for use on computers, since multiplication and division by m, would only imply shifting operations. Further in the report k will be used when referring to the tunable parameter.

In order to produce a Golomb Rice code from the input value N with the parameter k, the output code is divided into the quotient part q and the remaining part r.

$$q = \left\lfloor \frac{N}{2^k} \right\rfloor = (N >> k) \tag{2.5}$$

$$r = N \mod 2^k \tag{2.6}$$

The quotient is represented by unary coding [9, p. 57] (q-length string of '1', ending with a '0'), and the remainder is represented by regular binary coding, with a code length of k. The resulting code length will thus be:

$$N'_{len} = q + 1 + k \tag{2.7}$$

To illustrate this with an example, let say we want to encode $N = 9$ with parameter $k = 2$:

$$q = \left\lfloor \frac{9}{2^2} \right\rfloor = 2 \Rightarrow q_{bit} = 110$$

$$r = 9 \mod 2^2 = 1 \Rightarrow r_{bit} = 01$$

The resulting code will be:

$$N'_{bit} = 11001 \text{ and } N'_{len} = 2 + 1 + 2 = 5$$

2.3 LOCO-I

As mentioned above, the LOCO-I algorithm which was implemented in C++ during the preliminary project, is the basis for this thesis. For convenience, the explanation of the implementation will be repeated here as a theory background for the VHDL implementation. ([1] section 4.1)

LOCO-I is the algorithm used in the JPEG-LS codec [3]. In the next sections the different components of the LOCO-I algorithm will be presented. The modeling part of the scheme consists of a first order prediction step, followed by a context modeller. The coding part uses Golomb-Rice codes to represent the data, because of its ability to create optimal prefix codes for geometrically distributed stochastic variables.

Prediction

The predictor of LOCO-I is context dependent, and it consists of a fixed term and an adaptive term. The prediction residual can be written as,

$$\varepsilon = \hat{x} - x + C(c). \tag{2.8}$$

Here x is the actual value, $\hat{x}$ is the value of the fixed predictor and $C(c)$ is the adaptive term determined by the context c. The calculation of $C(c)$ is done in the context modeling stage, and will be further discussed.

The fixed term is determined by the neighboring pixels a, b and c (figure 3.2). More specifically, the fixed predictor guesses [11],

$$\hat{x} = \begin{cases} \min(a,b) & \text{if } c \geq \max(a,b) \\ \max(a,b) & \text{if } c \leq \min(a,b) \\ a+b-c & \text{otherwise.} \end{cases} \tag{2.9}$$

The predictor switches between three simple predictors. It can be interpreted as picking b in cases where a vertical edge exists left of the current location, a in cases of an horizontal edge above the current location, or $a+b-c$ if no edge is detected [11].

The expected value of the prediction residual, resulting from the fixed part of the prediction is implicitly zero.

$$E(\varepsilon_{fixed}) = E(\hat{x} - x) = 0. \tag{2.10}$$

However, if the context of the sample to be predicted is considered, there is usually an offset in the expected value present (e.g. parts of image with an exponential gradient). The purpose of the adaptive term is to cancel out the context dependent offset, and it is calculated during the context modeling part of the algorithm. The offset results in a new expected value,

$$E(\varepsilon) = E(\hat{x} - x + C(c)) = C(c). \tag{2.11}$$

Context determination

In order to index each sample into a specific context, the context must be calculated and quantized into a limited set of possible values. The context is based on three gradients present at the current sample. More specifically,

$$\begin{aligned} g_1 = &\quad d-b \\ g_2 = &\quad b-c \\ g_3 = &\quad c-a. \end{aligned} \tag{2.12}$$

g is further quantized into 9 regions:

$$q(g) = \begin{cases} 0 & \text{if } g = 0 \\ \pm 1 & \text{if } g = \pm\{1,2\} \\ \pm 2 & \text{if } g = \pm\{3,...,6\} \\ \pm 3 & \text{if } g = \pm\{7,...,20\} \\ \pm 4 & \text{if } g = \pm\{e | e \geq 21\} \end{cases} \tag{2.13}$$

These regions can be adjusted for different types of images, but those presented here are the default ones for an 8-bit/sample alphabet, which is the only alphabet considered in this report, and hence the only regions considered. The quantized values are denoted by q_1, q_2 and q_3, and gives a total of $9 * 9 * 9 = 729$ possible combinations. This can be further reduced by assuming that opposite q-triplets in terms of sign, represents the same context information, only opposite. So if the first non-zero element of the q-triplet is negative, the signs are switched, and a sign register is set, so the decoder can switch back the signs. This results in a total of 365 possible combinations of q. The q-triplet is converted in to a parameter c, further referenced to as the *context index*.

Context modeling

In the above sections both the prediction process and the coding process includes context dependent parameters. These parameters represents respectively the expected value, and the absolute deviation of the stochastic variable ε. In order to find the expected value, C, it would be intuitive to sum all previous values of ε within the given context (B), and divide by N numbers of samples, to get the average value. Likewise, the absolute deviation could be calculated by summing the *absolute value* of ε (A), and divide by N. However, in both cases, this includes a division operation which is not desirable due to its complexity. In the task of finding the absolute deviation, both the accumulated sum, A, and occurrences, N, is passed on to the coder, and the need for a division operation is not necessary during this phase (and the way the coder handles this task, entirely eliminates the need for a division operation.)

The authors of [3] propose a way of calculating C, without the need for a division operation, which is also less sensitive to "outliers", i.e. typical large errors can affect values of C until it returns to its stable value. The procedure is shown in the below code display.

Display 2.1: *Context computation procedure*

```
A += abs(e);    // accumulate abs of prediction residual
B += e;         // accumulate prediction residual
N++;            // update occurrence counter
if ( B <= -N ){
  C--;
  B += N;
  if ( B <= -N ){
    b = -N + 1;
  }
}else if ( B > 0 ){
  C++;
  B -= N;
  if ( B > 0 ){
    B = 0;
  }
}
```

This procedure calculates all the context dependent parameters, $C(c)$, $A(c)$ and $N(c)$, needed by the prediction and coding phases.

Golomb coding

The prefix codes used for source coding is based on Golomb codes (referring to table 2.2 in section 2.2). Golomb codes has a tunable parameter m that is given by the context of the sample, and is determined by the accumulated sum of prediction residuals stored in $A(c)$, where c represents the context. $A(c)$ is thus computed during the context modeling stage described below. In order to be able to code negative values of ε, and utilize the one-sided optimal prefix codes provided by Golomb, the two-sided distribution is mapped to a one-sided distribution using a mapping function,

$$M(\varepsilon) = 2|\varepsilon| - u(\varepsilon), \tag{2.14}$$

where the indicator function $u(\varepsilon) = 1$ if $\varepsilon < 0$. This will index ε in the interleaved sequence 0,-1,1,-2,2,...

The Golomb coding itself is performed on the value to be encoded, $M(\varepsilon)$, with the parameter k. The code is broken into two parts, the quotient part q and the remaining part r.

$$q = \left\lfloor \frac{M(\varepsilon)}{2^k} \right\rfloor \tag{2.15}$$

$$r = M(\varepsilon) \mod 2^k \tag{2.16}$$

As mentioned above, the quotient is represented by unary coding, and the remainder is represented by regular binary coding. And the resulting code length is thus

$$len = q + 1 + k \tag{2.17}$$

The tunable parameter k is computed using the information stored in $A(c)$ and $N(c)$. The process can be described by this C-code:

Display 2.2: *Computing 'k'*

```
for ( k=0; (N<<k)<A; k++ );
```

This process will choose the next 2^k exceeding the average value of the accumulated sum of prediction residuals, A, over N previous samples. The motivation for this is obvious when looking at the coding of the quotient part, which is required to be low in order to obtain a short code length. In other words, choosing 2^k just above the value to be coded, will result in the quotient part being zero and the code length is then given by $k+1$. Since the encoding process has to be causal, k cannot be chosen based on the current sample (which would lead to the shortest code length regardless), but has to be based on previous values, hence the context modeling.

CHAPTER

THREE

SYSTEM LEVEL MODELING

The first stage in the design process consisted of developing a system level model of all the internal processes in the compression/decompression algorithm, as well as providing an interface for the whole design which comply with the specified input and output requirements. System C is used as the modeling platform, which enables high level description of hardware. It is built on top of the C++ language, where each module is implemented as a C++ class. The System C package includes a simulation engine, and the module-classes is registered as parallel event driven processes by the engine. These processes is the equivalent to clock driven modules in hardware.

The use of a modeling platform based on the C++ language makes it simple to create an interface that represents the communication with the outside design. This is realized through file reading and writing. The input data is read from a file stored on the disk, and then the 24 bit RGB samples are passed on to the design each clock cycle, creating a data stream of samples. Likewise, the output data from the design is written to another file.

Some of the temporary data created by the modules inside the design is also written to disk in order to check for compression ratios, and buffer overrun/underrun in the FIFOs.

Referring to section 2.2 image compression consist of two main stages, modeling and source coding. The modeling stage tries to convert each sample into a value which lies within a limited range, using a causal model which is reversible. While the source coding stage is for representing samples within this limited range with fewer bits than the original sample. This also means that samples lying outside of this region must be encoded with longer bit sequences. Consequently, the output from the source coding consist of samples with variable bitrate, imposing the need for a packing module which packs series of samples into a packet of fixed bit length.

The modeling stage need to cache previous samples, since these are used as modeling variables, as well as store information regarding how well the model perform in a given context, in order to improve the model for future samples.

This results in three main stages and internal memory for both the encoder and the decoder, and is illustrated in the below figure.

The above stages are presented in the following sections, starting with the internal memory which is called *modeling memory*. Followed by the modeling stage, the source coding stage, Golomb packing, and finally how the complete system is put together.

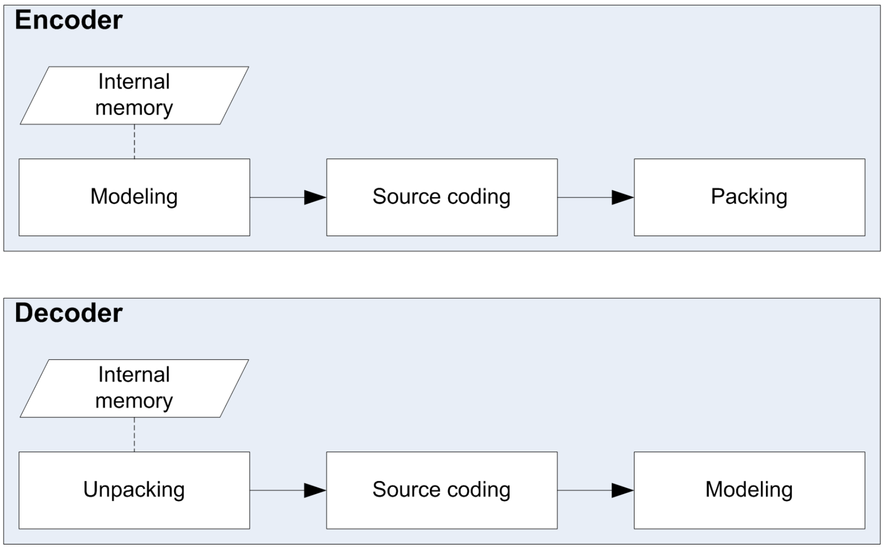


Figure 3.1: *Encoder and decoder stages*

3.1 Modeling memory and variables

Neighbouring pixels

Since neighboring pixels are read during the modeling phase, their value have to be stored in a cache register. The number of elements in this register is given by the number of subsequent pixels from the current pixel down to the earliest input pixel used by the modeller. The template used to describe neighboring pixels in this document is shown in the figure below. If pixel *c*, *b* and *d* is used by the modeller, the total elements that have to be stored by the register is given by $line\_width + 1$, where $line\_width$ is given by number of pixels in one line of the input video (in 1080p video this would be 1920 pixels) If f is included the total elements is $2 * line\_width$, consequently doubling the cache size. For the doubling of cache size to be expedient, the addition of f in the modeller, should give a corresponding increase in compression efficiency.

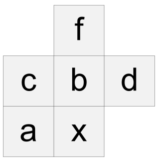


Figure 3.2: *Causal template of current sample 'x'*

The cache has been written as a shift register, with taps on the relevant neighboring pixels. Each time a new input pixel is ready to be modeled it is shifted in to the register at *x* (current pixel). A new input arrive each clock cycle. During encoding, the time from input arrives to it is shifted in to the register is one clock cycle, because no processing is necessary. This allows for the use of *a* in the prediction model. However, during decoding, the current pixel is not known until it is decoded by the same modeller. Consequently, *a* is not known by the modeller, since decoding takes more than one clock cycle to complete. This excludes *a* as a modeling variable and only pixels on previous lines are applicable (*b*, *c*, *d* and *i*).

Context register

The context register is where all the context information is stored. This information consists of the following variables:

A: Aggregate absolute value of the error residual, used by the Golomb source coder to determine the best *k*. This is done by performing the following operation on *A* and *N*,

$$2^k N > A, \tag{3.1}$$

and increment k until the equation is satisfied, up to a maximum value of $k = 8$. Since A is the aggregated absolute value, it will never go below 0. The maximum value is given when $k = 8$. Consequently, A will be in the range of $[0, 256N]$.

B: This is a temporary variable used by the context modeller to determine if C should be incremented, decremented or remain unchanged. From the context modeling algorithm (code display 2.1), B will always be in the range $[-N, 0]$

C: The error adjustment variable used by the predictor. It is increased or decreased until it reaches the current error residual. Since the error residual cannot be greater or lower than ± 255, this will also be the range for the variable A.

N: This is a temporary variable used by the context modeller to keep track of the number of pixels processed in the current context. (Also used by the Golomb source coder together with A)

As proposed by the LOCO-I algorithm, in order to keep the register size to a minimum, A, B and N is halved each time N reaches a predefined number of samples, N_0. Then, the range for each of the variables will depend on N_0 (except for C) and is summarized in the following table, along with the required bits for each variable in the context register, if we assume that N_0 is set to 32, which is the value used in the VHDL implementation.

Table 3.1: *Range of context variables*

Variable	Range	Number of bits ($N_0 = 32$)
$A < 2^8 * N_0$	$[0, 256N_0]$	13
B	$[-N_0, 0]$	6
C	$[-255, 255]$	9
N	$[0, N_0]$	6
Total		34

3.2 Modeling

This section describes the prediction model developed during the system level modeling phase of the project. The processing tasks in this model are simply referred to as modeling (i.e. modeling of input data to yield predicted output). The model is based on the LOCO-I algorithm, but is modified in order to comply with hardware limitations.

The following list summarizes the steps through the LOCO-I modeling phase:

- Find the context in which the pixel occurs, based on neighboring pixels. This is given an integer value within a specified range.
- Read information from context memory.
- Predict pixel value based on neighboring pixels.
- Adjust the predicted value based on information read from context memory.
- Update the context information based on how well the modeller performed.

Context determination (index module)

The gradients applied to determine the context of the current pixel, used by the LOCO-I algorithm, are based on pixel *a*, *b* and *c*. Because of the causality principle during decoding which excludes *a* as a valid variable, *a* could be substituted by *i* or simply left as 0. In the latter case, g_3 would be of no use leading to a reduction in possible contexts. Both scenarios are investigated further and compression results versus register sizes are presented in the results chapter. If *i* is included as a variable the gradients would be computed as follows:

$$\begin{aligned} g_1 &= d - b \\ g_2 &= b - c \\ g_3 &= b - f. \end{aligned} \tag{3.2}$$

As stated by the LOCO-I algorithm the gradients has to be further quantized in order to reduce possible contexts, denoted by q_0, q_1 and q_2. They are quantized in to 9 regions (-4 to 4), resulting in $9*9*9 = 729$ possible combinations, further halved by introducing a sign variable and then map equal gradients with opposite sign to the same context. This gives a total of 365 contexts.

If we think of the q-triplet as numbers in the base-9 number system the equivalent base-10 integer, c, for each possible triplet would be,

$$c = q_1 \cdot 9^2 + q_2 \cdot 9^1 + q_3 \cdot 9^0. \tag{3.3}$$

This conversion is not particularly suitable for hardware implementation, because it requires several multiplications and additions. It would be better if the radix of q was a power of 2. So, by choosing 8 quantization regions instead, c can be calculated directly in digital hardware by putting the 3-bit values of each q side by side,

$$c = q_1 \,\&\, q_2 \,\&\, q_3, \tag{3.4}$$

resulting in 9 bits. And then further reduce it by setting the sign bit if MSB is '1', and then use opposite sign values of each q instead, giving a total of 8 bits, or 256 unique contexts. The region represented by 0b100 has no unique 2's complement, and is thus omitted, resulting in 7 regions given by the following list:

Table 3.2: *Quantization regions*

g range	q	q bit
$[-128, -7]$	-3	101
$[-6, -3]$	-2	110
$[-2, -1]$	-1	111
0	0	000
$[1, 2]$	1	001
$[3, 6]$	2	010
$[7, 127]$	3	011

Example: If we have the following gradients,

$$\begin{aligned} g_1 &= -30 \\ g_2 &= 5 \\ g_3 &= -1 \end{aligned}$$

they will quantize into the following regions,

$$\begin{aligned} q_1 &= -3 &&= 101_2 \\ q_2 &= 2 &&= 010_2 \\ q_3 &= -1 &&= 111_2 \end{aligned}$$

Since MSB q_1 is '1', negative q-values are used. The resulting context is calculated as follows,

$$\begin{aligned} c = &\; -q_1 \,\&\, -q_2 \,\&\, -q_3 \\ = &\; 011 \,\&\, 110 \,\&\, 001 \\ = &\; 11110001_2 = 241_{10} \end{aligned}$$

Predictor and adjustment by context

Since A is excluded as a modeling variable, because of the causality principle, the predictor proposed by the LOCO-I algorithm is not applicable. The performance of several different predictors which meet the causality principle is evaluated and presented in the results chapter. Since the choice of predictor does not change the overall system model, and would be fairly simple to replace at a later time, the current predictor used in the rest of this chapter and also the one implemented in VHDL is,

$$p = b. \tag{3.5}$$

However, several predictor models were investigated and is further discussed. In the encoder, the error residual adjusted by the context information is given by,

$$\varepsilon = x - p - C, \tag{3.6a}$$

where x is the original pixel value, and C is the context adjustment variable. If the sign bit is set during context determination phase, the negative value of the error residual, before the context adjustment, is encoded. This is calculated directly as,

$$\varepsilon = -x + p - C, \tag{3.6b}$$

The decoder receives ε and finds the original pixel value x by reversing the above operations,

$$x = \varepsilon + p + C, \tag{3.7a}$$

or,

$$x = -\varepsilon + p - C, \tag{3.7b}$$

depending on the sign bit.

Update context information

The context update process is the same algorithm as the one used in the LOCO-I algorithm in code display 2.1 and will be repeated here for convenience. There has also been added a functionality that halves B, C and N each time the occurrence counter N reaches N_0 (As proposed by [3]). This is to reduce the maximum values, and hence the memory needed to store these variables. And also make the context information produced in earlier parts of the image increasingly less significant.

Display 3.1: *Context computation procedure*

```
A += abs(e);
B += e;
N++;

if (N > N0) {
  B = B/2;
  C = C/2;
  N = N/2;
}
if ( B <= -N ){
  C--;
  B += N;
  if ( B <= -N ){
    b = -N + 1;
  }
}else if ( B > 0 ){
  C++;
  B -= N;
  if ( B > 0 ){
    B = 0;
  }
}
```

3.3 Source coding

The source encoder receives the binary coded sample from the modeling stage and entropy encodes the data. The purpose is to represent the data with as few bits as possible. Consequently, the output has to be of variable width. This is not possible to achieve in hardware because the width of the output bus has to be fixed. Thus, another integer output representing the width of the current data output must be included. The bits outside of the size range, specified by this integer, will not be valid and is truncated later by the data packing module.

Similarly, the decoder must be able to read a variable sized data input. This procedure is more complex than during encoding because the length has to be calculated by reading the bits in the sample serially, due to the nature of prefix codes. And the start of next sample is not known until the length is known. Consequently, the decoder must receive fixed sized data packets, read the data serially and ask for another packet before it reaches the end of the previous packet.

Reduce maximum bit width

The width of the data output bus must be equal to the longest possible entropy code. By using Golomb codes, and an alphabet size of 256 (8 bit), the worst case bit length would occur if the sample $N = 255$ is encoded using $k = 0$. The unary encoded data to be transmitted will then have the value (referring to equation 3.8 and 2.7),

$$q = \left\lfloor \frac{N}{2^k} \right\rfloor = \left\lfloor \frac{255}{2^0} \right\rfloor = 255, \tag{3.8}$$

resulting in a size of,

$$N_{len} = q + 1 + k = 255 + 1 + 0 = 256. \tag{3.9}$$

When data is encoded, the packer module shifts data in to a register according to the length of the data. This is done every clock cycle, and the size of the data can potentially be of maximum length each clock cycle, resulting in the registers being 2 times as large as N_{max}

Also, the packets which are written to memory cannot be of smaller size than the maximum bit length, due to the fact that a series of worst case samples could be transmitted subsequently. This scenario would result in a buffer overrun in the packing module if the registers were smaller than the maximum bit length. Further, the the packet size cannot be greater than what the bandwidth of the external memory allows.

Thus, reducing maximum bit width is crucial in order to keep the registers in both the encoder and decoder, as well as the data packing modules, within a reasonable size. As proposed by the authors of the JPEG-LS algorithm [3], the data size can be reduced by imposing a maximum length q_{max} on the quotient variable q in the Golomb code. When q is greater than or equal to q_{max}, q is set to q_{max}, k is set to 8, and the remainder r contains the 8 bit unencoded sample. The maximum bit length will then be,

$$N_{max\_len} = q + 1 + k = q_{max} + 9. \tag{3.10}$$

The task of finding an appropriate q_{max} will be further discussed in the discussion chapter. During the rest of this chapter and the VHDL implementation chapter, q_{max} is assumed to be 7, resulting in a maximum length of 16 bits.

Encoder

The creation of Golomb codes is a fairly simple task in hardware, and not much calculation is necessary. As mentioned in the theory section, the Golomb codes consist of a unary encoded quotient q and a binary encoded remainder r, after the division of the input sample N, where 2^k is the divisor. Since the divisor is a power of 2, all divisions can be executed by the use of shifting operations.

Since the output data length must equal the maximum bit length as explained in the previous section, the invalid data can be padded with ones. These ones will continue the unary encoded q, which in turn eliminates the need to specify the start of q in the data output. The start of q will be specified through the size variable, which is computed by the arithmetic operation,

$$s = q + 1 + k = (N >> k) + 1 + k. \tag{3.11}$$

This is best explained through an example. The inputs, calculations needed to be done, and the expected output are listed in the following table. The example data to be encoded is 10101010_2.

Input	Binary	Decimal
k		6
N	$n_7\ n_6\ n_5\ n_4\ n_3\ n_2\ n_1\ n_0$	
(example)	1 0 1 0 1 0 1 0	170
Calculations		
$q = (N >> 6)$	$x\ x\ x\ x\ x\ x\ n_7\ n_6$	
	$x\ x\ x\ x\ x\ x$ 1 0	2
$r = N \mod 2^6$	$x\ x\ n_5\ n_4\ n_3\ n_2\ n_1\ n_0$	
	$x\ x$ 1 0 1 0 1 0	42
Output		
$data_{out}$	1 1 1 1 1 1 1 1 1 0 $n_5\ n_4\ n_3\ n_2\ n_1\ n_0$	
	x x x x x x x 1 1 0 1 0 1 0 1 0	
$size_{out}$	$q+1+k = 2+1+6$	9

The output data will then consist of the k least significant bits of the input, then a zero indicating end of q, and the rest will be padded with ones. This data is output together with the *size* variable.

Decoder

Since the length of the sample to be read is not known when the decoder loads data to its registers, it need to load packages of data of a fixed size. The package size must be equal or greater than the maximum bit length of the encoded data. As mentioned, we assume that the maximum length is 16 bits, and that the decoder receives packages of this size. Further, the register size in the decoder must be 3 times larger than this, resulting in 48 bits, in order to avoid buffer underrun.

The sample to be read may span two packages, i.e n number of bits stored at the end of package p, and m number of bits stored at the start of package $p+1$. If the beginning of a new package always starts with the first bit of a new sample, some functionality that enables the decoder to know the end of a package have to be implemented. The implementation presented here assumes that the input packages consist of a continuous stream of data, not interrupted by the end of a package.

An example of how the data is stored in the register is illustrated in the figure below. The first sample to be read starts at bit 0, and the next starts at bit 9. The only way the decoder knows that the next sample starts at bit 9 is by counting *q* until it reaches '0', and then add *k* number of *r*'s. Since a new sample has to be decoded each clock cycle, the task of finding next sample must not take more than one cycle to complete.

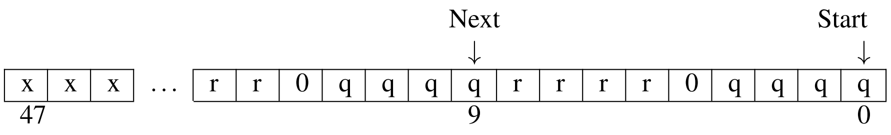


Figure 3.3: *Decoder register*

Data from start of current pixel to the start of next pixel is read by the next module. This module calculates *q* and *r* and combine them to give the corresponding 8 bit binary encoded data.

Determine 'k'

The calculation of *k* relative to the current sample has to be done before the Golomb coder starts encoding or decoding. The task to be executed is given in code display 2.2. It reads *A* and *N* from the context register, and based on this information outputs the corresponding *k*. The task can be seen as 9 parallel processes, checking $(N << k) \geq A$ for all possible *k*'s (0-8), and then output the right *k* based on the results.

Display 3.2: *Parallel computation of 'k'*

```
else if( (N << 1) >= A ) k = 1;
else if( (N << 2) >= A ) k = 2;
else if( (N << 3) >= A ) k = 3;
else if( (N << 4) >= A ) k = 4;
else if( (N << 5) >= A ) k = 5;
else if( (N << 6) >= A ) k = 6;
else if( (N << 7) >= A ) k = 7;
else k = 8;
```

3.4 Golomb packing

The packer module receives the variable bit length data from the Golomb encoder. As mentioned, this is represented by a data bus of 16 bit, and a size variable. This data has to be truncated relative to the size before it is stored in the registers in the packer module. In order to avoid buffer overrun in the register, the size has to be at least 2 times as large as the data bus, in order to avoid buffer underrun.

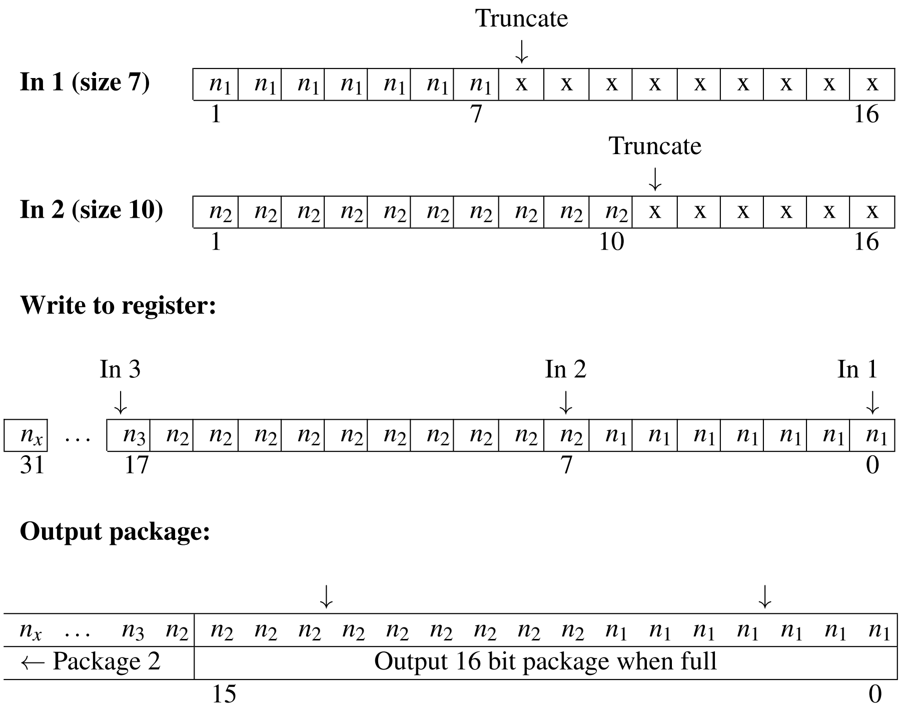

Truncate
In 1 (size 7)
n_1 n_1 n_1 n_1 n_1 n_1 n_1 x x x x x x x x x
1 7 16
Truncate
In 2 (size 10)
n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 x x x x x x
1 10 16
Write to register:
In 3
In 2
In 1
n_x … n_3 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_1 n_1 n_1 n_1 n_1 n_1 n_1
31 17 7 0
Output package:
n_x … n_3 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_2 n_1 n_1 n_1 n_1 n_1 n_1 n_1
← Package 2
Output 16 bit package when full
15 0

Figure 3.4: *Golomb packer register*

Figure 3.4 shows how the input data is truncated and added continuously to the register, and output to the memory writer. It works in a circular manner, so when the second package (bits 31 down to 16) is written, it proceeds at the start of the register (bit 0).

3.5 System put together

One of the main limitations when the algorithms are mapped to hardware is the need for pipelining, in order to keep up with the frequency requirement. This challenges the causality principle, because not all data are valid at the same time through the pipelined design. The algorithm was divided into top modules, so that a pipeline which are valid for both the encoder and decoder could be developed. All internal memory reads and writes in the LOCO-I algorithm were identified, as well as all internal dependencies. This is listed in table 3.3.

Table 3.3: *Top level module dependencies and internal memory reads/writes*

Encoder Module	Dependency	Memory read	Memory write
Context index	None	Line cache	N/A
Modeling	Context index	Line cache, Context mem	Line cache
Context update	Context index, Modeling	Context mem	Context mem
Find_k	Context index	Context mem	N/A
Source coding	Modeling, Find_k	N/A	N/A
Decoder			
Context index	None	Line cache	N/A
Find_k	Context index	Context mem	N/A
Source coding	Find_k	N/A	N/A
Modeling	Source coding, Context index	Line cache, Context mem	Line cache
Context update	Context index, Modeling	Context mem	Context mem

Based on this table, it was found that the decoder/encoder pair could be divided into a total of 5 top module pairs, placed in such a way in the pipeline that the internal memory writes and reads would contain the exact same data, and occur at the same stage. And also in such a way that the internal pipeline of each of the top modules could have an arbitrary length. The last criteria arose because of the need for a dynamic model which can be adapted to the not yet developed VHDL code, since we simply do not know at this stage how long the pipelines need to be in order for the implemented VHDL module to comply with the speed requirements. Thus, a model which includes the a dynamic delay of the output data of each System C implemented top module, relative to the length of its internal pipeline path, was necessary.

The pairs identified are the context determination part, the modeling part (both prediction and correction by context), updating of context information, the procedure of finding *k*, and source coding (Golomb).

In order to make the decoding process valid, it must perform operations on the data in the same order as the encoder. And it must also read to and write from the modeling memory, which contains the modeling variables, in the same order and at exactly the same pipeline stage as the encoder, in order for the stored data to be equal in both the decoder and the encoder. A top level model based on these criteria and the above table gives us the following pipeline diagram:

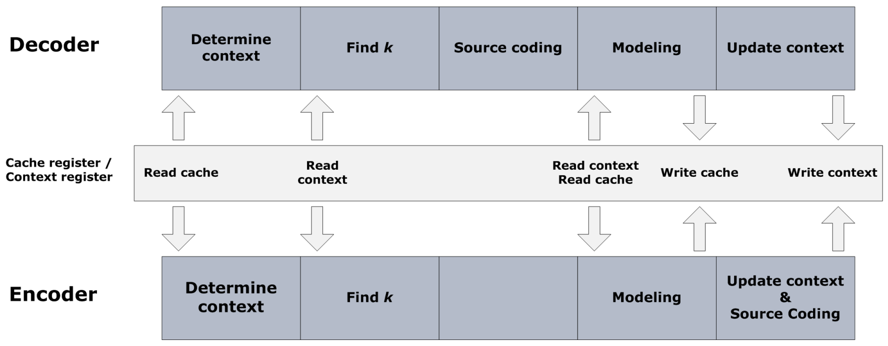


Figure 3.5: *Toplevel pipeline diagram*

Memory package and header data

The 16 bit output from the Golomb packer is about to be sent to memory for storage. A single FIFO represents the common memory of all three components, and additional header info is necessary in order to tell the elements apart. In order to reduce the total number of headers, 8 Golomb packages are combined into one larger 128 bit package. The total width of the elements in the memory FIFO is thus 130 bits, if it is assumed that 2 bits are needed in order to specify one of the three color components. It has not been decided at this stage how the packing scheme is best implemented, so this temporary scheme is used because it bears similarities to several of the possibilities discussed in chapter 7.3.

Memory package:

hh_x	Gp_0	Gp_1	Gp_2	Gp_3	Gp_4	Gp_5	Gp_6	Gp_7
$2b$	$128b$							

Figure 3.6: *Header data and Golomb packages (Gp_n) combined into one large package*

Example FIFO content:

Queue pos.		**130 bit package, 2b header + 128b data**	
#0		10_G	8 x Green
#1		01_R	8 x Red
#2		11_B	8 x Blue
#3		10_G	8 x Green
...			...
#n		10_G	8 x Green

Figure 3.7: *Packages queued in the FIFO*

When a package is pulled from the memory, the header is read first in order to identify the color and make sure that the 128 data bits are delivered to the correct decoder. The data portion is then split into the 8 smaller Golomb packages, and those are written to a 16 bit FIFO that act as a buffer for the decoding module.

System flow

Figure 3.8 illustrates all modules included in the System C implementation, and how they are connected.

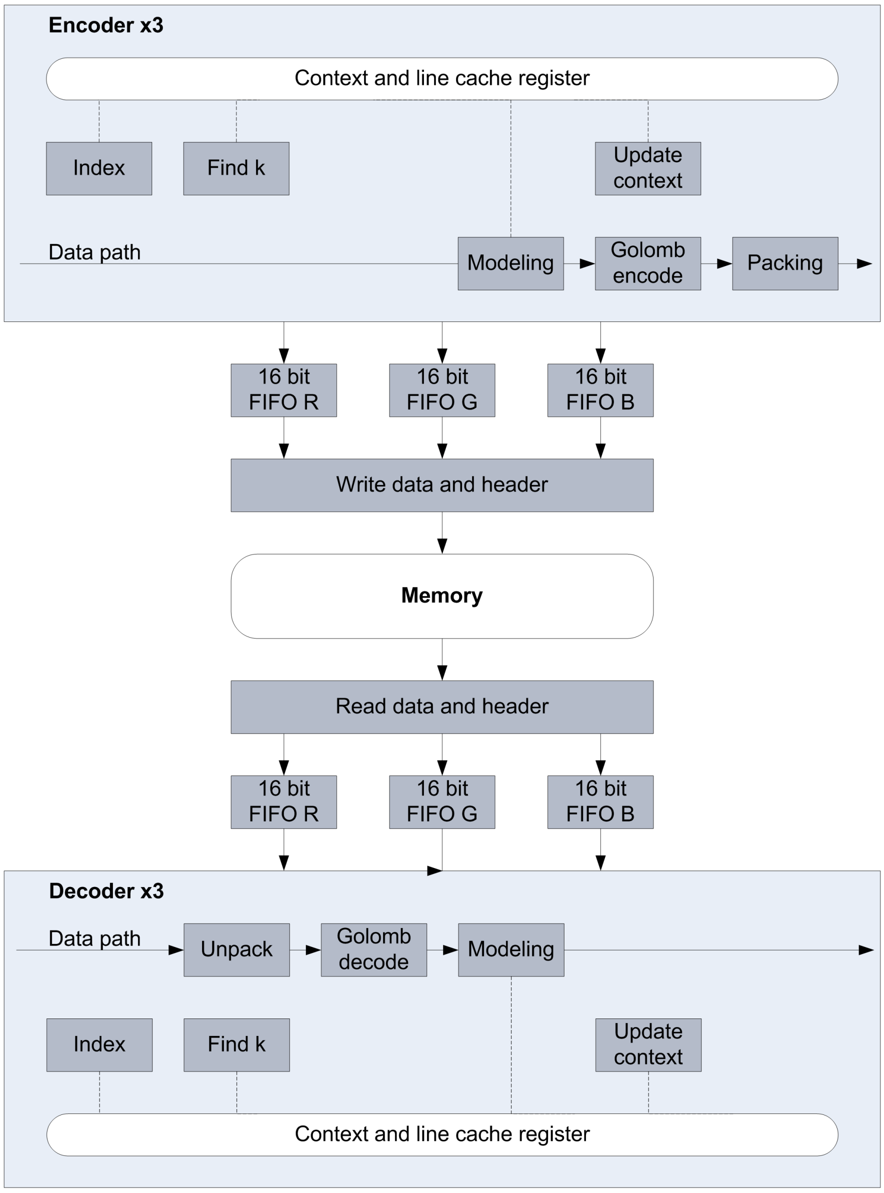


Figure 3.8: *System flow*

CHAPTER

FOUR

VHDL IMPLEMENTATION

Based on the above system level model, each of the top level modules *index*, *find_k*, *modeling*, *coding* and *packing* were realized using the VHDL language. The following sections will give an explanation of how each of the modules were implemented, and particularly how they are pipelined in order to comply with the speed requirement. This is important so that the above described system level model can be adapted to the signal delays caused by the pipelines, and further checked for consistency regarding the validity of data, and to see of this has any impact on the compression ratio.

In order to simplify the implementation, and only concentrate on the parts that is relevant in order to specify signal delays, FPGA resource usage, and critical paths for the whole design, only the relevant parts are implemented. This include an encoder module which encodes only one color component, that is assumed to input 8 bit data each clock cycle and output 16 bits Golomb packages, when the write buffer is full. And, a decoder module which inputs 16 bit Golomb packages, and outputs the decoded 8 bit color component each clock cycle.

Memory packing modules which create memory packages and RGB header data packages, are not implemented. A reason for this is that it is not clear at this stage what is the best way of doing this. Also, this is not relevant in order to make a working system. But a discussion on different possible implementations is given in chapter 7.

The test bench however, instantiates three of these encoder/decoder pairs, one for each of the color components, and emulates the peripheral memory and the memory packing modules. This makes it possible to simulate the whole design in order to check for validity of the decoded data, and also if the compression ratios are comparable to the ones obtained by the System C implementation.

One would think that the best way of checking for consistency between the System C implementation and the VHDL implementation, was comparing the Golomb packages created by the encoder. But it turned out to be very hard to obtain the exact same output data for both implementations, because only a tiny difference in the algorithm, results in completely different encoded data. So the validation of the VHDL code consisted of comparing the compression ratios obtained by the two systems, and then checking if the decoded data was identical to the original data.

4.1 Determine context and ’k’

index: The implementation of this module is pretty straight forward. It is pipelined in order to comply with the speed requirements. The pipeline consist of three processing stages, *gradient*, *quantize* and *index* which corresponds to the stages described in section 3.2 (Equation 3.2, table 3.2 and equation 3.4 respectively). The inputs are samples stored in the line cache, and the output is the context index, and sign.

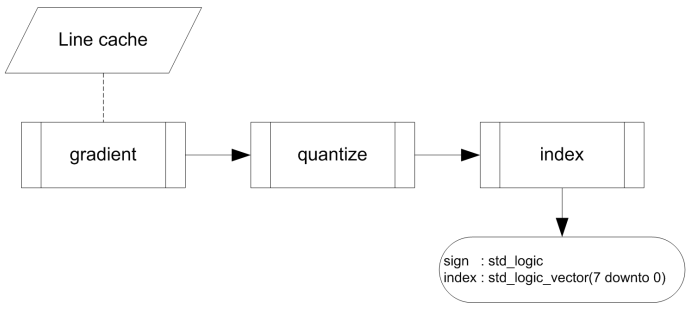


Figure 4.1: *Index module flow*

find_k: The calculations done during this stage is the same as those in code display 3.2. The output k, which takes on values from 0 to 8, is represented in 3 different ways:

- *k_out* : 9 bits. Only one bit is set, and its position represents the integer.
- *k_less_out* : 9 bits. All bits below the k position is set.
- *k_int_out* : 4 bit binary encoded integer value.

Different modules require these different representations.

It calculates all possible outputs in parallel and selects the output based on A and N stored in context memory. Since it has to read from context memory, the *index* module must compute the index-variable first since it acts as the address to the context memory. Because of the reading delay imposed by the embedded memory that is used on the FPGA, the module is pipelined through two stages, where the first stage is devoted to getting data from context memory only and store them in registers.

The *loco_index* and *loco_k* modules are instantiated by both the encoder and the decoder, and the VHDL code is given in appendix A.1.

4.2 Prediction and context update

modeling Prediction and context update has been combined into the same module. The prediction part which produces the error residual to be encoded is given in equation 3.6a/b and the reversed operation (during decoding) in equation 3.7a/b, and the context update procedure is given in code display 3.1. There are many arithmetic calculations needed to be done during these two stages, and in order to keep the critical paths short enough after synthesis, the procedures had to be pipelined. Code displays 4.2 and 4.3 show how the above mentioned procedures are divided into separate pipeline stages, for the encoder and decoder respectively. The complete pipeline is illustrated in figure 4.2, and the full VHDL code is given in appendix A.2 and A.5 for the encoder and decoder side respectively.

The reading of context memory is done in a separate stage (stage 1), because the memory read delay is quite large compared to register read delay. Also, during decoding, *data_in* has to be mapped to a two sided distribution (equation 2.14). This is done by the following VHDL code.

Display 4.1: *Golomb mapping function (Decoder)*

```
if data_in(0) = '1' then
  e(6 downto 0) <= not data_in(7 downto 1);
  e(7)          <= '1';
else
  e(6 downto 0) <= data_in(7 downto 1);
  e(7)          <= '0';
end if;
```

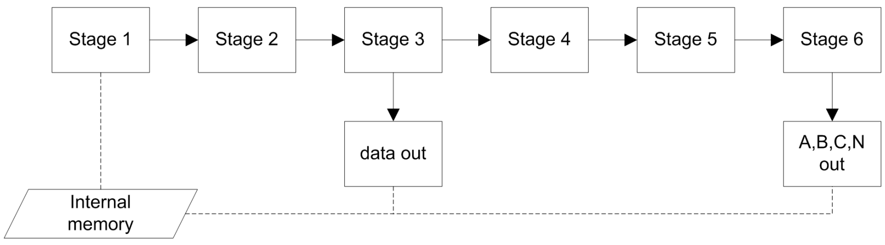


Figure 4.2: *Prediction and context update flow*

Display 4.2: *Pipelinging of prediction and context determination (Encoder)*

```
//--------------------------------------------------------------------
// STAGE 1
//
// Read A,B,C,N from context memory
// Read data_in, P (Predictor) and sign

//--------------------------------------------------------------------
// STAGE 2
  if (sign == 1) {
    e = data_in - P - C;
  }else{
    e = P - data_in - C;
  }

  if (N > N0) {
    A = A/2;
    B = B/2;
    N = N/2;
  }
//--------------------------------------------------------------------
// STAGE 3
  Q_out = e;

  A += abs(e);
  B += e;
  N++;
//--------------------------------------------------------------------
// STAGE 4
  if ( B <= -N ) {
    C--;
    B += N;
//--------------------------------------------------------------------
// STAGE 5
    if ( B <= -N ) {
      B = -N + 1;
    }
  }
//--------------------------------------------------------------------
// STAGE 4
  else if ( B > 0 ) {
    C++;
    B -= N;
//--------------------------------------------------------------------
// STAGE 5
    if ( B > 0 ) {
      B = 0;
    }
  }
//--------------------------------------------------------------------
// STAGE 6
  A_out = A;
  B_out = B;
  C_out = C;
  N_out = N;
```

Display 4.3: *Pipelinging of prediction and context determination (Decoder)*

```
//---------------------------------------------------------------------
// STAGE 1
//
// Read A,B,C,N from context memory
// Read data_in, P (Predictor) and sign

  e = map(data_in);
//---------------------------------------------------------------------
// STAGE 2
  if (sign == 1) {
    data = P + C + e;
  }else{
    data = P - C - e;
  }

  if (N > N0) {
    A = A/2;
    B = B/2;
    N = N/2;
  }
//---------------------------------------------------------------------
// STAGE 3
  Q_out = data;

  A += abs(e);
  B += e;
  N++;
//---------------------------------------------------------------------
// STAGE 4
  if ( B <= -N ) {
    C--;
    B += N;
//---------------------------------------------------------------------
// STAGE 5
    if ( B <= -N ) {
      B = -N + 1;
    }
  }
//---------------------------------------------------------------------
// STAGE 4
  else if ( B > 0 ) {
    C++;
    B -= N;
//---------------------------------------------------------------------
// STAGE 5
    if ( B > 0 ) {
      B = 0;
    }
  }
//---------------------------------------------------------------------
// STAGE 6
  A_out = A;
  B_out = B;
  C_out = C;
  N_out = N;
```

4.3 Golomb encoder

gol_enc_mux: The computation of the Golomb code is explained in the System level modeling chapter 3.3. In order do this in hardware a series of 16 MUX's is used. The input data is n (8 bit sample from modeling stage), 1 or 0, and the selector is determined by its position relevant to k. Data bits below k is given the value of its respective n. The bit at position k is 0, and bits above k is given value 1. The MUX is illustrated in the figure below, and is the same as the one proposed in the discussion section of the preliminary project [1].

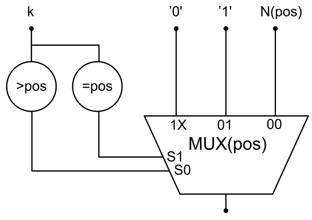


Figure 4.3: *Golomb MUX*

A series of 16 MUX's will give the following output data, if we assume that $k = 6$ and q becomes 2, as an example.

Input															
								q	q	r	r	r	r	r	r
								n_7	n_6	n_5	n_4	n_3	n_2	n_1	n_0
Output										↓	↓	↓	↓	↓	↓
x	x	x	x	x	x	x	q	q	q	r	r	r	r	r	r
1	1	1	1	1	1	1	1	1	0	n_5	n_4	n_3	n_2	n_1	n_0

Figure 4.4: *Calculation of golomb code*

The MUX module is called *gol_enc_mux* in the VHDL code.

gol_enc_shift: In order to find the *size* variable the length of q has to be calculated, this is done by shifting the input sample k times to the right. All possible results are calculated in parallel, and a MUX outputs the right q length based on the input k (equation 2.17)

gol_enc_top: The input data is geometrically distributed around 0 in the range [-128, 127], and has to be mapped to a one sided distribution, since the Golomb encoder only takes positive integers. This mapping function is given in the below in the VHDL code and corresponds to the mapping function given in the theory section, equation 2.14. This is a clocked process.

Display 4.4: *Golomb mapping function*

```
if input(7) = '1' then
  input_t1(7 downto 1) <= not input(6 downto 0);
  input_t1(0)          <= '1';
else
  input_t1(7 downto 1) <= input(6 downto 0);
  input_t1(0)          <= '0';
end if;
```

When the length of the q sequence is calculated by the shifting process, a process checking if q is 7 bits or larger is implemented, in order to comply with the maximum Golomb bit length. If this is the case, the r part of the output is set to the data given by the input sample N, and the size variable is set to 16. Otherwise, the output is set to the one calculated by the multiplexers, and the size variable is given by $q_{length} + 1 + k$. This is illustrated by the following VHDL code, and is a clocked process.

Display 4.5: *Select golomb output*

```
if (q_length > 6) then
  length_out <= "1111"; -- 16
  output     <= "11111110" & input;
else
  length_out <= q_length + 1 + k;
  output     <= output_mux;
end if;
```

The whole process of computing Golomb code is pipelined through 3 stages, and is illustrated in the figure below. All code is given in

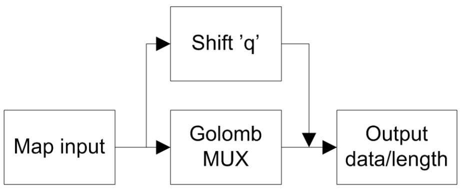


Figure 4.5: *Golomb encoder top level flow*

Golomb packer

wbuf Variable length Golomb codes require a module capable of buffering the output until a package of some constant size can be passed along. This module is called *Golomb Packer*, and the 16 bit packages it outputs are called *Golomb packages*. The module receives 16 bits of data from the Golomb encoder. These 16 bits contain the Golomb code word and usually some invalid data, so the size input is used to communicate how many of the input bits that make up the code word.

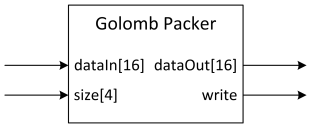


Figure 4.6: *Golomb Packer - main inputs and outputs*

For a buffer array A_{GP} of size N, the write pointer w_i is the index of A_{GP} from where the first bit of a code word is written. Because this index can take on any of the N possible values, the module logic must be able to connect a 16 bit input to any of the N positions. Using a small N will therefore help reduce the module size. Additionally, using a multiple of 16 as N is desirable, as it minimizes the required wiring between A_{GP} and the GP module output. Using a size N equal to two Golomb packages ensures that there always is room to store a code word. The output alternates between only two different locations in the array.

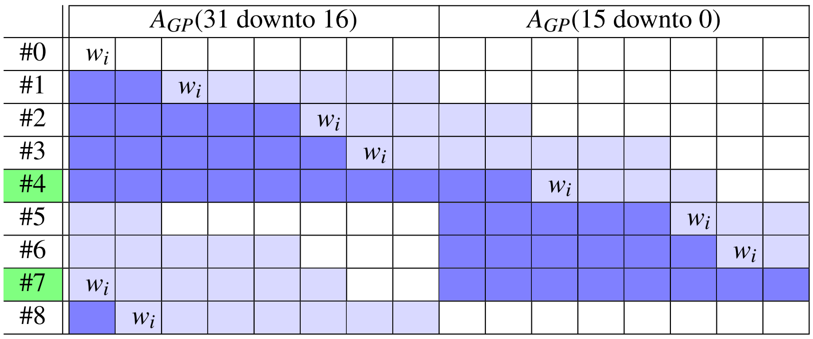


Figure 4.7: *Showing the A_{GP} array as it receives data and write Golomb packages when they are full*

Figure 4.7 shows A_{GP} as data is collected until a 16 bit package is ready to be outputted. Each space represents 2 bits of data. The white spaces are considered empty, light blue indicates the total 16 bit input, and the dark blue spaces indicate which of the 16 bits that are valid. When the array contains 16 bits of valid data, the *half* containing the valid data is pushed to the output, as indicated by green clk cells.

The input can typically be written straight into A_{GP} if there are at least 16 bit places behind the write position. If there is not enough room behind the write pointer, input must be written down to the last position, then continue from the front end of the array. The code displayed in 4.6 show the operations responsible for this functionality.

Display 4.6: *Golomb Packer wrap-around function*

```
if (writePos < 15) then
  writebuffer(writePos downto 0)      := input(15 downto x);
  writebuffer(31 downto writePos + 17) := input(x-1 downto 0);
else
  writebuffer(writePos downto writePos - 15) := input;
end if;
```

The remaining functionality can be summed up by the following steps:

1. Move write pointer and add to valid content tally according to size input
2. Check if valid content now is > Golomb package
3. Output if true, and subtract output length (16) from valid content tally

If a Golomb package is read to the output, the write port is set to '1' to signal a valid output.

All VHDL code presented in this section is given in appendix A.3.

4.4 Golomb decoder

The figure below shows each of the modules and the main registers which make up the Golomb decoder. It uses an indexed data register in order to keep track of where the samples are positioned. Each of the modules will be further explained, and also how the indexing system works.

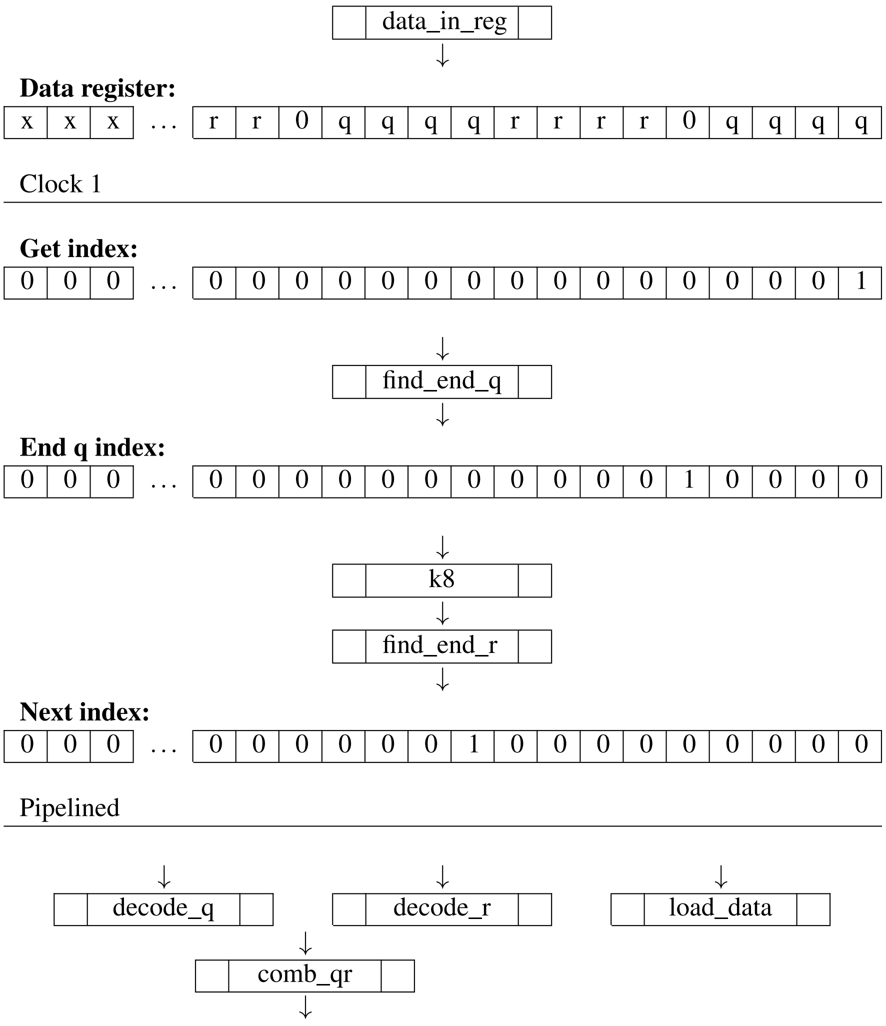


Figure 4.8: *Golomb decoder modules and registers*

Index registers

One possible way of reading the input samples which are of arbitrary length, is to start reading at the start of the data register, then shift the data register relative to the sample length, resulting in the next sample to be positioned at the start. And then start reading a new sample. Since the length of the input sample is not known until the sample is actually read, this procedure would in the best case need two clock cycles per sample to complete. This does not comply with the real time requirement of one sample per clock.

Thus, a system which consist of four parallel registers, one data register, and three registers that serves as index bits to the data register, was developed. These four registers, each consisting of 48 bits, are given the names:

- *data*
- *get_index*
- *end_q_index*
- *next_index*

data contains at any time 3 Golomb packets to be decoded. *get_index*, *end_q_index* and *next_index* has only one bit set at the rising edge of the clock, representing the start of sample, end of *q* sequence and the start of next sample respectively. This is illustrated in the figure below.

data	x	x	...	q	q	q	q	r	r	r	r	0	q	q	q	q
get_index	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0	1
end_q_index	0	0	...	0	0	0	0	0	0	0	0	1	0	0	0	0
next_index	0	0	...	0	0	0	1	0	0	0	0	0	0	0	0	0
	47															0

Figure 4.9: *Golomb decoder index registers*

Update index registers

All modules in this subsection is written to consist of only combinational logic, because all processing has to be done during one clock cycle. The top level design *update_registers* takes *get_index*-register, data register and *k* as inputs and calculates the new *end_q* register and *next_index* register using 3 sub-modules. This information is used by the *q* and *r* decoders to extract data at the right position, as well as being the basis for the new *get_index* updated on the next clock cycle, since *next_index* register is the consecutive *get_index* register. The 3 sub-modules are:

find_end_q: Since it is important that the critical path is as short as possible, it desired to parallelize the process as much as possible. We know that the unary encoded *q* cannot consist of more than 7 ones in a row followed by a zero. Based on this the *count_q* module was written and is shown in the figure below. It finds the first zero occurring in the 8 bit input data, and outputs its position if the *get* bit is set.

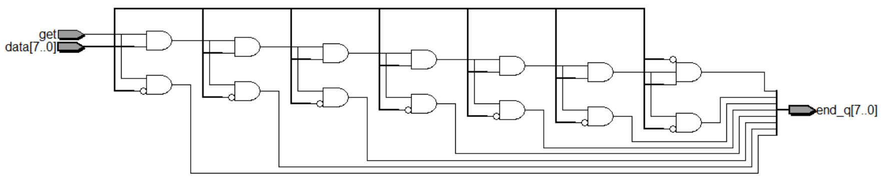


Figure 4.10: *count_q module*

48 such modules were implemented in an array, each representing one data bit position. It takes the data bit and get index at the modules current position and the subsequent 7 data bits as inputs. Based on this information, it outputs the position of the first zero occurring in the 8 bit sequence, and only if the get bit is set. Consequently, only one of the 48 modules will output a 8 bit signal with one bit active. All 7 bit data outputs from all modules are then processed through OR-gates resulting in a 48 bit signal with only one bit set, representing the end of the *q* sequence. This is stored in the *end_q_register*. This is illustrated in the below figure. The AND-gate pairs in the *count_q* module are represented as squares placed diagonally and relative to the data input. The data path produced by the example data stored in the data register, is marked in red. Only two *count_q* modules are included in the illustration for readability.

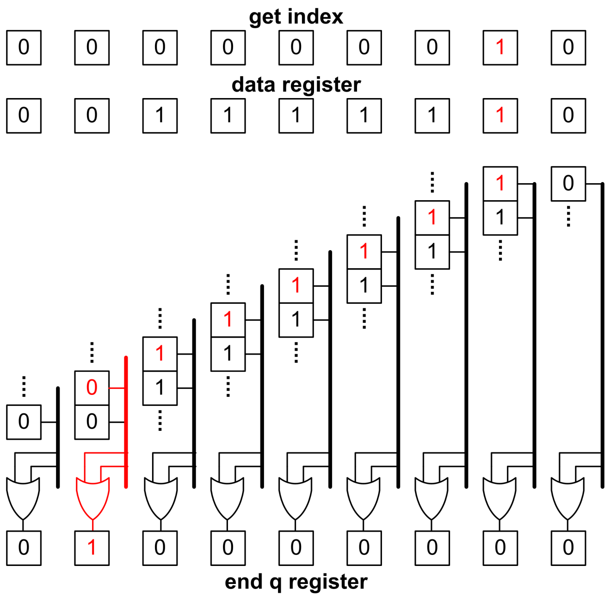


Figure 4.11: *find_end_q module*

k8: This module checks if the distance between *get_index* and *end_q* is 8 bits. It takes *get_register* and *end_q_register* as inputs, and uses 48 AND-gates checking if bit position n in the *get_register* and bit $n+8$ in the *end_q_register* is both '1'. In that case it sets it output high, indicating that $k = 8$, and the subsequent 8 r-bits consist of unencoded data.

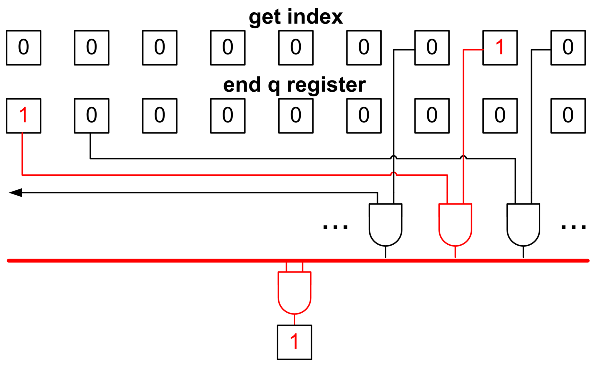


Figure 4.12: *k8 module*

find_end_r: Based on *k*, this module calculates the end of the *r* sequence, which is *k* bits after *end_q*. It takes *end_q_register* and *k* as inputs and calculates the end of *r* of all possible *k*'s, in 9 parallel processes (k = 0-8). Then it uses a MUX to select the right output based on the actual input *k*. The output data is given as a 48 bit register with only one bit set, representing the end of the *r* sequence. This register is the *next_index* and points to the start of the next sample.

Decode *q* and *r*

decode_q: All information needed to calculate *q* is stored in the *get_index* and *next_index* registers. This module takes the position of the *get* and *next* bit stored in these registers and convert them into binary coded integers. Then it calculates the difference of these two integers. Since maximum *q* length is 7, the output is truncated into 3 bits. This process is described in the below code, with *q_out* as the resulting *q* integer value. *q_out* is updated on rising edge, and the whole process takes one clock cycle to complete.

Display 4.7: *Convert 'get' and 'next' to integer*

```
with end_q select end_q <=
  "000000" when "000000000000000000000000000000000000000000000001",
  ...
  "101111" when "100000000000000000000000000000000000000000000000";

with get select get <=
  "000000" when "000000000000000000000000000000000000000000000001",
  ...
  "101111" when "100000000000000000000000000000000000000000000000";

calc_diff : process(clk) is
begin
  if rising_edge(clk) then
    q_temp <= end_q - get;
  end if;
end process calc_diff;

q_out <= q_temp(2 downto 0);
```

decode_r: This module has to extract the *r* bits stored in the data register at the position given by the *end_q_index* register. It uses an array of 8×48 multiplexers to select either the data in the data register or data from the previous MUX. The array is illustrated in figure 4.13. In the illustration the previous MUX is the one to the right. The select signal is given by the *get* bit, and it activates a whole diagonal row of MUXes. If the activate signal is high, it selects data from the data register, otherwise it selects data from the previous MUX. This enables the data bits to propagate all the way down to the left where it is read by the 8 bit output register. This process takes one clock cycle to complete.

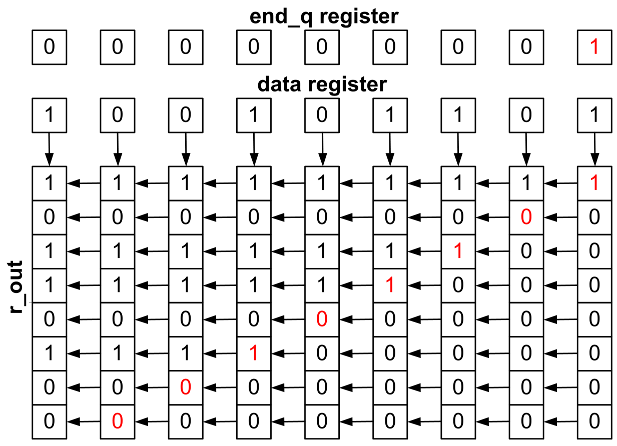


Figure 4.13: *decode_r module*

combine_qr This module takes the 3 bit q integer and the 8 bit r integer and combines them into the decoded 8 bit sample. This process is straight forward, and takes one clock cycle to complete. All possible combinations for the different k's are given in the following table. All 9 combinations are processed in parallel and a MUX selects the correct output based on k.

Table 4.1: *Combine 'q' and 'r'*

$k=0$	0	0	0	0	0	q_2	q_1	q_0
$k=1$	0	0	0	0	q_2	q_1	q_0	r_0
$k=2$	0	0	0	q_2	q_1	q_0	r_1	r_0
$k=3$	0	0	q_2	q_1	q_0	r_2	r_1	r_0
$k=4$	0	q_2	q_1	q_0	r_3	r_2	r_1	r_0
$k=5$	q_2	q_1	q_0	r_4	r_3	r_2	r_1	r_0
$k=6$	q_1	q_0	r_5	r_4	r_3	r_2	r_1	r_0
$k=7$	q_0	r_6	r_5	r_4	r_3	r_2	r_1	r_0
$k=8$	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0

Load data register

data_in_reg: The mechanism that loads new 16 bits Golomb packets in to the data register consists of two modules. The first module *data_in_reg* is the module that contains the data register and loads a new data packet when *ready_in* is active. The data register consists of 48 bits in order to store 16 bits packets in three fixed places. *ready_in* has two bits for selecting the portion of the register that is to be written to. It receives the packet at *data_in* and outputs the 48 bit register at *data_out*. The loading of data occurs in a circular manner, so the register represents a stream of continuous data.

load_data_in: This module is responsible of feeding *data_in_reg* with the *ready* signal, in order to load a new Golomb packet. It monitors the *get_index* and the *next_index* registers to learn if a previous Golomb packet has been fully read. Since the data register is divided into three 16 bit Golomb packet sections, it only need to monitor each of the sections for both *get_index* and *next_index*. E.g. if the *get* bit is set in section one, and *next* bit is set in section two, it means that section one is ready for a new packet on the next clock cycle. This is illustrated in the following figure. Here, the *get* bit is set in the first section, and *next* bit is set in section two. The active data path is marked in red. Since the max length of a sample is the same as the section size, there will never be an incident of *get* and *next* being in two registers not adjacent to each other, i.e. *get* is in section one and *next* is in section three.

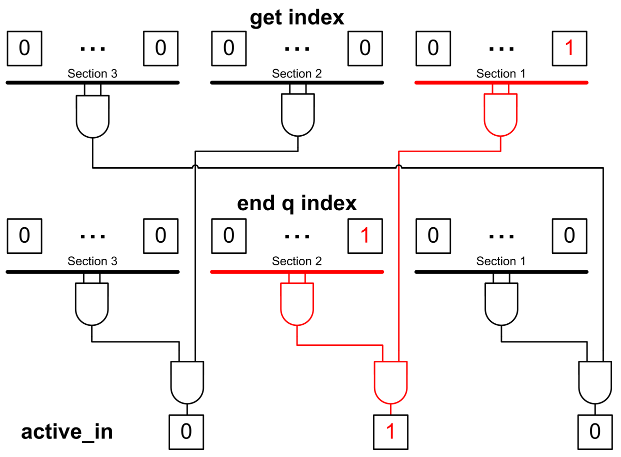


Figure 4.14: *load_data module*

All VHDL code presented in this section describing the Golomb decoder, is given in appendix A.6 in the same order as presented.

4.5 Modeling memory

The internal memory has been written in order to utilize the embedded memory on the Altera FPGA. The VHDL code for the following two modules, *line_cache* and *context_reg*, is given in appendix A.8

line_cache The line cache memory is written as a shift register with taps at the corresponding neighboring pixels to the current pixel that is being processed. During decoding, the fully decoded sample is shifted in after the modeling module has finished. Since the memory data has to be equal in both the encoder and the decoder, the encoder must shift in data at the same pipeline stage, even though the data is known at the first stage (input sample).

The context determination module uses neighboring samples to compute the gradients, as well as the prediction module. Since the prediction module is implemented at a later stage in the top level pipeline, the the taps which correspond to the neighboring pixels are not tapping the same memory elements as for the context determination module. This shift, has to correspond to the pipeline delay between these two modules.

context_reg This memory consist of the four context variables, A, B, C and N. The number of bits needed to be stored for each of the variables are 13, 6, 9 and 6 respectively, referring to table 3.1. This is repeated for 256 different contexts, and the *index* variable computed in the index module (context determination) is used as the memory address.

Both the *find_k* module and the modeling module must be able to read from the memory at the same time at different stages in the pipeline, at different addresses. While the modeling module is writing to a third address at the same time. Because of this the memory module has to support reading from two different locations as well as writing to a third location simultaneously.

4.6 Top level design

Based on the system level design from chapter 3, the above described VHDL modules were joined to yield a fully working encoder/decoder pair. Figure 4.15 and 4.16 shows how the modules are connected in order to comply with their internal pipeline length, for the decoder and encoder respectively. It is worth noting that reading and writing to both the line cache and context memory is done at the same pipeline stage for the encoder and decoder.

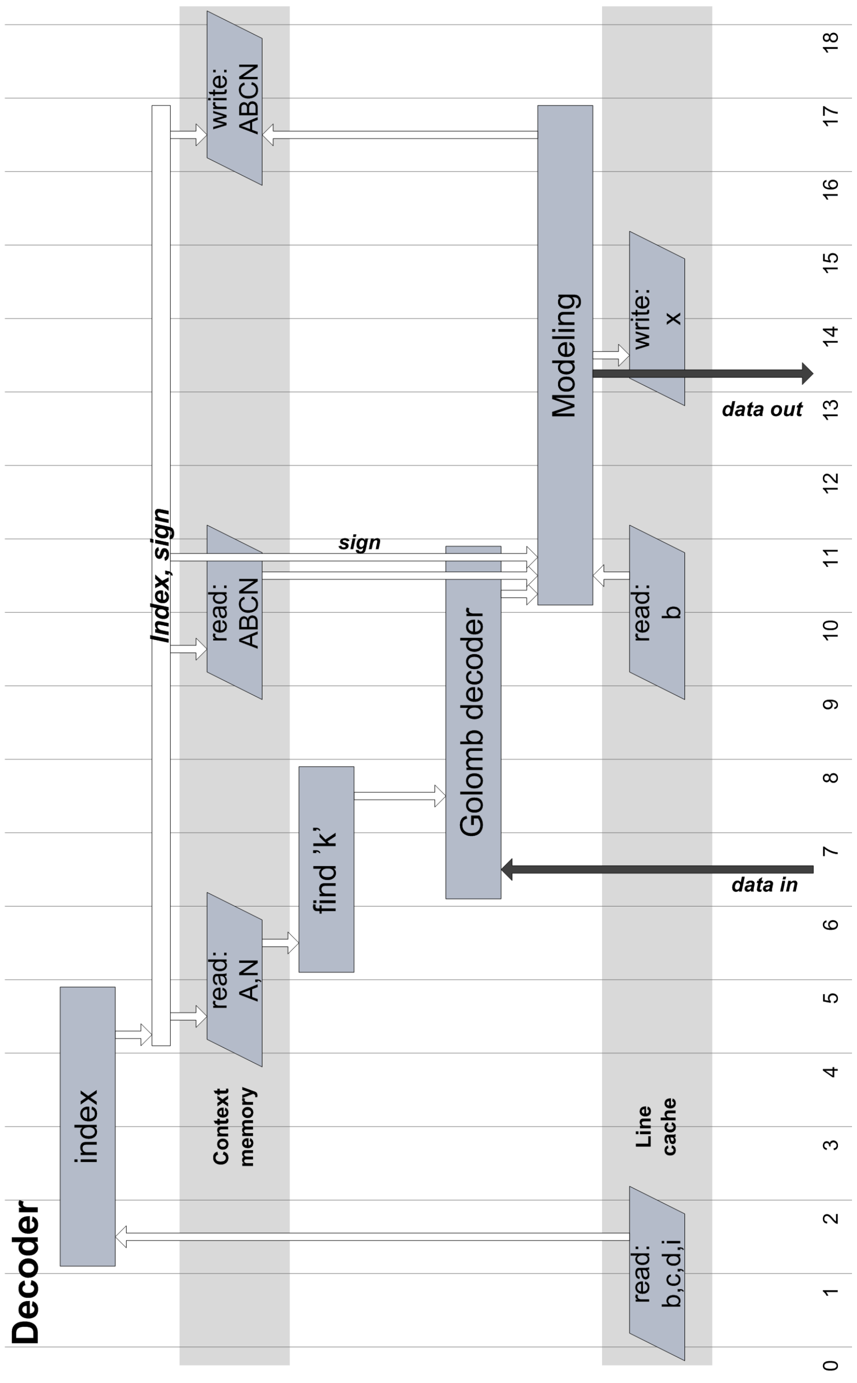


Figure 4.15: *Toplevel pipeline decoder side*

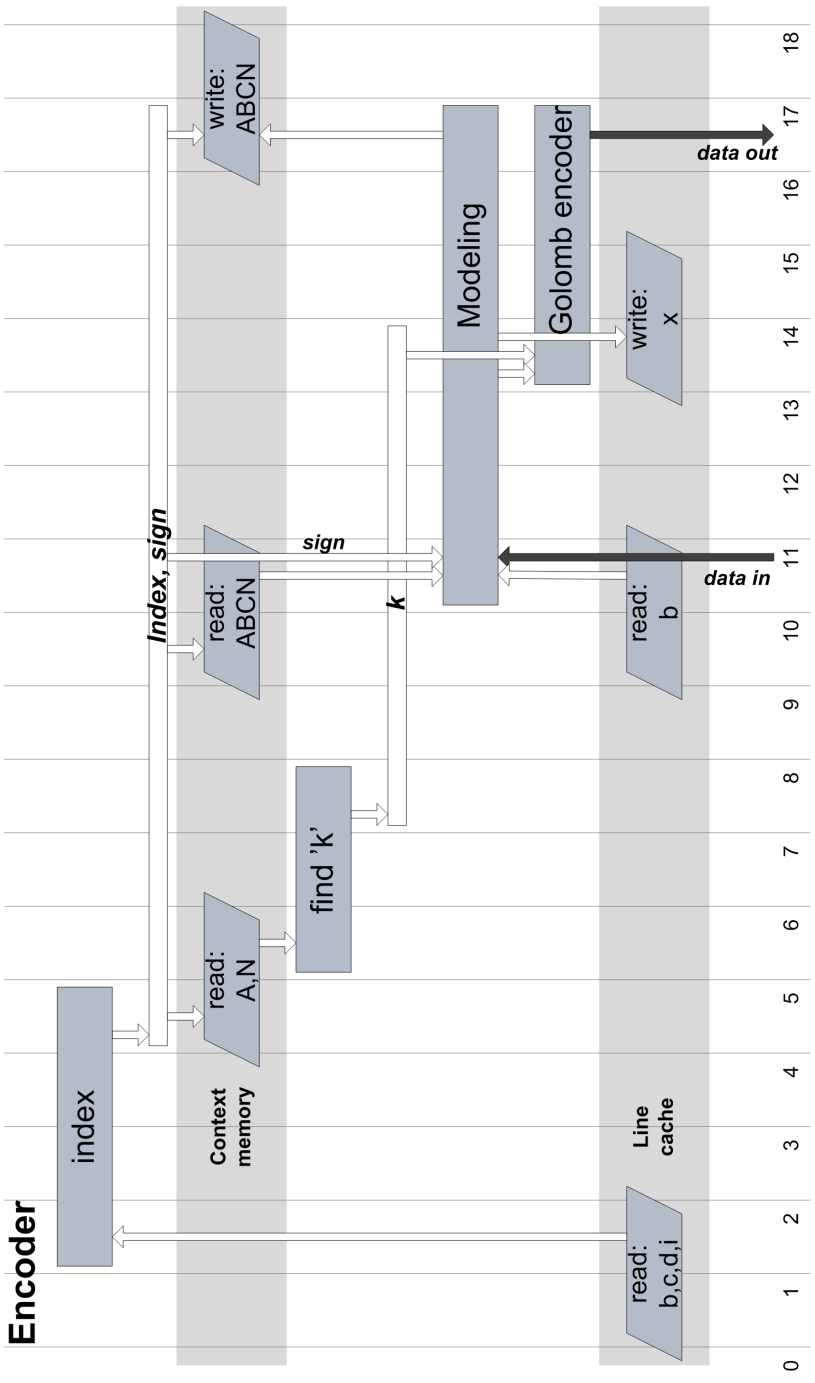


Figure 4.16: *Toplevel pipeline encoder side*

Signal delays

enc_signal_delays and dec_signal_delays The variables *index*, *sign* and *k* (decoder) are used by more than one module, in different stages of the pipeline. Thus, the signals must be stored and output at the right time. This is implemented by shifting the signals into shift registers each clock, and use taps placed relative to the reading modules position in the pipeline.

enc_reset and dec_reset The reset signal is also delayed relative to the pipeline path, so each of the modules are started at the right clock period, not processing or outputting invalid data.

The VHDL code for these four modules are given in A.2 and A.5 for the encoder and decoder side respectively.

The top level modules are given in appendix A.4 and A.7.

4.7 Proposed implementation to reduce resource usage

The above implementation uses context modeling in order to find the *k* value that gives the shortest code lengths, and to correct the prediction residual based on previous observations. This enables the algorithm to adaptively change the modeling parameters in order to ensure that the compression ratio is optimal of many types of images (Low/high noise, sharp edges, uniform areas etc.). However, context modeling require a significant amount of FPGA resources, and it would be interesting to see if it is still possible to acquire good compression ratios also when context modeling is not used.

The following list provides an overview of which modules that can be omitted, and the figure on the next page shows the new top level pipeline obtained when these modules are left out.

- Context memory
- Context determination module
- Find 'k' module
- Context update process in modeling module
- Shift registers for signal delays (*index*, *sign* and *k*)
- Simplified source encoding and decoding

This implementation would require *k* to be fixed. Compression ratios with different *k* values has been presented in figure 6.6, in order to find the most suitable for different types of images. A fully working implementation was synthesized and verified. The resource usage is presented in table 6.4 and 6.5.

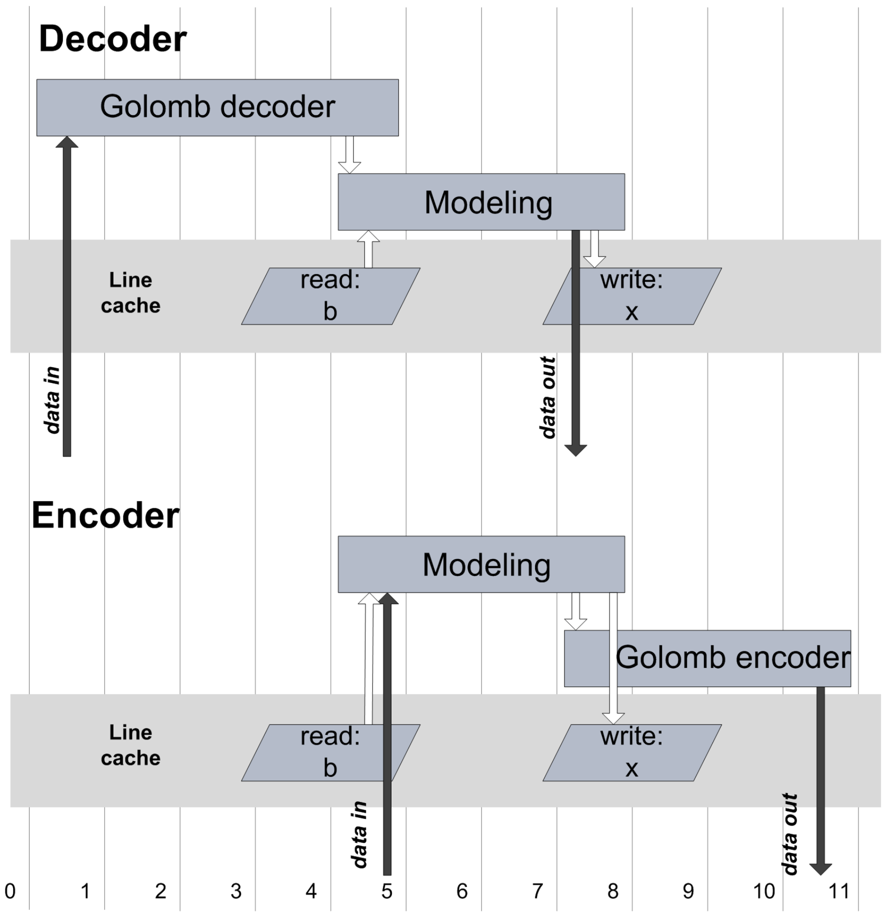


Figure 4.17: *Toplevel pipeline without context modeling*

CHAPTER

FIVE

TESTING AND VERIFICATION

The testbenches must instantiate the three modules necessary to process three color components, and take single frame data as input, as the scheme is meant to process only individual frames. The compressed data is written to a file on disk to be used by the decoder during decompression. Relevant compression statistics are presented after encoding, and the decoder checks that the decompressed information matches the original, thus verifying that the modules operate as expected.

Test images represent likely input data for a video conferencing unit. Both natural images, representing high entropy information, and captures of a computer screen, representing low entropy information, are used.

5.1 VHDL testbench

Input files

Bitmap binary files are not well suited to be read directly by a VHDL testbench using the textio package. RGB data must therefore be converted into a set of binary or hexadecimal vectors. The simplest and most readable way of reorganizing the data is to have one pixel per vector. Conversion is performed by a script written in MATLAB, outputting a file with 1920×1080 lines of 24bit hexadecimal vectors. Using hexadecimal vectors make them easier to read and take up less disk space. The first 10 values of the an input file are displayed in 5.1 below. Two and two hexadecimal symbols make up the values for R, G and B.

Display 5.1: *The first 10 vectors in the t1 test data file*

```
B4B2B2
B0AFAE
AEAEAD
AFAEAD
AEADAD
B0AEAE
B3B2B1
B4B3B2
B4B3B2
B4B3B1
...
```

Encoder testbench

The Encoder testbench instantiates 3 sets of encoders and Golomb packers, one for each color. Between their inputs and their outputs they operate independently. The input file is controlled by an input process, in which one line is read every clock cycle until the file is empty. A separate process manages the outputs. It is responsible for writing the Golomb packets to 3 separate output files whenever the Golomb packers' write signals are high. Figure 5.1 shows the overall flow in and out of the testbench as well as the upper level modules. The signal probing (Signal Spy) feature from the *modelsim_lib.util* package is used in order to debug signals that are buried deep within the design hierarchy without having to create unnecessary ports. Probing a signal and connecting it to a signal at the top level testbench is faster than digging through layers of modules with countless signals when setting up Modelsim simulations.

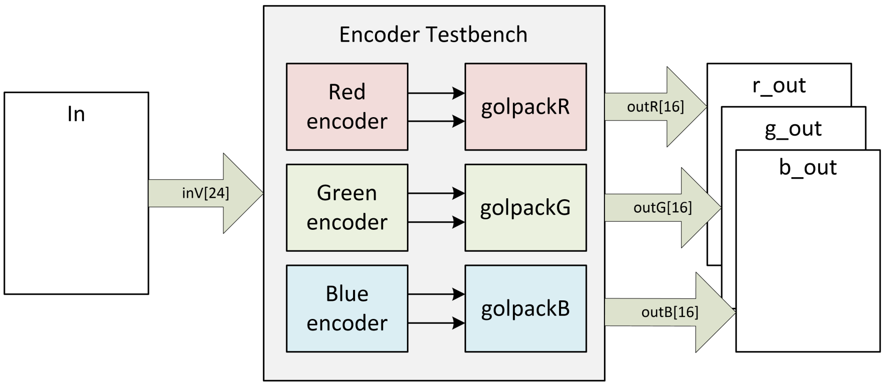


Figure 5.1: *Encoding testbench flow*

A compression ratio is calculated upon completion of the simulation, where bits out are divided by bits in and scaled to indicate the average number of bits per symbol (b/S). An example of the console output is displayed in 5.2. It shows compression ratio (b/S) per color component and total. In addition it lists the sizes of the header data for some different packet sizes.

Display 5.2: *Example of compression results printed to the simulation console (Transcript)*

```
# Ending simulation.
# Simulation statistics for t1:
#  -----------------
# R   : 1.772022e+000
# G   : 3.674012e+000
# B   : 1.945378e+000
#  - - - - - - - - - -
# Tot : 2.463804e+000
#  - - - - - - - - - -
# Head 16 : 239 KB
# Head 32 : 119 KB
# Head 64 : 59 KB
# Head 128: 29 KB
# Head 256: 14 KB
```

The input color data is not spectrally decorrelated and the operation is also not implemented as a module. Decorrelation calculation is therefore applied directly to the input vector within the input process. The code in 5.3 shows how the spectral decorrelation is performed when the *inV* vector is split between the three encoder inputs.

Display 5.3: *Spectral decorrelation step performed within the encoder testbench*

```
    inputR  <= inV(23 downto 16) - inV(15 downto 8);   -- R = R - G
    inputG  <= inV(15 downto 8);                       -- G = G
    inputB  <= inV(7 downto 0) - inV(15 downto 8);     -- B = B - G
```

The contents are found in the file *enc_tb.vhd* in [ref appendix].

Decoder testbench

The decoding testbench is pretty much mirroring the flow of the encoder testbench. It also instantiates three separate decoders, but it does not have an external buffering module similar to the Golomb Packer. The input process manages and loads packets from the three output files that were created by the encoder. It monitors the read signals and supply new packets whenever they are needed.

The decoder produces one pixel per clock cycle, so the output process need only to gather the three component outputs and reverse the spectral decorrelation that was applied in front of the encoders. This process also checks for errors by comparing each pixel to the original input file.

The contents are found in the file *dec_tb.vhd* in [ref appendix].

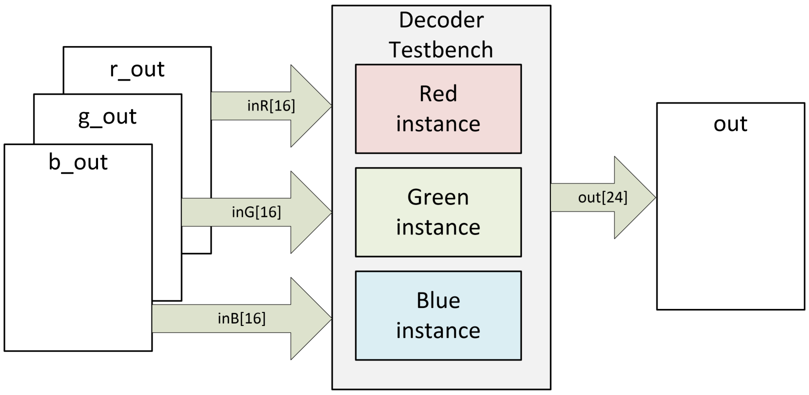


Figure 5.2: *Decoding testbench flow*

5.2 Test images and videos

The test images used are the same as in the preliminary project, and a video has been added to the set. Images (frames) from typical video conference situations were used, as well as "desktop"-images, i.e. snapshot of the screen and Power point presentations.

The individual frames in the short video sequence have a significant amount of what looks like random color fluctuation noise. Some of the other images do also have noise, but in those the fluctuations are in intensity. Variations in intensity is expected to respond well to the spectral decorrelation step.

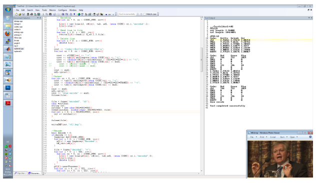
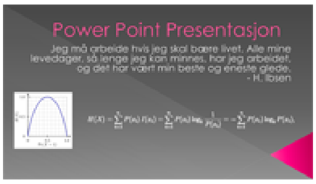
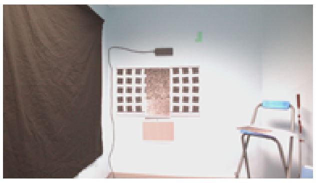
All images has a resolution of 1920×1080 pixels, and has been taken with a digital photo camera. The video has the same same resolution as the images, and the samples are represented by 24 bit RGB.

When the RGB information is divided into 3 individual components, the images consist of $1920 \times 1080 \times 3 = 6,220,800$ 8 bit samples each, and the video consist of $6,220,800 \times 60 = 373,248,000$ 8 bit samples to be processed.

The test set is presented in table 5.1.

The images *screen* and *pp* will be referred to as desktop images, because they represents screen captures of a computer desktop. The other images are referred to as natural images. Both types of images were included in the test set because of their relevance to video streams during conference calls.

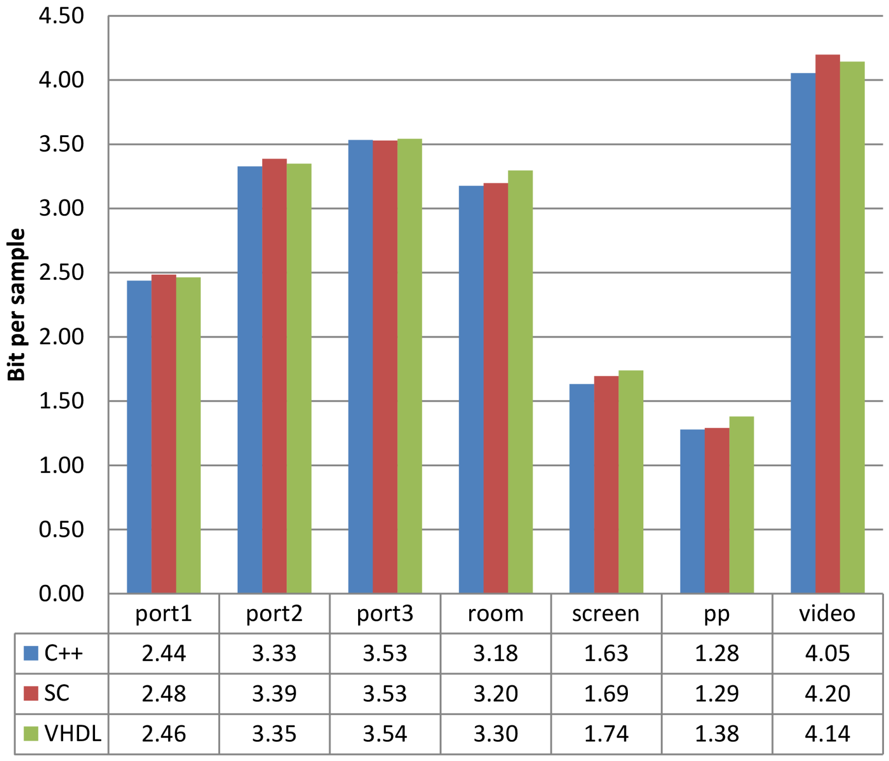
Table 5.1: *Test set*

Name	Specification	Image
portrait1	Portrait. Low noise (ISO 80). White, close to uniform background.	
portrait2	Portrait. High noise (ISO 1600) White, close to uniform background.	
portrait3	Portrait. Colored, not uniform background.	
room	Conference room. Low noise (ISO 80).	
screen1	Screen shot of MATLAB	
screen2	Power point presentation	 Power Point Presentasjon Jeg må arbeide hvis jeg skal bære livet. Alle mine levedager, så lenge jeg kan minnes, har jeg arbeidet, og det har vært min beste og eneste glede. - H. Ibsen $H(X) = \sum_{i=1}^{n} P(x_i) I(x_i) = \sum_{i=1}^{n} P(x_i) \log_b \frac{1}{P(x_i)} = -\sum_{i=1}^{n} P(x_i) \log_b P(x_i)$
video	60 frames video of conference room (1080p, 60fps, 24 bit RGB)	

5.3 Verification

During the verification phase of the VHDL design, it proved to be very hard to validate all internal data through all processing stages. This would require all computations and pipelines to be identical, since only one minor alteration, usually due to the C data types versus the logic vectors in VHDL, would propagate through the computation of all samples during the modeling stage, because of the high dependence on previous data. And result in completely different internal data. Based on this it was decided to only use the comparison of compression ratios between the C++, System C, and VHDL model as a measure on how successful the mapping of the original C++ algorithm had been. Figure 5.3 presents these results, and is further discussed in section 7.2 under 'Verification'.

The ability to decode the encoded data without error, and obtain the exact same data as the original input, is a strong indication that the algorithm is functioning as expected. Since the VHDL code has not been implemented on a FPGA in order to perform thorough tests on video streams, the validation is limited to using single frames and short video sequences in a simulation environment. The use of longer video sequences would simply take too long to process in order to be practically feasible. The final design was able to process all images and the video given in the test set, and completely decompresses all data.



	port1	port2	port3	room	screen	pp	video
🟦 C++	2.44	3.33	3.53	3.18	1.63	1.28	4.05
🟥 SC	2.48	3.39	3.53	3.20	1.69	1.29	4.20
🟩 VHDL	2.46	3.35	3.54	3.30	1.74	1.38	4.14

Figure 5.3: *Comparison of compression for C++, System C and VHDL models*

CHAPTER

SIX

RESULTS

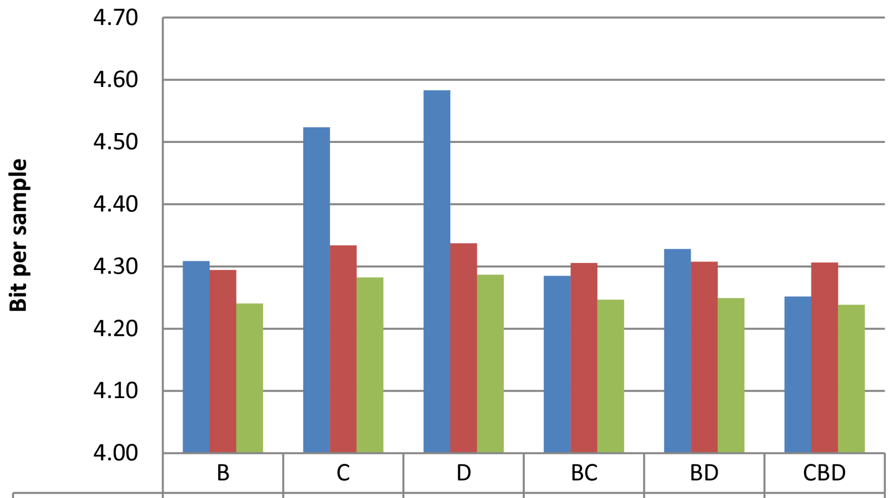
In this chapter the compression ratios obtained by running the test image set through the different implementations is presented. Some results were obtained during an early stage in order to find the modeling parameters to be further considered, and is based on the C++ implementation of the LOCO-I algorithm developed during the preliminary project. This includes the modeling variables, the maximum Golomb bit length and the *k*-parameter. Further, in section 6.4 the compression ratios obtained by the VHDL implementation is presented.

The synthesis results for both the encoder and the decoder is presented in section 6.5, and includes resource usage and maximum frequencies, as well as the top 5 worst-case timing paths. In section 6.6, results governing the simplified design, is presented, both compression and synthesis results.

6.1 Predictor and context determination variables

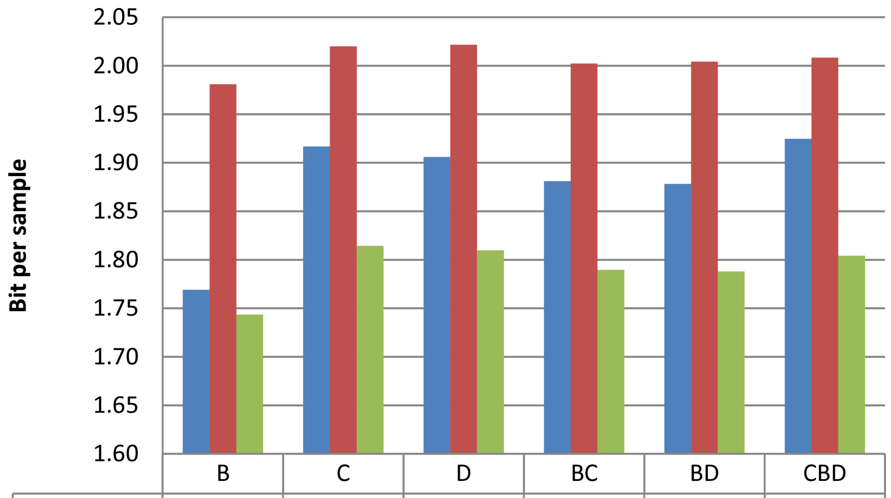
The following results gives information about the compression potential using different predictor and context determination variables. The purpose is to find the best variables in terms of compression and complexity. The C++ code developed during the preliminary project was used, and the test data consist of the images in the test set, represented in 24 bit RGB. The data is grouped in average for natural images and desktop images. The results are discussed in 7.1, but to summarize, it can be established that the choice of predictor is not significant to the compression results. Further, context modeling using two gradients to correct the error residual, worsen the desktop image compression ratio, compared to not correcting. Three gradients gives slightly better compression for both types of images.

All results are given in average bit length per 8 bit R/G/B input sample.



	B	C	D	BC	BD	CBD
🟦 No ctxt	4.31	4.52	4.58	4.28	4.33	4.25
🟥 b-c,b-d,0	4.29	4.33	4.34	4.31	4.31	4.31
🟩 b-d,b-c,b-i	4.24	4.28	4.29	4.25	4.25	4.24

Figure 6.1: *Predictors and context variables for natural images*

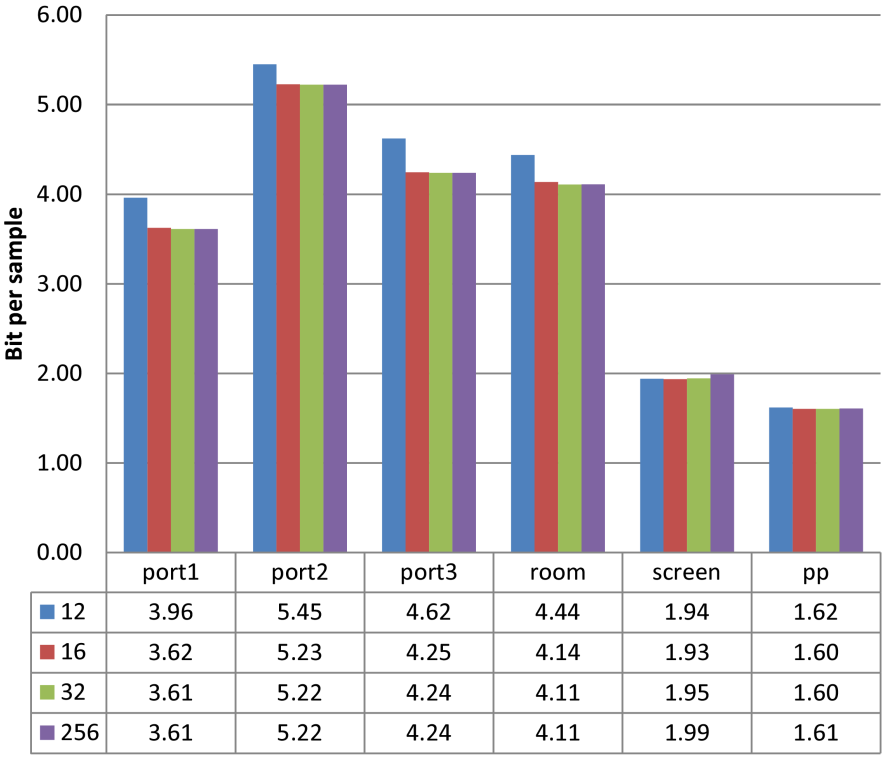


	B	C	D	BC	BD	CBD
🟦 No ctxt	1.77	1.92	1.91	1.88	1.88	1.92
🟥 b-c,b-d,0	1.98	2.02	2.02	2.00	2.00	2.01
🟩 b-d,b-c,b-i	1.74	1.81	1.81	1.79	1.79	1.80

Figure 6.2: *Predictors and context variables for desktop images*

6.2 Maximum golomb code length

The following results show the compression ratios when different maximum bit lengths are chosen regarding the Golomb data output. The test data consist of 24 bit RGB images, and is run through the C++ implementation. The results show that the difference in compression with bit lengths between 16 and 256 is negligible. While 12 bits are possible to use in order to reduce resource demand, 16 bit is the best choice regarding compression versus resource usage. All results are given in average bit length per 8 bit input sample.

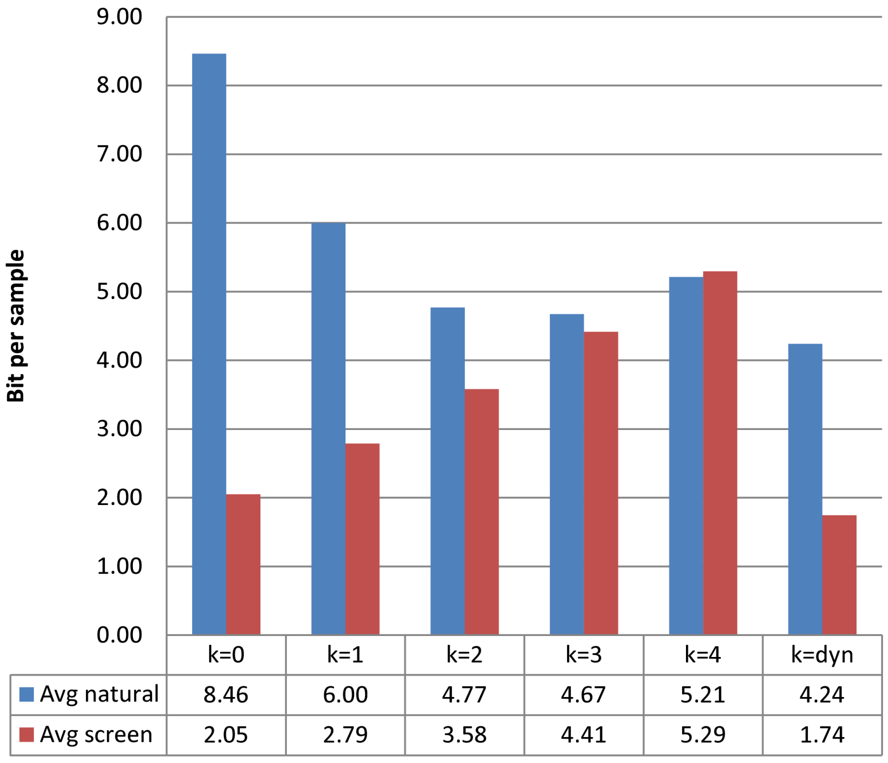


	port1	port2	port3	room	screen	pp
🟦 12	3.96	5.45	4.62	4.44	1.94	1.62
🟥 16	3.62	5.23	4.25	4.14	1.93	1.60
🟩 32	3.61	5.22	4.24	4.11	1.95	1.60
🟪 256	3.61	5.22	4.24	4.11	1.99	1.61

Figure 6.3: *Maximum code length impact on compression*

6.3 Fixed golomb ’k’ value

In order to test compressibility when fixed *k* values are used, in stead of dynamically adjusted values, the test image set was run through the C++ code from the preliminary project, in order to find out if this could be the basis for an alternative implementation. All results are given in average bit length per 8 bit input sample, and is grouped in average values for natural images and desktop images. Since *k* gives the lower bound for minimum bit length, values of more than 4 were not considered. The dynamic *k* variant is included for comparison.



	k=0	k=1	k=2	k=3	k=4	k=dyn
🟦 Avg natural	8.46	6.00	4.77	4.67	5.21	4.24
🟥 Avg screen	2.05	2.79	3.58	4.41	5.29	1.74

Figure 6.4: *Fixed ’k’ impact on compression*

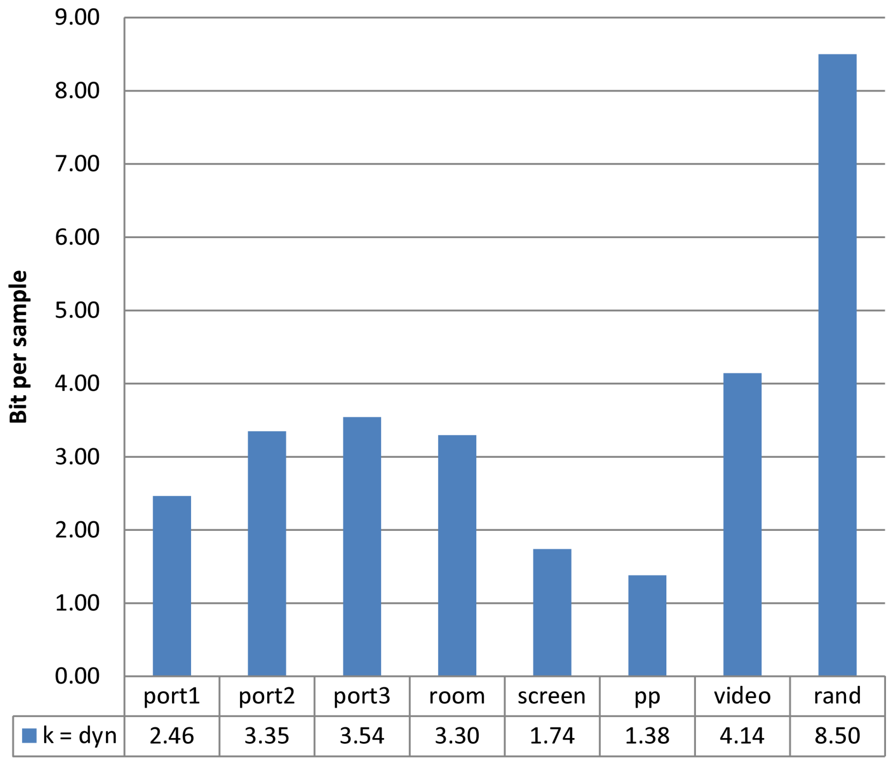
6.4 Compression results VHDL implementation

The compression ratios obtained by the final design, after all the test data was processed, is presented in figure 6.5. The test data consisted of spectrally decorrelated (R-G, G, B-G) images. And the results is represented in average bit lengths per 8 bit input data.

It shows that images with more noise has a lower degree of compressibility (when comparing *port*1 and *port*2). This is also the case for images with several objects that are causing sharper gradients than uniform areas (*port*3, *room*).

The *screen* and *pp* images have the highest compression because of their uniform nature, as expected. *video* has the lowest compression, and after further inspection it seems as if the noise generated by the image chip in the video camera is random across the spectral components, while the images taken by a digital photo camera has more 'grey' noise, indicating that the random data follows the same patterns in all three components. This leads to the conclusion that data from the video camera is less spectrally decorrelatable.

An image consisting of random data is also processed in order to check for worst case scenarios, and represent a video stream that entirely consists of noise. It shows that this is exactly 8.5 bits per 8 bit original sample, and can be backed up with the fact that Golomb encoded samples have a bit length in the range of 1 to 16 bits, which gives 8.5 as an average value if all internal modeling and thus all output lengths are random.



	port1	port2	port3	room	screen	pp	video	rand
🟦 k = dyn	2.46	3.35	3.54	3.30	1.74	1.38	4.14	8.50

Figure 6.5: *Final compression results from VHDL implementation*

6.5 VHDL Synthesis

Table 6.1: *Synthesis results encoder*

Analysis & Synthesis Resource Usage Summary

Resource	Usage
Estimated Total logic elements	984
Total combinational functions	911
Logic element usage by number of LUT inputs	
– 4 input functions	193
– 3 input functions	269
– <=2 input functions	449
Total registers	588
I/O pins	32
Total memory bits	44040
Maximum fan-out	700
Total fan-out	5372
Average fan-out	3.21

Analysis & Synthesis Resource Utilization by Entity

Hierarchy Node	LC Combinationals	LC Registers	Memory Bits
enc_top	911	588	44040
ctxt_reg	41	0	13312
enc_signal_delays	2	10	64
find_k	122	22	120
gol_enc	124	71	0
index	81	46	0
line_cache	69	49	30544
loco	392	281	0
reset	58	57	0

Max frequency

Slow 1200mV 0C Model Fmax	206.31 MHz
Slow 1200mV 85C Model Fmax	183.89 MHz

All VHDL code was synthesized using *Quartus II 12.1 (Web edition)*, for the device *Cyclone III EP3C8U484C6*.
The context modeling in loco is responsible for most of the LEs consumed by the encoder. Memory bits are used for line caching and the context registers. Frequencies lie well above what is necessary for real-time processing of 1080p 60fps video.

Table 6.2: *Synthesis results decoder*

Analysis & Synthesis Resource Usage Summary

Resource	Usage
Estimated Total logic elements	2383
Total combinational functions	2248
Logic element usage by number of LUT inputs	
– 4 input functions	1262
– 3 input functions	419
– <=2 input functions	567
Total registers	774
I/O pins	29
Total memory bits	43920
Maximum fan-out	866
Total fan-out	10629
Average fan-out	3.35

Analysis & Synthesis Resource Utilization by Entity

Hierarchy Node	LC Combinationals	LC Registers	Memory Bits
dec_top	2248	774	43920
ctxt_reg	41	0	13312
dec_modeling	374	290	0
dec_reset	53	53	0
dec_signal_delays	2	17	64
find_k	101	31	0
gol_dec_top	1517	279	0
comb_qr	32	8	0
data_in_reg	19	48	0
decode_q	229	3	0
decode_r	436	0	0
load_data_in	30	0	0
update_registers	555	0	0
index	81	47	0
line_cache	71	49	30544

Max frequency

Slow 1200mV 0C Model Fmax	169.23 MHz
Slow 1200mV 85C Model Fmax	152.86 MHz

The Decoder's LE resources is mostly spent on decoding the Golomb words, as well as the modeling which again stands out. Frequencies have now fallen, but are still above the threshold.

Worst-case timing paths

Critical paths: Slow 1200mV 85C Model Setup: 'clk'

Table 6.3: *Worst-case timing paths*

From Node	To Node	Data Delay (ns)
Encoder		
modeling|N_t3[0]	modeling|B_temp[5]	5.385
modeling|N_t3[0]	modeling|B_temp[4]	5.385
modeling|N_t3[0]	modeling|B_temp[2]	5.385
modeling|N_t3[3]	modeling|B_temp[5]	5.381
modeling|N_t3[3]	modeling|B_temp[4]	5.381
Decoder		
gol_dec_top|get_in[17]	gol_dec_top|get_in[41]	6.477
gol_dec_top|get_in[3]	gol_dec_top|get_in[47]	6.483
gol_dec_top|get_in[17]	gol_dec_top|get_in[47]	6.455
gol_dec_top|get_in[17]	gol_dec_top|get_in[10]	6.440
gol_dec_top|get_in[39]	gol_dec_top|get_in[41]	6.428

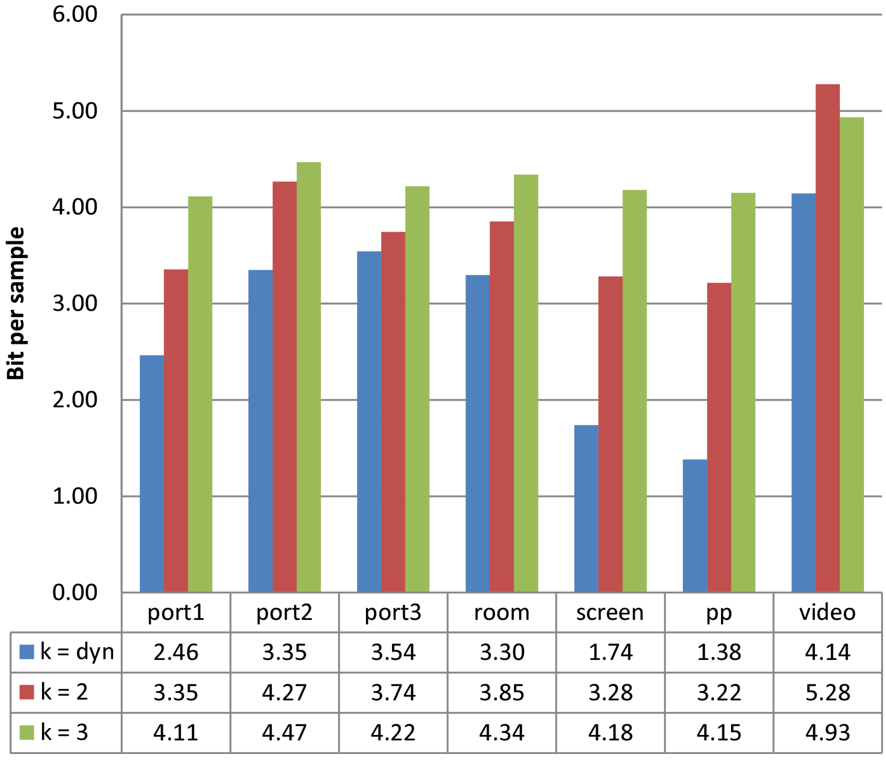
The encoder has its data delay bottleneck in the context modeling, between N_t3 and B_temp. The decoder has its longest data delay between two consecutive get_in shifts.

6.6 Simplified design

This section presents results governing the simplified design presented in section 7.2 under 'Proposed design to reduce resource usage'. Compression ratios for the whole test image set are presented in figure 6.6, and synthesis results in table 6.4 and 6.5 for the encoder and decoder side respectively.

The test image set was run through the VHDL test bench, for $k = 2$ and $k = 3$ which were considered as the most relevant candidates. Results from the dynamic k implementation is included for comparison.

Compression results



	port1	port2	port3	room	screen	pp	video
🟦 k = dyn	2.46	3.35	3.54	3.30	1.74	1.38	4.14
🟥 k = 2	3.35	4.27	3.74	3.85	3.28	3.22	5.28
🟩 k = 3	4.11	4.47	4.22	4.34	4.18	4.15	4.93

Figure 6.6: *Compression results for simplified design*

Synthesis results

Table 6.4: *Synthesis results for simplified design (encoder)*

Analysis & Synthesis Resource Usage Summary

Resource	Usage
Estimated Total logic elements	189
Total combinational functions	144
Logic element usage by number of LUT inputs	
– 4 input functions	37
– 3 input functions	13
– <=2 input functions	94
Total registers	163
I/O pins	32
Total memory bits	15320
Maximum fan-out	171
Total fan-out	934
Average fan-out	2.46

Analysis & Synthesis Resource Utilization by Entity

Hierarchy Node	LC Combinationals	LC Registers	Memory Bits
enc_top	144	163	15320
gol_enc	32	36	0
line_cache	34	11	15320
loco	24	32	0
reset	32	32	0

Max frequency	
Slow 1200mV 0C Model Fmax	379.79 MHz
Slow 1200mV 85C Model Fmax	333.89 MHz

The simplified encoder design trims LEs by dropping the complicating context modeling. Memory bits are used only for the line caching. Frequency rise significantly.

Table 6.5: *Synthesis results for simplified design (decoder)*

Analysis & Synthesis Resource Usage Summary

Resource	Usage
Estimated Total logic elements	1376
Total combinational functions	1287
Logic element usage by number of LUT inputs	
– 4 input functions	835
– 3 input functions	204
– <=2 input functions	248
Total registers	341
I/O pins	33
Total memory bits	15320
Maximum fan-out	349
Total fan-out	5446
Average fan-out	3.20

Analysis & Synthesis Resource Utilization by Entity

Hierarchy Node	LC Combinationals	LC Registers	Memory Bits
dec_top	1287	341	15320
dec_modeling	24	32	0
dec_reset	30	29	0
gol_dec_top	1190	261	0
comb_qr	8	8	0
data_in_reg	19	48	0
decode_q	218	3	0
decode_r	453	0	0
load_data_in	30	0	0
update_registers	317	0	0
line_cache	35	11	15320

Max frequency	
Slow 1200mV 0C Model Fmax	176.46 MHz
Slow 1200mV 85C Model Fmax	160.31 MHz

Again, the context modeling is simplified, reducing the resource consumption, and the Golomb decoding is slightly less consuming because of the fixed k-value. The frequency goes up slightly.

CHAPTER

SEVEN

DISCUSSION

7.1 Changes and considerations regarding the LOCO-I algorithm

Some changes were made to the LOCO-I algorithm in order to be able to implement it in hardware. These changes are mainly a consequence of the frequency requirement, but also a part of the attempt to reduce the implementation size. They include using a different set of variables during the prediction and context determination stage, and those imposed by the pipelining requirement. Considerations regarding maximum Golomb bit lengths, spectral decorrelation of input data and the possibility of bypassing uncompressed data in the memory packages are also discussed.

Effects of pipelining

During the modeling phase, several of the processes defined by the LOCO-I algorithm have to be pipelined on order to comply with the speed requirement. This does not alter the algorithmic steps, and the processing path produces the same internal output as the original algorithm. However, the pipeline delays the updating of the variables used to model the current input sample. Hence, the variables used are not based on data 'as close' to the current pixel as desired. This has several impacts on the compression efficiency. It can be divided in to how well the predictor performs and how correct the offset variable *C* is, based on context.

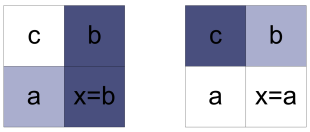
The predictor is based on data stored in the line cache. The predictor in LOCO-I is looking for gradients in the area around current pixel, and need *a* to function properly. As explained in 3.1, a is not applicable as a modeling variable. And because of this an other predictor has to be used.

The pipeline has two impacts on context modeling. First, how relevant the calculated context is to the current pixel. LOCO-I finds the gradients around current pixel in order to classify it in to a context. Since these variables are stored in line cache, the same arguments about not being able to use a as a variable goes here.

Secondly, after the context is determined, the implemented model uses 8 clock cycles from the context information is read from memory until the new context information is written to the same memory (section 4.2, 6 cycles to produce new variables and 2 cycles to write to memory). If 8 pixels of equal context index occurs subsequently, these pixels will be modelled using the same offset variable c, and therefore it will take 8 pixels before the compression is improved in that specific context. In natural images where the context of each pixel changes frequently, this is not expected to have an impact on compression efficiency. But, in computer screen images with long series of pixels occurring in the same context, this could have an impact. Results governing this assumption is discussed further in section 7.2, backed up with results.

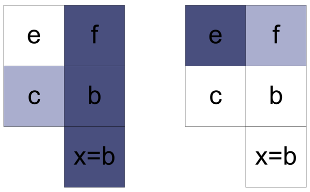
Predictor

As mentioned, the original LOCO-I predictor (equation 2.9) could not be used, because it requires a as a modeling variable. Thus, it was critical to test for other applicable predictors during the initial part of this thesis. The original predictor tried to guess the current pixel x, based on the possible existence of vertical or horizontal gradients around this pixel. This is illustrated in the following two figures,



where the left one represents the function: $x = \max(a,b)$, if $c \leq \min(a,b)$, and the right one represents: $x = \min(a,b)$, if $c \geq \max(a,b)$.

When all modeling variables have to reside on the above lines, the task of finding an appropriate predictor for x is limited, and it can be seen that the LOCO-I predictor approach in finding gradients is no longer reasonable.



Here, the best predictor would probably be $x = b$. Even though there still exists gradients in the area above, it would be difficult to extract this information in order to create a more sophisticated predictor, and is beyond the scope of this thesis.

However, it might be possible that using average values of the variables on the above line b, c and d (referring to the causal template in figure 3.2), could result in a better model than only using b alone. To find out this, the following predictors were tested using the C++ code developed during the preliminary project:

- a
- b
- c
- (a+b)/2
- (b+c)/2
- (a+b+c)/3

The results given in figure 6.1 and 6.2 shows the compression ratios obtained using these predictors for natural images and desktop images respectively. For natural images the difference is not significant, but models including b as a variable, tend to have better compression. And since using only $x = b$ is less complex and requires less resources, this would be the most preferable one.

For the desktop images, $x = b$ gives best results. Thus, this is the prediction model that was used during the rest of the implementation process.

It should be mentioned that during the initial literature study [1], a range of popular image compression algorithms were investigated. This included amongst others, the JPEG-LS [3], FELICS [4], PNG codec [14] and Lossless JPEG [15]. Since these are written as software codecs, with no real-time requirements, a was used as a variable in all codecs. Thus, none of these models could be used in this implementation (except for PNG, which supports the possibility of using $x = b$ as a parameter).

Context determination

The same arguments about the ability to use a as a modeling variable goes here. In the LOCO-I algorithm, the gradients used to calculate the context is $d - b$, $b - c$ and $c - a$, which means that the last gradient could not be used. During the initial testing, $c - a$ was substituted by both 0, and $f - b$, in order to see the impact this had on the compression ratio. By using 0 as a variable means that in practice only two gradients are used. The results are presented in figure 6.1 and 6.2 for natural and desktop images respectively. The results are compared to the case of not correcting the error residual by the context at all. For the natural images the results are not significant, but using 0 or $f - b$ as the gradients generally increases the compressibility, by 0.5% and 1.7% respectively. For the desktop images, using

0 as one of the gradients worsen the compression by 11.9%, while $f - b$ improved the compression by 1.7%.

By including f as a modeling variable, the cache that stores previous samples need to be doubled, since f resides on two lines above the current pixel, which can be seen in the causality template in figure 3.2. Even though not using context at all to correct the error residual does not significantly worsen the compression, the context still has to be determined by some set of gradients, on order to compute k used by the Golomb coder. However, it would be possible to use a fixed k value, and omit the context modeling completely, even though this decreases the compression ratio significantly for desktop images. This scenario is discussed later in section 7.2 under 'Proposed design to reduce resource usage'.

If we assume that context modeling is desirable, the increase in memory usage when f is added as a variable has to be weighed against the decrease in compression for desktop images when f is omitted. Further, if only two gradients are used, the amount of possible contexts is reduced, resulting in a smaller context memory. Referring to equation 3.4 only 5 bits would be necessary in order to address the context memory, instead of 8. This means that information regarding 32 different contexts in stead of 256 has to be stored, resulting in the context memory to be reduced by a factor of 8.

Table 3.1 indicates that 34 bits have to be stored for each context, which further means that 8704 memory bits are needed for 256 contexts and 1088 for 32 contexts. Adding this with the extra cache requirement for storing previous samples (referring to section 3.1 under 'Neighboring pixels'), the reduction in memory bits is significant if only two gradients are used, and is summarized in the following table.

Table 7.1: *Memory usage by context determination*

Gradients	Context mem usage	Cache usage	Total mem bits
3	34 x 256 = 8704	3840 x 8 = 30720	39424
2	34 x 32 = 1088	1921 x 8 = 15368	16456

Maximum Golomb bit length

During the initial review of the implementation it was clear that it was necessary to reduce the maximum possible bit lengths produced by the Golomb encoder. in section 3.3 under the 'Decoder' section it is stated that the width of the data bus output from the Golomb encoder has to equal the longest possible code. And by using an alphabet size of 256, which is the case when the input samples are 8 bits, the worst case bit length would be 256. This would cause the registers needed in both the encoder and decoder to be huge. Also, the critical path when decoding would be too long. Since it can be seen as a sequential bitwise operation. Based on this it is desirable to keep the maximum length as short as possible. The results presented in figure 6.3, shows the impact different lengths have on the compression ratio.

The differences between max lengths between 16 and 256 is negligible, and thus it would be desirable to use the shortest. Based on this observation a maximum length of 16 bits was used during the System C modeling and VHDL implementation phases. Also a bit length of 12 was tested out as a measure to reduce resource usage and critical paths, and will be discussed later in section 7.2.

Possibility to bypass uncompressed data in the memory packages

In order to reduce incidents where series of samples are encoded with more than 8 bits, which will result in negative compression in local areas of the image, a method which involved bypassing the 8 bit original samples instead was investigated. This functionality was implemented during the System C modeling phase, and involved switching between Golomb encoded samples and original samples during the memory package stage (section 3.5). At any time, the memory packer counted the number of encoded samples inside the 128 bit package to be output to memory, and if this was less than 16 samples (128/8), 16 original samples were joined in a package and output to memory instead. This also required the header data to include a bit which told the decoder if this was a compressed or uncompressed package.

This functionality was successfully implemented in System C and resulted in an average improvement in compression by around 2% in natural images and desktop images, and around 6% improvement in noisy images and random data, which is considered as worst case.

The Golomb packer was designed so that data is continuously adding to a packing register, which means that samples are usually split between two Golomb packages (i.e the next Golomb package does not start with a new sample). This caused the decoding process to be significantly more complex if unencoded samples were included. If the decoder starts to read a sample in package 1 and the end of this sample is supposed to be in package 2, this imposes a problem if package 2 consists of unencoded data. This was solved in System C by just omitting the data currently read by the Golomb decoder, pause the process until the uncompressed package was processed, and continue reading the next compressed Golomb package which then would start with a new sample.

In the VHDL implementation it was important to reduce the critical path during the serial processing of samples in order to find the next sample. This is described in section 4.4 Golomb decoder, under the 'Update index registers' procedure. By adding the extra functionality described above, the critical paths became simply too long in order to comply with the frequency requirement. Thus, this feature was not included in the VHDL code.

Still, as a subject for future work, it would be interesting to see if this could be done in another way, which does not impose the need to alter the Golomb decoding process. For instance, an external module could feed the Golomb decoder with trivial data, which matches the number of unencoded samples, and then switch between the Golomb decoded samples and the unencoded samples.

Spectral decorrelation

As stated by the authors of the JPEG-LS codec which is built upon the LOCO-I algorithm [ref side 30 JLS], for some color spaces (e.g. RGB), good decorrelation can be obtain through simple lossless color transformations as a pre-processing step to JPEG-LS. For example, compressing the (R-G, G, B-G) representation of the image ... yields saving between 25% and 30% over compressing the respective RGB representation.

Since this thesis concerns processing RGB data, this representation is implemented and used in the final design. The representation deals with the spectral redundancy occurring in RGB images, and is explained in section 2.2. The processing only includes two subtraction procedures (No modulo operation is necessary in hardware, since the bit lengths are kept to 8 bits, and the final carry bit is omitted), and induces very low additional resource usage compared to the compression improvement. Further, this is done as a pre-processing stage in the encoder, and a post-processing stage in the decoder, and does not have any impact on the overall design, nor does it effect the real time requirement.

7.2 VHDL implementation and synthesis

Golomb decoding

As mentioned, the the biggest challenge during the implementation phase was keeping the critical paths imposed by the bitwise processing of input data, to a minimum.

If the data to be encoded is not split between two or more parallel processes, i.e. only one encoder/decoder pair for each color component, a new variable bit length sample must be processed by the decoder each clock cycle. Consequently, pipelining during the extraction of samples from packed data, is not possible, and the counting procedure must consist of combinational logic without clocked processes, which is able to provide information about the position of next sample at each clock cycle. It would be possible to use several parallel encoder/decoder pairs, but this would result in the resource demand increase by the same factor as the number of parallel processes added, and also potentially decrease the compression ratio. Since resource usage had to be keep to a minimum, this was not considered as a possible solution.

Several solutions were written in VHDL to check for resource usage as well as the critical path lengths. The most intuitive way of implementation is by using shift registers which can shift a variable number of bits (1 - Golomb max length). By first counting the number of pixels in the current pixel, the register could be shifted according to this number, ready to read next pixel from the start of the register. Even though a working solution was written and synthesized, it was not possible to achieve the required speed. Since it requires both counting a maximum of 16 bits and shifting at the same clock period. Also, since the 16 bit input packages is a continuous stream of data, when a new package has to be written to the register, it must be allowed to be written at the end of previous package, which can be at any position within this 16 bit length. Consequently, a large large number of logic elements is needed to both load and shift data inside the registers.

Another solution incorporating a circle of elements, each representing one bit in a total of 3 Golomb packages (48 bits), was written and synthesized. Each element had a 4 bit register where the bits represented the data bit, if it was positioned at the first bit in current sample, if it was at the end of a q sequence and if it was the last bit in that sample. Based on input bits from the previous element and the internal register, it could determine if it was part of a q sequence, or a r sequence. Each clock cycle the information propagated through the 48 elements, and the information was ready to be read by subsequent logic. After synthesis it was clear that the critical path through the 48 elements was too long to comply with the speed requirement.

This solution gave the basis for the final design, but measures had to be taken in order to increase the speed. This included isolating the critical path of each sample which resulted in the $find\_end\_q$ and $find\_end\_r$ modules described in section 4.4. Further, The internal 4 bit register in each element was split into the $data$, *get_index*, *end_q_index* and *next_index* registers.

The synthesis results of the final design is presented in table 6.2. The Golomb decoding module (*gol_dec_top* and its sub-modules) uses 1517 combinational elements and 279 registers, this is 67% and 36% respectively of the total resource usage. Thus, it would be desirable to take measures in order to reduce this demand. A possible way of reducing the size is by having Golomb packages of shorter bit length, this is discussed further in section 7.2. But overall, this would be an important subject for further work.

Throughput

Even though the resource usage of the Golomb decoder is quite significant compared to the other modules in the complete decoder design, it is able to achieve the speeds given by the requirement. The synthesis results gives us a max frequency of 152.86 MHz and 169.23 MHz respectively for 85C and 0C model. Which gives a final throughput of at least $24 \times 152.86 = 3669$ Mbps. As stated by the requirements in section 1.2, a 1080p video has a data rate of 374 MB/s and operates at a frequency of 148.5 MHz.

In table 6.3 the top 5 critical paths for both the encoder and decoder is presented. It shows that the Golomb decoder is the slowest module, and in order to increase the throughput, this module has to be redesigned.

Proposed design to reduce resource usage

As an effort to reduce the resource demand, a simplified version of the design was written and synthesized. In figure 6.1 and 6.2 the effect of correcting the error residual after prediction based on context is presented. The results are discussed in section 7.1 where it was established that the increase in compression ratio was 1.7% for both natural and desktop images when using b, c, d and f as variables during the context determination. This is not a significant improvement, and this correction procedure could be omitted in order to lower the resource usage. However, the same context information is also used to dynamically adjust the k parameter used by the Golomb coder. So by omitting the context part completely the coder is forced to use a fixed k value.

Figure 6.4 shows compression results for different k values compared to the dynamic k variant, for natural images and desktop images. While using dynamic k ensures best compression for both types of images, it is clear that the choice of fixed k value has different impact depending on the image type.

It should be mentioned that the k parameter gives a lower bound on the compressed bit length possible to achieve, referring to equation 2.17 where it is stated that the minimum length would be $k+1$ if q is zero.

Natural images, which consist of much unpredictable information, has an improved compression relative to an increasing k value, as long as the lower bound imposed by k does not exceed the compression potential. For the desktop images used in the test set on the other hand, this potential is already below 2 bits per sample, thus it is desirable to keep k as low as possible.

Since the data used to produce the results is not spectrally decorrelated the compression potential is higher than if this kind of decorrelation was included, And since this only requires a simple pre-processing stage, it is assumed that the input data will be spectrally decorrelated.

The choice regarding which value of k to be used, must be based on a compromise of compression ratios relative to different types of input data. Based on the results obtained for the test set used, and also assuming that data will be spectrally decorrelated, both $k = 1$ and $k = 2$ is potential candidates for further testing.

Alternatively, a variant where k is calculated during a pre-processing stage to the encoder, and used on all pixels in predefined areas of the image, would be possible.

But it would also require k to be included in the header data of that confided area, in order for the decoder to be able to produce the correct output. To keep the header data to a minimum, these areas have to be of a certain size, such as one line of the image. Also computing k on a frame-by-frame basis would probably give satisfactory results, which is the case for the results discussed in this section. The scenario presented in this paragraph has not been a subject for this thesis, and no synthesis results regarding the necessary modules is presented, nor any compression results.

The proposed design was written in VHDL and synthesized in order to get an idea of the resource demand. The encoder has an estimate of 189 logic elements which is only 19% of the original designs 984 logic elements. While the decoder usage is at 1376 logic elements, which is 58% of the 2383 logic elements in the original decoder.

The module that uses most resource is still the Golomb decoder (1190 combinational and 261 registers), because of the high parallelism needed in order to comply with the speed requirement. The resource usage is roughly proportional to the data register size, because the operations can be seen as bitwise parallel processes on all data stored in the register at any time. As mentioned earlier the data register size need to be 3 times as large as the maximum Golomb bit length, in order to avoid buffer underrun.

Consequently, the proposed design discussed above does not reduce the Golomb decoder size significantly, and the only parameter that can reduce this is the maximum Golomb bit length, which has an impact on the data register size needed. Section 6.2 in the results chapter shows compression ratios for different lengths. The original implementation uses 16 as the maximum length, because this seems to give the best overall results. Even though the LOCO-I specification proposes to use bit lengths that are a power of 2 [3], it would still be possible to use 12 as the maximum length, which also would result in the Golomb package size to be 12. However, a more complicated scheme to create memory packages matching power of 2 bit lengths would be necessary.

Figure 6.6 presents the compression results obtained by running the test bench on this implementation instead of the original implementation, for each of the images in the test set. This is done for $k = 2$ and $k = 3$, compared to the original implementation which uses dynamic k. Giving k a value as low as 2 makes the compression more vulnerable to worst case input, but gives better compression when it comes to best case input. A k value of 3, results in a more stable compression ratio of around 4-5 bits per sample.

Verification

When the System C algorithm was mapped over to VHDL, it was hard to achieve the exact same compressed data output, because of all algorithmic steps involved. This would require extensive testing of all VHDL modules, and comparing the data output produced during all steps with the System C model. Since only small variations during the encoding stage, results in total different compressed data, because of the high dependency of previous encoded samples and the nature of how Golomb codes are produced, the validation consisted of comparing the decompressed data with the original input. Also by comparing the compression ratios obtained by both the System C model and the VHDL model, we can get an overview on the correctness of the implemented algorithm.

Both the System C model and the VHDL testbench were written in order for the decoded data to be compared to the input data. Both final designs did not encounter any errors during the processing of the test image set. Each image consist of 6,220,800 8 bit samples, and the video consist of 60 frames resulting in 373,248,000 samples. This gives a total of 410,572,800 samples for all 6 test images and the video, which were encoded and further decoded to yield the exact same data as the original data. This is a strong indication that the decoding process will be valid for any input data, and the implementation is considered as being successfully.

Figure 5.3 show the compression results obtained for the System C model and the VHDL model, compared to the original C++ code developed during the preliminary project, only corrected for the new modeling variables and maximum Golomb bit length discussed in section 7.1. The results varies with a maximum of 6.7% between the System C and VHDL model, which is considered as satisfactory, but indicates that further work governing fine tuning of the algorithmic steps is possible.

Another aspect regarding the comparison of compression ratios between the C++ and the VHDL model, is that it is possible to get a view on the impact the pipelining has on the context modeling part, since the C++ model does not include any pipelining. The similarities in compression indicates that this does not have a severe impact.

7.3 System level conciderations

Control module to reset when new frame arrives

The final design must reset all memory bits regarding the line cache and context memory when a new frame arrives, in order for the encoding process to be valid. Since the memory is implemented using the embedded memory bits on the FPGA, it cannot be reset using a single reset signal, so the initial data must be written to all memory bits during a reset phase.

The working solution receives a 'new frame' notification from the overall system, and respond with a 'ready' signal when all memory is reset. This requires a specific amount of time relative to the number of addresses in the context memory, and the number of elements in the line cache. The final design uses 256 context addresses and 3840 line cache elements. Each write in both of the memories takes one clock cycle to complete, thus the line cache is the bottleneck, resulting in 3840 clock cycles needed in order to reset all memory.

This process might use to many cycles to complete depending on the external data stream. In such a case, measurements have to be taken. This could be solved by including a register which contain valid bits for all memory location. This would require 256 registers for the context memory, but only one for the line cache, because the valid bit does not have to be set until all previous 3840 pixels are shifted in, meaning that the line cache contains two complete lines of original input data. This is left as a subject for further work.

Memory packing

Design of the packer and unpacker modules was not a priority, because their operation would very much depend on the memory space's addressability. Instead we decided to describe a basic interface capable of handling a few different alternatives, and rather discuss some of the possibilities.

The data packets that are written from the encoders must be read by the decoders in a specific order. This means that the three package types (R, G and B) must be recognizable and separable in memory in order to be distributed to the correct decoder module. For data of a predictable size, this would not be difficult. Uncompressed RGB data will take up a known slice of the memory for per frame, the three color components will consist of the same number of bits, and the number of transfers in and out of the DDR memory is also known. When the implemented compression scheme is applied, the resulting frame-to-frame data size becomes variable. The color components' contributions to the memory footprint will vary greatly depending on the input, which complicates the option of storing colors at separate predefined memory locations.

Looking at figure 7.1 a), it shows two uncompressed frames in memory. The colors are shown as separate blocks, but this is only done to better illustrate the result of pre-allocating a slice per color as a means to keeping them separable. Figure 7.1 b) shows the memory footprint when sending compressed data to the same memory slices. Each component can be pulled from memory in a predictable manner, but a considerable fragmentation will be introduced. The fragmented space could be released upon finalization of a frame, but finding an optimal use for the available space may be difficult. Any fragmented sections that stand unusable will negatively affect the final compression rate.

A way of reducing the fragmentation is depicted in figure 7.1 c). Here the components are grouped together so that the next frame can continue where the one before it ended. Free space will be located in front of some write pointer. The zig-zag lines represent longer sections of whatever is above and below them. Red has both an

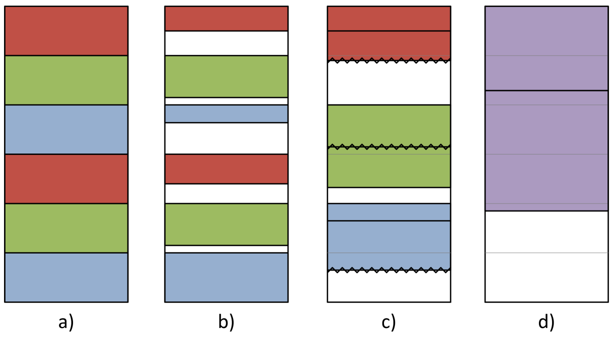


Figure 7.1: *Visualizing different memory allocations*

unspecified amount of data above and free space below it, while green is running out of space and approaching the blue allocated memory. The main apparent issue is the uncertainty of a variable space requirements for components relative to each other.

So far, the colors have been kept separate from each other, but with drawbacks such as fragmentation of the space that is saved. A solution to this is represented in figure 7.1 d). The purple color represents an interlacing of the components, as they are all written to the first available location as they arrive from the encoders. This storage scheme does not introduce fragmentation, but a specific color can no longer be pulled directly from a predefined location.

An issue with this memory structure, is the problems arising in the case of disproportioned compression rate between two color components. When all three components encoders write their memory packets to the same sequential memory location, a relatively even distribution is required when they are read back into the decoding module. If one component has reached high compression, and therefore write few packets to the memory, these packets can be spaced out by packets written from the other component encoders. Those packets must be picked and placed in their respective queues before a high-compressed packet can appear, and the queues must therefore be spacious enough to accommodate a worst case scenario.

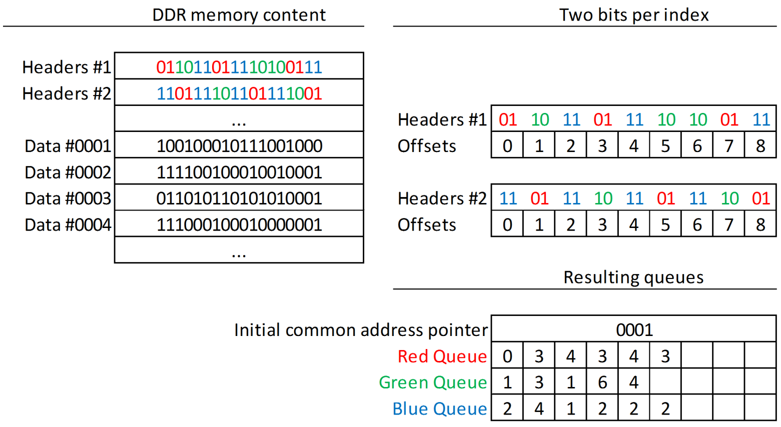
Knowing the best case output of one bit per pixel, we can use the packet size to approximate the worst case distribution in memory. A packet size of 16 would at the very least write a packet every 16 pixels. If the two remaining components concurrently produced the worst case output length of 16 bits per pixel, we see that two times 16 packets could appear in line before the dense packet. From empty, this scenario would require a capacity of 16 elements per FIFO, but they must also have enough packets ready for those long dry spells where all packets belong to the two other components. These runs will be no longer than 32 cycles. A size requirement of two times the Golomb packet size is also in line with what System C simulations show to be necessary in order to prevent underrun in all situations.

Separating the header data

An alternative to the previously described memory structure would be to store the header information separately. One can picture an array of the colors RGB, where the order equates to the sequential order that the data packets hold in memory. This can show advantageous for several reasons. Having no header data as part of the data packets themselves means that their sizes are a multiple of the decoder input. This simplifies the module feeding the decoder, because it eliminates the need to combine the end of a data packet with the front of the next.

The second advantage of storing header data separately can prove very beneficial if implemented efficiently. By reading the header packets ahead of time, we are basically in possession of address offsets for where each color component packet is stored in memory. In the simplest case one can use the information to decide whether the next packet can be read from memory and sent to a FIFO, or if this packet would hit a FIFO at its maximum capacity. Say if the red FIFO is full, while the two other colors are running low, the knowledge that the next data packet is blue and not red, can be used to reduce the risk of underruns.

The next step would be to fully utilize the information that the headers provide, and use the offsets to fetch the exact color component that is needed. Instead of storing rather large buffered data packets in FIFOs, one could store a single dynamic address pointer per color component. Three queues would hold the distances between one color's data packet and the next. The necessary data packet can then be found by grabbing a value from the queue and adding it to the address pointer.



Headers #1	011011011110100111
Headers #2	110111101101111001
	...
Data #0001	100100010111001000
Data #0002	111100100010010001
Data #0003	011010110101010001
Data #0004	111000100010000001
	...

Headers #1	01	10	11	01	11	10	10	01	11
Offsets	0	1	2	3	4	5	6	7	8

Headers #2	11	01	11	10	11	01	11	10	01
Offsets	0	1	2	3	4	5	6	7	8

Initial common address pointer	0001								
Red Queue	0	3	4	3	4	3			
Green Queue	1	3	1	6	4				
Blue Queue	2	4	1	2	2	2			

Figure 7.2: *Three packet queues built from header data*

Overhead data

Because the write buffer behind the encoder collects the variable length encoded words and produces full packages for storage, there is no need for any extra length information to be added to every package. Still, there might be some overhead data created by the encoder. The amount depends on how the compressed data packets can be stored and retrieved from memory. With three addressable memory sections, each holding only one color component's data, there is no need to store anything extra. If separate sections are infeasible, a vector of flags identifying a packet's color must be added to the post-compression size. The size of this overhead data will naturally depend on the number of packets produced by the encoders, as well as the size of the packets. Each packet adds 2 flag bits, so the total additional data per frame can be found to be a constant percentage of the compressed image data size. Percentages added to the data size is listed in table 7.2.

Table 7.2: *Size increase from header data*

Bits per package	Percentage increase
16	12.50 %
32	6.25 %
64	3.13 %
128	1.56 %
256	0.78 %

7.4 Design limitations

Bus traffic

To the lossy video encoder, the DDR contents (image frames) should look identical with and without the compression scheme. Depending on the interconnect topology of the system where this compression scheme is to be integrated, a common interconnection bus could potentially see a dramatic increase in traffic. Figure 7.3 is an example of how a system without any compression steps in front of its DDR memory might look. The data only touches the common bus once before it is cached in memory. The data can then be accessed by a module such as the lossy video encoder.

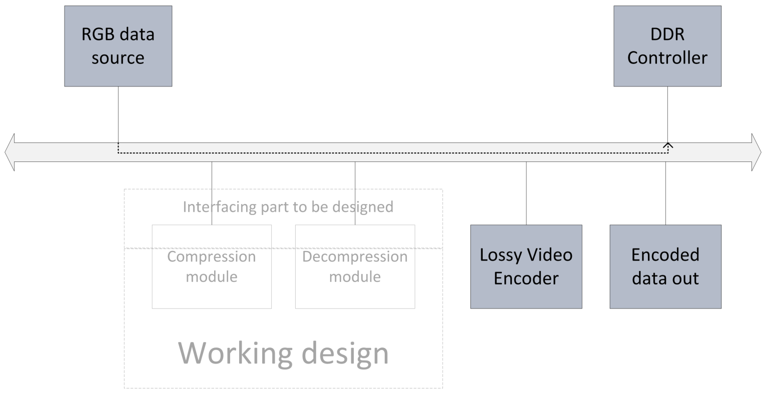


Figure 7.3: *Bus topology showing number of data transfers between RGB source and external video encoding module* ***without*** *the presented compression scheme*

If the compression step is added, the number of bus transactions in this configuration goes up. In figure 7.4 each transaction is labelled in order from 1 to 4. The RGB data only appears in memory after the fourth transaction.

1. RGB data to the compression module (encoding)
2. Encoded data packets to the DDR memory
3. Encoded packets from memory to the decompression module
4. Decoded RGB data to memory

Summing all the bus traffic T per transfer i gives a figure for the total traffic $\sum T_i$

$$\sum_{i=1}^{4} T_i = T + T_c + T_c + T = 2T_c + 2T \tag{7.1}$$

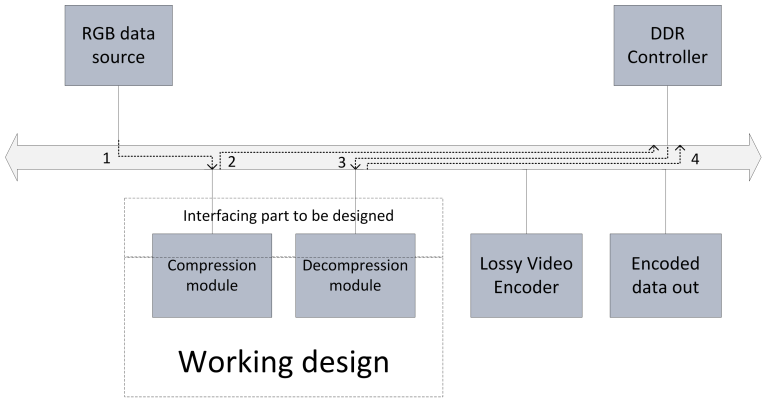


Figure 7.4: *Bus topology showing number of data transfers between RGB source and external video encoding module* ***with*** *the presented compression scheme*

where T_c is compressed at a rate c, whereas the compressionless configuration only had a $\sum T_i = T$. It is apparent that this configuration does not serve to reduce any bottlenecking through the memory bus either. The first transaction from the RGB source to the compression module can be moved off of the common bus. Any other data originating from the same source would either have to travel through the compression module via a by-pass, or the source must have a separate connection to the common bus. With this alteration, the sum of transactions are lowered to $\sum T_i = 2T_c + T$.

This still leaves two transactions that increase traffic, so where can the decompression module be placed to avoid them? Our compression scheme creates encoded independent frames, but the compressed data at random points within one frame is very much dependent on all of the data before it. If the decompression module is placed in front of the lossy video encoder, it could sequentially read a whole frame. It would then have to either feed the decompressed frame back to a memory location, or it would have to decompress the same frame every time it is needed - even a small section of it. Any module that requires relatively random access to sections of an image frame would have limited use of such a compression scheme. The bus transactions per frame is technically lowered from $T + n \cdot T$ to $T_c + n \cdot T_c$, n being the number of accesses to the data after it has been compressed in memory. This reduction comes at the cost of not having independent access to specific sections within a frame.

Decoding speed bottleneck

Another issue is the fact that the proposed decoder delivers only one pixel per clock cycle. This will amount to around $440MB/s$ worth of uncompressed RGB data based on a clock frequency of $148,5MHz$. That is far from the potential bandwidths of newer generations of DDR memory, which lie in the thousands of MB/s.

The restricted intra-frame access

The presented implementation delivers great lossless compression, but the causal encoding means that the whole frame must be decoded every time a section is read. This issue can be addressed by creating starting points at certain stages in a frame. The compression restarts at these points, cutting off one causal sequence and starting another. The decoder can then begin decoding from any of the starting points. Basic tests have shown that the compression rate is good even with a few 1920 pixels wide lines. The encoder requires few changes except for the more frequent restarts. Additional meta data must most likely be created in order to point to where specific frame sections are kept within a frame's allocated memory space.

The decoder throughput bottleneck

Adding intra-frame starting points does not alone increase the decoder throughput. It does however create an opportunity for parallelizing the decoding process. With independent starting points in the compressed data, one can increase the decoding throughput by running several decoders in parallel. Only a single encoder is needed still. Running four decoders in parallel with starting points at every 4 lines mean that 16 lines of data can be decompressed at around $4 \times 440\ {}^{MB}\!/_{s} = 1760\ MB/s$. This parallelization comes at the cost of higher FPGA resource usage, with the exact amount depending on which algorithm configurations are used (e.g. adaptive or fixed k).

A different solution

The DDPCM+GR scheme outlined in [2] has two main strengths over our implementation. One is its high level of parallelism, giving it the ability to raise throughput in exchange for increased area usage. The second strength is its low worst case, achieved by storing uncompressed values in the cases where the compression algorithm would have increased the size. The proposed primary solution has a theoretical worst case of 16-18 ${}^{b}\!/_{s}$, although the highest observed is 8.5 ${}^{b}\!/_{s}$, found using full spectrum pseudo-random noise.

It is not made clear how DDPCM+GR stores its packages in memory; more specifically, what is done with the bit positions that are left after the variable unary codes have all been appended. Based on their need to store the length and their testing with different bus bandwidths, one can assume that there is a rounding involved at either 8, 16 or 32 bit intervals. This practice would certainly reduce the average compression. It also seems like they could have cut 1 bit off their constant data by dropping the last terminating zero in the unary data. This would lower the best case package size to 75 bits. The length value can also be reduced by one bit, because the minimum and maximum packet size does not span more than 64 values. These two changes would have reduced their maximum and average compression.

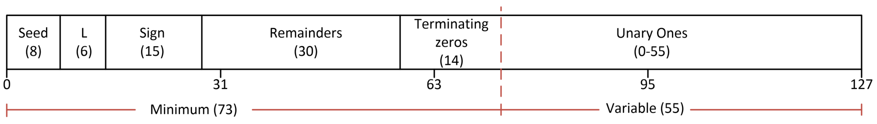


Figure 7.5: *Adjusted packing scheme. The minimum value represents the very least amount of data included in every package. The variable part allows for up to 55 bits in the unary part.*

It is possible that they made the decision to sacrifice these two bits because the alternative in figure 7.5 introduced at least one addition operation; finding the actual package length from $L + minimum$ instead of using the value of L directly. It could also have impacted their decoding in a way that required additional logic for deducing the location of the missing terminating zero. Seeing how high parallelism is an important characteristic of their design, they may valued the transistors saved more because the extra space would be factored into every single instance of their module.

7.5 Comparison of implementations

Compression

The primary design's best feature is clearly its average compression rate. Compared to the simplified design it delivers on average 14% better compression on the images in the test set. The simplified design's results depend on the choice of k-value. A high k locks the best case compression at a high number, as the best case can never be lower than $k+1$ bits per symbol. Still, a higher k is necessary to mitigate a large prediction error. The middle ground that seem to yield the best results on average is a constant value of $k=2$. The observed worst case with adaptive k is only 8.5, while a fixed k give 14,9 $^{b}/_{s}$ on pseudo-random noise.

DDPCM+GR has a best case compression rate of 4.75, which is far higher than the proposed implementations. It also has control of the worst cases, as it stores original uncompressed data instead if the compressed data turns out to be larger. Further comparison require an implementation of the design in order to run tests on the same images.

Table 7.3: *Average, worst and best case compression for our proposed primary, simplified design, and DDPCM+GR [2]*

	Primary solution	**Simplified solution**			**DDPCM+GR**
	Adaptive k	k=1	k=2	k=3	
Avg compr. [$^{b}/_{s}$]	2,74	3,88	3,86	4,34	-
Worst case [$^{b}/_{s}$]	8,50	15,40	14,90	14,00	8,00
Best case [$^{b}/_{s}$]	1,00	2,00	3,00	4,00	4,75

Resource usage

The simplified solutions uses fewer FPGA resources than the primary solution. Table 7.4 shows Logic Elements per instance, total, and LE per compression. The last figure is calculated with the following formula $\frac{LEtotal}{8-Avg.comp.}$, and reflects which solution achieves its compression most (resource) efficiently.

It is difficult to compare the resource consumption of the non-FPGA DDPCM+GR design directly, because they use Gate Equivalents as their unit of measurement.

Table 7.4: *Logic Element counts for the primary and simplified solution.*

	Primary solution		**Simplified solution**		**DDPCM+GR**
	Encoder	Decoder	Encoder	Decoder	
LE	984	2383	189	1376	-
LE total	10100		4695		16000 GEs
LE/Compression	1920		1134		-

Throughput

The total throughput for each of the two implementations is equal to three times the clock frequency of the instances, as each instance will process one 8 bit symbol every clock cycle. The simplified solution therefore has a slightly higher throughput because of its slightly higher maximum frequency. The numbers are summarized in table 7.5. It is clear that the *throughput per resource usage* is doubled by using the simplified design.

Note that the proposed implementations run one instance per RGB component, while DDPCM+GR uses 8 bit YUV as its input and is running 16 instances in parallel.

Table 7.5: *Throughput figures for the primary and simplified solution, and DDPCM+GR*

	Primary solution	**Simplified solution**	**DDPCM+GR**
Max Freq.	150 MHz	160 MHz	127 MHz *
Throughput	450 MB/s	480 MB/s	2720 MB/s
Parallelism	1	1	16
Throughput/Parallelism	450 $^{MB}/_{s}$	480 $^{MB}/_{s}$	170 $^{MB}/_{s}$
Throughput/LE	44,5 $\frac{kB/s}{LE}$	102,2 $\frac{kB/s}{LE}$	-
	*ASIC 0,15μm technology.		

CHAPTER

EIGHT

CONCLUSION

A lossless image compression scheme has been implemented in System C and VHDL in order to gauge its FPGA resource costs and operating frequency. The design is a compression module and decompression module, representing the core of a scheme, where added control circuitry adhering to the integrating systems requirements must be added in order to properly pack the compressed frame data in memory.

One compression module capable of compressing the eight bits of one component (R, G, or B) uses 984 Logic Elements, running at a frequency of 183,89 MHz. The decompression module requires 2383 LE, while operating at a frequency of 152,86 MHz. Adding up the total requirements for an RGB solution, the compression requires 2952 LE and the decompression requires 7149 LE.

Adjustments can be made with relatively little loss of compression rate, in order to reduce the resource costs. The adjustments involve dropping the context modeling, and using a constant k-value for the Golomb encoding. Both modules see a drastic decrease in FPGA resource usage if these changes are implemented. The resources requirement of the compression reduced by 80%, while the decompression module's requirements are reduced by 42%. The compression module's operating frequency nearly doubles to 333 MHz, and the decompression module rises to 160,31 MHz.

These figures does not include the FIFOs required to buffer compressed data at the decoder inputs. System C modeling showed that 32 Golomb packets must be speedily available to the decoder in order to counteract under-run, which would require the decompression module to pause operation. FIFOs at the encoder and decoder outputs are mostly depending on the integrating system's need for backpressure reduction due to high bus loads (busy bus).

Table 8.1: *Resource, frequency and compression rate summary*

	Primary solution		**Simplified solution**	
	Encoder	Decoder	Encoder	Decoder
LE				
·*Individual*	984	2383	189	1376
·*Total*	10101		4695	
Compression				
·*Natural*	3,27 b/S		4,10 b/S	
·*Screen*	1,43 b/S		3,25 b/S	
Speed				
·*Frequency*	189,89 MHz	152,86 MHz	333 MHz	160,31 MHz
·*Throughput*	569 MB/s	458 MB/s	999 MB/s	480 MB/s

Restrictions

The design has some restrictive properties that may prove to be problematic for an integrating system. Encoding results in a causal sequence of compressed data, meaning that the frame must be decompressed from start to end every time it is accessed from memory. This differs from the compressionless alternative, where any addressable section of the frame can be read from memory at any time. The proposed decoder does also have a low throughput when compared to the theoretical memory bandwidths on modern DDR architectures.

Letting the encoder restart its compression at pre-defined intervals, e.g. every 8 lines, will let an decoder choose to extract sections of image data instead of the whole image. This adjustments comes at the cost of reduced compression rate and an increased amount of meta data per frame, necessary for referencing the specific locations where image frame sections are found. When sections of the image are independently stored, it also allows for a parallelization of the decoding process. Multiple modules can decompress different sections of the image at the same time, increasing the theoretical throughput to $n \times DecoderThroughput$, at a cost of increased resource usage.

Final words

The study has shown that it is possible to implement compression schemes in VHDL that successfully reduces the memory bandwidth usage. The proposed implementations reach throughputs necessary to compress $1920 \times 1080 \times 60$ fps video in real time. If the primary design uses too many resources, the LE count can be reduced by using a simplified solution, without losing too much of the compression benefit. An implementation's usefulness depends greatly on how an integrating system expects to access the compressed image information. The presented design can be adapted to increase its usefulness, with specific use cases in mind.

Future work

- Work towards a specific system / use case. Adjust the design depending on exactly how compressed data must be accessible.
- Finalize the packing and unpacking depending on said system's specifications.
- Include better reset functionality when new frame arrives.
- Reduce Golomb decoder resource demand.
- Include functionality to bypass unencoded data, in order to reduce worst case scenarios.
- Further investigate the simplified design, and possibilities to compute k on a frame-by-frame basis.
- Check for algorithmic error resulting from the mapping of VHDL code from C code.
- Run tests on an FPGA.

APPENDIX

A

VHDL CODE

A.1 Find ’k’ and determine context

find_k.vhd

```
entity find_k is
  port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    A          : in  std_logic_vector(A_LEN - 1 downto 0);
    N          : in  std_logic_vector(N_LEN - 1 downto 0);
    k_out      : out std_logic_vector(8 downto 0);
    k_less_out : out std_logic_vector(8 downto 0);
    k_int_out  : out std_logic_vector(3 downto 0)
  );
end entity find_k;

architecture RTL of find_k is
  signal N_ctxt : std_logic_vector(N_LEN - 1 downto 0);
  signal A_ctxt : std_logic_vector(A_LEN - 1 downto 0);
begin
  comp_k : process(clk, rst) is
  begin
    if rising_edge(clk) then
      if rst = ’1’ then
        k_out      <= "100000000";
        k_less_out <= "011111111";
        k_int_out  <= "1000";
        A_ctxt     <= (others => ’1’);
        N_ctxt     <= (others => ’0’);

      --Select k based on A and N:
        else
          A_ctxt <= A;
          N_ctxt <= N;
          -- k = 0
          if "00000000" & N_ctxt >= A_ctxt then
            k_out      <= "000000001";
            k_less_out <= "000000000";
            k_int_out  <= "0000";
          -- k = 1
          elsif "0000000" & N_ctxt & "0" > A_ctxt then
            k_out      <= "000000010";
            k_less_out <= "000000001";
            k_int_out  <= "0001";
          -- k = 2
          elsif "000000" & N_ctxt & "00" > A_ctxt then
            k_out      <= "000000100";
            k_less_out <= "000000011";
            k_int_out  <= "0010";
          -- k = 3
          elsif "00000" & N_ctxt & "000" > A_ctxt then
            k_out      <= "000001000";
            k_less_out <= "000000111";
            k_int_out  <= "0011";
          -- k = 4
          elsif "0000" & N_ctxt & "0000" > A_ctxt then
            k_out      <= "000010000";
            k_less_out <= "000001111";
            k_int_out  <= "0100";
          -- k = 5
          elsif "000" & N_ctxt & "00000" > A_ctxt then
            k_out      <= "000100000";
            k_less_out <= "000011111";
            k_int_out  <= "0101";
          -- k = 6
          elsif "00" & N_ctxt & "000000" > A_ctxt then
            k_out      <= "001000000";
            k_less_out <= "000111111";
            k_int_out  <= "0110";
          -- k = 7
          elsif "0" & N_ctxt & "0000000" > A_ctxt then
            k_out      <= "010000000";
            k_less_out <= "001111111";
            k_int_out  <= "0111";
          -- k = 8
          else
            k_out      <= "100000000";
            k_less_out <= "011111111";
            k_int_out  <= "1000";
          end if;
        end if;
      end if;
    end process comp_k;
  end architecture RTL;
```

index.vhd

```
entity index is
  port(
    clk       : in  std_logic;
    rst       : in  std_logic;
    B_cache   : in  std_logic_vector(SMPL_LEN - 1 downto 0);
    C_cache   : in  std_logic_vector(SMPL_LEN - 1 downto 0);
    D_cache   : in  std_logic_vector(SMPL_LEN - 1 downto 0);
    I_cache   : in  std_logic_vector(SMPL_LEN - 1 downto 0);
    index_out : out std_logic_vector(7 downto 0);
    sign_out  : out std_logic
  );
end entity index;
architecture RTL of index is
  type g_arr is array (0 to 2)
    of std_logic_vector(SMPL_LEN - 1 downto 0);
  type q_arr is array (0 to 2) of std_logic_vector(2 downto 0);
  signal g          : g_arr;
  signal q          : q_arr;
  signal sign_t1    : std_logic;
  signal test_count : std_logic_vector(7 downto 0);
  signal qn         : q_arr;
begin
  -- compute gradient based on neighbouring pixels:
  gradient : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        g(0) <= (others => '0');
        g(1) <= (others => '0');
        g(2) <= (others => '0');
      else
        g(0) <= D_cache - B_cache;
        g(1) <= B_cache - C_cache;
        g(2) <= I_cache - B_cache;
      end if;
    end if;
  end process gradient;

  -- quantize the gradients:
  quantize : process(clk) is
  begin
    if rising_edge(clk) then
      sign_t1 <= '1';
      for i in 0 to 2 loop
        if (g(i) = "00000000") then       -- g = 0
          q(i)  <= "000";
          qn(i) <= "000";
        elsif (g(i) <= "00000010") then   -- 0 > g >= 2
          q(i)  <= "001";
          qn(i) <= "111";
        elsif (g(i) <= "00000110") then   -- 2 > g >= 6
          q(i)  <= "010";
          qn(i) <= "110";
        elsif (g(i) <= "01111111") then   -- 6 > g >= 127
```

```
          q(i)  <= "011";
          qn(i) <= "101";
        elsif (g(i) <= "11101011") then   -- -128 <= g <= -21
          q(i)  <= "101";
          qn(i) <= "011";
          if i = 0 then
            sign_t1 <= ’0’;
          end if;
        elsif (g(i) <= "11111001") then   -- -21 < g <= -7
          q(i)  <= "101";
          qn(i) <= "011";
          if i = 0 then                   -- Set sign bit
            sign_t1 <= ’0’;
          end if;
        elsif (g(i) <= "11111101") then   -- -7 < g <= -3
          q(i)  <= "110";
          qn(i) <= "010";
          if i = 0 then                   -- Set sign bit
            sign_t1 <= ’0’;
          end if;
        elsif (g(i) <= "11111111") then   -- -3 < g <= -1
          q(i)  <= "111";
          qn(i) <= "001";
          if i = 0 then                   -- Set sign bit
            sign_t1 <= ’0’;
          end if;
        end if;
      end loop;
    end if;
  end process quantize;

  -- Combine quantized regions:
  index : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = ’1’ then
        index_out  <= (others => ’1’);
        sign_out   <= ’0’;
        test_count <= "00000001";
      else
        if (sign_t1 = ’0’) then
          index_out  <= qn(0)(1 downto 0) & qn(1) & qn(2);
          test_count <= test_count + 1;
        else
          index_out <= q(0)(1 downto 0) & q(1) & q(2);
        end if;
        sign_out <= sign_t1;
      end if;
    end if;
  end process index;
end architecture RTL;
```

A.2 Modeling and signals - encoder

enc_modeling.vhd

```
entity loco is
  port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    data       : in  std_logic_vector(SMPL_LEN - 1 downto 0);
    loco_w_rst : in  std_logic;

    sign       : in  std_logic;
    P          : in  std_logic_vector(SMPL_LEN - 1 downto 0);

    A          : in  std_logic_vector(A_LEN - 1 downto 0);
    B          : in  std_logic_vector(B_LEN - 1 downto 0);
    C          : in  std_logic_vector(C_LEN - 1 downto 0);
    N          : in  std_logic_vector(N_LEN - 1 downto 0);

    A_out      : out std_logic_vector(A_LEN - 1 downto 0);
    B_out      : out std_logic_vector(B_LEN - 1 downto 0);
    C_out      : out std_logic_vector(C_LEN - 1 downto 0);
    N_out      : out std_logic_vector(N_LEN - 1 downto 0);

    B0_out     : out std_logic_vector(B_LEN - 1 downto 0);
    N0_out     : out std_logic_vector(N_LEN - 1 downto 0);
    Q_out      : out std_logic_vector(SMPL_LEN - 1 downto 0)
  );
end entity loco;

architecture RTL of loco is
  signal e       : std_logic_vector(C_LEN downto 0);
  signal A_t1    : std_logic_vector(A_LEN - 1 downto 0);
  signal B_t1    : std_logic_vector(C_LEN downto 0);
  signal C_t1    : std_logic_vector(C_LEN downto 0);
  signal N_t1    : std_logic_vector(C_LEN downto 0);
  signal A_t2    : std_logic_vector(A_LEN - 1 downto 0);
  signal B_t2    : std_logic_vector(C_LEN downto 0);
  signal C_t2    : std_logic_vector(C_LEN downto 0);
  signal N_t2    : std_logic_vector(C_LEN downto 0);
  signal A_t3    : std_logic_vector(A_LEN - 1 downto 0);
  signal B_t3    : std_logic_vector(C_LEN downto 0);
  signal C_t3    : std_logic_vector(C_LEN downto 0);
  signal N_t3    : std_logic_vector(C_LEN downto 0);
  signal e_abs   : std_logic_vector(C_LEN downto 0);
  signal A_t0    : std_logic_vector(A_LEN - 1 downto 0);
  signal B_t0    : std_logic_vector(B_LEN - 1 downto 0);
  signal C_t0    : std_logic_vector(C_LEN - 1 downto 0);
  signal N_t0    : std_logic_vector(N_LEN - 1 downto 0);
  signal P_t0    : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal A_temp  : std_logic_vector(A_LEN - 1 downto 0);
  signal B_temp  : std_logic_vector(C_LEN downto 0);
  signal C_temp  : std_logic_vector(C_LEN downto 0);
  signal N_temp  : std_logic_vector(C_LEN downto 0);
  signal B_t2t1  : std_logic_vector(C_LEN downto 0);
```

```
    signal B_t2t2  : std_logic_vector(C_LEN downto 0);
    signal C_t2t1  : std_logic_vector(C_LEN downto 0);
    signal C_t2t2  : std_logic_vector(C_LEN downto 0);
    signal B0_out1 : std_logic_vector(B_LEN - 1 downto 0);
    signal N0_out1 : std_logic_vector(N_LEN - 1 downto 0);
    signal B0_out2 : std_logic_vector(B_LEN - 1 downto 0);
    signal N0_out2 : std_logic_vector(N_LEN - 1 downto 0);
    signal data_t0 : std_logic_vector(SMPL_LEN - 1 downto 0);
    signal sign_t0 : std_logic;

  begin
    adj_e : process(clk, rst) is
      variable temp_e1 : std_logic_vector(C_LEN downto 0);
      variable temp_e2 : std_logic_vector(C_LEN downto 0);
    begin
      if rising_edge(clk) then
        if rst = '1' then
          e        <= (others => '0');
          e_abs    <= (others => '0');
          A_t1     <= (others => '0');
          B_t1     <= (others => '0');
          C_t1     <= (others => '0');
          N_t1     <= (others => '0');
          A_t0     <= (others => '0');
          B_t0     <= (others => '0');
          C_t0     <= (others => '0');
          N_t0     <= (others => '0');
          P_t0     <= (others => '0');
          B0_out   <= (others => '0');
          N0_out   <= (others => '0');
          B0_out1  <= (others => '0');
          N0_out1  <= (others => '0');
          B0_out2  <= (others => '0');
          N0_out2  <= (others => '0');
          data_t0  <= (others => '0');
          sign_t0  <= '0';
        else
          -- Pipeline registers:
          A_t0     <= A;
          B_t0     <= B;
          C_t0     <= C;
          N_t0     <= N;
          P_t0     <= P;
          B0_out1  <= B;
          N0_out1  <= N;
          B0_out2  <= B0_out1;
          N0_out2  <= N0_out1;
          B0_out   <= B0_out2;
          N0_out   <= N0_out2;
          data_t0  <= data;
          sign_t0  <= sign;

          -- Find error residual for sign = 1:
          -- Using input data, predictor (P) and context (C)
          if (sign_t0 = '1') then
            temp_e1 := (data_t0(7) & data_t0(7) & data_t0)
                     - (P_t0(7) & P_t0(7) & P_t0) - (C_t0(8) & C_t0);
        if temp_e1(7) = '1' then
          e_abs <= 0 - (temp_e1(7) & temp_e1(7)
                                  & temp_e1(7 downto 0)
          );
        else
          e_abs <= temp_e1(7) & temp_e1(7) & temp_e1(7 downto 0);
        end if;
        e <= temp_e1(7) & temp_e1(7) & temp_e1(7 downto 0);

      -- Find error residual for sign = 0:
      else
        temp_e2 := (P_t0(7) & P_t0(7) & P_t0)
         - (data_t0(7) & data_t0(7) & data_t0) - (C_t0(8) & C_t0);
        if temp_e2(7) = '1' then
          e_abs <= 0 - (temp_e2(7) & temp_e2(7)
                                  & temp_e2(7 downto 0)
          );
        else
          e_abs <= temp_e2(7) & temp_e2(7) & temp_e2(7 downto 0);
        end if;
        e <= temp_e2(7) & temp_e2(7) & temp_e2(7 downto 0);
      end if;

      -- Half B,C,N if N >= 32, and extend bit lengths:
      if N_t0(5) = '0' then
        A_t1 <= A_t0;
        B_t1 <= B_t0(5) & B_t0(5) & B_t0(5) & B_t0(5) & B_t0;
        C_t1 <= C_t0(8) & C_t0;
        N_t1 <= "0000" & N_t0;
      else
        A_t1 <= A_t0(A_LEN - 1) & A_t0(A_LEN - 1 downto 1);
        B_t1 <= B_t0(5) & B_t0(5) & B_t0(5) & B_t0(5)
                        & B_t0(5) & B_t0(B_LEN - 1 downto 1);
        C_t1 <= C_t0(8) & C_t0;
        N_t1 <= "0000001111";
      end if;
    end if;
  end if;
end process adj_e;

-- Context modeling procedure:
modeling : process(clk, rst) is
begin
  if rising_edge(clk) then
    if loco_w_rst = '1' then
      -- Reset pipelined registers:
      A_out  <= (others => '0');
      B_out  <= (others => '0');
      C_out  <= (others => '0');
      N_out  <= (others => '0');
      A_t2   <= (others => '0');
      B_t2   <= (others => '0');
      C_t2   <= (others => '0');
      N_t2   <= (others => '0');
      A_t3   <= (others => '0');
      B_t3   <= (others => '0');
      C_t3   <= (others => '0');
```

```
        N_t3   <= (others => '0');
        B_t2t1 <= (others => '0');
        B_t2t2 <= (others => '0');
        C_t2t1 <= (others => '0');
        C_t2t2 <= (others => '0');
        A_temp <= (others => '0');
        B_temp <= (others => '0');
        C_temp <= (others => '0');
        N_temp <= (others => '0');
      else
        A_t2   <= A_t1 + e_abs;
        B_t2   <= B_t1 + e;
        C_t2   <= C_t1;
        N_t2   <= N_t1 + 1;
        B_t2t1 <= B_t1 + e + N_t1 + 1;
        B_t2t2 <= B_t1 + e - N_t1 - 1;
        C_t2t1 <= C_t1 - 1;
        C_t2t2 <= C_t1 + 1;

        if ((B_t2 <= "0" - N_t2) and B_t2(C_LEN) = '1') then
          A_t3 <= A_t2;
          B_t3 <= B_t2t1;
          C_t3 <= C_t2t1;
          N_t3 <= N_t2;
          if ((B_t3 <= 0 - N_t3) and B_t3(C_LEN) = '1') then
            A_temp <= A_t3;
            B_temp <= 0 - N_t3 + 1;
            C_temp <= C_t3;
            N_temp <= N_t3;
          else
            A_temp <= A_t3;
            B_temp <= B_t3;
            C_temp <= C_t3;
            N_temp <= N_t3;
          end if;
        elsif (B_t2 > 0 and B_t2(C_LEN) = '0') then
          A_t3 <= A_t2;
          B_t3 <= B_t2t2;
          C_t3 <= C_t2t2;
          N_t3 <= N_t2;
          if ((B_t3 <= 0 - N_t3) and B_t3(C_LEN) = '1') then
            A_temp <= A_t3;
            B_temp <= (others => '0');
            C_temp <= C_t3;
            N_temp <= N_t3;
          else
            A_temp <= A_t3;
            B_temp <= B_t3;
            C_temp <= C_t3;
            N_temp <= N_t3;
          end if;
        else
          A_t3   <= A_t2;
          B_t3   <= B_t2;
          C_t3   <= C_t2;
          N_t3   <= N_t2;
          A_temp <= A_t3;
          B_temp <= B_t3;
          C_temp <= C_t3;
          N_temp <= N_t3;
        end if;
        -- Output new computed A,B,C,N
        A_out <= A_temp;
        B_out <= B_temp(B_LEN - 1 downto 0);
        C_out <= C_temp(C_LEN - 1 downto 0);
        N_out <= N_temp(N_LEN - 1 downto 0);
      end if;
    end if;
  end process modeling;

  -- Output error residual:
  update_output : process(clk, rst) is
  begin
    if rst = '1' then
      Q_out <= (others => '0');
    elsif rising_edge(clk) then
      Q_out <= e(SMPL_LEN - 1 downto 0);
    end if;
  end process update_output;
end architecture RTL;
```

enc_signal_delays.vhd

```
entity enc_signal_delays is
  port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    index_in   : in  std_logic_vector(7 downto 0);
    sign_in    : in  std_logic;
    k_less_in  : in  std_logic_vector(8 downto 0);
    k_in       : in  std_logic_vector(8 downto 0);
    k_int_in   : in  std_logic_vector(3 downto 0);

    index_out1 : out std_logic_vector(7 downto 0);
    index_out2 : out std_logic_vector(7 downto 0);
    sign_out   : out std_logic;
    k_less_out : out std_logic_vector(8 downto 0);
    k_out      : out std_logic_vector(8 downto 0);
    k_int_out  : out std_logic_vector(3 downto 0)
  );
end entity enc_signal_delays;

architecture RTL of enc_signal_delays is
  type k_array is array (ENC_K_DELAY downto 0)
    of std_logic_vector(8 downto 0);
  type k_less_array is array (ENC_K_DELAY downto 0)
    of std_logic_vector(8 downto 0);
  type k_int_array is array (ENC_K_DELAY downto 0)
    of std_logic_vector(3 downto 0);
  type index_array is array (ENC_INDEX_DELAY downto 0)
    of std_logic_vector(7 downto 0);
  type sign_array is array (ENC_SIGN_DELAY downto 0)
    of std_logic;

  -- k
  signal k      : k_array;
  signal k_less : k_less_array;
  signal k_int  : k_int_array;

  -- index
  signal index : index_array;
  signal sign  : sign_array;
begin
  propagate_signals : process(clk, rst) is
  begin

    if rising_edge(clk) then

      -- Propagate signals through registers:
      index(ENC_INDEX_DELAY downto 1)
                        <= index(ENC_INDEX_DELAY - 1 downto 0);
      sign(ENC_SIGN_DELAY downto 1)
                        <= sign(ENC_SIGN_DELAY - 1 downto 0);
      k(ENC_K_DELAY downto 1) <= k(ENC_K_DELAY - 1 downto 0);
      k_less(ENC_K_DELAY downto 1)
                        <= k_less(ENC_K_DELAY - 1 downto 0);
      k_int(ENC_K_DELAY downto 1)
```

```
                          <= k_int(ENC_K_DELAY - 1 downto 0);

      -- Input signals:
      index(0)  <= index_in;
      sign(0)   <= sign_in;
      k(0)      <= k_in;
      k_less(0) <= k_less_in;
      k_int(0)  <= k_int_in;

      -- Output signals:
      index_out1 <= index(4);
      index_out2 <= index(ENC_INDEX_DELAY);
      sign_out   <= sign(ENC_SIGN_DELAY);
      k_out      <= k(ENC_K_DELAY);
      k_less_out <= k_less(ENC_K_DELAY);
      k_int_out  <= k_int(ENC_K_DELAY);
    end if;
  end process propagate_signals;
end architecture RTL;
```

enc_reset.vhd

```
entity reset is
  port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    new_frame  : in  std_logic;
    ready      : out std_logic;
    ctxt_addr  : out std_logic_vector(CTXT_ADDR_SIZE - 1 downto 0);
    ctxt_rst   : out std_logic;
    cache_rst  : out std_logic;
    index_rst  : out std_logic;
    loco_k_rst : out std_logic;
    loco_rst   : out std_logic;
    loco_w_rst : out std_logic;
    gol_rst    : out std_logic
  );
end entity reset;

architecture RTL of reset is
  signal counter     : std_logic_vector(12 downto 0);
  signal index_rst0  : std_logic_vector(ENC_INDEX_RST downto 0);
  signal loco_k_rst0 : std_logic_vector(ENC_K_RST downto 0);
  signal loco_rst0   : std_logic_vector(ENC_LOCO_RST downto 0);
  signal gol_rst0    : std_logic_vector(ENC_GOL_RST downto 0);
  signal loco_w_rst0 : std_logic_vector(ENC_LOCO_W_RST downto 0);
begin

  -- Count through all addresses in context memory and write 0:
  empty_cache : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        counter   <= (others => '0');
        ctxt_addr <= (others => '0');
      else
        if new_frame = '1' then
          counter   <= counter + 1;
          ctxt_addr <= counter(CTXT_ADDR_SIZE - 1 downto 0);
        else
          counter   <= (others => '0');
          ctxt_addr <= (others => '0');
        end if;
      end if;
    end if;
  end process empty_cache;

  -- Delay reset signals to all modules in the design:
  reset : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        ready       <= '0';
        index_rst   <= '1';
        index_rst   <= '1';
        loco_k_rst  <= '1';
    loco_rst    <= '1';
    loco_w_rst  <= '1';
    gol_rst     <= '1';
    index_rst0  <= (others => '1');
    loco_k_rst0 <= (others => '1');
    loco_rst0   <= (others => '1');
    gol_rst0    <= (others => '1');
  else
    if new_frame = '1' then
      -- After counting two lines and line cache is empty:
      if counter(12) = '1' then
        -- Set input value:
        ready                              <= '1';
        index_rst0(0)                      <= '0';
        loco_k_rst0(0)                     <= '0';
        loco_rst0(0)                       <= '0';
        loco_w_rst0(0)                     <= '0';
        gol_rst0(0)                        <= '0';

        -- Propagate signals:
        index_rst0(ENC_INDEX_RST downto 1)
          <= index_rst0(ENC_INDEX_RST - 1 downto 0);
        loco_k_rst0(ENC_K_RST downto 1)
          <= loco_k_rst0(ENC_K_RST - 1 downto 0);
        loco_rst0(ENC_LOCO_RST downto 1)
          <= loco_rst0(ENC_LOCO_RST - 1 downto 0);
        loco_w_rst0(ENC_LOCO_W_RST downto 1)
          <= loco_w_rst0(ENC_LOCO_W_RST - 1 downto 0);
        gol_rst0(ENC_GOL_RST downto 1)
          <= gol_rst0(ENC_GOL_RST - 1 downto 0);
      else
        -- Else keep reset:
        ready       <= '0';
        index_rst0  <= (others => '1');
        loco_k_rst0 <= (others => '1');
        loco_rst0   <= (others => '1');
        loco_w_rst0 <= (others => '1');
        gol_rst0    <= (others => '1');
      end if;
    else
      -- Keep not reset after new frame has started:
      ready                              <= '0';
      index_rst0(0)                      <= '0';
      loco_k_rst0(0)                     <= '0';
      loco_rst0(0)                       <= '0';
      loco_w_rst0(0)                     <= '0';
      gol_rst0(0)                        <= '0';
      index_rst0(ENC_INDEX_RST downto 1)
        <= index_rst0(ENC_INDEX_RST - 1 downto 0);
      loco_k_rst0(ENC_K_RST downto 1)
        <= loco_k_rst0(ENC_K_RST - 1 downto 0);
      loco_rst0(ENC_LOCO_RST downto 1)
        <= loco_rst0(ENC_LOCO_RST - 1 downto 0);
      loco_w_rst0(ENC_LOCO_W_RST downto 1)
        <= loco_w_rst0(ENC_LOCO_W_RST - 1 downto 0);
      gol_rst0(ENC_GOL_RST downto 1)
        <= gol_rst0(ENC_GOL_RST - 1 downto 0);
```

```
          end if;
          -- Output delayed reset signals:
          index_rst  <= index_rst0(ENC_INDEX_RST);
          loco_k_rst <= loco_k_rst0(ENC_K_RST);
          loco_rst   <= loco_rst0(ENC_LOCO_RST);
          loco_w_rst <= loco_w_rst0(ENC_LOCO_W_RST);
          gol_rst    <= gol_rst0(ENC_GOL_RST);
          ctxt_rst   <= index_rst0(ENC_INDEX_RST);
          cache_rst  <= index_rst0(ENC_INDEX_RST);
        end if;
      end if;
    end process reset;
  end architecture RTL;
```

A.3 Source coding - encoder

gol_enc_mux.vhd

```
entity gol_enc_mux is
  Port(
    D : in  std_logic_vector(1 downto 0);
    Y : out std_logic;
    S : in  std_logic
  );
end gol_enc_mux;
architecture arch of gol_enc_mux is
-- Select q,r or 0 based on position (D):
begin
  with D select Y <=
    '1' when "00",
    '0' when "10",
    S when "01",
    S when others;
end arch;
```

gol_enc_shift.vhd

```
entity gol_enc_shift is
  port(
    clk : in  std_logic;
    rst : in  std_logic;
    k   : in  std_logic_vector(8 downto 0);
    O   : out std_logic_vector(SMPL_LEN - 1 downto 0);
    I   : in  std_logic_vector(SMPL_LEN - 1 downto 0)
  );
end gol_enc_shift;
architecture arch of gol_enc_shift is
begin
  process(clk) is
  begin
  -- Shift input based on k = 0 ... 8:
    if rising_edge(clk) then
        if k = "000000001" then
          O <= I(7 downto 0);
        elsif k = "000000010" then
          O <= "0" & I(7 downto 1);
        elsif k = "000000100" then
          O <= "00" & I(7 downto 2);
        elsif k = "000001000" then
          O <= "000" & I(7 downto 3);
        elsif k = "000010000" then
          O <= "0000" & I(7 downto 4);
        elsif k = "000100000" then
          O <= "00000" & I(7 downto 5);
        elsif k = "001000000" then
          O <= "000000" & I(7 downto 6);
        elsif k = "010000000" then
          O <= "0000000" & I(7 downto 7);
        elsif k = "100000000" then
          O <= "00000000";
        end if;
      end if;
  end process;
end arch;
```

gol_enc_top.vhd

```
entity gol_enc_top is
  port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    k_in       : in  std_logic_vector(8 downto 0);
    k_less_in  : in  std_logic_vector(8 downto 0);
    k_int_in   : in  std_logic_vector(3 downto 0);
    input      : in  std_logic_vector(SMPL_LEN - 1 downto 0);
    B          : in  std_logic_vector(B_LEN - 1 downto 0);
    N          : in  std_logic_vector(N_LEN - 1 downto 0);
    output     : out std_logic_vector(GOL_MAX_LEN - 1 downto 0);
    length_out : out std_logic_vector(3 downto 0)
  );
end entity gol_enc_top;

architecture beh of gol_enc_top is
  signal inReg     : std_logic_vector(GOL_MAX_LEN - 1 downto 0);
  signal k_less    : std_logic_vector(GOL_MAX_LEN - 1 downto 0);
  signal k         : std_logic_vector(GOL_MAX_LEN - 1 downto 0);
  signal k_int_t1  : std_logic_vector(3 downto 0);
  signal reg_t1    : std_logic_vector(GOL_MAX_LEN - 1 downto 0);
  signal input_t1  : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal q_t2      : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal k_less_t1 : std_logic_vector(8 downto 0);
  signal k_t1      : std_logic_vector(8 downto 0);
  signal reg_t2    : std_logic_vector(GOL_MAX_LEN - 1 downto 0);
  signal input_t2  : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal k_int_t2  : std_logic_vector(3 downto 0);
begin
  inReg(GOL_MAX_LEN - 1 downto SMPL_LEN) <= (others => '0');
  inReg(SMPL_LEN - 1 downto 0)           <= input_t1;
  k_less(8 downto 0)                     <= k_less_t1;
  k_less(GOL_MAX_LEN - 1 downto 9)       <= (others => '0');
  k(8 downto 0)                          <= k_t1;
  k(GOL_MAX_LEN - 1 downto 9)            <= (others => '0');
  -- Mapping of twosided function to onesided:
  mapping : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        input_t1 <= (others => '0');
      else
        if input(7) = '1' then
          input_t1(7 downto 1) <= not input(6 downto 0);
          input_t1(0)          <= '1';
        else
          input_t1(7 downto 1) <= input(6 downto 0);
          input_t1(0)          <= '0';
        end if;
      end if;
    end if;
  end process mapping;
  -- Find q:
  shiftreg : entity work.gol_enc_shift port map(
        clk => clk,
        rst => rst,
        k   => k_t1,
        I   => input_t1,
        O   => q_t2
      );
    -- Pipeline registers:
    delay_input : process(clk) is
    begin
      if rising_edge(clk) then
        if rst = '1' then
          input_t2  <= (others => '0');
          k_int_t1  <= (others => '0');
          k_int_t2  <= (others => '0');
          k_less_t1 <= (others => '0');
          k_t1      <= "000000001";
          reg_t2    <= (others => '0');
        else
          input_t2  <= input_t1;
          k_int_t1  <= k_int_in;
          k_int_t2  <= k_int_t1;
          k_less_t1 <= k_less_in;
          k_t1      <= k_in;
          reg_t2    <= reg_t1;
        end if;
      end if;
    end process delay_input;

    -- Instantiate n number of muxes to compute golomb code:
    n_mux_modules : for n in GOL_MAX_LEN - 1 downto 0 generate
      mux_module : entity work.gol_enc_mux port map(
          D(0) => k_less(n),
          D(1) => k(n),
          Y    => reg_t1(n),
          S    => inReg(n)
        );
    end generate;

    -- Find length and output data:
    reduce : process(clk) is
    begin
      if rising_edge(clk) then
        if rst = '1' then
          length_out <= (others => '0');
          output     <= (others => '0');
        else
          if (q_t2 > 6) then -- If q longer than gol max length
            length_out <= "1111";
            output     <= "11111110" & input_t2;
          else
            length_out <= q_t2(3 downto 0) + k_int_t2;
            output     <= reg_t2;
          end if;
        end if;
      end if;
    end process reduce;
  end beh;
```

wbuf.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;

--
-- Golomb Packer (WRITE BUFFER)
-- Collect encoded words from golomb encoder (genc)
-- and write Golomb packages when they are ready

entity wbuf is
  generic(                            -- number of input bits
    constant IN_WIDTH   : integer := 16;
    -- number of output bits
    constant OUT_WIDTH  : integer := 16;
    -- number of bits in the internal buffer
    constant WBUF_WIDTH : integer := 32
  );

  port(
    clk         : in  std_ulogic;
    rst         : in  std_ulogic;

    -- INPUTS
    -- Input from the golomb encoder
    in_wbuf     : in  std_logic_vector(IN_WIDTH - 1 downto 0);
    -- Length of the valid part of the input
    word_length : in  std_logic_vector(3 downto 0);
    -- Enable the wbuf Golomb packer
    wbuf_enable : in  std_logic;

    -- OUTPUTS
    -- Golomb packer output
    out_wbuf    : out std_logic_vector(OUT_WIDTH - 1 downto 0);
    -- The golomb packer is empty,
    -- do not expect additional Golomb packages
    wbuf_empty  : out std_logic;
    -- A Golomb packet is ready on the output
    wbuf_write  : out std_logic
  );

end entity wbuf;

architecture RTL of wbuf is
begin
  G_ENC : process(clk, rst, wbuf_enable) is
    variable wbuffer   : std_logic_vector(WBUF_WIDTH - 1 downto 0);
    variable init_done : boolean;

    variable num_elements : std_logic_vector(4 downto 0);
    variable write_pos    : std_logic_vector(4 downto 0);
    variable read_pos     : std_logic_vector(4 downto 0);

    --variable shiftL     : std_ulogic_vector(4 downto 0);
    variable shift_input : std_logic_vector(in_wbuf'range);
  begin
    if (rst = '1') then
      wbuffer      := (others => '0');
      init_done    := false;
      num_elements := (others => '0');
      write_pos    := (others => '1');
      read_pos     := (others => '1');

      out_wbuf   <= (others => '0');
      wbuf_empty <= '0';
      wbuf_write <= '0';

    elsif (rising_edge(clk) and wbuf_enable = '1') then
      if (init_done) then

        -- Shifting the input to the left so that the valid part
        -- starts at position 15 of 'shift_input'.
        -- This is done because the input (all 16 bits) can then
        -- be written directly to the array.
        -- The alternative would be to pick a variable number of
        -- bits and place them into the array, which
        -- is expected to increase the module complexity\size.
        shift_input(
          IN_WIDTH - 1 downto IN_WIDTH
          - conv_integer((word_length)) - 1
        ) := in_wbuf(conv_integer((word_length)) downto 0);

        -- Checks to see if the 'write_pos' is lower than 16,
        -- which will make the input wrap around 'wbuffer'
        -- and continue at the other end.
        -- (else copies 'shift_input' straight into 'wbuffer'
        if (conv_integer(write_pos) < IN_WIDTH) then
          wbuffer(
            conv_integer(write_pos) downto 0
          ) := shift_input(
            IN_WIDTH - 1 downto IN_WIDTH - 1 - conv_integer(write_pos)
          );

          wbuffer(
            WBUF_WIDTH - 1 downto WBUF_WIDTH - 1
              - (IN_WIDTH - 1 - conv_integer(write_pos) - 1)
          ) := shift_input(
            IN_WIDTH - 1 - conv_integer(write_pos) - 1 downto 0
          );
        else
          wbuffer(
            conv_integer(write_pos)
            downto conv_integer(write_pos) - IN_WIDTH + 1
          ) := shift_input;
        end if;

        -- Add 'word_length' to the number of elements (bits)
        -- currently in the wbuffer
        num_elements := num_elements + (word_length) + "1";
        -- Move the write position by subtracting the word_length
        -- (word_length from 0 to 15 represents a code word
        -- length of 1 to 16 bits)
        write_pos    := write_pos - (word_length) - "1";

        -- Writes data to the output if the number of
        -- elements is more than the Output width.
        -- wbuf_write is set to 1 to indicate that a
        -- Golomb package was written
        -- (else sets wbuf_write back to 0)
        if (num_elements >= OUT_WIDTH) then
          out_wbuf     <= wbuffer(
                            conv_integer(read_pos)
                            downto conv_integer(read_pos)
                            - OUT_WIDTH + 1
                          );
          wbuf_write   <= '1';
          read_pos     := read_pos - "10000";
          num_elements := num_elements - "10000";
        else
          wbuf_write <= '0';
          out_wbuf   <= (others => '0');
        end if;
        wbuf_empty <= '0';
      else
        num_elements := (others => '0');
        wbuf_empty   <= '1';
        write_pos    := (others => '1');
        read_pos     := (others => '1');
        init_done    := true;
      end if;

    -- When the enable goes low, there could still be data left
    -- in the write buffer. This ensures that the last golomb
    -- package is written even if it doesnt contain only valid bits.
    -- A counter in the decoder will know when all pixels have been
    -- decoded, and the unvalid bits will be ignored
    elsif (rising_edge(clk) and wbuf_enable = '0') then
      if (num_elements > 0) then
        out_wbuf     <= wbuffer(
                          conv_integer(read_pos)
                            downto conv_integer(read_pos)
                            - OUT_WIDTH + 1
                        );
        wbuf_write   <= '1';
        wbuf_empty   <= '0';
        num_elements := (others => '0');
      else
        wbuf_empty <= '1';
        wbuf_write <= '0';
        out_wbuf   <= (others => '0');
        read_pos   := (others => '1');
        write_pos  := (others => '1');
        init_done  := false;
      end if;
    end if;
  end process;
end architecture RTL;
```

A.4 Toplevel - encoder

enc_top.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.defs.all;

entity enc_top is
  port(
    clk       : in  std_logic;
    rst       : in  std_logic;
    data_in   : in  std_logic_vector(7 downto 0);
    new_frame : in  std_logic;
    ready     : out std_logic;
    data_out  : out std_logic_vector(GOL_MAX_LEN - 1 downto 0);
    size_out  : out std_logic_vector(3 downto 0)
  );
end entity enc_top;

architecture RTL of enc_top is
  signal data_in_c : std_logic_vector(7 downto 0);

  --index
  signal index_rst : std_logic;
  signal B_i       : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal C_i       : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal D_i       : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal I_i       : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal index_out : std_logic_vector(CTXT_ADDR_SIZE - 1 downto 0);
  signal sign_out  : std_logic;

  --k
  signal loco_k_rst : std_logic;

  signal A_k_out    : std_logic_vector(A_LEN - 1 downto 0);
  signal N_k_out    : std_logic_vector(N_LEN - 1 downto 0);
  signal k_out      : std_logic_vector(8 downto 0);
  signal k_less_out : std_logic_vector(8 downto 0);
  signal k_int_out  : std_logic_vector(3 downto 0);

  --loco
  signal loco_rst : std_logic;

  signal A_out      : std_logic_vector(A_LEN - 1 downto 0);
  signal B_out      : std_logic_vector(B_LEN - 1 downto 0);
  signal C_out      : std_logic_vector(C_LEN - 1 downto 0);
  signal N_out      : std_logic_vector(N_LEN - 1 downto 0);
  signal B_l        : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal sign_d     : std_logic;
  signal loco_w_rst : std_logic;
  signal index_d2   : std_logic_vector(7 downto 0);
  signal A_in       : std_logic_vector(A_LEN - 1 downto 0);
  signal B_in       : std_logic_vector(B_LEN - 1 downto 0);
  signal C_in       : std_logic_vector(C_LEN - 1 downto 0);
  signal N_in       : std_logic_vector(N_LEN - 1 downto 0);
  signal Q          : std_logic_vector(SMPL_LEN - 1 downto 0);

  --gol
  signal gol_rst  : std_logic;
  signal k_less_d : std_logic_vector(8 downto 0);
  signal k_d      : std_logic_vector(8 downto 0);
  signal k_int_d  : std_logic_vector(3 downto 0);

  --context
  signal ctxt_addr_rst : std_logic_vector(CTXT_ADDR_SIZE - 1 downto 0);
  signal ctxt_rst      : std_logic;

  --output
  signal data_out_c : std_logic_vector(GOL_MAX_LEN - 1 downto 0);
  signal size_out_c : std_logic_vector(3 downto 0);

  signal cache_rst   : std_logic;
  signal not_rst     : std_logic;
  signal index_d1    : std_logic_vector(7 downto 0);
  signal B_gol       : std_logic_vector(B_LEN - 1 downto 0);
  signal N_gol       : std_logic_vector(N_LEN - 1 downto 0);
  signal data_in_c1 : std_logic_vector(7 downto 0);
  signal data_in_c2 : std_logic_vector(7 downto 0);
  signal data_in_c3 : std_logic_vector(7 downto 0);
begin
  not_rst <= not rst;

  clocked_in_out : process(clk, rst) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        data_in_c <= (others => '0');
        data_out  <= (others => '0');
        size_out  <= (others => '0');
      else
        data_in_c  <= data_in;
        data_in_c3 <= data_in_c2;
        data_in_c2 <= data_in_c1;
        data_in_c1 <= data_in_c;

        data_out <= data_out_c;
        size_out <= size_out_c;
      end if;
    end if;
  end process clocked_in_out;

  index_module : entity work.index port map(
      clk       => clk,
      rst       => index_rst,
      B_cache   => B_i,
      C_cache   => C_i,
      D_cache   => D_i,
      I_cache   => I_i,
      index_out => index_out,
            sign_out  => sign_out
          );
        find_k_module : entity work.find_k port map(
            clk        => clk,
            rst        => loco_k_rst,
            A          => A_k_out,
            N          => N_k_out,
            k_out      => k_out,
            k_less_out => k_less_out,
            k_int_out  => k_int_out
          );
        modeling_module : entity work.loco port map(
            clk        => clk,
            rst        => loco_rst,
            loco_w_rst => loco_w_rst,
            data       => data_in_c,
            sign       => sign_d,
            P          => B_l,
            A          => A_out,
            B          => B_out,
            C          => C_out,
            N          => N_out,
            A_out      => A_in,
            B_out      => B_in,
            C_out      => C_in,
            N_out      => N_in,
            B0_out     => B_gol,
            N0_out     => N_gol,
            Q_out      => Q
          );
        gol_module : entity work.gol_enc_top port map(
            clk        => clk,
            rst        => gol_rst,
            k_in       => k_d,
            k_less_in  => k_less_d,
            k_int_in   => k_int_d,
            input      => Q,
            B          => B_gol,
            N          => N_gol,
            output     => data_out_c,
            length_out => size_out_c
          );
        ctxt_reg_module : entity work.ctxt_reg port map(
            clk          => clk,
            rst          => rst,
            write_0      => ctxt_rst,
            write0_addr  => ctxt_addr_rst,
            read_index   => index_d1,
            read_k_index => index_out,
            write_index  => index_d2,
            A_in         => A_in,
            B_in         => B_in,
            C_in         => C_in,
            N_in         => N_in,
            A_out        => A_out,
            A_k_out      => A_k_out,
            B_out        => B_out,
      C_out        => C_out,
      N_out        => N_out,
      N_k_out      => N_k_out
    );
  cache_module : entity work.line_cache port map(
      clk        => clk,
      shift      => not_rst,
      write_0    => cache_rst,
      sr_in      => data_in_c3,
      sr_tap_B   => B_i,
      sr_tap_C   => C_i,
      sr_tap_D   => D_i,
      sr_tap_I   => I_i,
      sr_tap_B_l => B_l
    );
  delay_module : entity work.enc_signal_delays port map(
      clk        => clk,
      rst        => rst,
      index_in   => index_out,
      sign_in    => sign_out,
      k_less_in  => k_less_out,
      k_in       => k_out,
      k_int_in   => k_int_out,
      index_out1 => index_d1,
      index_out2 => index_d2,
      sign_out   => sign_d,
      k_less_out => k_less_d,
      k_out      => k_d,
      k_int_out  => k_int_d
    );
  reset_module : entity work.reset port map(
      clk        => clk,
      rst        => rst,
      new_frame  => new_frame,
      ready      => ready,
      ctxt_addr  => ctxt_addr_rst,
      ctxt_rst   => ctxt_rst,
      cache_rst  => cache_rst,
      index_rst  => index_rst,
      loco_k_rst => loco_k_rst,
      loco_rst   => loco_rst,
      loco_w_rst => loco_w_rst,
      gol_rst    => gol_rst
    );
end architecture RTL;
```

A.5 Modeling and signals - decoder

dec_modeling.vhd

```
entity dec_modeling is
  port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    data       : in  std_logic_vector(SMPL_LEN - 1 downto 0);
    loco_w_rst : in  std_logic;
    sign       : in  std_logic;
    P          : in  std_logic_vector(SMPL_LEN - 1 downto 0);
    k_int_in   : in  std_logic_vector(3 downto 0);
    k_orig     : in  std_logic_vector(3 downto 0);
    A          : in  std_logic_vector(A_LEN - 1 downto 0);
    B          : in  std_logic_vector(B_LEN - 1 downto 0);
    C          : in  std_logic_vector(C_LEN - 1 downto 0);
    N          : in  std_logic_vector(N_LEN - 1 downto 0);
    A_out      : out std_logic_vector(A_LEN - 1 downto 0);
    B_out      : out std_logic_vector(B_LEN - 1 downto 0);
    C_out      : out std_logic_vector(C_LEN - 1 downto 0);
    N_out      : out std_logic_vector(N_LEN - 1 downto 0);
    Q_out      : out std_logic_vector(SMPL_LEN - 1 downto 0)
  );
end entity dec_modeling;

architecture RTL of dec_modeling is
  signal e       : std_logic_vector(C_LEN downto 0);
  signal A_t1    : std_logic_vector(A_LEN - 1 downto 0);
  signal B_t1    : std_logic_vector(C_LEN downto 0);
  signal C_t1    : std_logic_vector(C_LEN downto 0);
  signal N_t1    : std_logic_vector(C_LEN downto 0);
  signal A_t2    : std_logic_vector(A_LEN - 1 downto 0);
  signal B_t2    : std_logic_vector(C_LEN downto 0);
  signal C_t2    : std_logic_vector(C_LEN downto 0);
  signal N_t2    : std_logic_vector(C_LEN downto 0);
  signal A_t3    : std_logic_vector(A_LEN - 1 downto 0);
  signal B_t3    : std_logic_vector(C_LEN downto 0);
  signal C_t3    : std_logic_vector(C_LEN downto 0);
  signal N_t3    : std_logic_vector(C_LEN downto 0);
  signal e_abs   : std_logic_vector(C_LEN downto 0);
  signal A_t0    : std_logic_vector(A_LEN - 1 downto 0);
  signal B_t0    : std_logic_vector(B_LEN - 1 downto 0);
  signal C_t0    : std_logic_vector(C_LEN - 1 downto 0);
  signal N_t0    : std_logic_vector(N_LEN - 1 downto 0);
  signal P_t0    : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal A_temp  : std_logic_vector(A_LEN - 1 downto 0);
  signal B_temp  : std_logic_vector(C_LEN downto 0);
  signal C_temp  : std_logic_vector(C_LEN downto 0);
  signal N_temp  : std_logic_vector(C_LEN downto 0);
  signal B_t2t1  : std_logic_vector(C_LEN downto 0);
  signal B_t2t2  : std_logic_vector(C_LEN downto 0);
  signal C_t2t1  : std_logic_vector(C_LEN downto 0);
  signal C_t2t2  : std_logic_vector(C_LEN downto 0);
  signal sign_t0 : std_logic;
  signal e_t0    : std_logic_vector(7 downto 0);
  signal data_t1 : std_logic_vector(C_LEN downto 0);
begin
  mapping : process(clk) is
    variable temp2B : std_logic_vector(6 downto 0);
  begin
    if rising_edge(clk) then
      if rst = '1' then
        e_t0 <= (others => '0');
      else
       -- map error to two sided function:
        if data(0) = '1' then
          e_t0(6 downto 0) <= not data(7 downto 1);
          e_t0(7)          <= '1';
        else
          e_t0(6 downto 0) <= data(7 downto 1);
          e_t0(7)          <= '0';
        end if;
      end if;
    end if;
  end process mapping;

  -- decode input:
  adj_e : process(clk, rst) is
    variable temp_e     : std_logic_vector(C_LEN downto 0);
    variable temp_data1 : std_logic_vector(C_LEN downto 0);
    variable temp_data2 : std_logic_vector(C_LEN downto 0);
  begin
    if rising_edge(clk) then
      if rst = '1' then
        e       <= (others => '0');
        e_abs   <= (others => '0');
        A_t1    <= (others => '0');
        B_t1    <= (others => '0');
        C_t1    <= (others => '0');
        N_t1    <= (others => '0');
        A_t0    <= (others => '0');
        B_t0    <= (others => '0');
        C_t0    <= (others => '0');
        N_t0    <= (others => '0');
        P_t0    <= (others => '0');
        data_t1 <= (others => '0');
        sign_t0 <= '0';
      else
        -- Pipeline signals:
        A_t0    <= A;
        B_t0    <= B;
        C_t0    <= C;
        N_t0    <= N;
        P_t0    <= P;
        sign_t0 <= sign;

        -- find absolute value of e:
        temp_e := e_t0(7) & e_t0(7) & e_t0;
        if e_t0(7) = '1' then
          e_abs <= 0 - temp_e;
        else
```

```
          e_abs <= temp_e;
        end if;
        e <= temp_e;

        -- Find original sample value based on predictor (P) and
        -- context (C):
        temp_data1 := (P_t0(7) & P_t0(7) & P_t0)
          + (C_t0(8) & C_t0) + (e_t0(7) & e_t0(7) & e_t0);
        temp_data2 := (P_t0(7) & P_t0(7) & P_t0)
          - (e_t0(7) & e_t0(7) & e_t0) - (C_t0(8) & C_t0);

        -- Select right value based on sign value:
        if (sign_t0 = '1') then
          data_t1 <= temp_data1;
        else
          data_t1 <= temp_data2;
        end if;

        -- Extend bit lengths and half if N >= 32:
        if N_t0(5) = '0' then
          A_t1 <= A_t0;
          B_t1 <= B_t0(5) & B_t0(5) & B_t0(5) & B_t0(5) & B_t0;
          C_t1 <= C_t0(8) & C_t0;
          N_t1 <= "0000" & N_t0;
        else
          A_t1 <= A_t0(A_LEN - 1) & A_t0(A_LEN - 1 downto 1);
          B_t1 <= B_t0(5) & B_t0(5) & B_t0(5)
            & B_t0(5) & B_t0(5) & B_t0(B_LEN - 1 downto 1);
          C_t1 <= C_t0(8) & C_t0;
          N_t1 <= "0000001111";
        end if;
      end if;
    end if;
  end process adj_e;

  modeling : process(clk, rst) is
  begin
    if rising_edge(clk) then
      if loco_w_rst = '1' then
        A_out  <= (others => '0');
        B_out  <= (others => '0');
        C_out  <= (others => '0');
        N_out  <= (others => '0');
        A_t2   <= (others => '0');
        B_t2   <= (others => '0');
        C_t2   <= (others => '0');
        N_t2   <= (others => '0');
        A_t3   <= (others => '0');
        B_t3   <= (others => '0');
        C_t3   <= (others => '0');
        N_t3   <= (others => '0');
        B_t2t1 <= (others => '0');
        B_t2t2 <= (others => '0');
        C_t2t1 <= (others => '0');
        C_t2t2 <= (others => '0');
        A_temp <= (others => '0');
        B_temp <= (others => '0');
    C_temp <= (others => '0');
    N_temp <= (others => '0');
  else

  -- Context modeling procedure:
    A_t2   <= A_t1 + e_abs;
    B_t2   <= B_t1 + e;
    C_t2   <= C_t1;
    N_t2   <= N_t1 + 1;
    B_t2t1 <= B_t1 + e + N_t1 + 1;
    B_t2t2 <= B_t1 + e - N_t1 - 1;
    C_t2t1 <= C_t1 - 1;
    C_t2t2 <= C_t1 + 1;

    if ((B_t2 <= "0" - N_t2) and B_t2(C_LEN) = '1') then
      A_t3 <= A_t2;
      B_t3 <= B_t2t1;
      C_t3 <= C_t2t1;
      N_t3 <= N_t2;
      if ((B_t3 <= 0 - N_t3) and B_t3(C_LEN) = '1') then
        A_temp <= A_t3;
        B_temp <= 0 - N_t3 + 1;
        C_temp <= C_t3;
        N_temp <= N_t3;
      else
        A_temp <= A_t3;
        B_temp <= B_t3;
        C_temp <= C_t3;
        N_temp <= N_t3;
      end if;
    elsif (B_t2 > 0 and B_t2(C_LEN) = '0') then
      A_t3 <= A_t2;
      B_t3 <= B_t2t2;
      C_t3 <= C_t2t2;
      N_t3 <= N_t2;
      if ((B_t3 <= 0 - N_t3) and B_t3(C_LEN) = '1') then
        A_temp <= A_t3;
        B_temp <= (others => '0');
        C_temp <= C_t3;
        N_temp <= N_t3;
      else
        A_temp <= A_t3;
        B_temp <= B_t3;
        C_temp <= C_t3;
        N_temp <= N_t3;
      end if;
    else
      A_t3   <= A_t2;
      B_t3   <= B_t2;
      C_t3   <= C_t2;
      N_t3   <= N_t2;
      A_temp <= A_t3;
      B_temp <= B_t3;
      C_temp <= C_t3;
      N_temp <= N_t3;
    end if;
```

```
          -- Output new A,B,C,N:
          A_out  <= A_temp;
          B_out  <= B_temp(B_LEN - 1 downto 0);
          C_out  <= C_temp(C_LEN - 1 downto 0);
          N_out  <= N_temp(N_LEN - 1 downto 0);
        end if;
      end if;
    end process modeling;

    -- Output decoded data:
    update_output : process(clk, rst) is
    begin
      if rst = '1' then
        Q_out <= (others => '0');
      elsif rising_edge(clk) then
        Q_out <= data_t1(SMPL_LEN - 1 downto 0);
      end if;
    end process update_output;
  end architecture RTL;
```

dec_signal_delays.vhd

```
entity dec_signal_delays is
  port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    index_in   : in  std_logic_vector(7 downto 0);
    sign_in    : in  std_logic;
    k_less_in  : in  std_logic_vector(8 downto 0);
    k_in       : in  std_logic_vector(8 downto 0);
    k_int_in   : in  std_logic_vector(3 downto 0);
    index_out1 : out std_logic_vector(7 downto 0);
    index_out2 : out std_logic_vector(7 downto 0);
    sign_out   : out std_logic;
    k_int_out  : out std_logic_vector(3 downto 0)
  );
end entity dec_signal_delays;

architecture RTL of dec_signal_delays is
  type k_array is array (DEC_K_DELAY downto 0)
    of std_logic_vector(8 downto 0);
  type k_less_array is array (DEC_K_DELAY downto 0)
    of std_logic_vector(8 downto 0);
  type k_int_array is array (DEC_K_DELAY downto 0)
    of std_logic_vector(3 downto 0);
  type index_array is array (DEC_INDEX_DELAY downto 0)
    of std_logic_vector(7 downto 0);
  type sign_array is array (DEC_SIGN_DELAY downto 0)
    of std_logic;

  signal k      : k_array;
  signal k_less : k_less_array;
  signal k_int  : k_int_array;
  signal index : index_array;
  signal sign   : sign_array;
begin
  propagate_signals : process(clk, rst) is
  begin
    if rising_edge(clk) then

      -- Propagate signals through registers:
      index(DEC_INDEX_DELAY downto 1)
        <= index(DEC_INDEX_DELAY - 1 downto 0);
      sign(DEC_SIGN_DELAY downto 1)
        <= sign(DEC_SIGN_DELAY - 1 downto 0);
      k(DEC_K_DELAY downto 1)
        <= k(DEC_K_DELAY - 1 downto 0);
      k_less(DEC_K_DELAY downto 1)
        <= k_less(DEC_K_DELAY - 1 downto 0);
      k_int(DEC_K_DELAY downto 1)
        <= k_int(DEC_K_DELAY - 1 downto 0);

      -- Input signals
      index(0)    <= index_in;
      k(0)        <= k_in;
      k_less(0)   <= k_less_in;
```

```
            k_int(0)    <= k_int_in;
            sign(0)     <= sign_in;

            -- Output signals:
            index_out1 <= index(4);
            index_out2 <= index(DEC_INDEX_DELAY);
            sign_out   <= sign(DEC_SIGN_DELAY);
            k_int_out  <= k_int(DEC_K_DELAY);
          end if;
        end process propagate_signals;
      end architecture RTL;
```

dec_reset.vhd

```
entity dec_reset is
  port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    new_frame  : in  std_logic;
    ready      : out std_logic;
    ctxt_addr  : out std_logic_vector(CTXT_ADDR_SIZE - 1 downto 0);
    ctxt_rst   : out std_logic;
    cache_rst  : out std_logic;
    index_rst  : out std_logic;
    loco_k_rst : out std_logic;
    loco_rst   : out std_logic;
    loco_w_rst : out std_logic;
    gol_rst    : out std_logic
  );
end entity dec_reset;

architecture RTL of dec_reset is
  signal counter     : std_logic_vector(12 downto 0);
  signal index_rst0  : std_logic_vector(DEC_INDEX_RST downto 0);
  signal loco_k_rst0 : std_logic_vector(DEC_K_RST downto 0);
  signal loco_rst0   : std_logic_vector(DEC_LOCO_RST downto 0);
  signal gol_rst0    : std_logic_vector(DEC_GOL_RST downto 0);
  signal loco_w_rst0 : std_logic_vector(DEC_LOCO_W_RST downto 0);
begin

  -- Count through all addresses in context memory and write 0:
  empty_cache : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        counter   <= (others => '0');
        ctxt_addr <= (others => '0');
      else
        if new_frame = '1' then
          counter   <= counter + 1;
          ctxt_addr <= counter(CTXT_ADDR_SIZE - 1 downto 0);
        else
          counter   <= (others => '0');
          ctxt_addr <= (others => '0');
        end if;
      end if;
    end if;
  end process empty_cache;

  -- Delay reset signals to all modules in the design:
  reset : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        ready       <= '0';
        index_rst   <= '1';
        index_rst   <= '1';
        loco_k_rst  <= '1';
  loco_rst    <= '1';
  loco_w_rst  <= '1';
  gol_rst     <= '1';
  index_rst0  <= (others => '1');
  loco_k_rst0 <= (others => '1');
  loco_rst0   <= (others => '1');
  gol_rst0    <= (others => '1');
else

  if new_frame = '1' then
  -- After counting two lines and line cache is empty:
    if counter(12) = '1' then

      -- Set input value:
      ready                                <= '1';
      index_rst0(0)                        <= '0';
      loco_k_rst0(0)                       <= '0';
      loco_rst0(0)                         <= '0';
      loco_w_rst0(0)                       <= '0';
      gol_rst0(0)                          <= '0';

      -- Propagate signals:
      index_rst0(DEC_INDEX_RST downto 1)
        <= index_rst0(DEC_INDEX_RST - 1 downto 0);
      loco_k_rst0(DEC_K_RST downto 1)
        <= loco_k_rst0(DEC_K_RST - 1 downto 0);
      loco_rst0(DEC_LOCO_RST downto 1)
        <= loco_rst0(DEC_LOCO_RST - 1 downto 0);
      loco_w_rst0(DEC_LOCO_W_RST downto 1)
        <= loco_w_rst0(DEC_LOCO_W_RST - 1 downto 0);
      gol_rst0(DEC_GOL_RST downto 1)
        <= gol_rst0(DEC_GOL_RST - 1 downto 0);
    else
      -- Else keep reset:
      ready       <= '0';
      index_rst0  <= (others => '1');
      loco_k_rst0 <= (others => '1');
      loco_rst0   <= (others => '1');
      loco_w_rst0 <= (others => '1');
      gol_rst0    <= (others => '1');
    end if;
  else
    -- Keep not reset after new frame has started:
    ready                                <= '0';
    index_rst0(0)                        <= '0';
    loco_k_rst0(0)                       <= '0';
    loco_rst0(0)                         <= '0';
    loco_w_rst0(0)                       <= '0';
    gol_rst0(0)                          <= '0';

    index_rst0(DEC_INDEX_RST downto 1)
      <= index_rst0(DEC_INDEX_RST - 1 downto 0);
    loco_k_rst0(DEC_K_RST downto 1)
      <= loco_k_rst0(DEC_K_RST - 1 downto 0);
    loco_rst0(DEC_LOCO_RST downto 1)
      <= loco_rst0(DEC_LOCO_RST - 1 downto 0);
    loco_w_rst0(DEC_LOCO_W_RST downto 1)
```

```
          <= loco_w_rst0(DEC_LOCO_W_RST - 1 downto 0);
        gol_rst0(DEC_GOL_RST downto 1)
          <= gol_rst0(DEC_GOL_RST - 1 downto 0);
      end if;
      -- Output delayed reset signals:
      index_rst  <= index_rst0(DEC_INDEX_RST);
      loco_k_rst <= loco_k_rst0(DEC_K_RST);
      loco_rst   <= loco_rst0(DEC_LOCO_RST);
      loco_w_rst <= loco_w_rst0(DEC_LOCO_W_RST);
      gol_rst    <= gol_rst0(DEC_GOL_RST);
      ctxt_rst   <= index_rst0(DEC_INDEX_RST);
      cache_rst  <= index_rst0(DEC_INDEX_RST);
    end if;
  end if;
end process reset;
end architecture RTL;
```

A.6 Source coding - decoder

update_registers.vhd

```
entity update_registers is
  port(
    clk        : in  std_logic;
    k_in       : in  std_logic_vector(3 downto 0);
    k_in_bit   : in  std_logic_vector(8 downto 0);
    get_in     : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    data_in    : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    end_q_out  : out std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    get_out    : out std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    k_out      : out std_logic_vector(3 downto 0)
  );
end entity update_registers;

-- Top module for golomb index registers and counter:
architecture RTL of update_registers is
  signal end_q : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal k8    : std_logic;
begin
  end_q_out <= end_q;
  find_end_q_module : entity work.find_end_q port map(
      clk       => clk,
      end_q     => end_q,
      get_in_in => get_in,
      data_in   => data_in
    );
  k8_module : entity work.k8 port map(
      clk      => clk,
      get_in   => get_in,
      end_q_in => end_q,
      k8       => k8
    );
  find_end_r_module : entity work.find_end_r port map(
      clk      => clk,
      end_q    => end_q,
      k8       => k8,
      get_out  => get_out,
      k_in     => k_in,
      k_in_bit => k_in_bit,
      k_out    => k_out
    );
end architecture RTL;
```

find_end_q.vhd

```
entity find_end_q is
  port(
    clk       : in  std_logic;
    end_q     : out std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    get_in_in : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    data_in   : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0)
  );
end entity find_end_q;

architecture RTL of find_end_q is
  type end_q_arr is array (DEC_REG_SIZE - 1 downto 0)
    of std_logic_vector(7 downto 0);
  signal end_q_n : end_q_arr;
  type data_arr is array (DEC_REG_SIZE - 1 downto 0)
    of std_logic_vector(7 downto 0);
  signal data   : data_arr;
  signal get_in : std_logic_vector(DEC_REG_SIZE - 1 downto 0);

begin
  get_in <= get_in_in;

  -- Divide input portitons to the different 'q' counters:
  connect : for n in DEC_REG_SIZE - 8 downto 0 generate
    data(n) <= data_in(n + 7 downto n);
  end generate;
  data(DEC_REG_SIZE - 7) <= data_in(0)
    & data_in(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 7);
  data(DEC_REG_SIZE - 6) <= data_in(1 downto 0)
    & data_in(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 6);
  data(DEC_REG_SIZE - 5) <= data_in(2 downto 0)
    & data_in(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 5);
  data(DEC_REG_SIZE - 4) <= data_in(3 downto 0)
    & data_in(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 4);
  data(DEC_REG_SIZE - 3) <= data_in(4 downto 0)
    & data_in(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 3);
  data(DEC_REG_SIZE - 2) <= data_in(5 downto 0)
    & data_in(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 2);
  data(DEC_REG_SIZE - 1) <= data_in(6 downto 0)
    & data_in(DEC_REG_SIZE - 1);

  -- Instantiate n 'q' counters to find end of 'q':
  n_q_reg : for n in DEC_REG_SIZE - 1 downto 0 generate
    q_reg : entity work.count_q port map(
        clk   => clk,
        data  => data(n),
        get   => get_in(n),
        end_q => end_q_n(n)
      );
  end generate;

  -- Select and output the 'q' counter that found the end of 'q':
  output : for n in DEC_REG_SIZE - 1 downto 0 generate
    output_proc : process(end_q_n) is
    begin
```

```
        end_q(n) <= (
          end_q_n(n)(0)
          or end_q_n((n - 1) mod DEC_REG_SIZE)(1)
          or end_q_n((n - 2) mod DEC_REG_SIZE)(2)
          or end_q_n((n - 3) mod DEC_REG_SIZE)(3)
          or end_q_n((n - 4) mod DEC_REG_SIZE)(4)
          or end_q_n((n - 5) mod DEC_REG_SIZE)(5)
          or end_q_n((n - 6) mod DEC_REG_SIZE)(6)
          or end_q_n((n - 7) mod DEC_REG_SIZE)(7)
          );
      end process;
    end generate;
  end architecture RTL;
```

count_q.vhd

```
entity count_q is
  port(
    clk   : in std_logic;
    get   : in std_logic;
    data  : in std_logic_vector(7 downto 0);
    end_q : buffer std_logic_vector(7 downto 0)
  );
end entity count_q;

architecture RTL of count_q is
  type q_arr is array (0 to 7) of std_logic;
  signal q : q_arr;

-- Find end of 'q' based on get index and data index:
begin
  q(1)     <= data(0) and get;
  end_q(0) <= not data(0) and get;
  n_dec_reg : for n in 1 to 6 generate
    q((n + 1) mod 8) <= data(n) and q(n);
    end_q(n)         <= not data(n) and q(n);
  end generate;
  end_q(7) <= not data(7) and q(7);
end architecture RTL;
```

k8.vhd

```
entity k8 is
  port(
    clk      : in  std_logic;
    get_in   : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    end_q_in : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    k8       : out std_logic
  );
end entity k8;

architecture RTL of k8 is
  signal k8_arr : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal end_q  : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal get    : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
begin
  get   <= get_in;
  end_q <= end_q_in;

  -- Create AND array for all 8 bit distances between get and end_q:
  find_unenc : process(end_q, get, k8_arr) is
  begin
    for n in 0 to DEC_REG_SIZE - 1 loop
      k8_arr(n) <= get(n) and end_q((n + 7) mod DEC_REG_SIZE);
    end loop;
  end process;

  -- If none of the ANDs are '1' output '1', else output '0'
  or_k8 : process(k8_arr, clk) is
  begin
    if k8_arr = 0 then
      k8 <= '1';
    else
      k8 <= '0';
    end if;
  end process or_k8;
end architecture RTL;
```

find_end_r.vhd

```
entity find_end_r is
  port(
    clk       : in  std_logic;
    end_q     : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    k8        : in  std_logic;
    get_out   : out std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    k_in      : in  std_logic_vector(3 downto 0);
    k_in_bit  : in  std_logic_vector(8 downto 0);
    k_out     : out std_logic_vector(3 downto 0)
  );
end entity find_end_r;

architecture RTL of find_end_r is
  type end_r_arr is array (8 downto 0)
    of std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal end_r : end_r_arr;
begin

  -- Parallel shift of get index for all 9 'k' values:
  end_r(0) <= end_q(DEC_REG_SIZE - 2 downto 0)
    & end_q(DEC_REG_SIZE - 1);
  end_r(1) <= end_q(DEC_REG_SIZE - 3 downto 0)
    & end_q(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 2);
  end_r(2) <= end_q(DEC_REG_SIZE - 4 downto 0)
    & end_q(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 3);
  end_r(3) <= end_q(DEC_REG_SIZE - 5 downto 0)
    & end_q(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 4);
  end_r(4) <= end_q(DEC_REG_SIZE - 6 downto 0)
    & end_q(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 5);
  end_r(5) <= end_q(DEC_REG_SIZE - 7 downto 0)
    & end_q(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 6);
  end_r(6) <= end_q(DEC_REG_SIZE - 8 downto 0)
    & end_q(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 7);
  end_r(7) <= end_q(DEC_REG_SIZE - 9 downto 0)
    & end_q(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 8);
  end_r(8) <= end_q(DEC_REG_SIZE - 10 downto 0)
    & end_q(DEC_REG_SIZE - 1 downto DEC_REG_SIZE - 9);

  -- Select end_r based on 'k':
  get_out_proc : process(clk, k_in, end_r, k8, k_in_bit) is
  begin
    if k8 = '1' then
      k_out <= k_in;
      for n in 0 to DEC_REG_SIZE - 1 loop
        get_out(n) <=
          (end_r(0)(n) and k_in_bit(0))
          or (end_r(1)(n) and k_in_bit(1))
          or (end_r(2)(n) and k_in_bit(2))
          or (end_r(3)(n) and k_in_bit(3))
          or (end_r(4)(n) and k_in_bit(4))
          or (end_r(5)(n) and k_in_bit(5))
          or (end_r(6)(n) and k_in_bit(6))
          or (end_r(7)(n) and k_in_bit(7))
          or (end_r(8)(n) and k_in_bit(8));
        end loop;
      else
        k_out   <= "1000";
        get_out <= end_r(8);
      end if;
    end process;
  end architecture RTL;
```

decode_q.vhd

```
entity decode_q is
  port(
    clk      : in  std_logic;
    rst      : in  std_logic;
    end_q_in : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    get_in   : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    q_out    : out std_logic_vector(2 downto 0)
  );
end entity decode_q;

architecture RTL48 of decode_q is
  signal end_q  : std_logic_vector(DEC_REG_LEN - 1 downto 0);
  signal get    : std_logic_vector(DEC_REG_LEN - 1 downto 0);
  signal q_temp : std_logic_vector(DEC_REG_LEN - 1 downto 0);
begin
  q_out <= q_temp(2 downto 0);

  -- Calculate difference between get bit and end_q bit,
  -- represented in binary code:
  calc_diff : process(clk) is
  begin
    if rising_edge(clk) then
      q_temp <= end_q - get;
    end if;
  end process calc_diff;

  -- Decode position of get bit in get_index,
  -- to binary representation:
  with end_q_in select end_q <=
    "000000" when "000000000000000000000000000000000000000000000001",
    "000001" when "000000000000000000000000000000000000000000000010",
    "000010" when "000000000000000000000000000000000000000000000100",
    "000011" when "000000000000000000000000000000000000000000001000",
    "000100" when "000000000000000000000000000000000000000000010000",
    "000101" when "000000000000000000000000000000000000000000100000",
    "000110" when "000000000000000000000000000000000000000001000000",
    "000111" when "000000000000000000000000000000000000000010000000",
    "001000" when "000000000000000000000000000000000000000100000000",
    "001001" when "000000000000000000000000000000000000001000000000",
    "001010" when "000000000000000000000000000000000000010000000000",
    "001011" when "000000000000000000000000000000000000100000000000",
    "001100" when "000000000000000000000000000000000001000000000000",
    "001101" when "000000000000000000000000000000000010000000000000",
    "001110" when "000000000000000000000000000000000100000000000000",
    "001111" when "000000000000000000000000000000001000000000000000",
    "010000" when "000000000000000000000000000000010000000000000000",
    "010001" when "000000000000000000000000000000100000000000000000",
    "010010" when "000000000000000000000000000001000000000000000000",
    "010011" when "000000000000000000000000000010000000000000000000",
    "010100" when "000000000000000000000000000100000000000000000000",
    "010101" when "000000000000000000000000001000000000000000000000",
    "010110" when "000000000000000000000000010000000000000000000000",
    "010111" when "000000000000000000000000100000000000000000000000",
    "011000" when "000000000000000000000001000000000000000000000000",
    "011001" when "000000000000000000000010000000000000000000000000",
    "011010" when "000000000000000000000100000000000000000000000000",
    "011011" when "000000000000000000001000000000000000000000000000",
    "011100" when "000000000000000000010000000000000000000000000000",
    "011101" when "000000000000000000100000000000000000000000000000",
    "011110" when "000000000000000001000000000000000000000000000000",
    "011111" when "000000000000000010000000000000000000000000000000",
    "100000" when "000000000000000100000000000000000000000000000000",
    "100001" when "000000000000001000000000000000000000000000000000",
    "100010" when "000000000000010000000000000000000000000000000000",
    "100011" when "000000000000100000000000000000000000000000000000",
    "100100" when "000000000001000000000000000000000000000000000000",
    "100101" when "000000000010000000000000000000000000000000000000",
    "100110" when "000000000100000000000000000000000000000000000000",
    "100111" when "000000001000000000000000000000000000000000000000",
    "101000" when "000000010000000000000000000000000000000000000000",
    "101001" when "000000100000000000000000000000000000000000000000",
    "101010" when "000001000000000000000000000000000000000000000000",
    "101011" when "000010000000000000000000000000000000000000000000",
    "101100" when "000100000000000000000000000000000000000000000000",
    "101101" when "001000000000000000000000000000000000000000000000",
    "101110" when "010000000000000000000000000000000000000000000000",
    "101111" when "100000000000000000000000000000000000000000000000",
    "XXXXXX" when others;

  -- Decode position of end_q bit in end_q_index,
  -- to binary representation:
  with get_in select get <=
    "000000" when "000000000000000000000000000000000000000000000001",
    "000001" when "000000000000000000000000000000000000000000000010",
    "000010" when "000000000000000000000000000000000000000000000100",
    "000011" when "000000000000000000000000000000000000000000001000",
    "000100" when "000000000000000000000000000000000000000000010000",
    "000101" when "000000000000000000000000000000000000000000100000",
    "000110" when "000000000000000000000000000000000000000001000000",
    "000111" when "000000000000000000000000000000000000000010000000",
    "001000" when "000000000000000000000000000000000000000100000000",
    "001001" when "000000000000000000000000000000000000001000000000",
    "001010" when "000000000000000000000000000000000000010000000000",
    "001011" when "000000000000000000000000000000000000100000000000",
    "001100" when "000000000000000000000000000000000001000000000000",
    "001101" when "000000000000000000000000000000000010000000000000",
    "001110" when "000000000000000000000000000000000100000000000000",
    "001111" when "000000000000000000000000000000001000000000000000",
    "010000" when "000000000000000000000000000000010000000000000000",
    "010001" when "000000000000000000000000000000100000000000000000",
    "010010" when "000000000000000000000000000001000000000000000000",
    "010011" when "000000000000000000000000000010000000000000000000",
    "010100" when "000000000000000000000000000100000000000000000000",
    "010101" when "000000000000000000000000001000000000000000000000",
    "010110" when "000000000000000000000000010000000000000000000000",
    "010111" when "000000000000000000000000100000000000000000000000",
    "011000" when "000000000000000000000001000000000000000000000000",
    "011001" when "000000000000000000000010000000000000000000000000",
    "011010" when "000000000000000000000100000000000000000000000000",
    "011011" when "000000000000000000001000000000000000000000000000",
    "011100" when "000000000000000000010000000000000000000000000000",
    "011101" when "000000000000000000100000000000000000000000000000",
    "011110" when "000000000000000001000000000000000000000000000000",
    "011111" when "000000000000000010000000000000000000000000000000",
    "100000" when "000000000000000100000000000000000000000000000000",
    "100001" when "000000000000001000000000000000000000000000000000",
    "100010" when "000000000000010000000000000000000000000000000000",
    "100011" when "000000000000100000000000000000000000000000000000",
    "100100" when "000000000001000000000000000000000000000000000000",
    "100101" when "000000000010000000000000000000000000000000000000",
    "100110" when "000000000100000000000000000000000000000000000000",
    "100111" when "000000001000000000000000000000000000000000000000",
    "101000" when "000000010000000000000000000000000000000000000000",
    "101001" when "000000100000000000000000000000000000000000000000",
    "101010" when "000001000000000000000000000000000000000000000000",
    "101011" when "000010000000000000000000000000000000000000000000",
    "101100" when "000100000000000000000000000000000000000000000000",
    "101101" when "001000000000000000000000000000000000000000000000",
    "101110" when "010000000000000000000000000000000000000000000000",
    "101111" when "100000000000000000000000000000000000000000000000",
    "XXXXXX" when others;
end architecture RTL48;
```

decode_r.vhd

```vhdl
entity decode_r is
  port(
    clk      : in  std_logic;
    rst      : in  std_logic;
    end_q_in : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    data_in  : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    data_out : out std_logic_vector(7 downto 0)
  );
end entity decode_r;

-- Top module for mux array that produces 'r':
architecture RTL of decode_r is
  type data_arr is array (DEC_REG_SIZE - 1 downto 0)
    of std_logic_vector(7 downto 0);
  signal data : data_arr;

begin
  r_MUX_array0 : for i in 7 downto 0 generate
    r_MUX_array1 : for j in DEC_REG_SIZE - 2 downto 0 generate
      r_MUXs : entity work.decode_r_mux port map(
          clk       => clk,
          data_in   => data_in(j),
          active    => end_q_in((j - i - 1) mod DEC_REG_SIZE),
          data_out  => data(j)(i),
          prev_data => data(j + 1)(i)
        );
    end generate;
    r_MUX : entity work.decode_r_mux port map(
        clk       => clk,
        data_in   => data_in(DEC_REG_SIZE-1),
        active => end_q_in((DEC_REG_SIZE-1 - i - 1) mod DEC_REG_SIZE),
        data_out  => data(DEC_REG_SIZE-1)(i),
        prev_data => '0'
      );
  end generate;
  data_out <= data(0);
end architecture RTL;
```

decode_r_mux.vhd

```
entity decode_r_mux is
  port(
    clk       : in  std_logic;
    data_in   : in  std_logic;
    prev_data : in  std_logic;
    active    : in  std_logic;
    data_out  : out std_logic
  );
end entity decode_r_mux;

-- MUX to select 'r' in data register based on active signal
-- which is given by end_r bit.
architecture RTL of decode_r_mux is
begin
  process(data_in, prev_data, active) is
  begin
    if active = '1' then
      data_out <= data_in;
    else
      data_out <= prev_data;
    end if;
  end process;
end architecture RTL;
```

comb_qr.vhd

```
entity comb_qr is
  port(
    clk      : in  std_logic;
    rst      : in  std_logic;
    q        : in  std_logic_vector(2 downto 0);
    r        : in  std_logic_vector(7 downto 0);
    k        : in  std_logic_vector(3 downto 0);
    data_out : out std_logic_vector(7 downto 0)
  );
end entity comb_qr;

architecture RTL of comb_qr is
-- function to reverse any vector
  function rev_lv(a : in std_logic_vector) return std_logic_vector is
    variable result : std_logic_vector(a'RANGE);
    alias aa        : std_logic_vector(a'REVERSE_RANGE) is a;
  begin
    for i in aa'RANGE loop
      result(i) := aa(i);
    end loop;
    return result;
  end;
begin

  -- Combine 'q' and 'r' based on 'k':
  qr : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        data_out <= (others => '0');
      else
        if k = 0 then
          data_out <= "00000" & q;
        elsif k = 1 then
          data_out <= "0000" & q & r(0);
        elsif k = 2 then
          data_out <= "000" & q & rev_lv(r(1 downto 0));
        elsif k = 3 then
          data_out <= "00" & q & rev_lv(r(2 downto 0));
        elsif k = 4 then
          data_out <= "0" & q & rev_lv(r(3 downto 0));
        elsif k = 5 then
          data_out <= q & rev_lv(r(4 downto 0));
        elsif k = 6 then
          data_out <= q(1 downto 0) & rev_lv(r(5 downto 0));
        elsif k = 7 then
          data_out <= q(0) & rev_lv(r(6 downto 0));
        else
          data_out <= rev_lv(r);
        end if;
      end if;
    end if;
  end process qr;
end architecture RTL;
```

data_in_reg.vhd

```
entity data_in_reg is
  port(
    clk       : in  std_logic;
    rst       : in  std_logic;
    data_in   : in  std_logic_vector(DATA_IN_REG_SIZE - 1 downto 0);
    ready_in  : in  std_logic_vector(DATA_IN_REG_NUM - 1 downto 0);
    data_out  : out std_logic_vector(DEC_REG_SIZE - 1 downto 0)
  );
end entity data_in_reg;

architecture RTL of data_in_reg is
  type data_arr is array (0 to DATA_IN_REG_NUM - 1)
    of std_logic_vector(DATA_IN_REG_SIZE - 1 downto 0);
  signal data_reg : data_arr;
begin

  -- Output register array:
  clocked : process(clk, data_reg) is
  begin
    for n in 1 to DATA_IN_REG_NUM loop
      data_out(
        DATA_IN_REG_SIZE * n - 1 downto DATA_IN_REG_SIZE * (n - 1)
      ) <= data_reg(n - 1);
    end loop;
  end process clocked;

  process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        for n in 0 to DATA_IN_REG_NUM - 1 loop
          -- Initial data to start golomb decoder:
          data_reg(n) <= "1111111101111111";
        end loop;
      else

        -- Select part of register to update:
        for n in 0 to DATA_IN_REG_NUM - 1 loop
          if ready_in(n) = '1' then
            data_reg(n) <= data_in;
          else
            data_reg(n) <= data_reg(n);
          end if;
        end loop;
      end if;
    end if;
  end process;
end architecture RTL;
```

load_data_in.vhd

```
entity load_data_in is
  port(
    clk     : in  std_logic;
    rst     : in  std_logic;
    get_in  : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    get_out : in  std_logic_vector(DEC_REG_SIZE - 1 downto 0);
    ready   : out std_logic_vector(DATA_IN_REG_NUM - 1 downto 0)
  );
end entity load_data_in;

architecture RTL of load_data_in is
  type bank_arr is array (DATA_IN_REG_NUM - 1 downto 0) of std_logic;
  signal get_out_bank : bank_arr;
  signal get_in_bank  : bank_arr;
  signal get_in_t     : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal get_out_t    : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal ready_t   : std_logic_vector(DATA_IN_REG_NUM - 1 downto 0);
begin
  check_ready : process(clk, get_in, get_out, get_in_bank,
    get_out_bank, get_in_t, get_out_t, ready_t, rst
  ) is
  begin
    for n in 0 to DATA_IN_REG_NUM - 1 loop

      -- Reset and update signals:
      if rising_edge(clk) then
        if rst = '1' then
          get_in_t  <=
            "000000000000000000000000000000000000000000000001";
          get_out_t <=
            "000000000000000000000000000000000000000000000001";
        else
          get_in_t  <= get_in;
          get_out_t <= get_out;
        end if;
      end if;

      -- Use banks to determine where get_bit and end_q_bit
      -- is positioned:
      if get_in_t(
        DATA_IN_REG_SIZE * (n + 1) - 1 downto DATA_IN_REG_SIZE * n
      ) = 0 then
        get_in_bank(n) <= '0';
      else
        get_in_bank(n) <= '1';
      end if;

      if get_out_t(
        DATA_IN_REG_SIZE * (n + 1) - 1 downto DATA_IN_REG_SIZE * n
      ) = 0 then
        get_out_bank(n) <= '0';
      else
        get_out_bank(n) <= '1';
      end if;
      -- Check if get_bit and next_bit is in two adjacent registers:
      ready_t(n) <=
        get_in_bank(n) and get_out_bank((n + 1) mod DATA_IN_REG_NUM);

      -- Output signals:
      if rst = '1' then
        ready <= "000";
      else
        ready <= ready_t;
      end if;
    end loop;
  end process check_ready;
end architecture RTL;
```

gol_dec_top.vhd

```
entity gol_dec_top is
  port(
    clk       : in  std_logic;
    rst       : in  std_logic;
    -- write data:
    data_out  : out std_logic_vector(7 downto 0);
    k_out_t3  : out std_logic_vector(3 downto 0);
    -- get data:
    k_in      : in  std_logic_vector(3 downto 0);
    k_in_bit  : in  std_logic_vector(8 downto 0);
    data_in   : in  std_logic_vector(DATA_IN_REG_SIZE - 1 downto 0);
    ready_out : out std_logic
  );
end entity gol_dec_top;

architecture RTL of gol_dec_top is
  signal get_in      : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal end_q_out   : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal get_out     : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal end_q_in    : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal q           : std_logic_vector(2 downto 0);
  signal r           : std_logic_vector(7 downto 0);
  signal get_in_prev : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal r_temp1     : std_logic_vector(7 downto 0);
  signal k_out       : std_logic_vector(3 downto 0);
  signal k_in_t1     : std_logic_vector(3 downto 0);
  signal k_in_t2     : std_logic_vector(3 downto 0);
  signal ready       : std_logic_vector(DATA_IN_REG_NUM - 1 downto 0);
  signal data        : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal data_r      : std_logic_vector(DEC_REG_SIZE - 1 downto 0);
  signal k_c1        : std_logic_vector(3 downto 0);
  signal k_bit_c1    : std_logic_vector(8 downto 0);
  signal k_out_t1    : std_logic_vector(3 downto 0);
  signal k_out_t2    : std_logic_vector(3 downto 0);

begin

  -- Pipeline signals:
  delay_signals : process(clk) is
  begin
    if rising_edge(clk) then
      if rst = '1' then
        get_in(DEC_REG_SIZE - 1 downto 1)      <= (others => '0');
        get_in(0)                              <= '1';
        get_in_prev(DEC_REG_SIZE - 1 downto 1) <= (others => '0');
        get_in_prev(0)                         <= '1';
        end_q_in                               <= (others => '0');

      else
        get_in      <= get_out;
        end_q_in    <= end_q_out;
        get_in_prev <= get_in;
      end if;
      r_temp1  <= r;
    k_in_t1  <= k_out;
    k_in_t2  <= k_in_t1;
    data_r   <= data;
    k_out_t1 <= k_out;
    k_out_t2 <= k_out_t1;
    k_out_t3 <= k_out_t2;
  end if;
end process delay_signals;

-- Instantiate modules:
data_in_reg_module : entity work.data_in_reg port map(
    clk      => clk,
    rst      => rst,
    data_in  => data_in,
    ready_in => ready,
    data_out => data
  );

delay_k : process(clk) is

begin
  if rising_edge(clk) then
    if rst = '1' then
      k_c1     <= "1000";        -- Initial value of 'k':
      k_bit_c1 <= "100000000";
    else
      k_c1     <= k_in;
      k_bit_c1 <= k_in_bit;
    end if;
  end if;
end process delay_k;

count_module : entity work.update_registers port map(
    clk       => clk,
    k_in      => k_c1,
    k_in_bit  => k_bit_c1,
    get_in    => get_in,
    data_in   => data,
    end_q_out => end_q_out,
    get_out   => get_out,
    k_out     => k_out
  );
load_data_in_module : entity work.load_data_in port map(
    clk     => clk,
    rst     => rst,
    get_in  => get_in,
    get_out => get_out,
    ready   => ready
  );
ready_out <= ready(0) or ready(1) or ready(2);

decode_q_module : entity work.decode_q port map(
    clk      => clk,
    rst      => rst,
    end_q_in => end_q_in,
    get_in   => get_in_prev,
    q_out    => q
```

```
        );
      decode_r_module : entity work.decode_r port map(
          clk      => clk,
          rst      => rst,
          end_q_in => end_q_in,
          data_in  => data_r,
          data_out => r
        );
      comb_qr_module : entity work.comb_qr port map(
          clk      => clk,
          rst      => rst,
          q        => q,
          r        => r_temp1,
          k        => k_in_t2,
          data_out => data_out
        );
    end architecture RTL;
```

A.7 Toplevel - decoder

dec_top.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.defs.all;

entity dec_top is
  port(
    clk       : in  std_logic;
    rst       : in  std_logic;
    data_in   : in  std_logic_vector(15 downto 0);
    new_frame : in  std_logic;
    ready     : out std_logic;
    ready_gol : out std_logic;
    data_out  : out std_logic_vector(7 downto 0)
  );
end entity dec_top;

architecture RTL of dec_top is
  --index
  signal index_rst : std_logic;
  signal B_i       : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal C_i       : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal D_i       : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal I_i       : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal index_out : std_logic_vector(CTXT_ADDR_SIZE - 1 downto 0);
  signal sign_out  : std_logic;

  --k
  signal loco_k_rst : std_logic;

  signal A_k_out    : std_logic_vector(A_LEN - 1 downto 0);
  signal N_k_out    : std_logic_vector(N_LEN - 1 downto 0);
  signal k_out      : std_logic_vector(8 downto 0);
  signal k_less_out : std_logic_vector(8 downto 0);
  signal k_int_out  : std_logic_vector(3 downto 0);

  --loco
  signal loco_rst : std_logic;
  signal A_out      : std_logic_vector(A_LEN - 1 downto 0);
  signal B_out      : std_logic_vector(B_LEN - 1 downto 0);
  signal C_out      : std_logic_vector(C_LEN - 1 downto 0);
  signal N_out      : std_logic_vector(N_LEN - 1 downto 0);
  signal B_l        : std_logic_vector(SMPL_LEN - 1 downto 0);
  signal sign_d     : std_logic;
  signal loco_w_rst : std_logic;
  signal index_d2   : std_logic_vector(7 downto 0);
  signal A_in       : std_logic_vector(A_LEN - 1 downto 0);
  signal B_in       : std_logic_vector(B_LEN - 1 downto 0);
  signal C_in       : std_logic_vector(C_LEN - 1 downto 0);
  signal N_in       : std_logic_vector(N_LEN - 1 downto 0);
```

```
    --gol
    signal gol_rst   : std_logic;
    signal k_int_d   : std_logic_vector(3 downto 0);

    --context
    signal ctxt_addr_rst : std_logic_vector(CTXT_ADDR_SIZE - 1 downto 0);
    signal ctxt_rst      : std_logic;

    --output
    signal data_out_c : std_logic_vector(7 downto 0);
    signal cache_rst    : std_logic;
    signal not_rst      : std_logic;
    signal index_d1     : std_logic_vector(7 downto 0);
    signal data_out_gol : std_logic_vector(SMPL_LEN - 1 downto 0);
    signal k_out_gol    : std_logic_vector(3 downto 0);

  begin
    not_rst <= not rst;

    clocked_in_out : process(clk, rst) is
    begin
      if rising_edge(clk) then
        if rst = '1' then
          data_out <= (others => '0');
        else
          data_out <= data_out_c;
        end if;
      end if;
    end process clocked_in_out;

    index_module : entity work.index port map(
        clk       => clk,
        rst       => index_rst,
        B_cache   => B_i,
        C_cache   => C_i,
        D_cache   => D_i,
        I_cache   => I_i,
        index_out => index_out,
        sign_out  => sign_out
      );
    find_k_module : entity work.find_k port map(
        clk        => clk,
        rst        => loco_k_rst,
        A          => A_k_out,
        N          => N_k_out,
        k_out      => k_out,
        k_less_out => k_less_out,
        k_int_out  => k_int_out
      );
    modeling_module : entity work.dec_modeling port map(
        clk        => clk,
        rst        => loco_rst,
        loco_w_rst => loco_w_rst,
        data       => data_out_gol,
        sign       => sign_d,
        P          => B_l,
    k_int_in   => k_out_gol,
    k_orig => k_int_d,
    A          => A_out,
    B          => B_out,
    C          => C_out,
    N          => N_out,
    A_out      => A_in,
    B_out      => B_in,
    C_out      => C_in,
    N_out      => N_in,
    Q_out      => data_out_c
  );
gol_module : entity work.gol_dec_top port map(
    clk       => clk,
    rst       => gol_rst,
    data_out  => data_out_gol,
    k_out_t3  => k_out_gol,
    k_in      => k_int_out,
    k_in_bit  => k_out,
    data_in   => data_in,
    ready_out => ready_gol
  );
ctxt_reg_module : entity work.ctxt_reg port map(
    clk          => clk,
    rst          => rst,
    write_0      => ctxt_rst,
    write0_addr  => ctxt_addr_rst,
    read_index   => index_d1,
    read_k_index => index_out,
    write_index  => index_d2,
    A_in         => A_in,
    B_in         => B_in,
    C_in         => C_in,
    N_in         => N_in,
    A_out        => A_out,
    A_k_out      => A_k_out,
    B_out        => B_out,
    C_out        => C_out,
    N_out        => N_out,
    N_k_out      => N_k_out
  );
cache_module : entity work.line_cache port map(
    clk        => clk,
    shift      => not_rst,
    write_0    => cache_rst,
    sr_in      => data_out_c,
    sr_tap_B   => B_i,
    sr_tap_C   => C_i,
    sr_tap_D   => D_i,
    sr_tap_I   => I_i,
    sr_tap_B_l => B_l
  );
delay_module : entity work.dec_signal_delays port map(
    clk        => clk,
    rst        => rst,
    index_in   => index_out,
    sign_in    => sign_out,
            k_less_in  => k_less_out,
            k_in       => k_out,
            k_int_in   => k_int_out,
            index_out1 => index_d1,
            index_out2 => index_d2,
            sign_out   => sign_d,
            k_int_out  => k_int_d);
        reset_module : entity work.dec_reset port map(
            clk        => clk,
            rst        => rst,
            new_frame  => new_frame,
            ready      => ready,
            ctxt_addr  => ctxt_addr_rst,
            ctxt_rst   => ctxt_rst,
            cache_rst  => cache_rst,
            index_rst  => index_rst,
            loco_k_rst => loco_k_rst,
            loco_rst   => loco_rst,
            loco_w_rst => loco_w_rst,
            gol_rst    => gol_rst
          );
      end architecture RTL;
```

A.8 Embedded memory

line_cache.vhd

```
entity line_cache IS
  port(
    clk        : in  STD_LOGIC;
    shift      : in  STD_LOGIC;
    write_0    : in  std_logic;
    sr_in      : in  STD_LOGIC_VECTOR(SMPL_LEN - 1 downto 0);
    sr_tap_B   : out STD_LOGIC_VECTOR(SMPL_LEN - 1 downto 0);
    sr_tap_C   : out STD_LOGIC_VECTOR(SMPL_LEN - 1 downto 0);
    sr_tap_D   : out STD_LOGIC_VECTOR(SMPL_LEN - 1 downto 0);
    sr_tap_I   : out STD_LOGIC_VECTOR(SMPL_LEN - 1 downto 0);
    sr_tap_B_l : out STD_LOGIC_VECTOR(SMPL_LEN - 1 downto 0)
  );
end line_cache;

architecture arch OF line_cache IS
  --  SUBTYPE sr_width IS STD_LOGIC_VECTOR(SMPL_LEN-1 DOWNTO 0);
  type sr_length is array (CACHE_SIZE - 1 downto 0)
    of STD_LOGIC_VECTOR(SMPL_LEN - 1 downto 0);

  signal sr : sr_length;

  signal data : STD_LOGIC_VECTOR(SMPL_LEN - 1 downto 0);

begin
  write0 : process(sr_in, write_0) is
  begin
    if write_0 = '1' then
      data <= (others => '0');
    else
      data <= sr_in;
    end if;
  end process write0;

  process(clk)
  begin
    if (clk'EVENT and clk = '1') then
      if (shift = '1') then
        sr(CACHE_SIZE - 1 downto 1) <= sr(CACHE_SIZE - 2 downto 0);
        sr(0)                       <= data;
      end if;
    end IF;
  end process;
  sr_tap_B   <= sr(CACHE_SIZE - 1920 - CACHE_INDEX_OFFSET - 1);
  sr_tap_C   <= sr(CACHE_SIZE - 1920 - CACHE_INDEX_OFFSET);
  sr_tap_D   <= sr(CACHE_SIZE - 1920 - CACHE_INDEX_OFFSET - 2);
  sr_tap_I   <= sr(CACHE_SIZE - CACHE_INDEX_OFFSET - 1);
  sr_tap_B_l <= sr(CACHE_SIZE - 1920 - 1);
end arch;
```

ctxt_reg.vhd

```
entity ctxt_reg is
  port(
    clk          : in  std_logic;
    rst          : in  std_logic;
    write_0      : in  std_logic;
    write0_addr  : in  std_logic_vector(7 downto 0);

    read_index   : in  std_logic_vector(7 downto 0);
    read_k_index : in  std_logic_vector(7 downto 0);
    write_index  : in  std_logic_vector(7 downto 0);

    A_in         : in  std_logic_vector(A_LEN - 1 downto 0);
    B_in         : in  std_logic_vector(B_LEN - 1 downto 0);
    C_in         : in  std_logic_vector(C_LEN - 1 downto 0);
    N_in         : in  std_logic_vector(N_LEN - 1 downto 0);

    A_out        : out std_logic_vector(A_LEN - 1 downto 0);
    A_k_out      : out std_logic_vector(A_LEN - 1 downto 0);
    B_out        : out std_logic_vector(B_LEN - 1 downto 0);
    C_out        : out std_logic_vector(C_LEN - 1 downto 0);
    N_out        : out std_logic_vector(N_LEN - 1 downto 0);
    N_k_out      : out std_logic_vector(N_LEN - 1 downto 0)
  );
end entity ctxt_reg;

architecture RTL of ctxt_reg is
  type Aarr is array (CTXT_INDEX_RANGE - 1 downto 0)
    of std_logic_vector(A_LEN - 1 downto 0);
  type Barr is array (CTXT_INDEX_RANGE - 1 downto 0)
    of std_logic_vector(B_LEN - 1 downto 0);
  type Carr is array (CTXT_INDEX_RANGE - 1 downto 0)
    of std_logic_vector(C_LEN - 1 downto 0);
  type Narr is array (CTXT_INDEX_RANGE - 1 downto 0)
    of std_logic_vector(N_LEN - 1 downto 0);

  signal A          : Aarr;
  signal B          : Barr;
  signal C          : Carr;
  signal N          : Narr;
  signal dataA      : std_logic_vector(A_LEN - 1 downto 0);
  signal dataB      : std_logic_vector(B_LEN - 1 downto 0);
  signal dataC      : std_logic_vector(C_LEN - 1 downto 0);
  signal dataN      : std_logic_vector(N_LEN - 1 downto 0);
  signal write_addr : std_logic_vector(7 downto 0);

begin
  write0 : process(
    A_in, B_in, C_in, N_in, write_0, write0_addr, write_index
  ) is
  begin
    if write_0 = '1' then
      dataA      <= (others => '0');
      dataB      <= (others => '0');
      dataC      <= (others => '0');
      dataN      <= (others => '0');
      write_addr <= write0_addr;
    else
      dataA      <= A_in;
      dataB      <= B_in;
      dataC      <= C_in;
      dataN      <= N_in;
      write_addr <= write_index;
    end if;
  end process write0;

  ctxt_mem : process(clk) is
  begin
    if rising_edge(clk) then
      A_out   <= A(conv_integer(unsigned(read_index)));
      A_k_out <= A(conv_integer(unsigned(read_k_index)));
      B_out   <= B(conv_integer(unsigned(read_index)));
      C_out   <= C(conv_integer(unsigned(read_index)));
      N_out   <= N(conv_integer(unsigned(read_index)));
      N_k_out <= N(conv_integer(unsigned(read_k_index)));

      A(conv_integer(unsigned(write_addr))) <= dataA;
      B(conv_integer(unsigned(write_addr))) <= dataB;
      C(conv_integer(unsigned(write_addr))) <= dataC;
      N(conv_integer(unsigned(write_addr))) <= dataN;
    end if;
  end process ctxt_mem;
end architecture RTL;
```

A.9 Testbenches

Encoder

```
library ieee;
library modelsim_lib;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;
use std.textio.all;
use modelsim_lib.util.all;
use work.defs.all;

-- Encoder testbench:
-- Instantiates 3 encoders (R,G,B)
-- Reads input from a single (proprietary) input file,
-- built from 24b bitmap image data
-- Produces 3 output files, one per color component,
-- containing compressed data
-- Compression rate is calculated and displayed at the end

entity enc_tb is
  generic(
    constant IN_WIDTH  : integer := 8;
    constant OUT_WIDTH : integer := 16;
    -- Half the clock period.
    -- The frequency will be 1/(2*PERIOD) = 100 MHz
    constant PERIOD    : time    := 50 ns;

    -- file name affix ( Note:  the simulation expects the file
    --                        to have 'vhdl' as an suffix)
    constant fil       : string  := "t7";

    -- Folder where input file is located
    constant path      : string  := "../";

    -- Used to scale the compression calculation
    -- Should be the number of pixels in the input
    constant size      : real    := 1920.0 * 1080.0
  );
end entity enc_tb;

architecture RTL of enc_tb is
  signal clk      : std_logic;
  signal rst      : std_logic;
  signal inputR   : std_logic_vector(IN_WIDTH - 1 downto 0);
  signal inputG   : std_logic_vector(IN_WIDTH - 1 downto 0);
  signal inputB   : std_logic_vector(IN_WIDTH - 1 downto 0);
  signal newFrame : std_logic;
  signal rdyR     : std_logic;
  signal rdyG     : std_logic;
  signal rdyB     : std_logic;

  signal outputR : std_logic_vector(OUT_WIDTH - 1 downto 0);
  signal outputG : std_logic_vector(OUT_WIDTH - 1 downto 0);
  signal outputB : std_logic_vector(OUT_WIDTH - 1 downto 0);

  signal writeR : std_logic;
  signal writeG : std_logic;
  signal writeB : std_logic;

  -- Enc -> wbuf connect
  signal encRwbufR : std_logic_vector(GOL_MAX_LEN - 1 downto 0);
  signal encGwbufG : std_logic_vector(GOL_MAX_LEN - 1 downto 0);
  signal encBwbufB : std_logic_vector(GOL_MAX_LEN - 1 downto 0);

  signal sizeOutR : std_logic_vector(3 downto 0);
  signal sizeOutG : std_logic_vector(3 downto 0);
  signal sizeOutB : std_logic_vector(3 downto 0);

  signal wbufEmptyR : std_logic;
  signal wbufEmptyG : std_logic;
  signal wbufEmptyB : std_logic;

  signal wbufEn : std_logic;

  shared variable numPixelsIn         : integer := 0;
  shared variable bitsR, bitsG, bitsB : real    := 0.0;
  shared variable bR, bG, bB          : integer := 0;

  signal qR, qG, qB : std_logic_vector(7 downto 0);

  signal delayWbufEn : std_logic := '0';
  signal writeQ      : std_logic := '0';

begin

  -- ENCODER INIT
  encR : entity work.enc_top
    port map(clk       => clk,
           rst       => rst,
           data_in   => inputR,
           new_frame => newFrame,
           ready     => rdyR,
           data_out  => encRwbufR,
           size_out  => sizeOutR
    );
  encG : entity work.enc_top
    port map(clk       => clk,
           rst       => rst,
           data_in   => inputG,
           new_frame => newFrame,
           ready     => rdyG,
           data_out  => encGwbufG,
           size_out  => sizeOutG
    );

  encB : entity work.enc_top
    port map(clk       => clk,
           rst       => rst,
           data_in   => inputB,
           new_frame => newFrame,
           ready     => rdyB,
           data_out  => encBwbufB,
           size_out  => sizeOutB
  );

-- GOLOMB PACKER (WRITE BUFFER) INIT
wbufR : entity work.wbuf
  port map(clk         => clk,
           rst         => rst,
           in_wbuf     => encRwbufR,
           word_length => sizeOutR,
           wbuf_enable => wbufEn,
           out_wbuf    => outputR,
           wbuf_empty  => wbufEmptyR,
           wbuf_write  => writeR);
wbufG : entity work.wbuf
  port map(clk         => clk,
           rst         => rst,
           in_wbuf     => encGwbufG,
           word_length => sizeOutG,
           wbuf_enable => wbufEn,
           out_wbuf    => outputG,
           wbuf_empty  => wbufEmptyG,
           wbuf_write  => writeG);
wbufB : entity work.wbuf
  port map(clk         => clk,
           rst         => rst,
           in_wbuf     => encBwbufB,
           word_length => sizeOutB,
           wbuf_enable => wbufEn,
           out_wbuf    => outputB,
           wbuf_empty  => wbufEmptyB,
           wbuf_write  => writeB);

clock_generator : process
begin
  clk <= '1';
  wait for PERIOD / 2;
  clk <= '0';
  wait for PERIOD / 2;
end process;

--/ ---------------------------------
--/ CONNECT SIGNALS ACROSS DIFFERENT
--/ MODULES
--/ Used to probe entity signals that aren't
--/ connected directly to ports
drive_sig_process : process
begin
  init_signal_driver("/enc_tb/encR/Q", "qR");
  init_signal_driver("/enc_tb/encG/Q", "qG");
  init_signal_driver("/enc_tb/encB/Q", "qB");
  wait;
end process drive_sig_process;
-- ---------------------------------
```

```
--/ ---------------------------------
--/ CONTROLLING WHEN WBUF IS ENABLED
--/ AND WHEN TO START WRITING Q's
--/ The Golomb packer is enabled 1 cycle before the output is valid
--/ and stopped when the last valid output has passed.
--/ writeQ is only used to probe the Q signals within the encoder
--/ entities for debugging convenience.
wbufEnable : process(clk) is
  variable count : std_logic_vector(3 downto 0) := (others => '0');
begin
  if (rising_edge(clk)) then
    if (delayWbufEn = '1') then
      if (count > "0101") then
        wbufEn <= '1';
        writeQ <= '1';
      elsif (count > "0011") then
        writeQ <= '1';
        wbufEn <= '0';
        count  := count + "1";
      else
        wbufEn <= '0';
        writeQ <= '0';
        count  := count + "1";
      end if;
    else
      if (count > "0101") then
        wbufEn <= '1';
        writeQ <= '1';
        count  := count - "1";
      elsif (count > "0001") then
        wbufEn <= '1';
        writeQ <= '0';
        count  := count - "1";
      else
        wbufEn <= '0';
        writeQ <= '0';
      end if;
    end if;
  end if;
end process;
-- ---------------------------------
--/ PROCESS CONTROLLING INPUT
--/ Input is read from the file defined with the constants
--/ Loops new input every clock cycle until the input file
--/ is empty.
--/ Disable signal is triggered, and statistics are calculated

inputctrl : process
  file inn : text is in path & fil & "vhdl";
  variable buf_in, buf_nSymb : line;
  variable inV               : std_logic_vector(23 downto 0)
                                              := (others => '0');
  variable cR, cG, cB, cT    : real;
  variable overhead          : integer;
begin
  rst <= '1';
  wait for PERIOD;
```

```
inputR <= (others => '0');
inputG <= (others => '0');
inputB <= (others => '0');
rst    <= '0';

wait for PERIOD;
newFrame <= '1';
while (rdyR = '0' or rdyG = '0' or rdyB = '0') loop
  wait for PERIOD;
end loop;
wait for PERIOD;
wait for 5 ns;
newFrame <= '0';
WAIT FOR 45 ns;
wait for 9 * PERIOD;
delayWbufEn <= '1';

while not ENDFILE(inn) loop
  -- Get line from input file
  READLINE(inn, buf_in);
  -- Save value to variable
  HREAD(buf_in, inV);

  -- Spectral decorrelation of the inputs.
  inputR <= inV(23 downto 16) - inV(15 downto 8);
  inputG <= inV(15 downto 8);
  inputB <= inV(7 downto 0) - inV(15 downto 8);

  -- TESTING UTEN R-G,G,B-G DEBUG!!!
  --    inputR <= inV(23 downto 16);
  --    inputG <= inV(15 downto 8);
  --    inputB <= inV(7 downto 0);

  -- Count pixels
  numPixelsIn := numPixelsIn + 1;
  wait for PERIOD;
end loop;
inputR <= (others => '1');
inputG <= (others => '1');
inputB <= (others => '1');
wait for PERIOD;
delaywbufEn <= '0';

wait for 10 * PERIOD;

-- --------------------------------------
-- END SIMULATION : CALCULATE STATISTICS
-- Compression per component and total
-- Overhead added for different packet
-- sizes. (large packets, not golomb packets)
-- --------------------------------------
cR       := bitsR / real'(size);
cG       := bitsG / real'(size);
cB       := bitsB / real'(size);
overhead := (bR + bG + bB) * 2 / 16 / 8000;
cT       := (bitsR + bitsG + bitsB) / real'(size * 3.0);
write(OUTPUT, "Ending␣simulation." & LF);
```

```
    write(OUTPUT, "Simulation␣statistics␣for␣" & fil & ":" & LF);
    write(OUTPUT, "␣-----------------␣" & LF);
    write(OUTPUT, "R␣␣␣:␣" & real'image(cR) & LF);
    write(OUTPUT, "G␣␣␣:␣" & real'image(cG) & LF);
    write(OUTPUT, "B␣␣␣:␣" & real'image(cB) & LF);
    write(OUTPUT, "␣-␣-␣-␣-␣-␣-␣-␣-␣-␣" & LF);
    write(OUTPUT, "Tot␣:␣" & real'image(cT) & LF);
    write(OUTPUT, "␣-␣-␣-␣-␣-␣-␣-␣-␣-␣" & LF);
    write(OUTPUT, "Head␣16␣:␣" & natural'image(overhead)&"␣KB" & LF);
    overhead := overhead / 2;
    write(OUTPUT, "Head␣32␣:␣" & natural'image(overhead)&"␣KB" & LF);
    overhead := overhead / 2;
    write(OUTPUT, "Head␣64␣:␣" & natural'image(overhead)&"␣KB" & LF);
    overhead := overhead / 2;
    write(OUTPUT, "Head␣128:␣" & natural'image(overhead)&"␣KB" & LF);
    overhead := overhead / 2;
    write(OUTPUT, "Head␣256:␣" & natural'image(overhead)&"␣KB" & LF);

    write(OUTPUT, "␣-----------------␣" & LF);
    write(OUTPUT, "Simulation␣complete.");
    report "" severity failure;
-- ----------------------------------
end process;

-- ---------------------------------
--/ PROCESS CONTROLLING OUTPUT
--/ Output is written every time the write signals are high
--/ 3 output files, 1 per color component.
--/ The total number of bits outputted are counted for use
--/ in compression calculation.
--/ Additional information is written for information (packet order)
--/ and debugging (probed Q from within encoder entities) purposes.

writectrl : process(clk)
  file outR : text open WRITE_MODE
    is path & fil & "\" & fil & "_outR";
  file outG : text open WRITE_MODE
    is path & fil & "\" & fil & "_outG";
  file outB : text open WRITE_MODE
    is path & fil & "\" & fil & "_outB";

  file wrorder : text open WRITE_MODE
    is path & fil & "\" & fil & "_wrorder";

  file qoutR : text open WRITE_MODE
    is path & fil & "\" & fil & "_qR";
  file qoutG : text open WRITE_MODE
    is path & fil & "\" & fil & "_qG";
  file qoutB : text open WRITE_MODE
    is path & fil & "\" & fil & "_qB";

  variable buf_outR, buf_outG, buf_outB, buf_qR,
    buf_qG, buf_qB, buf_wOrdr : line;

begin
  if (rising_edge(clk)) then
    if (writeR = '1') then
```

```
          WRITE(buf_outR, outputR(15 downto 0), RIGHT, OUT_WIDTH / 2);
          WRITELINE(outR, buf_outR);
          bitsR := bitsR + 16.0;
          bR    := bR + 16;

          WRITE(buf_wOrdr, 1);
          WRITELINE(wrorder, buf_wOrdr);
        end if;
        if (writeG = '1') then
          WRITE(buf_outG, outputG(15 downto 0), RIGHT, OUT_WIDTH / 2);
          WRITELINE(outG, buf_outG);
          bitsG := bitsG + 16.0;
          bG    := bG + 16;

          WRITE(buf_wOrdr, 2);
          WRITELINE(wrorder, buf_wOrdr);
        end if;
        if (writeB = '1') then
          WRITE(buf_outB, outputB(15 downto 0), RIGHT, OUT_WIDTH / 2);
          WRITELINE(outB, buf_outB);
          bitsB := bitsB + 16.0;
          bB    := bB + 16;

          WRITE(buf_wOrdr, 3);
          WRITELINE(wrorder, buf_wOrdr);
        end if;

        if (writeQ = '1') then
          HWRITE(buf_qR, qR, RIGHT, 2);
          WRITELINE(qoutR, buf_qR);

          HWRITE(buf_qG, qG, RIGHT, 2);
          WRITELINE(qoutG, buf_qG);

          HWRITE(buf_qB, qB, RIGHT, 2);
          WRITELINE(qoutB, buf_qB);
        end if;

      end if;
    end process;
  end architecture RTL;
```

Decoder

```
library ieee;
library modelsim_lib;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;
use std.textio.all;
use modelsim_lib.util.all;
use work.defs.all;

-- Decoder testbench:
-- Instantiates 3 decoders (R,G,B)
-- Reads inputs from 3 (proprietary) input files,
-- created by the Encoder testbench
-- Produces 1 output file, containing uncompressed data,
-- identical to the Encoder input file
-- No statistics are calculated.
--
-- NOTE: Input files read from 'path\fil'  ,  output file written to
-- 'path\', where 'path' and 'fil' are defined below.
-- 'fil' folder must be created manually within the 'path' folder.
-- The simulation cannot create folders.

entity dec_tb is
  generic(
    constant IN_WIDTH  : integer := 16;
    constant OUT_WIDTH : integer := 8;
    -- Half the clock period.
    -- The frequency will be 1/(2*PERIOD) = 100 MHz
    constant PERIOD    : time    := 50 ns;
    -- file name affix ( Note:  the simulation expects the file
    --                     to have 'vhdl' as an suffix)
    constant fil       : string  := "t6";

    -- Folder where input file is located
    constant path      : string  := "../";
    -- Number of pixels in frame data
    constant size      : integer := 1920 * 1080
  );
end entity dec_tb;

architecture RTL of dec_tb is
  signal clk    : std_logic;
  signal rst    : std_logic;
  signal inputR : std_logic_vector(IN_WIDTH - 1 downto 0);
  signal inputG : std_logic_vector(IN_WIDTH - 1 downto 0);
  signal inputB : std_logic_vector(IN_WIDTH - 1 downto 0);

  signal outputR : std_logic_vector(OUT_WIDTH - 1 downto 0);
  signal outputG : std_logic_vector(OUT_WIDTH - 1 downto 0);
  signal outputB : std_logic_vector(OUT_WIDTH - 1 downto 0);

  signal readR, readG, readB : std_logic;
  signal rdyR, rdyG, rdyB    : std_logic;
```

```
    signal newFrame            : std_logic;

    signal active : std_logic := '0';

    shared variable activate            : std_logic := '0';
    shared variable numPixelsIn         : integer   := 0;
    shared variable numPixelsOut        : integer   := 1;
    shared variable bitsR, bitsG, bitsB : real      := 0.0;

    -- Function that reverses a std_logic_vector
    function reverse_any_vector(a : in std_logic_vector)
      return std_logic_vector is
      variable result : std_logic_vector(a'RANGE);
      alias aa        : std_logic_vector(a'REVERSE_RANGE) is a;
    begin
      for i in aa'RANGE loop
        result(i) := aa(i);
      end loop;
      return result;
    end;

  begin

    -- DECODER INIT
    decR : entity work.dec
      port map(
        clk       => clk,
        rst       => rst,
        data_in   => inputR,
        new_frame => newFrame,
        ready     => rdyR,
        ready_gol => readR,
        data_out  => outputR
      );

    decG : entity work.dec
      port map(
        clk       => clk,
        rst       => rst,
        data_in   => inputG,
        new_frame => newFrame,
        ready     => rdyG,
        ready_gol => readG,
        data_out  => outputG
      );

    decB : entity work.dec
      port map(
        clk       => clk,
        rst       => rst,
        data_in   => inputB,
        new_frame => newFrame,
        ready     => rdyB,
        ready_gol => readB,
        data_out  => outputB
      );
```

```
--/ ---------------------------------
--/ GENERATES THE CLOCK
--/
clock_generator : process
begin
  clk <= '1';
  wait for PERIOD / 2;
  clk <= '0';
  wait for PERIOD / 2;
end process;

--/ ---------------------------------
--/ CONNECT SIGNALS ACROSS DIFFERENT
--/ MODULES
--/ Used to probe entity signals that aren't
--/ connected directly to ports
--drive_sig_process : process
--  begin
--    init_signal_driver("/dec_tb/encR/Q", "qR");
--    init_signal_driver("/dec_tb/encG/Q", "qG");
--  init_signal_driver("/dec_tb/encB/Q", "qB");
--  wait;
--end process drive_sig_process;
-- ---------------------------------


--/ ---------------------------------
--/  ACTIVATE OUTPUT WRITING
--/  Sets the active signal high at the
--/  right time after activate has been
--/  set in the input process below
act : process(clk)
  variable count : integer := 0;
begin
  if (rising_edge(clk)) then
    if (activate = '1') then
      if (count > 13) then
        active <= '1';
      else
        count := count + 1;
      end if;
    else
      active <= '0';
    end if;
  end if;
end process;

-- ---------------------------------
--/ PROCESS CONTROLLING INPUT
--/ Input is read from the files defined with the constant 'fil'
--/ Waits until decoder module is ready, then sets input,
--/ listens to 'read' port, updates inputs when requested.
--/ Loops as long as at least one input file is not empty
--/ Leaves loop when files are empty, waits for Output process
--/ to finish and end the simulation.

inputctrl : process
```

```
    file infileR : text is in path & fil & "\" & fil & "_outR";
    file infileG : text is in path & fil & "\" & fil & "_outG";
    file infileB : text is in path & fil & "\" & fil & "_outB";

    file readorder : text open WRITE_MODE
      is path & fil & "\" & fil & "_readorder";

    variable buf_inR, buf_inG, buf_inB, buf_nSymb, buf_rOrdr
    : line;
    variable inR, inG, inB
    : std_logic_vector(IN_WIDTH - 1 downto 0) := (others => '0');
    variable readVector
    : std_logic_vector(2 downto 0)            := "000";
  begin
    rst <= '1';
    wait for PERIOD;
    inputR <= (others => '0');
    inputG <= (others => '0');
    inputB <= (others => '0');
    rst    <= '0';

    wait for 5 * PERIOD;
    newFrame <= '1';
    READLINE(infileR, buf_inR);
    READ(buf_inR, inR);
    readVector(2) := '1';

    READLINE(infileG, buf_inG);
    READ(buf_inG, inG);
    readVector(1) := '1';

    READLINE(infileB, buf_inB);
    READ(buf_inB, inB);
    readVector(0) := '1';

    inR := reverse_any_vector(inR);
    inG := reverse_any_vector(inG);
    inB := reverse_any_vector(inB);

    inputR <= inR;
    inputG <= inG;
    inputB <= inB;
    while (rdyR = '0' or rdyG = '0' or rdyB = '0') loop
      wait for PERIOD;
    end loop;
    wait for 5 ns;
    newFrame <= '0';
    wait for 45 ns;
    activate := '1';

    -- LOOP AS LONG AS AT LEAST ONE FILE IS NOT EMPTY
    while (not ENDFILE(infileR)) or (not ENDFILE(infileG))
    or (not ENDFILE(infileB)) loop
      readVector := "000";
      if (readR = '1' and not ENDFILE(infileR)) then
        READLINE(infileR, buf_inR);
        READ(buf_inR, inR);
      inR := reverse_any_vector(inR);

      readVector(2) := '1';
    end if;

    if (readG = '1' and not ENDFILE(infileG)) then
      READLINE(infileG, buf_inG);
      READ(buf_inG, inG);
      inG := reverse_any_vector(inG);

      readVector(1) := '1';
    end if;

    if (readB = '1' and not ENDFILE(infileB)) then
      READLINE(infileB, buf_inB);
      READ(buf_inB, inB);
      inB := reverse_any_vector(inB);

      readVector(0) := '1';
    end if;

    WRITE(buf_rOrdr, conv_integer(readVector(2)));
    WRITE(buf_rOrdr, string'("␣"));
    WRITE(buf_rOrdr, conv_integer(readVector(1)));
    WRITE(buf_rOrdr, string'("␣"));
    WRITE(buf_rOrdr, conv_integer(readVector(0)));
    WRITELINE(readorder, buf_rOrdr);

    -- Set input signals
    wait for 5 ns;
    inputR <= inR;
    inputG <= inG;
    inputB <= inB;
    -- Count pixel
    wait for 45 ns;
  --      wait for PERIOD;
  end loop;

  wait for 8 * PERIOD;
  wait;
end process;

-- ----------------------------------
--/ PROCESS CONTROLLING OUTPUT
--/ Output starts writing when active is high.
--/ Continues until the correct number of pixels is reached,
--/ as defined by the 'size' constant.
--/ Compares outputs with the values in the original input file.
--/

writectrl : process(clk)
  file originalFile : text open READ_MODE is path & fil & "vhdl";
  file outFile : text open WRITE_MODE is path & fil & "_out";

  variable buf_inOriginal, buf_out : line;
  variable decoded, original : std_logic_vector(24 - 1 downto 0);
```

```
    begin
      if (rising_edge(clk)) then
        if (active = '1' and numPixelsOut < (size) + 1) then

          -- Spectral "recorrelation"
          decoded := (outputR + outputG)&outputG&(outputB + outputG);

          -- DEBUG UTEN R-G,G,B-G:
          --decoded := outputR & outputG & outputB;

          HWRITE(buf_out, decoded, RIGHT, 6);
          WRITELINE(outFile, buf_out);
          numPixelsOut := numPixelsOut + 1;
          if (not ENDFILE(originalFile)) then
            READLINE(originalFile, buf_inOriginal);
            HREAD(buf_inOriginal, original);
            if (original /= decoded) then
              WRITE(OUTPUT, "Output␣did␣not␣match␣the␣original␣value␣");
              WRITE(
                OUTPUT, "on␣pixel␣number␣"
                & natural'image(numPixelsOut) & "." & LF
              );
            end if;
          end if;
        elsif (numPixelsOut > size) then
          -- -------------------------------------
          -- END SIMULATION :
          -- -------------------------------------
          write(OUTPUT, "␣-----------------␣" & LF);
          write(OUTPUT, "Simulation␣complete.");
          report "" severity failure;
        -- -------------------------------------
        else
        -- something , waiting for active.
        end if;
      end if;
    end process;
  end architecture RTL;
```

A.10 Definitions

defs.vhd

```
  -- DEFINITIONS
  -------------------------------------------------------------------
  constant A_LEN         : integer := 13;
  constant B_LEN         : integer := 6;
  constant C_LEN         : integer := 9;
  constant N_LEN         : integer := 6;
  constant CTXT_INDEX_RANGE : integer := 256;
  constant CTXT_ADDR_SIZE   : integer := 8;
  constant SMPL_LEN         : integer := 8;
  constant GOL_MAX_LEN      : integer := 16;
  constant DEC_REG_SIZE     : integer := 48;
  constant DEC_REG_LEN      : integer := 6;
  constant DATA_IN_REG_SIZE : integer := 16;
  constant DATA_IN_REG_NUM  : integer := 3;

  constant CACHE_SIZE       : integer := 1917 + 1920;
  constant CACHE_INDEX_OFFSET : integer := 10;

  constant ENC_K_DELAY     : integer := 5;
  constant ENC_INDEX_DELAY : integer := 11;
  constant ENC_SIGN_DELAY  : integer := 5;

  constant ENC_INDEX_RST  : integer := 1;
  constant ENC_K_RST      : integer := 5;
  constant ENC_LOCO_RST   : integer := 11;
  constant ENC_LOCO_W_RST : integer := 13;
  constant ENC_GOL_RST    : integer := 14;

  constant DEC_K_DELAY     : integer := 2;
  constant DEC_INDEX_DELAY : integer := 11;
  constant DEC_SIGN_DELAY  : integer := 5;

  constant DEC_INDEX_RST  : integer := 1;
  constant DEC_K_RST      : integer := 5;
  constant DEC_LOCO_RST   : integer := 11;
  constant DEC_LOCO_W_RST : integer := 13;
  constant DEC_GOL_RST    : integer := 5;

end defs;
```

BIBLIOGRAPHY

[1] Hafskjold, Stian R. ; Fagerheim, Fredrik J. (2012) *Lossless video compression in an FPGA for reducing DDR memory bandwidth usage (Preliminary work).*

[2] Hong-Sik Kim ; Joohong Lee ; Hyunjin Kim ; Sungho Kang ; Woo Chan Park (2011) *A Lossless Color Image Compression Architecture Using a Parallel Golomb-Rice Hardware CODEC* IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, Vol. 21, No. 11, NOVEMBER 2011

[3] Xiaowen Li ; Xinkai Chen ; Xiang Xie ; Guolin Li ; Li Zhang ; Chun Zhang ; Zhihua Wang (2007) *A Low Power, Fully Pipelined JPEG-LS Encoder for Lossless Image Compression*

[4] Tsung-Han Tsai ; Yu-Hsuan Lee ; Yu-Yu Lee (2010) *Design and Analysis of High-Throughput Lossless Image Compression Engine Using VLSI-Oriented FELICS Algorithm.* IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, Vol. 18, No. 1, JANUARY 2010

[5] Chen, X. ; Canagarajah, N. ; Nunez-Yanez, J. L. ; Vitulli, R. (2007) *Hardware Architecture for Lossless Image Compression Based on Context-based Modeling and Arithmetic Coding*

[6] Seroussi, G. (2000) *lossless compression of continuous-tone images*. HP laboratories

[7] Seroussi, G. (2000) *Optimal Prefix Codes for Sources with Two-Sided Geometric Distributions*. IEEE transactions on information theory, vol. 46, no. 1

[8] Golomb, S. W. (1966) *Run-length Encodings*. IEEE transactions on information theory, vol. IT-12, pp. 399-401

[9] Sayood, K. (2003) *Lossless Compression Handbook*. Elsevier Science (USA)

[10] Gonzales, R. C. (2007) *Digital Image Processing*. Pearson Prentice Hall

[11] Weinberger, M. J. (1996) *LOCO-I: A low complexity, context-based, lossless inage compression algorithm*. HP laboratories

[12] *New LZW Data Compression Algorithm and Its FPGA Implementation*. Wei Cui (

[13] Arruebo, B. (2004) *Arithmetic compression software*. Downloaded from: http://sourceforge.net/projects/aricom/ [accessed 20.12.2012].

[14] Duce, D. (2003) *Portable Network Graphics (PNG) Specification (Second Edition)* http://www.w3.org/TR/PNG/ [accessed 20.12.2012]. W3C Recommendation

[15] Wallace, G. K. (1992) *The JPEG Still Picture Compression Standard*. IEEE transactions on consumer electronics, vol. 3, no. 1