Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2019

A Resource Efficient, High Speed FPGA Implementation of Lossless Image Compression for 3D Vision

Martin Hinnerson

Master of Science Thesis in Electrical Engineering

A Resource Efficient, High Speed FPGA Implementation of Lossless Image Compression for 3D Vision:

Martin Hinnerson

LiTH-ISY-EX-ET--19/5205--SE

Supervisor: **Mattias Johannesson**
SICK IVP AB
Olov Andersson
ISY, Linköpings universitet

Examiner: **Kent Palmkvist**
ISY, Linköpings universitet

Division of Computer Science
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

High speed laser-scanning cameras such as Ranger3 from SICK send 3D images with high resolution and dynamic range. Typically the bandwidth of the transmission link set the limit for the operational frequency of the system. This thesis show how a lossless image compression system in most cases can be used to reduce bandwidth requirements and allow for higher operational frequencies.

A hardware encoder is implemented in PL on the ZC-706 development board featuring a ZYNQ Z7045 SoC. In addition, a software decoder is implemented in C++. The encoder is based on the FELICS and JPEG-LS lossless compression algorithms and the implementation operate at 214.3 MHz with a max throughput of 3.43 Gbit/s.

The compression ratio is compared to that of competing implementations from Teledyne DALSA Inc. and Pleora Technologies on a set of typical 3D range data images. The proposed algorithm achieve a higher compression ratio while maintaining a small hardware footprint.

This thesis has resulted in a patent application.

Acknowledgments

First of all i would like to thank my supervisor at SICK IVP, Mattias Johannesson. I am very thankful for the insight and discussions along the way. My thanks also go to the rest of the 3D camera team at SICK for the weekly feedback and great company.

From Linköping University: my examiner Kent Palmkvist, many thanks for your feedback and support.

Lastly, a special thank you to my friends Jacob, Tim and Oscar. Your support and company during my studies have been very valuable.

Linköping, May 2019
Martin Hinnerson

Contents

Notation

Nomenclature

Notation	Meaning
x	Current Sample
$\hat{x}$	Sample Prediction
H	Entropy
q	Quotient
r	Remainder
P	Probability function
C	Correction Value
N	Bit depth
k	Division factor in Golomb-Rice codes

Abbrevations

Abbrevation	Meaning
FPGA	Field Programmable Gate Array
PL	Programmable Logic
PS	Processing System
PLL	Phase Locked Loop
FELICS	Fast & Efficient Lossless Image Compression System
CALICS	Context Adaptive Lossless Image Compression System
SFALIC	Simple Fast and Adaptive Lossless Image Compression
LOCO-I	LOw COmplexity, LOssless COmpression for Images
RLE	Run Length Encoding
JPEG	Joint Photographic Experts Group
JPEG-LS	Joint Photographic Experts Group - Lossless
SZIP	Compression System from Consultative Committee for Space Data Systems
LZ	Lempel, Ziv
CR	Compression Ratio
ROI	Region Of Interest
SOC	System on Chip
GRC	Golomb Rice Coding
SABC	Simplified Adjusted Binary Coding
EDF	Exponential Distribution Function
PDF	Probability Distribution Function

1 Introduction

1.1 Motivation

In highly automated industries like manufacturing, assembly and quality control, fast and precise sensors are necessary. In many such applications, a 3D representation of the measured objects is desireable. A widely used machine vision technique to calculate a height map of an object is laser line triangulation using high speed cameras.

Systems based on laser triangulation 3D imaging typically send uncompressed 3D data over a transmission link to a host computer for further processing. The bandwidth of the link sets a limit for the camera operation speed since images are being sent in a continuous manner and not buffered on the device. Hardware upgrades to allow for faster communication comes with a high cost, and a cheaper solution to increase throughput is desired.

Digital grayscale and RGB-images typically have high spatial and temporal redundancy. Lossless compression schemes like JPEG-LS, FELICS and RLE can be used to increase information density and reduce the size of image files. The 3D height map images are comparable to natural grayscale image and a solution like this, with high enough compression and decompression speed, could be used to increase total throughput.

The system of interest for this thesis is the Ranger3 camera from SICK. Ranger3 is a high-speed 3D camera intended to be the vision component in a machine vision system. Ranger3 uses laser line triangulation to make measurements of the objects passing in front of the camera and sends the measurement results to a PC. The camera is used in industries as the machine vision element for applications like size rejection, volume measurement, quality control, predictive maintenance

or for finding shape defects.

1.2 Problem Formulation

The purpose of this thesis is to evaluate if high-speed lossless compression algorithms can be used to reduce the bandwidth usage of the transmission link in the Ranger3 system. Since the goal is to increase the camera operation speed, the software decoder has to be fast enough to match the operation speed of the camera. After evaluation, an algorithm is selected and a compression module is implemented on the ZC706 development board using VHDL. In addition a decompression algorithm is implemented for the host PC using C++. The system setup is depicted in Figure 1.1.

The following questions are studied in this thesis:

Table 1.1: *Guidelines of the project*

Specification	**Index**
Can a compression factor of 1,5-2 be reached for typical 3D range images? This is a guideline set by SICK for the expected performance.	1
Can the hardware encoder be implemented using only a small amount of the available resources of the Z-7030 FPGA (same as in Ranger3)? (aim for <10% LUT/Memory usage)	2
Can the encoder operate at the current FPGA clock frequency of 200 MHz?	3
Can the throughput of the encoder exceed that of the GigE link, ie. 125MB/s?	4
Can the software decoder decompress profiles at a frequency of at least 7kHz for full-frame images? This is the current specified limit for full-frame.	5

1.3 Limitations

To limit the scope of the project, the system is implemented on the ZC706 development board and not the complete Ranger3 system. The Ranger3 camera is capable of sending more information like reflected peak intensity and scatter data, in addition to 3D range data. However, in this thesis the main focus is compression of 3D range data.

The communication link is considered ideal, meaning there is no noise on the

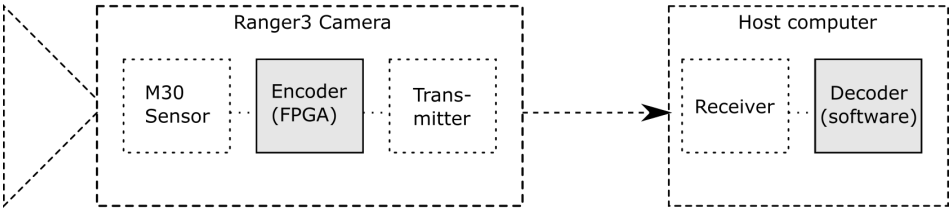


Figure 1.1: *System setup with Ranger3 and host computer. Highlighted modules Encoder and Decoder are the area of interest in this thesis.*

channel. Ranger3 use the GigE Vision standard which feature retransmission of lost and corrupted data, however in this thesis all information sent is expected to arrive at the desired location.

1.4 Thesis Outline

In the first chapters, related theory to the subject and the hardware platform are presented. This leads into a pre-study where the selection of a suitable base for the project is made. After this a more in depth description of the implementation is presented followed by the results. Finally a discussion about the performance and possible improvements. A detailed outline of the thesis is as follows:

Chapter 2 include the relevant theory and background on lossless image compression and the Ranger3 system.

Chapter 3 present how compression algorithms are compared and evaluated and describe the selection of an algorithm.

Chapter 4 explain the method of how the system was design and verified, both in software and hardware.

Chapter 5 present the results such as performance numbers and hardware utilization.

Chapter 6 contains a discussion of the method used and the results of the project.

Chapter 7 give a short conclusion to the project and some final remarks.

2

Theory and Background

2.1 Laser-line 3D triangulation

Laser-line 3D triangulation is a technique used in many industrial applications like manufacturing, assembly and quality control. The basic concept of laser-line triangulation is simple. A narrow line of light is projected on the surface of interest. This line will appear distorted for an observer with another perspective than the light source when an object of varying height is moved through it. By analysis of this distortion an accurate representation of the surface under the line can be calculated, this is called a *profile*.

In addition to the camera and laser projector, a system moving the object through the laser line is required. When an object has been passed through the laser line the calculated profiles can be combined to create a complete 3D height map of the object. Figure 2.1 is an example where the model has been colored to better visualize height.

In the most common configuration, the laser line is projected perpendicular to the measurement plane and the camera is positioned at an angle in front or back. The advantage of having the laser in a configuration like this is that the line distortion is concentrated to one direction which will simplify the calculations and reduce the calibration complexity. The drawback is that the camera view the object at an angle which means that the camera needs a greater depth of field. Another drawback is that when the object vary in height, some parts of the laser line can be blocked and invisible for the camera. Pixels in a *profile* where the laser-line was blocked or the intensity was below a set threshold are called *missing data*. Figure 2.2 further describe the system setup.

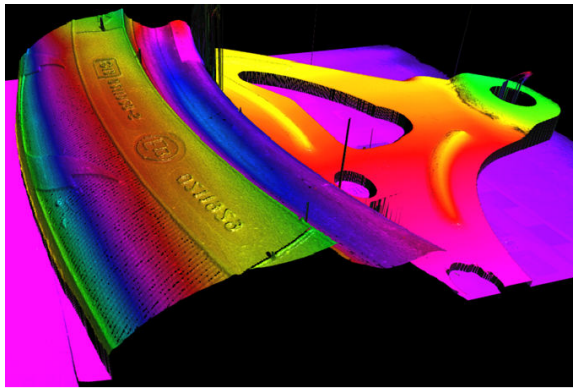


Figure 2.1: *Example of 3D image (From Ranger3 Operating Instructions, 2018 [16])*

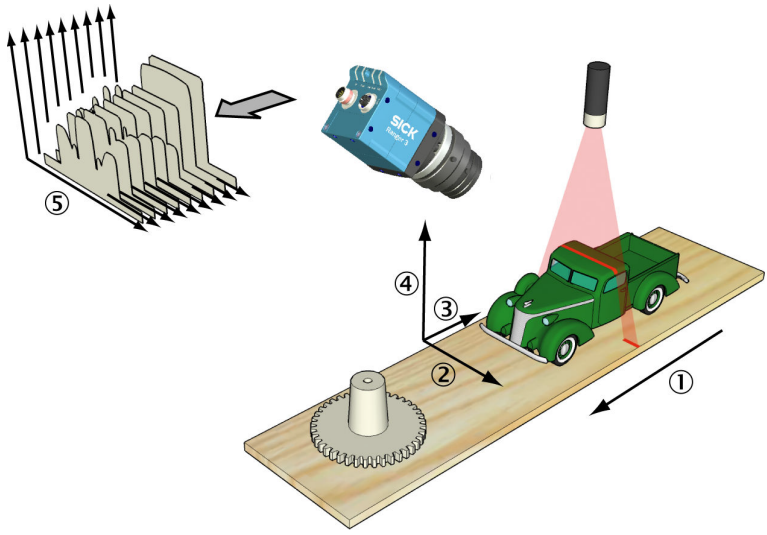


Figure 2.2: *Measuring the range of a cross-section of an object (From Ranger3 Operating Instructions, 2018 [16])*

1. *Transportation direction*
2. *X (width)*
3. *Y (negative transport direction)*
4. *Z (range)*
5. *Profiles*

2.2 Ranger3

Terminology

Term	Meaning
3D image	A point cloud where the object shape is represented by three coordinates
block	In the GigE Vision protocol a image is named block.
profile	Values for each measurement point along a cross-section of the object.
range	The height measurement of a point in a profile
intensity	The intensity value of the pixel in a 2D sensor image.
reflectance	The reflected peek intensity of the laser line when measuring 3D profiles.
height map	A frame where the values represent height or depth and not intensity.
linerate	Operational frequency in profiles per second

The range data provided by the Ranger3 camera can be seen as a regular grayscale image except instead of intensity, the pixel values corresponding to height along a profile. The bit depth provided can be set to 8, 12 or 16-bit with the normal case being 12-bit. This mean that the pixel value range is $[0, 2^N - 1]$, where N is the bit-depth.

The biggest difference between the 3D range images and a normal grayscale image is the existence of *missing data*. In the Ranger3 system, *missing data* is represented by the pixel value 0. In the camera sensor, a threshold for the reflected intensity can be set, resulting in missing data for all pixels below the set threshold. This means that with proper calibration, the background of a scanned object contain missing data, and not only the features of the target where the field of view was blocked.

Additional specifications of the ranger3 can be seen in Table 2.1.

Table 2.1: *Ranger3 specification*

Specification	
Bit depth, N	8-16 bits/pixel
Image Width	2560 pixels
Linerate	7 kHz full frame and up to 46 kHz in ROI
Data Transmission rate	1000Mbit/s

2.2.1 Hardware

The Ranger3 camera use the SICK M30 CMOS sensor which enables image processing directly on the sensor. In addition to the sensor a Xilinx ZYNQ-7030 SoC

is used. The data produced by the sensor pass through the FPGA before it is transmitted via the GigE link. The Z-7030 is a feature-rich SOC containing a dual-core ARM Cortex-A9 based processing system and 28nm Xilinx programmable logic in a single device. [22]

The Z-7030 programmable logic specification is specified in Table 2.2.

Table 2.2: *Z-7030 Programmable Logic [22]*

Programmable Logic Cells	125k
Look-Up Tables (LUTs)	78 600
Flip-Flops	157 200
Block RAM (# 36 kb Blocks)	9.3Mb (265)
DSP Slices (12x25 MACCs)	400

2.2.2 GigE Vision

The transmission protocol used in the Ranger3 camera follow the GigE Vision protocol which is a standard defined for high-performance industrial cameras. It is based on the Internet Protocol (IP) standard and runs on the UDP protocol. GigE Vision provide a framework for transmission of high-speed video related control and image data over Ethernet networks. Error correction and retransmission of corrupted data are important features in the protocol.

2.3 Hardware Setup

The hardware on which the system will be implemented and tested is as stated not the full Ranger3 camera. Instead the Xilinx ZC706 development board is used together with 2 additional modules. Figure 2.3 show the system used and figure 2.4 a block diagram of it. The additional M30 adapter include the SICK CMOS camera sensor used in Ranger3, which means the development board can be used as a camera. A Gigabit Ethernet module is present for faster communication with the development board. The development board include 1GB of DDR3 memory which is used during evaluation for image storage.

2.3.1 Xilinx ZYNQ-7045

The development board include a Z-7045, a larger model of the ZYNQ family than what is used in Ranger3, packing some additional resources in hardware. This could be useful for testing and verification hardware in addition to the encoder, but does not affect the final implementation.

2.3.2 AXI4-S bus

For communication between PL and PS, a bus called AXI4 is used. There are three versions of the protocol, AXI4, AXI4-Lite and AXI4-Stream. The hardware

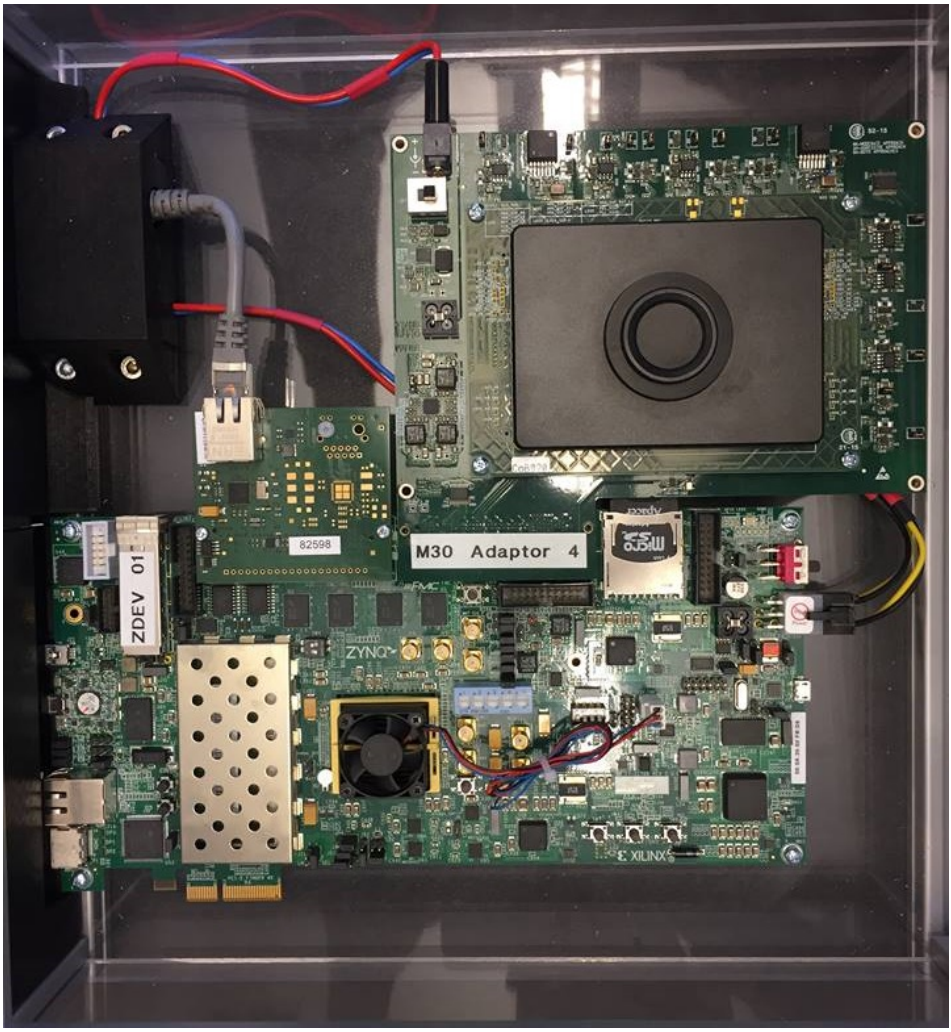


Figure 2.3: *Hardware setup of ZC706 development board with M30 adapter and additional GigE connection.*

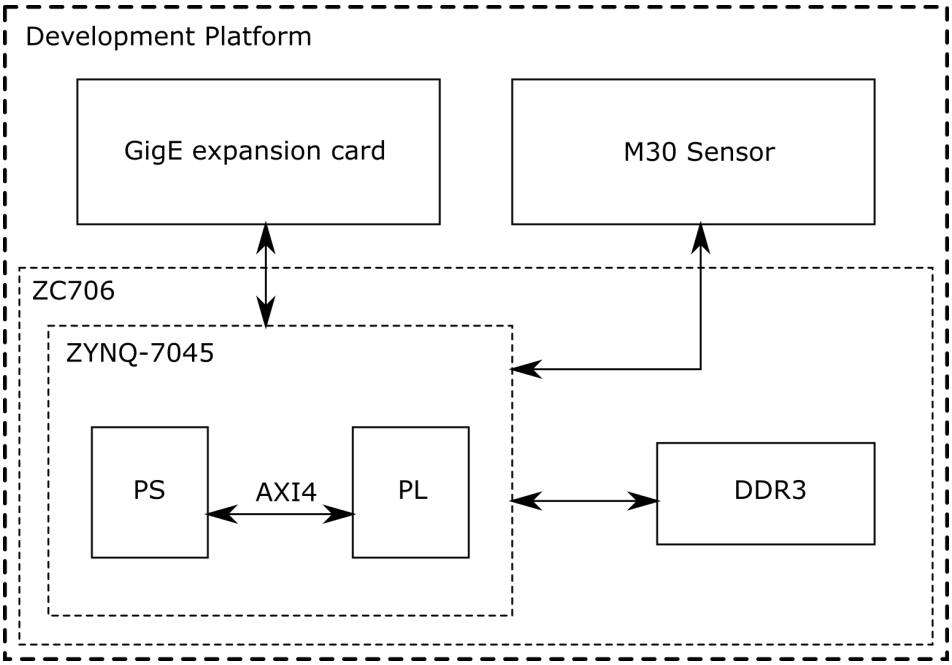


Figure 2.4: *Block diagram of development platform.*

encoder will have to read and write larger bursts of data containing the unencoded and encoded images. Therefore AXI-Stream is used which feature data-only bursts and don't need a memory mapped address for single data. This means that a set start address could be defined indicating the start address where data is read/written and the following data can be transferred in bursts. This allow high throughput and an easy to implement protocol, without the unnecessary features of the full AXI4 interface. In the AXI4 interface, both PL and PS can act as master and request a read/write operation. The implemented encoder will act as slave when receiving unencoded pixel values and as master when writing them back to memory.

In addition to the AXI4 bus the AXI Direct Memory Access IP is used to allow direct read/write from DDR. This IP fully comply with the AXI-Stream interface used by the encoder.

2.4 Lossless Image Compression

Digital data compression is the task of encoding information using fewer bits than the original representation. The goal of this is typically to reduce storage space or minimize transport bandwidth. By minimizing the redundant information in digital files, data can be compressed either lossless or lossy. Lossless compression is when the compression process is perfectly reversible, meaning that it

is possible to reverse to achieve an exact copy of the original. Lossy compression is when this cannot be achieved, typically because of quantization in the compression algorithm.[15] Lossless compression techniques are the area of interest in this thesis.

A digital image is typically defined as an array of pixels. The number of pixels in the array is referred to as the resolution. In a grayscale image the pixel is represented by a non-negative integer value describing the intensity of that pixel. The bit-depth of an image define the range of values that a pixel can have.

Compression algorithms designed for images typically consist of a decorrelation step followed by entropy coding. Natural images, also referred to as continuous-tone images, have a high correlation between neighboring pixels, i.e. spatial redundancy, the decorrelation step is used to minimize this redundancy. Many image compression algorithms are predictive. A predictive algorithm use a function that predicts the pixel values and then calculates the difference between the prediction and the actual pixel value. This prediction error is encoded instead of the original pixel value. The predictors used in the prediction function are typically a small number of neighboring pixels. For normal grayscale images the prediction error distribution is close to symmetrically exponential. This results in values that are easier to compress since the image entropy [2.4.1] has been decreased. [11][20]

After image decorrelation the error values are encoded using an entropy coding method. An arithmetic entropy coder could get arbitrarily close to the optimum compression, but in practice this type of implementation is relatively slow and not as perfect as theoretically possible. Another approach is to use prefix codes such as Huffman coding. Prefix codes encode symbols with binary codewords of varying lengths based on the statistical probability of a given symbol. This type of coding can be done relatively fast compared to arithmetic coding. [15] When the probability distribution is known, and symmetrically exponential, huffman codes can be replaced by faster entropy coders such as Golomb and Golomb-Rice coders.[11] Using prefix codes for image entropy coding, the resulting code length can never be less than 1 bit per pixel. To improve on this some algorithms like LOCO-I [20] and CALICS [21] use special methods to encode "flat" regions. One such method is Run Length Coding (RLE).

Another approach to reduce the entropy of images before coding is transforms like DCT and wavelet-transforms used in JPEG and JPEG-2000. These methods however are more suitable for lossy image compression and even though algorithms like JPEG-2000 have lossless modes the compression ratio and speed are better in the predictive algorithms.[15]

2.4.1 Entropy

The information contents of the outcome $\hat{x}$ of a random variable x can be quantified via the **Shannon information** [2.1].

Definition 2.1. The Shannon information is given by

$$-log_2(P(x = \hat{x}))$$

This definition of the shannon information has the properties that

- the information associated with an outcome decreases with its probability
- the information associated with a pair of two independent outcomes is equal to the sum of the information associated with each individual outcome

A digital image can be seen as a two-dimensional signal. The shannon information of a pixel will be higher for pixel values that are less occurring in the image and lower for pixel values that occur more often. If a probability distribution of the entire image is generated, a measurement of the average information in a pixel can be calculated. This is called the Shannon **entropy** [2.2].

Definition 2.2. The average of the Shannon information of x is called the **entropy**. The M-**ary entropy function** is given by

$$H = -\sum p_m log_2(p_m)$$

where $p_m = P(x = m)$ is the probability of the m-th symbol.
The unit of entropy is **bits** since the base of the log is 2.

This definition of entropy is very useful as it can be used as a measure of compressibility. If an image is seen as a set $\mathcal{X}$ of an experiment, the entropy is a measure of the average number of binary symbols needed to code the source. Shannon showed that the best that an entropy based lossless compression scheme can do is to encode a source with the average number of bits equal to the entropy of the source. [9]

2.4.2 Golomb Coding

Golomb coding, invented by Solomon W. Golomb, is a lossless data compression method. Golomb codes are suitable in situations where smaller values are more likely to occur than larger values, which is the case after image decorrelation used in most predictive coding algorithms. Golomb codes divides the input value n into a quotient q followed by a remainder r. The denominator m is a tunable parameter. The quotient q can take on values 0, 1, 2, ... and is represented in unary code. The unary code for a positive integer n is simply n 1s followed by a 0. [15] The remainder r can take on $0, 1, 2, ..., m-1$ and is represented in regular binary coding. The variables q and r are calculated as

$$q = \left\lfloor \frac{n}{m} \right\rfloor \tag{2.1}$$

$$r = n \bmod (m) \tag{2.2}$$

2.4.2.1 Golomb-Rice codes

In the special case when m is a power of two the computation of q can be realized with shift operations instead of division with reduces the computational complexity significantly in a hardware implementation. Equations [2.1] and [2.2] can be rewritten as

$$q = \left\lfloor \frac{n}{2^k} \right\rfloor \tag{2.3}$$

$$r = n \bmod (2^k) \tag{2.4}$$

where k now is the tunable parameter and $m = 2^k$.

This corresponds to removing the k least significant bits of n and encoding the remaining bits as a unary number. After this the k least significant bits are sent directly.

The final code length of the Golomb-Rice code can be determined by

$$l = q + 1 + k \tag{2.5}$$

For example, a Golomb-Rice code representation for $n = 11$ with $k = 2$:

$$\begin{cases} q = \left\lfloor \frac{15}{2^2} \right\rfloor = 3 \\ r = 15 \bmod 2^2 = 3 \end{cases} \Rightarrow \begin{cases} q_{unary} = 1110 \\ r_{bin} = 11 \end{cases} \Rightarrow code = 111011$$

and in the same way for $n = 2$ with $k = 1$:

$$\begin{cases} q = \left\lfloor \frac{2}{2^1} \right\rfloor = 1 \\ r = 2 \bmod 2^1 = 0 \end{cases} \Rightarrow \begin{cases} q_{unary} = 10 \\ r_{bin} = 0 \end{cases} \Rightarrow code = 100$$

This example does not make it obvious to why a coding like this is desireable since the codes are longer than regular binary coding. However a coding like this is needed in order for a decoder to be able to separate codewords of varying length.

2.5 Relevant Algorithms

This chapter describe two main algorithms that are highly relevant to the thesis, JPEG-LS and FELICS.

2.5.1 JPEG-LS

JPEG-LS is based on the LOCO-I algorithm. It was first introduced by Weinberger [20] as the new standard for lossless and near-lossless image compression of natural images. The algorithm uses a prediction stage followed by a context modeling stage to decorrelate the pixels in the image. This is followed by a entropy coding stage based on Golomb-Rice codes. The system also has a run mode active

when the context is flat. When this mode is active the system encode pixels using a RLE approach. The full block diagram of the JPEG-LS encoder can be seen in Figure 2.5.

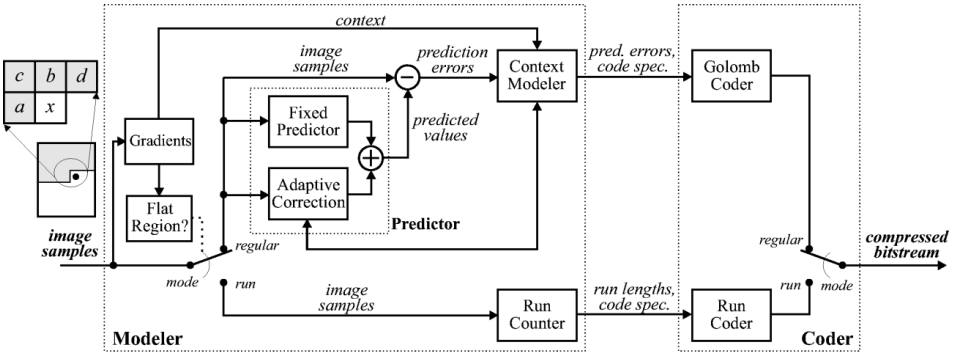


Figure 2.5: *JPEG-LS Block Diagram (from Weinberger[18] 2000)*

2.5.1.1 Context Calculation

To compress a pixel x, the pixel itself and its neighbors a, b, c and d as shown in Figure 2.5 are used. In the **Gradients** block, the context of the current pixel is determined by three gradients according to Equation 2.6. These gradients will be used by the **Context Modeler** which will adaptively adjust parameters for both the **Adaptive Correction** and the **Golomb Coder** depending on the context.

$$\begin{cases} g_1 = d - b \\ g_2 = b - c \\ g_3 = c - a \end{cases} \tag{2.6}$$

The gradients are quantized into different regions to reduce the number of context vectors $Q = [q_1, q_2, q_3]$. For a 8-bit per pixel alphabet, the default quantization regions are

$$q(g) = \begin{cases} 0 & \text{if } g = 0 \\ \pm 1 & \text{if } g = \pm\{1, 2\} \\ \pm 2 & \text{if } g = \pm\{3, ..., 6\} \\ \pm 3 & \text{if } g = \pm\{7, ..., 20\} \\ \pm 4 & \text{if } g = \pm\{21, ...\} \end{cases} \tag{2.7}$$

Since each q_i $(i = 1, 2, 3)$ can have 9 different values there are 729 available context vectors. The opposite q-triplets in terms of sign represent the same context information, only opposite. Therefor the context is reduced using Equation 2.8.

$$P(\varepsilon = \Delta | C_t = [q_1, q_2, q_3]) = P(\varepsilon = -\Delta | C_t = [-q_1, -q_2, -q_3]) \tag{2.8}$$

Here ε is the fixed prediction error, Δ is the error value and C_t represents the quantized context triplet. The boundaries in Equation 2.7 are adjustable parameters but must center around 0. Note that for a 16-bit implementation these regions will change. The standard does not specify the regions for other than 8-bit images.

2.5.1.2 Fixed Prediction

In the regular mode, a *fixed* prediction of the current pixel $\hat{x}$ is made using Equation 2.9. The first case tries to predict horizontal edges above the current pixel x and the second tries to predict a vertical edge left of x. When no horizontal of vertical edge is present the last case will be picked. This corresponds to the value if x belonged to the plane defined by pixels a, b and c.

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

When a prediction has been made, the residual error is computed as $\varepsilon = \hat{x} - x$. For continuous-tone images where the spatial redundancy is high, the probability distribution of the residual error ε will be represented by a TSGD (Two-Sided Geometric Distribution) centered at zero. In a TSGD, the probability of an integer value ε of the prediction error is proportional to $\theta^{|\varepsilon|}$, where $\theta \in [0,1]$ controls the two-sided exponential decay rate. However, a more general model is used to remove the dc offset that is typically present in prediction error signals. This offset is due to integer-value constraints and possible bias in the prediction step. An additional offset parameter μ is used. The offset is broken into an integer part R (bias) and a fractional part s (shift), such that $\mu = R - s$, where $(0 \leq s < 1)$. Thus, the TSGD parametric class $P_{(\theta,\mu)}$ assumed by JPEG-LS for the residuals of the fixed predictor at each context, is given by (2.10).

$$P_{(\theta,\mu)} = \theta^{|\varepsilon - R + s|}, \varepsilon = 0, \pm 1, \pm 2, ... \tag{2.10}$$

In the **Adaptive Correction** stage, the integer offset R is canceled and Equation 2.10 reduces to 2.11.

$$P_{(\theta,\mu)} = \theta^{|\varepsilon + s|}, \varepsilon = 0, \pm 1, \pm 2, ... \tag{2.11}$$

This reduces the range for the offset and is matched to the Golomb codes. The model of (2.11) is depicted in Figure 2.6.

2.5.1.3 Adaptive Correction

The adaptive part of the prediction is used to cancel the integer part R of the offset due to the fixed predictor.

The correction value C could be calculated as the average value of the previous samples in the same context

$$C = \left\lceil \frac{B}{N} \right\rceil \tag{2.12}$$

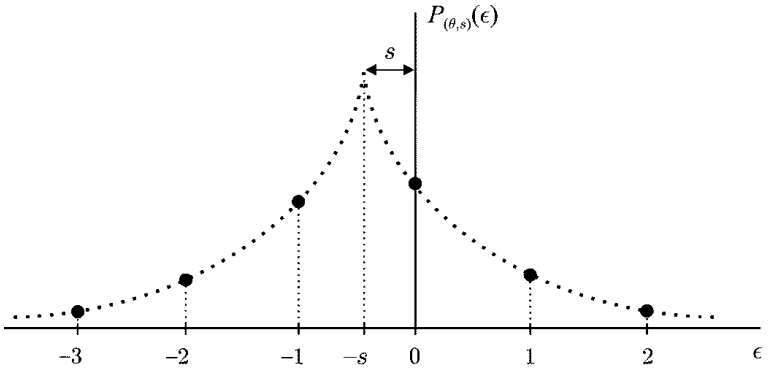


Figure 2.6: *Two-Sided geometric distribution of prediction residuals (from Weinberger[18] 2000)*

where B is the cumulative value of the N previous ε. Likewise, the absolute deviation could be calculated by

$$C_a = \left\lceil \frac{A}{N} \right\rceil \tag{2.13}$$

where A is the cumulative value of the N previous $|\varepsilon|$. This is used in the entropy coder for adaptive k parameter selection.

There are however two main problems with this implementation. Atypically large errors can affect future values of C until it returns to its typical value and the calculation of C and C_a requires division, which is a complex operation in hardware.

To solve this the correction value is instead calculated according to the code in Listing 2.1.

```
1     A = A + abs(e);
2     B = B + e; // accumulate prediction residuals
3     N = N + 1; // update occurrence counter
4     //update correction value and shift statistics
5     if (B <= -N) {
6         C = C - 1;
7         B = B + N;
8         if (B <= -N)
9             B = -N + 1;
10    }
11    else if (B > 0) {
12        C = C + 1;
13        B = B - N;
14        if (B > 0)
15            B = 0;
```

```
16      }
```

Listing 2.1: *Bias Computation procedure*

The correction value C is added to the error value ε and the result is mapped so that the numbers are nonnegative integers according to Equation 2.14.

$$M(\varepsilon) = \begin{cases} 2\varepsilon & \varepsilon \geq 0 \\ 2|\varepsilon| - 1 & \varepsilon < 0 \end{cases} \tag{2.14}$$

2.5.1.4 Entropy Coding

The entropy coding of JPEG-LS is Golomb-Rice Codes, explained in Section 2.4.2. The tunable parameter k given by the context of the sample is determined by the accumulated sum of prediction residuals $A(Q)$ for a specific context Q, which is calculated in the **Context Modeler**. The computation of k is described in Listing 2.2.

```
1     for(k=0; (N<<1) < A; k++);
```

Listing 2.2: *Computation parameter k*

The quotient q and the remainder r are calculated according to Equation 2.3 and Equation 2.4 respectively, where $n = M(\varepsilon)$.

2.5.1.5 Run Mode

JPEG-LS features a run mode that is used to encode regions where the gradient is flat. The regular mode of the encoder can never achieve compression with less than one bit per sample. However in flat regions RLE could be used to achieve a much lower CR than the regular mode.

2.5.1.6 Context Conflict Problem

Both FPGA and VLSI implementations of [18][12][4][8] discuss the data dependency problem that arise with the adaptive context correction and k-value parameter selection. The context conflict problem occur when two consecutive pixels reside in the same context, meaning that the second pixel depend on the updated context parameters A, B, C from the first pixel. With one pixel processed per cycle in the pipeline, this will lead to a situation where the second pixel will be processed with out-of-date context parameters. This could be solved by stalling the pipeline which will obviously reduce throughput.

2.5.2 FELICS

FELICS (Fast, Efficient, Lossless Image Compression System) encodes pixels in raster-scan order and uses the pixel's two closest neighbors to directly obtain a prediction for the current pixel. The current pixel x is predicted using the prediction template described in Figure 2.7. Case 4 is the normal case for prediction of the residual and Case 1-3 are special cases for pixels on the first line and first

column where Case 4 cannot be applied. The first 2 pixels on the first line are not encoded.

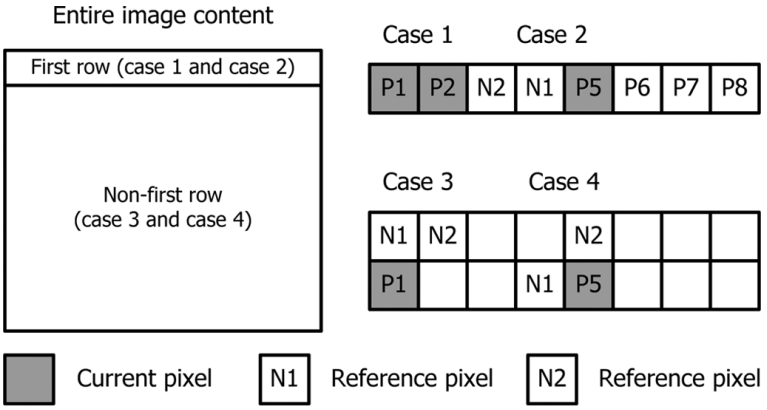


Figure 2.7: *FELICS prediction model (From T. Tsai, Y. Lee, and Y. Lee, 2010 [19])*

To calculate a prediction residual ε the following steps are followed:

1. According to the prediction template [Figure 2.7] find the two reference pixels N_1 and N_2.
2. Compute

$$\begin{cases} L = min(N_1, N_2) \\ H = max(N_1, N_2) \\ \Delta = H - L \end{cases}$$

3. (a) If $L \leq x \leq H$, use one bit to encode **In Range**. Then use adjusted binary code to encode $\varepsilon = x - L$ in $[0, \Delta]$.

   (b) If $x < L$ use one bit to encode not in range and one bit to encode **Below Range**. Then use Golomb-Rice Code to encode the non-negative integer $\varepsilon = L - x - 1$.

   (c) If $x > L$ use one bit to encode not in range and one bit to encode **Above Range**. Then use Golomb-Rice Code to encode the non-negative integer $\varepsilon = x - H - 1$.

The reasoning to step 3 is that the probability distribution model of the predictions follow the model depicted in Figure 2.8. When the current pixel reside in range, i.e. $L \leq x \leq H$, the pixel values follow a almost uniform distribution. There is however a slight peak in the middle which is why adjusted binary coding is used to give shorter codewords for pixels in the middle and longer for pixels

closer to the edges. When $x < L$ or $x > H$ the probability distribution is exponentially decreasing, which is why Golomb-Rice codes are used to encode these pixels. These regions are called below range and above range respectively.

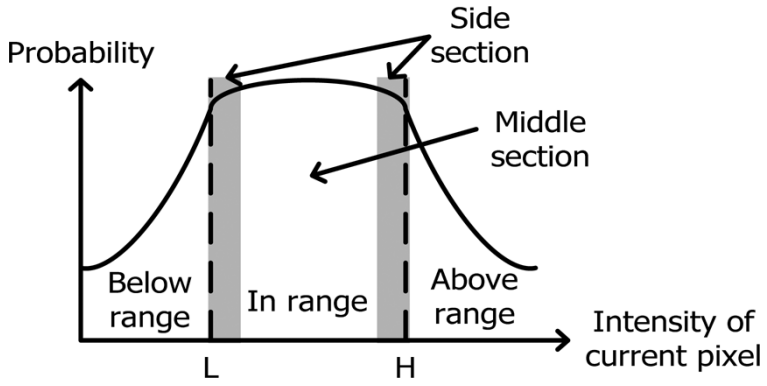


Figure 2.8: *Probability Distribution Model (From T. Tsai, Y. Lee, and Y. Lee, 2010 [19])*

2.5.2.1 Adjusted Binary Coding

When the current pixel value is in range, the algorithm applies adjusted binary coding on $x - L$ in the range $[0, \Delta]$. Adjusted binary coding take advantage of the fact that values in the middle of the range are slightly more probable, and therefore assign shorter codewords for those values.

- If $\Delta + 1$ is a power of two, regular binary code using $log_2(\Delta + 1)$ bits are used.
- Otherwise $lower\_bound = \lfloor log_2(\Delta + 1) \rfloor$ bits are assigned to the middle section and $upper\_bound = \lceil log_2(\Delta + 1) \rceil$ are assigned to the edge values.

To calculate if a value should use $lower\_bound$ bits or $upper\_bound$ bits a threshold and a shift_number are calculated according to Equation 2.15.

$$\begin{cases} range = \Delta + 1 \\ threshold = 2^{upper\_bound} - range \\ shift\_number = \frac{range - threshold}{2} \end{cases} \tag{2.15}$$

The values in the range are circular shifted by $shift\_number$ bits and the $threshold$ tells which numbers should use $lower\_bound$ and $upper\_bound$ bits. An example of this is presented in Figure 2.9.

2.5.2.2 *k* Parameter Selection

The original FELICS algorithm applies a simple adaptive scheme to select k parameters for the Golomb Codes. Since different Δ values incur different exponential decay rates of the OSGD due to diversity in images, FELICS uses a cumu-

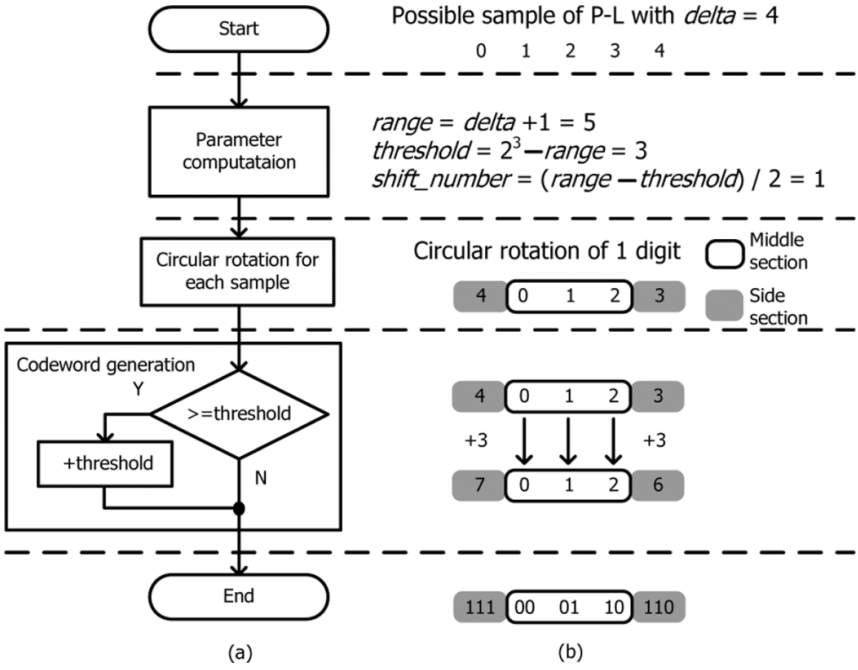


Figure 2.9: *Adjusted Binary Encoding example (From T. Tsai, Y. Lee, and Y. Lee, 2010 [19])*

lation table for the possible Δ-values and selected k-values. During adaptive encoding the k-value that contains the minimum cumulative codeword length is selected as the most efficient k-value. The cumulative codeword length is then updated for each k-value in the table. This method requires a storage of (bit_resolution $\times$ Δ_{max} $\times$ number of k values) bits.

3
Pre-study

To find a suitable algorithm for the specifications of the thesis, a pre-study was done. There are many ways to evaluate and compare compression performance. The first section of the chapter describe how the problem is approached and what methods are used to evaluate the algorithms. This is followed by a description of the algorithms that are evaluated and the results of the evaluation. Lastly, a discussion of the results and a motivation to the selected algorithm is presented.

3.1 Evaluation Approach

As described in Section 1.2 the problem considered in this thesis is if a high-speed lossless compression algorithm can be used to reduce the bandwidth usage of the transmission link in the Ranger3 system. To evaluate which implementations and algorithms that are most suitable for the problem a few properties are studied:

- Compression Ratio: how much the algorithm is able to compress typical range data images.
- Throughput: how much data the encoder can process each second.
- Hardware cost: how much of the available resources on the Z-7030 the implementation would use.
- Memory Cost: how much memory or buffering that is required for the hardware implementation.
- Computational complexity: are the operations required by the algorithm feasible to perform in real-time on a FPGA?

3.2 Evaluation Method

This section present the methods used for evaluation of compression algorithms. At first, an initial evaluation is done on general compression techniques where only the relevant techniques for the system requirements are selected. After this, a more in-depth evaluation of the selected algorithms are presented.

3.2.1 Compression Algorithms

To find suitable lossless image compression algorithms general information was first gathered from introductory books like [11] and [15]. This gave a overview of the most common algorithms and their features and strengths. In addition to this, published papers and conference proceedings relevant to the subject was used. This was the main source for specific hardware implementations used for comparison.

3.2.2 Evaluation of Performance Measures

This section present a more detailed description of the properties that are studied in the algorithm evaluation.

3.2.2.1 Compression Ratio

The compression ratio CR is defined as the ratio of the number of bits required to represent the data before compression to the number of bits required after compression.

$$\text{CR} = \frac{N_i}{N_f} \tag{3.1}$$

The compression ratio of a certain algorithm will vary with the source. For a fair comparison, the average CR over a range of typical images should be considered.

3.2.2.2 Throughput

Throughput is the maximum processing rate of the algorithm. The throughput of a certain algorithm will vary with the processing system. In image compression algorithms the unit of measurement is often **pixels per second**. If throughput were measured in **bits per second** the throughput would vary with the bit-depth. Since the encoder is required to operate on up to a bit-depth of 16, the max throughput could be measured in **bits per second** for this case. One of the system requirements is that the encoder should handle a operational frequency of at least 200 MHz. This means that a throughput of at least 200 MHz $\times$ 16 bits/pixel = 3.2 Gbit/s is required.

3.2.2.3 Hardware Cost

As described in Section 1.2 The hardware cost of the compression algorithm should only use a small percentage of the available resources on the Z-7030 FPGA. While it is possible to look at individual properties like Programmable Logic Cells, Look-Up Tables (LUTs), Flip-Flops, Block RAM usage and DSP Slices, they

are very hard to estimate. Implementations using different hardware from different manufacturers will be considered but is hard to precisely compare. Memory requirements and computational complexity as described below will give a better comparison of the evaluated algorithms.

3.2.2.4 Memory Cost

Contextual pixel-parameters, algorithm parameters, and statistical models are often stored during compression. Since the compression has to be done in real-time at high frequency, a raster-scan ordered algorithm is preferred. Raster-scan order means that the image is processed pixel by pixel, line by line. Some algorithms divide the image into blocks, with each block being processed individually. This however would require buffering of several lines of the image before processing and result in a bigger delay of the output code. There are also two-pass algorithms that process the image twice, building a statistical model of the image in the first pass and encoding each pixel according to the model in the second pass. Algorithms like this is not suitable for the real time application considered in this thesis since it would induce a big delay and require a significant amount of memory.

3.2.2.5 Computational Complexity

The computational complexity of each pipeline stage has to be simple enough to be able to run at the desired speed. Operations like division and multiplication are slow unless implemented in a pipelined fashion or using specific DSP-cores, and often not suitable for lossless operation. Therefore algorithms that can be realized with simple logic and arithmetic operations are preferred. In addition, algorithms dependent on a dynamic model which update during compression could introduce throughput limiting data dependencies in the pipeline. [3]

3.3 Intermediate Results

Lossless image compression techniques can be divided in two categories, prediction based and dictionary based techniques.

There are many well developed dictionary-based compression algorithms including, LZ77-LZ78[5], LZW, arithmetic coding[10] and huffman coding[7]. In these methods, repetitive and frequently occurring patterns are assigned shorter codewords. A table of codewords is created based on the statistics of the image and then used to encode the image. The probability table is usually created by two-pass methods where the image is scanned twice (or more) to build a statistical model on which the image can be compressed. Another approach is to use a fixed probability table which is created based on assumptions of the images the encoder will handle. Although there are implementations of these type of algorithms that present high throughput and good compression, the hardware cost, memory cost and computational complexity is to high and wont fit the requirements of this thesis. [5][11]

The prediction-based algorithms exploit the spatial redundancy in images by predicting the current pixel value based on previous pixels and then coding the prediction error. Since each prediction is only based on earlier pixel parameters in a raster scan order, the encoding can be done in one pass. Prediction based algorithms considered are, LOCO-I [20], CALICS [21], FELICS [6], SFALIC [17] and SZIP [2].

LOCO-I is the algorithm used in the JPEG-LS compression standard. The algorithm present good compression with a low hardware cost and complexity. The algorithm is adaptive, which means there are memory requirements but compared to the dictionary based encoders it is fairly low. [3][20]

CALICS is an ambitious algorithm which use a large context of pixels to model the gradient of the current region. This is used for accurate prediction of the current pixel. An additional bias cancellation method is employed as well to further correct the prediction. Even though this algorithm have shown very good lossless compression of images the encoding speed is very low, making it unusable in the Ranger3 system. [21]

FELICS is designed to be a simple and efficient system with minimal loss of performance compared to other predictive encoding schemes. The compression is slightly lower compared to CALICS and LOCO-I but the hardware cost, memory cost and computational complexity is very low. There exist hardware implementations which present very high throughput. [6][19]

SFALIC is another predictive and adaptive encoding algorithm using a simple model with good compression results and throughput. However the lack of hardware implementations make it hard to evaluate if the requirements would be fulfilled.[17]

SZIP use the techniques introduced by Rice of adaptively selecting between a range of coders depending on their performance for the current sample. Prefix codes are used to indicate which coder is used and a predictive scheme is used to calculate the residual. This encoder divide the image into blocks where all pixel in the same block is coded using the same coder. The Consultative Committee for Space Data Systems (CCSDS) has recommended this as the standard for lossless data compression in space missions.[2]

Teledyne DALSA Inc. which is a company working with high performance digital imaging have a patented technology called Dalsa TurboDrive. The technology is used in a very similar setup as the one considered in this thesis, where a hardware encoder is introduced to compress images to reduce the bandwidth usage of a Ethernet link. They employ a simple differential scheme where the difference of pixels in the vertical direction is encoded using a smallest number of bits needed to represent the difference of the biggest and smallest residual of a segment.[13]

3.3.1 Initial Selection

The properties of the most interesting compression algorithms are summarized in Table 3.1. Since most implementations and references are designed for 8-bit

images, the hardware resources presented would be significantly increased for a 16-bit implementation. This is taken into consideration in the selection of algorithms. The two algorithms LOCO-I and FELICS are selected as the most suitable candidates because of their good compression, low hardware footprint and available high speed implementations.

LOCO-I which is the standard recommended in JPEG-LS, is a widely developed and tested image compression system. When implemented in hardware the main problem is the data dependencies between pipeline stages when the adaptive context model is updated, reducing the throughput of the encoder significantly. Many publications try to work around this problem by stalling the pipeline or using special conflict detectors, however very few implementations reach a throughput of 200 MPixel/s. [3] The run mode of the JPEG-LS standard is often omitted in hardware implementations.

The FELICS algorithm has some of the most resource efficient implementations while still reaching good compression ratios. [19] With its simple, combined prediction, modeling and Golomb-Rice coder it is easy to adapt and implement in a pipelined architecture. T.Tsai et.al present a parallel architecture achieving a throughput of 546 MPixel/s (273 MPixel/s per parallelism) in a TSMC 0.13μ m technology. This implementation suggest a static k parameter for the Golomb coder, resulting in the removal of data dependencies in the pipeline and cumulation-tables for the parameter.

Table 3.1: *Summary of relevant algorithms and their properties. All implementations referenced are designed for 8-bit images. Performance numbers gathered from [12][4][8][3][19][14][2]*

Algorithm	Standard	CR	Throughput [MB/s]	Hardware Cost	Memory Cost	Computational Complexity
LOCO-I [12][4][8][3][14]	JPEG-LS	2	51, 113, 155, 196, 265	Low	Medium	Low
CALICS [21][14]	-	2.1	5	-	-	Medium
FELICS [19]	-	1.9	273	Low	Low	Low
SFALIC [17]	-	2	60	-	Low	Low
LZMA [14]	7ZIP	1.6-2	125	High	High	Med
SZIP [2][14]	CCSDS	1.9	410	-	Med	Low
LZW [14]	GIF	1.3-1.5	198	-	High	High

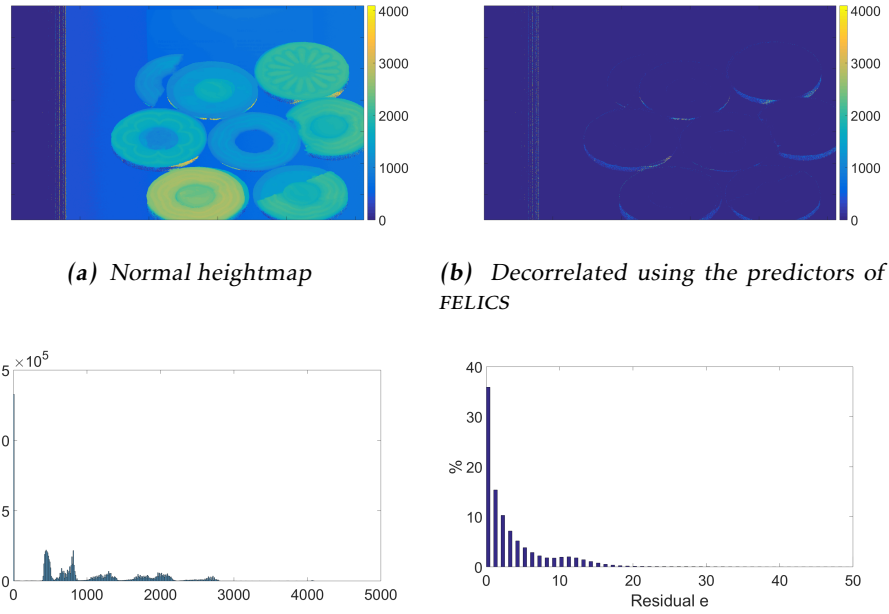
3.3.2 Further Evaluation

To further evaluate the JPEG-LS and FELICS algorithms and their performance on 3D range data images, a set of typical images from the Ranger3 camera was used to study the potential compression. The images used in the evaluation is **Train**, **Camera**, **Cookies** and **Fish** which all can be seen in Appendix A.2.

The prediction model used in FELICS and LOCO-I was implemented in Matlab.

The adaptive correction of the prediction residual used in LOCO-I was not used. An example of a test image and its decorrelated version is depicted in Figure 3.1b. From figure 3.1d it is clear that the decorrelated image has a exponentially decreasing distribution of parameters which is why Golomb codes are used.

When comparing the static decorrelation using the model of FELICS and LOCO-I, Figure 3.2 it can be seen that the PDF of FELICS has a faster decay rate, especially for small residual parameters. This is true for all tested images. The reason is partly because a sign bit is used to invert negative residuals compared to LOCO-I which use the mapping function in Equation 2.14. With the residuals more spread out, the algorithm becomes more dependent on dynamic adaptation of the k parameter used in the Golomb codes. FELICS on the other hand, with most of its residual values in a smaller range, will not be as dependent on the selection of the k parameter.



(a) *Normal heightmap*

(b) *Decorrelated using the predictors of FELICS*

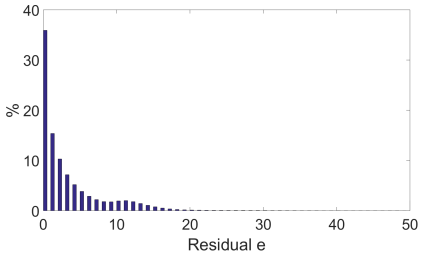
(c) *Histogram of normal heightmap*

(d) *Histogram of prediction residuals ε. x-axis clipped at x=50.*

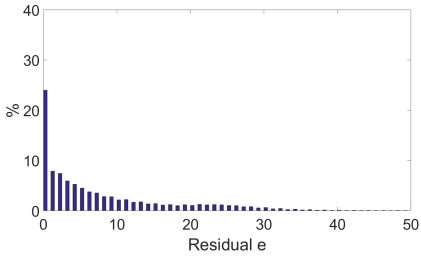
Figure 3.1: *The image* ***Cookies*** *before and after decorrelation using the FELICS prediction model.*

3.3.3 Entropy

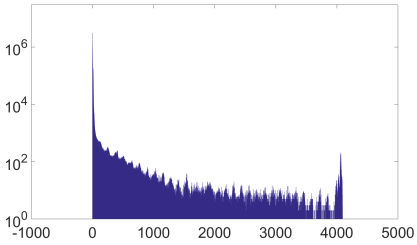
As described in Section 2.4.1, the entropy of an image gives a lower bound for how much the image can be compressed using a entropy based lossless compression scheme. This was used to further analyse the Matlab models of FELICS and LOCO-I.



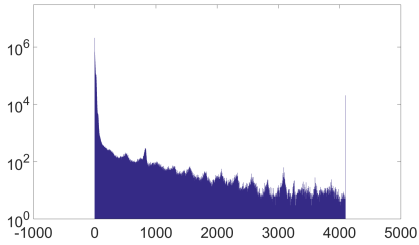
(a) FELICS *clipped at x=50*



(b) LOCO-I *clipped at x=50*



(c) Full historgram in logarithmic scale, FELICS



(d) Full historgram in logarithmic scale, LOCO-I

Figure 3.2: *Comparison of static decorrelation using predictors of* FELICS *and* LOCO-I *on the image* ***Cookies****.*

By dividing the bit depth with the entropy we can get an approximate limit of the compression ratio CR. In the FELICS algorithm the residuals are mapped in different regions using either one or two bits for in range or outside of range. These additional bits was adjusted for by analysing each test image how the residuals are distributed in these regions and adding the proper number representing the index bits to the entropy. The resulting potential compression ratios using a entropy based encoder is depicted in Figure 3.3. In addition to the LOCO-I and FELICS models a model representing the scheme of Dalsa TurboDrive was used. This model simply use the vertical difference of profiles in the image.

In all 4 test image the three models used produce a similar potential compression ratio, with the FELICS model lagging slightly behind the two others. This is believed to be mainly because of the second index bit used when pixels reside out of range. The **Camera** image is very compressible with a potential CR of 2 without decorrelation. This is caused by the large amount of missing data in the image. Since the result only represent that of entropy based encoders, the compression of the **Camera** image is not significantly better that the others. Using a coder like RLE could result in a significantly better CR in this case.

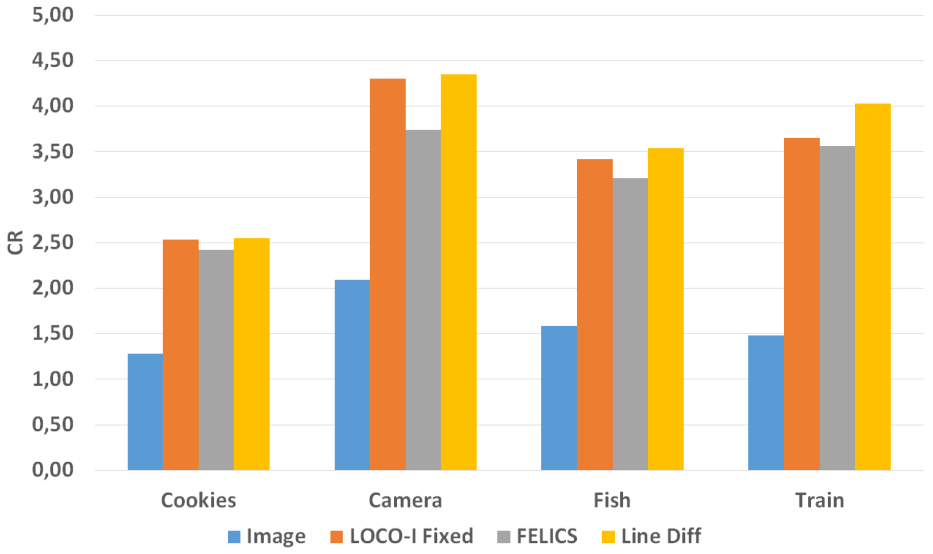


Figure 3.3: *Potential compression of test images. FELICS adjusted for index bits.* ***Image*** *represent the original image without decorrelation.*

3.4 Selection of Algorithm

The selected algorithm is predictive and static and is designed to compress Range data images. The algorithm is based on the FELICS algorithm with a few changes. The adaptive entropy coding in FELICS are replaces by Golomb-Rice codes with static k-parameters. Since a static k-parameter is used, the Golomb codes can be very long for large residuals if the k-parameter is selected poorly. To remove these bigger codes a maximum code length is added similar to what is done in JPEG-LS [20]. This limits the unary codes to a maximum bit length of q_{max}. When q_{max} is reached, the pixel value is encoded uncompressed, instead of the residual. A similar technique is used in the Rice coder in SZIP.[2] The main difference of Range data compared to natural grayscale images are the sequences of *missing data*. To encode a sequence of *missing data* efficiently, a RLE module is added to the algorithm. When a run is identified, the pixels in the run are encoded using RLE instead of the predictive model. This is similar to the RLE used in JPEG-LS but will have to use a different context determination for "flat" regions.

The FELICS prediction model is selected mainly because of two properties it had which differ from LOCO-I. Firstly, the context used for prediction is easily adapted to only depend on pixels on the same line. With the regular predictors used in FELICS and LOCO-I, the decoder will always be dependent on pixels from previous lines, meaning that parallel decoding of profiles will not be possible. With these dependencies removed, multiprocessesing could be used to improve decoding speed. The GigE Vision protocol will send the compressed image in packages.

In the case of packet loss, the protocol allow for retransmission. With the inter-line dependencies removed, a packet loss does not mean that the rest of the image is lost, only the affected profiles. This allow the decoder to work without retransmission, or while waiting for retransmission. Secondly, the faster decay rate of the PDF for the residuals make the FELICS algorithm less dependent on a dynamic selection of k-parameter. This could allow good compression even when using a static model which will reduce both memory requirements and data dependencies in the pipeline.

From Figure 3.4 it can be seen that by using only case 1 and 2 of the FELICS prediction model, the potential compression ratio based on the image entropy is only reduced slightly. The average loss over the 4 test images is 9%.

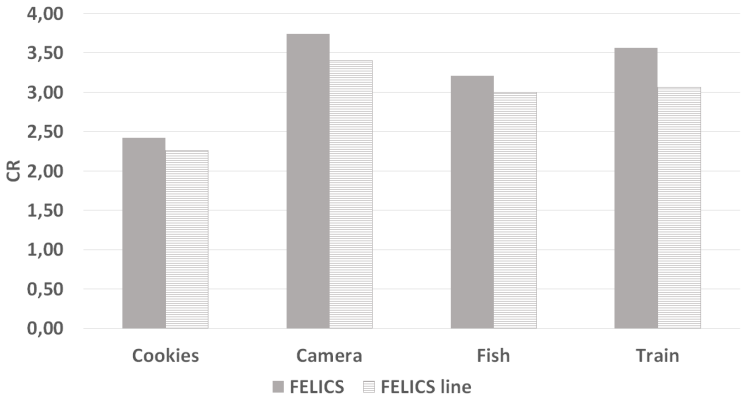


Figure 3.4: *Comparison of FELICS predictor vs line independent predictor using only the two previous pixels parameters as predictors.*

4

System Modeling

The first stage in the design process consisted of developing a software encoder and decoder to verify the expected performance of the algorithm. C++ is used to implement the algorithm and the software is tested on typical images from the Ranger3 camera. The finished software implementation can load images of the .raw or .bin file format and will output an encoded .bin file with the encoded version of the image. After this the algorithm will decompress the image again and verify that the reconstructed image is identical to the original. In addition a performance evaluation of the compression and decompression is output in a log file.

Referring to Section 3.4, the algorithm is based on FELICS and LOCO-I as described in Section 2.5.2. The algorithm is designed to be simple, fast and resource efficient while still exploiting compressible properties of 3D range data. A efficient coding principle is used without larger data dependencies while still maintaining competitive coding efficiency. To conclude, the compression consist of two main stages, prediction/modeling and entropy coding. The modeling stage tries to reduce the entropy, i.e. increase the information density. This is done in a causal way so that the process is reversible by the decompression algorithm on the host computer. The modeling stage uses the two preceding pixel values on the same line as a context to calculate a prediction of the current pixel. The prediction error is then encoded in the coding step using either Simplified Adjusted Binary Coding (SABC) or Golomb Rice Coding (GRC) depending on the context. In addition the system can adaptively switch to **run mode** when the context consist of only missing data. The source coder will output bit-strings of variable length, and a **bit-packer** is needed. The bit-packer packs the incoming bit-strings into fixed length words which can be output in the final code. Since the system compress images in a line-independent scan order, the total resulting code length of

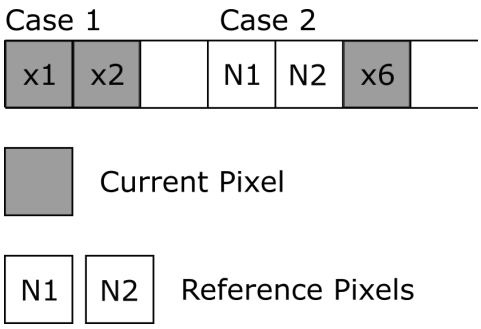


Figure 4.1: *F* prediction model*

each line is output together with the compressed code. This removes data dependencies between lines and allows a decoder to operate on multiple lines in parallel.

4.1 System Description

For reference, the compression system presented in this thesis will from now on be called F* (temporary name). This section present a more in depth description of the F* algorithm. Section 2.5.2 describe the algorithm on which most of the system is based and Section 3.4 explain the reasoning to why this algorithm is chosen.

4.1.1 Modeling Stage

With lines encoded individually the prediction model has to be adjusted slightly compared to the original prediction model of FELICS. The solution is to only use Case 1 and 2 from the prediction template in Figure 2.7. The first two pixels of a line is passed unencoded and the rest of the pixels use the two closest preceding pixels as the context model, see Figure 4.1. Original FELICS used the experimentally verified assumption that the PDF depending on its predictors would look like Figure 2.8. Since the predictors in F* only depend on the previous two values on the same line, the assumed probability distribution function is adjusted to Figure 4.2. One could assume that the value of x should be closer to N_1 than N_2 for a continuous image, which would result in the dotted line in Figure 4.2. However in F* the probability of values in range are coded with SABC [Section 4.1.2.2] which does not exploited the specific distribution. The probability distribution in range can therefore be considered as flat.

With the new context model, the prediction residual ε is calculated using the same steps as described in Section 2.5.2. The residual ε is calculated according to Table 4.1.

In addition to the residual calculation a method to detect flat regions for adaptive run length encoding is introduced. The context used to determine if run mode

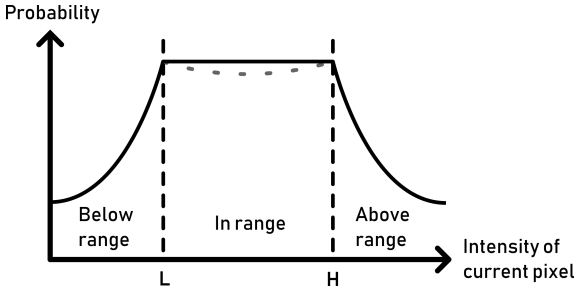


Figure 4.2: *F* Probability distribution of values depending on its predictors*

Table 4.1: *Residual calculation in different contexts*

Context	ε
In Range	$x - L$
Below Range	$L - x - 1$
Above Range	$x - H - 1$

should be used is the same as above. The two closest preceding pixels in the same line together with the current pixel intensity is used to determine whether the encoder should enter run mode. If $N1$, $N2$ and x all are 0, the encoder will enter run mode, starting with the next pixel on the line. This is explained further in Figure 4.3.

To summarize, the modeling stage consist of

1. Calculate L, H and Δ from $N1$ and $N2$.
2. Determine where x reside in the context of L and H.
3. Calculate the residual ε.
4. Determine which coder should be used in the coding stage.
5. Determine if run mode should be activated for the next pixel.

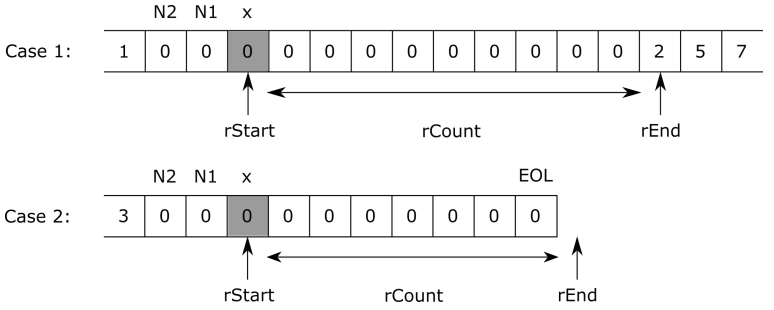


Figure 4.3: *F* run mode context*

Table 4.2: Index code and coder depending on context

Context	Index Code	Coder
In Range	0	SABC
Above Range	10	GRC
Below Range	11	GRC
Flat ($x = N_1 = N_2 = 0$)	-	RLE

6. If in run mode:
    - Check if the run has ended or end of line is found.
    - Exit run mode and output $rCount$ if the run mode has ended.
    - Increment $rCount$ otherwise.

4.1.2 Source Coding

The purpose of the source coder is to receive the binary residual from the modeling stage and encode the data with as few bits as possible. Consequently, the output of the source coding stage is of variable bit width. When implemented in hardware, output codes will always be of the same bit width, and instead only a set of those will be valid. To specify which bits are valid another positive integer output is used to indicate the bit width of the current codeword.

The source coding techniques used in F* are Golomb-Rice Coding (GRC), Simplified Adjusted Binary Coding (SABC) and run length coding (RLE). Which source coding scheme is used for a specific pixel is encoded with a index code of either one or two bits, see Table 4.2. When the pixel x reside in range the index code is `0` and SABC is used. When the pixel reside below range or above range the index codes used are `10` and `11` respectively and Golomb-Rice codes are used. No index code is needed for the run length mode.

4.1.2.1 Golomb-Rice Coding

The Golomb-Rice coding used in F* are very similar to the one used in FELICS and LOCO-I. The basis is described in Section 2.4.2. The main difference in F* is that a static k-parameter is used, meaning that the k-parameter has to be the same for all contexts of the image. The k-parameter can however be adjusted in setup since images will vary in different applications and the most suitable k-parameter for that application can be selected. F* also implement a maximum code length of the unary codes, since large residuals could result in a very large codeword if the k-parameter is not selected properly. The maximum code length of the unary code is called q_{max}.

Definition 4.1. $q_{max} = N - 2$, where N is the bit-depth.

Since the index code used in the Golomb-Rice encoder is of bit length 2, we define q_{max} according to Definition 4.1 so that the index code followed by the unary code

has a maximal length equal to the bit-depth. The q_{max} code is not followed by a zero at the end like the unary code does.

When q_{max} is reached, instead of following the unary code with the remainder, we send the original pixel value with its regular binary representation. For example if the bit-depth is 8, the current pixel value $x = 100$, $\varepsilon = 25$, $k = 0$ (bad selection in this case) and we are below range, the codeword would be:

$$\begin{cases} index = 10 \\ q_{unary} = q_{max} = 111111 \\ r_{bin} = x = 01100100 \end{cases} \Rightarrow \begin{cases} codeword = 1011111101100100 \\ codeLength = 16 \end{cases}$$

With regular Golomb-Rice and the same parameters, the codeword would be:

$$\begin{cases} index = 10 \\ q_{unary} = 11111111111111111111111110 \\ r_{bin} = 0 \end{cases} \Rightarrow$$

$$\Rightarrow \begin{cases} codeword = 10111111111111111111111111100 \\ codeLength = 29 \end{cases}$$

In other words, q_{max} sets a limit to the maximum codeword length and with Definition 4.1, the codewords will never be longer that two times the bit-depth.

4.1.2.2 Simplified Adjusted Binary Coding

In FELICS, Adjusted Binary Coding is used as explained in Section 2.5.2.1. The adjusted binary codes assume the probability distribution of Figure 2.8 which is no longer relevant with the new predictors. In F* this is therefore simplified further and will remove the required rotational shift-operations of the original FELICS. The coding scheme used instead is called SABC (Simplified Adjusted Binary Coding) and was introduced by T.Tsai et.al [19].

The regular Adjusted Binary coding would require three procedures: parameter computation, circular rotation, and codeword generation. These operations could be translated to arithmetic operations suitable for hardware implementation. However the processing speed could be seriously limited. [19].

In SABC, the binary value is always coded using *upper_bound* bits instead of only in the edge cases where x is close to L or H. This means that in some cases, a suboptimal code using one extra bit to represent the codeword is used, compared to the original FELICS.

4.1.2.3 Run Length Coding

F* include a run mode similar to that of JPEG-LS. Run mode in F* is only used for missing data since the probability of longer runs of the same value is very low for pixels values other than zero. Even very flat surfaces will have some noise and inconsistencies in the height-map. This is experimentally proven using a range

Table 4.3: Runs of zeroes compared to runs of any value. Bit-depth 12

Run Length	**4**			**9**		
Image	**Any**	**Zeroes**	**%**	**Any**	**Zeroes**	**%**
Train	268	104	38.8	96	79	82.3
Camera	7151	6495	90.8	4796	4796	100
Fish	107147	86210	80.5	42347	42067	99.3
Cookies	21787	15882	72.9	11791	11770	99.8
Average			**70.8**			**95.4**

of test images as can be seen in Table 4.3. It can be seen that for longer runs where RLE is advantageous over regular GRC, most of the runs are runs of zeroes. Runs of values other than zero are often so short that RLE is not advantageous, especially considering that the first 3 pixels of the run will still be coded without RLE.

How run mode is activated is explained in Section 4.1.1. When the encoder enter run mode it will simply count the number of pixels between `rStart` and `rEnd`. When `rEnd` is reached, case 1 in Figure 4.3, the current pixel x will be encoded in the normal way and the cumulated run count will be encoded as a regualar binary value. In case 2 when end of line is reached, the encoder will go back to normal mode and the run count stopped and encoded.

The Ranger3 camera has a maximum line width of 2560 pixels, which means that the maximum length of a encoded run is 2557 since 3 pixels are needed to determine `rStart`. Because of this the run length is always encoded as a binary value using $\lceil log_2(2557) \rceil = 12$ bits.

The decoder will decode lines from left to right and F* won't need any index-bits telling the decoder that RL-mode is active. By looking at the last 3 decoded pixels, the decoder will know when a RL-value is expected.

4.1.3 Data Packer

Since the source coder output variable length bit strings a packer is used to concatenate these bit string into fixed size words. The maximal length of a codeword is two times the bit-depth as described in Section 4.1.2.1. The encoder is designed to handle a bit-depth of up to 16 bits, meaning that the maximal codeword length is 32 bits. Therefore the bit packer is designed to pack and output words of constant width 32. In order to avoid buffer overflow in the bit packer, the buffer has to be at least 64 bits wide.

As can be seen Figure 4.4, when a complete word of 32 bits is available, it is output and the buffer is shifted, leaving only the bits that were not output. This way the encoder will always output words of constant bit width.

Since lines of the image is encoded individually, the special case when a line ends has to be handled. When end of line is reached, there are three different cases

that can happen. These cases and their solution is described below:

1. A complete 32 bit word is not available.
    - Concatenate zeroes at the end of the line until a full 32 bit word is available.
2. Exactly one 32 bit word is available.
    - Output the normal way.
3. More than a 32 bit word is available.
    - Output the first 32 bit word as normal.
    - Output the second 32 bit word with concatenated zeroes at the end.

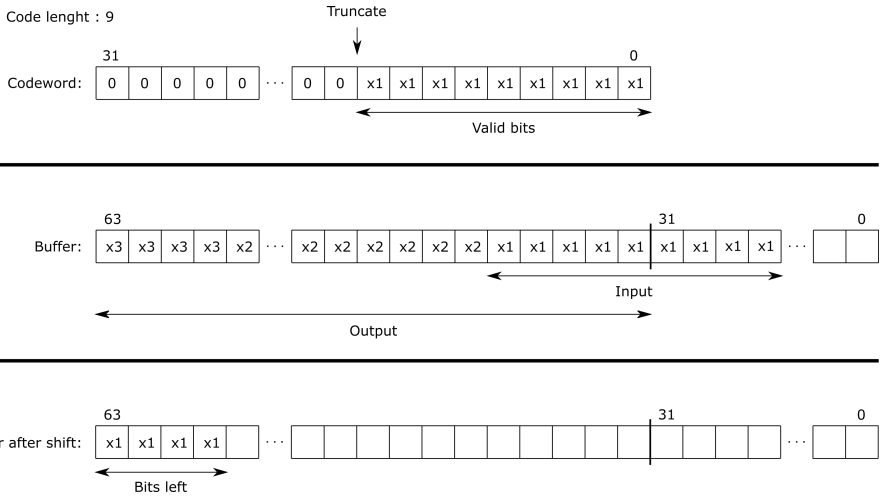


Figure 4.4: *F* Bit packer*

4.1.3.1 Code Size Counter

As stated in Section 4.1.1, F* encode the image in a line-independent raster scan order.

Every time a 32 bit word is output a counter is incremented. When a line has been encoded, the total count for the specific line is stored in a separate vector, other than the one containing the codes. This vector is sent as chunk data in the GigE-Vision link together with the encoded image data.

The software decoder will receive the compressed image together with the line length vector. This way the decoder can find individual lines in the compressed bit string, and decompress them in parallel. This is explained further in Section 4.1.4.

4.1.4 Decoder

So far the system has been explained with the encoder in mind. From this description a decoder is fairly straightforward to design. The steps used in the encoder, modeling, source coding and bit packing, is simply reversed.

The decoder will receive the string of compressed data together with a vector containing the code size of each line. The vector containing the code sizes is used to find the start of each line. After the start of a line is found, it can be decoded individually from the others. The decoder does not need to know the size of the code since it will always know how many pixels has been decoded so far and the expected line width is always known beforehand.

4.2 Software Implementation

After the system was modeled, a software implementation of both the encoder and decoder was created in C++. The encoder is created so that the system can be evaluated further before a hardware implementation is created. The decoder will be used to verify the functionality of both the software and hardware encoder. This chapter only describe the most relevant parts of the software implementation. Parts left out are considered to be easily implemented according to the description in Section 4.1. All the code was written using C++ and standard libraries with one exception. To allow easy multiprocessing of the decoder, OpenMP was used. OpenMP is a API which allows for shared memory multiprocessing programming in C++. The use of this is described further in section 4.2.2. The code was compiled using G++ and C++17.

The encoder read RAW or binary image files and encode them using the proposed encoder. The resulting compressed image is then written to a binary file. No specified file format including header information about the compressed image is used. This means that the encoder need to know information such as image width and height, k-parameter used and if run length encoding was enabled.

4.2.1 Encoder

A flow chart describing the encoding of a line is depicted in Figure 4.5.

To store the image and resulting code, the container `std::vector<uint16_t>` is used. When loading a image file, each pixel value will correspond to one element in the vector, even if the bit depth is only 8 or 12 bits. The resulting code vector use the same container. The new codes created by the encoder is shifted into a `uint64_t` buffer according to Listing 4.1. As soon as the buffer contains at least 16 valid bits, they are pushed into the `OutCode` vector. When a full line has been encoded, the size of the `CodeOut` vector is saved in another vector `line_length` containing the size of all the lines in the image.

```
currentCode <<= newCodeLength;
currentCode |= newCode;
currentCodeLength += newCodeLength;
```

Listing 4.1: *Update current code and length*

4.2.1.1 Source Coding

The `newCode` and `newCodeLength` are created by one of the 3 different coders. In the Golomb Rice coder, q and r are calculated according to Listing 4.2, where N is the residual.

```
uint16_t q = N >> k;
uint16_t r = N % (1 << k);
```

Listing 4.2: *Calculate q and r*

The unary code is then created according to Listing 4.3.

```
uint16_t unaryEncode(uint16_t q, uint8_t &length)
{
  uint16_t u = 0;

  if (q >= MAX_Q)
  {
    maxQReached = true;
    q = MAX_Q;
  }
  while (q--)
  {
    u = (u << 1) | 1; //Shift left and add one (=shift in a '1')
    length++;
  }

  length++;
  u = (u << 1); //add 0 at the end

  return u;
}
```

Listing 4.3: *Create unary code*

4.2.2 Decoder

The decoder follow the flow chart of Figure 4.6. The processing steps are reversed from the encoder. The compressed image containing variable length codewords has to be decoded bit by bit. Similar to the code buffer in the encoder, a `uint64_t` is used as a buffer where 16 bit words are appended so that there are always at least 32 bits available for every pixel loop. When a pixel has been encoded it is pushed into a std::vector<uint16_t>. When all lines have been decoded, this vector can be written into a RAW or binary image file.

As mentioned before, OpenMP was used to allow parallel decoding of lines. This was used according to Listing 4.4.

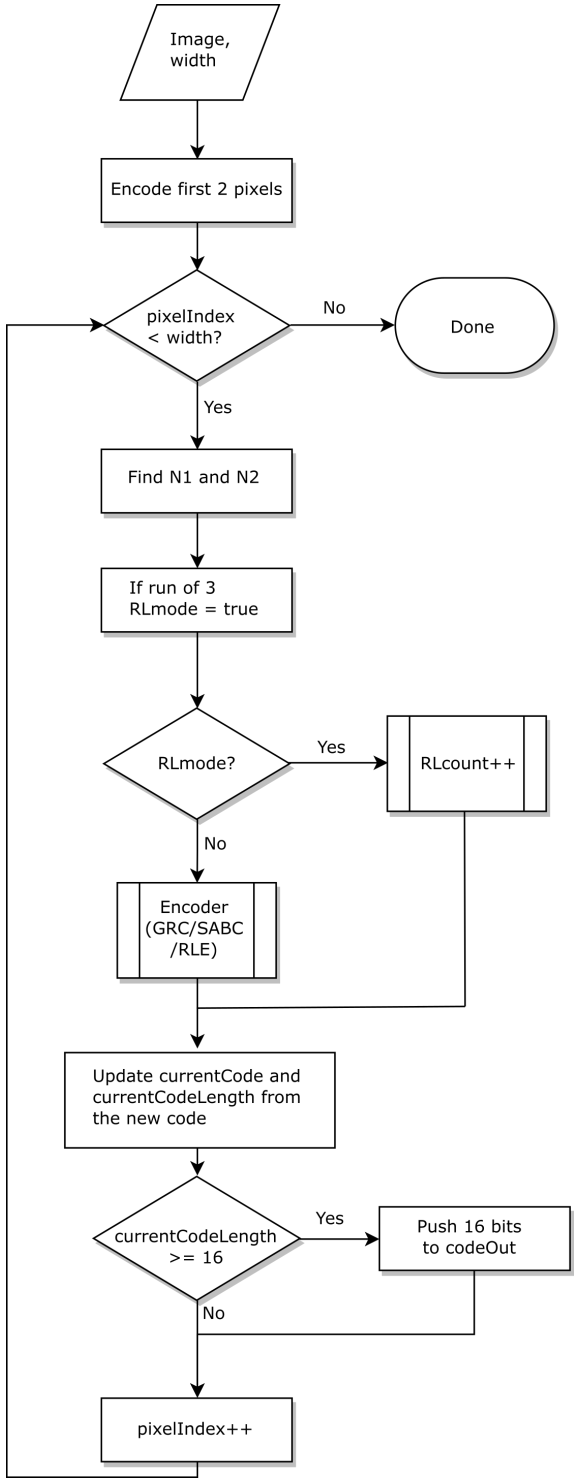


Figure 4.5: *Flowchart of main encoding loop*

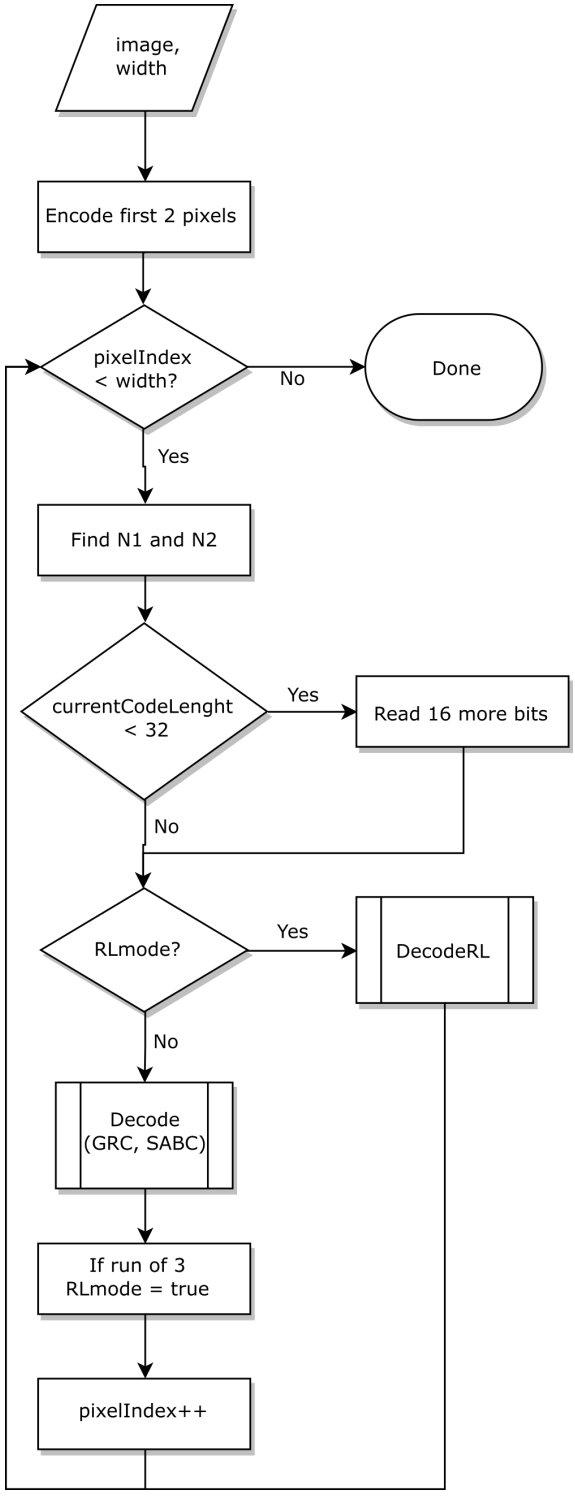


Figure 4.6: *Flowchart of main decoding loop*

```
#pragma omp parallel for num_threads(4)
for (int i = 0; i < im.height; i++)
{
  felics::decodeLine(codeOut, lineLength, newImage[i], im.width, i, k,
  RLE);
}
```

Listing 4.4: *Parallel execution of decoder using OpenMP*

4.2.3 Test and Verification

To test the performance of the software encoder and decoder, a number of images and parameters were used. The hardware of the computer running the tests can be seen in Appendix A.1. The test consisted of the following steps.

1. Load the RAW image into a vector.
2. Loop over and encode each line in the image.
3. Write the resulting bit stream into a binary file.
4. Read the file again.
5. Decompress the lines of the image in parallel using 4 threads.
6. Write the reconstructed image to file.
7. Compare the original and reconstructed image and verify that they are equal.
8. Report encoding/decoding time, compression ratio etc.

Listing 4.5 show an example of the output from the test program. The comment saying that the image size is adjusted for 12 bit is because of the fact that the 12 bit images will be stored in a binary file using 16 bits per pixel value. This means that 4 extra zeroes are padded onto each pixel, resulting in a larger image than what is required. In the Ranger3 system, these pixels will be packed together which will result in the *adjusted* image size. The `real` image size is therefore not as interesting as the adjusted. In the case of 16 or 8 bit images, this is not something that has to be adjusted because binary files can be stored using 8 or 16 bit values.

```
$ ./TestImage.exe camera12x2560x1000.raw 2560 1000 12 2
Compression Results
------------------------------------------------------------------------
Image                          : camera12x2560x1000.raw
Width                          : 2560
Height                         : 1000
Resolution                     : 12
k-value                        : 2
RLE                            : 1
Encoding time                  : 429.851 ms
Average line encoding time     : 0.429851 ms
Total Decoding time            : 23.935 ms
Average line decoding time     : 0.023935 ms
Profile decoding frequency     : 41.7798 kHz
Input image size               : 3840 kB
Output code size               : 878 kB
Compression Ratio              : 4.37278
Average Code                   : 2.74425 bits/pixel
OBS, the image size is adjusted for 12 bit! Real image size is: 5120 kB,
    CR: 5.83037
```

Listing 4.5: *Example of test result*

4.2.4 Test Images

To evaluate the system further, 10 different test images were used to get averages from images with different complexity and properties. The images used can be seen in Appendix A.2. The parameters that can be varied in the system is the k-parameter for the Golomb encoder and the enable of the run Length encoder. For each image, a sweep of these parameters was done to find the optimal settings for each test image. The k-parameter was tested in the range $[0, N]$ and run length mode on/off.

4.3 VHDL Implementation

Using the system level model and the software implementation, a hardware model of the encoder was designed. The system is divided into 4 main stages, *input*, *intensity processing*, *source coding* and *packing*. This chapter describe how these stages are designed and implemented in VHDL. A description of how each of the modules are tested is included as well. Testing of functionality is very critical since the output of the hardware has to be exactly the same as in software in order for the software decoder to be able to decode the compressed images.

4.3.1 Top level

A figure of the top level design can be seen in 4.7. The design consist of a 5 stage pipeline, with the data packer divided into two steps. The static prediction model and golomb parameters allow the design to be fully causal, meaning each pipeline stage only depend on signals from previous stages. With this design, each individual module could be pipelined further if the speed requirements are not met, as long as the other modules in the same stage are delayed accordingly.

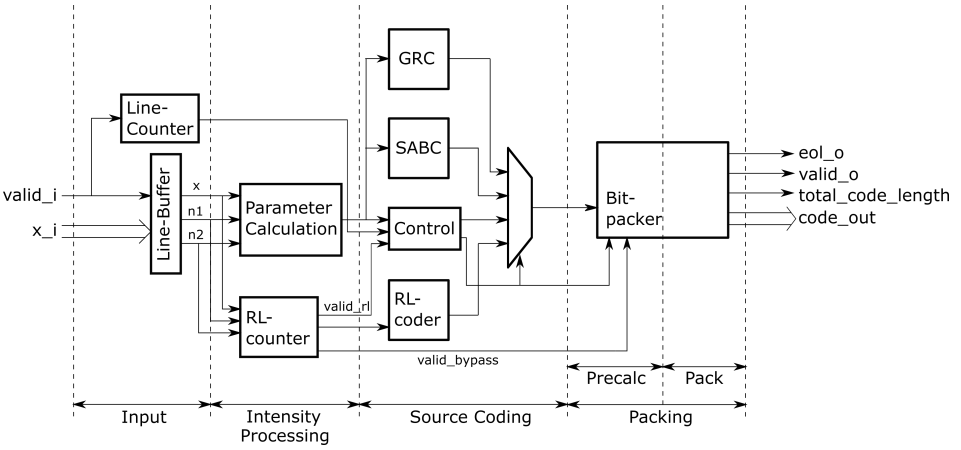


Figure 4.7: *F* System Overview*

The system has 4 parameters which is defined in setup and is not changed during encoding, these are `line_size`, *k*`_val`, `rl_enable` and `resolution`. These signals has to be set before encoding and is not changed during encoding. Since there is no codes or header information used in the encoder, telling what these control signals are set to, the decoder has to know the configuration used to be able to decode the image. The run mode can be disabled by setting `rl_enable` low. The pixel resolution and line size can be set with `resolution` and `line_size` respectively. Which *k* value to use for the golomb encoder is not easy to know beforehand. If the *k* value is chosen poorly, the compression will not be as good. Later in Section 6.2.5 a method for selection of *k* value is discussed. A good default value is also suggested based on experimental results using typical images.

The encoder has a simple input interface with a `valid_i` signal indicating each cycle that a valid pixel is present on `x_i`. The system is designed to handle max throughput, meaning that a new pixel can be input each clock cycle. The system will output 32 bit words with a similar `valid_o` signal indicating each new code out. The extra signal `eol_o` is used to indicate when a line ends and when the `total_code_length` is updated, indicating the total code size of the last line.

4.3.2 Input

This stage consist of two sub-modules; a pixel counter and a pixel buffer. The purpose of the pixel buffer is to store and hold older pixel values that are needed in the prediction module. Since the implemented algorithm only need the two preceding pixel values, the pixel buffer is simply a 3 stage shift register, storing 16 bits per stage with outputs from all its stages.

The pixel counter will keep track of where in a line the encoder is. The stream of pixels entering the encoder does not have a specified line break, therefore the encoder has to know the expected line width and keep track of when a new line

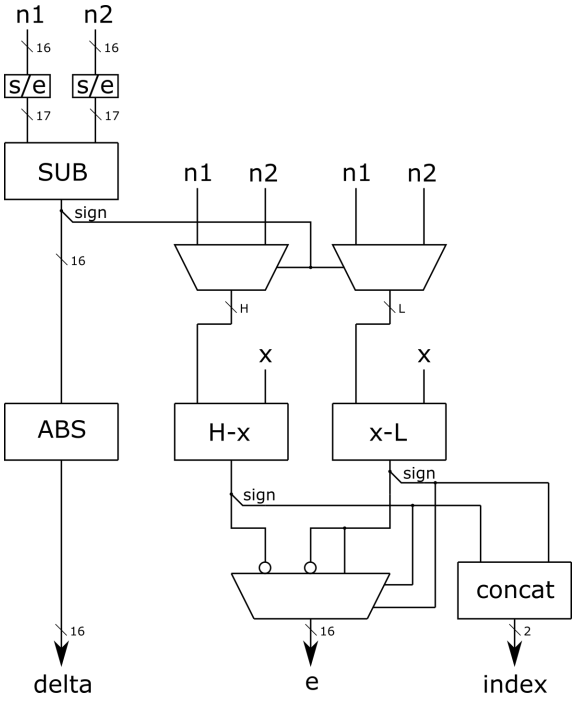


Figure 4.8: *Architecture of intensity processing module*

should start. On the last pixel of a line the signal `eol` is set **high** to indicate other blocks of the system which are sensitive to this. In addition the pixel counter use the signal `no_encoding` to indicate if the incoming pixels should not be encoded at all. This is used for the first two pixels of each line since they should be left unencoded. The line width used by the encoder is known to the pixel counter through the `line_size` signal.

4.3.3 Intensity Processing

The next stage of the encoder is responsible for the intensity processing and selection of encoder. The intensity processing module take x, $n1$ and $n2$ and calculate the residual ε, Δ and the index codes used for coder selection. This module is divided into smaller blocks according to Figure 4.8. The design of this was inspired from [19] and show how block propagation can be minimized. In the FPGA implementation however the VHDL code will be synthesized into equivalent logic using LUTs and Adders. The module require only 2 levels of adders and some signal selection. However if even higher speeds are required, additional pipeline stages could be implemented here.

In addition to the intensity processing block, a run length counter is used to detect and count runs of missing data. The run length counter also serve as a control module telling the coding stage when valid signals are present at the next

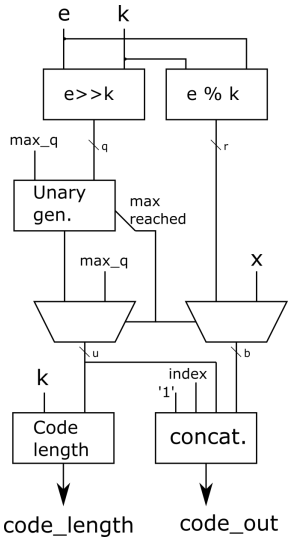


Figure 4.9: *Architecture of Golomb Rice coder*

pipeline stage, the coding stage.

When the run length counter detect $x = n1 = n2 = 0$ it will enter run length mode and start counting the run according to Section 4.1.2.3. When in run length mode, the signal `rl_mode` is used to tell the rest of the system that run length mode is active. When a run ends, `valid_rl` is pulsed **high** and the counter returns to normal mode. In normal mode `valid_i` is just bypassed to `valid_o`.

4.3.4 Source Coding

In the coding stage of the hardware encoder, four different modules are present. All the coders will react to their input and produce a codeword and code length. The control block will select the correct output. Each block is described in more detail below.

4.3.4.1 Golomb Rice Coder

The golomb rice coder receive x, e and the index for above/below range and will produce the golomb code and code length. Section 4.1.2.1 explain the specifics of the Golomb Rice coder used and Section 2.4.2 explain the general theory behind it. An overview of the architecture is depicted in Figure 4.9.

In this module the residual ε's corresponding signal is called `e`. As explained in Section 2.4.2.1, the quotient q can be calculated by shifting the input k bits to the right. The remainder correspond to the k least significant bits of the input.

The value of q is compared to q_{max}. If q is less than q_{max}, normal golomb codes are used. Firstly, the length of the code is calculated according to Equation 4.1. This differ from Equation 2.5 with an addition to include the 2 index bits when

Table 4.4: *Selection on coder based on control signals from previous pipeline stage*

no_encoding	rl_mode	index	coder
1	x	xx	No encoding
0	1	xx	RLE
0	0	1x	GRC
0	0	0x	SABC

GRC is used. Since the signal `code_out` is of constant width, the initial value of its bits are set to 1. The normal golomb code then simply corresponds to setting the bit with index k to 0 and the bits below k to their corresponding bits in the input vector `e`. However in addition to this the index code, telling the decoder if the pixel reside above range or below range has to be set. The pixel at index $l - 2$ is therefore set according to the incoming `index` signal.

$$\texttt{code\_length} = q + 1 + 2 + k \tag{4.1}$$

If q is greater than or equal to q_{max}, a special code will be output as explained in Section 4.1.2.1. The code length will be set to two times the pixel depth, and the remainder is replaced with the original pixel value x. The index bit representing out of range does not matter in this case.

4.3.4.2 Simplified Adjusted Binary Coder

To determine how many bits n is needed to represent the residual ε in binary representation, Equation 4.2 can be used. While this equation is easy to implement in software, a hardware implementation suitable for FPGA is not straightforward. However, since the residual is a unsigned value, the proposed method is to search for the most significant bit set to 1 instead. This means that `code_length` is calculated as the index of the most significant bit set to high, plus one.

$$n = \lceil log_2(\varepsilon + 1) \rceil \tag{4.2}$$

The signal `code_out` is always 32 bits even if the code from this module can have a maximal length equal to the bit depth. Therefore the unused bits will always be set to 0.

4.3.4.3 Control

The control module will select the correct codeword and code length from the different coders. The signals `index`, `rl_mode` and `no_encoding` are used to select the correct coder according to Table 4.4.

This block will also pass through the original pixel value x as a codeword and code length which is used when the system is set to not encode.

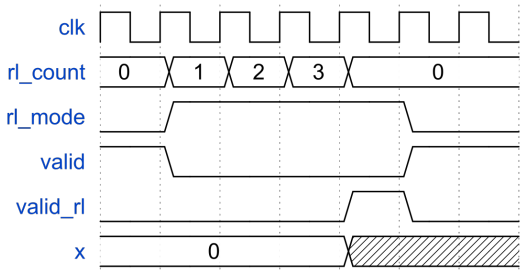


Figure 4.10: *Signal timing diagram for* `valid_rl` *signal*

4.3.4.4 Run Length Coder

The last coder used in F* is the run length coder. The run length counter has already been described in Section 4.3.3 and output the run length as a unsigned value using 12 bits. This means that all this coder has to do is to concatenate this value into a 32 bit codeword and output the code length representing the valid bits of the codeword. As discussed in Section 4.3, the code length when using RLE is always 12 bits.

Two problematic corner cases arise with the adaptive switching between the entropy coders and the run length coder. Firstly, because a new pixel value can be entered into the encoder each cycle, the RLcount has to be pushed to the bit packer before the new code from the pixel value interrupting the run length has to be encoded. This is solved by having the RLcoder always encoding RLcount+1 and then bypassing the third pipeline stage with a **valid_rl** signal so that the run length count is encoded on the clock cycle before the new code. See Figure 4.10.

The second problematic case occur when a run of exactly 3 zeroes are found, which means that run mode will be activated but the actual run length is zero. Considering Figure 4.11, an additional signal `valid_bypass` is used to indicate that a run length of 0 should be appended to the output. This is needed since this could occur while the output from the entropy coders are valid at the same time. Luckily the solution is simple since only runs of zeroes are considered, making the output of the entropy coders completely predictable for this corner case. When x, $n1$ and $n2$ are all 0, the output will be valid from the SABC and the code will be `00`. The output from the coding stage is therefor always 0 with a length of 14 when `valid_bypass` is high and no stalling of the pipeline is needed.

4.3.5 Data Packer

In Section 4.1.3 the data packing problem was discussed. The data packer has to first truncate the incoming codeword according to the code length and then concatenate this code with the already stored codes in the buffer. When the codeword is of sufficient length, i.e. 32 bits, those bits will be output from the encoder and the buffer will be shifted, leaving the leftover bits in the buffer. By implementing

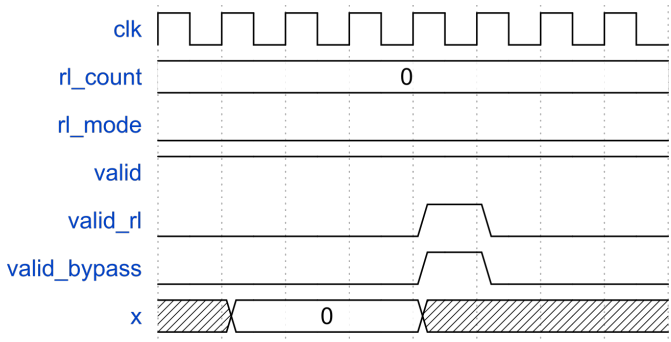


Figure 4.11: *Signal timing diagram for* `valid_bypass` *signal*

the buffer in this way, the output will always be the 32 leftmost bits of the buffer and there is no need for wrap around functionality, see Figure 4.12. There are a significant amount of checks and computations that has to be done. Therefore the block is split into two stages so that the speed requirement of 200MHz is met.

The first stage of the data packer consist of shifting the valid bits of the codeword so that the first valid bit is in the MSB position. This has to be done so that the full 32 bit codeword can be added to the current position of the buffer. Since the maximal codeword length is fixed to 32 bits, the buffer is 64 bits wide and there will never be overflow.

The second stage of the data packer consist of the buffer, a buffer pointer and a output counter. The buffer pointer will always point to the index of the last valid bit. When there are no valid bits in the buffer, the pointer will be set to the buffer size, i.e. the maximal buffer index + 1. Figure 4.12 make it clear that it is the buffer pointer that indicate the valid bits of the buffer and when the packer should output a new 32 bit word. The signal `eol_i` is used for the special cases when the encoder reach end of line as discussed in Section 4.1.3.

The output counter is used to count every 32 bit word that is output to from the encoder. When a line ends, the total word count is output on the 12 bit signal `total_code_length` and is not updated again until a new line has finished.

4.4 Simulated Testing of Hardware

Testing and verification of the hardware encoder is crucial in order to produce equivalent output as the software encoder. Therefore the unit testing framework VUnit[1] was used for testing of each individual hardware module. VUnit introduce functionality for continuous and automated testing of HDL code. It is a complement to the traditional testing approach introducing automatic discovery of testbenches as well as libraries with useful verification tools.

For each module a testbench was written. The approach was to cover corner cases

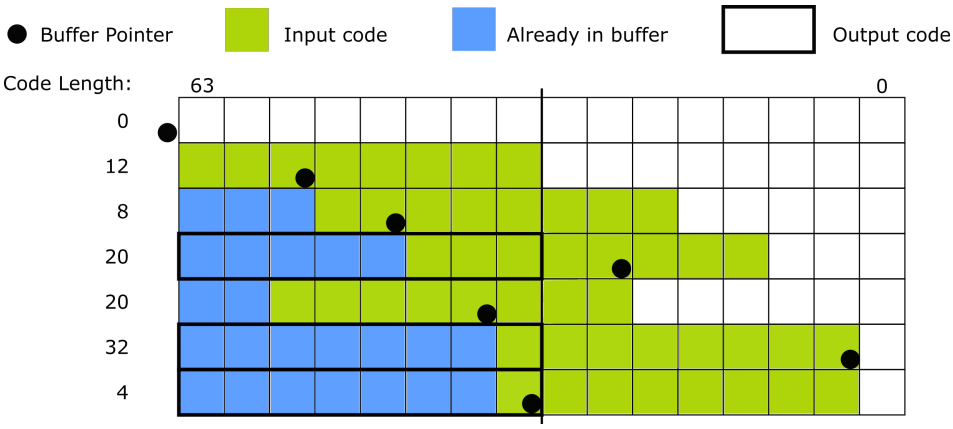


Figure 4.12: *Packing of data into buffer. A square represent 4 bits.*

with hard tests but also inputting random stimuli to the testbench and verify the output against a "high level VHDL" description. The high level description of each module was easily created from the software implementation created in Section 4.2. Using this testing approach the development of the VHDL system can continuously be tested and verified. VUnit use a script to find and run all testbenches and using its `check` functionality, verify if each testbench pass or fail.

VUnit were set to compile and simulate the tests using Modelsim, meaning that a signal view could be seen if need be. This was useful during some stages of development when the VUnit test fail and it was not clear why.

In addition a top level testbench was created which could input test images and produce the output code and code length for the full image. VUnit make it possible to load RAW files into an array which then can be passed through the system and the output is written back to file. The resulting compressed output is then loaded into Matlab and compared with the corresponding output from the software encoder. This way the hardware encoder was verified to produce the exact same output as software. Since the output is the same as in software, there is no need to run hardware to find the compression ratio for a specific image.

4.5 Target Verification

To evaluate the functionality of the synthesized and implemented hardware, Vivado SDK was used to write a test program for the ARM-core of the ZYNQ. The program load binary image data through UART into RAM. This image data is run through the encoder block and written back to RAM at another location using the provided DMA IP. By using images with a known output, the resulting code can be checked to verify that the output of the encoder is as expected. Due to time limitations, a full system datapath using hardware encoder followed by software

decoder has not been implemented. The main additional work that would have to be done is the implementation of the GigE transmission link on the Xilinx ZC706 since UART is to slow for continuous image streaming. With the detailed testing done in simulation, the system is considered to be sufficiently verified without this. The full system implemented on the ZYNQ is depicted in Appendix A.1.

5 Result

5.1 Pre-study

The results of the pre-study was presented in Section 3.4.

5.2 System Modeling

5.2.1 Software implementation

The implemented encoder/decoder was tested on the test images shown in A.2. Firstly the compression ratio of the system is compared against the potential CR calculated in the pre-study. As can be seen in Figure 5.1 the CR of the proposed system is in all cases except one, lower than the potential CR using the static FELICS or LOCO-I predictors. On the image **Camera** the system has a CR greater than the potential when RLE is active. Since a minimal, one directional context is used for prediction, it is expected to perform worse than the potential determined by the entropy. The relationship between the actual CR and the potential CR can be seen as a measure of coding efficiency. From the 4 images used in the pre-study, the proposed algorithm reach an average coding efficiency of 90% compared to *FELICS potential*. This is calculated using the results with RLE turned off.

Secondly the system was tested on the full range of test images from Section A.2 and compared against two evaluation programs from Pleora Technologies and Teledyne Dalsa. The evaluation program *Dalsa TurboDrive Calculator* load a 8-16 bit image and calculate the linerate with or without TurboDrive compression based on a typical 110 MB/sec GigE link bandwidth. The results can be be directly converted to a compression ratio. The evaluation program *eBUS SDK Com-*

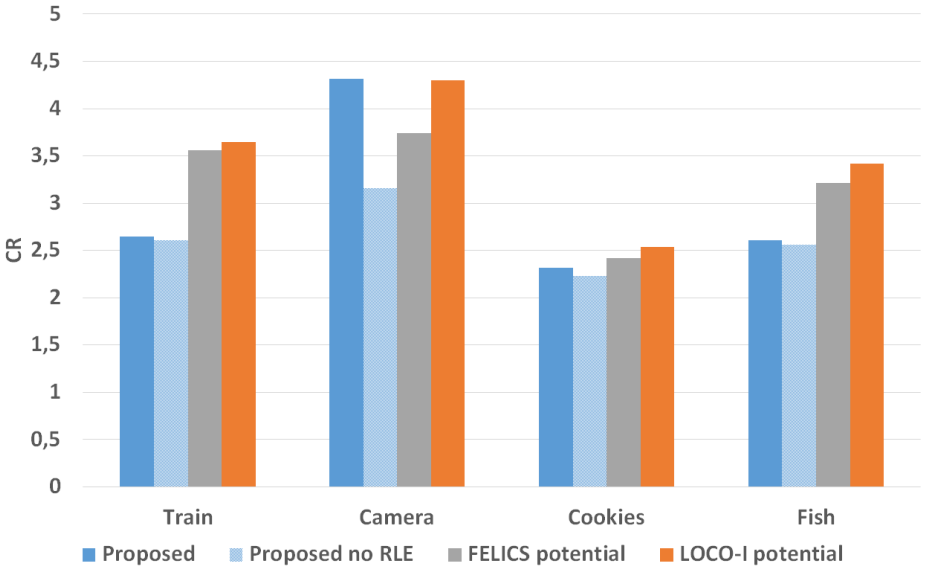


Figure 5.1: *CR of proposed algorithm vs calculated potential*

Table 5.1: *CR and decoding linerate of tested systems on all test images. All tests run using 4 cores on the system from Section A.1*

	CR			**Decoding linerate [kHz]**		
System	Best	Worst	Average	Best	Worst	Average
Proposed	4.31	2.32	2.76	126.42	50.95	74.19
Pleora	2.63	1.31	1.77	127.82	103.51	122.97
TurboDrive	2.51	1.65	1.90	*no data*	*no data*	*no data*

pression, PTC Evaluation Tool from Pleora Technologies load a 8-16 bit image and calculate the compression ratio, encoding time and decoding time using their compression technique. The results of the comparison is presented in Figure 5.2 and the best k in Table 5.2. The average CR of the 3 systems can be seen in Table 5.1. The system proposed by Pleora achieve a average decoding linerate of 122.97 kHz compared to the proposed system of 74.19 kHz. The TurboDrive evaluation program did not report decoding linerate.

In addition to the typical range data images the system was tested with artificial images and reflectance data images. The artificial images **Noise**, and **Zeros** can

Table 5.2: *Best k-parameter for tested images.*

Image	Train	Camera	Cookies	Knacke	Solderpaste	Ventil	Fish	Miscscene	Tincan	Wood
Bit-depth	12	12	12	12	12	12	13	14	13	13
k	2	2	2	2	1	2	1	2	3	2

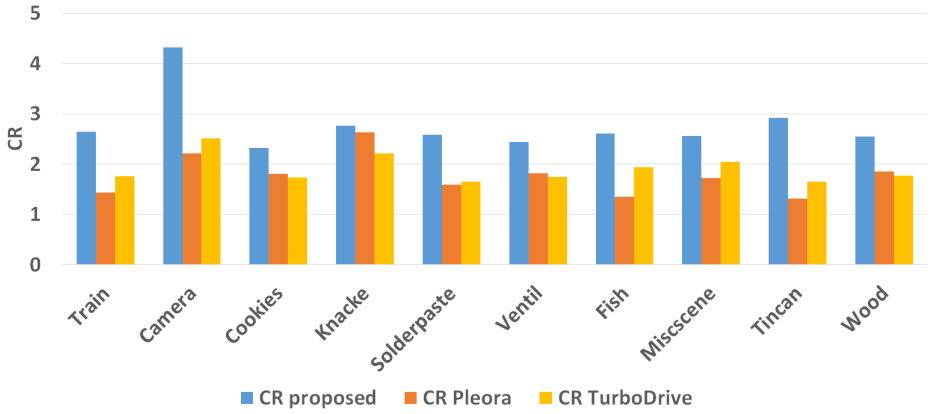


Figure 5.2: *Comparison of CR for Proposed algorithm, Pleora and TurboDrive*

Table 5.3: *Compression ratio of tested images using proposed system, PTC Evaluation Tool and Dalsa Turbo Drive. Bit-depth = 12*

	Compression Ratio			
	Proposed		Pleora	TurboDrive
Run mode	yes	no	no	no
Train	2.65	2.61	1.44	1.76
Camera	4.31	3.16	2.20	2.50
Cookies	2.31	2.23	1.81	1.74
Knacke	2.76	2.65	2.63	2.21
Solderpaste	2.58	2.50	1.59	1.64
Ventil	2.44	2.32	1.82	1.74
Fish	2.61	2.56	1.35	1.93
Miscscene	2.56	2.45	1.73	2.04
Tincan	2.91	2.47	1.31	1.65
Wood	2.55	2.54	1.85	1.77
Average	**2.77**	**2.55**	**1.77**	**1.90**

Table 5.4: Compression ratio on artificial images using proposed system, PTC Evaluation Tool and Dalsa Turbo Drive. Bit-depth = 12

	Compression Ratio			
	Proposed		Pleora	TurboDrive
Run mode	yes	no	no	no
Noise	0.91	0.91	1.00	0.99
Ones	5.96	5.96	7.94	6.45
Zeros	480	5.96	7.94	6.45

Table 5.5: Compression ratio on reflectance data using proposed system. Bit-depth = 8

Image	Camera	Cookies	Solderpaste	Fish	Miscscene	Wood
CR	5.33	2.91	2.53	3.31	2.73	1.92

be seen in Figure 5.4. The figure **Noise** contains random pixel values and as can be seen, the proposed algorithm cannot compress it. Instead the resulting image has increased in size by 9%. The presented result is for the best k-parameter and will be worse with sub-optimal selection. TurboDrive show a slight loss as well and Pleora manage to keep the same size. The image **Ones** display the superior performance of the RLE used in F* to compress missing data.

Table 5.5 show the results of test images containing reflectance data instead of range 3D data. As can be seen for the image **Camera** the high CR of 5.33 is reached since reflectance data can contain missing data as well.

5.2.2 Hardware Implementation

The performance results like max throughput and hardware utilization where determined by the Vivado synthesis and place and route tools configured for the Xilinx ZC706 development board. The final post-implementation utilization results is presented in Table 5.6. This includes the verification hardware used around the encoder IP, such as DMA core, FIFO queues and AXI utilities which all can be seen in figure Figure A.1.

Table 5.6: Final post-implementation utilization of FPGA primitives, with and without verification hardware.

	Utilized		
Primitive	Full design		Encoder
LUT as Logic	6582	(3.00%)	874
LUT as Memory	873	(1.24%)	0
FF	8570	(1.96%)	389
BRAM	4.5	(0.83%)	0
Slices	2723	(5.13%)	284

The system is verified up to a clock frequency of 214.3 MHz which means a max throughput 214.3 MPixel/s. This translates to encoding of full-frame profiles at 83.7 kHz and corresponds to Full-HD (1920x1080) video encoding at 103.8 Hz. With a bit depth of 16, the maximal throughput is 3.43 Gbit/s = 428.6 MB/s. At 214.3 MHz a worst negative slack of +0.201 ns is reported. This occur in the packing module with a total of 19 logic levels (LUTs/Adders) in the data path .

The estimated power consumption was 1.924 W using the default settings for voltage and environment variables such as ambient temperature (25.0 °C). This estimate is for the entire SoC and not for the encoder block. A block-specific estimation for the encoder is not provided by Vivado.

6 Discussion

The following sections discuss and evaluate the presented method and results.

6.1 Method

This section discuss the method used for selection of algorithm, system modeling both in software and hardware, and how the implementation was evaluated.

6.1.1 Pre-study

The purpose of the pre-study was to study related work in order to find and select a base for the project. With image compression being a wide field of research, it was not obvious to what algorithms would be suitable. With the project having a hardware utilization requirement, these requirements could be compared against published work to narrow down the options available. It became clear that the predictive, entropy-based compression algorithms would be the best option with JPEG-LS (LOCO-I) being the most popular and widely implemented system.

After studying typical test images and further evaluating the potential compression of different entropy based coders, the choice was made to implement a system based on concepts from both LOCO-I and FELICS. The choice of implementing a custom design instead of copying a finished design greatly increased the flexibility of the system. This means that the special properties of the 3D Range data images such as missing data could be exploited to improve performance, or achieve similar performance even if simplifications are made. This has already been discussed in Section 3.4.

The choice to leave out the dynamic parameter selection and residual correction from the algorithms of FELICS and LOCO-I was done mainly to minimize mem-

ory requirements and reduce complex data dependencies that arise in hardware implementations. One shortcoming is that there was no in depth study of the performance impact of leaving out these features. It is however clear that even though these features could increase the compression ratio, it would reduce the rate of encoding and more importantly reduce the software decoding rate. This is discussed further in the following sections.

6.1.2 System Modeling

The choice to implement the full system in software (C++) first meant that a precise evaluation of compression ratio could be done relatively fast. With one of the project requirements to implement a decoder in C++ the choice of language was obvious. The software implementation was straightforward to verify, with the decoder verifying the functionality of the encoder.

The use of VUnit for unit tests of hardware design proved to be a major help during development. Even though the time spent writing tests was greater than that writing the actual implementation, this was considered helpful and time saving in the long run. The main advantage of using a unit testing method was that during refactoring of code or later modifications of the system, the functionality was easily verified and affected modules could be changed.

The hardware modeling of the systems was mainly based on those of [19][18][4]. The modifications and simplifications done in the creation of the proposed system was made with the help of the supervisor from SICK and other members of the Ranger3 development team.

6.1.3 Evaluation

The evaluation of the performance rely on the relevance of the images used. Therefore the gathering of test images was made from a wide range of typical applications of the Ranger3 system. The images vary in bit depth, complexity, runs of missing data and noise. Since the output of the software encoder is identical to that of the hardware encoder, the compression ratios for the test images will be identical using the hardware encoder.

6.2 Result

6.2.1 Comparison Against Requirements

The performance numbers achieved by the proposed image compression system are summarized in Table 6.1. All performance measures set in the beginning of the project was met. The compression factor is better than the set goal for all images tested. The run length encoder vastly improve the compression in cases where longer runs of missing data is present. The operational frequency of the encoder had a specific requirement of 200 MHz since the current PL clock in the Ranger3 is set to this frequency. The highest achieved clock frequency was 214 MHz. With a worst negative slack of +0.201 there is a theoretical potential for

Table 6.1: *Comparison of achieved performance for typical images and project specification*

Specification	Goal	Achieved
Compression factor	1.5-2	2.3-4.3
Operational frequency	200 MHz	214 MHz
Throughput of hardware encoder	125 MB/s	428.6 MB/s
Software decompression linerate	7 kHz	74 kHz

a higher system frequency, however next increment in PL frequency that can be generated from the PLL of the ZYNQ SoC is 250 MHz which is to high. With the current implementation a theoretical limit of 225.2 MHz could be reached with an external PL clock.

A limitation of the system is the throughput of the software decoder. Running on all 4 cores on a Intel i7-7700 the decoder reach an average linerate of 74kHz which exceed the specified goal. This is however the average over the tested images with a worst decoding linerate of 50 kHz. However even higher speeds than this would be desireable with the camera link having a maximum bandwidth of 1 Gbit/s. Consider the case where the camera only send 3D range data using 16 bits per pixel. Excluding the auxillary data transmitted from the camera in addition to range data, the maximal linerate would be 24.4 kHz. Now assume a compression ratio of 2.5 in the encoder, resulting in a linerate of 61 kHz, exceeding that of the decoder. Even thought this is a extreme case, the decoder is considered a bottleneck of the system. A typical high-speed application is to run 12-bit range data with a sensor linerate of 46kHz. This result in a data rate of 1.41 Gbit/s which is to high and the frame has to be cropped in width. With a compression ratio of 2.5, only 0.57 Gbit/sec would have to be transmitted and full frame is possible.

6.2.2 Comparison to related work

When comparing against TurboDrive and the compression system from Pleora, F* perform better when in comes to compression ratio. With a average CR of 2.76 compared to 1.77 and 1.9 respectively, F* clearly exploit the compressible features of the 3D range data images better. The run length encoder of F* will as expected greatly improve performance for image with larger regions of missing data. The image **Camera** highlight the efficiency of RLE in images with larger regions of missing data with a 36% increase in CR when active. It is pointed out that even with run mode turned of, the proposed algorithm reach an average compression ratio of 2.55.

As discussed earlier, there is no standardized way to measure throughput in image compression systems in published work. Since the system is capable of bit-depths of 8-16 bits, the fair comparison is measured in pixels per second. Table 6.2 present the throughput of various JPEG-LS and FELICS encoders. Out of the FPGA implementations, the proposed methods achieve the highest opera-

Table 6.2: *Comparison with other implementations of related algorithms*

Design	Algorithm	Technology	Throughput [MPixel/s]	Operational frequency [MHz]	Hardware usage	Memory usage	Compatible bit-depth
2019	F* (proposed)	ZYNQ Z7030	214.3	214.3	284 slices	4 B	8-16
2008 [19]	FELICS	TSMC 0.13μ m	273	273	12.97 k gates	1.9 kB	8
2018 [18]	JPEG-LS	Virtex-6 FPGA	51.68	51.68	8354 slices	262.8 kB	8-12
2013 [8]	JPEG-LS	Cyclone-2	113.03	113.03	30k equivalent gates	12.7 kB + one row of image pixels	8
2011 [4]	JPEG-LS	Stratix-2	155.2	155.2	573 ALUTs	9.49kB	8
2008 [12]	JPEG-LS	TSMC 0.09μ m	265	265	25.7 k equivalent gates	23.8 kB	8

tional frequency. Since the technological advances in FPGA computing has been significant in recent years, the older implementations diminish in relevance when it comes to throughput and operational frequency. This makes it very difficult to judge the quality of the older implementations. In addition, most FPGA papers present a design with the goal to perform in a certain domain. This makes the specific requirements of this thesis hard to compare against other implementations of related encoders. With this said, the proposed design seem area and memory efficient while achieving a high throughput.

6.2.3 Decoder Bottleneck

The easy solution to the bottleneck problem of the software encoder is to run it on a more powerful system using more cores. With lines being decoded separately, the throughput of the decoder scale linearly with the number of cores used. With the development of processors employing higher and higher core counts, even in consumer grade equipment, the choice to tweak the design to allow for multi-threaded workload is considered beneficial.

The other solution is to improve the software. By running a profiler on the program, potential memory inefficiencies could be found. Decoding require bit manipulation and nested loops scanning through the bits of a given sequence. These are operations hard for the compiler to optimize and might result in many run-time checks. A future improvement could be to use system specific inline intrinsics to further optimize the bit operations of the decoder.

6.2.4 Effects of Pipelining

Without the adaptive k parameter selection and residual correction used in FELICS and JPEG-LS, the data dependency problems discussed in Section 2.5.1.6 are removed. The current 5 stage pipeline could therefore be divided further without complications to reduce the worst case propagation delay of subsystems and allow a higher operational frequency. Since the clock in the Ranger3 PL is specified to 200 MHz, this was not explored further. The corner cases in the run length encoder described in Section 4.3.4.4 could be solved in the same way even if addi-

tional stages where added. This would require careful adjustments of the bypass signal `valid_bypass` so that the timing diagram in Figure 4.11 hold.

6.2.5 *k* Parameter Selection

With a static k parameter for the GRC, the question arise as to what is a suitable selection. Since the Ranger3 operate in a fixed environment with similar subjects of measurement, a parameter sweep using gathered images from the related use case will directly give the proper selection.

A more formal analysis of the relationship between k parameter and coding efficiency is discussed in [19]. For 8-bit natural grayscale images, k parameter in the range [1, 2, 3] are recommended with the value of 2 presenting better performance than the others. For the 8 and 12 bit images tested with the proposed encoder, the recommended values gave the best result. With 16 bit images, the variance of the residual values are greater and higher k parameters had better results. For noisy images, a higher k parameter will suppress the large residuals however the small residuals will be coded less efficient since the length of the Golomb codes will always be $l > k + 1$ bits.

6.2.6 Adaptation of RLE

The run length encoder of F* is an addition not found in the original FELICS algorithm. The JPEG-LS standard define an adaptive RLE used when the full context is flat. The hardware implementations of [18][12][4][8] all however exclude this from the system. The reasoning being that the run mode is only effective for artificial images and seldom used when encoding natural images. A comparison of the hardware implementation of the run length encoder has therefore not been made.

A suggested improvement of the implemented RLE is to keep track of how many pixels are left in the line when the RL-mode is started and then encode the RL-number using $\lceil log_2(remainingPixels) \rceil$ bits. This is possible since we encode lines individually and would result in shorter codes for runs that start in the second half of the line. For example if there are 50 pixels left on the current line, the RLnumber can be encoded using only $\lceil log_2(50) \rceil = 6$ bits instead of 12. This is however a rather small improvement for a longer run.

The RLE addition to the algorithm has proven to be very beneficial for certain scenarios. As long as the setup of the camera is calibrated, RLE will allow for significant increase in CR. If the measured object does not use the full frame of the image, the Ranger3 can use a fixed ROI to extract and use a specific area of the image. If the object does not arrive in the same area of the image or if several objects are present at once, a small ROI is not possible to define and unnecessary pixels have to be transmitted. Here RLE can essentially remove the background around the objects without a specific ROI set.

6.2.7 Requirements of Transmission Link

When using an encoding scheme before transmission, the determinism is lost and the decoder won't know how much data it is expected to receive. Because of this, a reliable transmission channel with error correcting functionality is of high importance. In GigE Vision, fixed packet size UDP-packets are used for image transfers. Each data packet has an ID and the receiver can identify lost packets via this ID. Each frame (block) is started with a leader and ends with a trailer packet. The Ranger3 camera keeps an 128 MB frame buffer for retransmission purposes, and this can also be beneficial since it evens out the variable bit rate from the encoding block.

6.2.8 Multi-part Implementation

As briefly discussed, various types of image data can be produced by the Ranger3 in addition to range 3D data. These types are called **parts**. In some applications where multiple parts are used and the bandwidth of the transmission link maximized, multi-part encoding might be of interest. Although the system is designed with 3D range data in mind, compression of reflectance images show good results as shown in Table 5.5. Since a threshold for the valid light intensity can be set directly in the camera sensor, reflectance and scatter data contain missing data as well, making them suitable for the implemented RLE.

With the small footprint of the hardware encoder, additional instances could be added to the FPGA for parallel use in a multi-part system. Again the bottleneck of the system would be the software decoder which would be even more dependent on the core count of the system. If additional parts should be transmitted but not processed in real time on the receiving side, there is no speed drawback with using multiple instances of the encoder in hardware and transmitting the encoded parts for reduced bandwidth usage.

7

Conclusion

This chapter piece together the used method and the obtained result with respect to the problem formulation. A general conclusion of the project is presented and future work discussed.

7.1 General Conclusion

This thesis has investigated, implemented and evaluated a compression system with the goal to reduce bandwidth usage of a transmission link and allow for higher operational frequencies. Considering the guidelines of the project as stated in the problem formulation, all goals have been met. The proposed system is not an optimization for any of the guidelines in Table 1.1 but rather a balance between all of them. The compression ratio exceed that of both the guidelines and competing implementations for similar systems. In applications where larger regions of missing data is present, the RLE module greatly improve compression and encoding/decoding speed. The choice to implement a static k-parameter selection reduced the memory and hardware footprint on the FPGA, resulting in a usage of around 1% of the available resources. In addition it removed throughput-limiting data dependencies from the pipeline. Adding a maximal length for the golomb codes was a important addition to the FELICS algorithm to remove atypically large residuals, especially when using a static k. The recommended k-parameter according to tested images is in the range of 1-3. For noisy images a higher value might be needed. A parameter sweep is recommended for optimal k for a specific setup. The hardware encoder reach a high throughput of 214.3 MPixel/s which translates to 3.43 Gbit/s when 16-bit data is used, exceeding that of the GigE transmission link by 243%. A bottleneck of the current implementation is the software decoder. The single threaded performance is approximately

twice that of the specified 7 kHz for typical 3D range images. With the goal to increase operation speed of the system, the decoder has to be run on multiple cores for linerates above 14 kHz with full frame images (2560 pixels wide). The modification of FELICS to encode lines individually and transmit the code size for each line was a vital addition for this to be possible. For high speed applications a high core-count processor is therefore recommended for the receiving computer.

7.2 Future Work

One direct addition to the system already discussed is the adaptive length of the RLE code. By keeping track of the line index of the current pixel, the system will always know the maximum run length possible and the RLE code can be adjusted accordingly. To implement this a slight modification to the C++ and VHDL code and a reverification of the entire system has to be done.

The context model used in the prediction stage is only dependent on the two closes preceding pixels. By extending the context further, a more accurate approximation of the horizontal gradients could be made, possibly making better predictions for noisy curvatures. This would require modifications to how the selection between GRC and SABC is done.

Even though this thesis chose to remove the dynamic selection of k, an implementation using this is expected to result in a better CR. Additionally it could require one less parameter to adjust during calibration of the system. With a coding efficiency of 90% compared to the potential compression using the proposed predictors, the expected improvement in CR is small. This is however not explored in this thesis. The line independent encoding used in the proposed system is a factor to why dynamic k won't be very effective. Using only the contexts from the same line, the statistical domain is to small for proper adaptation of the parameter for all contexts.

A interesting improvement to the system would be optimization of the C++ code. Profiling tools could be used to find inefficient parts of the software. In addition, system intrinsics could be used to optimize operations such as bit-setting and shifting. Potentially, the SIMD architecture present in most modern systems could be used for vectorization of inner loops, which could improve single threaded performance.

A final and rather obvious area of interest is to implement the hardware encoder in the Ranger3 camera. The main problems to solve for this to be possible is the integration with the GigE Vision protocol used in the transmitter and the integration of the decoder in the software used on the receiving side.

Appendix

A

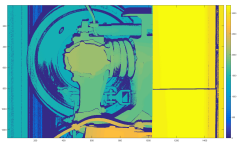
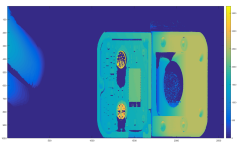
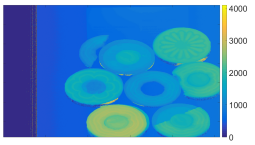
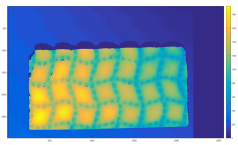
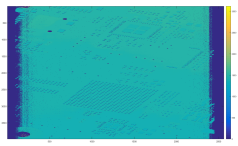
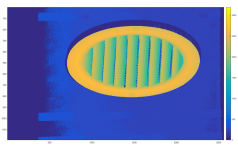
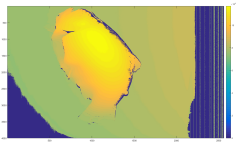
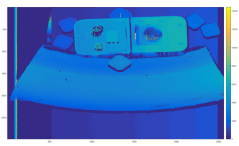
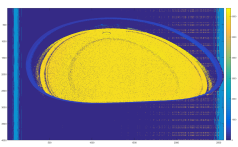
Testing

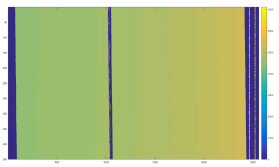
A.1 System Specifications

- CPU : Intel(R) Core(TM) i7-7700 CPU Quad Core @ 3.60GHz
- RAM : 24 GB @ 2667 MHz
- OS : Windows 10

A.2 Test Images

Table A.1: *Test Images, color indicate pixel intensity*

Name	Bit Depth	Width	Height	Description	Image
Train	12	1536	1280	Varying image with low missing data	
Camera	12	2560	1000	Much missing data	
Cookies	12	2560	3400	Sharp edges with flat areas around	
Knacke	12	2560	2991	Only missing data on right edge	
Solderpaste	12	2560	3988	Sharp, but not big edges	
Ventil	12	2560	1196		
Fish	13	2560	4000	Continuous curvature	
Miscscene	14	2560	2990		
Tincan	16	2560	4000		

Name	Bit Depth	Width	Height	Description	Image
Wood	16	2560	500	Flat but varying surface	
Noise	12	2560	1000	Random noise	
Ones	12	2560	1000	Only pixel intensity of 1	
Zeroes	12	2560	1000	Only pixel intensity of 0	

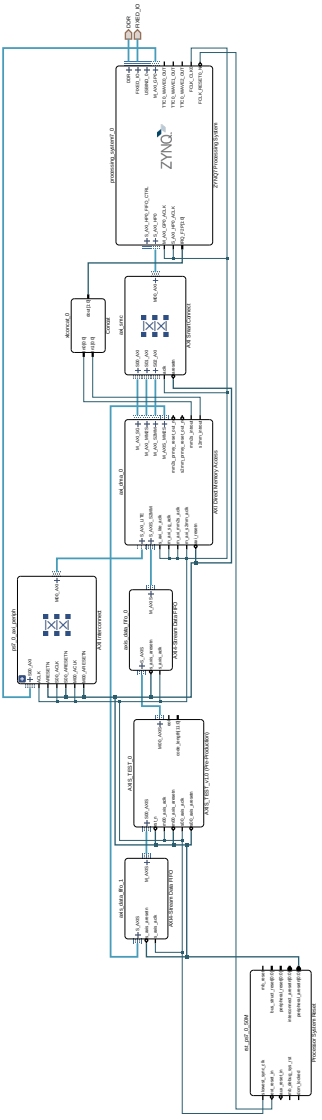


Figure A.1: *High-level schematic of the components of the ZC706*

B

Pre-study

B.1 Additional images used in evaluation

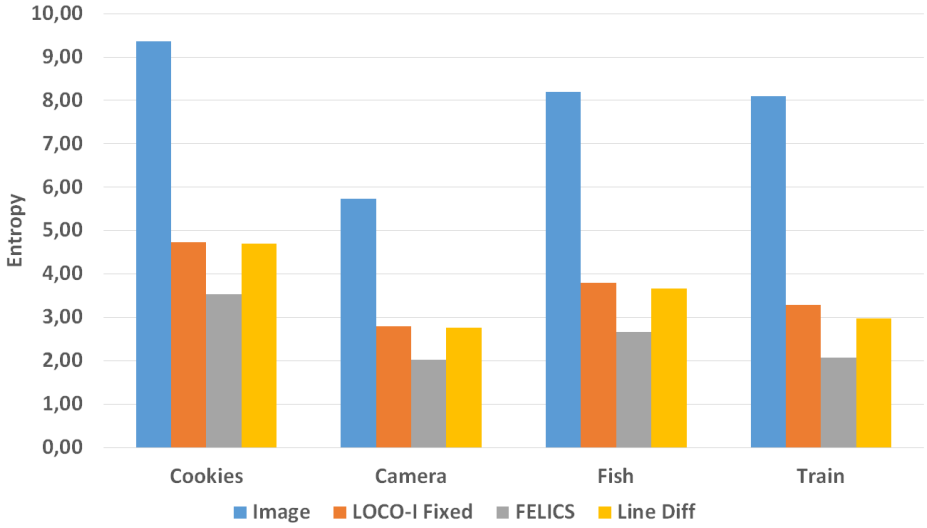


Figure B.1: *Entropy of test images*

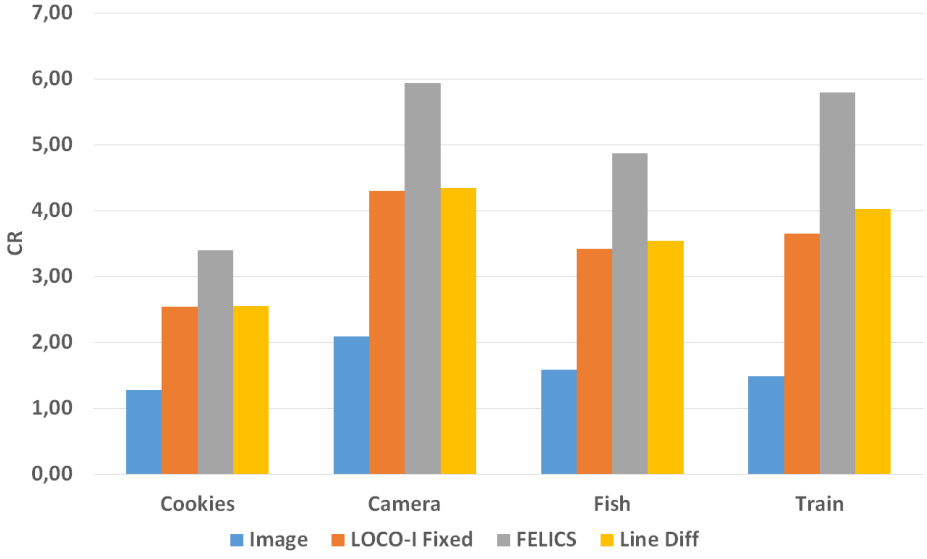
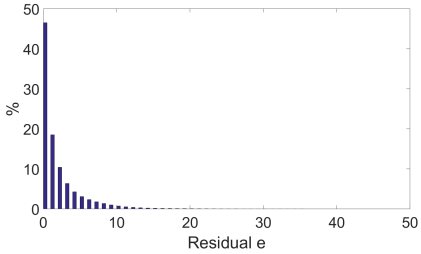
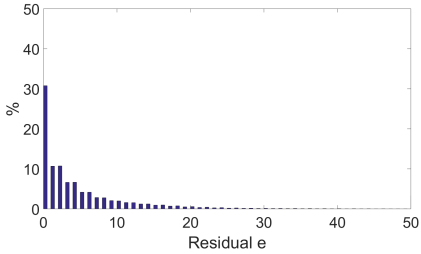


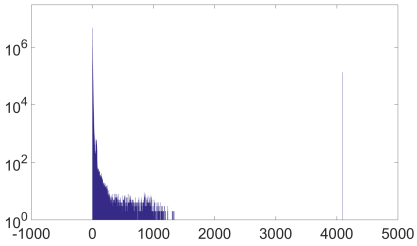
Figure B.2: *Potential compression of test images*



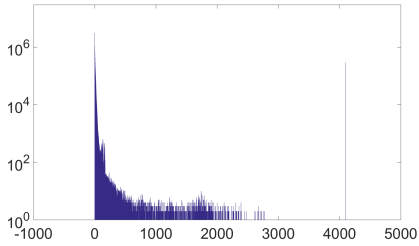
(a) FELICS clipped at x=50



(b) LOCO-I clipped at x=50

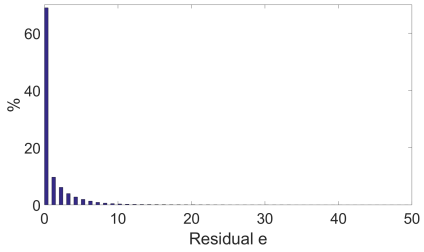


(c) Full historgram in logarithmic scale, FELICS

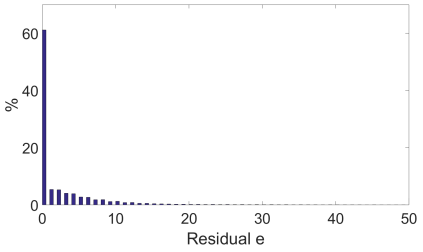


(d) Full historgram in logarithmic scale, LOCO-I

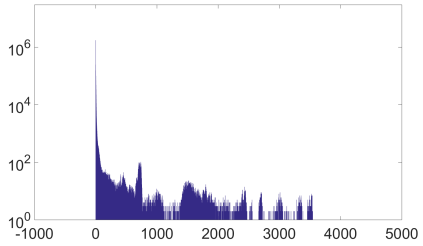
Figure B.3: *Comparison of static decorrelation using predictors of FELICS and LOCO-I on the image* ***Fish***



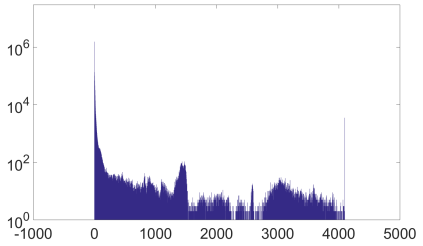
(a) FELICS clipped at x=50



(b) LOCO-I clipped at x=50



(c) Full historgram in logarithmic scale, FELICS



(d) Full historgram in logarithmic scale, LOCO-I

Figure B.4: *Comparison of static decorrelation using predictors of FELICS and LOCO-I on the image* ***Camera***

Bibliography

[1] Lars Asplund. Vunit. URL `https://vunit.github.io/`.

[2] CCSDS 121.0-B-2. Recommendation for space data system standards, lossless data compression. Standard, CCSDS Secretariat, Space Operations Mission Directorate, NASA Headquarters, Washington, DC 20546-0001, USA, May 2012.

[3] L. Chen, L. Yan, H. Sang, and T. Zhang. High-throughput architecture for both lossless and near-lossless compression modes of loco-i algorithm. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 1–1, 2018. ISSN 1051-8215. doi: 10.1109/TCSVT.2018.2881040.

[4] H. Daryanavard, O. Abbasi, and R. Talebi. Fpga implementation of jpeg-ls compression algorithm for real time applications. In *2011 19th Iranian Conference on Electrical Engineering*, pages 1–4, May 2011.

[5] Mohamed A. Abd El ghany ; Aly E. Salama ; Ahmed H. Khalil. Design and implementation of fpga-based systolic array for lz data compression. *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS'07)*, pages 3691–3695, 2007.

[6] P. G. Howard and J. S. Vitter. Fast and efficient lossless image compression. In *[Proceedings] DCC '93: Data Compression Conference*, pages 351–360, March 1993. doi: 10.1109/DCC.1993.253114.

[7] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40(9):1098–1101, 1952.

[8] L. Kau and S. Lin. High performance architecture for the encoder of jpeg-ls on sopc platform. In *SiPS 2013 Proceedings*, pages 141–146, Oct 2013. doi: 10.1109/SiPS.2013.6674495.

[9] Erik G. Larsson. *Signals, Information and Communication*. LiU-Press, Linköping, 2016.

[10] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *IEEE Transac-*

tions on Circuits and Systems for Video Technology, 13(7):620–636, July 2003. ISSN 1051-8215. doi: 10.1109/TCSVT.2003.815173.

[11] D. Salomon; G. Motta. *Handbook of Data Compression*. Springer, London, 5 edition, 2010. ISBN 978-1-84882-902-2. With Contributions by David Bryant.

[12] M. Papadonikolakis, V. Pantazis, and A. P. Kakarountas. Efficient high-performance asic implementation of jpeg-ls encoder. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007. doi: 10.1109/DATE.2007.364584.

[13] Teledyne DALSA Inc. Patrick Sicard. Lossless data compression and decompression apparatus, system, and method, 2016.

[14] B. Rusyn, O. Lutsyk, Y. Lysak, A. Lukenyuk, and L. Pohreliuk. Lossless image compression in the remote sensing applications. In *2016 IEEE First International Conference on Data Stream Mining Processing (DSMP)*, pages 195–198, Aug 2016. doi: 10.1109/DSMP.2016.7583539.

[15] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 4 edition, 2012. ISBN 978-0-12-415796-5.

[16] *Operating Instructions Ranger3 3D Vision*. SICK AG.

[17] Roman Starosolski. Simple fast and adaptive lossless image compression algorithm. *Software: Practice and Experience*, 37(1):65–91, 2007. doi: 10.1002/spe.746. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.746`.

[18] H. Daryanavard ; O. Abbasi ; R. Talebi. Fpga implementation of jpeg-ls compression algorithm for real time applications. *Iranian Conference on Electrical Engineering*, 19:1–4, 2011.

[19] T. Tsai, Y. Lee, and Y. Lee. Design and analysis of high-throughput lossless image compression engine using vlsi-oriented felics algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(1):39–52, Jan 2010. ISSN 1063-8210. doi: 10.1109/TVLSI.2008.2007230.

[20] M. J. Weinberger, G. Seroussi, and G. Sapiro. The loco-i lossless image compression algorithm: principles and standardization into jpeg-ls. *IEEE Transactions on Image Processing*, 9(8):1309–1324, Aug 2000. ISSN 1057-7149. doi: 10.1109/83.855427.

[21] X. Wu and N. Memon. Context-based, adaptive, lossless image coding. *IEEE Transactions on Communications*, 45(4):437–444, April 1997. ISSN 0090-6778. doi: 10.1109/26.585919.

[22] *Zynq-7000 SoC Data Sheet: Overview*. XILINX, 7 2018. v1.11.1.