Analyzing and implementation of compression algorithms in an FPGA

Markus Janis

2011-06-21
Analyzing and implementation of compression algorithms in an FPGA

Examensarbete utfört i Elektronikdesign vid Tekniska högskolan vid Linköpings universitet

Markus Janis

Examinator Qin-Zhong Ye

Norrköping 2011-06-21
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Markus Janis
Abstract

The thesis is performed at ÅF AB in Stockholm. One of the development projects needed a compression algorithm. The work has been done in two major stages. Background theory was compiled and evaluated with respect to the suitability for FPGA implementation. One implementation phase were also done where the algorithms that was suitable was implemented. The system the algorithm was integrated into was composed of a Xilinx Virtex 5 FPGA platform integrated into a system developed at ÅF AB. The development was mainly done in VHDL, other programming languages such as Matlab and C++ was also used. A testbench was constructed to evaluate the performance of the algorithms with respect to the ability to compress data in a test file. This test showed that the Run length encoding was most suited for the task. The result of this test was however not the only source of information for a choice of algorithm. Due to a privacy agreement some variables is not included in the report. The design constructed was designed to act as a foundation for future thesis work within ÅF AB.
# Table of Contents

Abstract ......................................................................................................................... i  
List of figures ................................................................................................................... iv  
List of tables .................................................................................................................... iv  
1. Introduction .................................................................................................................. 1  
  1.1 Background ............................................................................................................... 2  
  1.2 Purpose ..................................................................................................................... 2  
  1.3 Problematisation ....................................................................................................... 3  
  1.4 Specification ............................................................................................................. 4  
  1.5 Boundaries ............................................................................................................... 5  
  1.6 Disposition ............................................................................................................... 6  
2. Theory and Background ............................................................................................... 7  
  2.1 Design environments ............................................................................................... 7  
  2.2 Hardware ................................................................................................................... 7  
  2.3 Software .................................................................................................................... 8  
    2.3.1 Xilinx ISE Design suite 12.2 ................................................................................ 8  
    2.3.2 Modelsim PE ...................................................................................................... 9  
    2.3.3 Notepad ++ ......................................................................................................... 9  
    2.3.4 Tiny Hexer ........................................................................................................ 9  
    2.3.5 Visual Studio 2008 ............................................................................................ 9  
    2.3.6 Matlab ............................................................................................................. 10  
  2.4 FPGA environment .................................................................................................... 10  
  2.5 Compression solutions ............................................................................................. 11  
  2.6 Run Length Encoding ............................................................................................... 12  
    2.6.1 Theory .............................................................................................................. 12  
    2.6.2 FPGA/System compatibility ............................................................................. 14  
    2.6.3 Discussion ...................................................................................................... 16  
  2.7 Huffman coding ........................................................................................................ 17  
    2.7.1 Theory .............................................................................................................. 17  
    2.7.2 FPGA/System compatibility ............................................................................. 20  
    2.7.3 Discussion ...................................................................................................... 22  
  2.8 Lempel Ziv 77 ........................................................................................................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8.1 Theory</td>
<td>26</td>
</tr>
<tr>
<td>2.8.2 FPGA/System compatibility</td>
<td>29</td>
</tr>
<tr>
<td>2.8.3 Discussion</td>
<td>30</td>
</tr>
<tr>
<td>2.9 Deflate</td>
<td>31</td>
</tr>
<tr>
<td>2.9.1 FPGA/System compatibility</td>
<td>31</td>
</tr>
<tr>
<td>3. Result</td>
<td>32</td>
</tr>
<tr>
<td>3.1 Benchmarking</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1 Comparison</td>
<td>34</td>
</tr>
<tr>
<td>3.2 Decompression</td>
<td>34</td>
</tr>
<tr>
<td>4. Conclusion and discussion</td>
<td>36</td>
</tr>
<tr>
<td>4.1 Conclusion</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Discussion</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Future research</td>
<td>38</td>
</tr>
<tr>
<td>References</td>
<td>39</td>
</tr>
<tr>
<td>Appendix A</td>
<td>41</td>
</tr>
<tr>
<td>Appendix B</td>
<td>43</td>
</tr>
<tr>
<td>Appendix C</td>
<td>46</td>
</tr>
<tr>
<td>Appendix D</td>
<td>56</td>
</tr>
</tbody>
</table>
List of figures

Figure 2.1: ISE Buildup (Xilinx 2009) ..................................................... 8
Figure 2.2: FPGA Environment .......................................................... 11
Figure 2.3: Black and white message ................................................... 12
Figure 2.4: ITU-T T4 Standard ............................................................. 12
Figure 2.5: Message 1 ................................................................. 13
Figure 2.6: Message 2 ................................................................. 13
Figure 2.7: Binary message .......................................................... 17
Figure 2.8: Huffman tree ............................................................... 18
Figure 2.9: Compressed Huffman Message ......................................... 19
Figure 2.10: Correct data ............................................................... 21
Figure 2.11: Distorted data ............................................................ 21

List of tables

Table 3.1: Huffman frequency table ................................................... 18
Table 3.2: Difference in Huffman coding ........................................ 22
Table 3.3: Huffman table for testbench ........................................... 24
Table 3.4: LZ77/LZ78 compression ................................................. 27
1. Introduction

Data compression have a long history, it has been around since the beginning of electronics. Data compression can also be referred to as source coding or bit-rate reduction. It is the name for encoding information to minimize or at least reduce the number of bits used to encode information. Compression is usually used to minimize the space used on a hard disk or to minimize the amount of data transmitted through a transmitter that has limited bandwidth. (Grajeda et al. 2006)

Many of the things we use today uses compression in different forms, when watching a DVD or listening to music we use compressed data in different forms.

There are many positive effects from compressing data such as smaller size and less bandwidth usage; there are also some negative effects. The data must be decompressed to be read and this is an operation that takes recourses from the processor or hardware that needs the decompressed data.

Data compression is often divided into lossy compression and lossless compression. Lossy compression is compression that is done with some losses in the original message. It is often used in audio and video applications and other situations where some data can be lost without the message being distorted beyond recognition (Grajeda et al. 2006).

The other category of data compression that will be used in this thesis work is the lossless form of data compression. Lossless data compression keeps the entire original message during the compression. The compressed data can be decompressed at any time and the entire message is kept down to the last bit. Lossless compression is used when no data can be lost, which is the case in this thesis work.

The algorithms are supposed to be implemented in hardware using an FPGA. This implementation means that design should be written in VHDL design. FPGA:s are an integrated circuit that can be configured to a specific purpose.
1.1 Background

ÅF AB is a consultancy company that has the whole world as its market and focuses on energy, environment infrastructure and industry. The company is divided into four divisions; Energy, Industry, Infrastructure and Technology. One of the development projects have big data flow with a slow connection between the tool and a computer connected via a universal serial bus (USB) connection. This is an incentive to apply a compression algorithm to reduce the amount of data sent through the connection.

1.2 Purpose

Field programmable gate arrays (FPGAs) are used in many applications where large amounts of data are handled. There is a need to compress data to reduce the data volumes when data is sent to other applications through interfaces that use lower data rates than the data rate the FPGA is handling. The thesis work attempts to reduce the bandwidth demands of communication to other devices.
1.3 Problematisation

The thesis work contains both a theoretical and a practical part. The theory consists of different compression algorithms to be examined and compared. This provides a foundation for the algorithm to accommodate this kind of compression is best. Also, the possibilities to implement the solution in an FPGA are to be examined and considered in the assessment. A so-called "benchmarking" is done to study which compression method that are best suited for the task.

The practical part is the implementation of such an algorithm. The platform that the algorithm will be implemented on is a high performance, complex FPGA platform with high level of technology complexity. Implementation will be carried out at ÅF's office using tools available on site. If the time frame allows, several algorithms are synthesized in which a selection can be made depending on the type of data sent between the devices. Decoding on a PC should also be designed for full integration into the existing system. Implementation will follow the Design Template used by ÅF Group and other standards that the company uses. The implementation will be adapted for use on the platform that ÅF uses. This creates limitations and adjustments that need to be made from the simulation environment for Very High Speed Integrated Circuit Hardware Description Language (VHDL) design for this design to possibly run on the FPGA system.
1.4 Specification

The algorithm has some constraints that cannot be changed and are needed to be met. The design would have to be written in VHDL for the possibility to implement the algorithm in an FPGA. A decoding algorithm would need to be implemented for decoding of data that are compressed.

- No data can be lost
- Data compressed must have possibility of decompression
- Compression must be reached without using too much recourses from the FPGA in the form of:
  - 150 I/O (input/output)
  - 72Kb Memory
  - 200 Registers
  - 400 LUTs (look up tables)
- The algorithm must send data at the same rate as data is received
- Design must be written in design suited for Virtex 5 compatible FPGAs
- The design must cope with at least 130 MHz implementation
- The design must be designed using state machines
- Benchmarking should be done for comparison of algorithms
- The designed algorithm must be properly documented according to ÅF:s standard

These specifications must be met for the algorithm to be implemented. The algorithm has some specifications that are preferred to be reached but not necessary. They are not necessary for the functionality of the algorithm but they would increase the usability of the algorithm.

- The design can cope with more than 180 Mhz
- The algorithm picks out relevant data for more efficient compression
- The thesis work contains both a theoretical and a practical part.

The theory consists of different compression algorithms to be examined and compared. Also, the possibilities to implement the solution in an FPGA are to be examined and considered in the assessment. A so-called "benchmarking" is done to study which compression method that are best suited for the task.
1.5 Boundaries

The boundaries of this thesis were made after the start of the work with implementing the designs began. The boundaries consist of laying the foundation for a future implementation of the most suitable compression algorithm for the kind of use that is needed in the tool. The thesis will only cover compression algorithms that are lossless due to the nature of the data compressed. The data need to be preserved to provide sufficient information on the receiving end. The design must not use more resources than the system can spare for this kind of operations.
1.6 Disposition

The thesis is divided into four major parts. The introduction part is first and this is followed by an explanation of the theory and background of three compression algorithms and suitability for FPGA integration. The theory and the suitability for FPGA integration are followed by a discussion about how the algorithm can be integrated into an FPGA based system. The discussion also includes a short discussion about how an actual possible integration can be done. In the next part are the results presented. The final part is concluding the major insights gained and are discussing the results from the perspective of these insights. The reference system is the Harvard system.
2. Theory and Background

2.1 Design environments

The development made in this thesis work was done by programming designed in VHDL and C++. The different programming languages demanded their own separate development environments. The developed design were then tested and implemented into the hardware in the existing system (tool) at ÅF.

2.2 Hardware

The hardware consists of:

- A PC (personal computer) with a USB port
- The tool developed by ÅF
- A USB connection

The development of the algorithm is mainly done on the PC with tests done on the tool. The software development will be developed on the PC with different software tools.
2.3 Software

2.3.1 Xilinx ISE Design suite 12.2

The Xilinx development environment is the foundation in the FPGA programming. The design suite includes a variety of development tools for all the different stages of FPGA development. The synthesis was mainly carried out using a script that was developed in-house. The main components of the script were based on Xilinx tool. Built into the Xilinx tool are a variety of previously defined configuration utilities for the Xilinx system. The configuration utilities can control the components of the FPGA and optimize them for best use. The configuration utility is included into the design using Intellectual property (IP) blocks. The ISE design suite can synthesize, implement, verify and program the device under test (DUT). The different functions and how they interact are shown in figure 2.1. The Xilinx ISE suite is designed to play a central role in developing FPGA logic. There are tools from other developers that can be used instead of the ISE suite, the synthesis done in this particular project was however done in ISE environment.

![ISE Buildup Diagram](Xilinx 2009)

Figure 2.1: ISE Buildup (Xilinx 2009)
The design suite has a lot of tools that could replace the functions of other programs used in the project. The Xilinx Synthesis Technology (XST) suite is a tool that has many of the functions of some of the other programs. The reasons for not using Xilinx tools in a bigger part of the project are that the knowledge of these programs was lower than the knowledge of other programs with the same functions.

2.3.2 Modelsim PE

Modelsim is software that simulates the logical setup of the FPGA. In the simulation a plot over the signals included in the design is shown to evaluate the performance of the design. The model is however only a behavior model simulation; the timing of the circuit is not considered in this simulation program. One difference from Xilinx ISE tool is the graphical display of the signals which displays the logical setup in a representative way. The in-house knowledge was greater in this program than the ISE program.

2.3.3 Notepad ++

The design was written in Notepad ++. Notepad ++ is a development program for a large variety of programming languages. The program has a lot of benefits and useful perks. An extension can be installed that makes the program compatible with VHDL design.

2.3.4 Tiny Hexer

The program was used to read the random access memory (RAM) buffer files recorded from the FPGA. The program is compatible with hexadecimal data files.

2.3.5 Visual Studio 2008

The Visual Studio 2008 development tool is a tool for developing applications running on a PC. The environment provides a great variety of different tools for developing software. The development of the software run on the PC was pre-developed without any compression algorithm implemented. The software therefore only needed to be modified to code with the new conditions of data encoding.
2.3.6 Matlab

Matlab is a mathematical software tool with almost endless possibilities. The programming language has the same structure but differs from C design in some aspects. The differences mainly consists of mathematical operations that can be carried out in a more simple and efficient way. (Jönsson 2004)

Matlab was used to construct models of the compression algorithms to evaluate the performance of the compression on a reference file. This helps to evaluate the performance of the different compression algorithms without considering the suitability for FPGA implementation.

2.4 FPGA environment

The algorithm will be implemented into a system that uses a Virtex 5 Xilinx FPGA. The implementation will be integrated into the system on a 32 bit bus. The bus is distributing data to be sent via the USB connection. The bus is however bound to several control signals. These signals are in close connection with the data sent in the bus. Therefore, the data compression needs to have these signals integrated into the block where the compression is done. The block where the compression algorithm will be integrated have some attributes that needs to be considered when designing the block. Since the compression algorithm uses a varying number of clock cycles to work on the data, the data will need to be stored to wait for the processing of the previously sent data to be finished. This implies a memory to be used as a buffer to allow the algorithm time to process the data before sending it. This is done using a (first in first out memory circuit) FIFO for best performance.

A word of FIFO is 33 bits wide to include a control signal that indicates the last word in the message. This will be an indicator to the algorithm to stop the compression and to forward the message to the receiving block together with “end of package” signal. A counter which counts the number of times data is sent is also used for control purposes.

The control signals are generated in a block separated from the actual compression algorithm. The block has however close connections to the compression algorithms main block. An overview of the compression block is shown in figure 2.2. The figure describes the basic layout of the compression algorithm as it will be implemented.
Figure 2.2: FPGA Environment

2.5 Compression solutions

Three different methods of compression were chosen that reach the specifications of the project. The algorithms were chosen based on the fact that the algorithms are lossless and are applicable into the specified system. Some algorithms were not chosen due to obvious problems with implementing the algorithm in a field programmable gate array (FPGA) with the specified data flow (i.e. such as the Move to Front transform (MTF)).

Some of the expressions used in this thesis are general for all algorithms. The term compression ratio refers to the amount of compression that can be accomplished using a certain algorithm. Compression ratio is defined as:

\[
\text{Compression Ratio} = \frac{\text{Compressed Size}}{\text{Uncompressed Size}}
\]

A lot of the discussion in this thesis is concerning the recourses and performance of the FPGA system. The system used can however not be specified in this thesis due to privacy agreement reasons. The system is composed of a Virtex 5 Xilinx FPGA with a fixed number of look up tables (LUTs), registers and memories (Xilinx 2009).
2.6 Run Length Encoding

2.6.1 Theory

Run Length Encoding (RLE) is one of the simplest compression algorithms for binary design that exist (Blelloch E. 2010). It only compresses the most obvious types of space usage. The algorithm in its most common structure uses a special sign that indicates when a repetition of one sign is repeated and the next sign is the number of repetitions that the decoder should write in the data stream. This sign is a sign that could exist in the data itself and it must be indicated in some way if it does. The way it is indicated is by using a control sign which is repeated twice to indicate that this sign is not the control sign but only appears in the data. The decoder then recognizes the repetition and writes only one sign in the unpacked data. The decoder also recognizes that this is not an indicator for how many signs in a row there are since there is two control signs in a row.

One of the most useful aspects with the RLE algorithm is that it only uses one pass to encode and to decode the data (Golomb 1966). This is an efficient algorithm that does not use much resources from the system that encode the data.

One of the most famous usages of the RLE coding is the ITU-T T4 standard developed for fax machines (Belloch E. 2010). The design was developed using a dummy white pixel in the beginning of each row, the original message is shown in figure 2.3

```
bbwwwwwwwwwwwwwwwwbbwbbwwwwwwwwbbwbb
```

Figure 2.3: Black and white message

The original message was then transformed to the following (shown in figure 2.4). The first pixel indicates the white color that comes first when sending the numbers to the receiver. The standard also indicates that only the number of pixels was sent (the run length) and when the pixels change color the next run length will be sent.

```
wbbwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwbb
```

Figure 2.4: ITU-T T4 Standard
The message will be: (1,2,15,4,7,4) and it will be less data transmitted then the original size of the message.

The T4 standard also uses a static Huffman table to compress the data even more (Belloch E. 2010). The Huffman design is divided into multiples of 64 to ensure the consistency of the compression. Huffman coding will be explained further down.

There are also other forms of RLE compression. One way of encoding data is to always send the sign and the number of repetitions. The sequence (shown in figure 2.5)

\[
\text{acccbbcecb}
\]

Figure 2.5: Message 1

will be encoded to (a,1),(c,3),(b,2),(c,3),(b,2). The coded message is then suitable for a probability coder such as a Huffman table (Belloch E. 2010). This because shorter lengths are more frequent in the data than longer lengths.

There are many forms of RLE compression, the fundamental method is the same but the way of implementing it can vary a great deal. One way of implementing the RLE algorithm can be to repeat every sign at least twice and then start the counter and count how many more repetitions there are (Lemire et al 2009). The data in figure 2.6 are compressed using this method would become AA2-BB2-Z-WW-K. This may not reach a good compression ratio, however, the system does not use much resources from the system that compresses the data. The method does not use many resources to unnecessary counters or other resource taking activities. This is one of the major advantages with the RLE encoding and this method emphasizes the advantages of the algorithm.

\[
\text{AAAABBBBZWWK}
\]

Figure 2.6: Message 2

Another method of compressing data using the RLE algorithm is to use a Boolean operator to indicate if data are repeated or not. The data in figure 2.6 would then become A-True-4, B-True-4, Z-False, W-True-2, K-False (Lemire et al 2009). This is a way of minimizing the system usage if Boolean operators have benefits for the system usage. It also minimizes the need for counters. The method described is one of many variations in within the RLE method.
There are also methods that eliminate the need for counter entirely. One of these methods is an algorithm that stores the sign and the position of the sign. This method is preferred if a search function is needed on the compressed data. A binary search is preferred in that case and the performance of this search would not decrease by much (Lemire et al 2009). The string in figure 2.6 would be compressed to 1A-5B-9Z-10W-12K.

To help vectorisation characters can be grouped into instances of characters to enable greater compression or more suitable compression patterns. The method could be adapted to any length of data that is useful for the system. The data in figure 2.6 could be compressed into 2AA-2BB-1ZW-1WK if the data vectors were chosen to vectors of two characters.

One of the downsides of using RLE is that random access is slower due to the nature of compression. The compression ratio is also often not as good than other compression methods due to the problems with variations in the data. Since the RLE algorithm only compresses repeated data, the field of use is limited. Data with large amounts of repetitions within the data is the best candidate for RLE compression.

### 2.6.2 FPGA/System compatibility

A problem with the RLE algorithm is that it does not catch up after buffering values into the memory. If the values from the source arrive into the buffer memory every clock cycle the algorithm has the physical limitation of writing values to the memory at the speed of one value every clock period. Since the algorithm needs two clock cycles to calculate how many signs that are the same in a row and to send out the recognition sign and finally the number of signs the process will build up a bigger buffer. Another drawback with the RLE algorithm using the recognition sign is that if the recognition sign appears in the data the sign must be sent twice to show that it does not indicate how many signs in a row that are the next sent word. These delays are not very large when considering them one at a time. When considering the flows of data the FPGA will be designed to be capable of handling, the amount of processing needed can however not be overlooked. It is very important to use the right amount of memory since memory is a costly resource in an FPGA. One problem appears if the different “anomalies” appear after each other. If this is the case some exceptions must be taken in the algorithm. One way to simulate a memory is to use a counter to keep track of which state the previous state was. This form of remembering variables are however not a good way of keeping values in memory since it is not customary to use space in the FPGA this way (Sjöholm et al. 2003).
To choose RLE algorithm some understanding of the system is needed. Knowledge of the signals and data flows is important when choosing the RLE method. Since the different methods have different advantages, the implementation of the algorithm plays a big part in the choice. A good knowledge of the system is preferred in the choice of algorithm. Since Boolean operators are not used in the same way in FPGA:s and VHDL coding as in other environments it will need some adaption to fit into the design. The VHDL coding does however have support for Boolean operators.

Support for vectorisation could at first glance seem like something that is very suitable for a FPGA since a lot of the programming is done in standard logic vectors (std_logic_vector). The problem with vectorisation is that the vectors often have the same length as the bus they are sent through. This causes a problem when longer vectors are constructed. If the vectors are smaller than the bus size and they have a possibility of being sent through the bus together with the values, this method would be a good solution.

The method of repeating a sign or value twice and then counting the number of following repetitions are a suitable method for FPGA design when the bus length is set. The method is a simple algorithm that does not require a lot of programming to implement. It is therefore a good candidate for this kind of implementation. Another version of this implementation uses a control sign that indicates a repetition of a sign. The control sign indicates the repetition of other signs. If the sign appears in the data, it is helped by adding an extra signal for verification.

The method of always sending the amount of repetitions even if the sign is only repeated once would be a good method for implementing into an FPGA if the number of repetitions is sent in a bus separated from the data. This would be a good solution to compress the data given that an extra bus can be constructed within the structure of the existing system.

One of the common features of all the RLE algorithms is that they all count the number of repetitions in some way. This is one of the more limiting factors for the speed of the algorithm. Though some of the versions of the RLE algorithm can adopt various algorithms to avoid counters being implemented in a large extent some counter must be used in some form. Unfortunately variable length vectors are not an alternative when programming this type of implementations. This will affect the performance of the algorithm negatively (Lemire et al 2009) since the bus will need to have the widest range of the vector as a set value.
2.6.3 Discussion

The implementation of the RLE algorithm is done first since the algorithm in itself is not hard to design. The algorithm has a lot of advantages and can be seen as an introduction to compression algorithms (Lemire et al 2009).

The choice between which type of the RLE algorithm that is most suited for this kind of implementation was done considering the data stream and the possibilities to manipulate it. The data could be manipulated in any way needed as long as it can be restored (lossless compression implied). The compression will be added into an existing design which makes any other signals hard to implement into the design since the signals need to be included into the existing bus. This excludes the algorithms that need an external signal to operate efficiently. The data being compressed has some values that cannot occur in the flow naturally. This advocates the use of a control sign to indicate repetitions of data. Since the data is random in its nature and often not repeated in short length with the exception of some values, the algorithm representing every value with the number of repetitions are excluded since the length of the data would probably become longer with this algorithm. The algorithm that uses an array-like structure with storing the value and position of a sign could be an alternative to compressing the data. The nature of the data is however not optimal for this kind of compression for the same reason as the algorithm that send the length of data on every appearance.

The most suited RLE algorithm for the task in this thesis is the form of algorithm with recognition sign to indicate compression. One of the reasons for this is the nature of the data. the data cannot contain certain signs and this fact makes the implementation of this type preferred. Since the bus length is fixed, other signals other than the data cannot be used. This is also a reason why this method is preferred.
2.7 Huffman coding

2.7.1 Theory

Huffman coding is a compression method that uses a conversion table and the coding is carried out through using this table to “encode” the words into shorter messages. The decoding has the same table and decodes the message to its original content. The table has however two restrictions that must be followed. The first restriction is that no two messages will consist of identical arrangements of coding the message. The other is that the message designs will be constructed in such a way that no additional information is needed to know the starting and the ending points of a sequence (Huffman D. 1952).

The first restriction means that no two messages can be confused in the decoding procedure. The coder will start to decode a message and the binary design cannot be decoded in any other way than one specific message. One example of this is shown in figure 2.7 where the message sent is shown.

```
111 101 111 011 010 101 010 110 111 101 111
```

Figure 2.7: Binary message

The message is sent from the transmitter, encoded and the receiver receives the message and starts the decoding process. The receiver starts with the sequence 111 10 and this could be interpreted as the beginning of the sequence but it could also be interpreted as the middle part of the sequence 111 10-111 10-11. This is a problem with all kinds of transmission of data. The most common way of handling this problem is to have a fixed bit length and thereby padding shorter messages with zeros. The Huffman coding uses a table with non-confusable values (Huffman D. 1952). This is done by creating a tree of descending edges with labeled 1 or 0. This is shown in figure 2.8.
The design shown in figure 2.8 is derived from a frequency table. This table is created by analyzing how many times a sign or a word is used in the data which is meant for compression (Deutsch 1996). This table is shown in table 3.1. The table is showing how many times words of three bits are used in the message in figure 2.8. After the table is derived from the data and both the transmitter and receiver possesses the table, the compression can be conducted. The compression is the most effective when the frequency count is consistent with the actual number of words sent in one particular message. Further investigation of the advantages of dynamic and static tables is done under the respective headings in this thesis.

<table>
<thead>
<tr>
<th>Word</th>
<th>Frequency</th>
<th>Huffman design</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>3</td>
<td>00</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>0110</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0111</td>
</tr>
</tbody>
</table>

Table 3.1: Huffman frequency table

The resulting coded word will then be as shown in figure 2.9. The resulting message is 24 bits long instead of the original 33 bits. This shows the ability of the Huffman algorithm to compress data. The example used is also a good example of the negative aspects of Huffman coding. The words which have the lowest frequencies have a bigger word size than the original word size. This is one aspect of Huffman coding that must be considered when using this kind of compression. If the frequency distribution is close to uniformly
distributed over the content data, Huffman coding might not have a big compressing effect.

1 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 1 1 1 1 0 0 1

Figure 2.9: Compressed Huffman Message

The disadvantages of implementing Huffman designs in an FPGA can be described by J. Vitter (1987) in his paper “Design and Analysis of Dynamic Huffman Designs” where he writes:

“One disadvantage of Huffman’s method is that it makes two passes over the data: one pass to collect frequency counts of the letters in the message, followed by the construction of a Huffman tree and transmission of the tree to the receiver; and a second pass to encode and transmit the letters themselves, based on the static tree structure. This causes delay when used for network communication, and in file compression applications the extra disk accesses can slow down the algorithm.”

Efficient use of resources is one of the main criteria for an efficient implementation of compression method. This may not be the case with Huffman coding if it is used without considering the efficiency perspective. The problem can however if not avoided be diminished by using a set of previously defined tables.

2.7.1.1 Dynamic Huffman table

Dynamic Huffman table design implies that a table is sent with every different data stream (Deutsch 1996). This is a method which is suited for data which varies a lot. If a dynamic Huffman table is made properly the compression ratio can increase (Deutsch 1996). The disadvantage with this method is however the extra information that needs to be sent in the form of transmitting the table from the transceiver to the receiver. This is one reason why the amount of information versus the amount of extra data must be considered. Another reason why the extra amount of data must be considered is how much more effective the transmitted data will be considering the nature of the distortions. The distortions are most probably not equally distributed over each data stream. The distortions often appear in the middle of a data stream and this is devastating to the compression ratio. The problem with distortion of the signal can however not be helped by using a dynamic table that changes with every data stream. The Huffman coding required in the case of sudden distortions and irregularities would be a form of smart dynamic Huffman coding. The Huffman design would need to be able to recognize bad compression ratio in the middle of the data stream.
This is however a difficult algorithm to write and it acquires a lot of resources to monitor the output.

2.7.1.2 Static Huffman table

Static Huffman coding is very much like it sounds. The table is constructed with statistical input from the source data. The data is then analyzed with respect to frequency (Deutsch 1996). The result is the foundation for the Huffman table. This method implies a predefined table that is constructed before the algorithm is used the first time. The positive aspect of this is that the algorithm does not use any resources to construct the table during data transmission. The negative aspect is that the compression ratio is generally lower than that of the dynamic approach (Deutsch 1996). One benefit with the static Huffman design compared with the dynamic Huffman design is the smaller amount of data needed to be sent. Since the table is predefined, all the systems that come in contact with the data already know how it is encoded. This makes any handling of the data much less resource taking both for the receiver and the transmitter.

2.7.2 FPGA/System compatibility

One possible solution could be to create a Huffman table out of the values reached in the variations of the data stream that are sent through the different inputs. This would make for an acceptable compression ratio without compromising the speed that is necessary to avoid losing data due to memory overflow in the FIFO memory.

A possible problem with this would be possible distortions of the signal that makes the signal move outside the area covered by the Huffman table. If the data is sent in the manner shown in figure 2.10 the distorted signal could be something that looks like indicated in figure 2.11. This occurrence can arise for a number of reasons, the interesting part for this thesis would be that the data is shifted on the A axis and the B axis. The values in figure 2.11 is quite different from the values in figure 2.10 which makes any coding based on the values in figure 2.10 less effective than coding which is independent of the values in the data or is adaptive to the values in the data.
This would make the Huffman table useless seen to the compression ratio. This causes a problem that is difficult to handle. Since this distortion could happen at any time, the algorithm would need to recognize when its effectiveness goes down and change the table dynamically.

This is a very difficult algorithm to write and there is a big risk that the compression ratio would be bad or even negative. The signal is constructed by several signals which are out of phase. If the phases shifted slightly the information would change interval and the Huffman coding would be useless.

The simulation of this design would not take this into consideration when simulating the Huffman algorithm. Therefore an aspect that needs to be considered would be that the Huffman algorithm would have a minor or no effect when distortion of the signal occurs. A way of dealing with these distortions would be to use a form of Huffman coding which adapts itself to changes in the input signal. This uses however a lot more of bandwidth due to the fact that it has to send the Huffman table that is constructed to the receiver before it can start the actual transmission. This makes the compression gained from the actual procedure much smaller since the compression has to compensate for this increase in information before it starts being useful. The problem with distortion is something that needs to be considered when implementing Huffman design into an FPGA.

To evaluate which algorithm to put the emphasis on in the project a lot of different aspects are important. The speed of the algorithm will be evaluated with the aspect of throughput.

The suitability of the Huffman algorithm depends on many factors. One of the more important aspects for the suitability is the usage of recourses compared to how much the algorithm actually compresses the data. In the case of dynamic Huffman table, the collection of data and the formation of
the initial table for the compression could acquire a lot of resources that does not match the performance of the algorithm. In the FPGA environment efficiency is the most important feature of an algorithm.

### 2.7.3 Discussion

When using Huffman coding it is important to decide what kind of Huffman table is the more optimal for the implementation in mind. The different approaches to building a Huffman table have all their different advantages and draw backs. (Deutsch 1996)

<table>
<thead>
<tr>
<th>Huffman table with predefined coding table</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Uses less space in the FPGA</td>
</tr>
<tr>
<td>+If predefined data are accurate, good compression could be achieved</td>
</tr>
<tr>
<td>-Less compression than dynamic Huffman coding</td>
</tr>
<tr>
<td>-If the signal changes in position for any reason the compression would disappear</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Huffman table with table defined at first run</th>
</tr>
</thead>
<tbody>
<tr>
<td>+The compression will be better for that particular stream of data</td>
</tr>
<tr>
<td>-If the signal change in the middle of the transmission the compression disappears</td>
</tr>
<tr>
<td>-Must have “sample data” to create table</td>
</tr>
<tr>
<td>-Less efficiency than continuous verification</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Huffman table with continuous verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Best compression of the different Huffman methods</td>
</tr>
<tr>
<td>-Not resource efficient</td>
</tr>
<tr>
<td>-Low data throughput</td>
</tr>
</tbody>
</table>

Table 3.2: Difference in Huffman coding

When considering the different forms of Huffman coding shown in table 3.2 for FPGA implementation one implementation stands out as the least recourse taking algorithm. The Huffman coding with the predefined coding table is the alternative that has the most potential for an FPGA implementation in an early stage. The implementation could then be extended in further work to include a dynamic Huffman table. This will be further investigated under Future Research.
The predefined table has the advantage of having the table already defined. This leaves the hardware to be used for compression and not to calculate the conditions of the compression. The method is therefore suited for a simple implementation within the timeframe of this thesis work. The most optimal form of Huffman coding would be a dynamic table with different compression for different data sets. This method would however require a lot of computation and a great deal of memory to execute. Another problem with any Huffman coding is the fact that any new data that passes through the bus must be indexed in the Huffman table. This is not in itself a difficult operation and it can be done with a relatively small logic. The limiting factor is however the space in the bus it takes to send the design of the new entry and the question of how much time and space this takes from the normal data sending. A problem with the data in the bus is that it is very hard to predict which values that passes through the bus. This makes the implementation of a Huffman table more difficult. Since the tool is dependent on a continuous data flow to run properly is this method not an optimal choice for this implementation.

One way Huffman coding could be implemented on this system is that the hexadecimal alphabet would be encoded as a Huffman tree; this means that any message can be encoded in the data stream. This form of coding does however not cause a major increase of compression ratio. The static binary coding of hexadecimal values is represented by four binary digits (IEEE std). The Huffman table constructed for this particular data stream would have a search string length from one to six binary digits. The Huffman table can be constructed to use fewer digits if a smaller Huffman table is desired; the particular frequency table (shown in table 3.3) illustrates the large amount of zeros in the data. This means that a greater compression ratio will be reached using a shorter design for the zeros in the design.
The data shown in table 3.3 indicates a use of the sign zero that is used almost double the amount than the amount of other signs used combined. This gives an indication of the need for extreme compression needed for the zeros while as the other signs are not used as frequent and therefore not necessary to compress to gain as high compression ratio as possible without over working the algorithm. The amount of work done to achieve greater compression ratio must stand in relation to the actual ratio gained from such work.

The algorithm can of course be modified to include longer strings of data within the Huffman table. The search string needed to find one particular value will however grow with the addition of additional values.

A possible implementation of the Huffman algorithm would be to have one node which indicates an addition to the existing Huffman tree. A form of semi-dynamic Huffman tree with additions and modifications done to further increase the compression ratio. The Huffman tree would be built up on the same basic principle as a normal Huffman table with the exception of one node end being the value “adds another value”. This branch, if called would move one of the existing values to a branch further down, this would in turn give place for a new string to be added on the other side of this branch. The new added value could be a string that occurs frequently in the data. The algorithm could count frequently used strings of data without blocking the bus with unnecessary Huffman data being sent as a dynamic Huffman table.
would imply. The benefits and disadvantages of this kind of algorithm would however need to be further investigated to ensure a beneficial integration into the system. The system would however have all the basic signs already encoded which means any message sent over the bus will be encoded.

One option is to use an adaptive form of Huffman coding with the help of probability coding such as the one described by Yuriy A. Reznik in his paper “Practical Binary Adaptive Block Coder” (2007).
2.8 Lempel Ziv 77

2.8.1 Theory

The Lempel-Ziv algorithm was first presented in 1977 (Ziv et al. 1977). The year of presentation together with the initials makes for the name of the algorithm. The algorithm is called the LZ77 algorithm and a new version was released the year after, it is called LZ78. The algorithms are also known as LZ1 and LZ2. The Lempel-Ziv algorithm is the basis for many different forms of variations of the algorithm.

The difference between the LZ77 and the LZ78 is that the LZ77 works on data within a window, the LZ78 algorithm however works on all past data as well. The algorithm explained in this thesis is the original LZ77 algorithm.

The LZ77 algorithm has laid the basis for many compressions that is used widely today such as the graphics formats GIF, TIFF and JPEG (Grajeda et al. 2006).

The LZ77 algorithm uses a window of data that is called the search buffer. The algorithm also uses another window that is called the look-ahead buffer. These buffers inspire the name sliding window compression. The basic function of the algorithm is to replace signs that have been used previously in the data (Grajeda et al. 2006).

The window divided into two parts but the parts are dependent on each other. The algorithm searches in the search buffer for the longest possible match from the look-ahead buffer. The search buffer can be compared with a dictionary for the look-ahead buffer. The dictionary is however dynamically changing as the data flows through the algorithm.
The search buffer or the dictionary in table 3.4 includes the symbols that have already been encoded. The look-ahead buffer in table 3.4 is containing the data yet to be encoded. The data in the look-ahead buffer will be matched with the longest possible phrase in the search buffer. Once a matching phrase is found a codeword is sent containing the distance to the beginning of the match and the length of the match. This information is completed with the next sign in the data (Grajeda et al. 2006)(Ziv et al. 1977). Each of the matching messages can be visualized with $M = (M_o, M_l, M_n)$. Here, the $M_o$ points to the starting position of the matching phrase within the search buffer. This value is calculated seen from the current position and counting backwards into the search buffer (Grajeda et al. 2006). The next value in the codeword is $M_l$ which is the length of the match. The length is counted from the first matching value to the last value of the matching string. $M_n$ is the value of the next sign in the look-ahead buffer that does not match the string in the search buffer. The first two values ($M_o$ and $M_l$) are 0 if the value in the look-ahead buffer does not match any of the values in the search buffer. The procedure of matching an as long phrase as possible will then start again with the search buffer moved to the new position with the encoded values in the buffer. The look-ahead buffer also moves to the next value in the data ready for compression. This procedure is then repeated till the end of data stream. The decoding of this algorithm is done in the reversed order from the encoding. This enables the decoding search buffer to know where the compressed phrases are referring.

<table>
<thead>
<tr>
<th>Search buffer</th>
<th>Look-ahead buffer</th>
<th>Output $(M_o, M_l, M_n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>_she_sells_sea_shells</td>
<td>(0,0,_)</td>
<td></td>
</tr>
<tr>
<td>_ she_sells_sea_shells</td>
<td>(0,0,s)</td>
<td></td>
</tr>
<tr>
<td>_s he_sells_sea_shells</td>
<td>(0,0,h)</td>
<td></td>
</tr>
<tr>
<td>_sh e_sells_sea_shells</td>
<td>(0,0,e)</td>
<td></td>
</tr>
<tr>
<td>_she _sells_sea_shells</td>
<td>(4,2,e)</td>
<td></td>
</tr>
<tr>
<td>_she_s_e ll_s_sea_shells</td>
<td>(0,0,l)</td>
<td></td>
</tr>
<tr>
<td>_she_s_e l ls_sea_shells</td>
<td>(1,1,s)</td>
<td></td>
</tr>
<tr>
<td>_she_s_e lle ls_sea_shells</td>
<td>(6,3,a)</td>
<td></td>
</tr>
<tr>
<td>_she_s_e lle_s_sea_shells</td>
<td>(14,4,l)</td>
<td></td>
</tr>
<tr>
<td>_she_s_e lle_s_sea_shells</td>
<td>(1,1,s)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: LZ77/LZ78 compression
The Lempel-Ziv algorithm can be applied to any discrete source without any *a priori* knowledge of the source data (Ziv et al. 1977).

The negative aspects of the Lempel-Ziv algorithm (LZ77/LZ78) are that it is very susceptible to channel errors during the transmission of the encoded data. The data will easily become corrupted and there is not a great possibility to check and correct the error occurred. One other negative aspect of the algorithm is that it is the most effective when big portions of the data stream are repeated. The search buffer must be adopted to fit the size of the repetitions to get an effective compression ratio.

### 2.8.1.1 Variations

There are many variants of the LZ77/78 algorithm. There are far too many to evaluate them all within the timeframe of this thesis work. Some of the algorithms are however worth a mention.

One of the developments of the Lempel-Ziv algorithm was done in 1985 by V. Miller and M. Wegman. The modified algorithm searches for the longest string already in the dictionary and adds a compilation of the previous entry into the dictionary with the current. This makes for an efficient compression. However, the dictionary fills up very fast and to compensate for this problem V. Miller and M. Wegman suggests a deletion of the low frequency entries.

In 1988 J. Storer modified the LZMW algorithm. This was an improvement that eliminated some of the complexity of the LZMW algorithm. The modified algorithm is called LZAP where the AP represents “all prefixes”. The difference from the “original” LZMW algorithm consists of the addition of all the prefixes into the dictionary. The next match (or entry into the dictionary) will be added together with the previous entry but with the addition of all the partial matches’ as well.

The LZRW algorithm uses hash tables to further increase the compression speed. This compression method was developed by Ross Williams who has published a series of compression algorithms based on the LZ77 algorithm. (R.N. Williams 1991)

There are of course many other versions of the Lempel-Ziv algorithm. The reported algorithms are a selection to account for the knowledge of other algorithms within the field.
2.8.2 FPGA/System compatibility

Integrating the algorithm into an FPGA is not very different from integrating any other compression algorithm. The main problem is the resource allocation and the size of the search buffer. Since a bigger search buffer means a possibility for a greater compression ratio. The bigger buffer is however not a certainty for a greater compression. The LZ77 algorithm implies that two counters will be used to keep track of the compression phrase. These counters can easily be set on other busses to enhance the effectiveness of the algorithm. Compression could be indicated with a single bit indicating if the value sent on the bus is found in the search buffer or not.

One of the benefits of writing algorithms in VHDL for FPGAs is that all the data not necessarily need to go over the same bus. The data could easily be divided into several busses to save space. This was however not an option in the implementation for this particular thesis work. Since the only bus being used in the predefined block is one bus of 32 bits width there would mean much work to rewrite the predefined blocks to fit the compression algorithm only one option. The option for this kind of setup would be to reduce the actual data flow to 16 bits and to use the other 16 bits as counters. Using this method the data flow would be reduced to 50 % of its capacity. This decrease in data flow would have to be compensated with buffer memories to store the information hindered by the reduced data flow. The buffer would probably be large and not stand in perspective to the compression ratio.

The only other option left is to utilize the full bus length for both the data and the counters. This option is however not usable in the particular system researched in this thesis. One of the major flaws with this design would be that data only could be sent every three clock cycles but the incoming data arrives every clock cycle. This would mean that the data would need a very large buffer to compensate for the slower data flow.
2.8.3 Discussion

The compression ratio would probably also be bad if the Lempel-Ziv algorithm would be implemented since the nature of the data is less repetitive than needed for the algorithm to be useful. The Lempel-Ziv algorithm is more useful on text than on (almost) random data.

The LZ77 algorithm can only send data every three clock cycles in the FPGA due to the limitation of bus width and the fact that only one bus can be used. This is an argument not to use the LZ77 algorithm.

If the compression algorithm could be designed in the way specified first in the paragraph above the compression ratio could reach acceptable levels. The compression would most probably not be good considering the nature of the data but acceptable. This compression method combined with the Huffman algorithm would reach a better compression ratio. This is evaluated in the deflate paragraph. The Lempel-Ziv algorithm alone would probably not be suited for this particular implementation.

There are of course many other forms of the Lempel-Ziv algorithm available to be used for compression. The many forms of algorithms within this field are so vast that it could be a foundation to a whole other thesis work. It is therefore hard to know with certainty that the original Lempel-Ziv algorithm chosen was the best suited for this particular purpose. One of the limitations of the thesis work was however a time limit and since no better algorithm was found within the timeframe the original algorithm seemed like the natural choice to research for possible implementation.
2.9 Deflate

The deflate algorithm is using two different types of compression, Huffman coding and LZ77 compression. Huffman coding uses look up tables for more efficient compression whereas the LZ77 algorithm uses a form of “sliding window”. The algorithms are both useful for compression as they are; Deflate however combines the algorithms to get optimal use out of the algorithms (Deutsch 1996).

2.9.1 FPGA/System compatibility

The deflate method could be very effective but due to the two passes over the data needed, the algorithm may not be suited for the purpose of compressing data to send in high speed within an FPGA. The FPGA are designed to be fast and is therefore needed to compress data to be sent in a way that can be handled very fast.

The Lempel-Ziv algorithm is as concluded earlier not suitable for this particular implementation and this makes the deflate compression less suited for the implementation than for example the Huffman algorithm would be by itself. The Lempel-Ziv algorithm would only slow down the output and cause the FPGA to use a lot of memory to store the data while waiting to be compressed.
3. Result

3.1 Benchmarking

Models using Matlab was constructed to evaluate the performance of the algorithms. The algorithms were tested using a test file consisting of data that was representative for the data that was going to be sent in the bus where the compression algorithm was going to be placed. The benchmarking of the algorithms is a way of showing the benefits and the disadvantages of the different compression algorithms. A testbench was constructed with a sample file as input (Shown in Appendix A). This file represents one possible variation of input data that will pass through the bus in the system.

The file is not a sample of the actual input, the file is however a representative for the data that was available for construction of the algorithm. The data available for construction of the compression algorithm was not representative for the data that will pass through the bus once implemented. The data is however the only data that was available for development. This fact makes the decisions related to the data difficult to base on the information gained from the benchmarking and other conclusions based on the data within the limits of this thesis. All of the decisions regarding compression possibilities for the different algorithms can be based on the results of the benchmarking, consideration must however be taken to the fact that other circumstances will reign when the algorithms are actually implemented. Since benchmarking is the best way of gaining knowledge of a particular algorithms performance on one particular type of data the results of the tests can however be considered valid.

The benchmarking does not take the amount of resources used into consideration. The benchmarking is only evaluating the algorithms ability to compress this kind of data, not the performance in the amount of resources taken by the algorithm. The data was edited to remove any space signs or new row signs to be certain that no distractions from the actual data were present.

The RLE algorithm was able to compress the data from 1920 signs to 119 signs. This is a reduction of 93.8 % which is a significant decrease of data. This was mostly due to the fact that zeros are distributed in large quantities in some portions of the file. This is the main benefit of the RLE algorithm, when trying to get rid of large amounts of repetitions. This is however not necessarily present in the data the algorithm will be set to operate on. Based on the sample file, the RLE algorithm is however very effective.
The Huffman algorithm was not as successful as the RLE algorithm in encoding this particular data file. The data was decreased by 84.49 % which is a good compression ratio. The file size is however ten times as big as the RLE algorithm. When working with compression this is a big difference.

The Huffman algorithm is more suited for written texts where i.e. a and e is more used than x and y which is not the case in the reference file. The Huffman table has a greater possibility of compressing a file without the large amounts of zeros that are present in the reference file.

The RLE algorithm with a recognition sign was used for the test. A Huffman tree was done only once. Since a dynamic table would not increase the performance in a positive way.

The LZ77 algorithm was considered for implementation, the algorithm was however dismissed due to the obvious lack of ability of compressing data with the structure that will occur in the implementation of the algorithm. This was decided with regards to the nature of the data in its original form and can therefore not be reported here. The benchmarking of this algorithm would therefore not have been relevant for the choice of algorithm and was therefore not included in the benchmarking.
3.1.1 Comparison

Since the algorithms compress the data in different ways, different methods are more effective in different situations. The RLE algorithm does not modify the binary length of the signs within the file, the algorithm reduces the amounts of repetitions instead. This is useful in implementations with fix length alphabets where the bit length is difficult to change. Here is where the main benefit of implementing Huffman coding is shown. The Huffman table shortens the length of the characters that are most frequently used and makes the length of the least frequently used signs longer. The need for the Huffman coding to be useful is when the least used signs are used a very low number of times where the most used signs are used with a high frequency. This is when the algorithm is the most beneficial for the compression ratio. The reference file has a high frequency of one sign and drastically lower frequency of the rest of the signs. This is conditions that benefit the RLE algorithm more than the Huffman algorithm. The result is however not consistent for all types of data that might pass through the bus where the algorithm will be implemented and this fact must be considered in the decision of algorithm for implementation. Also, the amount of resources used must be considered.
3.2 Decompression

Integrating the solutions into the software was very different from coding the algorithms in VHDL. The focus here was to save space without considering the amount of resources used which meant allocating as little memory space as possible. A big difference between designing hardware and software is that software focuses on the data that is sent in the busses made in the hardware programming. This is a different approach to programming and this could be a problem for a programmer, adapting to another way of thinking. Sometime was taken in the beginning to learn the environment in Visual Studio and to understand the necessary parameters for a successful integration into the existing system. The code written in C++ is attached in Appendix B.
4. Conclusion and discussion

4.1 Conclusion

The first algorithm implemented was the RLE algorithm; an implementation of a simple form of Huffman coding was also done. The performance of the Huffman algorithm was however hindered due to the time limit of the thesis work.

The algorithms were chosen with respect taken to a continuous data flow. When considering a division into packets, another algorithm may be applicable on the bus the algorithm is placed to compress. A higher rate of compression can also be achieved when packet sizes can be set as the data have a finite amount of values.

The conditions for the implementation when a compression algorithm was considered were different than the condition of the final placement of the algorithm. This difference would cause a bigger problem than it would seem. When considering a flow of data some compression methods are less suitable due to the natural constraints of the data stream and the nature of the algorithm. Considering the nature of the data flow the algorithm chosen was a correct choice.

The data available for construction of the algorithm was however not consistent with the data flows the algorithm will be set to compress. This is something that future research will need to consider. The algorithms chosen was however a good foundation to develop more suited algorithms in the future.
4.2 Discussion

The implementation was hindered due to obstacles that appeared in the implementation phase of the algorithm. More delay was added due to the fact that the tool was under development for other applications during the time of the thesis work. The other projects had higher priority and therefore delayed the thesis work further than it would have been without these influences. The implementation was first added into the system on a raw data flow bus. This demanded a type of compression that could compress data without the need of a start and beginning of the data. The implementation was later moved into another block deeper in the system which meant that different control signals were needed. This complicated the work to some extent and delayed the project further. Along the working process of the implementation of the algorithm some understandings of the different algorithms were made. These understandings were however in line with the early assumptions that were made. The thesis work was to serve as a foundation for future thesis works within the same field. This was enabled in the form of a frame for implementation of compression algorithms. The frame handles the different control signals that are needed for implementation.

The conclusions in this thesis are based on conclusions drawn from material reported within the frames of this thesis. Some of the decisions are however partially or entirely influenced by conclusions and facts drawn from material not reported in this thesis. Due to the privacy agreement surrounding the thesis work some facts and knowledge gained from the work are hard to illustrate without revealing too much information. Unfortunately this circumstance makes the result presented in this thesis hard to interpret. Some of the decisions may seem founded in false assumptions, the problem with this is the lack of data to prove the composition of the data that will flow through the bus when the tool is implemented and used in the field. This thesis has illustrated most of the difficulties with the data. All the difficulties and conditions was however not possible to show or to explain.
4.3 Future research

Since a lot of energy has gone into adopting the compression block into the existing project, some of the focus was lost from finding the best compression for the data passing through the bus. Future research within this field could be to further investigate the possibility for compressing the data with consideration of the data passing through the bus. Also, an adaptive compression algorithm that adapts the compression method depending on which data that is used could be a possible extension of this thesis. For example, a Huffman algorithm that generates different tables depending on which kind of data type is chosen would be a very efficient way of compressing data in a reliable and effective way. Since work has been done integrating the design into the existing layout some work can be reused so that focus can be set on compressing data.
References


Belloch Guy E., “Introduction to Data Compression”, September 2010


Golomb S “Run-Length encodings”, July 1966

Jönsson Per “MATLAB beräkningar inom teknik och naturvetenskap”, 2004


Sjöholm Stefan and Lindh Lennart “VHDL för konstruktion”, 1994

Storer, J A, “Data Compression Methods and Theory”. COMPUTER SCIENCE PRESS, INC., 1803 RESEARCH BOULEVARD, ROCKVILLE, MD 20850 (USA) , 1988, 413 pp

Virgilio Zuñiga Grajeda, Claudia Feregrino Uribe and Rene Cumplido Parra, “Parallel Hardware/Software Architecture for the BWT and LZ77 Lossless Data Compression Algorithms”, October 2006

Vitter Scott Jeffery “Design and Analysis of Dynamic Huffman Codes”, April 1987


Appendix A
#pragma once
#include "stdafx.h"
#include "DeCompression.h"

void * DeCompression::RLE( void * RLEsource, WORD sizeOfRLEdata, WORD * pUnpackedSize)
{
    void * destination;
    DWORD unpackedSize = 0;
    WORD numberOfdata;
    DWORD * tempArray;
    bool success = false;
    int nummer, m = 0, i = 4;
    *pUnpackedSize = sizeOfRLEdata;
    tempArray = (DWORD*)RLEsource;
    //TODO: Calculate unpackedSize
    for(int i = 0; i < (sizeOfRLEdata / i); i++)
    {
        DWORD temp, nextTemp;
        temp = tempArray[i];
        if (temp == 0xffffffff)
        {
            nextTemp = tempArray[i + 1];
            if (nextTemp == 0xffffffff)
            {
                *pUnpackedSize = *pUnpackedSize - 4;
                i++;
            }
            else
            {
                numberOfdata = (WORD)nextTemp;
                nummer = (int)numberOfdata;
                if (nummer <= 2)
                {
                    /*pUnpackedSize = pUnpackedSize - 8;
                }
                else
                {
                    *pUnpackedSize = *pUnpackedSize + (numberOfdata - 2) * 4;
                }
            }
        }
        else
        {
            *pUnpackedSize = *pUnpackedSize + (numberOfdata - 2) * 4;
        }
    }
i++;  
}  
}  
destination = (DWORD*)LocalAlloc(GMEM_FIXED,(*pUnpackedSize));  
//TODO: unpack  
m=0;  
for(int j=0;j < (sizeOfRLEdata/i);j++)  
{  
DWORD temp,nextTemp,prevTemp;  
temp=tempArray[j];  
if(temp==0xffffffff)  
{  
nextTemp=tempArray[j+1];  
if (nextTemp==0xffffffff)  
{  
((DWORD*)destination)[m] = temp;  
j++;  
m++;  
}  
else  
{  
prevTemp=tempArray[j-1];  
numberOfdata = (WORD)tempArray[j+1];  
((DWORD*)destination)[m] = prevTemp;  
ummer = (int)numberOfdata;  
if(nummer<=2)  
{  
((DWORD*)destination)[m] = prevTemp;  
}  
else if(nummer==3)  
{  
for(int k=1;k < nummer-2;k++)  
{  
((DWORD*)destination)[m + k] = prevTemp;  
}  
m=m+nummer-3;  
}  
else  
{  
for(int k=1;k < nummer-1;k++)  
{  
}  
}  
}  
}  
}
((DWORD*)destination)[m + k] = prevTemp;

    }
    m=m+nummer-2;
    }
    j++;}
} else
{
    ((DWORD*)destination)[m] = temp;
    }
    m++;
}
success=true;
return destination;
}

DWORD DeCompression::Conversion(DWORD dword)
{
    return((dword>>24)&0x000000FF)|((dword>>8)&0x0000FF00)|
((dword<<8)&0x00FF0000) | ((dword<<24)&0xFF000000);}
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

describe entity runlengthencoding is
  port(
    i_clk    : in  std_logic;
    i_rst    : in  std_logic;
  );
i_wren : in  std_logic;
i_data_in : in  std_logic_vector(32 downto 0);
i_pkt_end : in  std_logic;
i_pkt_length : in  std_logic_vector(15 downto 0);

-- To --
o_kompout : out std_logic_vector(31 downto 0);
o_kompdatavalid : out std_logic;
o_kompout_last : out std_logic;
o_kompstatreg : out std_logic_vector(31 downto 0);
o_pkt_length : out std_logic_vector(15 downto 0);
o_pkt_end : out std_logic);
end runlengthencoding;

architecture rle_arch of runlengthencoding is

component RLE_fifo
port ( 
clk: IN std_logic;
srst: IN std_logic;
din: IN std_logic_VECTOR(32 downto 0);
wr_en: IN std_logic;
rd_en: IN std_logic;
dout: OUT std_logic_VECTOR(32 downto 0);
full: OUT std_logic;
empty: OUT std_logic;
valid: OUT std_logic;
prog_full: OUT std_logic;
prog_empty: OUT std_logic);
end component;

--------------------------

------
-- Signals
--------------------------
------
signal s_data_d1 : std_logic_vector(31 downto 0);
signal s_input : std_logic_vector(32 downto 0);
signal prog_full : std_logic;
signal prog_empty : std_logic;
signal full : std_logic;
signal empty : std_logic;
signal valid : std_logic;
signal rd_en : std_logic;
signal s_dataout  : std_logic_vector(31 downto 0);
signal s_dataout_d : std_logic_vector(31 downto 0);
signal s_validout : std_logic;
signal s_validout_d : std_logic;
signal status     : std_logic_vector(31 downto 0);
signal s_runlength : integer:=1;
signal s_pkt_end   : std_logic;
signal s_pkt_end_d : std_logic;
signal s_pkt_end_d2: std_logic;
signal s_pkt_length_2 : UNSIGNED(13 DOWNTO 0);
-- state type declarations
     type state_t is (idle_st, start_st, start_2_st, number_st, send_st,
                    send_data_st, wait_st);
signal state : state_t;

begin
     rle_fifo_1 : RLE_fifo
        port map (  
          clk => i_clk,
          srst => i_rst,
          din => i_data_in,
          wr_en => i_wren,
          rd_en => rd_en,
          dout => s_input,
          full => full,
          empty => empty,
          valid => valid,
          prog_full => prog_full,
          prog_empty => prog_empty);

process(i_clk, i_rst)
begin
     if (i_clk'event and i_clk = '1') then
          if (i_rst = '1') then  -- resets registers
              rd_en <= '0';
          end if;
     end if;
end process;
s_data_d1 <= x"00010001";
state <= idle_st;
s_validout <= '0';
s_runlength <= 1;
s_dataout <= x"00000000";
status <= x"00000000";
s_pkt_end <= '0';
s_pkt_end_d <= '0';

else
    case state is
        --Waits for the first IDLE-sign
        when idle_st =>
            if (full = '0' and empty = '0') then
                if (prog_empty = '0') then
                    --Waits until the buffersize in the FIFO is big enough
                    rd_en <= '1';   --Starts the reading of the memory
                    state <= start_st;   --shows that the algorithm are working
                else
                    state <= idle_st;
                end if;
            else
                s_validout <= '0';
                state <= idle_st;
            end if;
        when start_st =>
            if (full = '0' and empty = '0') then
                if (valid = '1') then
                    if (s_input(31 downto 0) = x"FFFFFFFF") then
                        --if the data is the controlsign
                        s_dataout <= s_input(31 downto 0);
                        s_validout <= '1';
                        rd_en <= '0';
                        s_data_d1 <= s_input(31 downto 0);
                        --saves the data in the s_data_d1 register to be sent in the next
                        state <= send_st;
                    else
                        s_data_d1 <= s_input(31 downto 0);  --if the data is'nt the
                        control sign
                        rd_en <= '1';   --sets the FIFO for reading on the next
        end when;

    end case;
s_dataout <= s_input(31 downto 0);  --sends the data to the output since there is nothing to compare with
    s_validout        <= '1';
    state             <= start_2_st;
  end if;
  else
    s_validout        <= '0';  --if the data is not valid
  end if;
  --when read from the FIFO the algorithm waits for the FIFO
  rd_en             <= '1';
  state             <= start_st;
  end if;
  else
    s_validout        <= '0';
    state             <= idle_st;
  end if;  --Reads another set of data from the bus and
  --compares them.
    when start_2_st =>
      if(s_pkt_end='1')then
        s_pkt_end <= '0';
      end if;
      if (s_input(32) = '0') then
        if (full = '0' and empty = '0') then
          if (s_input(31 downto 0) = x"FFFFFFFF" and s_runlength = 1
              and valid = '1') then  --if the read data is the controlsign
            s_dataout <= s_input(31 downto 0);  --sends the previous
            s_validout <= '1';
            rd_en     <= '0';
            s_data_d1 <= s_input(31 downto 0);
            state     <= send_st;  --goes to a state
          where the controlsign is sent twice
          else
            if (s_input(31 downto 0) = s_data_d1 and valid = '1') then
              --if the read data is the same as the previous data
              s_validout <= '0';
              s_runlength <= s_runlength + 1;
              --the counter counts the number of times a sign is repeated
              rd_en     <= '1';
              state     <= start_2_st;
            else
              if (s_runlength > 1 and valid = '1') then
                --if the sign has been repeated but now the repetition has ended
                if (s_runlength = 2) then

s_validout <= '1';
s_data_d1  <= s_input(31 downto 0);
s_dataout  <= s_input(31 downto 0);
state      <= start_2_st;
rd_en      <= '1';
s_runlength <= 1;
else
  rd_en      <= '0';
s_dataout  <= x"FFFFFFFF";
--the controlsign is sent
s_validout <= '1';
s_data_d1  <= s_input(31 downto 0);
s_dataout  <= s_input(31 downto 0);
state      <= number_st;
--and the number of times the sign has repeated are sent next
end if;
elsif (valid = '1') then
  s_validout  <= '1';
s_data_d1   <= s_input(31 downto 0);
s_dataout   <= s_input(31 downto 0);
state       <= start_2_st;
rd_en       <= '1';
else
  s_validout <= '0';
  state     <= start_2_st;
end if;
else
if (s_input(31 downto 0) = x"FFFFFFFF" and s_runlength =1 and valid = '1')
  then --if the read data is the controlsign
    s_dataout <= s_input(31 downto 0); --sends the previous data
    s_validout <= '1';
    rd_en      <= '0';
    s_data_d1  <= s_input(31 downto 0);
    state      <= send_st;
    --goes to a state where the controlsign is sent twice
  else
    if (s_input(31 downto 0) = s_data_d1 and valid = '1') then
      --if the read data is the same as the previous data
      s_validout      <= '0';
      s_runlength     <= s_runlength + 1;
      --the counter counts the number of times a sign is repeated
      rd_en            <= '1';
    else
      ...
state <= start_2_st;
else
     if (s_runlength > 1 and valid = '1') then
--if the sign has been repeated but now the repetition has ended
       if (s_runlength = 2) then
          s_validout <= '1';
          s_data_d1 <= s_input(31 downto 0);
          s_dataout <= s_input(31 downto 0);
          state <= start_2_st;
          rd_en <= '1';
          s_runlength <= 1;
       else
          rd_en <= '0';
          s_dataout <= x"FFFFFFFF";
       end if;
     else
       if(s_runlength > 1 and valid = '1')then
          rd_en <= '0';
          s_dataout <= x"FFFFFFFF";
          --the controlsign is sent
       else
          s_validout <= '0';
          state <= start_2_st;
       end if;
     end if;
end if;
else
     if(s_runlength > 1 and valid = '1')then
          rd_en <= '0';
          s_dataout <= x"FFFFFFFF";
          --the controlsign is sent
          s_validout <= '1';
          s_data_d1 <= s_input(31 downto 0);
          s_dataout <= s_input(31 downto 0);
          state <= start_2_st;
          rd_en <= '1';
     else
          s_validout <= '0';
          state <= start_2_st;
     end if;
end if;
else
     if(s_runlength > 1 and valid = '1')then
          rd_en <= '0';
          s_dataout <= x"FFFFFFFF";
          --the controlsign is sent
          s_validout <= '1';
          s_data_d1 <= s_input(31 downto 0);
          state <= number_st;
          s_pkt_end_d <= '1';
     end if;
end if;
elsif(s_input(31 downto 0) = x"FFFFFFFF" and valid = '1') then
    s_dataout <= s_input(31 downto 0);  --sends the previous data
    s_validout <= '1';
    rd_en     <= '0';
    s_data_d1 <= s_input(31 downto 0);
    state     <= send_st;
    s_pkt_end_d <= '1';
elsif (valid = '1') then
    s_validout <= '1';
    s_data_d1 <= s_input(31 downto 0);
    s_dataout <= s_input(31 downto 0);
    state     <= start_2_st;
    rd_en     <= '1';
    s_pkt_end  <= '1';
else
    s_validout <= '0';
    state     <= start_2_st;
end if;
end if;
when number_st =>
    if(s_pkt_end_d='1') then
        s_pkt_end  <= '1';
        s_pkt_end_d <= '0';
    end if;
    s_dataout <= std_logic_vector(to_unsigned(s_runlength, 32));  --when the controlsign was sent the next sign to be sent is the number of repets
    s_validout <= '1';
    rd_en      <= '0';
    s_runlength <= 1;
    state      <= send_data_st;
when send_st =>  --sends the allready collected data from the FIFO
    if(s_pkt_end_d='1') then
        s_pkt_end  <= '1';
        s_pkt_end_d <= '0';
    end if;
    s_dataout <= x"FFFFFFFF";  --shows that the data have been sent
    s_validout <= '1';
    rd_en      <= '1';
    s_data_d1  <= s_input(31 downto 0);
    state      <= start_2_st;
when send_data_st =>  --sends the allready collected data from the FIFO  
  if(s_pkt_end='1')then  
    s_pkt_end <= '0';  
  end if;  
  s_dataout <= s_data_d1;  --shows that the data have been sent  
  s_validout <= '1';  
  rd_en <= '1';  
  state <= wait_st;  
when wait_st =>  
  s_dataout <= s_input(31 downto 0);  
  s_validout <= '1';  
  state <= start_2_st;  
when others =>  
end case;  
end if;  
end if;  
end process;  

process(i_clk, i_rst)  
begin  
  if (i_clk'event and i_clk = '1') then  
    if (i_rst = '1') then  
      -- resets registers  
      s_pkt_length_2 <= (OTHERS => '0');  
      s_pkt_end_d2 <= '0';  
      s_dataout_d <= (OTHERS => '0');  
      s_validout_d <= '0';  
      o_pkt_length <= x"0000";  
    else  
      s_pkt_end_d2 <= s_pkt_end;  
      s_dataout_d <= s_dataout;  
      s_validout_d <= s_validout;  
      if (s_pkt_end = '1') then  
        o_pkt_length <= STD_LOGIC_VECTOR(s_pkt_length_2 + 1) & "00";  
        s_pkt_length_2 <= (OTHERS => '0');  
      elsif (s_validout = '1') then  
        s_pkt_length_2 <= s_pkt_length_2 + 1;  
      end if;  
    end if;  
  end if;  
end if;  
end process;
-- Assign outputs

o_kompout <= s_dataout_d;
o_kompdatavalid <= s_validout_d;
o_kompstatreg <= status;
o_pkt_end <= s_pkt_end_d2;
o_kompout_last <= s_pkt_end_d2;
end rle_arch;
Appendix D

Testbench code, RLE:

clear all;

fid = fopen('referens.txt');
[A_orig count_orig] = fread(fid);

fid = fopen('referens.txt');
[A count] = fread(fid);

%A=sort(A)%% contains all characters

[r c d]=size(A);
k=1;
x=1;
RL=1;
n=1;
while k<r
    if A(k)==999
        n=r;
        while n>k
            A(n+1) = A(n);
            n=n-1;
        end;
        n=1;
        A(k+1)=999;
    else
        if A(k)==A(k+1)
            k=k-1;
            RL = RL + 1;
            A(k+1) = [];
            r = r-1;
        else
            if RL > 1
                n=r;
                while n>k
                    A(n+2) = A(n);
                    n=n-1;
                end;
                n=1;
                A(k+1)=999;
                A(k+2)=RL;
                RL=1;
                k=k+2;
            end;
        end;
    end;
k=k+1;
end;
if RL>1
    A(k+1)=00;
    A(k+2)=RL;
    RL=1;
end;

Huffman:

clear all;

feed = fopen('referens.txt');
[Fq count_2] = fread(feed);

Fq = sort(Fq);

[numb b c]=size(Fq);

unique =0;
c_count=0;

it=1;
count=1;

while it<numb
    if Fq(it) == Fq(it+1)
        count=count+1;
    else
        c_count=c_count+1;
        unique(c_count,2)=Fq(it);
        unique(c_count,1)=count;
        count=1;
    end

    it= it + 1;
    if(it == numb)
        c_count=c_count+1;
        unique(c_count,2) = Fq(it);
        unique(c_count,1) = count ;
    end
end;

unique = sortrows(unique, 1);
unique = flipdim(unique, 1);
symbol_length = [0 0];
[r_unique c_unique]=size(unique);
it2=1;

k=r_unique;
length_c = unique;
length_c(:,3)=0;

while it2<k
    [r_temp c_temp]=size(unique(r_unique,:));
    [r_temp2 c_temp2]=size(unique(r_unique-1,:));
    m=1;
    c_temp3=1;
    for n=1:c_temp2
        if(unique(r_unique-1,n)== 0)
            break;
        else
            c_temp3=c_temp3+1;
        end;
    end;
    while m<c_temp
        if (unique(r_unique,m+1)==0)
            unique(r_unique-1,c_temp3) =unique(r_unique,m+1);
            c_temp3=c_temp3+1;
        else
            break;
        end;
        m=m+1;
    end;
    unique(r_unique-1,1)=unique(r_unique,1)+unique(r_unique-1,1);
    unique(r_unique,:)=[];
    [r_temp4 c_temp4]=size(unique(r_unique-1,:));
    for n=2:c_temp4
        a=find(length_c(:,2)==unique(r_unique-1,n),1);
        length_c(a,3)=length_c(a,3)+1;
    end;
    unique=flipdim(sortrows(unique,1),1);
    it2=it2+1;
    r_unique=r_unique-1;
end;

summa=0;
for o=1:c_count
    summa=summa+length_c(o,1)*length_c(o,3);
end;