Data compression and decompression scheme for on-chip communication

Author: Saksham Kiroriwal (1520925)

Supervising Professor: Prof. Dr.-Ing. Roman Obermaisser
Supervising tutor: Dr. Daniel Onwuchekwa

A document presented for the Studienarbeit

Department Elektrotechnik und Informatik, Embedded Systems
Universität Siegen
07/07/2021

Contents

List of Figures

List of Tables

1 Introduction

The field of information technology has developed greatly in these recent years. This development also called for storage of huge amounts of data. With the ever-growing demands of the information storage, compression has become a necessity. It is only logical to save storage space because of compression. This saved space can be used to store other data too. The scope of the compression is not only limited to the information storage but also facilitates low transfer time for the messages to be transferred. Low transfer time, not only plays a crucial role in time-critical applications but also reduces the power consumption of the devices. Compression is the science of representing the information in a compact form[6].

Today there are several kinds of compression algorithms available that are used in different cases depending on the data that needs to be compressed. For example, some algorithms might use dictionary to represent a commonly occurring string of symbols with a shorter codes and a reference to the code is made in the compressed format.Some algorithms exploit existing patterns in the data. Some standard algorithms are used to represent the data in terms of other parameters so that the bits per symbol in the code can be reduced. Few early examples of data compression are Morse Code, Grade 1 and Grade Braille code. For example, in Morse Code, the letters that have the highest frequency of occurrence, are assigned the shorter sequences. Because of this scheme, the average time required to transmit the message is reduced. Text compression can usually succeed by removing all unnecessary characters, instead inserting a single character as reference for a string of repeated characters, then replacing a smaller bit string for a more common bit string. Broadly speaking the data compression is of two types: *lossless* compression and *lossy* compression.

The field of information technology has significantly developed in these recent years. This development also called for the storage of vast amounts of data. With the ever-growing demands of information storage, compression has become a necessity. Compression techniques reduce the amount of required storage in any devices. In addition, it reduces the transfer time of data in off-chip and on-chip communication and plays a crucial role in power management techniques. Compression is the science of representing the information in a compact form[6].

Today, several compression algorithms are available in different cases depending on the data that needs to be compressed. For example, some algorithms might use a dictionary to represent a commonly occurring string of symbols with a shorter code. A reference to the code is made in compressed format. Some algorithms exploit existing patterns in the data. Some standard algorithms are used to represent the data in terms of other parameters so that the bits per symbol in the code can be reduced. Few early examples of data compression are Morse Code, Grade 1 and Grade Braille code. For example, in Morse Code, the letters that have the highest frequency of occurrence are assigned the shorter sequences. Because of this scheme, the average time required to transmit the message is reduced. Text compression can usually succeed by removing all unnecessary characters instead of inserting a single character as a reference for a string of repeated characters, replacing a smaller bit string for a more common bit string. Broadly speaking the data compression is of two types: *lossless* compression and *lossy* compression.

The field of information technology has developed greatly in these recent years. This development also called for storage of huge amounts of data. With the ever-growing demands of the information storage, compression has become a necessity. It is only logical to save storage space because of compression. This saved space can be used to store other data too. The scope of the compression is not only limited to the information storage but also facilitates low transfer time for the messages to be transferred.

Low transfer time, not only plays a crucial role in time-critical applications but also reduces the power consumption of the devices. Compression is the science of representing the information in a compact form[6].

Today there are several kinds of compression algorithms available that are used in different cases depending on the data that needs to be compressed. For example, some algorithms might use dictionary to represent a commonly occurring string of symbols with a shorter codes and a reference to the code is made in the compressed format.Some algorithms exploit existing patterns in the data. Some standard algorithms are used to represent the data in terms of other parameters so that the bits per symbol in the code can be reduced. Few early examples of data compression are Morse Code, Grade 1 and Grade Braille code. For example, in Morse Code, the letters that have the highest frequency of occurrence, are assigned the shorter sequences. Because of this scheme, the average time required to transmit the message is reduced. Text compression can usually succeed by removing all unnecessary characters, instead inserting a single character as reference for a string of repeated characters, then replacing a smaller bit string for a more common bit string. Broadly speaking the data compression is of two types: *lossless* compression and *lossy* compression.

1.1 Lossless Compression Techniques

Lossless Compression is used when the loss of any information cannot be tolerated. It means the original data and the reconstructed data should be similar. Bits used to represent the data are reduced by locating and removing statistical redundancies. Because of this, no information is lost during the process. Text compression is a significant area for lossless Compression. The Lempel–Ziv (LZ) compression methods are among the most popular algorithms for lossless storage[4]. The most prominent models include the Huffman algorithm and Randomized algorithms. Arithmetic coding is another technique used in lossless data compression. These algorithms will be discussed in detail later.

1.2 Lossy Compression

As the name suggests, the data is not reconstructed precisely. Only an approximation of the original data can be regenerated. Typically the encoder reduces the number of bits that are used to represent the samples, and the unnecessary information is discarded. Higher compression is possible with lossy compression techniques as compared to the lossless compression techniques. These kinds of compression schemes are incorporated for image, audio or video compression. For example, the human eye is more sensitive to the changes in luminosity rather than the subtle colour changes. This fact is exploited in the image compression schemes where some samples are rounded off. A majority of these schemes were made for the multimedia industry in the 1980s. The exactness of the regenerated data to the original data depends on the application.

1.3 Measurement of Performance

One way of expressing the performance of the compression is *compression ratio*. As the name suggests, it is simply the ratio of the number of bits required to represent the data before and after the compression. Another way of describing the compression ratio is by reducing the amount of data needed as a percentage of the original size. Another way of expressing the performance is *rate*. It is defined as the number of bits required to represent a single sample.

In lossy compression, the reconstructed data differs from the original data. The difference between the original and the reconstructed data is called *distortion.* For the lossy compression schemes, terms like fidelity and quality are also used.

1.4 Information Theory

In the information theory, Claude Elwood Shannon defined a quantity called *self-information.* This quantity is a way to look at the measure of information. If an event has a low probability, the amount of self-information contained is low. It is intuitive because if an event with low probability, occurs; it contains a lot of information about what must have happened for that to occur. Self information is defined by

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A)$$

Another important quantity is *entropy.* Entropy is the average self-information associated with the experiment. Consider a set of independent events A_i, which are sets of outcomes of some experiment S, such that S is the sample space. Then the average self-information associated with the random experiment is given by

$$H = \sum P(A_i) i(A_i) = -\sum P(A_i) \log_b P(A)$$

[4] .Shannon also proved that entropy is a measure of the average number of binary symbols needed to code the source's output.

1.5 Modeling and Coding

There are two phases for developing the compression algorithms for the different types of data, namely: *modelling* and *coding.* In the first phase, we try to extract the information about any dependency or redundancy in the data and develop a model. Before we develop techniques to code the data, we need to understand the mathematical model for the data. This can be achieved in various ways. Coming up with a good model can prove to help estimate the entropy of the system. Good models also lead to more efficient compression algorithms.

In the second phase, the description of the model and the data encoding is done. Usually, binary alphabets are used. For example, consider a sequence of numbers 7,9,9,11,10,13,13,15. If we store these numbers, we need 5 bits per sample. But this data can also be looked at from a different perspective. Consider a polynomial y=n+6, where n is the position of the sample in the data. If 'y' is subtracted from all the samples, we are left with a sequence like this; 1,1,0,1,-1,1,0,1. Now the sequence only has three distinct kinds of samples. Now we need only 2 bits per sample to store the information. When the data is to retrieved back, the same algorithm can be applied in reverse. The end node should need only the compressed data and the polynomial that describes the model.

Coding means the assignment of binary sequences to the elements of an alphabet. The set of binary sequences is called a code, and the individual members of the set are called code words. An alphabet is a collection of symbols called symbols. Code that assigns fixed-length codes to the elements is called *fixed-length code.* The ASCII code is a type of fixed-length code. Some algorithms incorporate the codes which represent the frequently occurring symbols using fewer bits. Morse code uses the same concept. The average number of bits per symbol is also called the *rate* of the code.

2 Lossless Compression

2.1 Huffman Coding

This technique was developed by David Huffman as part of a class assignment; the class was the first-ever in the area of information theory and was taught by Robert Fano at MIT. The codes generated using this technique or procedure are called Huffman codes. These codes are prefix codes and are optimum for a given model (set of probabilities). The Huffman procedure is based on two observations regarding optimum prefix codes[6]. Prefix codes are codes in which no code is a prefix of another code.

1. In an optimum code, symbols that occur more frequently (have a higher probability of occurrence) will have shorter code words than symbols that occur less frequently.
2. In an optimum code, the two symbols that occur least frequently will have the same length[6].

Huffman Coding is a variable-length coding. David Huffman also proved that Huffman coding is the most efficient way to encode a text-based on single character encoding. For Huffman coding, the probability distribution for the text is to be known. The algorithm is as follows:

1. Select two symbols with the least frequency and place the one with the larger frequency out of the two, on the right. The left child has less frequency than the right child.
2. The probability of the parent node is equal to the sum of the probabilities of the children.
3. Move the parent node with new probability up the tree, according to step 1.
4. Repeat the steps until all the elements are put in the tree.

Example: Consider the an alphabet $\mathcal{A} = \{ABCDABCCCEDAABCDEDDD\}$. The probability distribution for the elements is P(A)=0.2, P(B)=0.15, P(C)= 0.25, P(D)= 0.3, P(E)= 0.1. To start with tree, E and B have the least probability. B is kept on the right to E. The node now has the probability equal to the sum of the probabilities of the children. The probability of the node is 0.25. So B is represented by code 1 and E is represented by 0. The node B'E' is now compared with other symbols. In the next step, node B'E' is denoted by 1 and A is denoted by 0. By continuing the steps, Huffman Tree (as shown in figure 2) will be obtained.

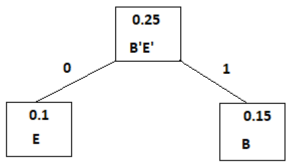


Figure 1: First node in the Huffman Tree

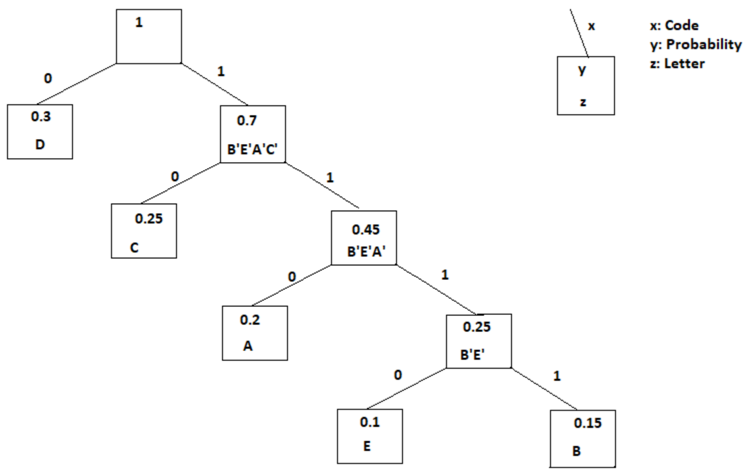


Figure 2: Complete Huffman Tree

Symbol	Count	Code	No. of Bits
A	4	110	12
B	3	1111	12
C	5	10	10
D	6	0	6
E	2	1110	8

Table 1: Huffman Code for the above example

The final Huffman tree that will be obtained will be like the one shown in the picture above. For decoding, just follow the tree. For example, in 1110, denotes the symbol E. It can be seen that the average bit number of each symbol is 2.4 bits/symbol. If we were to use the standard coding, it would take 3 bits per symbol to represent all the symbols.

Huffman algorithm is one of the most popular compression algorithms, because of its flexibility to be used for different kinds of files and easy implementation. Huffman algorithm has one disadvantage, that the probability distribution of the symbols is to be known beforehand.

Huffman coding procedure generates an optimum code. The length of any code will depend on several things, including the size of the alphabet and the probabilities of individual letters. Using Kraft-McMillan inequality, it can be proved that, the Huffman code for the source, has an average code length 'l' bounded below by the entropy and bounded above by the entropy plus 1 bit.

$$H(S) <= l <= H(S) + 1.$$

Huffman coding is optimal among all methods in any case where each input symbol is a known independent and identically distributed random variable having a probability that is dyadic. A dyadic distribution is a probability distribution whose probability mass function is

$$F(n) = 2^{-n}, n \epsilon N$$

2.2 Adaptive Huffman Coding

Huffman coding is dependent on the prior knowledge of the probability distribution of the symbols. But if this information is not present, then two passes are required for the encoding. In the first pass, the statistics are collected and in the second pass, the algorithm is implemented for encoding. But this is not very practical in all cases. To convert this algorithm into a one-pass procedure, Faller and Gallagher independently developed adaptive algorithms to construct the Huffman code based on the statistics of the symbols already encountered. These were later improved by Knuth and Vitter[6].

For adaptive Huffman coding, two new parameters were introduced, *weight* and *node number*. The weight of each leaf is written inside the node. It is simply the number of times, the symbol of that node has been encountered. There are two types of nodes: the external node and the internal node. External nodes are the ones which do not have any child leaf and internal nodes are the ones which have child leaves. The weight of the internal node is equal to the sum of the weights of the leaf nodes. If the number of symbols is n, the total number of nodes (internal nodes plus external nodes) is *2n-1*. The node number is a unique number that is assigned to the node from $y_1, y_2 .... y_{2n-1}$.

Adaptive Huffman coding is also a variable-length coding in which, neither transmitter nor receiver knows anything about the statistics of the source sequence at the start of transmission. The tree at both the transmitter and the receiver consists of a single node that corresponds to all symbols not yet transmitted (NYT) and has a weight of zero. As transmission progresses, nodes corresponding to symbols transmitted will be added to the tree, and the tree is reconfigured using an update procedure. Before the beginning of transmission, a fixed code for each symbol is agreed upon between transmitter and receiver.[6]

Adaptive Huffman algorithm doesn't require the prior knowledge of the statistics of the data, and it assigns code to the symbol on the fly. Because of this, Huffman coding is more sensitive to transmission errors. It requires more space to store the data and all the trees have to be transmitted to decode the code. Not only this, but the hardware implementation of the adaptive Huffman Coding also becomes difficult. This paper aims to reduce the latency of the data transmission in Ethernet, by implementing compression techniques on FPGA. We won't discuss more the adaptive Huffman code since it is out of the scope of the paper.

Huffman coding algorithm and adaptive Huffman coding algorithm are very popular. But they both are variable-length codes. In certain situations, algorithms like Tunstall code and Golomb can be useful. These algorithms are fixed-length codes.

2.3 Canonical Huffman Code

Now that we have discussed the Huffman coding, we know how the Huffman tree is made and it's use. The encoding is easy and efficient. But to decoding with this scheme is not efficient. However, decoding the bitstream generated using this technique is inefficient.Decoders(or Decompressors)require the knowledge of the encoding mechanism used in order to decode the encoded data back to the original characters. Hence information about the encoding process needs to be passed to the decoder along with the encoded data as a table of characters and their corresponding codes. In regular Huffman coding of a large data, this table takes up a lot of memory space and also if a large no. of unique characters are present in the data then the compressed(or encoded) data size increases because of the presence of the codebook. Therefore to make the decoding process computationally efficient and still maintain a good compression ratio, Canonical Huffman codes were introduced.

In Canonical Huffman coding, the bit lengths of the standard Huffman codes generated for each symbol is used. The symbols are sorted first according to their bit lengths in non-decreasing order and then for each bit length, they are sorted lexicographically. The first symbol gets a code containing all zeros and of the same length as that of the original bit length. For the subsequent symbols, if the symbol has a bit length equal to that of the previous symbol, then the code of the previous symbol is incremented by one and assigned to the present symbol. Otherwise, if the symbol has a bit length greater than that of the previous symbol, after incrementing the code of the previous symbol zeros are appended until the length becomes equal to the bit length of the current symbol and the code is then assigned to the current symbol. This process continues for the rest of the symbols.

The normal Huffman Coding assigns variable length optimal codes to the symbols in the alphabet. This is done according to the procedure as discussed above. For example the variable length codes that have been assigned are the ones discussed above in the table 1.

In Canonical Huffman Coding, bit length stays the same but the codes are different from the normal huffman coding. To do so, the symbols are first sorted lexicographically, i.e. by the codeword length and then alphabetically. So according to the algorithm discussed above, the new Huffman Codes would be:

Symbol	Code
D	0
C	10
A	110
B	1110
E	1111

Table 2: Huffman Code for the above example

Symbol	Code
A	110
B	1110
C	10
D	0
E	1111

Table 3: Canonical Huffman Code for the above example

With the canonical version we have the knowledge that the symbols are arranged alphabetically and we are also aware of the canonical algorithm. This provides us with the advantange that only the canonical code bit length needs to be transmitted to the decoder. Once the length of codewords for different symbols is received, the codes can be regenerated at the decoder end. This saves us the time and bandwidth for the transmission of Huffman tree. The canonical codes can then be obtained using incrementation and bitwise left shift operations.

2.4 Tunstall Coding

Most of the variable-length codes that we look at in this book encode letters from the source alphabet using codewords with varying numbers of bits: codewords with fewer bits for letters that occur more frequently and codewords with more bits for letters that occur less frequently. The Tunstall code is

an important exception. In the Tunstall code, all codewords are of equal length. However, each codeword represents a different number of letters. The main advantage of a Tunstall code is that errors in codewords do not propagate, unlike other variable-length codes, such as Huffman codes, in which an error in one codeword will cause a series of errors to occur.[6]
This property of equal length codes, makes is a very suitable algorithm where constant data transmission rate is required.

Suppose we want an n-bit Tunstall code for a source that generates indepedent and identically distributed letters from an alphabet of size N. The number of codewords is 2n. We start with the N letters of the source alphabet in our codebook. Remove the entry in the codebook that has the highest probability and add the N strings obtained by concatenating this letter with every letter in the alphabet (including itself). This will increase the size of the codebook from N to N +(N -1)The probabilities of the new entries will be the product of the probabilities of the letters concatenated to form the new entry. Now look through the N +(N -1) entries in the codebook and find the entry that has the highest probability, keeping in mind that the entry with the highest probability may be a concatenation of symbols. Each time we perform this operation we increase the size of the codebook by N -1. Therefore, this operation can be performed K times, where $N + K(N - 1) \leq 2^n$.

Similar to the Huffman Code, Tunstall Code also depends on the probability distribution of the symbols in the given alphabet. First, we decide the length of the Tunstall Code that we wish to make. So this means that the maximum number of different codes possible is 2^n, where n is the length of the Tunstall code. The symbols are arranged in the descending order of the probabilities. The one with the highest probability is combined with the other symbols. The probability of the new set of symbols is equal to the product of the probabilities of the two symbols. Now, these sets of symbols are considered as a separate symbol and the probabilities are compared with other others'. The same procedure is repeated until we arrive at the maximum possible number of entries. We try to make as many combinations of symbols as possible but the number should be less than 2^n. For example, consider the probability distribution given below, and we wish to make 3 bit Tunstall Code.

$$\mathcal{A} = A, B, C, D$$

$$P(A) = 0.4, P(B) = 0.35, P(C) = 0.25$$

Symbol	Probability
A	0.4
B	0.35
C	0.25

Table 4: Source symbols and their probabilities

Symbol	Probability
B	0.35
C	0.25
AA	0.16
AB	0.14
AC	0.1

Table 5: Symbols after first iteration and their probabilities

Symbol	Code
C	000
AA	001
AB	010
AC	011
BA	100
BB	101
BC	110

Table 6: Symbols after second iteration and their codes

We start out with the symbols arranged with the descending probabilities. In the first iteration, we find out that 'A' has the highest probability, so it is taken off the table and combined with other symbols. Then new probabilities for the symbols is calculated by multiplying the probabilities of the two. In the second iteration, we see that 'B' has the highest probability, so it is taken off the table and combined with the symbols in the first table. After this, we have 7 entries in the table, we combine 'B' with symbols in second table 2, then we will have 10 entries which will be more than the maximum possible number of entries. So we stop here and the codes are assigned.

2.5 LZ77

LZ77 is a type of loseless compression technique that was developed by Abraham Lempel and Jacob Ziv in 1977. This pair also published another algorithm in 1978, which came to be known as LZ78. These two compression techniques form the basis of the LZ compression family and also many other modern day compression techniques. DEFLATE algorithm is one of the prominent applications of the LZ77 compression. Due to its adaptability, it is used in various formats. It was also proved that the Lempel Ziv algorithm's compression rate asymptotically approaches the entropy, as the size of the sequence reaches infinity.[7]

It is a dictionary based coding technique in which a sliding window is maintained. The sliding window comprises of the search buffer and the look ahead buffer. The encoding and decoding is done from the sliding window, based on the previously seen characters. The encoding of each repeated character is shown by **length-distance pair**. A dictionary is maintained of the phrases that are found and it is then appended to the final file. The length of the dictionary is fixed to a certain value.

The sliding window consists of two buffers, look ahead buffer and search buffer. The look ahead buffer contains the characters to be encoded. These characters are compared to the ones in the search buffer for a match. If a match is found, the phrase or the match is encoded using the length-distance pair. The size of the sliding window is one of the key factors affecting the compression efficiency.

The length-distance pair indicates that the "length" number of characters are same as the characters "distance" characters behind. In LZ77 algorithm, the compressor searches back through the search buffer until it finds a match to the first character in the look-ahead buffer. There could be more than one matches exist in the search buffer, and the compressor will find the one match having the longest length. When the longest match is found, the compressor encodes it with a triple (D, L, C) where:

- D = distance of the search cursor from the start of look-ahead buffer
- L = length of longest match

- C = next character in look-ahead buffer beyond longest match [1]

Suppose a given string "cabracadabrarrarrad" is to be compressed using the LZ77 technique. The search buffer is of length seven and the look ahead buffer is of the length six. Considering this, the current condition follows. "cabraca" in the search buffer and the "dabrar" in the look ahead buffer. Since the first literal was never encountered in the dictionary before, 'd' is encoded as $< 0, 0,'d' >$. The format is $< distance, length, nextCharacter >$. Now the elements in the search buffer and the look ahead buffer are "abracad" and "abrarr" respectively. As we know, LZ77 tries to find the longest match for the string in the look ahead buffer with the string in the search buffer, we find that "abra" reappears. So the encoding will be $< 7, 4,'r' >$. The sliding window now will look like "adabrar" in the search buffer and "rarrad" in the look ahead buffer. Now the look-ahead buffer contains the string "rarrad". Looking back in the window, we find a match for 'r' at an offset of one and a match length of one, and a second match at an offset of three with a match length of what at first appears to be three. It turns out we can use a match length of five instead of three.[6]. The encoding will be $< 3, 5,'d' >$. As we see, the match has to start in the search buffer and it can extend into the look ahead buffer.

The efficiency of the LZ77 algorithm depends on the length of the sliding window. Since the match has to start in the search buffer, a potential match can be neglected if the match falls out of the search buffer. But even with that, this technique was named IEEE Milestone in 2004. This is used in parallel with Huffman Coding in a compression technique known as DEFLATE.

2.6 DEFLATE

This specification defines a lossless compressed data format that compresses data using a combination of the LZ77 algorithm and Huffman coding. The data can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an a priori bounded amount of intermediate storage. DEFLATE is compression technique that uses a combination of LZ77 and Huffman encoding to compress data. It was designed by Phil Katz. This technique was later specified in the RFC 1951.

DEFLATE works in two phases. The data is divided into blocks. Each block is preceded with a 3-bit header.
The first bit also known as BFINAL is set if and only if this is the last block of the data set.

- 1 : "It is the last block in the stream"
- 0 : "There are more blocks in the stream".

The next two bits, BTYPE specify how the data is compressed.

- 00 : no compression
- 01 : compressed with fixed Huffman codes
- 10 : compressed with dynamic Huffman codes
- 11 : reserved (error)

In the compressed blocks, LZ77 and Huffman algorithm are applied. Similar to the LZ77 algorithm, if a duplicate series of bytes is spotted, then a back reference to the match is inserted. This is similar to what was explained in the LZ77 section.[2]

Use of Huffman in the "DEFLATE" format, BTYPE=01
The Huffman codes used for each alphabet in the "deflate" format have two additional rules:

- All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent
- Shorter codes lexicographically precede longer codes.

This implementation allows for match lengths of between 3 and 258. At each step the encoder examines three bytes. If it cannot find a match of at least three bytes it puts out the first byte and examines the next three bytes. So, at each step it either puts out the value of a single byte, or literal, or the pair $< matchlength, offset >$. The alphabets of the literal and match length are combined to form an alphabet of size 286 (indexed by 0–285). The indices 0–255 represent literal bytes and the index 256 is an end-of-block symbol. The remaining 29 indices represent codes for ranges of lengths between 3 and 258, as shown in Table 5. The table shows the index, the number of selector bits (extra bits) to follow the index, and the lengths represented by the index and selector bits. For example, the index 277 represents the range of lengths from 67 to 82. To specify which of the sixteen values has actually occurred, the code is followed by four selector bits.[6]

Code	Extra Bits	Lengths	Code	Extra Bits	Lengths	Code	Extra Bits	Lengths
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Table 7: Codes for representation of match length[2]

The index values are represented using a Huffman code. The Huffman code is specified in Table 6.

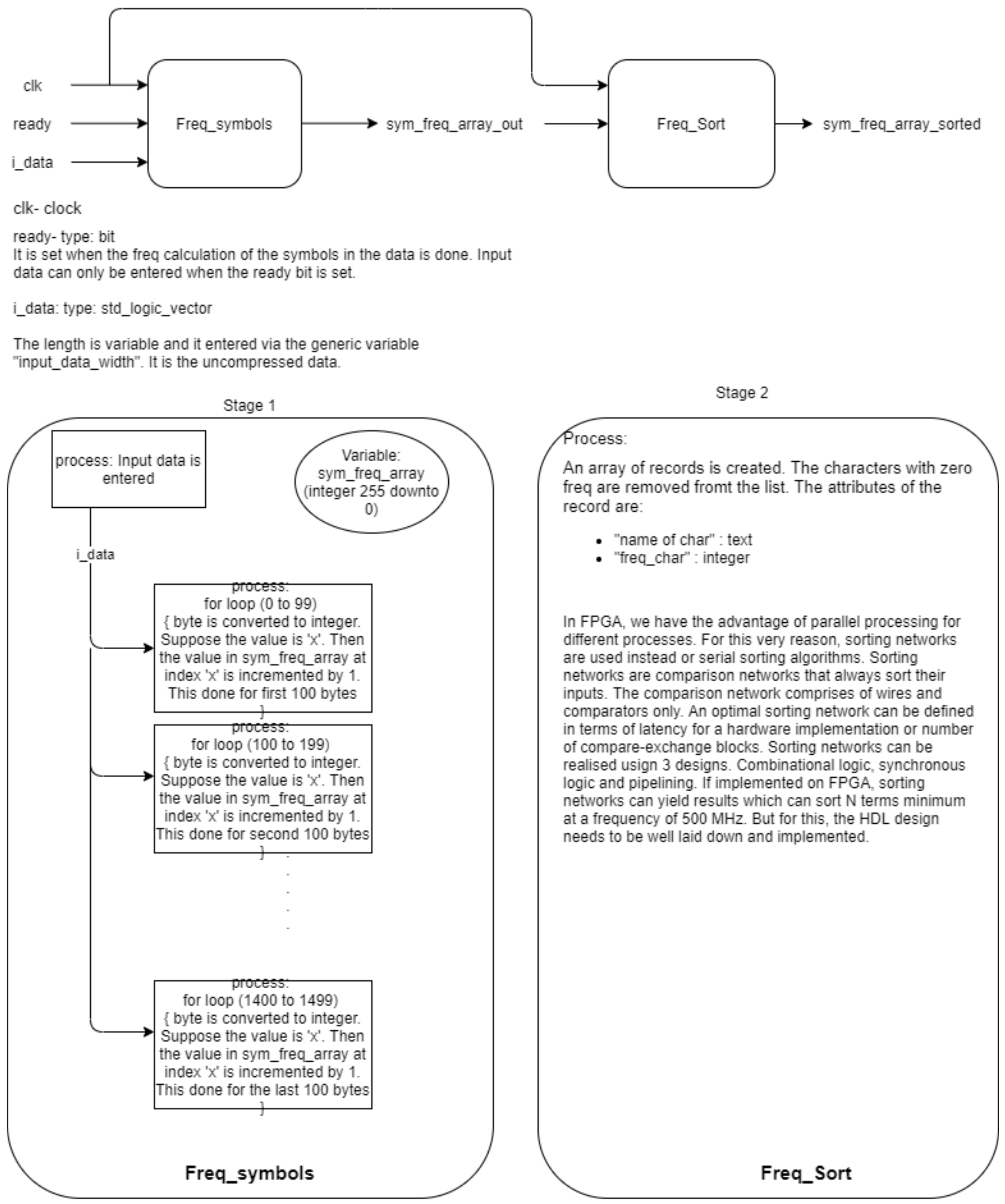
Lit Value	No. of Bits	Codes
0-143	8	00110000 through 10111111
144-255	9	110010000 through 111111111
256-279	7	0000000 through 0010111
280-287	8	11000000 through 11000111

Table 8: Huffman codes for the match length alphabet[2]

The offset can take on values between 1 and 32,768. These values are divided into 30 ranges. The thirty range values are encoded using a Huffman code (different from the Huffman code for the literal and length values) and the code is followed by a number of selector bits (extra bits) to specify the particular distance within the range.[6]

3 Hardware Implementation of Huffman Coding on FPGA

The realization of Huffman Coding on FPGA can be done in five stages. These stages have been mention in the figure representing the design flow of the Huffman Encoder. Huffman Coding requires the probability distribution of the symbols. Since the Huffman algorithm incorporated only addition operations on the probabilities, the codes can also be generated using the frequency of the symbols. In the first stage, the frequency of the symbols is calculated.



clk- clock

ready- type: bit
It is set when the freq calculation of the symbols in the data is done. Input data can only be entered when the ready bit is set.

i_data: type: std_logic_vector

The length is variable and it entered via the generic variable "input_data_width". It is the uncompressed data.

process: Input data is entered

process:
for loop (0 to 99)
{ byte is converted to integer. Suppose the value is 'x'. Then the value in sym_freq_array at index 'x' is incremented by 1. This done for first 100 bytes
}

process:
for loop (100 to 199)
{ byte is converted to integer. Suppose the value is 'x'. Then the value in sym_freq_array at index 'x' is incremented by 1. This done for second 100 bytes
}

.
.
.
.
.

process:
for loop (1400 to 1499)
{ byte is converted to integer. Suppose the value is 'x'. Then the value in sym_freq_array at index 'x' is incremented by 1. This done for the last 100 bytes
}

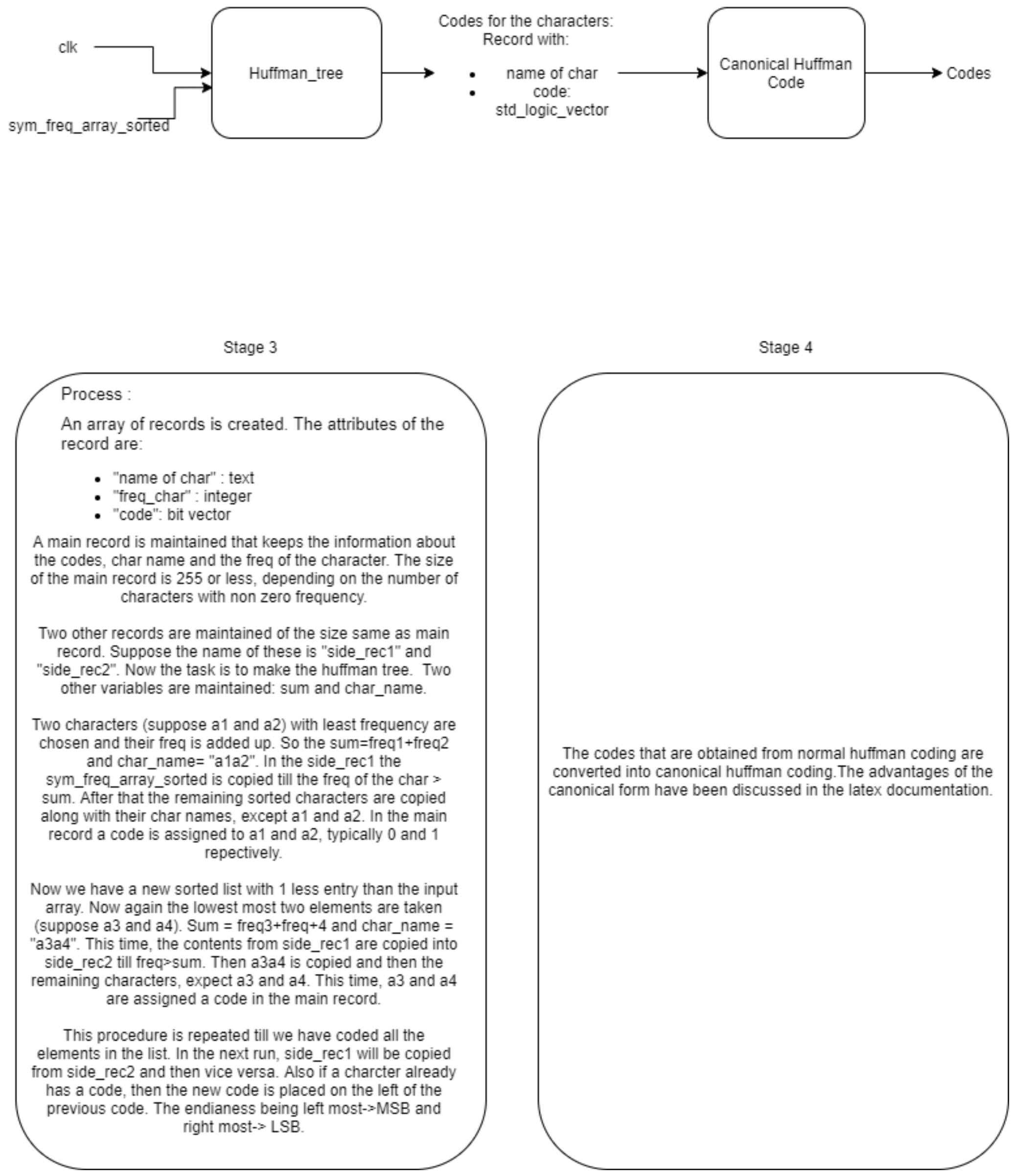
Process:

An array of records is created. The characters with zero freq are removed fromt the list. The attributes of the record are:

- "name of char" : text
- "freq_char" : integer

In FPGA, we have the advantage of parallel processing for different processes. For this very reason, sorting networks are used instead or serial sorting algorithms. Sorting networks are comparison networks that always sort their inputs. The comparison network comprises of wires and comparators only. An optimal sorting network can be defined in terms of latency for a hardware implementation or number of compare-exchange blocks. Sorting networks can be realised usign 3 designs. Combinational logic, synchronous logic and pipelining. If implemented on FPGA, sorting networks can yield results which can sort N terms minimum at a frequency of 500 MHz. But for this, the HDL design needs to be well laid down and implemented.

Figure 3: Flow Chart for hardware implementation of Huffman Encoding



Process :

An array of records is created. The attributes of the record are:

- "name of char" : text
- "freq_char" : integer
- "code": bit vector

A main record is maintained that keeps the information about the codes, char name and the freq of the character. The size of the main record is 255 or less, depending on the number of characters with non zero frequency.

Two other records are maintained of the size same as main record. Suppose the name of these is "side_rec1" and "side_rec2". Now the task is to make the huffman tree. Two other variables are maintained: sum and char_name.

Two characters (suppose a1 and a2) with least frequency are chosen and their freq is added up. So the sum=freq1+freq2 and char_name= "a1a2". In the side_rec1 the sym_freq_array_sorted is copied till the freq of the char > sum. After that the remaining sorted characters are copied along with their char names, except a1 and a2. In the main record a code is assigned to a1 and a2, typically 0 and 1 repectively.

Now we have a new sorted list with 1 less entry than the input array. Now again the lowest most two elements are taken (suppose a3 and a4). Sum = freq3+freq+4 and char_name = "a3a4". This time, the contents from side_rec1 are copied into side_rec2 till freq>sum. Then a3a4 is copied and then the remaining characters, expect a3 and a4. This time, a3 and a4 are assigned a code in the main record.

This procedure is repeated till we have coded all the elements in the list. In the next run, side_rec1 will be copied from side_rec2 and then vice versa. Also if a charcter already has a code, then the new code is placed on the left of the previous code. The endianess being left most->MSB and right most-> LSB.

The codes that are obtained from normal huffman coding are converted into canonical huffman coding.The advantages of the canonical form have been discussed in the latex documentation.

Figure 4: Flow Chart for hardware implementation of Huffman Encoding

3.1 First Stage

In the first stage, instead of reading a stream of data, all of the data is read in one clock cycle since the data is stored in a memory-mapped register. After receiving the data, the std logic vector type data is converted into an unsigned integer type. A frequency calculation procedure is defined. During the procedure call, the values in integer type input are read. The value in ”symbol freq” at index number same as the value in the integer input data is incremented by one. Once the input data is ready in the integer format, the frequency calculation procedure is called in multiple parallel running processes. An array, named ”symbol freq”, of length 255 downto 0 of type integer is maintained. The same procedure is repeated for the complete data. By converting the input data to integer the frequency

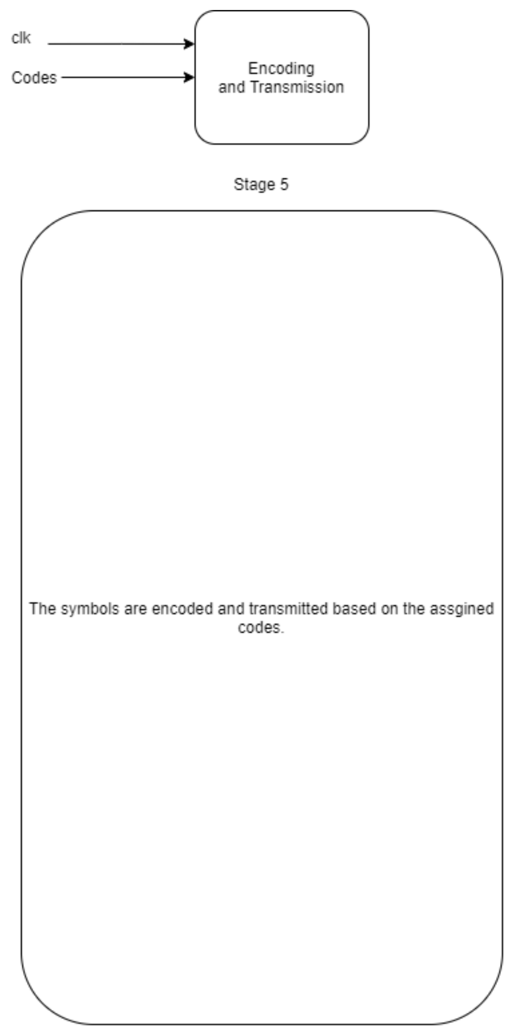
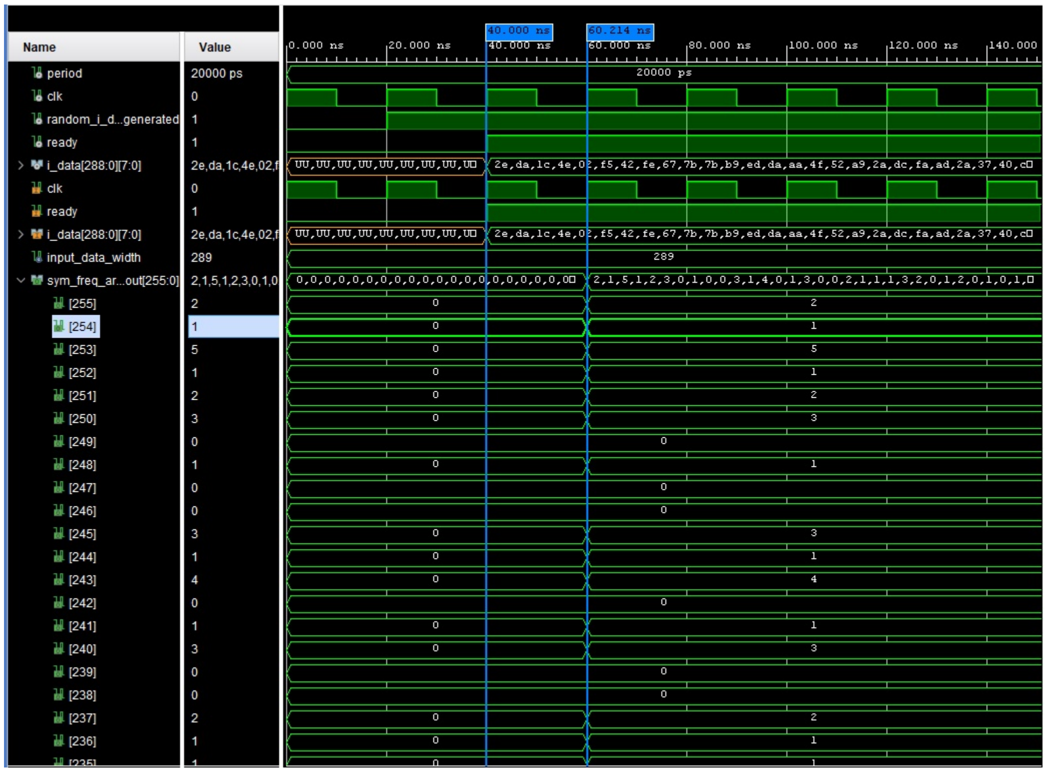

clk
Codes
Encoding and Transmission
Stage 5
The symbols are encoded and transmitted based on the assgined codes.

Figure 5: Flow Chart for hardware implementation of Huffman Encoding

calculation of the symbols becomes easier as the input value actually maps to the index in ”symbol freq” array. At the end of the second clock cycle, we are able to receive the frequency of the symbols in the data.

3.2 Second Stage

In the second stage, the symbols are required to be sorted in descending order by their frequencies. In FPGA, we have the advantage of parallel processing for different processes. For this very reason, sorting networks are used instead or serial sorting algorithms. Sorting networks are comparison networks that always sort their inputs. The comparison network comprises of wires and comparators only. An optimal sorting network can be defined in terms of latency for a hardware implementation or number of compare-exchange blocks. Sorting networks can be realized using 3 designs. Combination logic, synchronous logic and pipe-lining. If implemented on FPGA, sorting networks can yield results which can sort N terms minimum at a frequency of 500 MHz. But for this, the HDL design needs to be well laid down and implemented.[3]



Name	Value
period	20000 ps
clk	0
random_i_d...generated	1
ready	1
i_data[288:0][7:0]	2e,da,1c,4e,02,f
clk	0
ready	1
i_data[288:0][7:0]	2e,da,1c,4e,02,f
input_data_width	289
sym_freq_ar...out[255:0]	2,1,5,1,2,3,0,1,0
[255]	2
[254]	1
[253]	5
[252]	1
[251]	2
[250]	3
[249]	0
[248]	1
[247]	0
[246]	0
[245]	3
[244]	1
[243]	4
[242]	0
[241]	1
[240]	3
[239]	0
[238]	0
[237]	2
[236]	1
[235]	1

Figure 6: Simulation of calculation of frequency of occurrence of symbols

3.2.1 Sorting Network

As discussed, sorting networks comprise of comparators and wires. They differ from general comparison sorting algorithms in the sense that they are not capable of handling input lengths larger than what they are designed for. The sequence of comparisons is set prior and is independent of the result of the comparisons. This makes sorting networks more suitable for hardware implementation as parallel processes can be implemented to execute the sorting network. The efficiency of the sorting network is defined by depth. Depth is informally defined as the highest number of comparators an input can face in the network.

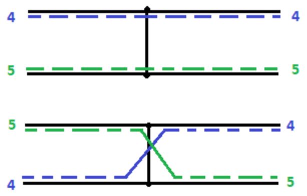


Figure 7: Example of exchange block

In sorting network, wire run from left to right and carry one value at a time. A comparator is shown by vertical line joining two wires. If the wire encounter a comparator, the comparator swaps the values if and only if the top wire's value is greater or equal to the bottom wire's value. Figure 7 block has two inputs and two outputs and the block compares the two inputs and swaps them if they are not in order, all in a single operation, then the results are passed to the outputs. Data moves from left to right and each pair of dots connected with a vertical bar represents a compare-exchange block. We can say that a compare-exchange block is a sorting network of size 2. In hardware terms it consists of a comparator and two 2:1 muxes, where the select input of the two muxes is driven by the output of the comparator. Larger sorting networks can be built out of multiple compare-exchange blocks connected properly. Figure 8 is an example of a sorting network for N=4, it has size 5 and depth 3[3]:

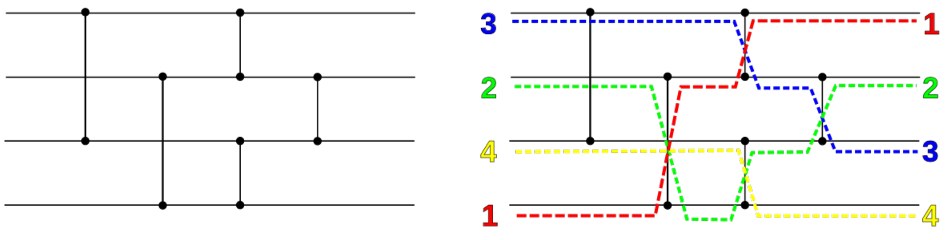


Figure 8: Example of sorting network

An optimal sorting network can be defined in terms of minimal depth, which would translate to latency for a hardware implementation, or number of compare-exchange blocks, which is proportional to design area. In general we will consider size as the determining factor for an optimal sorting network and if there is a tie on size than the one with the smallest depth is the best.

The challenge of designing an optimal sorting network for a given input size N is a very difficult one - no general method for doing that exists and it is almost certain that designing an optimal sorting network is an NP-complete problem, meaning that no polynomial solution actually exists - the complexity of the design problem grows at least exponentially with the size N. We only know the optimal sorting networks for sizes up to 10. For sizes between 11 and 17 we know the minimum depth and bounds for the size, for example we know that for N=16 a solution with 60 compare-exchange exists and that no solution with less than 53 blocks is possible but nobody knows what the actual best solution is.[3]

While implementing sorting network on hardware, we need to see that we will be able to make the network for fixed input N. If the number is less than that then we can do padding and reduce the number of outputs but handling input greater than N is not possible. As far as optimal sorting network is considered, we can implement various algorithms. Bubble sort and insertion sort can be implemented easier than the other algorithms but they order of complexity is $\{O(N^2)\}$. Classic sequential sorting algorithms like Quick Sort are not a good match for parallel hardware implementations. A good choice turns out to be an $\{O(N(\log_2 N)^2)\}$ sorting algorithm called Batcher or odd-even merge sort. For numbers smaller than 100 it gives results close to optimal network and for numbers less than 9 it gives results that are as good as optimal sorting networks.[5]

Even-odd merging networks are built following a recursive definition that is assumed to be efficient when number of elements N is a power of two. The even-odd merging networks are at the heart of the network. Two sorted sequences are merged into a single sorted sequence. The input of the size 2N is divided into sequences of the size N and are sorted using even-odd sorter $E_{OS}(N)$. The sorted outputs are then merged using an even-odd merger $E_{OM}(N)$ of size N. The even-odd sorter is built recursively. An example of E_{OS} has been shown in the figure 9.

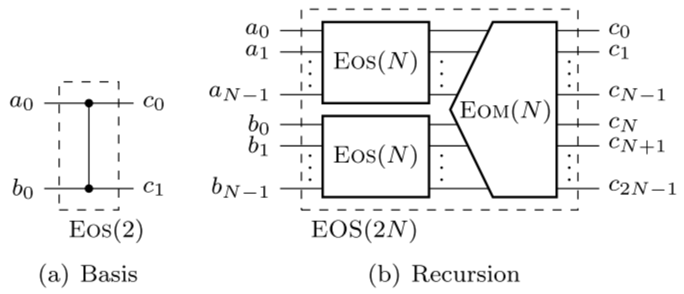


Figure 9: Recursive definition of E_{OS}

Given are two sorted input sequences $\{a_0, a_1, a_2, ...a_{N-1}\}$ and $\{b_0, b_1, b_2, ...b_{N-1}\}$ for an even-odd merger $E_{OM}(2N)$. The N even-indexed elements $\{a_0, a_2, a_4, ...\}$ are mapped to the a-inputs of the "even" merger. The N odd indexed elements $\{a_1, a_3, a_5, ...\}$ are mapped to the a-inputs of the "odd" merger. The b inputs are mapped similarly. As it can be easily shown the inputs of the even and odd mergers are sorted, hence, each produces a sorted sequence at the output. The two sequences are then combined by an array of $2N-1$ comparators.By unrolling the recursion of $E_{OS}(N)$ and $E_{OM}(N)$ a sorting network consisting of comparators is created. An example of E_{OM} has been shown in the figure 10 and an even-odd merging network that is able to sort eight inputs is shown in Figure 11.[5]

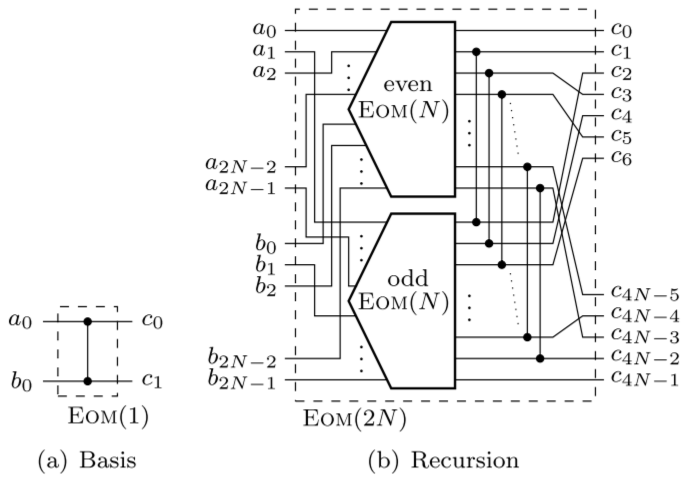


Figure 10: Recursive definition of E_{OM}

4 Conclusion

Stage one of the proposed Huffman Compression algorithm was completed successfully, in which the frequency of occurrence of the 8-bit symbols was calculated within 2 clock cycles of receiving of input data. Further code can be developed for the later stages which have been proposed in this document. As discussed, Huffman Algorithm requires sorting of symbols according to their frequency for every new input data, a fast sorting algorithm becomes a crucial step to reduce the time taken for compression.

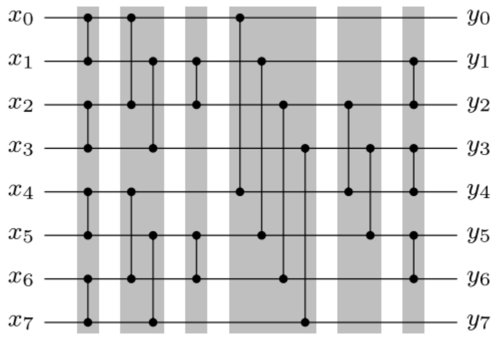


Figure 11: Even-odd merging network for 8 inputs

The order of complexity of sorting an array using bubble sort or selection sort is quadratic so sorting networks is a very promising algorithm to implement a fast compression algorithm.

References

[1] Euccas Chen. Understanding zlib. www.euccas.me/zlib/.

[2] L. Peter Deutsch. Deflate compressed data format specification version 1.3. tools.ietf.org/html/rfc1951#section-7, May 1996.

[3] Element14 FPGAguru. The art of fpga design - post 31. www.element14.com/community/groups/fpga-group/blog/2018/07/11/the-art-of-fpga-design, 2018.

[4] F. Navqi Saud; Naqvi R.; Riaz R.A.; Siddiqui. *Optimized RTL design and implementation of LZW algorithm for high bandwidth applications.* Electrical Review, (April 2011.

[5] Gustavo Alonso Rene Mueller, Jens Teubner. Sorting networks on fpgas, Feb 2012.

[6] Khalid Sayood. *Introduction to Data Compression.* Elsevier, 2006.

[7] Peter Shor. Limpel-ziv notes. www-math.mit.edu/~shor/PAM/lempel_ziv_notes.pdf, October 2005.