FPGA-Based Lossless Data Compression Using GNU Zip

by

Suzanne Rigler

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Lossless data compression algorithms are widely used by data communication systems and data storage systems to reduce the amount of data transferred and stored. GNU Zip (GZIP) [1] is a popular compression utility that delivers reasonable compression ratios without the need for exploiting patented compression algorithms [2, 3]. The compression algorithm in GZIP uses a variation of LZ77 encoding, static Huffman encoding and dynamic Huffman encoding. Given the fact that web traffic accounts for 42% [4] of all internet traffic, the acceleration of algorithms like GZIP could be quite beneficial towards reducing internet traffic. A hardware implementation of the GZIP algorithm could be used to allow CPUs to perform other tasks, thus boosting system performance.

This thesis presents a hardware implementation of GZIP encoder written in VHDL. Unlike previous attempts to design hardware-based encoders [5, 6], the design is compliant with GZIP specification and includes all three of the GZIP compression modes. Files compressed in hardware can be decompressed with the software version of GZIP. The flexibility of the design allows for hardware-based implementations using either FPGAs or ASICs. The design has been prototyped on an Altera DE2 Educational Board. Data is read and stored using an on board SD Card reader implemented in NIOS II processor. The design utilizes 20 610 LEs, 68 913 memory bits, and the on board SRAM, and the SDRAM to implement a fully functional GZIP encoder.

Acknowledgments

All of this could not have be accomplished without the guidance from Dr. Andrew Kennings and Dr. William Bishop. A thank you is also extended to Alexei Borissov, Kris Vorwerk and my family and friends. Specifically my Mom, Ethel Rigler, whose strength and wisdom is an inspiration to me on a daily basis.

For my Dad

Marvin R. Rigler

(1952-2001)

You placed me on this path years ago.

I wish you were here to see me reach the end.

Table of Contents

List of Tables

List of Illustrations

Chapter 1

Introduction

The spread of computing has led to an explosion in the volume of data to be stored on hard disks and sent over the Internet. This growth has led to a need for data compression, that is, the ability to reduce the amount of storage or Internet bandwidth required to handle data. There are lossless and lossy forms of data compression. Lossless data compression is used when the data has to be uncompressed exactly as it was before compression. Text files are stored using lossless techniques, since losing a single character can in the worst case make the text dangerously misleading. Archival storage of master sources for images, video data, and audio data generally needs to be lossless as well. Lossy compression, in contrast, works on the assumption that the data doesn't have to be stored perfectly. Much information can be simply thrown away from images, video data, and audio data, and when uncompressed such media will still be of acceptable quality. Data compression occurs just about everywhere. All the images sent and received on the Internet are compressed, typically in JPEG or GIF formats. Most modems use data compression. HDTV broadcasts are compressed using MPEG-2. Also, several file systems automatically compress files when stored.

To meet the growing demand for compression, Jean-loup Gailly and Mark Adler created the compression utility GNU Zip (GZIP) [1]. The GZIP utility compresses data to create a smaller and less demanding representation for storing and transferring data. The GZIP utility was designed as a replacement for COMPRESS. COMPRESS is a UNIX compression program based on a

variation of LZW [2, 3] where common substrings in a file are replaced by 9-bit codes. Since LZW is a patented algorithm, GZIP became a popular tool that achieves better compression ratios than COMPRESS and it is free from patent issues. The GZIP utility is based on the DEFLATE algorithm [8], which is a lossless data compression algorithm that uses a combination of LZ77 and Huffman coding. Since its release, GZIP has become the preferred tool for file compression and is used in storage systems. More recently, GZIP has become a popular tool to compress and decompress websites. One of the main features of using GZIP on the Internet is speed. It has been reported by the website USWeb that using GZIP to compress its website has shown a decrease by up to 40% in page load time [9]. Another benefit of GZIP is potentially a better relationship with search engine crawlers. For example, Google can read a GZIP file much more quickly than crawling a site manually. When Google is updating its search queries, it attempts to not monopolize internet traffic. If a website is slow, the spider may believe it is taxing the web servers resources and visit the website less often. Thus, changes made to the website less often. The GZIP compression tool has become a standardized part of the HTTP protocol and most web browsers have built-in GZIP decompression software. The HTTP/1.1 protocol allows for clients to optionally request the compression of content from the server [10]. The standard itself specifies two compression methods: GZIP and DEFLATE. Both are supported by many HTTP client libraries and almost all modern browsers.

1.1 Thesis Contributions and Motivations

This thesis presents the design and description of a hardware implementation for GZIP compression written entirely in VHDL. The specific contributions of this thesis include:

- A hardware implementation of GZIP that conforms to the GZIP specification (i.e., files compressed in hardware can be decompressed in software by standard software implementations of GZIP);
- A hardware implementation which, unlike other hardware implementations, supports all

three of the GZIP compression modes; and

- A fully functional prototype of the hardware implementation implemented on an Altera DE2 prototype board.

The motivation for a hardware implementation of GZIP should be clear from the previous description of the uses of GZIP. A hardware implementation of GZIP can offer the possibility of high-speed compression and offload bandwidth consuming compression tasks from a CPU, thereby freeing up valuable CPU resources. A specific hardware implementation of GZIP on a single chip (e.g., a low cost FPGA) reduces the need for a fully-functional PC to perform compression tasks. Hence, applications of a hardware implementation of GZIP would include deployment inside of storage area networks, web servers and other Internet-related appliances such as load balancers, firewall VPN servers and integrated routers and switches.

1.2 Thesis Organization

Chapter 2 outlines the necessary background information for the reader to understand the GZIP algorithm. The compression algorithms covered include: GZIP, LZ77 and Huffman. Chapter 3 discusses a hardware implementation of GZIP compression using a top-down approach. Included in this design are block instantiations of Huffman encoders and an LZ77 encoder. These blocks are described in detail in Chapter 4 and Chapter 5 respectively. Chapter 6 provides a detailed description of the FPGA prototype and the interaction necessary to test the GZIP implementation. Chapter 7 analyzes the experiments run and compares the results against a software version of GZIP. Chapter 8 provides a few closing remarks.

Chapter 2

Background

This Chapter outlines the necessary background information for the reader to understand the GZIP algorithm. Since the GZIP algorithm utilizes a LZ77 encoder and several Huffman encoders, these algorithms will be discussed in detail. Section 2.5 evaluates existing hardware implementations of GZIP and Section 2.6 provides a few general comments about this document.

2.1 GZIP Algorithm

A GZIP file consists of a series of data members, where each member simply appear one after another in a file. The data members include header information, compressed blocks, and end-of-file information.

2.1.1 File Structure

The beginning of each file compressed by GZIP contains a ten byte header with optional fields present. The header contains information to specify that the file follows GZIP format and the state of the computer at the time of compression. The formal representation of the GZIP file format is presented in Figure 2.1. The first two bytes (ID1 and ID2) are unsigned identification numbers that are used by GZIP to identify the file as being in GZIP format; they are the values 31 and 139. The next byte (CM) is the compression method used by GZIP, the customary method chosen is

the DEFLATE algorithm which is represented by the value 8. The fourth byte (FLG) contains a flag that expresses some of the possible options available in GZIP. If the first bit of the flag is set, the file is probably ASCII text. This is an optional indicator, which the compressor may set by checking a small amount of the input data to see whether any non-ASCII characters are present. If a non-ASCII character is found, the bit is cleared, thus indicating binary data. If the second bit is set, a cyclic redundancy check for the GZIP header is present immediately before the compressed data. The third bit of the flag is used to indicate if any extra fields are present. If fields are present, they immediately follow the header information. If the fourth bit is set, an original file name is present terminated by a zero byte following any extra fields. The fifth bit determines whether a zero-terminated file comment is present. This comment is not interpreted; it is only present for human information. The remaining bits of the flag are reserved for the GZIP algorithm. Following the flag, the next four bytes (TIME) of the GZIP header represent the most recent modification time of the original file being compressed. The ninth byte (XFL) in the header is used to determine the specific compression method used by DEFLATE. The DEFLATE algorithm can be executed in two different methods. Using maximum compression and a longer runtime the algorithm assigns the ninth byte the value 2. Using less than maximum compression and shorter runtime, the algorithm assigns the ninth byte the value 4. Finally, the last byte (OS) of the GZIP header represents the operating system used for compression. The GZIP header is followed by blocks of compressed data, terminated by an end-of-block character. Following the compressed data, GZIP writes eight bytes of data used for decompression. The first four bytes (CRC32) are a cyclic redundancy check value for the compressed data and the last four bytes (ISIZE) contain the size of the original uncompressed data modulo 2^{32}.

2.1.2 Block Structure

Following the header information a file compressed by GZIP consists of a series of blocks, corresponding to successive blocks of input data. The amount of data processed for compression can vary as long as it is smaller than 32 768 bytes. GZIP determines the amount of data to process

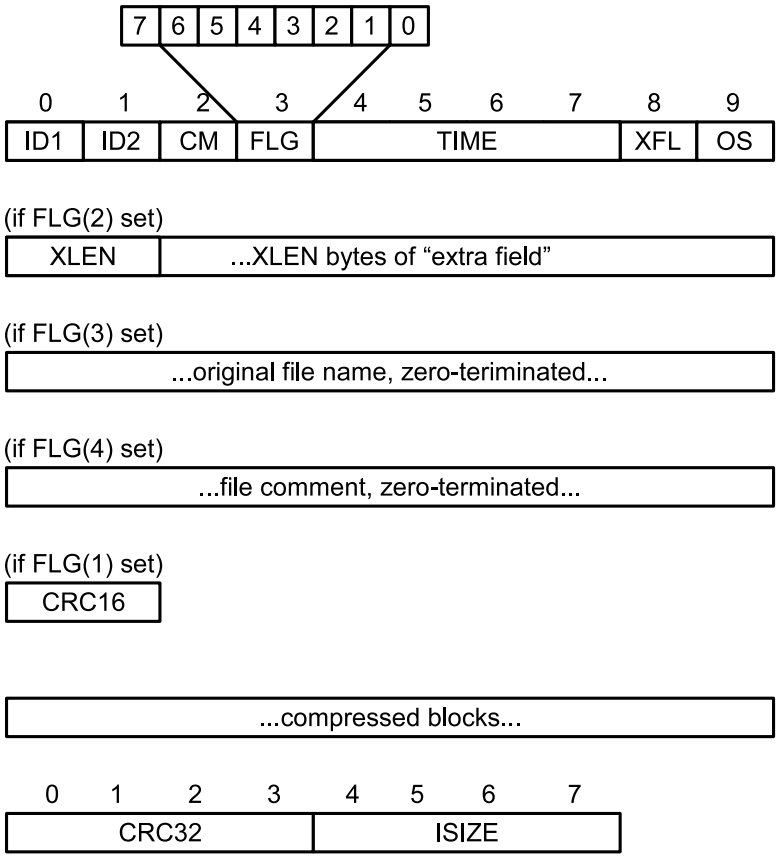


Figure 2.1: GNU Zip (GZIP) File Structure

based on a heuristic function with evaluates when it is beneficial to begin a new block. The header of each block contains one bit to indicate if this is the last block to be processed and is followed by two bits describing the compression mode used. The GZIP algorithm may compress each block in three different modes: stored, fixed and dynamic. In the stored mode, the data is not compressed; therefore the block header is simply followed by the original uncompressed data. In either the fixed or dynamic mode, each block is compressed using a combination of the LZ77 and Huffman coding algorithms. The input data is first encoded using the LZ77 algorithm and the output of the LZ77 algorithm is then encoded using the Huffman algorithm. The compressed information from the LZ77 algorithm consists of literal bytes and distance-length pairs to previous strings. Both the literals and distance-length pairs are represented using a Huffman tree, using one tree for literals and lengths and the other tree for distances. In the fixed mode, the frequency of the characters are defined in advance, so the necessary Huffman trees are stored in look-up tables and the output of LZ77 algorithm is simply encoded with values from the look-up tables. However, in the dynamic mode the Huffman codes are generated based on the actual frequencies of the input data and must be computed for each block. This requires a representation of the dynamic Huffman trees to follow the block header since the Huffman trees for each block are independent of those from previous or subsequent blocks. The compressed data encoded using the dynamic Huffman trees follows the representation of the dynamic Huffman trees in the output file. Regardless of the compression mode used, each block is terminated by an end-of-block marker.

Initially the block of data to be compressed is processed by the LZ77 coding algorithm which produces flags, literals, match distances and match lengths. The literal bytes from the alphabet $\{0,\ldots,255\}$ and the match lengths from the alphabet $\{3,\ldots,258\}$ are merged into a single alphabet $\{0,\ldots,285\}$ where values $0,\ldots,255$ represent literal bytes, the value 256 indicates the end-of-block, and values $257,\ldots,285$ represent match lengths. Similarly, the match distances from the alphabet $\{1,\ldots,32\,768\}$ are mapped into the alphabet $\{0,\ldots,29\}$. The alphabet representations used are presented in Table 2.1 and Table 2.2. In both of the alphabet mappings, extra bits are often required to be able to extrapolate the original value during decompression. As mentioned above,

the literals and match lengths $\{0,\ldots,285\}$ are encoded by one Huffman tree and the match distances $\{0,\ldots,29\}$ are encoded with a separate Huffman tree. Once the two dynamic Huffman trees have been created, GZIP determines whether compressing the block of data with dynamic Huffman trees or static Huffman trees will produce a higher compression ratio. The static Huffman trees are stored in a look-up table in both the compressor and decompresser and are presented in Table 2.3 and Table 2.4. If dynamic Huffman compression is beneficial then a representation of the dynamic literal-length Huffman tree (DLLHT) and the dynamic distance Huffman tree (DDHT) must occur at the beginning of the block to be able to reconstruct the Huffman trees for decompression purposes. Due to restrictions placed on the Huffman algorithm, further discussed in Section 2.3, the Huffman trees are guaranteed to be unique, requiring only the code lengths to be sent and not the code for each value. The Huffman algorithm also places a restriction on the code length for each value allowing a maximum of 15 bits. This ensures the code lengths of DLLHT and DDHT are in the alphabet $\{0,\ldots,15\}$. This allows a third dynamic Huffman tree (DCLHT) to be created with alphabet $\{0,\ldots,18\}$ to compress the output of DLLHT and DDHT tree. The values 0,...,15 are the code lengths and the values 16, 17 and 18 are used for special repeating values which are presented in Table 2.5. The precise output format of the dynamic Huffman compression block is presented in Figure 2.2. If a static Huffman tree was used, it is not necessary to output any trees since the decompresser has access to the static codes. Once the necessary Huffman trees have been written to the file the newly compressed information follows using either the static or dynamic Huffman representation, followed by an end-of-block marker.

The last block processed is followed by eight bytes of information required for GZIP decompression (see Figure 2.1). The first four bytes (CRC32) contain a cyclic-redundancy check for the compressed data and the last four bytes (ISIZE) are the number of bytes compressed modulo 2^{32}.

2.2 Ziv-Lempel Coding Algorithm

Ziv-Lempel (LZ) is a generic compression algorithm utilizing regularities in a bit stream [11]. The LZ algorithm is a lossless dictionary based scheme, allowing the original information to be

Table 2.1: Dynamic Literal-Length Huffman Tree (DLLHT) Alphabet

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

Table 2.2: Dynamic Distance Huffman Tree (DDHT) Alphabet

Distance	Extra Bits	Code	Distance	Extra Bits	Code	Distance	Extra Bits	Code
1	0	0	33-48	4	10	1 025-1 536	9	20
2	0	1	49-64	4	11	1 537-2 048	9	21
3	0	2	65-96	5	12	2 049-3 072	10	22
4	0	3	97-128	5	13	3 073-4 096	10	23
5-6	1	4	129-192	6	14	4 097-6 144	11	24
7-8	1	5	193-256	6	15	6 145-8 192	11	25
9-12	2	6	257-384	7	16	8 193-12 288	12	26
13-16	2	7	385-512	7	17	12 288-16 384	12	27
17-24	3	8	513-768	8	18	16 385-24 576	13	28
25-32	3	9	769-1024	8	19	24 577-32 768	13	29

Table 2.3: Static Literal-Length Huffman Tree (SLLHT)

Literal Value	Code Length	Code
0-143	8	48 through 191
144-255	9	400 through 511
256-279	7	0 through 23
280-287	8	192 through 199

Table 2.4: Static Distance Huffman Tree (SDHT)

Value	Code Length	Code	Value	Code Length	Code	Value	Code Length	Code
0	5	0	10	5	10	20	5	20
1	5	1	11	5	11	21	5	21
2	5	2	12	5	12	22	5	22
3	5	3	13	5	13	23	5	23
4	5	4	14	5	14	24	5	24
5	5	5	15	5	15	25	5	25
6	5	6	16	5	16	26	5	26
7	5	7	17	5	17	27	5	27
8	5	8	18	5	18	28	5	28
9	5	9	19	5	19	29	5	29

Table 2.5: Dynamic Compressed-Length Huffman Tree (DCLHT) Alphabet

Literal Value	Extra Bits	Code
0-15	0	0-15
16	2	Copy the previous code 3-6 times
17	3	Repeat a code length 0 for 3-10 times
18	7	Repeat a code length 0 for 11-138 times

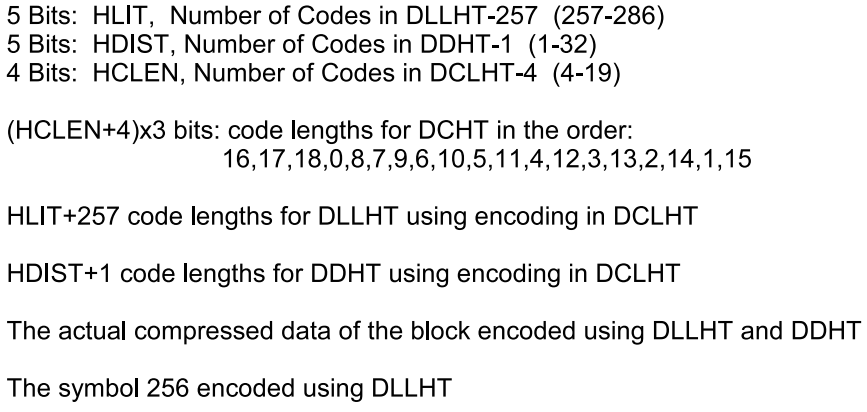


Figure 2.2: Dynamic Compressed Block Format

reconstructed from compressed data. There are multiple versions of LZ compression; LZ77, LZ78 and LZW being the most common. LZ78 and LZW both generate better compression over a finite bit stream compared to LZ77 [12]. However, LZ78 and LZW both utilize static dictionaries. For this type of design, a look-up table holding the recurring symbols is required. Using a look-up table to decompress data would result in higher hardware requirements for the LZ78 and LZW algorithms. On the other hand, LZ77 utilizes a dynamic dictionary and, as a result, has a smaller impact on the memory required for decompression. Since LZ78 and LZW are not used in GZIP, they will not be described any further.

LZ77 processes data from left to right, inserting every string into a dictionary. Quite often the dictionary is limited by memory, thus a sliding dictionary is used. A sliding dictionary keeps track of the most recent strings seen, discarding any previous strings. If a string does not occur anywhere in the dictionary, then it is emitted as a literal sequence of bytes. If a match is found, then the duplicate string is replaced by a pointer to the previous string in the form of distance-length pair. The distance-length pair is composed of two parts, the first being the distance from the current element to the element in which the match starts, and the second is the length of the match (see Figure 2.3). With respect to the LZ77 used by GZIP, the newly compressed information is also accompanied by a flag byte which precedes the data allowing the GZIP algorithm to be able to distinguish literals and distance-length pairs. In the standard LZ77 algorithm the flag is emitted individually as a bit, but in GZIP the LZSS variation is used. LZSS is a derivative of LZ77 that allows 8 flag bits to be grouped together into one byte [13]. To achieve optimum compression it is important that the closest match in the dictionary that does not sacrifice match length is found. This is key because matches that are nearest are encoded in the fewest number of bits.

Although the LZ77 algorithm implemented with a sliding dictionary can be run in linear time [14], it is slow and consumes large amounts of memory. This can be solved by using a hashing algorithm, which increases speed and reduces memory requirements. GZIP and almost all programs using the LZ77 scheme use hashing data structures [15].

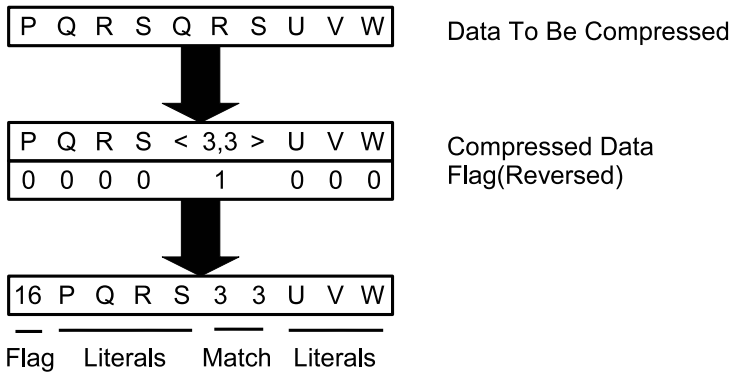


Figure 2.3: Ziv-Lempel (LZ77) Encoding Example

2.3 Huffman Coding Algorithm

Huffman coding is a lossless statistical data compression scheme [16]. The algorithm is based on the concept of mapping an alphabet to a different representation composed of strings of variable size, such that symbols that have a higher probability of occurring have a smaller representation than those that occur less often.

There are two methods of Huffman coding: static and dynamic. In the static method the frequencies of each character in the alphabet are assigned before the program begins and are stored in a look-up table. In the dynamic method, on the other hand, must make one pass through the text to determine the frequency of each character. Once the histogram has either been calculated or provided, the two algorithms are identical. Elements are selected two at a time, based on frequency; lowest frequency elements are chosen. The two elements are made to be leaf nodes of a node with two branches. The frequencies of the two elements selected are then added together and this value becomes the frequency for the new node (see Figure 2.4). The algorithm continues selecting two elements at a time until a Huffman tree is complete with the root node having a frequency of 100.

The classic Huffman coding algorithm is nondeterministic, thus allowing a data set to be represented as more than one possible tree. The GZIP program applies two additional rules to ensure that each data set has at most one possible tree representation. Elements that have shorter codes are placed to the left in the Huffman tree of those with longer codes. In Figure 2.4, D and

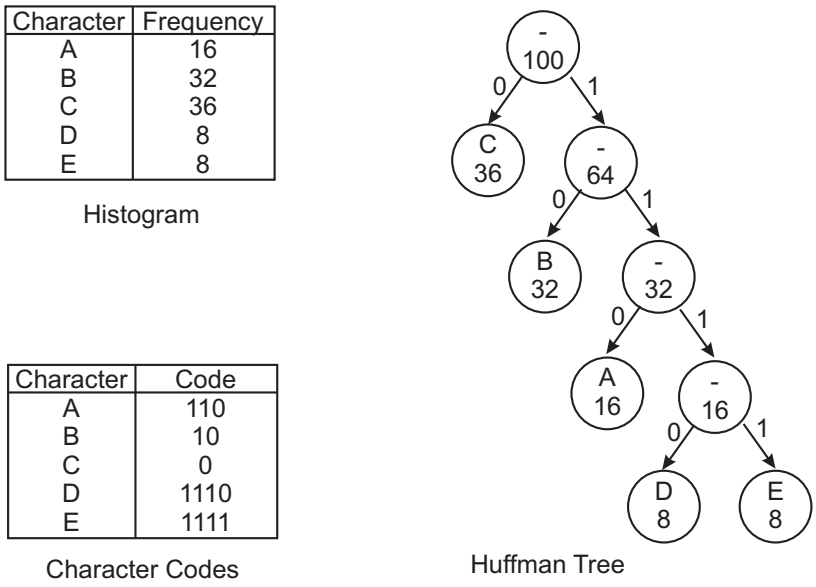


Figure 2.4: Huffman Encoding Example

E end up with the longest codes, so they would be in the right branch of the tree. Also, among elements with codes of the same length, those that come first in the element set are placed to the left in the Huffman tree. Since D and E are the only elements with code length four and D comes first in the data set, D will be assigned the 0 branch and E will be assigned the 1 branch. When these two restrictions are placed on the Huffman coding algorithm, there is at most one possible tree for every set of elements and their respective code lengths. This fact allows for a substantial optimization to be made in the Huffman coding algorithm. Since each Huffman tree is deterministic and can be recomputed using the code lengths, it is not necessary to send each code word for decompression purposes.

2.4 GZIP Example

An example of GZIP encoding is illustrated in Figures 2.5-2.8. The original sentence and the output of the LZ77 encoder is presented in Figure 2.5. The elements in black text are literals, the red elements are the flags, the blue elements are the match lengths and the green elements are the

match distances. Each element is then sent to the appropriate Huffman tree to create the dynamic Huffman codes. The literals (black) and match lengths (blue) are encoded in one tree (DLLHT) in Figure 2.6. Also, the match distances (green) are encoded in another Huffman tree (DDHT) in Figure 2.7. In both cases the static representations that already existed in GZIP are also presented in each Figure. Once DLLHT and DDHT have been computed, DCLHT can be computed using the code lengths from the previous Huffman trees. After the DCLHT has been computed (see Figure 2.8), the GZIP algorithm determines with method of compression will achieve the highest compression ratio. In this case, the stored length would be 504 bits, the fixed length would be 425 bits and the dynamic length would be 433 bits. So the block would then be compressed using the fixed mode, which uses the static Huffman codes.

2.5 Recent Advances

Despite the importance of the GZIP algorithm, relatively few complete hardware implementations exist. For example, [5] presents a GZIP encoder and decoder which implements only the static Huffman encoding specified by the GZIP standard. The implementation was written in VHDL and tested on an FPGA. In comparison, this thesis presents a GZIP implementation that includes all three of the GZIP compression modes. It is also written in VHDL and tested on an FPGA. By implementing all three compression modes, the implementation described in this thesis can achieve better overall compression ratios. In [6], a GZIP encoder is implemented using a CAM

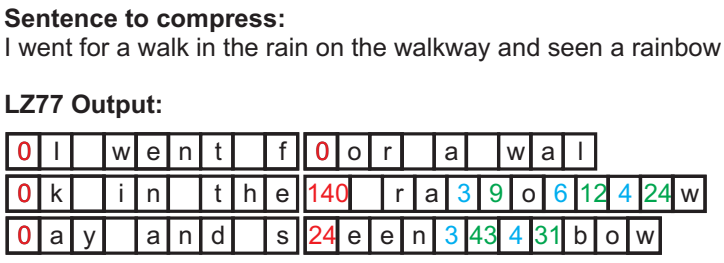


Figure 2.5: GZIP Encoding Example (Part 1)

Literal-Length Huffman Tree:

Character	Dynamic Code	Static Code
	“00”	“01010000”
l	“10100”	“01111001”
a	“010”	“10010001”
b	“111000”	“10010010”
d	“10101”	“10010100”
e	“0110”	“10010101”
f	“111001”	“10010110”
h	“111010”	“10011000”
i	“111011”	“10011001”
k	“111100”	“10011011”
l	“10110”	“10011100”
n	“0111”	“10011110”
o	“1000”	“10011111”
r	“10111”	“10100010”
s	“11000”	“10100011”
t	“11001”	“10100100”
w	“1001”	“10100111”
y	“111101”	“10101001”
256	“111110”	“0000000”
257(3)	“11010”	“0000001”
258(4)	“11011”	“0000010”
260(6)	“111111”	“0000100”

Figure 2.6: GZIP Encoding Example (Part 2)

Distance Huffman Tree:

Distance	Dynamic Code	Static Code
6(9,12)	“00”	“0110”
8(24)	“01”	“01000”
9(31)	“10”	“01001”
10(43)	“11”	“01010”

Figure 2.7: GZIP Encoding Example (Part 3)

Dynamic Compressed-Length Huffman Tree:

Symbol	Dynamic Code
0	“00”
2	“100”
3	“11110”
4	“1111”
5	“01”
6	“111”
17	“11111”
18	“110”

Figure 2.8: GZIP Encoding Example (Part 4)

(Content Addressable Memory). Consequently, the design is expensive in terms of its hardware requirements and is not necessarily portable to other architectures. The implementation presented in this thesis does not use architecturally specific blocks like CAM and is therefore more easily ported to other architectures. Although not necessarily described in the context of GZIP, additional prior work on hardware-based implementations of LZ77 and Huffman encoding can be found.

Dictionary based implementations (LZ77) fall into one of three categories: the microprocessor approach [17], the CAM approach [18] and the systolic array approach [19]. The first approach does not explore full parallelism and is not attractive for real time applications. The CAM approach is able to achieve high throughput by exploiting parallelism and can outperform C-programs running on CPU's [20]. However one drawback of the CAM approach is that it is costly in terms of hardware requirements. The systolic array approach, as compared to the CAM approach, is not as fast but has lower hardware requirements and has improved testability. These three approaches are architecture specific and cannot be easily ported to different architectures. The proposed state machine implementation in VHDL is architecture independent and can be easily transferred to other architectures. The flexibility of the design allows for hardware-based implementations on FPGA boards and inside ASICs.

Many different groups have investigated implementations of static Huffman coders [21–25]. As mentioned above, a static coder assumes that the signal distribution is known when the algorithm begins. This type of assumption can cause negative side effects. If the actual signal distribution is not close to the assumed distribution, coding overheads can occur. Dynamic Huffman implementations have also been implemented [26, 27]. Since these implementations do not include the two additional rules GZIP uses to ensure the Huffman tree is deterministic, they require the entire tree to be sent to the decompresser rather than just the code lengths. This increases the overall runtime and compression ratio and is not a feasible option.

2.6 General Comments

Since most data compression techniques can work on different types of digital data, such as characters or bytes in image files, data compression literature speaks in general terms of compressing symbols. Many of the examples in this document refer to compressing characters, simply because a text file is very familiar to most readers. However, in general, compression algorithms are not restricted to compressing text files. Hence, throughout the remainder of this thesis, the terms symbol and character are used interchangeably. Similarly, most of the examples talk about compressing data in files, just because most readers are familiar with that idea. However, in practice, data compression applies just as much to data transmitted over a modem or other data communications link as it does to data stored in a file. There's no strong distinction between the two as far as data compression is concerned.

2.7 Summary

This Chapter has discussed the relevant background information regarding the GZIP algorithm. The LZ77 and Huffman algorithms have been described in detail because they play a significant role in GZIP compression. Despite the importance of GZIP, only a few prior attempts have been made to implement the algorithm in hardware. These prior implementations have been (1) architecture dependent, (2) expensive in terms of their hardware requirements, and/or (3) incomplete in so far as they did not implement all the compression modes specified by GZIP. These issues serve to make the implementation described in the remainder of this thesis interesting.

Chapter 3

GZIP Hardware Implementation

This Chapter begins a top-down description of the GZIP hardware implementation. Section 3.1 provides a block-level overview of the GZIP encoder and Section 3.2 gives details regarding the internal structure of the GZIP encoder itself. Section 3.3 gives an in depth explanation of the GZIP hardware implementation. The implementation requires block instantiations of LZ77 and Huffman encoders. These blocks are described in detail in Chapter 4 and Chapter 5 respectively.

3.1 Block Structure

The block structure for the GZIP encoder is illustrated in Figure 3.1. The `clock` signal is self explanatory. The `reset` is an active low input signal used to reset the entire circuit. The signals `inputSend`, `inputValid`, `input` and `eof` are used to load the data to be compressed into the encoder. The clock cycle after `inputSend` is equal to one, the signal `input` is assigned the new value from the input source which raises `inputValid` to one indicating new data is available. If the end-of-file had been reached, the signal `eof` is set to one. Similarly for output, when `outputValid` is one, the value at `output` is written to the output file. Once the algorithm is complete, the signal `outputDone` is assigned one, indicating that we are done with the file and it may be closed. The signals `sram_address`, `sram_data`, `sram_we` and `sram_q` are used to read and write to a RAM.

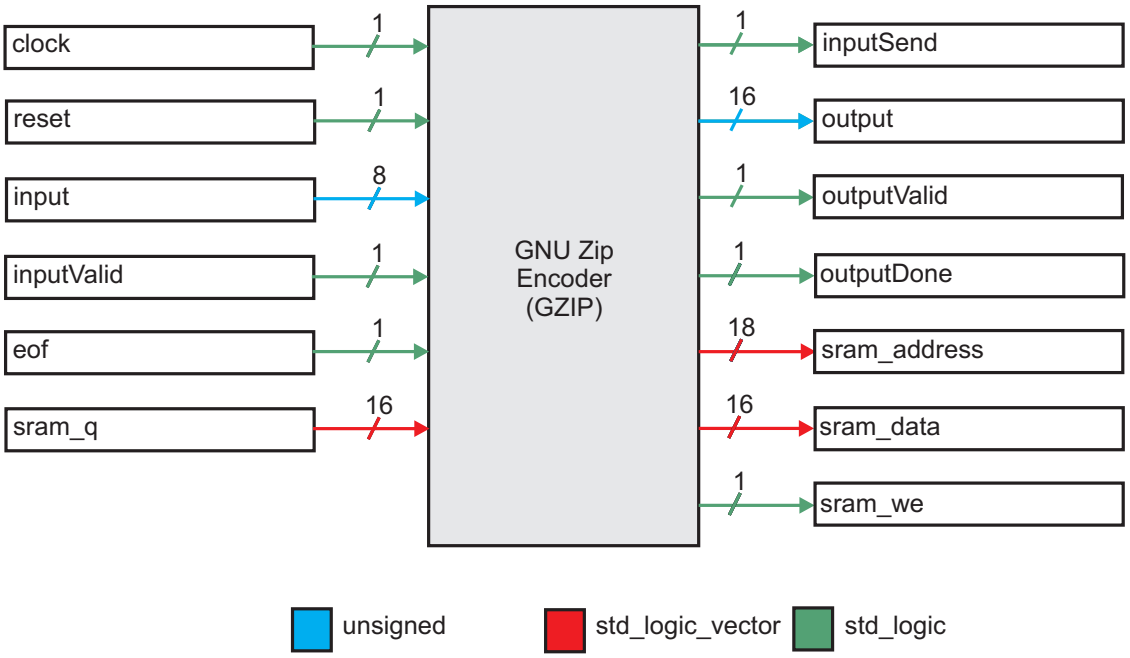


Figure 3.1: GNU Zip (GZIP) Encoder Block

3.2 Major Components

Based on the description of GZIP in Chapter 3, the majority of the GZIP hardware implementation is based on the instantiation of a number of different blocks and the interaction between them. This requires four block instantiations, one LZ77 encoder and three dynamic Huffman encoders. The design also requires two static Huffman encoders, static literal-length Huffman tree (SLLHT) and static distance Huffman tree (SDHT). The overall block interaction can be seen in Figure 3.2.

The GZIP encoder holds the representation of the static Huffman trees used. SLLHT is stored in two RAMs; `SLLHTCodes` and `SLLHTLengths` of size 286×9 bits and 286×4 bits respectively. The `SLLHTLengths` RAM is a dual-port RAM which allowing sequential reads from the Huffman encoder block. Similarly, SDHT is stored in one RAM called `SDHTCodes` of size 30×5 bits. Only one RAM is required because all codes in the SDHT have a length of 5.

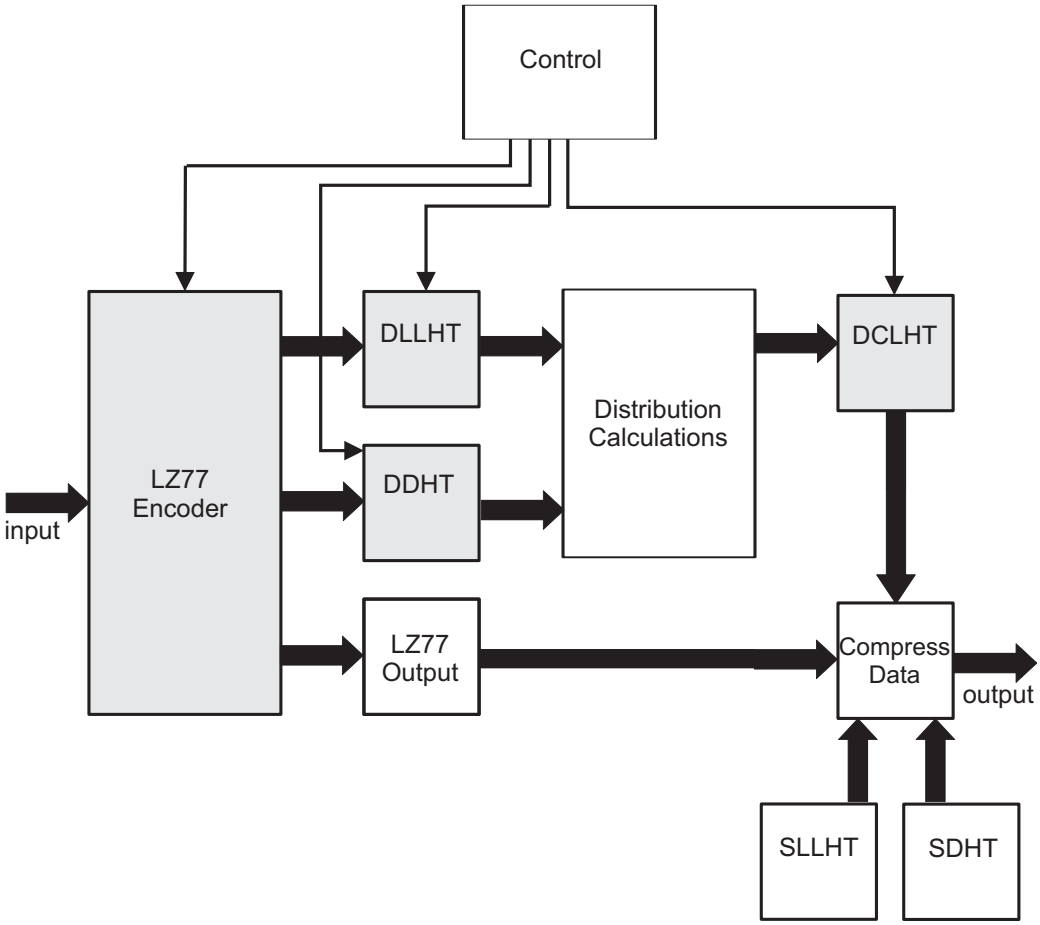


Figure 3.2: GNU Zip (GZIP) Internal Structure

3.3 Implementation

The GZIP algorithm essentially has three key responsibilities. The first is to control the data flow throughout the algorithm. This includes control signals for the LZ77 encoder and three dynamic Huffman encoders. The second and third parts are two distinct sets of calculations. The first set of calculations computes the distribution for DCLHT and the second chooses the data compression method and outputs the compressed data. The details of these three important parts are discussed in detail below.

3.3.1 Data Flow and Control Signals

This Section provides a description of the flow of data through the GZIP encoder. The GZIP compression hardware begins by waiting for a `reset` signal from the user. Once a `reset` is received, the Huffman encoders and the LZ77 encoder are also reset. While the four internal blocks are initializing, the ten bytes of header information are written to the output file. A function `sendBits` has been created to send data to a file. It takes as parameters the value to send and the number of bits to send. In the `sendBits` function if two bytes of data are ready to be written to the output file the value is assigned to `output` and `outputValid` is assigned one. To send the ten header bytes, five states are required calling `sendBits` in each state. The GZIP encoder then continues to sit in a wait state until the LZ77 encoder is complete and DLLHT and DDHT have received all their necessary data and can begin construction of the Huffman trees. The GZIP encoder then enters another wait state until DLLHT is done calculating its code lengths. Now that the code lengths for DLLHT have been computed, an iteration must be performed on each code length to build the distribution for DCLHT. As previously mentioned, DCLHT is used to compress the output of the dynamic Huffman trees by taking advantage of repeating code lengths (see Table 2.5). This distribution calculation is discussed in detail in Section 3.3.2. Once the DLLHT has been processed the algorithm must wait until the DDHT code lengths have been calculated and then repeats the set of calculations. Once DLLHT and DDHT have been processed, the algorithm waits until the DCLHT code lengths have been calculated. After all the DCLHT

code lengths have been computed the algorithm enters another set of calculations to determine the compression mode for the block and then writes the newly compressed information back to the file (see Section 3.3.3).

If the input file is longer than 32 768 characters then the algorithm will need to process more than one block of data. This occurs by resetting the LZ77 encoder and the three Huffman encoders and looping the GZIP algorithm to the wait state after the ten bytes of header information has been written to the output file. Otherwise, if the end-of-file as occurred then the GZIP algorithm must write the last eight bytes of the GZIP header information. If the output file is not on a byte marker, zero bits are emitted until the byte is complete. In the next four clock cycles, the cyclic redundancy check is written in four bytes and the original file input size is written in four bytes. One thing to note about the implementation is that the algorithm only processes block sizes of 32 768 characters where the software version of GZIP processes varying sized blocks.

3.3.2 Distribution Calculation

To ensure that the dynamic Huffman tree representation is as small as possible, the code lengths for DLLHT and DDHT are compressed using a third Huffman tree, DCLHT. This set of calculations compute the necessary input for DCLHT. Since both DLLHT and DDHT are processed in the same manner, the description provided will be in terms of a general Huffman tree. The description provided expands on what occurs in the Distributions Calculations box in Figure 3.2 which is further expanded in the flow diagram in Figure 3.3 and Figure 3.4. The distribution calculations begins by addressing the Huffman trees `lengths` RAM, which contains each code length, at zero in the first state. In state three, variables `prevLen` and `count` are assigned zero. One clock cycle later the RAM access is complete and in the fourth state, a variable `nextLen` is assigned `lengths[0]`. If `lengths[0]` does not equal zero `maxCount=7` and `minCount=4`, otherwise `maxCount=138` and `minCount=3`. The variables `maxCount` and `minCount` are used throughout to determine the appropriate signal to send to DCLHT in accordance with the description in Table 2.5.

In state five the main loop of the distribution calculation is entered and if `j`, which is initially

zero, is less than or equal to the signal `maxCode`, then `lengths` is addressed at `j+1`. The signal `maxCode` is used by the Huffman tree to indicate the highest code processed on input as to avoid processing extra symbols that did not occur in the Huffman tree. Also in state five, `curLen` is assigned `nextLen` and `count` is incremented. Two clock cycles are then waited for the `lengths` read and in state eight `nextLen` is assigned `lengths[j+1]`.

State nine increments `j`, and considers the following five different conditions in the specified order.

1. (`count` $<$ `maxCount`) and (`curLen` $=$ `nextLen`)
2. (`count` $<$ `minCount`)
3. (`curLen` $\neq$ `0`)
4. (`count` ≤ 10)
5. Otherwise

If condition one is satisfied, the algorithm loops back to state five, indicating the same length was processed as seen in the previous iteration. Otherwise, if condition two is satisfied, indicating there is not enough repeating code lengths to bother compressing, `curLen` is sent to DCLHT `count` times. This involves a loop of four states since the Huffman tree can only process data every four clock cycles. If the third condition is true, the value 16 is sent to Huffman tree. A value of 16 is used to indicate that the previous code has been repeated three to six times. If `curLen` does not equal zero and `curLen` does not equal `prevLen`, then `curLen` must also be sent, requiring a four cycle wait once again. In either case, in state ten `count` is assigned zero and `prevLen` is assigned `curLen`. Also if `nextLen` equals zero then `maxCount=138` and `minCount=3`, otherwise if `curLen` equals `nextLen`, `maxCount=6` and `minCount=3` . If neither of those conditions hold true, `maxCount=7` and `minCount=4`. The algorithm then loops back to state five. If the fourth condition is satisfied, a 17 is sent to DCLHT and the algorithm proceeds to state ten and then loops back to state five. A 17 implies that a code length of zero has been repeated three to ten times. Finally, if the previous four conditions were not satisfied, an 18 is sent to the Huffman tree, the algorithm proceeds to state ten and then loops back to state five. An 18 implies that a code length of zero has

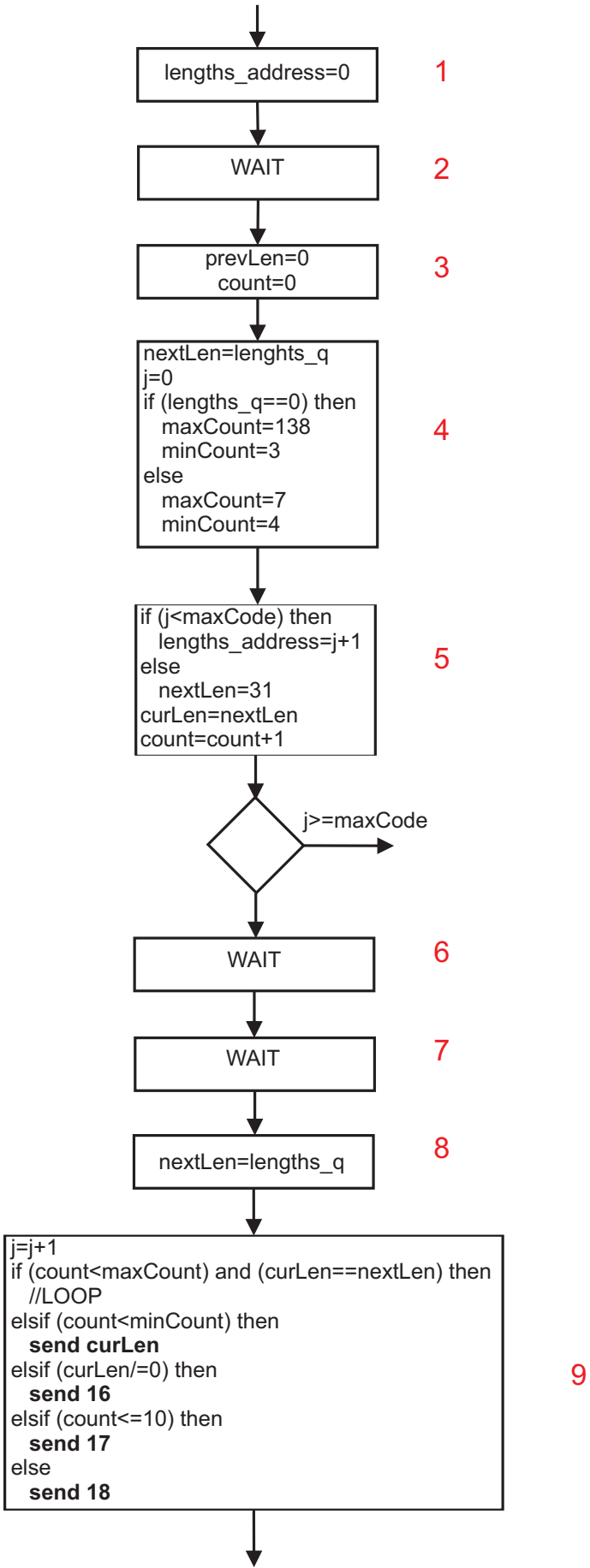
been repeated 11 to 138 times.

In state five, if `j` is greater than or equal to `maxCode` then the distribution is complete for this specific Huffman tree.

3.3.3 Compress and Output Data

Once the DCLHT code lengths have been determined, the GZIP algorithm must determine the compression mode and compress the data. First, the GZIP algorithm iterates through each DCLHT code length to determine how many compressed-lengths codes must be sent. When the DCLHT representation is written to the output file the code lengths are sent in a specific order to take advantage of the fact that some code lengths will occur less often then others. The code lengths are output by indexing `lengths` in the following order: 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15. To send only the required number of codes, a loop is executed on the code lengths in the reverse order to eliminate sending as many as possible code lengths equal to zero. So if `lengths[15]` and `lengths[1]` equal zero, then only the first 17 code lengths would be written. This loop is accomplished in three states, allowing for the RAM access and the two required wait states. The number of code lengths to be written is stored in the variable `index`.

At this point, the GZIP algorithm must choose the method of compression. Each Huffman tree has computed the number of bits required to send the block both dynamically and statically and the LZ77 encoder has stored the number of bytes to reproduce the original data. The total dynamic block length (DBL) is computed using Equation 3.1. The calculation includes the cost to send all of the dynamic Huffman trees and compressed data. The three extra constants at the end of Equation 3.1 account for sending the number of codes in each tree as specified in Figure 2.1. Also, `index` is multiplied by three because each code length in the DCLHT is sent in three bits.


lengths_address=0
1
WAIT
2
prevLen=0
count=0
3
nextLen=lenghts_q
j=0
if (lengths_q==0) then
maxCount=138
minCount=3
else
maxCount=7
minCount=4
4
if (j<maxCode) then
lengths_address=j+1
else
nextLen=31
curLen=nextLen
count=count+1
5
j>=maxCode
WAIT
6
WAIT
7
nextLen=lengths_q
8
j=j+1
if (count<maxCount) and (curLen==nextLen) then
//LOOP
elsif (count<minCount) then
send curLen
elsif (curLen/=0) then
send 16
elsif (count<=10) then
send 17
else
send 18
9

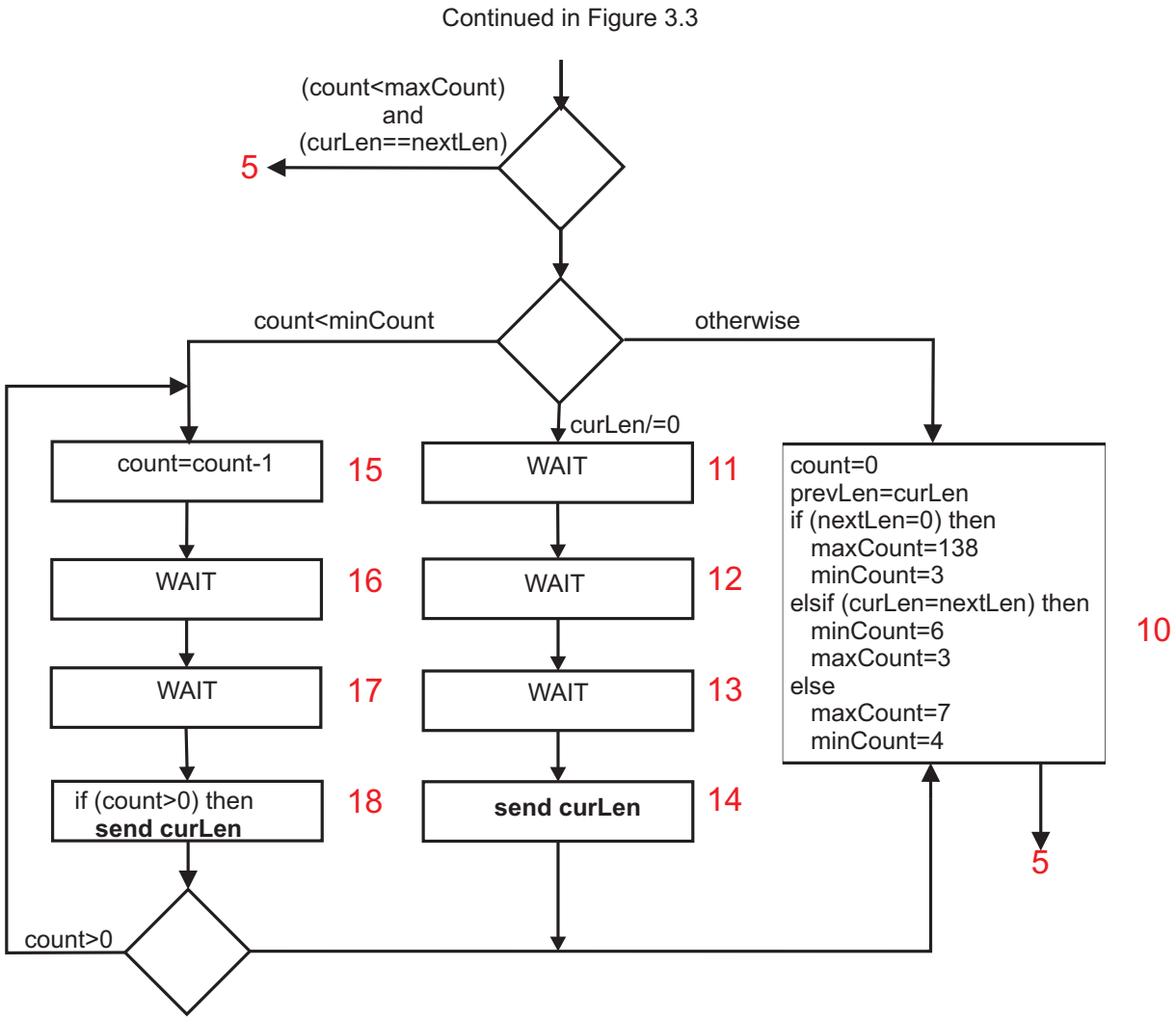


Figure 3.4: Distribution Calculations Flow Diagram (Part 2)

$$\text{DBL} = (\text{DLLHT } \texttt{dynamicLength}) + (\text{DDHT } \texttt{dynamicLength}) + (\text{DCLHT } \texttt{dynamicLength}) + (\texttt{index} + 1) * 3 + 5 + 5 + 4 \tag{3.1}$$

The static block length (SBL) is computed in Equation 3.2. This accounts for only sending the compressed data since tree representations are not necessary in the static case.

$$\text{SBL} = (\text{DLLHT } \texttt{staticLength}) + (\text{DDHT } \texttt{staticLength}) \tag{3.2}$$

The GZIP algorithm chooses the compression mode, fixed, dynamic, or stored, based on whichever of DBL, SBL or stored block length is the smallest. A block header is written to the file regardless of compression mode, the first bit for end-of-file and the last two bits for compression mode. If the stored compression mode is chosen the original data is written to the file. This requires a three state loop to allow reading the original data from RAM. However, if the dynamic method was chosen a representation of the Huffman trees must be written to the file. As in Figure 2.2 the first 14 bits are the number of codes that are going to be written for each Huffman tree. The first five bits are the number of DLLHT code lengths to be sent, DLLHT `maxCode-257`. The DLLHT `maxCode` is guaranteed to be at least 257 because of the end-of-block marker, allowing GZIP to subtract 257 to be able to send the value in five bits. The middle five bits are the number of DDHT code lengths to be sent, DDHT `maxCode-1`. The last four bits are the number of DCLHT code lengths to be sent, `index-4`. Following the first 14 bits, the DCLHT is written. A loop of three states is necessary to allow for the RAM access and each code length is output in three bits. Next the DLLHT is written to the file by going through the distribution function defined previously. Rather than sending the codes to a Huffman tree, the values are referenced in DCLHT and written to the file by calling the `sendBits` function. DDHT is written in the same manner as DLLHT.

Once the dynamic tree representations are complete, the fixed and stored methods are identical

except for which Huffman tree the code value is looked up in. The GZIP algorithm then begins to compress the data by looping through all the values stored from the output of the LZ77 encoder. Remember that flags, literals and distance-length matches were outputted by the LZ77 encoder and stored into the RAM. Two variables are used in this section called `flag` and `flagUsed` where initially `flagUsed` is equal to eight. The algorithm reads each value from the RAM and based on the values of `flag` and `flagUsed` determines how to process the data. If `flagUsed` equals eight then the input is stored in `flag` and `flagUsed` is set to zero. Otherwise, if `flagUsed` does not equal eight and bit 0 of `flag` equals zero then a literal is being processed and its code is looked up in the literal-length tree, either the dynamic tree or the static tree based on the compression mode, and `sendBits` is called. If bit 0 of `flag` is one, then a distance-length match is being processed. This is accomplished by referencing the mapping of the first value in the literal-length tree and sending the code. Followed by reading in another value and referencing its mapping in the distance tree and calling `sendBits`. Each time either a literal or match is processed, `flag` is shifted left by one bit and `flagUsed` is incremented. This process continues until all data has been compressed. Each read from the RAM to retrieve the input requires three clock cycles and each code look up and output requires three clock cycles. As previously mentioned, an end-of-block marker is required following the compressed data, so the value 256 is referenced in the literal-length tree and written to the file.

3.4 Summary

This chapter has outlined the implementation of GZIP compression in hardware. A block overview has been provided, as well as a detailed description of the implementation. The implementation has been broken down into three distinct parts. The first was the overall data flow of the algorithm, including the control signals for the internal encoders. The second and third parts of the implementation were necessary calculations described in terms of state machines written in VHDL.

Chapter 4

Ziv-Lempel Hardware Implementation

This Chapter discusses the hardware implementation of a LZ77 encoder. The LZ77 implementation consists of RAMs and a state machine to compress the data which is written in VHDL. The overall block structure of the LZ77 encoder is described in Section 4.1. Further, details regarding the encoder are discussed in the remaining Sections.

4.1 Block Structure

The block structure for the LZ77 encoder is presented in Figure 4.1. The signals `clock` and `reset` are connected directly to the same signals as the GZIP block. The signals `input`, `inputSent` and `inputValid` are used to provide the LZ77 encoder with data to compress. The signals `sram_address`, `sram_q`, `sram_we` and `sram_data` are used to read and write to a RAM that is necessary for the design. When the algorithm is complete the signal `lzDone` is set to indicate the algorithm is complete and construction of the dynamic Huffman trees may begin. The signals `outLength` and `inLength` are assigned the number of bytes output by the LZ77 encoder and the number of bytes read in by the encoder. The newly compressed values calculated by the LZ77 encoder are transferred directly to Huffman blocks using the signals `tree1Data`, `tree1Valid`, `tree2Data` and `tree2Valid`. Finally, the signal `inputCRC` is the cyclic redundancy check for the input data.

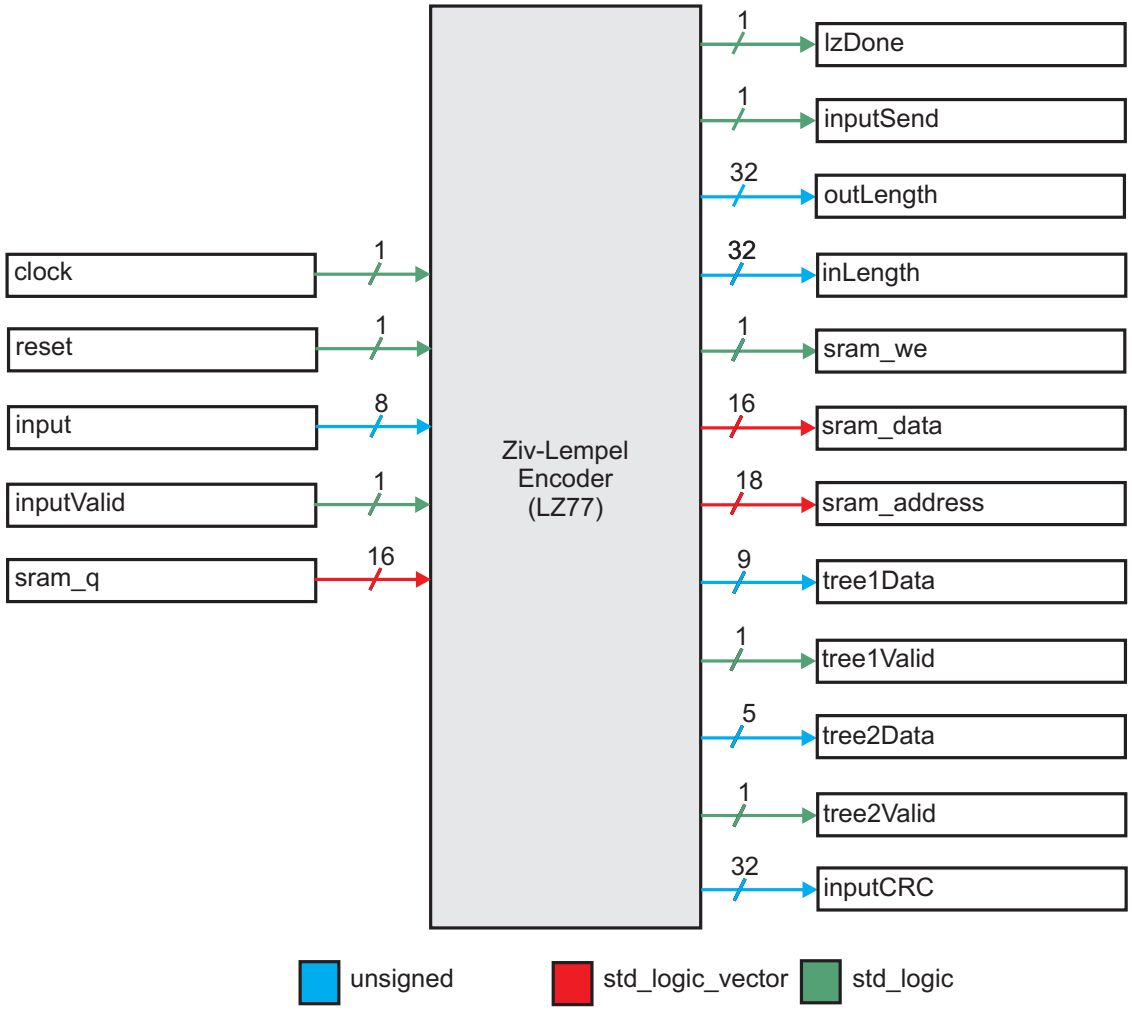


Figure 4.1: Ziv-Lempel (LZ77) Encoder Block

4.2 Major Components

This Section introduces a number of RAMs and global variables used throughout the implementation of the LZ77 encoder. To simplify the process of finding matches for compression, a chaining hashing method is used in the GZIP version of LZ77. The hashtable is composed of `hSize` locations, each of which index the head of a linked list. Each linked list contains the previous elements hashed to that location in the table. The hashtable is composed of two RAMs: `head` and `prev` (see Figure 4.2). The RAM `head` is of size `hSize`$\times$16 bits and represents the most recent string hashed into that position in the hashtable, where `hSize`=32 768. Similarly the RAM `prev` of size `dSize`$\times$16 bits contains the previous strings hashed to that position in the hashtable, where `dSize`=32 768. By storing the hashtable in this manner, if a match were to be found in the hashtable, the first one would be the closest.

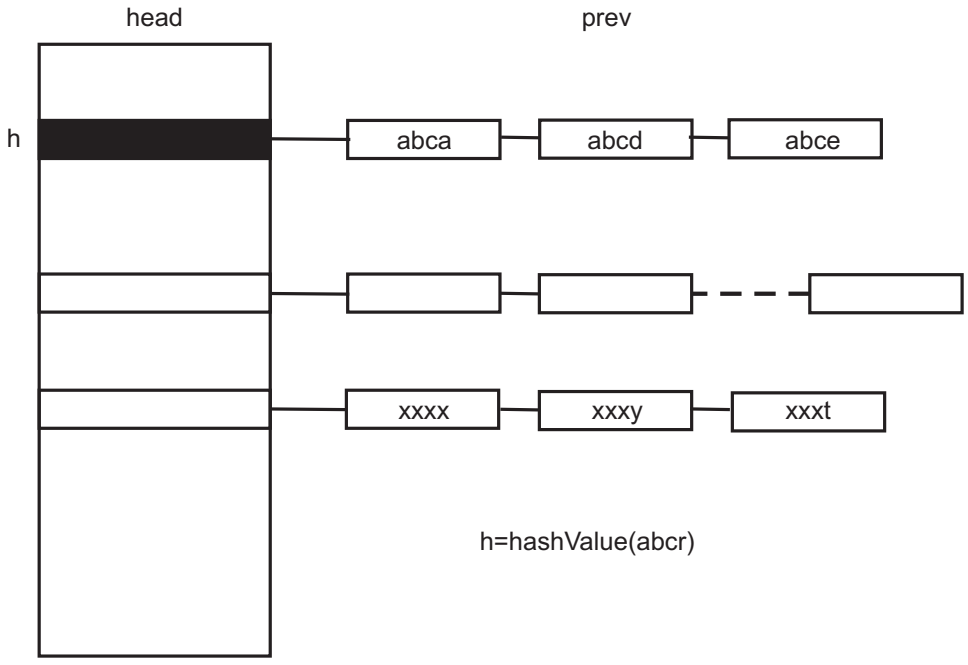


Figure 4.2: Ziv-Lempel (LZ77) Hashtable

The implementation requires a RAM to store the input data. It is called `window` and is of size `wSize`$\times$8 bits, where `wSize`=32 768. A variable `lookAhead` is used to store the number of characters left to be processed and a variable `strStart` is used to index `window`.

An array `encodeData` of size 16×16 bits is used to store the distance-length pairs and literals to be emitted until `flagPos` equals 128, indicating eight iterations of the loop have been completed. After eight iterations have been performed `encodedData` may hold a maximum of 16 elements if eight distance-length pairs have been stored and a minimum of 8 elements if we have eight literals.

Finally, to be able to complete the compression later in the GZIP block, the output of the LZ77 encoder is stored in a RAM of size $33\,792\times16$ bits called `compress`.

Although it is not part of the LZ77 compression algorithm, as input is read, a cyclic redundancy check is computed for the data to ensure the data has been properly recreated in the decompresser. This requires a RAM `crcTable` which is of size 256×32 bits and is initialized with fixed values from a file.

4.3 Implementation

The actual implementation can be broken down into four key parts; initialization, reading data, hash setup and data compression. Each part will be described in detail below using a state machine approach.

4.3.1 Initialization

The purpose of the initialization is two-fold. Several global variables need to be assigned their correct start-up values and the hashtable needs to be cleared by assigning all locations in `head` the value zero. The implementation requires two states for initialization; the first state assigns the required variables with the correct start-up values. During data compression the variable `matchAvailable` is used as a flag to determine if a match should be processed now or whether to wait until the next iteration of the state machine. Also, the variables `flag` and `flagPos` are used to calculate and store the flag values once compression is complete. The second state loops initializing each value of `head` in the hashtable with zero.

4.3.2 Reading Data

Once the initialization is complete, `window` must be filled with characters from the alphabet $\{0 \ldots 255\}$ to be compressed. This process is illustrated in Figure 4.3. A request for a character is sent by assigning `inputSent` the value one. The state machine loops in this state until it can confirm the request has been received, indicated by `inputValid` being non-zero. Also in this state, if this is not the first character to be computed then `inputCRC` is updated using Equation 4.1. In the second state, the value, `input`, is available. If `inputValid` is equal to one, `input` is then assigned to `window[lookAhead]` and `lookAhead` and `inLength` are incremented. To compute the cyclic redundancy check for the input data the RAM `crcTable` is indexed at the value of `input`. These states continue looping while `lookAhead` is less than `wSize` and end-of-file is not reached. At the end of this process, the signal `lookAhead` indicates how many symbols are stored in `window` to be compressed.

$$\texttt{inputCRC} = \texttt{crcTable[input]} \text{ xor } \texttt{inputCRC} >> 8 \tag{4.1}$$

4.3.3 Hash Setup

In the GZIP version of LZ77, the sliding dictionary is implemented as a hashtable. To insert every possible string into the table would be expensive and time consuming. GZIP only considers string of length `minMatch`, where `minMatch=3`. The value 3 was chosen because if a match were to be found, the distance-length pair would not be longer than the original word. An incremental hash function is used allowing the hash value to be incrementally updated for each string rather than having to repeat the calculation `minMatch` times for every string. In order to begin the LZ77 algorithm, the hash value `h` must be updated for the first `minMatch-1` characters (see Figure 4.4). The calculation is accomplished by addressing `window` with the value of `j`, which initially is zero. The next state simply waits for the RAM access. In state three, the hash calculation is completed with `window[j]` as the parameter. The hash calculation is completed in one clock

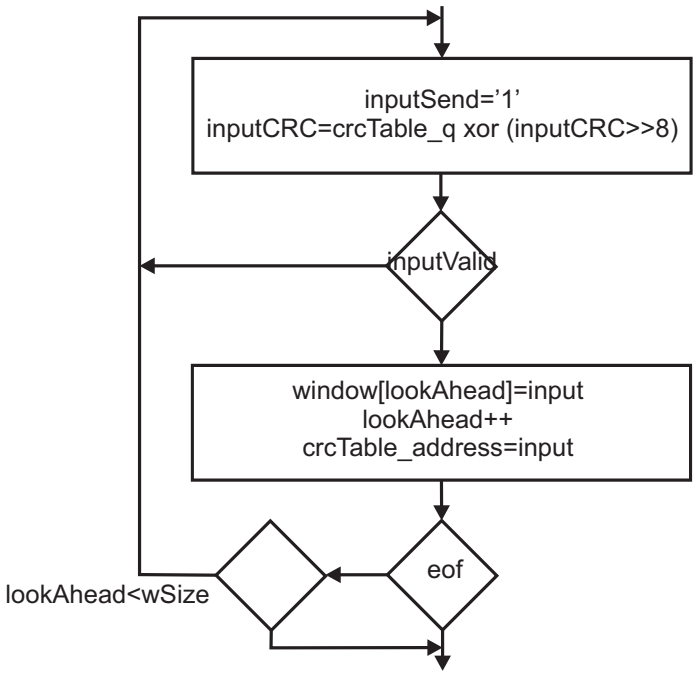


Figure 4.3: Reading Data Flow Diagram

cycle as presented in Equation 4.2. Once the calculation is complete, `j` is incremented, `window` is addressed at the value of `j`, and the state machine loops back to the second state provided `j` is less than `minMatch-1`. At completion of the hash setup all the necessary variables have been initialized and the algorithm is ready to begin compressing the data.

$$\texttt{h} = ((\texttt{h} << 5) \text{ xor } \texttt{parameter}) \text{ and } (\texttt{hSize-1}) \tag{4.2}$$

4.3.4 Data Compression

The data compression portion of the algorithm is the most resource intensive portion of the LZ77 encoder. All the necessary variables have been initialized, including clearing the hashtable. The data to compress has been stored in `window` and the signal `lookAhead` holds the number of symbols located `window`. The initial hash setup has been performed by calculating the hash value for the first `minMatch-1` symbols. An illustration of the entire data compression process

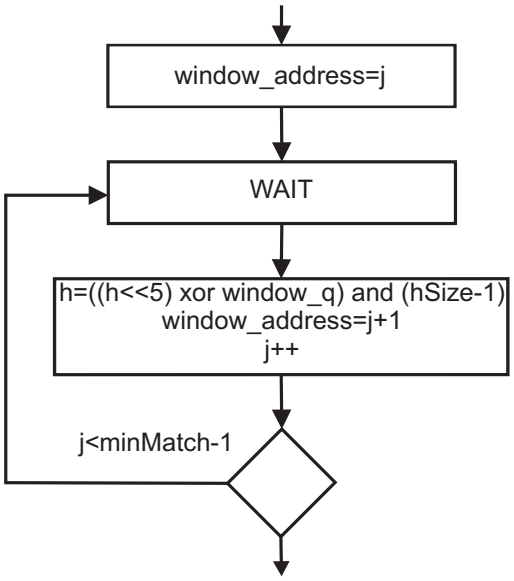


Figure 4.4: Hash Setup Flow Diagram

can be see in Figure 4.5. If `lookAhead` is greater than zero then the compression portion of the algorithm is entered, otherwise the algorithm is complete. The first task is to insert the string of length `minMatch` into the hashtable, this is accomplished in the `insertString` routine and will be discussed in detail in Section 4.3.5.

After the string of length `minMatch` is inserted in the hashtable, the previous match needs to be saved by storing the current length and distance in `prevMatch`. If the value of `hashHead`, which was computed in `insertString` does not equal zero then the `longestMatch` routine is entered, otherwise the function is skipped. The `longestMatch` routine is discussed in detail in Section 4.3.6. The purpose of the `longestMatch` routine is to search through the hashtable and return a match, which could have a length of zero if no match was found.

After exiting the `longestMatch` routine a comparison is performed to determine what, if anything, should be stored for output. If a match was found in the previous iteration of the loop and is better than the new match, the distance-length pair of the previous match will be stored. The first state stores the output to `encodedData`, adjusts the `flag` to specify a distance-length pair has been encoded, decrements `lookAhead` by the previous match length and assigns `matchAvailable` the value zero. Also in the first state, the signals `tree1Data` and `tree2Data` are assigned the mapped

values of the previous match length and match distance and `tree1Valid` and `tree2Valid` are set to one. A second state is then needed to insert all the substrings of length `minMatch` into the hashtable by calling `insertString` several times and incrementing `strStart`. If the first statement was not satisfied, a check is performed to see if `matchAvailable` equals one. If the condition holds true, then no match was found the last iteration and a better match was not found this time, so a literal is stored. The first state addresses `window` at `strStart-1`. The second and third states wait for the RAM access and the fourth state stores the literal to `encodedData`, decrements `lookAhead` and increments `strStart`. Also in the fourth state the signal `tree1Data` is assigned the value of the literal and `tree1Valid` is assigned one. If neither of the previous conditions hold true, the algorithm waits until the next iteration to determine what to do. This is accomplished by assigning one to `matchAvailable`, decrementing `lookAhead`, and incrementing `strStart`.

After processing the information provided by the longest match routine, a check is required to determine if the output is ready to be processed. If `flagPos` is equal to 128 then eight elements have been encoded, either literals or distance-length pairs. If this is true then the `flag` and data stored in `encodedData` is stored in RAM for later use, `flag` and `flagPos` are reset and `outLength` is incremented for each byte stored. If this is not the case, `flagPos` is shifted left by one position to prepare for the next iteration in the loop. After all the checks are complete, the algorithm proceeds back to the first state of the state machine and repeats the entire process.

4.3.5 Insert String Routine

The `insertString` routine inserts the string of length `minMatch` into the hashtable (see Figure 4.6). At first, `window` is addressed at `strStart+minMatch-1`. A wait state is then entered to wait for `window` to be read. In state three, the hash value is updated by executing Equation 4.2 with `window[strStart+minMatch-1]` as the parameter. Once the hash value is updated, `head` is read at the new hash value `h`. Once again, a wait state is required to wait for `head` to be read. In state five, the insertion is performed. It can be seen that insertions are quick and simple, and deletions

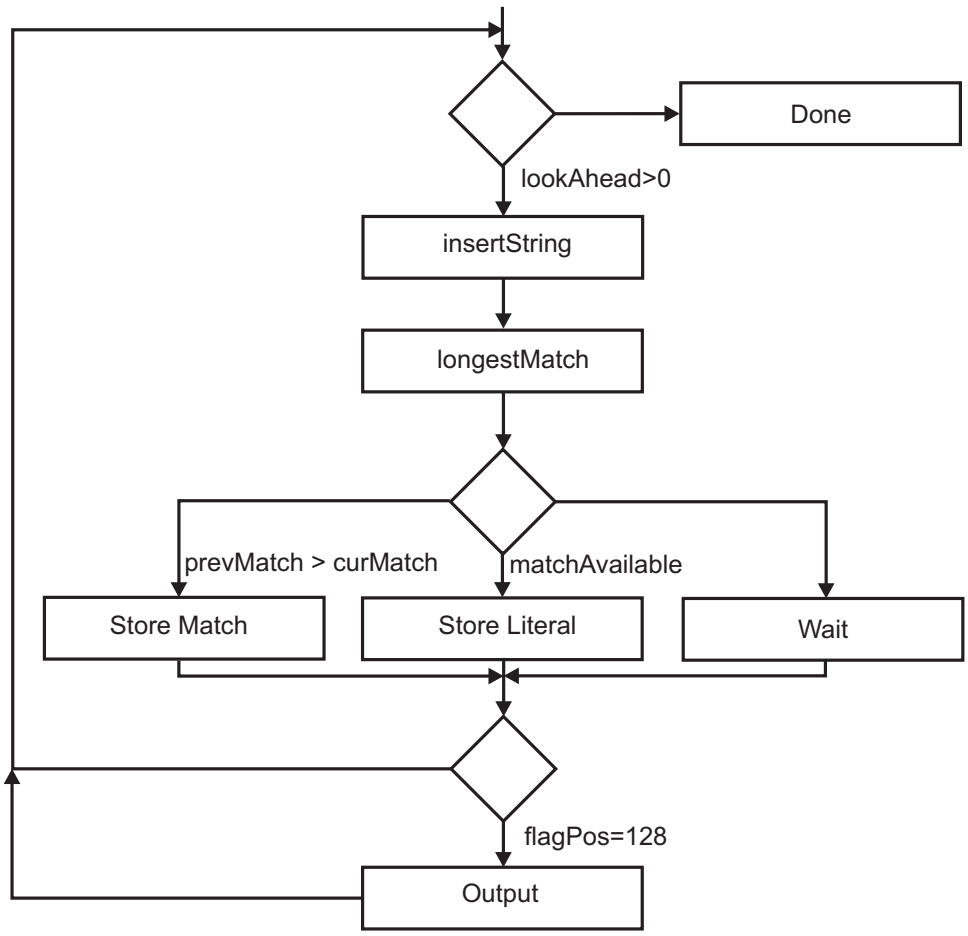


Figure 4.5: Ziv-Lempel (LZ77) Flow Diagram

are unnecessary since a sliding dictionary is used and older strings will be overwritten.

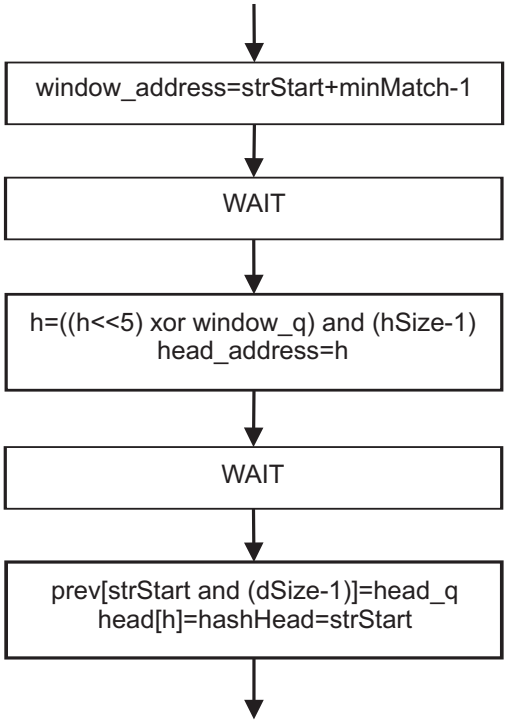


Figure 4.6: Insert String Flow Diagram (insertString)

4.3.6 Longest Match Routine

The `longestMatch` routine searches though the hashtable, finding any possible duplicate strings within the hashtable, this process is illustrated is Figure 4.7. The main feature of the `longestMatch` routine is reading `window` in two different locations and comparing the values returned. Two variables are used to keep track of these locations: `scan` and `match`. Also, a variable called `curMatch` is used to keep track of the head of the linked list that is currently being processed. To begin, `match` and `curMatch` are assigned the value of `hashHead` and `scan` is assigned the value of `strStart`. The `window` RAM is then addressed at both locations and the necessary clock cycles for RAM accesses are waited. If `window[scan]` is equal to `window[match]` then `scan` and `match` are incremented and we return to the third state. One restriction is placed on this loop, allowing a maximum match length of 258, ensuring a reasonably sized set of lengths to encode.

In state six, a comparison is made between the length of the newly found match and any previous match that may have been found. If the new match length is longer, then it is stored, otherwise it is discarded. The algorithm must then continue to process any other strings that have also been hashed to the same value. Previously, `prev` was addressed at `curMatch`, at this point the value is accessed and is assigned to `curMatch`. If the new value of `curMatch` is greater than zero and less than 128 elements in the linked list have been processed, the rest of the linked list is then checked for matches. This is accomplished in the same manner as before, but only with a new value of `curMatch`. Once all necessary assignments are made the algorithm loops back to state two. After 128 elements of the linked-list have been processed or the list comes to an end, the `longestMatch` routine is complete.

4.4 Summary

This Chapter discussed a hardware implementation of LZ77. An overall block description was provided as well as an in depth discussion of the implementation. The implementation was broken down into four parts; initialization, reading data, hash setup and compressing data. Each part was discussed in detail with a state machine approach.

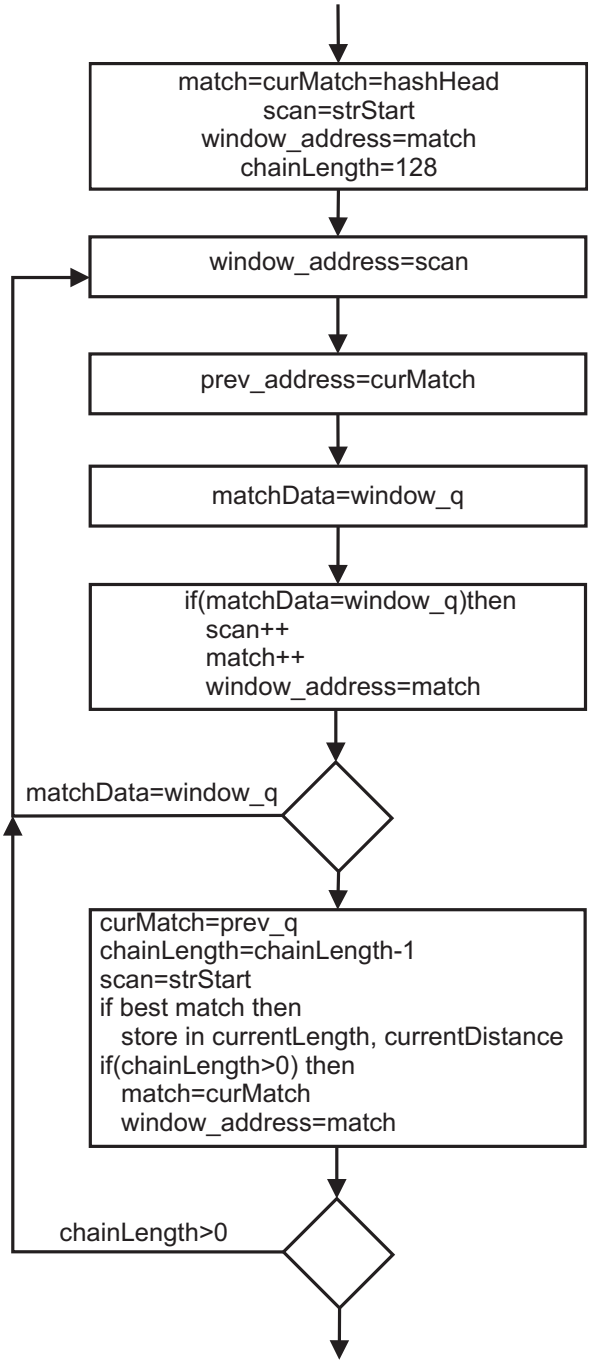


Figure 4.7: Longest Match Flow Diagram (longestMatch)

Chapter 5

Huffman Hardware Implementation

This Chapter discusses in detail the hardware implementation of dynamic Huffman encoders written in VHDL. The three encoders described are important part of the GZIP design. A block overview is provided for each encoder and a general detailed description is provided for all three.

5.1 Block Structure

Due to the different sizes of the three Huffman trees and the slightly different manner each tree interacts with the GZIP block, each Huffman tree has a different block structure. The block structure for each Huffman tree is illustrated in Figure 5.1, Figure 5.2 and Figure 5.3. The `clock` signal in each block is the same clock used in the GZIP encoder. Each block also includes a `reset` signal which is used each time a new block of data is processed. For DLLHT and DDHT, the `input` signal comes directly from LZ77 encoder using `tree1Data` and `tree2Data`. Similarly, the signal `inputValid` for DLLHT and DDHT are assigned the values of `tree1Valid` and `tree2Valid` from the LZ77 encoder. For DCLHT the signals `input` and `inputValid` are assigned by the GZIP encoder. The signal `inputDone` indicates that all the input has been received by the Huffman tree. For DLLHT and DDHT, this signal is assigned the value of `lzDone` in the LZ77 coder. Once the Huffman tree has been built, GZIP needs to be able to access the `lengths` and `codes` RAMs. This is accomplished using the signals `dynamicCodes_address`,

dynamicCodes_q, dynamicLengths_address and dynamicLengths_q. In Figure 5.2 the signals staticLengths_address and staticLengths_q allow the DLLHT encoder to read SLLHT from outside the block. The output signals dynamicLengthsDone and dynamicCodesDone are used to indicate that the code lengths and the codes are finished being calculated. The signals dynamicLength and staticLength are used to assist GZIP in determining which compression mode to use and store the length of the block if it was encoded using the dynamic and static Huffman tree. Finally, in Figure 5.1 and 5.2, maxCode stores the maximum input value that was processed by the specific Huffman tree.

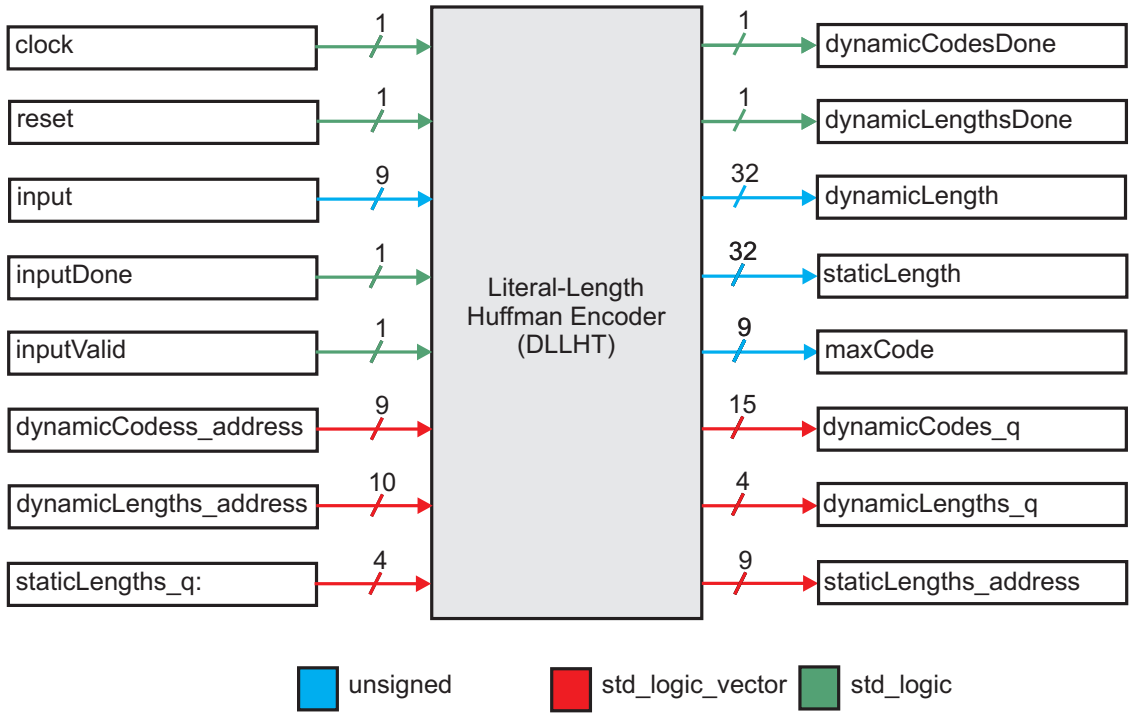


Figure 5.1: Literal-Length Huffman Tree (DLLHT) Encoder Block

5.2 Major Components

As mentioned in Chapter 2, three dynamic Huffman trees are required to compress any block of data. Each Huffman tree will use the same data structures but they will be of a different size since each tree has a different alphabet to be encoded, requiring the data structures to be different

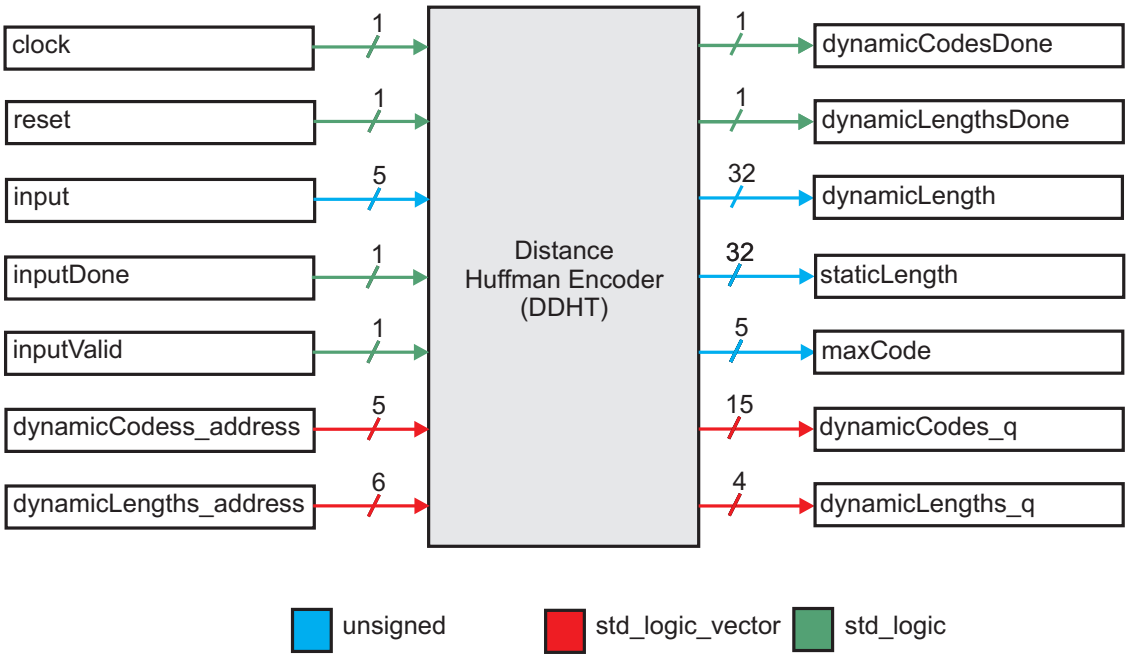


Figure 5.2: Distance Huffman Tree (DDHT) Encoder Block

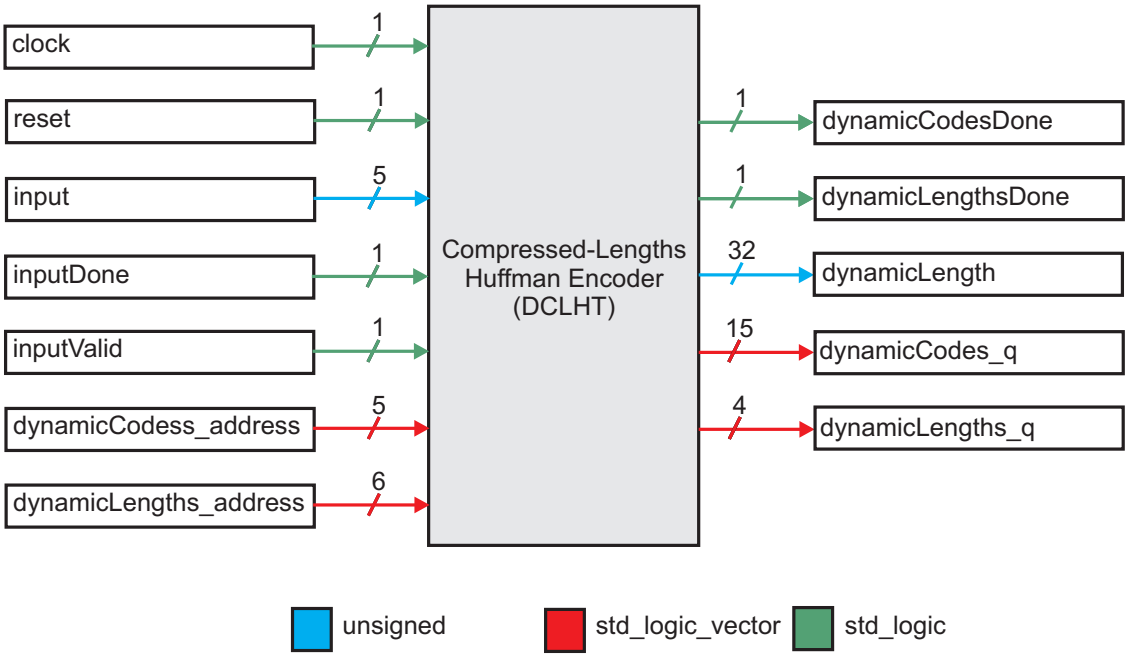


Figure 5.3: Compressed-Lengths Huffman Tree (DCLHT) Encoder Block

sizes (see Table 5.1). This Section introduces the required RAMs and global variables used by the Huffman encoder.

The main feature of a dynamic Huffman coder is that the histogram is calculated from the input data and not read from a look-up table. A RAM is required to store the frequencies of each character, `freq` is initialized to contain all zeros and is incremented each time a character is seen.

Since the main purpose of the Huffman coder is to build a Huffman tree, two RAMs will be required to store the values. A RAM called `parent` is used to keep track of every node's parent in the tree. Similarly, the RAM `depth` is used to keep track of the depth of each node in the Huffman tree.

Perhaps the most important step in the actual Huffman algorithm is to choose two elements with the smallest frequencies. This is accomplished using a heap structure. A RAM `heap` is used to store the heap and it is re-heaped every time an element is removed or added. A variable `heapLen` is used to keep track the length of the heap and the variable `heapMax` is used to store the highest unused element in the heap.

Once the Huffman algorithm is complete, the length of each character in the tree must be computed. This requires a dual-port RAM called `lengths` which does not require the usual 16 bits because GZIP places a restriction on the code length of 15.

Once the code lengths have been calculated the codes can be determined, requiring a dual-port RAM called `codes`. Once again, only 15 bits are required because each code word can only be 15 bits long. The `lengths` and `codes` RAM are dual-port to allow for Huffman tree to be retrieved outside the block.

Two RAMs are used when determining the code lengths and the codes called `nextCount` and `bCount` of size 16×15 bits each.

5.3 Implementation

The Huffman encoder can be described as a six state process including:

1. Initialization

2. Read input
3. Initial heap setup
4. Huffman algorithm
5. Compute code lengths
6. Compute code words

Each stage will be described in the remainder of this Section.

5.3.1 Initialization

The purpose of the initialization is to clear the necessary RAMs and to assign variables with their correct start-up values. The initialization step requires two states. The first state assigns various variables their required initialization values, including `maxCode=-1`. The second state loops, assigning all values of `freq`, `codes`, `lengths` and `bCount` the value zero.

5.3.2 Reading Input

To create a dynamic Huffman tree, one pass must be made through the data and the frequency updated for each character seen. This is accomplished using four states. The first state receives the new input character and address the `freq` RAM at that location. Two cycles are used waiting for the memory ready to complete and in the fourth state the frequency is incremented. The Huffman encoder loops in these four states until all data has been received, indicated by the signal `inputDone`. Once this loop is complete the Huffman algorithm has the statistics needed for the histogram, allowing for dynamic Huffman coding.

Table 5.1: Dynamic Huffman Encoder RAMs

	`freq`	`parent`	`depth`	`heap`	`lengths`	`codes`
DLLHT	572×16 bits	572×16 bits	572×16 bits	572×16 bits	572×4 bits	286×15 bits
DDHT	60×16 bits	60×16 bits	60×16 bits	60×16 bits	60×4 bits	30×15 bits
DCLHT	38×16 bits	38×16 bits	38×16 bits	38×16 bits	38×4 bits	19×15 bits

5.3.3 Heap Setup

To compute the dynamic Huffman tree, the algorithm must be able to choose two elements at a time based on their frequency. A min heap is used to retrieve the smallest element to be processed. This requires the initial elements to be inserted into the heap, and is completed in four states. The first state addresses `freq` at a counter `j` which initially equals zero. States two and three wait for the RAM access and state four does the insertions into the heap. If the `freq[j]` does not equal zero, implying that the character was seen in input, then value `j` is inserted in `heap` at `heapLen+1`, where `heapLen` initially equals zero. Also, each element is assigned a depth of zero by assigning `depth` at `j` equal to zero. This loop continues until all elements in `freq` are processed. Once the loop is complete, a re-heap is required for the first `heapLen/2` elements; this process is described in Section 5.3.4. The variable `maxCode` is assigned the maximum value of `j` that is inserted into the heap. Once the necessary re-heaps have been completed, heap setup is complete and the Huffman algorithm may begin.

5.3.4 Huffman Algorithm

Since a heap is being used to determine the smallest values, a re-heap must be performed every time an element is removed or added. This involves a tremendous amount of sequential reads from RAM as illustrated in Figure 5.4 and Figure 5.5. It begins by reading the first element in the heap, `heap[k]`, where `k=1`. The variable `k` is used to keep track of the element we are considering swapping, in this case, the root of the tree. The variable `a` is used to index the left and right children of `k`. Initially `a` is equal to $2 \times$`k`. After two clock cycles, `heap[k]` is stored in the variable `v` for safe keeping. In the next several clock cycles, `heap` is read at `left=a` and `right=a+1`, and `freq` and `depth` are read at `heap[left]`, `heap[right]` and `heap[k]`. Once all the necessary reads have been completed, the comparisons can be made to determine if any elements need to be swapped. The comparison and swapping can be seen in state 12. The algorithm loops back up to the sequential reads of `left`, `right`, and `k`, until either the `left` or `right` index is larger than the `heapLen` or the re-heap is complete. Just before the re-heap is complete, the value at `v` is written

to the heap at k thus completing the swap.

The actual Huffman algorithm is quite simple once the re-heaping is taken care off. It begins by getting the smallest element from the heap and assigning it to the variable p, decrementing the heapLen and executing a re-heap. Once the re-heap is complete, the next smallest element is removed from the heap and assigned to the variable q. In the next five states, a read is performed of depth and freq at p and q and the results are stored in pDepth, qDepth, pFreq, qFreq are stored. At the same time, heap is written the values p and q at heapMax-1 and heapMax-2 respectively. Once the reads of freq and depth are complete, depth at maxCode+1 is assigned the max(pDepth,qDepth)+1, and freq is assigned pFreq+qFreq. The value of parent is also assigned at p and q with the value maxCode+1. Now that the iteration is almost complete, heap at one is assigned the new value maxCode+1, a re-heap is performed and maxCode is incremented. This loops continues while heapLen is greater than or equal to two. Once the algorithm is complete, the dynamic Huffman tree is stored in the RAMs depth and parent.

5.3.5 Calculating Code Lengths

Once the Huffman tree has been computed and all the pointers in parent have been assigned, computing the code lengths is quite simple. Unfortunately, because of sequential RAM accesses a number of states are required (see Figure 5.6 and 5.7). Initially the heap is addressed at a counter j, which is initialized to zero. Two cycles later when the value is accessible, parent is addressed at heap[j]. Once again, two cycles must pass before the value is accessible. In the next state, lengths is addressed with the newly available parent[heap[j]]. Two cycles later, length at heap[j] is assigned lengths[parent[heap[j]]]+1. If the value heap[j] is less than maxCode, which implies that it was one of the original input characters, then bCount at lengths[parent[heap[j]]]+1 is incremented by one. It should be mentioned that this is an original copy of maxCode before it was incremented in the Huffman algorithm. Once this is complete, j is incremented and the loop starts over until every element of the tree is processed. This routine is similar to traversing a tree, where a child node has length equal to one plus its

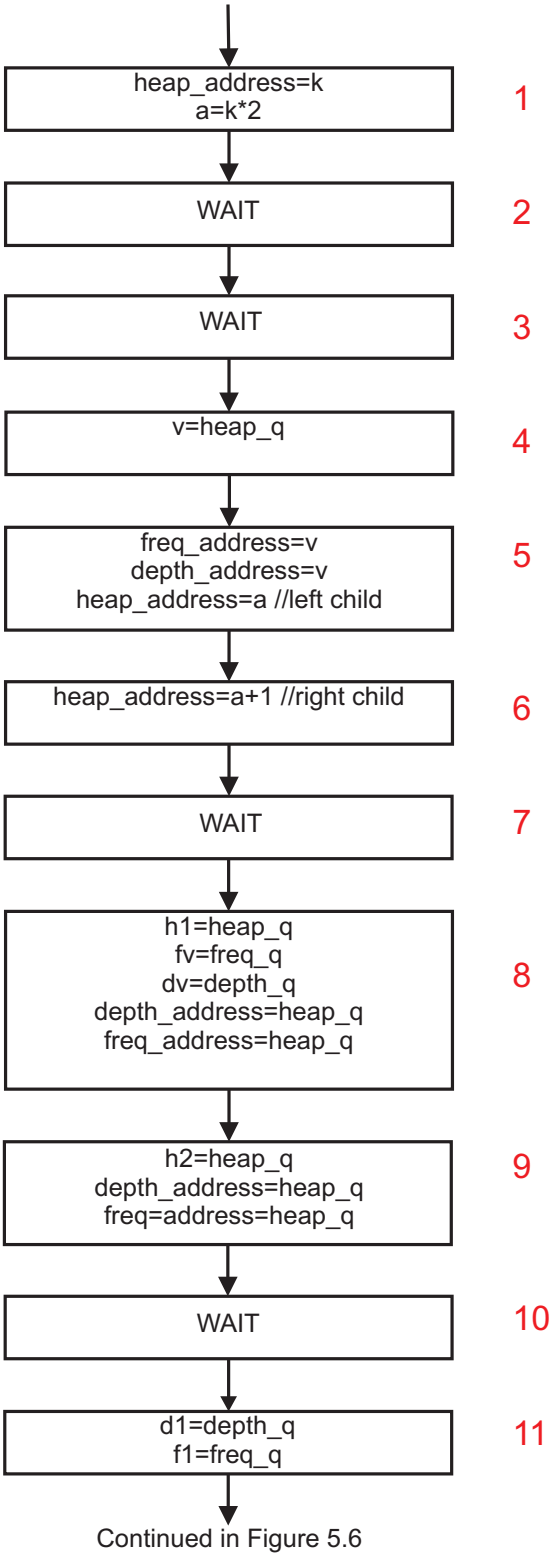


Figure 5.4: Re-heap Flow Diagram (Part 1)

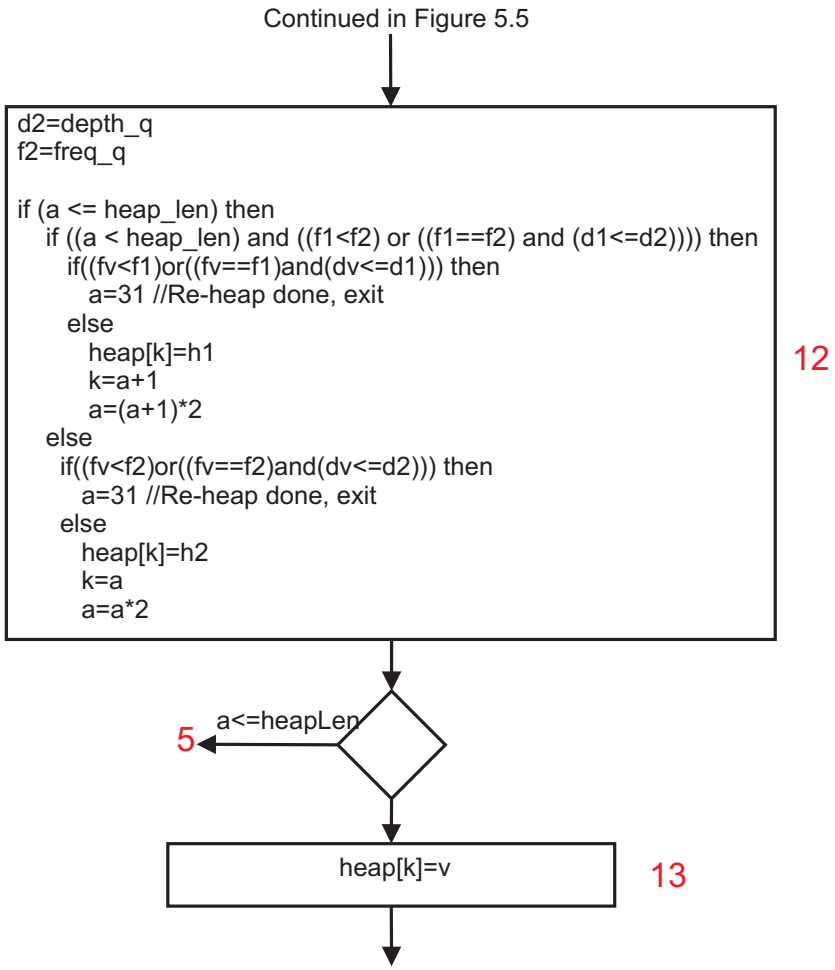


Figure 5.5: Re-heap Flow Diagram (Part 2)

parent's length. The assignment to `bCount` at the end creates a list of how many code lengths there are for each length. Once this loop is complete, all code lengths have been determined and stored in `lengths`.

For GZIP to be able to determine which compression mode will produce the highest compression ratio, it requires the potential length of the block when compressed by dynamic Huffman trees and static Huffman trees. At this point in the algorithm, this involves the number of times a character occurs and the number of bits required to send that character in the static and dynamic representations. The dynamic calculation can be computed by addressing `freq` and `lengths` at the appropriate location. The two values are simply multiplied together and added to the current value of `dynamicLength` which is initially zero. This calculation occurs for all three Huffman trees. The static calculation is only required for DLLHT and DDHT because if a static representation is chosen DCLHT is never used. This requires addressing `freq` and `staticLengths` for the DLLHT calculation. The DDHT calculation only requires the frequency because all the codes have length five. The static calculation is performed by multiplying the static length by the frequency and adding it to the current value of `staticLength` which is initially zero.

5.3.6 Calculating Codes

To ensure that the decompresser and compressor can compute the same codes, GZIP calculates the code words based on the code lengths. Calculating the code words based on the `code` lengths requires two separate loops as illustrated in Figure 5.8 and 5.9. The first loop runs from `j` equal to zero to `maxLength`, which equals 15, assigning the first code for each length to `nextCount`. A variable `code` is used in this loop and is initially zero. The loop begins by reading the value of `bCount` at `j`, this of course takes two clock cycles. In the third cycle, code is incremented by `bCount[j]` and shifted left by one position and written to `nextCount[j]`. This loop stores the first code for each code length in `nextCount`.

The second loop runs from `j` equal to zero to `maxCode`. The `lengths` RAM is addressed at the variable `j`, and after two clock cycles, if `lengths[j]` does not equal zero `nextCount` is addressed

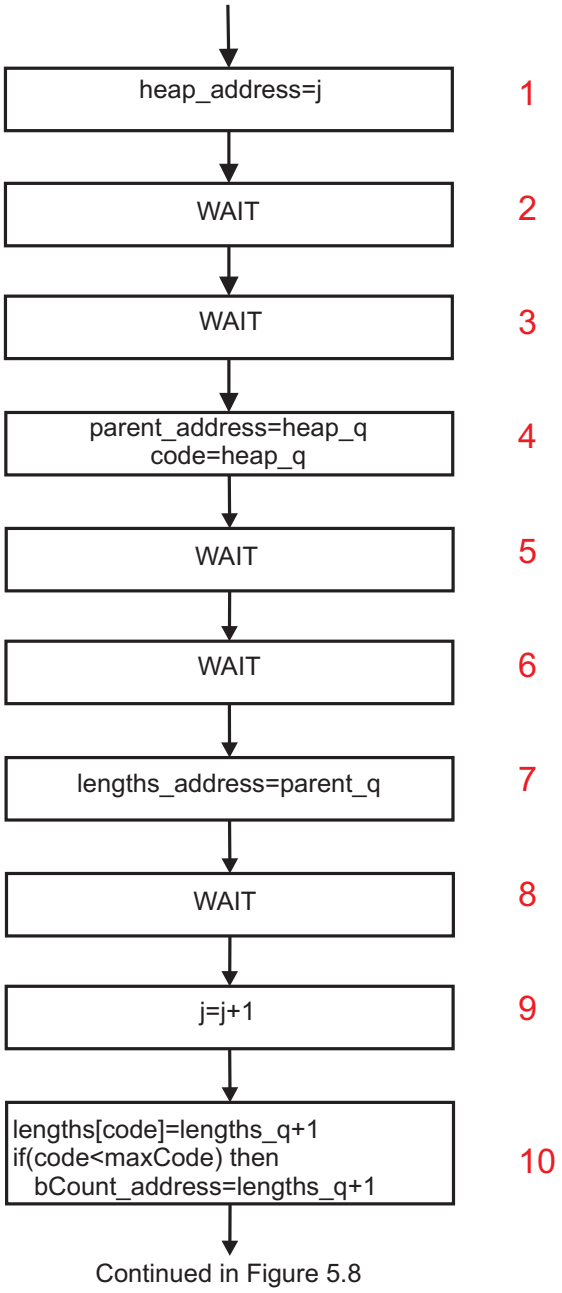


Figure 5.6: Code Length Computation Flow Diagram (Part 1)

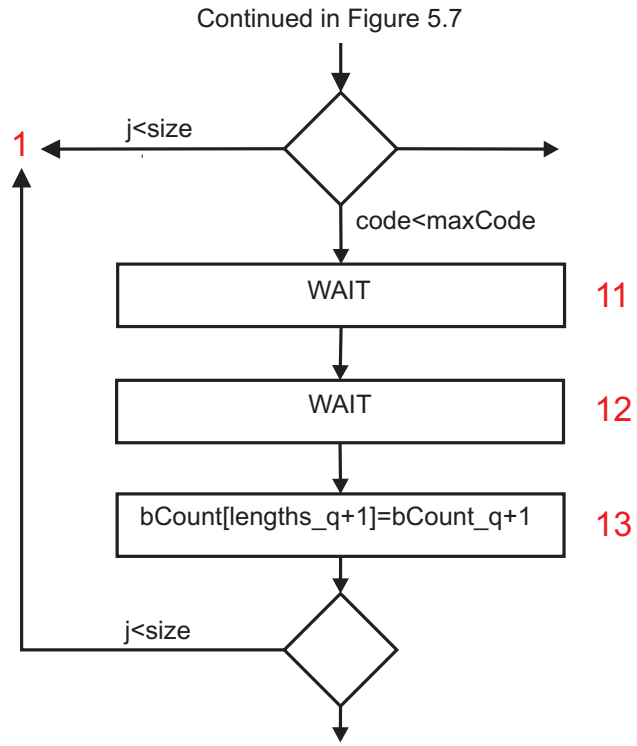


Figure 5.7: Code Length Computation Flow Diagram (Part 2)

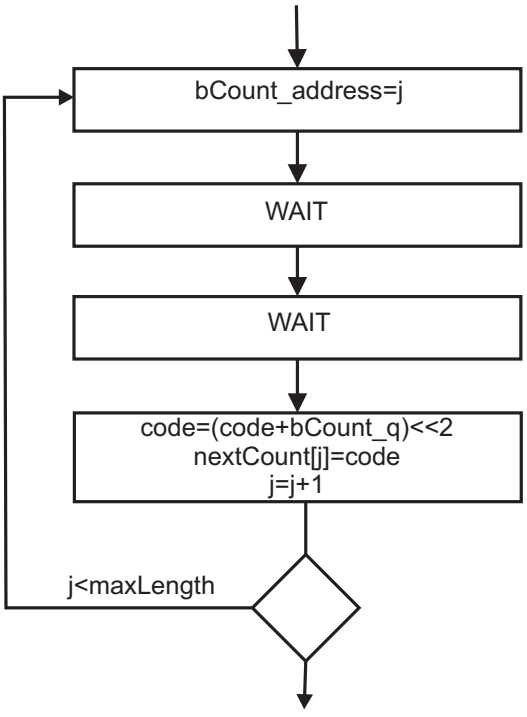


Figure 5.8: Code Computation Flow Diagram (First Loop)

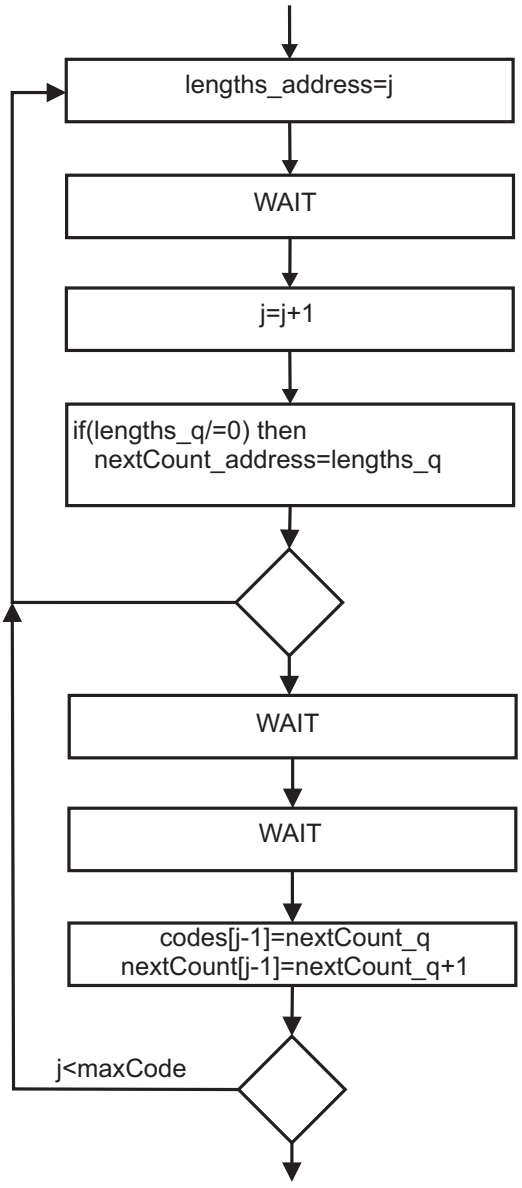


Figure 5.9: Code Computation Flow Diagram (Second Loop)

with the value `lengths[j]`. Two cycles later, `codes` is assigned the value `nextCount[lengths[j]]` at `j` and `nextCount[lengths[j]]` is incremented by one. This loop effectively reads `nextCount` to get the first available code for that code length, assigns it to the tree and then increments it for the next code.

5.4 Summary

This Chapter has discussed in detail the hardware implementation of three Huffman encoders that are necessary for the implementation of GZIP. An overview of each block has been provided and described. Each encoder uses RAMs and a state machine to produce a dynamic Huffman tree.

Chapter 6

FPGA Prototype

In the design of any hardware, an actual working prototype is important for proof of concept. Consequently, this Chapter discusses the actual implementation of our GZIP hardware on an FPGA prototype board; namely the Altera DE2 board [7].

Section 6.1 provides a description of the Altera DE2 board components and specifications in detail. In using this prototype board, the different RAMs used in the design needed to be partitioned to the available board resources. Section 6.2 provides a breakdown of all the RAMs used in the design and specifies their locations on the DE2 board. Finally, Section 6.3 provides a brief description of the overall system layout, the test harness used and the interactions between the different components.

6.1 Altera DE2 Components

This Section outlines the structure and specifications of the Altera DE2 Board as specified in the DE2 User Manual [7]. A photograph of the Altera DE2 board is provided in Figure 6.1. The GZIP hardware design utilizes the following features:

Cyclone II 2C35 FPGA

- 33 216 LEs
- 105 M4K RAM blocks

- 483 840 total RAM bits
- 35 embedded multipliers
- 4 PLLs
- 475 user I/O pins
- FineLine BGA 672-pin package

SRAM

- 512 kByte Static RAM memory chip
- Organized as 256k x 16 bits
- Accessible as memory for the Nios II processor and by the DE2 Control Panel

SDRAM

- 8 MByte Single Data Rate Synchronous Dynamic RAM memory chip
- Organized as 1M x 16 bits x 4 banks
- Accessible as memory for the Nios II processor and by the DE2 Control Panel

SD Card Socket

- Provides SPI mode for SD Card access
- Accessible as memory for the Nios II processor with the DE2 SD Card Driver

The DE2 board also incorporates many useful input and output devices including; pushbutton switches, toggle switches, LEDs, 7-segment displays, and a LCD display. Several extra features not used in the design are also available; USB 2.0, 10/100 Ethernet, 1 MByte Flash memory, IrDA transceiver, expansion headers, VGA video DAC, TV decoder, audio CODEC, PS2 port and a RS-232 port. These features allow for a wide range of circuit designs to be created by the user. Figure 6.2 provides the block diagram for the Altera DE2 Board. All connections are made through the Cyclone II FPGA device allowing the user to configure the FPGA to implement any design.

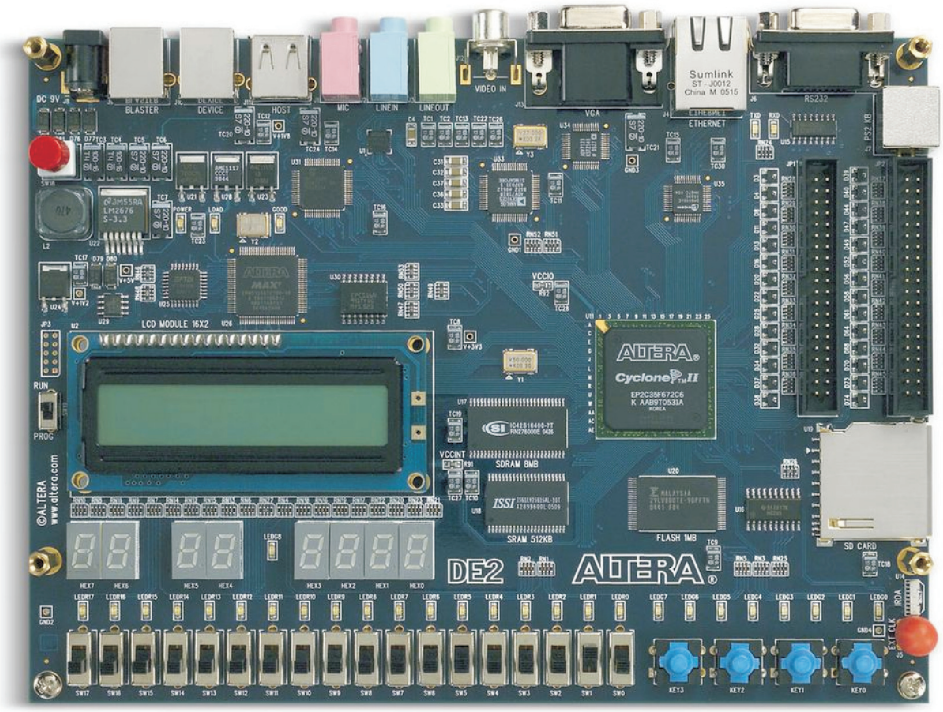


Figure 6.1: Altera DE2 Board [7]

6.2 Assignment of RAM

All of the blocks described in this thesis (namely, the GZIP encoder, the LZ77 encoder, and each of the dynamic Huffman encoders) required several RAMs to be able to complete their calculations. Due to the memory limitations of our FPGA, some of these RAMs have been offloaded onto an off-chip SRAM. Tables 6.1, 6.2 and 6.3 list each RAM used in each component of the design and specifies their location on the development board. Given an FPGA with enough memory bits available, the SRAM would not be needed by the design.

Table 6.1: GZIP RAM Breakdown

Name	Size	Location
`SLLHTLengths`	286×4 bits	FPGA
`SLLHTCodes`	286×9 bits	FPGA
`SDHTCodes`	30×5 bits	FPGA

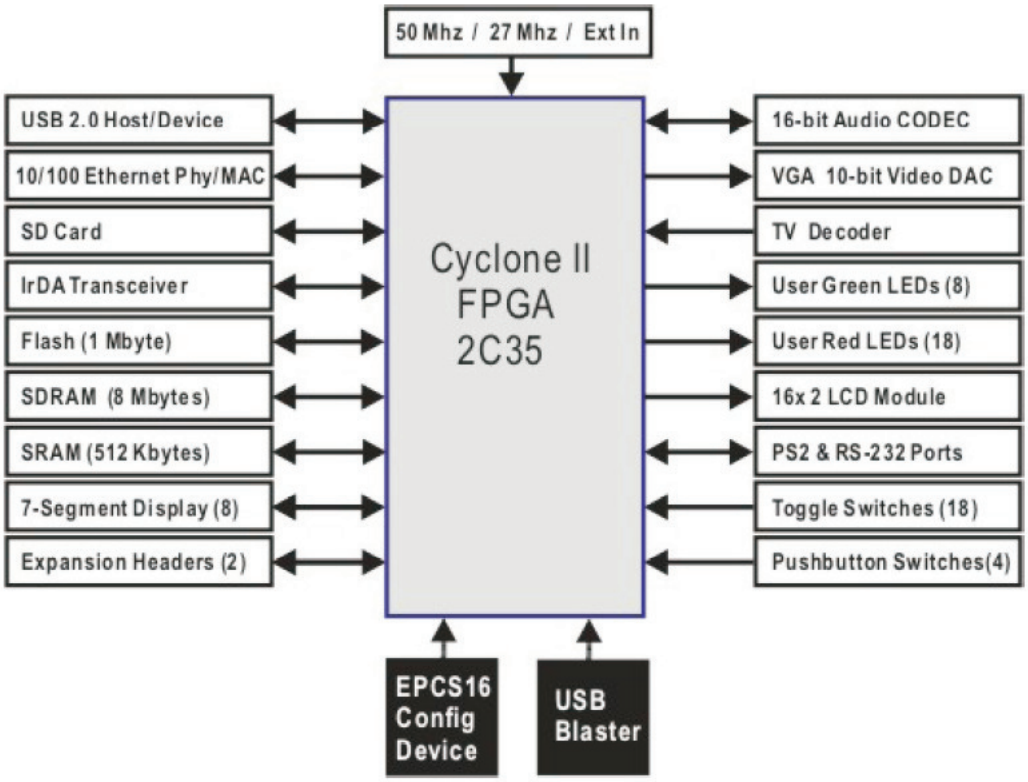


Figure 6.2: Altera DE2 Block Diagram [7]

Table 6.2: LZ77 RAM Breakdown

Name	Size	Location
`head`	32 768×16 bits	SRAM
`prev`	32 768×16 bits	SRAM
`window`	32 768×8 bits	SRAM
`encodeData`	16×16 bits	FPGA
`compress`	33 792×16 bits	SRAM
`crcTable`	256×x32 bits	FPGA

Table 6.3: Huffman RAM Breakdown

Name	Size	Location
DLLHT `freq`	572×16 bits	FPGA
DLLHT `parent`	572×16 bits	FPGA
DLLHT `heap`	572×16 bits	FPGA
DLLHT `lengths`	572×4 bits	FPGA
DLLHT `codes`	286×15 bits	FPGA
DLLHT `nextCount`	16×15 bits	FPGA
DLLHT `bCount`	16×15 bits	FPGA
DDHT `freq`	60×16 bits	FPGA
DDHT `parent`	60×16 bits	FPGA
DDHT `heap`	60×16 bits	FPGA
DDHT `lengths`	60×4 bits	FPGA
DDHT `codes`	30×15 bits	FPGA
DDHT `nextCount`	16×15 bits	FPGA
DDHT `bCount`	16×15 bits	FPGA
DCLHT `freq`	38×16 bits	FPGA
DCLHT `parent`	38×16 bits	FPGA
DCLHT `heap`	38×16 bits	FPGA
DCLHT `lengths`	38×4 bits	FPGA
DCLHT `codes`	19×15 bits	FPGA
DCLHT `nextCount`	16×15 bits	FPGA
DCLHT `bCount`	16×15 bits	FPGA

6.3 System Layout

To facilitate testing of the hardware implementation of GZIP, a test harness was created using a NIOS II processor and the SDRAM. The interface used links the NIOS processor to the on board SD Card reader. The SDRAM is used as stack space for the processor, thus not placing further demands on the available memory (M4K) blocks within the FPGA. Figure 6.3 illustrates the interaction between all board components; the SDRAM, SRAM, SD Card reader and the FPGA. Each time the design wants to send or receive data from the SD Card, an interrupt is sent to the processor. The processor can only service new interrupts when not currently serving another interrupt.

6.4 Summary

This Chapter has outlined the FPGA prototype. The FPGA, NIOS processor, SRAM and SDRAM have been utilized in the design. A general description has been included for the prototype and the Altera DE2 Board specifics have been provided. A further breakdown of all the RAMs used throughout the design has been provided, including their location on the prototype board.

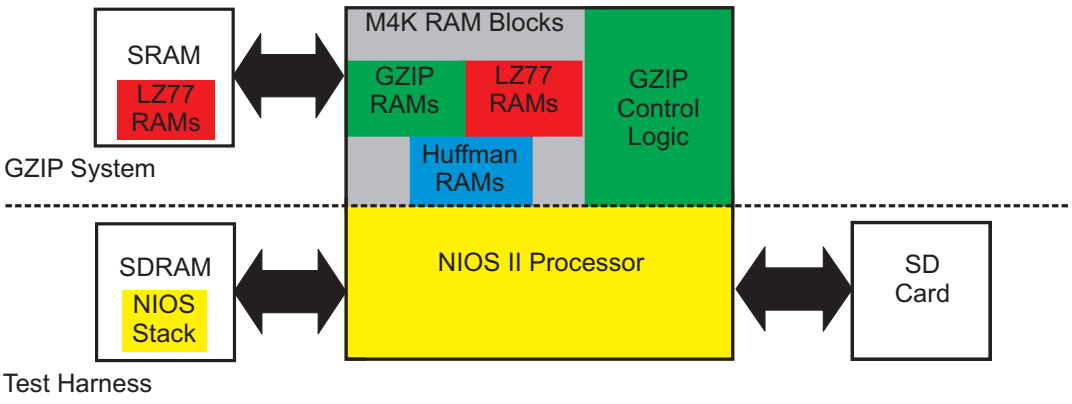


Figure 6.3: Complete Overview of Design

Chapter 7

Experiments

To evaluate our hardware implementation of GZIP a series of tests were completed on a set of benchmarks. In Section 7.1 the testing procedure is discussed in detail and the results for compression ratio and runtime are provided. In Section 7.2 the FPGA resource utilization is provided and broken down for each component. In the last Section, some design analysis is provided, including discussion of the critical path for the design.

7.1 Test Results

The benchmarks used are from the University of Calgary corpus [28]. This corpus is present in a majority of data compression research papers and provides a good comparison. The list of files and the number of bytes in each file are provided in Table 7.1. The benchmarks were tested using three different methods. The first method will use the standard software GZIP on a UNIX personal computer with a 2.8 GHz processor and 1 GB RAM. The second method tests our hardware implementation in simulation. Finally, the third method tests the hardware implementation on the Altera DE2 board. By testing in this manner a comparison can be made between compression ratios and relative runtime.

The runtime results for each method are illustrated in Figure 7.1. Some of the columns have been truncated to allow viewing of the CPU results. For the complete results, refer to Table 7.2.

Table 7.1: University of Calgary Corpus

Graph Index	Filename	File Size (Bytes)
1	bib	111261
2	book1	768771
3	book2	610856
4	geo	102400
5	news	377109
6	obj1	21504
7	obj2	246814
8	paper1	53161
9	paper2	82199
10	paper3	46526
11	paper4	13286
12	paper5	11954
13	paper6	38105
14	progc	39611
15	progl	71646
16	progp	49379
17	trans	91692

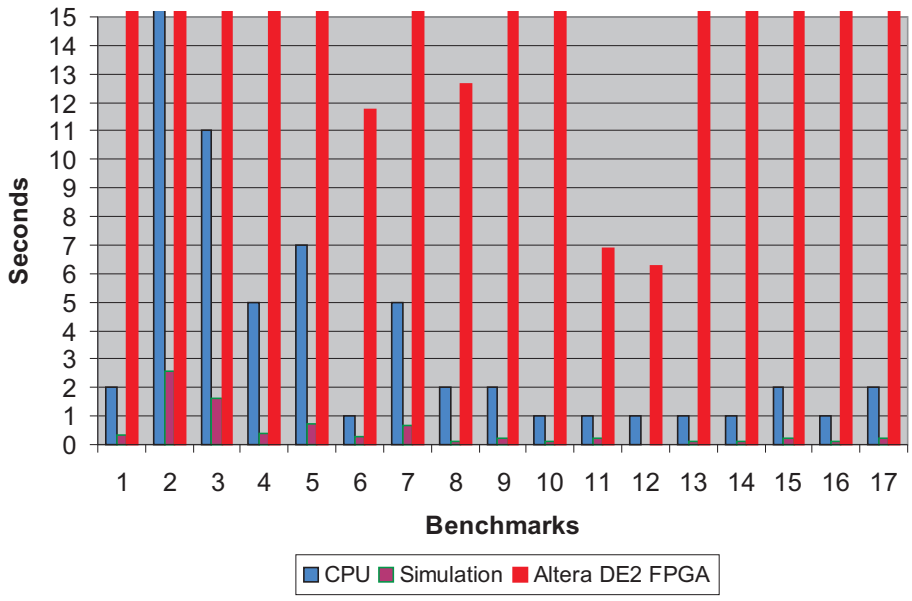


Figure 7.1: Compression Runtime Results

Table 7.2: Compression Runtime Results

File	CPU Runtime	Simulation Runtime	FPGA Runtime
1	2s	0.33s	55.92s
2	17s	2.56s	775.57s
3	11s	1.62s	595.71s
4	5s	0.37s	64.95s
5	7s	0.75s	184.36s
6	1s	0.26s	11.77s
7	5s	0.66s	113.24s
8	2s	0.13s	12.62s
9	2s	0.24s	39.65s
10	1s	0.12s	22.92s
11	1s	0.25s	6.86s
12	1s	0.02s	6.28s
13	1s	0.1s	18.05s
14	1s	0.1s	18.66s
15	2s	0.23s	29.48s
16	1s	0.13s	20.38s
17	2s	0.2s	38.72s

As previously mentioned, when testing on the Altera DE2 board an SD Card is used for input and output. This causes a great deal of latency since the GZIP compression algorithm often has to wait for the SD Card access. This does not allow for a fair comparison between software GZIP and our Altera DE2 hardware implementation. The simulation on the other hand assumes that accessing the input and output source will only take two clock cycles, providing a more accurate comparison to software. Both the simulation and Altera DE2 hardware implementations are driven by a 50 MHz clock, allowing us to compute the number of seconds based on the required number of clock cycles. For example, if 100 000 000 clock cycles were required to perform compression the runtime would be computed to be 2 seconds as in Equation 7.1.

$$Runtime = \frac{Clock\,Cycles}{Frequency} = \frac{100\,000\,000}{50\,000\,000} = 2\text{ s} \tag{7.1}$$

It is apparent that the hardware simulation is slightly faster than the software implementation.

This is misleading because it does not account for gate delays. In reality, software will be faster as the speed of memory access on a CPU exceeds the speed of accessing RAM in hardware.

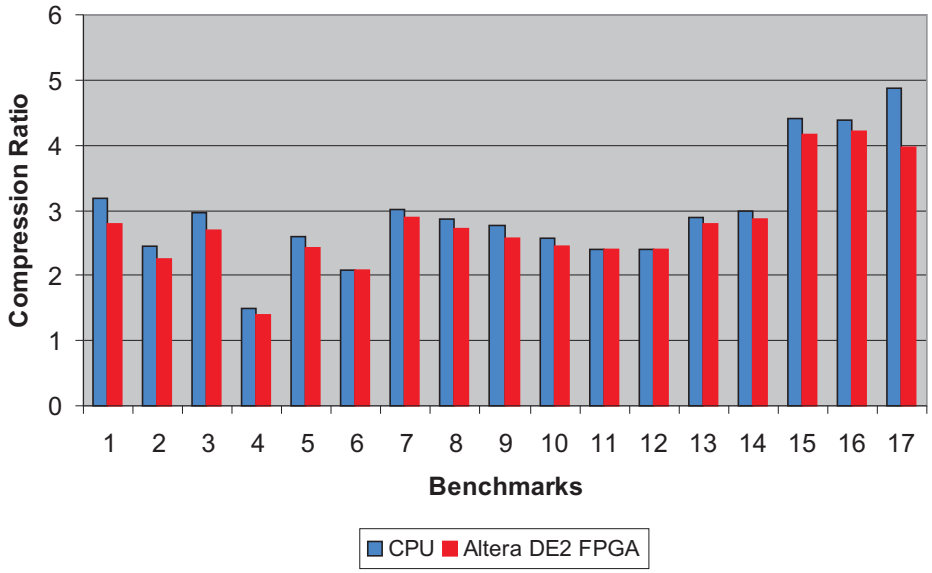


Figure 7.2: Compression Ratio Results

The compression ratio results for each method are illustrated in Figure 7.2. It is only necessary to report the results for the CPU method and the Altera DE2 method since the simulation compression ratios will be the same as the FPGA compression ratios. There is a slight variation between the software and hardware compression ratios, on average 2%. This is caused by the fact the software version of GZIP allows the LZ77 algorithm to reference matches in previous blocks. The hardware implementation does not allow this due to memory restrictions on the FPGA.

7.2 FPGA Resources Utilized

The FPGA resource requirements for the hardware implementation of GZIP compression are outlined in Table 7.3.

The resources used by the design can be further broken down for each individual component and are presented in Table 7.4. Keep in mind that the GZIP encoder contains the DLLHT, DDHT, DCLHT, SLLHT, SDHT and the LZ77 encoder. It is apparent that the block instantiations within

Table 7.3: FPGA Resource Utilization for Hardware Implementation of GZIP Compression

Resource	Number Used	Percent of FPGA Used
Logic Elements	20 610	62%
Registers	3884	-
Pins	431	91%
Virtual Pins	0	-
Memory Bits	69 913	14%
9-bit Multipliers	26	37%
PLLs	1	25%

the GZIP encoder consume 43% of the logic elements, 60% of the registers, 80% of the memory bits and 100% of the 9-bit multipliers of the GZIP encoder resources. Thus, the remaining resources utilized are consumed by the connections between the blocks and the NIOS II processor.

Table 7.4: FPGA Resource Utilization Breakdown by Component

Component	Logic Elements	Registers	Memory Bits	9-bit Multipliers
GZIP Encoder	18380	2673	59673	26
• DLLHT	2066	573	40234	10
• DDHT	1980	533	4410	10
• DCLHT	1842	502	2969	6
• SLLHT	0	0	3718	0
• SDHT	0	0	150	0
• LZ77 Encoder	2077	734	8192	0
NIOS II Processor	2230	1211	10240	0

7.3 Design Analysis

The design was compiled using Altera Quartus II software and was clocked with a 50 MHz clock. After further analysis it was discovered that one portion in the implementation was actually being clocked at 25.6 MHz. The critical path in the design was from the DDHT and DCLHT dual-port `lengths` RAM to the output of data in the GZIP encoder. Since the necessary wait states were provided, this was not an issue and this portion of code can be run at 50MHz.

7.4 Summary

This Chapter has compared the hardware implementation of GZIP with the current software version available. It has been found that the hardware version achieves a reasonable runtime given the prototype limitations and would have been better had a input/output device with less latency had been used. The compression ratio was found to be within 2% of the GZIP software utility. Given the design tradeoffs discussed earlier, this is an acceptable difference. Section 7.2 provides a breakdown of the FPGA resouces used by component. The last Section discusses the clock frequency and critical path for the design.

Chapter 8

Conclusion

8.1 Summary and Contributions

This thesis discussed the design and implementation of GZIP in hardware. Several details were investigated, including: LZ77, Huffman and GZIP itself.

Previous research in the field has been limited to hardware implementations of Huffman and LZ77 encoders. Several individuals in the field have alluded to the idea of implementing GZIP in hardware, but limited their implementation to the static Huffman encoding (fixed mode). *However, the implementation discussed in this thesis supports all compression modes of a GZIP compression utility and its output can be decompressed with any standard software GZIP utility.* By allowing GZIP compression to be offloaded from a CPU, a processor can be freed to complete other tasks, allowing tasks to be completed in parallel.

The compression ratio and runtime generated by the hardware implementation of GZIP was compared to the software version of GZIP and found to be favorable. The compression ratio was on average within 2% and reasonable runtime was recorded given the limitations of our FPGA prototype. The choice to make each LZ77 data block independent caused the variation in compression ratio. Also, the runtime would have been much quicker had a device with less input/output latency been used. Since this implementation is written in VHDL, it is fully portable

to a variety of hardware architectures. As FPGA technology improves, the performance of the hardware implementation of GZIP will improve with out the need to change the design.

8.2 Future Directions

There is still room to improve the performance and quality of results from the GZIP hardware implementation. Our implementation of the LZ77 encoder assumes each block or compressed data is independent. This causes the beginning of each block to be mostly literals as there is no data to compare against and potentially find a match. This could be avoided by allowing a match to occur within the previous 32 768 characters regardless of block boundary. Another avenue to investigate could include allowing blocks to be different sizes based on some heuristic function rather than a fixed size. In the dynamic Huffman encoder, it was found that the reorganization of the heap was the most resource intensive process. If a parallel heap structure existed in hardware, our implementation speed would improve significantly. Also, in the LZ77 encoder a large number of sequential reads were required to find the longest match. If a large enough dual-port RAM was available RAM accesses could have been performed in parallel. Another idea to investigate includes a hardware implementation of GZIP decompression.

Bibliography

[1] J. Gailly, *GZIP the Data Compression Program*, 1993. `ftp://ftp.gnu.org/gnu/GZIP/GZIP-1.2.4.tar.gz`.

[2] T. A. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer.*, vol. 17, pp. 8–19, 1984.

[3] J. Ziv and A. Lempel, "Compression of Individual Sequences Via Variable-Rate Coding," *IEEE Transactions on Information Theory*, 1978.

[4] S. Leinen, *Long-Term Traffic Statistics*, 2001. `http://www.cs.columbia.edu/~hgs/internet/traffic.html`.

[5] M. Akil, L. Perroton, S. Gailhard, J. Denoulet, and F. Bartier, "Architecture for Hardware Compression/Decompression of Large Images," *SPIE Electronic Imaging*, vol. 4303, pp. 51–58, 2001.

[6] J. V. P. Rauschert, Y. Klimets and A. Kummert, "Very Fast GZIP Compression by Means of Content Addressable Memories," vol. 4, pp. 391–394, December 2004.

[7] Altera Corporation, *Development And Education Board Version 1.3*, 2006. `http://www.altera.com/education/univ/materials/boards/DE2_UserManual.pdf`.

[8] L. Deutsch, *DEFLATE Compressed Data Format Specification Version 1.3*, 1996. `ftp://ftp.uu.net/pub/archiving/zip/doc/`.

[9] USWeb, *The Online Marketing Benefits of GZIP*, 2006. `http://blog.usweb.com/archives/the-value-online-marketing-benefits-of-GZIP/`.

[10] Network Working Group, *Hypertext Transfer Protocol – HTTP/1.0*, 1996. `http://www.w3.org/Protocols/rfc1945/rfc1945`.

[11] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[12] Z. Li and S. Hauck, "Configuration Compression for Virtex FPGAs," *Field Programmable Custom Computing Machines*, pp. 147–159, 2001.

[13] J. Storer and T. Szymanski, "Data Compression via Textual Substitution," *Journal of the ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[14] N. Larsson, "Extended Application of Suffix Trees to Data Compression," *Proceedings of the Conference on Data Compression*, p. 190, 1996.

[15] T. C. Bell and D. Kulp, "Longest-Match String Searching for Ziv-Lempel Compression," *Software - Practice and Experience*, vol. 23, no. 7, pp. 757–771, 1993.

[16] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098–1101, September 1952.

[17] W. Huang, N. Saxena, and E. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[18] K.-J. Lin and C.-W. Wu, "A Low-Power CAM Design for LZ Data Compression," *IEEE Trasactions on Computers*, vol. 49, October 2000.

[19] S.-A. Hwang and C.-W. Wu, "Unified VLSI Systolic Array Design for LZ Data Compression," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 9, August 2001.

[20] W. Huang, N. Saxena, and E. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[21] N. Dandalis and V. Prasanna, "Configuration Compression for FPGA-Based Embedded Systems," *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.

[22] T. Lee and J. Park, "Design and Implemntation of a Static Huffman Encoding Hardware Using a Parallel Shifting Algorithm," *IEEE Transactions on Nuclear Science*, vol. 51, pp. 2073–2080, October 2004.

[23] T. Jahnke, S. Schößler, and K. Sulimma, *Pipeline Huffman Encoder with 10 bit Input, 32 bit Output*. EDA Group of the Department of Computer Science at the University of Frankfurt. `http://www.sulimma.de/prak/ss00/projekte/huffman/Huffman.html`.

[24] S. Mathen, "Wavelet Transform Based Adaptive Image Compression on FPGA," Master's thesis, Unviersity of Calicut, 1996. `http://www.ittc.ku.edu/projects/ACS/documents/sarin_presentation.pdf`.

[25] J. Nikara, *Parallel Huffman Decoder with an Optimize Look Up Table Option on FPGA*. PhD thesis, Tampere University of Technology, 2004.

[26] M. W. E. Jamro and K. Wiatr, "FPGA Implementation of the Dynamic Huffman Encoder," *Proceedings of IFAC Workshop on Programmable Devices and Embedded Systems*, 2006.

[27] S. Sun and S. Lee, "A JPEG Chip for Image Compression and Decompression," *Journal of VLSI Signal Processing Systems*, vol. 35, no. 1, pp. 43–60, 2003.

[28] University of Calgary, *Calgary Text Compression Corpus*. `fttp://ftp.cpcs.ucalgary.ca/pub/projects/text.compression.corpus`.