Design of FPGA-based Accelerators for Deflate Compression and Decompression using High-Level Synthesis

by

Morgan Ledwon

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Integrated Circuits and Systems

Department of Electrical and Computer Engineering

University of Alberta

Abstract

As the volumes of data that are transmitted over the internet continue to increase, data compression is becoming increasingly necessary in order to efficiently utilize fixed data storage space and bandwidth. The Deflate compression algorithm is one of the most widely used lossless data compression algorithms, forming the basis of the .zip and .gzip file formats as well as the Hypertext Transfer Protocol (HTTP). The provision of Field-Programmable Gate Arrays (FPGAs) to implement hardware accelerators alongside conventional Central Processing Unit (CPU) servers in the Internet cloud is becoming increasingly popular. The ability for FPGAs to be rapidly reprogrammed and the inherently parallel resources that they provide makes FPGAs especially well suited for certain cloud-computing applications, like data compression and decompression. High-Level Synthesis (HLS) is a relatively new technology that enables FPGA designs to be specified in a high-level programming language like C or C++, instead of a hardware description language, as done conventionally, and enables designs to be implemented at a faster pace. This thesis examines the design and implementation of FPGA-based accelerators for both Deflate compression and decompression using high-level synthesis.

In Deflate compression, a balance between the resulting compression ratio and the compression throughput needs to be found. Achieving higher compression throughputs typically requires sacrificing some compression ratio. In Deflate decompression, the inherently serial nature of the compressed format makes task-level parallelization difficult without altering the standard format.

In order to maximize the decompression throughput without altering the format, other sources of parallelism need to be found and exploited. Both our compressor and decompressor designs were specified in C++ and synthesized using Vivado HLS for a clock frequency of 250 MHz on a Xilinx Virtex UltraScale+ XCVU3P-FFVC1517 FPGA. Both were tested using the Calgary corpus benchmark files. In the design of the compressor, many different areas of the design that affect the trade-off between compression ratio and compression throughput, such as the hash bank architecture and hash function, are examined. Our implemented compressor design was able to achieve a fixed compression throughput of 4.0 GB/s while achieving a geometric mean compression ratio of 1.92 on the Calgary corpus. In the design of the decompressor, various FPGA hardware resources are utilized in order to increase the amount of exploitable parallelism such that the decompression process can be accelerated. Our decompressor design was able to achieve average input throughputs of 70.73 MB/s and 130.58 MB/s on dynamically and statically compressed files, respectively, while occupying only 2.59% of the Lookup Tables (LUTs) and 2.01% of the Block Random-Access Memories (BRAMs) on the FPGA.

Preface

Chapter 4 of this thesis has been published as M. Ledwon, B.F. Cockburn, and J. Han, "Design and evaluation of an FPGA-based hardware accelerator for Deflate data decompression," in IEEE Canadian Conference of Electrical and Computer Engineering (CCECE), 2019, pp. 195-200. The design work and manuscript composition was done by myself with supervision and manuscript edits contributed by B.F. Cockburn and J. Han.

Acknowledgements

I would like to thank my supervisors, Bruce Cockburn and Jie Han, for giving me the opportunity to work alongside them and for providing guidance during my MSc program. Thank you Bruce for helping me overcome the many challenges and road blocks I encountered during my work and thank you for showing confidence in me even when I wasn't confident in myself.

I would like to thank the members of our industrial collaborator, Eideticom, specifically Roger Bertschmann, Stephen Bates, and Saeed Fouladi Fard, for proposing the idea for this project, for supporting the funding for this project, and for giving me the opportunity to work together with them. Thank you Saeed for the technical advice and assistance throughout the duration of the project.

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for providing grant funding for this project and I would like to thank the Government of Alberta for awarding me three Queen Elizabeth II graduate scholarships.

Contents

List of Tables

List of Figures

List of Abbreviations

ALM Adaptive Logic Module 18, 21, 23, 49, 50

ASCII American Standard Code for Information Interchange 8, 13, 27, 45

ASIC Application-Specific Integrated Circuit 2, 3, 15

BRAM Block Random-Access Memory iii, ix, 22, 23, 29–31, 36, 38, 49–52, 54, 56, 57, 63, 66, 67, 71, 74, 77, 78, 81

CAD Computer-Aided Design 3

CPU Central Processing Unit ii, 2, 3, 15, 16, 18, 21, 23, 24, 26, 31, 35

CTZ Count Trailing Zeroes 39

DMA Direct Memory Access 49, 72

DRAM Dynamic Random-Access Memory 22

EOB End-of-Block 8, 10, 12, 24, 25, 28, 56, 62

FF Flip-Flop 22, 23, 29, 49–51, 71, 74

FIFO First-In, First-Out 29, 56, 57, 60, 61, 64, 66, 71, 75

FPGA Field-Programmable Gate Array ii, iii, viii, 2–4, 15, 16, 18, 19, 21–25, 28–32, 34–36, 48–50, 54, 56, 67, 71, 74, 77, 78, 80–82

FSM Finite-State Machine 61

FVP First Valid Position 39, 40

GPU Graphics Processing Unit 2, 3, 16, 24, 28, 32

HLS High-Level Synthesis ii, 3

HTTP Hypertext Transfer Protocol ii, 5

II Initiation Interval 35, 39, 48, 49, 62, 66, 70, 71, 78, 82

IP Intellectual Property 3, 31, 74, 81

LUT Lookup Table iii, 22, 23, 28–32, 34, 49–51, 62, 71, 74, 77, 81

MUX Multiplexer 23, 38, 39, 76

PWS Parallel Window Size ix, 15, 17–22, 24, 34, 35, 37, 40, 41, 43, 44, 49, 51, 52, 54

ROM Read-Only Memory 16, 20, 23, 47, 62

RTL Register-Transfer Level 48, 83, 84

Chapter 1

Introduction

1.1 Motivation

Data compression is used to save data storage space and to more effectively utilize fixed installed communication bandwidth. In today's information age, the volumes of data that are collected, processed, and then transmitted over the Internet continue to increase. Consequently, efficient software and hardware implementations of data compression and decompression algorithms are becoming increasingly valuable. Higher compression ratios[1] allow more data to be packed into the same storage space. In addition, faster compression and decompression engines facilitate faster data transmission.

There are two main types of data compression: lossy and lossless. In lossless compression, the data must be completely restored from a compressed form to its original content following decompression. Conversely, lossy compression is performed on data where loss of information can be tolerated and the compression process can be accelerated by sacrificing some hopefully less important information from the data. For example, the quality of audio and visual data can be degraded to an acceptable extent using lossy compression algorithms (e.g., MPEG-4 for video compression [1]) in order to reduce the data bandwidth for multimedia streaming when necessary. However, many types of data (e.g., financial data, literature, textual documentation, software code,

[1]The compression ratio is the ratio between the size of the uncompressed data and the size of the compressed data. A higher compression ratio is generally preferred. E.g., a compression ratio of 2.00 means a compressed file is half the size of its uncompressed form.

genomic data) cannot tolerate any loss in fidelity, so they must be compressed losslessly. Because no information loss can be tolerated, lossless compression cannot be accelerated as aggressively as lossy compression.

Deflate [2] is one of the most widely used lossless compression algorithms today, forming the basis for the widely used .zip, .gz, and .png compressed file formats [3]. In Deflate compression, there is an inverse relationship between the compression time and the compression ratio. Typically, more time can be spent compressing data in order to achieve a greater compression ratio (up to a certain point). Conversely, the compression process can be accelerated by performing less aggressive compression, which is more likely to provide a lower compression ratio. A balance needs to be struck when choosing how much time to spend on compression while still making the process worthwhile (by maximizing the compression ratio) and cost competitive (by minimizing the energy and hardware area usage).

In Deflate decompression, the compression ratio of the received compressed data has already been determined by the compressor and thus the compression ratio is no longer a design parameter that can be traded off in order to increase the speed of decompression. The goal in decompression is to restore the original data as quickly as possible with minimized implementation costs. Unfortunately, the inherently serial format of the Deflate standard makes acceleration difficult using parallel processing strategies. As described later, the standard Deflate compressed format can be altered to easily facilitate acceleration through parallel decompression, but this approach requires deviating from a standard that is already widely being used. Other methods of accelerating the decompression process that respect the constraints of the standard are understandably preferred.

Hardware accelerators are computing platforms that can be used to accelerate certain classes of computations beyond the capabilities of software running on conventional computer Central Processing Units (CPUs). Hardware accelerators are typically implemented using moderately parallel multi-core CPUs, massively parallel Graphics Processing Units (GPUs) [4], Field-Programmable Gate Arrays (FPGAs) [5]–[7], and Application-Specific Integrated Circuits

(ASICs). Coinciding with the cloud computing paradigm [8], hardware accelerators are now commonly integrated within cloud servers to accelerate certain algorithms. For example, Amazon Web Services provide access to FPGA-embedded cloud instances (called F1 instances) on a pay-as-you-need basis [9].

There are generally two main methods of accelerating a computation workload [10]: by increasing the clock frequency or by performing more operations of the workload in parallel per clock cycle. Due to the breakdown of Dennard scaling [11], clock frequencies can no longer be substantially scaled up anymore without exceeding practical power and thermal limitations, leaving parallelization as the main solution for accelerating computations [10]. FPGAs and GPUs have massively parallel architectures that often enable them to exploit parallelism more effectively and directly than is possible with CPUs. ASIC designs can be fully customized to achieve the maximum performance for a desired algorithm, but they come at the cost of longer design time, greater engineering risk, high mask costs and, therefore, higher implementation costs.

FPGAs have many advantages. They can be quickly reprogrammed such that accelerator designs can be altered or changed easily as needed. The flexible programmable fabric of hardware resources in an FPGA allows various low-level optimizations to be made to designs, for example, using minimized precision bit-widths (i.e., integer bit widths of any size, not just of powers of two) for storage and computation to save time, area, and power. FPGA-based designs can be developed, verified, and encapsulated as Intellectual Property (IP) blocks that can be later instantiated and easily interfaced together, giving the ability to create modular pipelined designs that can be optimized for maximum throughput.

FPGA designs are typically expressed in a hardware description language like VHDL [12] or Verilog [13]. A relatively new Computer-Aided Design (CAD) technology called "high-level synthesis" (HLS) aims to make FPGA design quicker and easier by allowing designers to specify the required system behaviour in a higher-level language like C or C++, which is then synthesized and optimized automatically into a hardware design (i.e., an FPGA configuration). Although the idea of high-level synthesis was proposed over 30 years ago, it wasn't until recently that the quality of results provided by high-level synthesis solutions became good enough for the technology to become viable [14]. Vivado HLS [15] is the name of the high-level synthesis tool released by Xilinx in 2013 for synthesizing designs for use on Xilinx FPGAs.

This thesis investigates the challenges associated with optimizing and accelerating both Deflate compression and decompression using custom accelerator hardware configurations on FPGAs. Our goal for compression is to design an accelerator that can provide competitive compression throughputs (on the order of multiple GB/s) while maintaining sufficient compression ratios (above 2.00). Our goal for decompression is to design an accelerator that achieves the highest throughputs possible without making any assumptions about the compressed data or making any changes to the standard compressed Deflate format. We aim to do this using an FPGA as the hardware accelerator platform and high-level synthesis as the means of creating the designs, specifically targeting Xilinx FPGAs and using the Vivado HLS tool.

1.2 Outline

The outline of the thesis is as follows: Section 2.1 in Chapter 2 provides background information on the Deflate compression algorithm and format. Sections 2.3.1 and 2.3.2 describe other works related to compression and decompression, respectively. The design of our compressor is described in Chapter 3 and the design of our decompressor is described in Chapter 4. Finally, the results of our work are summarized and suggestions for future work are given in Chapter 5.

Chapter 2

Background

2.1 The Deflate Lossless Compression Algorithm

Deflate is a lossless compression algorithm (and resulting data format) [2] that forms the basis for many widely used file formats. It was originally created by Philip Katz to be used in his software application PKZIP [16] and was shared in the public domain. The Deflate format was later utilized by Jean-loup Gailly and Mark Adler in their open-source software application, gzip [17], and the software library, zlib [18], which was created in order to facilitate the usage of the PNG lossless compressed image format. As a result, Deflate is now commonly used by many different compression and decompression applications [19]. The most notable of these applications include the Hypertext Transfer Protocol (HTTP) standard as well as the aforementioned .zip [20], .gz [21], and .png [22] file formats. Each of these compressed file formats includes a header and footer wrapper that encloses the binary Deflate compressed data payload. Before the Deflate file format is given, we will first describe the basic compression algorithm. Some aspects of the Deflate compression algorithm may vary depending on the implementation, as long as the resulting compressed data adheres to the format described in [2].

2.1.1 Algorithm Details

The Deflate compression algorithm is actually the concatenation of two other well-known compression algorithms: it performs 1) byte-level compression us-

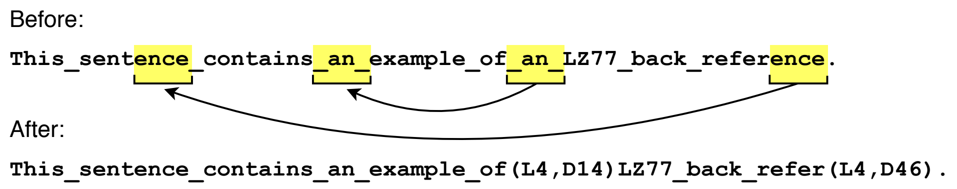
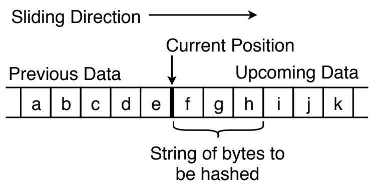

Before:
This_sentence_contains_an_example_of_an_LZ77_back_reference.
After:
This_sentence_contains_an_example_of(L4,D14)LZ77_back_refer(L4,D46).

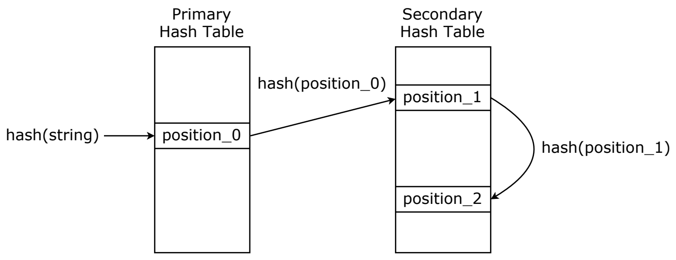
Figure 2.1: An example of LZ77 compression. Note: the physical representation of the length-distance pairs is not defined in the Deflate specification as they will eventually be replaced with Huffman codes. How the length-distance pairs are represented in between the LZ77 encoding and the Huffman encoding is left up to the compressor implementation.

ing a variation of LZ77 encoding [23] followed by 2) bit-level compression using Huffman encoding [24]. Since the LZ77 encoding format is patented, Deflate describes a similar but more general algorithm for replacing duplicated strings. A data file to be compressed is scanned and repeating strings of characters are replaced with length-distance pairs, as shown in Fig. 2.1. As each byte of the file is read, it is recorded in a history buffer and any potential previous matches are sought in that buffer. The length-distance pairs correspond to the length of a match and the distance back within the history buffer that it occurred. In the Deflate format, matching strings of lengths of up to 258 bytes long and match distances of up to 32,768 bytes are allowed. It is desirable to match the longest possible string in the input to a previously stored string in order to compress the greatest number of bytes into a single length-distance pair.

The Deflate specification recommends using chained hash tables to find potential matching strings, though any match-finding method can be used. Typically, a fixed-width sliding window is used, which slides across the sequence of input bytes, as shown in Fig. 2.2a. A string of bytes, the number of which is chosen by the compressor, are taken from the input sequence, hashed (i.e., compressed into a fixed-length hash signature), and looked up in a primary hash table, which contains the positions of all of the most recently hashed and stored strings. If more than one string maps to the same hash signature, the table will contain a link chaining from that signature's position in the primary hash table to a secondary hash table. A chain of such links can be traversed to find all the positions of matching strings that have the same hash value,



(a) Sliding Window



(b) Chained Hash Tables

Figure 2.2: A standard approach to performing string matching is using a sliding window and chained hash tables.

as shown in Fig. 2.2b. These potential matches are then compared to the current input sequence and the longest match is used to replace those input bytes with a length-distance pair. In order to achieve greater compression, more time can be spent searching for the longest possible match. Limiting the search for a match will save time but may lower the obtained compression ratio. A compressor that is focused on maximizing the compression ratio will thus, in general, need to spend more time performing the LZ77 matching step.

After LZ77 encoding, the data is split into blocks for Huffman encoding. The sequence of unmatched literals and length-distance pairs in each block, which are both referred to as *symbols*, will be encoded by replacing each symbol with a variable-length codeword. Huffman codes[1] are a type of *prefix code*, which means that the code for one symbol will never be the prefix of another code. In Deflate, the Huffman codes used are also *canonical*, which means that all of the codes of the same length have consecutive binary values. This means that the Huffman code tables can be recreated using only the sequence of code lengths [25]. There are two different Huffman encoding methods used by Deflate: *static* (or fixed) and *dynamic*. In static encoding, the Huffman codes used are all from pre-defined tables that are listed in the Deflate specifications [2]. These tables are not included with the compressed data. In dynamic encoding, the Huffman codes are created based on the frequencies of the symbols in each block (measured during the preceding LZ77 encod-

[1]Following the convention used in the Deflate literature, the *codewords* in a Huffman code will themselves be referred to as Huffman *codes*.

ing stage). The most frequently occurring symbols are typically assigned the shortest Huffman codes in order to achieve the greatest amount of compression. Each dynamically encoded block will have a unique set of dynamic codes that are optimized for the symbols in that block. The dynamic code tables for each dynamic block are encoded along with the compressed data and are used by the decompressor to decode the data during decompression. Though considerably more complicated, encoding Deflate blocks using dynamic Huffman codes instead of static codes allows greater compression ratios to be achieved.

There are actually two Huffman code tables used for each block: one for literals and lengths (shown in Table 2.1), and the other for distances (shown in Table 2.3). This can be done because a distance code will only ever follow a length code, which allows the same binary codes (with different meanings) to be re-used between the two tables. There is also one special symbol, called the *End-of-Block (EOB)* marker, that is included in the literal/length Huffman table. Symbols 0 to 255 directly correspond to the American Standard Code for Information Interchange (ASCII) literal values 0 to 255, symbol 256 is the EOB marker, and symbols 257 to 287 are the length symbols, which are described in Table 2.2. Note that length symbols 286 and 287 are not actually used for encoding but are, rather, used in the construction of the Huffman code. The distance table (Table 2.3) contains 30 different distance symbols from 0 to 29. Each length and distance symbol corresponds to a base length or distance value and may be followed by a number of extra bits. These extra bits correspond to a binary offset that is added to the base length or distance value after decoding.

In Deflate static Huffman encoding, the literal/length codes are from 7 to 9 bits long (shown in Table 2.1) and the distance codes are fixed-length 5-bit codes that are just the binary values of their symbols (e.g., the static code for distance symbol 5 is 0b00101). In dynamic encoding, both the literal/length and distance codes can be from 1 to 15 bits long. In both forms of encoding, static and dynamic, the length codes may be followed by 0 to 5 extra bits (as shown in Table 2.2) and distance codes may be followed by 0 to 13 extra bits (as shown in Table 2.3).

Table 2.1: Deflate Literal/Length Symbol Table and Corresponding Static Codes [2]

Symbol Type	Symbol Decimal Values	Static Code Lengths	Static Binary Codes
Literal	0 - 143	8 bits	00110000 - 10111111
	144 - 255	9 bits	110010000 - 111111111
End-of-block	256	7 bits	0000000
Length	257 - 279	7 bits	0000001 - 0010111
	280 - 287	8 bits	11000000 - 11000111

Table 2.2: Deflate Length Symbol Definitions Table [2]

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

Table 2.3: Deflate Distance Symbol Definitions Table [2]

Distance Symbol	Distance Values	Extra Bits	Distance Symbol	Distance Values	Extra Bits	Distance Symbol	Distance Values	Extra Bits
0	1	0	10	33 - 48	4	20	1025 - 1536	9
1	2	0	11	49 - 64	4	21	1537 - 2048	9
2	3	0	12	65 - 96	5	22	2049 - 3072	10
3	4	0	13	97 - 128	5	23	3073 - 4096	10
4	5, 6	1	14	129 - 192	6	24	4097 - 6144	11
5	7, 8	1	15	193 - 256	6	25	6145 - 8192	11
6	9, 10	2	16	257 - 384	7	26	8193 - 12288	12
7	13 - 16	2	17	385 - 512	7	27	12289 - 16384	12
8	17 - 24	3	18	513 - 768	8	28	16385 - 24576	13
9	25 - 32	3	19	769 - 1024	8	29	24577 - 32768	13

2.1.2 Dynamic Table Construction

As mentioned above, the dynamic Huffman codes used are unique to each dynamic block, so the dynamic code tables need to be included in the compressed data along with each block. In order to include the tables while ensuring that they take up as little space as possible, they are compressed using the following method. Since the Huffman code tables are canonical, they can be recreated using only the sequence of code lengths in the table. An algorithm for constructing a Huffman table using a sequence of code lengths as well as an example are given in Section 3.2.2 of [2]. In Deflate, this sequence follows the order of the literal/length symbol values from 0 to 287. For example, the first code length in the sequence will be for symbol 0, followed by the code length for symbol 1, etc. This sequence is immediately followed by the sequence of code lengths for the distance symbols 0 to 29. Any unused symbols are given code lengths of 0 and if they come at the end of the sequence, they are deleted from the sequence in order to save space. At a minimum, there will be 257 literal/length code lengths in the sequence (for the 256 literals plus the EOB symbol) and 1 distance code length (indicating that no length-distance pairs exist in the data). Once the full sequence of code lengths is input and the number of codes of each length is counted, the codes can be reconstructed starting from the smallest code length.

The code length sequence used to reconstruct the Huffman tables is also compressed using a combination of run-length encoding and Huffman encoding. A 19-symbol Huffman code table, shown in Table 2.4, is used to encode the code length sequence. In this table, symbols 0 to 15 correspond to code lengths 0 to 15, while symbols 16, 17, and 18 are used to encode runs of re-

Table 2.4: Deflate Code Length Symbol Definitions Table [2]

Code Length Symbol	Meaning
0 - 15	Code lengths 0 - 15
16	Copy previous code length 3 - 6 times (2 extra bits).
17	Repeat code length of 0, 3 - 10 times (3 extra bits).
18	Repeat code length of 0, 11 - 138 times (7 extra bits).

peating code lengths. Again, unique codes are created for these symbols and so another sequence of code lengths must be included in order to reconstruct this table. This sequence of “code length” code lengths is included in the compressed block in the following fixed order: 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15. The first four code lengths are always included in this sequence while the rest appear in the order of the likelihood that they will be used. Code lengths for the symbols 8, 7, and 9 come first as they are the most likely code lengths to be used while code lengths of 1 and 15 are relatively rare and so they appear at the end of the sequence. As done before with the literal/length and distance code length sequence, any unused code lengths at the end of the sequence are removed. The process of reconstructing the Huffman tables for a dynamic data block is described step-by-step as follows:

1. The number of literal/length codes, distance codes, and code length codes is read from the compressed block.

2. The sequence of “code length” code lengths is read and the number of codes of each length is counted.

3. The “code length” code length count is used to construct the code length Huffman table.

4. This code length Huffman table is used to read and decode the sequence of code lengths for the literal/length and distance code tables.

5. The literal/length and distance code length counts are used to construct the literal/length and distance Huffman tables.

2.1.3 Compressed Format Details

As described above, a file compressed using Deflate is split into blocks of arbitrary size which are then each Huffman encoded individually. None of the binary data within a static or dynamic Deflate block is guaranteed to be byte-aligned. In a dynamic block, the last bit of the code length sequence is immediately followed by the first bit of the first Huffman code in the data.

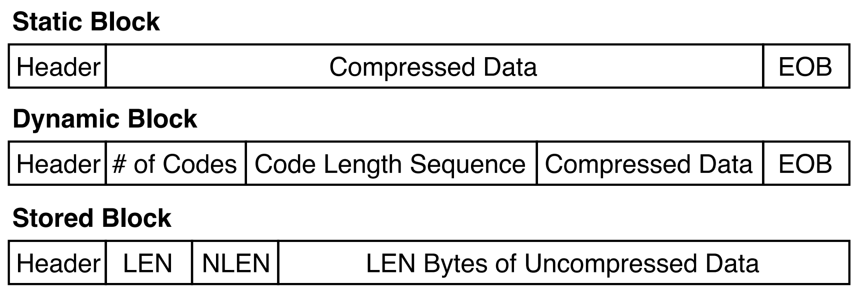


Figure 2.3: Deflate block types and their contents (not to scale).

Since the codes are variable-length, each code must be decoded one at a time, bit-by-bit, starting with the first Huffman code. The length of a Deflate block is unknown until the EOB code has been found and decoded. The beginning of the next block begins immediately after the EOB code. Besides static and dynamic blocks, a third type of block exists that is called a *stored* block, which contains uncompressed data. Stored blocks are typically used in order to avoid attempting to compress data which cannot be usefully compressed further, for example, data which has already been compressed, in order to prevent expanding the data[2]. Figure 2.3 shows the structures of the three different block types. Each block begins with a 3-bit header that is composed of two flags: BFINAL (1 bit), which indicates whether a block is the last block in the compressed file, and BTYPE (2 bits), which describes the type of block. The BTYPE value is 0b00 for stored blocks, 0b01 for static blocks, and 0b10 for dynamic blocks. The BTYPE value of 0b11 is unused and indicates an erroneous block.

There are a few subtle but critical pieces of information about how the compressed data is actually stored. In the Deflate format, all of the data elements (i.e., Huffman codes, block headers, code lengths, etc.) are packed into bytes starting with the least significant bit of the byte. All of these elements, except for Huffman codes, are packed with their least significant bit first while the Huffman codes are packed with their most significant bit first. This means that the Huffman codes are effectively reversed in bit order while

[2]The amount of expansion that can occur will depend on the compressor implementation. In the worst case scenario, zlib adds 5 bytes per 16 kB stored block (an overhead of about 0.03%). The zlib wrapper also adds 6 bytes of overhead to the file [18].

all other data elements appear in the correct bit order (i.e., most significant bit to least significant bit). Each byte of compressed Deflate data is meant to be processed from right to left with new bytes being appended to the left as they are read.

2.2 The Calgary Corpus Benchmark

The Calgary corpus [26] is a dataset composed of 14 files of various data types with different sizes that has been commonly used as a benchmark in compression and decompression research. This benchmark has been used by many other researchers whose work is described in Section 2.3. For purposes of comparison, we also use the Calgary corpus to measure the performance of both our compressor and decompressor designs. Table 2.5 describes the 14 different Calgary corpus files. The type of data contained within a file plays a key factor in the compression ratios achieved as well as the speed of compression and decompression.

As mentioned above, dynamic Huffman blocks can be used to get higher compression ratios than static blocks. This makes sense because the dynamic blocks use Huffman codes that are tailored to the statistics of each block while static blocks use standard Huffman codes that are a compromise for all types of data files. Figure 2.4 shows the difference in compression size of the Cal-

Table 2.5: The Calgary Corpus Benchmark Files [26]

File Name	Size (bytes)	Description
bib	111,261	Bibliography containing ASCII text in Unix “refer” format
book1	768,771	Fiction book containing unformatted ASCII text
book2	610,856	Non-fiction book containing ASCII text in Unix “troff” format
geo	102,400	Geophysical seismic data in 32-bit number format
news	377,109	USENET batch file containing ASCII text
obj1	21,504	Object code executable of file “progp”
obj2	246,814	Object code executable of Macintosh program
paper1	53,161	Technical paper containing ASCII text in Unix “troff” format
paper2	82,199	Technical paper containing ASCII text in Unix “troff” format
pic	513,216	1728x2376 bitmap image
progc	39,611	Source code file written in C
progl	71,646	Source code file written in LISP
progp	49,379	Source code file written in Pascal
trans	93,695	Transcript of terminal session

gary corpus files when compressed using static codes versus dynamic codes. These numbers were obtained by compressing the files using zlib with the default settings[3]. The arithmetic mean (average) compression ratio across the dynamically compressed corpus is 3.51 and the geometric mean is 3.19. For the statically compressed files, the arithmetic mean is 3.03 and the geometric mean is 2.76. Both means are described here because both figures of merit are reported by the compression related works described in Section 2.3.1. Compared to the arithmetic mean, the geometric mean places more weight on each value averaged and will therefore always be lower. This makes it more useful when averaging numbers with widely different values (like the very large compression ratio of the "pic" file and the very small compression ratio of the "geo" file). While the arithmetic mean is obtained by dividing the sum of a set of n values like so:

$$\frac{1}{n}\sum_{i=1}^{n} a_i = \frac{a_1 + a_2 + \cdots + a_n}{n}$$

the geometric mean is obtained by taking the nth root of the product of n values as follows:

$$\left(\prod_{i=1}^{n} a_i\right)^{\frac{1}{n}} = \sqrt[n]{a_1 a_2 \ldots a_n}$$

By using dynamically created Huffman codes instead of static codes, the compression ratios are improved by an average of 15.7%. However, using dynamic Huffman encoding requires more time to perform both compression and decompression. During compression, the symbol frequencies need to be measured and the code tables need to be constructed before encoding. In decompression, the compressed code length sequence needs to be read and used to reconstruct the code tables. When using static codes, encoding/decoding can begin immediately using the fixed standard code tables.

[3]The zlib compression settings include compression effort (the amount of the time to spend searching for matches), history buffer size, sliding window size, and compression strategy (Huffman encoding only without LZ77 encoding, static Huffman encoding only without dynamic, etc.) among other things.

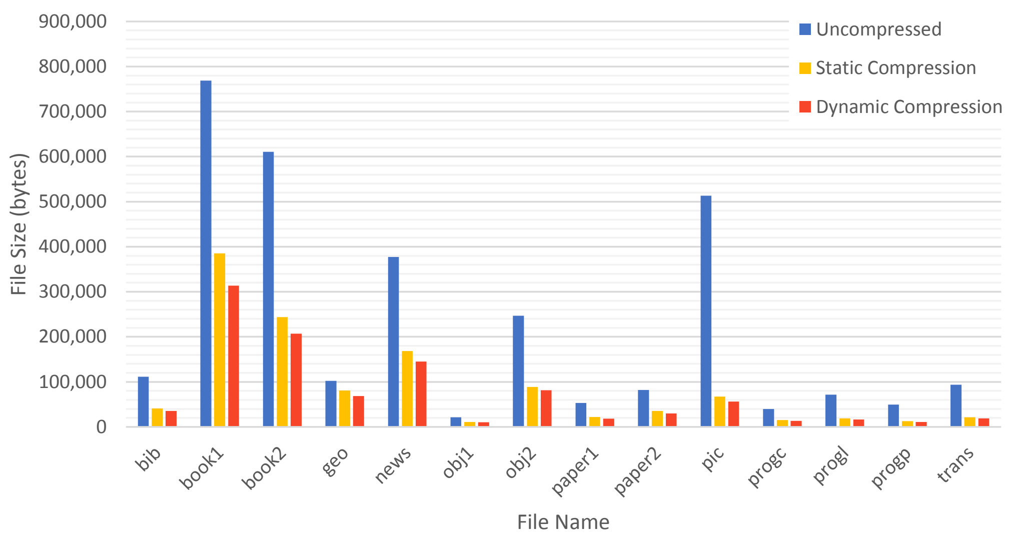


Figure 2.4: Compression of the Calgary corpus using zlib with the default compression settings.

2.3 Literature Review

2.3.1 Works Related to Compression

In this section, a few of the most recent and most significant published designs of FPGA-based Deflate compression accelerators are described. Many compression accelerators have been developed over the years for a variety of hardware platforms, including CPUs and ASICs, but only FPGA designs are analyzed in detail here since FPGA-based accelerators are the main focus of this thesis. These designs share a similar architecture composed of a constant-throughput pipeline that compresses a fixed number of bytes in every clock cycle using a parallel sliding window. The size of this window is often referred to as the Parallel Window Size (PWS). The main focus of these designs appears to be on the LZ77 encoding portion, which is the more complicated of the two stages of Deflate compression. Instead of spending time searching for matches, only one clock cycle is spent obtaining potential matching strings and comparing them to the strings in the sliding window. Afterwards, only the best matches that are found within the window constraint are kept. Multiple potential matches are looked up and compared simultaneously with multiple substrings within the sliding window in every cycle in order to quickly compress the data at a constant rate. How the data history is stored and how potential matches are found will greatly impact the performance and area of the design. Most seemingly throughput-focused designs perform the Huffman encoding using static codes only. This restriction greatly simplifies the interface between the LZ77 encoder and the Huffman encoder as the data can be encoded as soon as it is received using the fixed standard code tables stored in Read-Only Memories (ROMs). Although these FPGA-based compressor designs typically provide lower compression ratios than are possible with software compressors, they are able to compress data at a much faster rate. The FPGA-based compressor designs examined are described chronologically below. The design of our compressor, described in Chapter 3, was influenced by each of these designs.

The authors of [27] designed an FPGA-based accelerator for Deflate compression using OpenCL and Intel/Altera's OpenCL compiler. OpenCL is a high-level C-like programming language that can be used to specify hardware acceleration kernels for parallel computing platforms that include CPUs, GPUs, and/or FPGAs. Their design appears to be heavily based on the Verilog architecture described in [28] by researchers at IBM. Reference [28] is unfortunately no longer available for access. In traditional LZ77 encoder designs, the history dictionary, which contains all of the previous input bytes arranged in chronological order, and the hash table, which contains the positions of previously hashed strings, occupy two separate memories. Multiple single-byte reads may be required to read an entire string from the history dictionary unless the memory is partitioned to allow for more read ports that can be accessed in parallel. Multiple copies of the history dictionary can be used to allow conflict-free simultaneous lookup of multiple strings, however, hash collisions (i.e., when two or more strings produce the same hash signature) can still occur into the hash table and these collisions will reduce the number of positions that can be read and written, resulting in potential matches being missed and thus leading to a reduction of the compression ratio. In the design in [27], the history dictionary and the hash table are combined into the same memory. This memory stores entire strings (instead of individual bytes) as

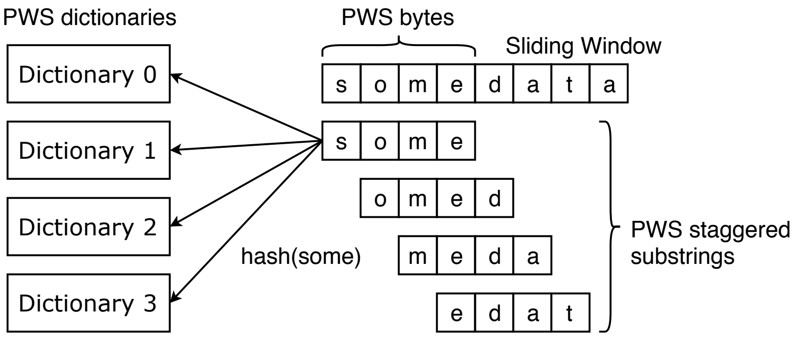


Figure 2.5: The LZ77 sliding window architecture from [27] with a PWS of 4. Every dictionary is accessed simultaneously by every one of the substrings.

well as their positions. When the hash signature of an input string is given to the memory, a similar string with the same hash signature is returned. These returned strings are then compared byte-by-byte with the substrings within the sliding window in order to find matches.

Fig. 2.5 shows the sliding window architecture from [27] with a PWS of 4 bytes, though in the actual design, a PWS of 16 was used. In every clock cycle, 16 bytes are read from the input stream and are shifted into a 32-byte wide sliding window (twice the length of the PWS). Each byte in the first half of the 32-byte window is the start of a 16-byte substring, with a total of 16 staggered substrings occupying the entire window. For each 16-byte substring in the 32-byte window, a hash value is calculated in parallel and then looked up in 16 different dictionaries. Sixteen 16-byte potential matches are obtained from each dictionary for each substring for a total of 256 potential matches. The 16 different history dictionaries are replicated 16 times so that each one has 16 read ports and all 16 substring reads can occur simultaneously without the possibility of hash conflicts. Each substring in the parallel window is then written to one of the 16 dictionaries. The dictionaries are indexed using a 10-bit hash signature and have 1024 indexes each, which each store a 16-byte string and its position. The 256 potential matches are compared to the 16 window substrings and the longest match for each substring is kept. Since the substrings and the potential matches are 16 bytes long, the longest possible match length is 16 bytes. The 16 best matches are then filtered using a series of filters. Matches that are overlapped by matches from the last iteration and matches that overlap each other in the current iteration are filtered away using last-fit heuristics. The matches in the window are replaced with markers and are then Huffman encoded. In this architecture, the Huffman trees are created by software executing on a CPU and are then passed to the FPGA for encoding. Whether only static or dynamic Huffman encoding is used is not disclosed. The symbols are encoded and a series of barrel shifters are used to align the variable-length codes.

Throughout the paper the authors describe certain changes that they made to the OpenCL code in order to optimize it for running on FPGAs. Though the code may be functionally equivalent, certain OpenCL syntaxes may lead to inefficient hardware synthesis. For example, long adder chains, which tend to use more logic area and have greater delay, may be selected by the compiler instead of faster balanced adder trees. These optimization techniques are also applicable when using Vivado HLS to synthesize C/C++ as done in our research. Like in [28], which was the basis of their design, they also mapped their design for an Intel/Altera Stratix V A7 model FPGA. For comparison, the design in [28] had a clock frequency of 200 MHz and an input throughput of 3.0 GB/s. In [27], although they report that their design processes 16 bytes every clock cycle, they achieved an input throughput of 2.84 GB/s with a clock frequency of 193 MHz, which seems to imply that their final design actually used a PWS of 15 bytes, not 16, since a design that processes 16 bytes/cycle at 193 MHz should have a throughput of 3.088 GB/s. Their design has a depth of 87 stages compared to the 17-stage design in [28] and uses more chip area. Reference [27]'s design uses 47% of the logic and 70% of the RAM on the StratixV A7 chip, which one could reasonably interpret to be 110,318 of the 234,720 ALMs[4] and 1,792 of the 2,560 M20K memory blocks, while the design from [28] only uses 45% of the logic (105,624 ALMs) and 45% of the RAM (1,152 M20Ks). The advantage of [27]'s design comes from the enhanced productivity that comes from using OpenCL instead of Verilog, though it comes at the cost of less-efficient area usage. They evaluated their design

[4]Adaptive Logic Modules (ALMs) are the standard logic cell used in Intel/Altera FPGAs.

using the Calgary corpus and achieved a geometric mean compression ratio of 2.17 across the entire corpus, which is the highest value of the compressor designs compared here. This may be because they use software-created dynamic Huffman tables, while the other designs only utilize static Huffman encoding. It may also be because of the fully connected hash dictionary architecture that they used, which is not susceptible to hash collisions.

Reference [5] describes an FPGA-based compressor design that was created by researchers from Microsoft in response to [27] and [28]. Their design was programmed in SystemVerilog also for use with an Intel/Altera Stratix V FPGA. They use a pipelined parallel window architecture that is scalable to other sizes. In their design, they use the traditional method of separately storing the string match positions in a hash table and the data history in a dictionary. In order to reduce hash table lookup conflicts, the hash table is split into multiple banks, where each bank contains a subset of the possible hash indexes; using more banks reduces the likelihood of two hashed values accessing the same bank. In their design, they use 32 hash banks to index 65,536 hash values (2,048 indexes per bank). These hash banks are also run at double the clock frequency that is used by the rest of the system to allow two reads and two writes to be made to each bank in every clock cycle. This allows two hash values to access the same bank without conflict; any further accesses must be dropped, however, which negatively impacts the compression ratio. In order to store multiple positions for each match, each hash bank can also be split into multiple depth levels with each containing a different previous position. When a hash bank is accessed, all stored previous positions are read simultaneously, the oldest position is shifted out, and the latest input position is shifted in. This technique of hash bank depth levels is described but does not appear to have been used in the final design by [5].

A block diagram of the scalable pipeline architecture from [5] is shown in Fig. 2.6. Although their design is scalable in terms of window size, for illustration, a design with a PWS of 16 bytes will be described. In every cycle, up to 16 positions are obtained from the hash tables and are simultaneously looked up in 16 copies of the history dictionary. Each dictionary copy is partitioned

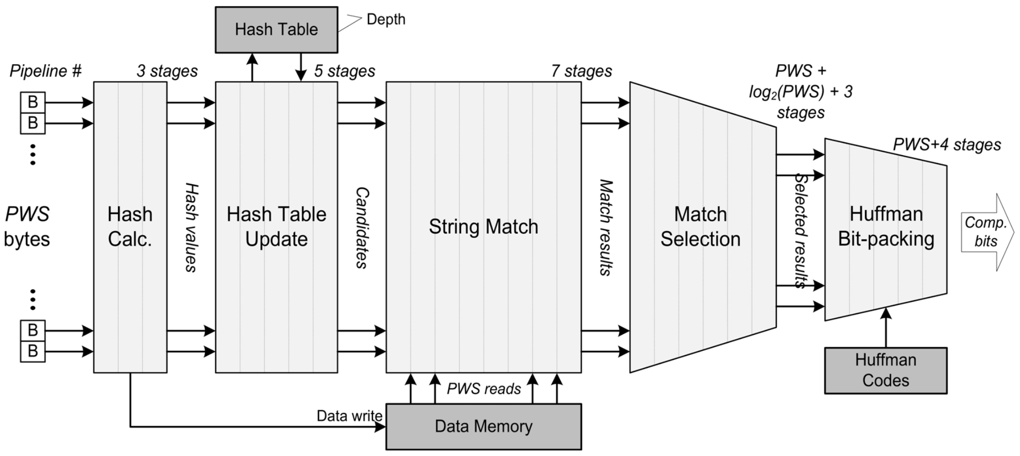
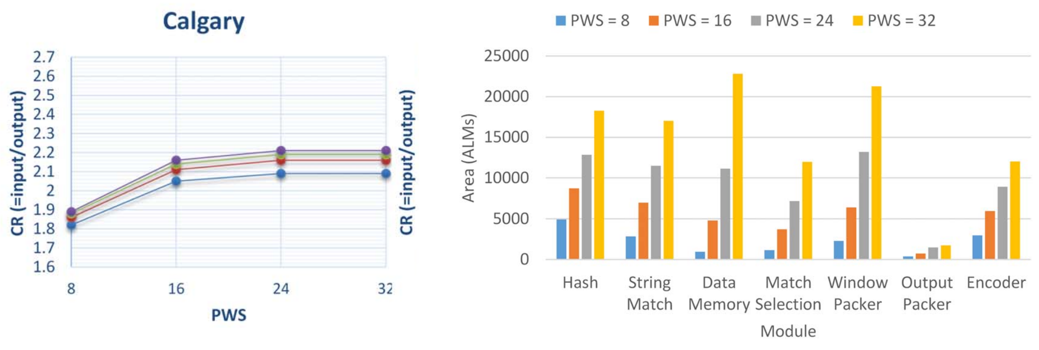


Figure 2.6: The scalable compressor pipeline architecture from [5].

to have 16 read ports so that 16 consecutive bytes can be read simultaneously (the history dictionaries store individual bytes, not entire strings). Sixteen 16-byte strings are returned and compared to the substrings in the input window. A series of match selection stages iteratively filter out conflicting matches in a similar manner as was done in [27]. The selected matches for the window are then Huffman-encoded across multiple pipeline stages. In order to maintain the same steady throughput on both the LZ77 encoding and Huffman encoding portions of the pipeline, only static Huffman codes are used. As LZ77 encoded windows are written out, their contents are passed through to static codebook ROMs to be immediately encoded. If dynamic codes were to be used, the Huffman encoding portion would not be able to begin until the first block from the LZ77 encoding portion was finished and the dynamic code tables were assembled. This would also require an intermediate storage space between the two Deflate stages for holding the LZ77 encoded data (this approach is investigated in [29]). In each stage of the Huffman encoder, one of the window boxes is encoded and then shifted and ORed into an encoded window using barrel shifters. A double buffer, which is located at the output of the pipeline, collects the encoded windows. When the double buffer is over half full, the lower half writes its data to the output.

The authors of [5] synthesized their design with various PWS values from 8 to 32 bytes and obtained performance results for each. With a PWS of 16



(a) (b)

Figure 2.7: The effect of varying the PWS on a) the compression ratio on the Calgary corpus and b) the hardware area cost (Figures from [5]).

bytes, their design can compress at an input throughput of 2.8 GB/s with a clock frequency of 175 MHz. The pipeline has 58 stages and it achieves a mean compression ratio of 2.05 on the Calgary corpus. It is unspecified whether this is an arithmetic mean or geometric mean. While the throughput of their design is slightly less than that of [27] and [28] for a 16-byte PWS, their design is much more area-efficient. It uses 39,078 ALMs, which is about one third of the area of the other two designs. Unfortunately, the number of memory blocks used is not disclosed. Their architecture is also designed to be easily scalable to various window widths. As they found from their results, shown in Fig. 2.7, the compression ratio increases logarithmically with the window width while the area for most modules increases quadratically, meaning that diminishing returns are obtained when scaling the window width increasingly high. A 32-byte PWS design has an input throughput of 5.6 GB/s and average compression ratio of 2.09 but uses 2.8 times more chip area than the 16-byte PWS design.

In [6], which focuses on task co-scheduling between CPUs and FPGAs, a compression accelerator is developed using Vivado HLS. Since the focus of paper is not on accelerating compression, minimal details on their compressor design are shared and their results are not very competitive. They based their design on [27] and [5] and used a similar 16-byte PWS pipeline. They used a Xilinx Kintex UltraScale KU115 FPGA with a clock frequency of 200 MHz and achieved a peak bandwidth of 2.8 GB/s. This is less than the expected bandwidth of 3.2 GB/s because of pipeline bubbles (i.e., unused clock cycles) caused by accessing the FPGA external Dynamic Random-Access Memory (DRAM). When compressing the Calgary corpus, they achieved a geometric mean compression ratio of 1.73. Their design uses 12.6% of the Lookup Tables (LUTs) and 4.9% of the Flip-Flops (FFs) on the KU115 chip, which is 83,583 of the 663,360 LUTs and 65,009 of the 1,326,720 FFs. The number of Block Random-Access Memories (BRAMs) used is not disclosed.

As reported in [5], single-engine compressor designs start to become unscalable for very wide PWS values. Consequently, the authors of [7] focused on using multiple compressor engines in parallel in order to improve the compression throughput while maintaining area efficiency. By increasing the number of compressors instead of increasing the PWS, the throughput can be increased while the area usage scales linearly instead of quadratically. This comes at the cost of reduced compression ratios as multiple compressors cannot compress as efficiently as a single compressor due to match opportunities being lost. As also discovered in [5], the compression ratio increases minimally with a wider PWS. This disappointing improvement is because matches of length 24 and 32 are, in practise, rarely encountered in most data files. For these reasons, the authors of [7] chose to use a PWS of 16 bytes for their compressors.

Along with designing a multi-engine compression system, the authors of [7] also made a few improvements over the single compressor design from [5]. Instead of running the hash banks at double the system clock frequency, which limits the overall system clock frequency, they chose to run them using the same clock frequency in order to achieve a higher overall clock frequency. They also use the hash bank architecture with multiple depth levels to store and compare with more previous matches. They use a combination of the banked hash table design from [5] and the combined hash table dictionary from [27]. Their compressor design uses one hash dictionary containing both strings and positions that is split into 32 banks, with each bank having 3 depth levels. At every clock cycle, the 16 substrings in the input window are hashed and will attempt to access one of the 32 banks. Each accessed bank returns a string and position from each of the 3 depth levels. Instead of mapping the strings read from each bank back to the substring that accessed it for comparison, which would require 16-byte wide Multiplexers (MUXs), they perform the comparisons at the output of each bank before mapping the best match length to each substring, which only requires 5-bit wide MUXs (as the lengths of the matches can only be from 0 to 16). The comparisons can be performed at each bank with the input substrings as they are written to the bank since the read takes place one clock cycle before the write operation. This MUX optimization helps to reduce the area consumption and improve the clock timing. As with previous designs, they perform the Huffman encoding portion using static codes only and using a series of pipeline stages composed of ROM encoders and shift-OR operations.

Each compressor has a clock frequency of 200 MHz and can compress 3.2 GB/s. One compressor engine on its own occupies 38,297 ALMs and is able to achieve an average compression ratio of 2.10 on the Calgary corpus. For comparison with previous works, [27] and [5], they implement a multi-engine compressor on a Stratix V FPGA, which has enough area to fit three compression engines on it. This provides a collective input compression throughput of 9.6 GB/s and an average compression ratio of 2.05. They also implement a four-engine compressor design on a larger Intel Arria 10 FPGA capable of compressing 12.8 GB/s with a compression ratio of 2.03. They found, however, that the CPU-FPGA interface ends up limiting these throughputs to practical values of 3.9 GB/s for the three-engine compressor and 10.0 GB/s for the four-engine compressor.

A summary of all the previous works described is shown in Table 2.6. For

Table 2.6: Past Hardware-Accelerated Deflate Compressors

Reference	FPGA Platform	Language	Clock Frequency	Input Throughput	Area Utilization	Compression Ratio
[28] (2013)	Stratix V A7	Verilog	200 MHz	3.00 GB/s	105,624 ALMs, 1,152 M20Ks	?
[27] (2014)	Stratix V A7	OpenCL	193 MHz	2.84 GB/s	110,318 ALMs, 1,792 M20Ks	(Geo.) 2.17
[5] (2015)	Stratix V	SystemVerilog	175 MHz	2.80 GB/s	39,078 ALMs, ? M20Ks	2.05
[6] (2018)	KU115	C/C++ (Vivado HLS)	200 MHz	2.80 GB/s	83,583 LUTs, 65,009 FFs, ? BRAM18Ks	(Geo.) 1.73
[7] (2018)	Stratix V	?	200 MHz	3.20 GB/s	38,297 ALMs, ? M20Ks	2.10

a fair comparison, the results shown for [5] are for a PWS of 16 bytes and the results shown for [7] are for one compressor engine. The geometric mean compression ratios are identified in the table while the remaining means are uncertain to be geometric or arithmetic. As seen from Table 2.6, although they provide lower compression ratios than software compressors (from Section 2.2: zlib achieves geometric mean compression ratios of 3.19 and 2.76 using dynamic and static compression, respectively), FPGA-based compression accelerators can achieve throughputs on the order of multiple GB/s. This is orders of magnitude faster than software implementations, which can typically compress around 50 MB/s[5], with some implementations reaching over 300 MB/s [30][31]. GPU-based compression accelerators seem to be rarer but also perform on the scale of hundreds of MB/s [32]. Of the above described FPGA-based compressor designs, the one from [27] stands out for achieving the highest possible compression ratios using a fully connected hash dictionary design, while the design from [7] appears to be the fastest and most scalable single-core compressor design. These two designs, in particular, influenced our design the most.

2.3.2 Works Related to Decompression

To perform Deflate decompression (sometimes referred to as Inflate), the two main encoding stages, Huffman encoding and LZ77 encoding, need to be reversed. The ability to perform these processes in parallel, however, is significantly hampered by the Deflate compressed format. First, the Huffman encoded blocks need to be parsed and decoded but, as mentioned in Section 2.1, the locations of the boundaries between the blocks are completely unknown. This means each block must be processed serially until the EOB marker has been decoded, which impacts the ability to perform Huffman decoding on more than one block in parallel.

There is one technique for overcoming the problem of finding exploitable

[5]The multi-core server workstation that was used to perform the design and synthesis work in this thesis, containing an i7-3930K CPU and 32 GB of RAM, was able to achieve an average input compression throughput of 7.74 MB/s when dynamically compressing the Calgary corpus.

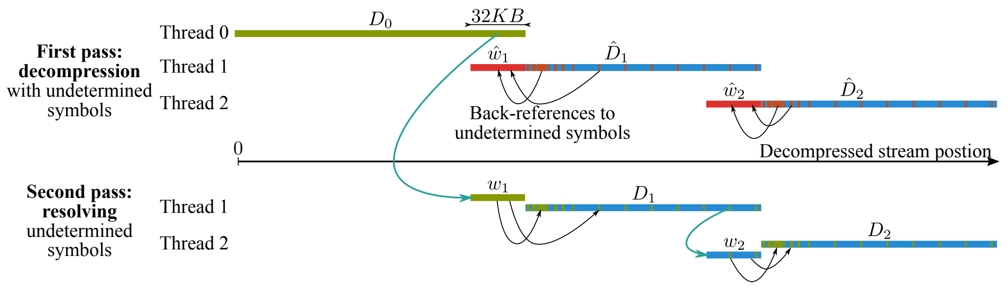
parallelism called *speculative parallelization.* Once the EOB code is known (either by identifying a block as static or by assembling its dynamic code tables), the compressed data can be scanned ahead for the block's EOB code. If a block's EOB code is found (and therefore the next block boundary is located), the next block can begin being Huffman decoded in parallel with the current block. Unfortunately, the possibility of finding false-positive boundaries exists due to 1) the absence of byte-alignment in the compressed format and 2) the extra bits that follow the length and distance codes, which can mimic the EOB code. Deflate blocks can also be any size and a compressed file could even be composed of only a single Deflate block, meaning that block boundaries can be difficult to find or even be non-existent. It is for these reasons that the process is speculative and any potential speedup obtained is statistical.

Speculative parallelization helps to accelerate the Huffman decoding process but it leaves unaddressed the LZ77 decoding process, which is also difficult to parallelize. Since the length-distance pairs can have distances up to 32,768 bytes away and because Deflate blocks can be any size, it is possible for a length-distance pair to point to data held in another block. This inter-block dependency prevents LZ77 decoding from being performed on multiple blocks in parallel. Some proposed solutions to these problems by related works will be examined. Since the number of published works on decompression accelerator designs is much smaller than on compression accelerators, a broader look will be taken at various papers across different platforms beyond just FPGA-based designs. Decompression throughput can be measured at the input (i.e., bytes of compressed data processed per second) or the output (i.e., bytes of uncompressed data processed per second) and will be specified as such when known.

In paper [33], the authors propose a configurable software algorithm for speculative parallelization. In this algorithm, they add two tests to validate the accuracy of a prospective block boundary. The first test checks the 3-bit block header that follows a block boundary; if BTYPE is 0b11, corresponding to an erroneous block, the boundary is known to be false. The second test is an attempt to perform Huffman table reconstruction for a dynamic Huffman block; a false boundary may cause errors to occur during the table construction process. It is still possible for false-positive boundaries to pass both of these tests and, if so, it isn't until the first Huffman decoder has processed up to the point that the second decoder started from that it is known for certain, whether or not the speculative work done is valid. The scanner in their algorithm is configurable with different parameter settings like "minimum block length" and "speculated range" in order to aid with boundary prediction. In order to parallelize the LZ77 portion of decompression, they rely on the presence of stored blocks within the compressed file. Since zlib has the tendency to create stored blocks with sizes above 16 kB, the 32 kB history buffer of an LZ77 decoder can be filled using two consecutive stored blocks. This creates a point in the data stream where the LZ77 decoding can be done in parallel. As the authors of [33] found, however, the presence of these stored blocks is unreliable. Many compressed files do not contain any stored blocks at all, and those files that do may not have the blocks uniformly distributed to take advantage of. Because of this, [33] reports speedups of only 1.24 to 1.80 times faster than sequential software decompression (actual decompression throughputs are not given).

The authors of [34] expand on the work done in [33] and implement a speculatively parallel decompression algorithm using Apache Spark [35], a software framework for parallel computing. Using this, multiple speculative threads are spawned across a cluster of CPUs and are each given parts of the compressed file to decompress in parallel. Each thread performs tests to check for false-positive boundaries as done in [33] and any invalid threads are cancelled. Once all threads are finished, the results are merged and output. In the worst-case scenario, the data is decompressed sequentially using a single thread. No information is given on if and how the LZ77 decoding process is performed in parallel. Using a cluster of six CPUs (containing a total of 24 threads), they achieve average input decompression throughputs between 18 and 24 MB/s, which is a speedup of about 2.6 times compared to sequential decompression (about 7 MB/s).

The authors of [36] investigated the feasibility of randomly accessing parts



Thread 0
D_0
$32KB$
First pass: decompression with undetermined symbols
Thread 1
$\hat{w}_1$
$\hat{D}_1$
Thread 2
$\hat{w}_2$
$\hat{D}_2$
Back-references to undetermined symbols
0
Decompressed stream postion
Second pass: resolving undetermined symbols
Thread 1
w_1
D_1
Thread 2
w_2
D_2

Figure 2.8: A diagram showing the two-pass speculative parallelization method (Figure from [36]).

of compressed DNA files, specifically of the FASTQ format [37], and developed a parallel decompression algorithm to accelerate this. These FASTQ files contain DNA sequences and are typically very large, requiring them to be compressed losslessly using an algorithm like Deflate. Because of the Deflate format, in order to access even a small portion of the large FASTQ file, the entire compressed file must be decompressed. For this reason, the authors investigated the ability to begin decompressing a file from a random point within it. Like previous works, they use speculative parallelization to find block boundaries where decompression can be started. FASTQ files are a text-based format that contains byte-aligned ASCII characters with values from 0 to 127. The authors exploit this as another way of testing for false-positive block boundaries. If a literal character outside of the range is decoded, the speculated block is invalid. In order to parallelize the LZ77 decoding process, they perform two passes on the data. On the first pass they perform LZ77 decoding on each block and skip any back-references that point to undetermined data. Then the data is passed through a second time and the skipped back-references are resolved. A diagram showing this process is shown in Fig. 2.8. While decompressing compressed FASTQ files using 32 threads in parallel, they are able to achieve average input decompression throughputs of 611 MB/s.

As can be seen from the previous paragraphs, the standard Deflate format makes parallelization of the processes difficult. The authors of [4] attempt to overcome these problems by altering the format. By making relatively simple changes to the Deflate format, they could achieve tremendous decompression speedups. In their so-called "Gompresso" format, each Deflate block is a fixed size and is split up further into sub-blocks. The size of each of these sub-blocks is included in a file header at the beginning of the compressed file, allowing each sub-block to be easily parsed and Huffman decoded in parallel. Since the blocks in the format are compressed in parallel as well, LZ77 back-references between blocks don't exist. This means LZ77 decoding of each block can also be performed in parallel. The authors take things a step further and include the option of preventing nested back-references within a block, eliminating intra-block dependencies as well. This allows a single block to be processed by multiple threads in parallel. The authors implemented their Gompresso compression/decompression framework using GPUs and were able to achieve average output decompression throughputs of over 13 GB/s. This throughput was limited by the PCIe interface and was actually closer to 16 GB/s ignoring data transfer rates. In order to take advantage of these impressive decompression speeds though, the files needed to be compressed using the Gompresso format.

The following related works are decompressor designs implemented using FPGAs. Instead of attempting to parallelize the Huffman and LZ77 decoding processes, these designs aim to exploit the parallel resources on FPGAs in order to accelerate them. The authors of [38] describe a two-core pipelined system composed of a Huffman decoder and an LZ77 decoder. For simplicity, the Huffman decoder is only capable of decoding static Huffman blocks. The Huffman decoder has a 32-bit wide input interface. Since static literal/length codes are anywhere from 7 to 9 bits long, they can be decoded by taking 9 bits from the input and looking them up in a 512-index LUT[6]. The table identifies the data as a literal, a length, or the EOB symbol. If the symbol is a length, the extra bits are obtained and the following 5-bit distance code is looked up in another LUT. The Huffman decoder takes 3 clock cycles to decode a literal and 6 cycles to decode a length-distance pair. The authors claim that the Huffman

[6]All 9-bit codes will appear once in the table, all 8-bit codes will appear twice, and all 7-bit codes will appear four times.

decoder is also capable of processing dynamic codes if the Huffman code tables are recreated off-chip and then given to the FPGA-based accelerator core. Since the dynamic codes can be up to 15 bits long, however, this would require two 32,768 index LUTs to decode both literals/lengths and distances in the same manner. Following decoding, the literal or length-distance pair is then written to the output. In between the Huffman decoder and LZ77 decoder is a First-In, First-Out (FIFO) memory that alleviates stalling between the two cores and allows them to be run at different clock frequencies. The LZ77 decoder is responsible for resolving all of the length-distance pairs and contains a circular buffer with enough space to hold the last 32,768 bytes. This circular buffer is assembled using dual-port BRAMs so that read and write operations can be performed simultaneously. The LZ77 decoder will record literals in the buffer as it receives them and passes them to the output. When it receives a length-distance pair, it reads 1 byte of the back-referenced string from the circular buffer every clock cycle while writing it back to the head of the buffer and the output. The Huffman decoder has a clock frequency of 160 MHz while the LZ77 decoder has a clock frequency of 212 MHz. Their design was synthesized for a Xilinx XC4VFX12 FPGA and both cores together occupied 128 FFs (2%), 387 LUTs (3%), and 18 RAMB16s (50%). They tested their design using static-Huffman-only compressed versions of the Calgary corpus and achieved a maximum output throughput of 205.71 MB/s, a minimum output throughput of 79.26 MB/s, and an average output throughput of 158.64 MB/s.

The authors of [39] implemented a single-core zlib decoder as part of an expandable zlib decoding system for FPGAs. The system was designed to incorporate multiple zlib decoders to allow for the decompression of multiple files in parallel. Much of their work describes how they directly translated the software code of the zlib library into VHDL hardware code. This included strategies like adapting function inputs and outputs into buffers for streaming as well as changing the dynamically allocated arrays into statically defined FIFOs and BRAMs. Their design is capable of processing both static and dynamic Huffman codes but the amount of time required to decode each

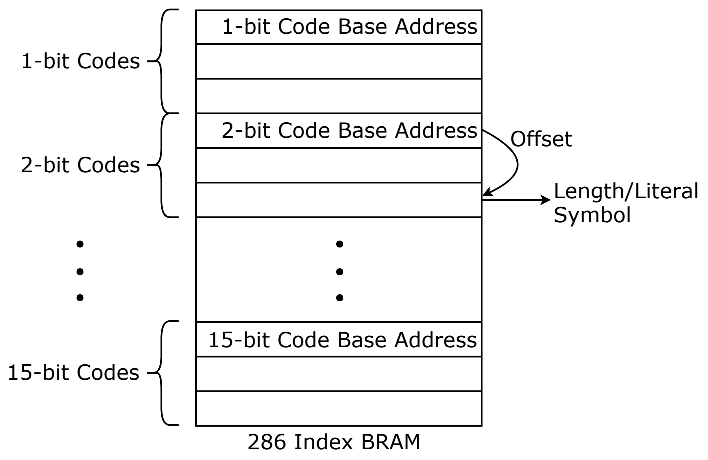


Figure 2.9: The area-efficient Huffman table memory design from [40].

was unfortunately not disclosed. Their design was synthesized for a Xilinx XC5VLX110T FPGA and has an area utilization of 20,596 LUTs (29%) and 11 BRAM tiles (7%). With a clock frequency of 120 MHz, their zlib decoder was able to achieve a maximum input throughput of 125 MB/s. The test files they decompressed and the average throughput speeds were not described.

In [40], the authors implement an FPGA-based decompressor core that uses an efficient Huffman table memory design. As mentioned above when describing reference [38], in order to decode the dynamic 15-bit Huffman codes directly using LUTs, 32,768 index tables are required. Since there are only 286 literal/length symbols and 30 distance symbols, the majority of the table entries would be duplicates for all of the codes that are less than 15 bits. The authors of [40] came up with a design that slightly complicates the decoding process but is able to fully utilize a 286-index table for literal/lengths and a 32-index table for distances. As mentioned in Section 2.1, the Huffman codes used by Deflate are canonical, meaning that all of the codes of each length have consecutive values. This means if the base address and base value for each code length are recorded, each code can be looked up relative to its base address. This process involves comparing the unknown-length code bits to the 15 different base code values to determine the actual bit-length of the currently held code. Then the address for the code is calculated using its base address and it can be decoded using the table, as shown in Fig. 2.9. This design was used by us in our Huffman decoder and is described in more detail in Section 4.2.1. Few other details regarding the design in [40] were given. It was designed as a decompressor core to be used as part of a multi-core system for an Intel/Altera Cyclone III 80 FPGA. The FPGA resource utilization is not described. The authors report a maximum decompression throughput of 300 MB/s but they do not list the test files decompressed or give average values. It is unknown whether this throughput was measured at the input or the output. They also do not specify whether this result is from using one core or multiple cores in parallel.

A proprietary decompression IP core created by Cast Inc. is available for purchase on Xilinx's website [41]. Though many details of the design are not disclosed, it can be used as a point of comparison. Their core is stated to output 3 bytes per clock cycle on average and operates with a clock frequency of 125 MHz, which equates to an average output throughput of 375 MB/s. When processing static-only compressed files, the core can run at a frequency of 165 MHz for an output throughput of 495 MB/s. One sample implementation is given for a Xilinx XCKU060 FPGA: when configured for dynamic mode the core uses 8,250 LUTs and 29 BRAM tiles (58 BRAM18Ks) and when configured for static-only mode the core uses 5,392 LUTs and 10.5 BRAM tiles (21 BRAM18Ks).

Table 2.7 shows a summary of all the decompression related works described in this section. The throughputs are specified as average or maximum and input or output values as disclosed in their respective works. As seen from the results, FPGA-based decompression accelerators are able to provide input throughputs on the order of hundreds of MB/s, much higher than most software decompressors are able to, which typically have values around 20 MB/s[7]. The results from [4] show that simple alterations to the standard Deflate format can allow for tremendous improvements in decompression throughput. While the strategy of speculative parallelization is a good idea for potentially

[7]The multi-core server workstation that was used to perform the design and synthesis work in this thesis, containing an i7-3930K CPU and 32 GB of RAM, was able to achieve an average input decompression throughput of 17.14 MB/s on the dynamically compressed Calgary corpus.

Table 2.7: Past Deflate Decompression Accelerators

Reference	Platform	Throughput	Details
[33] (2013)	Software	1.24-1.80x speedup	Over sequential decompression
[34] (2017)	Software	18-24 MB/s (average, input)	Using 24 threads on ApacheSpark
[36] (2019)	Software	611 MB/s (average, input)	On FASTQ format files with 32 threads
[4] (2016)	GPU	13 GB/s (average, output)	On Gompresso format files
[38] (2007)	FPGA	158.64 MB/s (average, output)	On static-only Calgary corpus files
[39] (2009)	FPGA	125 MB/s (max, input)	Unknown test files
[40] (2010)	FPGA	300 MB/s (max)	Unknown test files
[41] (2016)	FPGA	375 MB/s (average, output)	On unknown dynamic test files
		495 MB/s (average, output)	On unknown static test files

speeding up decompression, other more reliable strategies for improvement still remain, like harnessing the inherent parallelism available in FPGAs. The work done in [39] made us aware of the challenges associated with performing dynamic Huffman decoding using LUTs while the design in [40] provided a useful solution to this problem. All of the designs showcase the importance of testing decompression using a standard benchmark of test files, as the types of files being decompressed play a critical role in the decompression throughputs that are achievable.

2.4 Summary

In this chapter, we described the background information on Deflate and explored works related to Deflate compression and decompression. In Section 2.1, we detailed the Deflate compression algorithm and format, and in Section 2.2 we briefly described the Calgary corpus benchmark, which we will use to test our designs. In Section 2.3.1, we described the design progression of related works on FPGA-based compression accelerators. Current state-of-the-art compressor designs utilize a constant-throughput pipeline with a parallel sliding window in order to rapidly perform LZ77 matching, thereby sacrificing compression ratio in order to maximize compression throughput. In Section 2.3.2, we explained some of the fundamental limitations of the Deflate format that prevent the parallelization of decompression and examined a variety of works that attempt to work around these limitations. Though the number of published FPGA-based decompressor designs is relatively small, we were able to find a few that provided useful ideas upon which to base our design.

Chapter 3

Compressor Design

Our plan for an FPGA-based compressor design was to follow the current state-of-the-art architecture as used by most other related works (described in Section 2.3.1) and to improve the resulting compression throughput and compression ratio further while implementing our design using Vivado HLS. This architecture utilizes a parallel sliding window (of PWS bytes) to perform LZ77 matching on multiple substrings at a time. The compressor design is pipelined to compress data at a fixed rate using static Huffman encoding only. Performing the Huffman encoding operation is relatively simple as it involves encoding characters using tables stored in FPGA LUTs and then shifting and OR-ing together the various bit-width codes in the output window. Most of the focus of previous designs appears to be on the LZ77 encoder and on maximizing the compression throughput. As pointed out in [7], in order to increase the compression throughput of a compressor system, it is more sustainable to utilize multiple compressor cores than it is to increase the PWS of a single compressor. Match lengths (which are limited by the PWS value) greater than 16 bytes are also rare, and so the consensus seems to be that increasing the PWS for the purposes of achieving higher compression ratios is not worth it. Utilizing multiple parallel compressors results in less-efficient matching being performed (due to each compressor relying only on its own data history) and causes the compression ratio to decrease slightly. Most previous designs have focused on maximizing the compression throughput even at the cost of a lower compression ratio. Currently, however, the maximum achievable throughputs obtained by multi-compressor architectures are being bottle-necked by the CPU-FPGA interface as reported by [7]. For the above reasons, we chose to focus on implementing a single compressor core design with the goal of achieving high compression ratios while still maintaining competitive throughput values. The proposed compressor has PWS value of 16 bytes and will utilize a constant-throughput pipeline by performing static Huffman encoding only. Both the LZ77 encoder and Huffman encoder will be pipelined with an Initiation Interval (II) of 1, meaning that they can read in new input data (i.e., 16 bytes) every clock cycle. Our desired clock frequency is 250 MHz (4 ns period), which, when compressing 16 bytes per clock cycle, would provide a compression throughput of 4.0 GB/s. Unlike most other earlier works on compressor cores, our designs were implemented using Vivado HLS in order to evaluate the capabilities and limitations of high-level synthesis.

3.1 Initial Design

Since our focus lay more on increasing the compression ratio than increasing the throughput, our initial design adopted the dictionary architecture from [27], which had the highest geometric mean compression ratio of all the previous works when compressing the Calgary corpus. This design differs from the others in how it uses a fully connected dictionary design instead of a hash banking design. Enough dictionaries and dictionary copies are used so that each of the 16 substrings in the window (with a PWS of 16) is able to look up 16 potential matches for a total of 256 total match comparisons being performed in every clock cycle. Because of the fully connected design, no hash collisions can occur and, as a result, no matches are ever dropped, which leads to higher compression ratios than in other designs, where potential matches are dropped as a result of hash collisions. This type of fully connected dictionary architecture comes at a cost, however. Each stored string requires 16 bytes while their positions will require another 4 bytes (if we assume 32-bit positions are being used). With a 10-bit hash value, each dictionary will have 1024 indexes (as in [27]) and will be replicated 16 times to allow uncontested access. In total:

20 bytes × 1024 indexes × 16 dictionaries × 16 copies = 5,242,880 bytes of storage required. On a Xilinx FPGA, this requires 2048 BRAMs (BRAM18Ks specifically) for the strings and 512 BRAMs for the positions totalling 2560 BRAMs altogether, which is more BRAMs than most contemporary FPGAs provide. Assuming that the number of available BRAMs is not an issue, the other problem that we encountered with this design was with meeting clock timing due to all of the extra routing delays incurred by the relatively large layout area. Vivado HLS was easily capable of synthesizing a design that could logically meet timing; however, following implementation in Vivado, the design ended up requiring a clock period around 7 ns due to all of the routing delays, which is almost double that of our target clock period of 4 ns. Because of this massive difference, we decided to change our design to one that would require fewer BRAMs and, consequently, would be easier to meet post-route timing.

3.2 Final Design

A block diagram of our final compressor design is shown in Fig. 3.1. It is composed of only two cores, an LZ77 encoder and a Huffman encoder, both of which were specified in C++ and synthesized using Vivado HLS. Both of the cores have AXI-Stream interfaces [42], a standard interface used by Xilinx FPGA systems to facilitate high-speed unidirectional data transfers in streaming architectures. The AXI-Stream interface functions using handshake

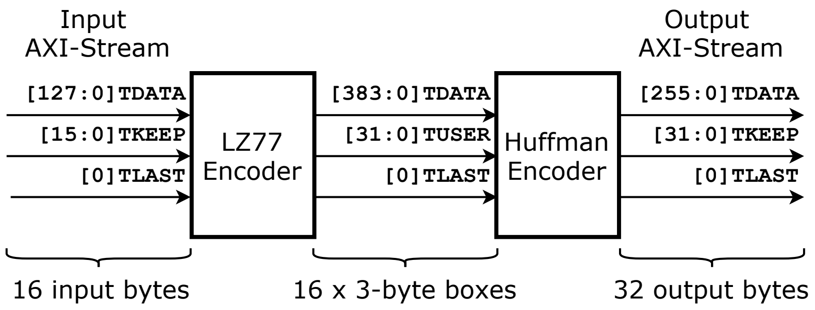


Figure 3.1: Final compressor block diagram with AXI-Stream interfaces described.

signals (not shown in the block diagram), which consist of a forward travelling TVALID signal and a backwards travelling TREADY signal. TDATA is the name of the main AXI-Stream data-carrying channel, while the remaining signals, TKEEP, TUSER, and TLAST, are optional sideband signals. The TKEEP signal is used by multi-byte data streams to indicate which bytes in the current transfer are valid, the TLAST signal is used to indicate the last transfer in a data stream, and the TUSER signal can be employed by the AXI-Stream user for passing any other desired data alongside TDATA. In our case, we use the TUSER signal in between the two encoders in order to indicate which bytes are matched literals, unmatched literals, or length-distance pairs. This is explained in further detail below.

3.2.1 LZ77 Encoder

As mentioned before, we followed the same structure as other previous works and used the parallel pipelined LZ77 encoder design with a PWS of 16 bytes. A block diagram describing the stages of the LZ77 encoder pipeline is shown in Fig. 3.2. At every clock cycle, 16 bytes of data are read from the input stream and shifted into a 32-byte window. Across this 32-byte window are 16 staggered substrings, each 16 bytes long (see Fig. 2.5). It is these 16 substrings that will attempt to be matched with previous strings in order to replace them with length-distance pairs. To do this, the first 5 bytes from each substring

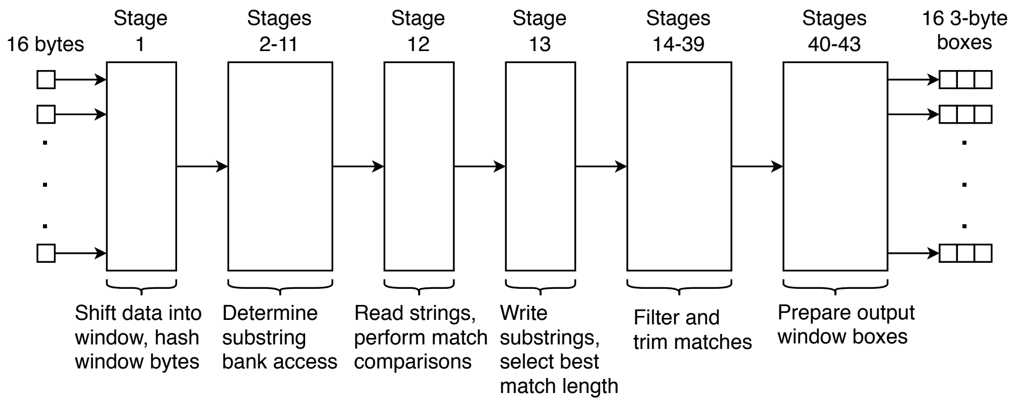
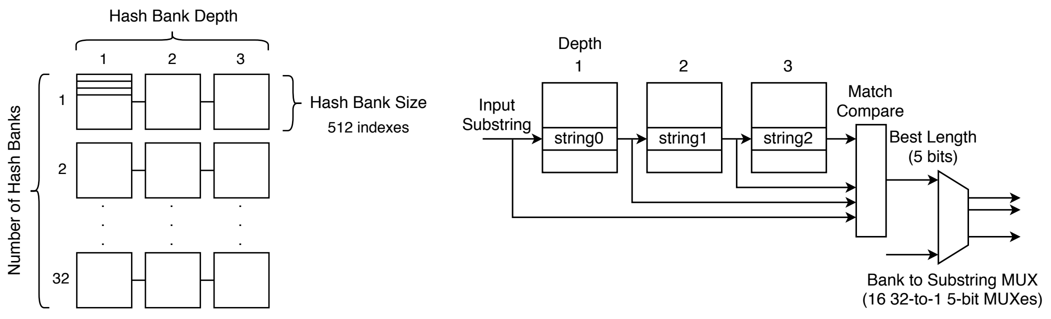

16 bytes
Stage 1
Stages 2-11
Stage 12
Stage 13
Stages 14-39
Stages 40-43
16 3-byte boxes
Shift data into window, hash window bytes
Determine substring bank access
Read strings, perform match comparisons
Write substrings, select best match length
Filter and trim matches
Prepare output window boxes

Figure 3.2: LZ77 encoder pipeline diagram.

are taken and hashed to generate 16 hash signatures. The hashing process is explained in more detail later in Section 3.2.2. We used the hash bank architecture from [7], which utilizes 32 hash banks with each bank having 3 depth levels. Each of our hash banks has 512 indexes in order to fully utilize the BRAM storage space, as shown in Fig. 3.3a. These hash banks are accessed by the 16 substrings based on their hash signatures with priority given to the last substring first, since last-fit match selection is performed later. If a substring is granted access (i.e., another substring has not already accessed its desired bank), that substring will be written to the bank and location that its hash signature addresses. Otherwise, the substring is dropped and is not recorded or compared with previous strings. As the substrings are recorded in the dictionary banks, new strings are recorded in the top depth bank while each previously stored string is written from one depth to the next, effectively shifting each of the strings down a level with the oldest string being discarded. At the same time, the previous strings from each depth level are read and compared to the input substrings to find potential matching strings. With a depth of 3 levels, 3 potential matches are read and compared to each input substring that was granted access, as shown in Fig. 3.3b. We used the MUX optimization from [7], in which the match comparisons are performed at the output of the banks so that only the resulting best match length needs to be MUXed back to the substring window. Since the match length can only be



(a) (b)

Figure 3.3: a) The hash bank architecture that was used in the final design. b) The match comparisons are performed at the outputs of the banks before being MUXed back to the substring window.

from 0 to 16, 5-bit wide MUXes can be used instead of 16-byte wide MUXes.

As was done in [27], we perform the match comparisons by comparing each byte simultaneously and storing the Boolean result of each byte compared as a bit in a bit-string. This is done instead of comparing each byte sequentially and incrementing the length for each matching byte, as this approach would result in an inefficient and relatively long select-adder chain. In order to convert the resulting bit-string back into an integer value, we invert it and use the built-in Count Trailing Zeroes (CTZ) operator[1]. The longest match length out of all of the potential matches is kept and MUXed back to the substring window. Following this, each substring in the window will have a match length. Substrings that were not granted bank access are given match lengths of 0. Since the matches found for each substring will conflict, the matches need to be filtered in some manner. We followed the last-fit match selection heuristic used in [27], in which the last match in the window is always kept as it is more likely that later matches will be longer than earlier ones. One consequence of allowing matches to span from the front half of the sliding window to the second half is that matches found in the current iteration may conflict with matches in the following iteration. The last match selected is used to calculate the First Valid Position (FVP) in the window for the next iteration, where the window bytes have not already been replaced with a length-distance pair. As explained in [27], there is a feedback path from the computation of the FVP, which is used in the next iteration. In order to prevent pipeline bubbles from occurring and the II from being greater than 1, the FVP needs to be calculated as soon as possible. As a result, once the FVP for the next iteration is calculated, it cannot be changed by any of the match filters that follow.

To perform match selection, we again utilize a bit-string of 16 bits, with each bit indicating the validity of a match within the window. As matches are filtered, these bits are set to 0. As can be seen from Fig. 3.2, 26 of the 43 pipeline stages are used for match-filtering. The filtering process is split across four main filters, which we have called Filters A to D. In Filter A, which

[1]See the "C Builtin Functions" section of the Vivado HLS User Guide [43] for more information.

directly follows the best match selection stage, matches that have lengths less than 3 or distances greater than 32,768 are filtered out (as defined by the Deflate specifications). Next, in Filter B, the matches are compared to the FVP in the window and any matches with characters that have been covered by the previous iteration are filtered. Following this, the FVP for the next iteration is calculated and the filters that follow can no longer alter this value. Here we have also used the match "reach" heuristic from [27]. The reach of each match is defined as its position in the window plus the length of its match, representing how far a match length reaches within the window. Using this, in Filter C, the reach of each match is compared to the others in order to find matches that have the same reach as each other. In these cases, the match with the longer length can be kept instead, even though it may not necessarily be the last match, as this will not alter the FVP. In the final filter, Filter D, the last-fit match selection is performed in which earlier matches that conflict with later matches in the window are filtered. As an improvement over previous

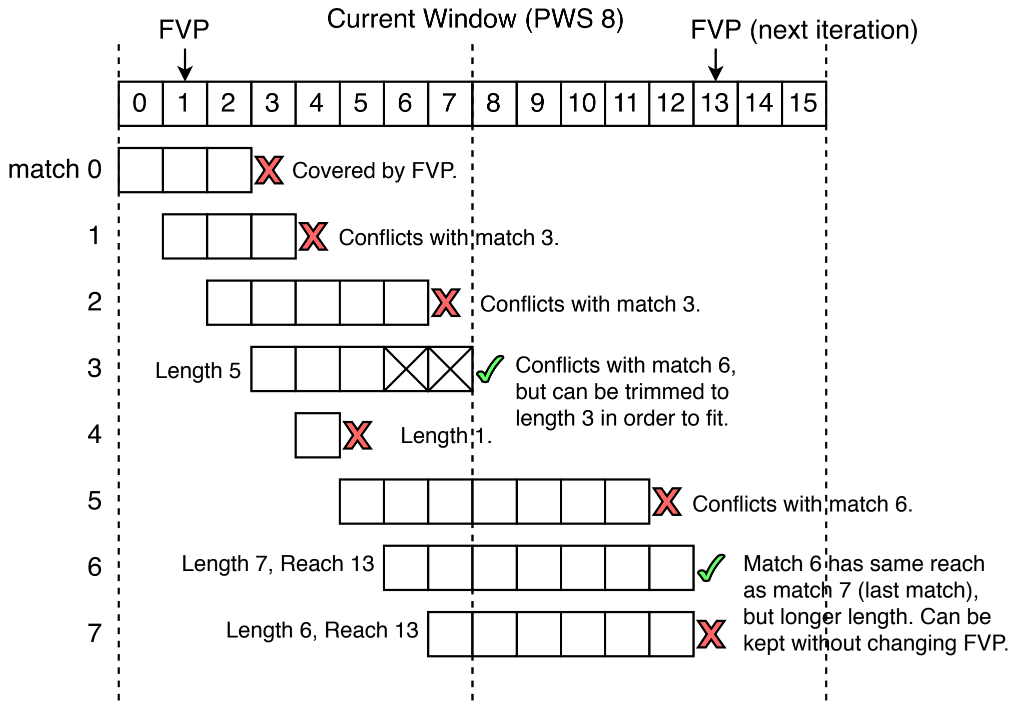
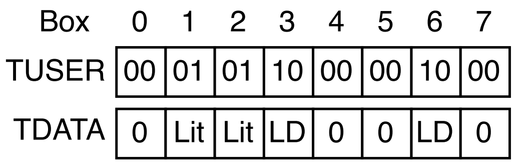


Figure 3.4: LZ77 encoder match filtering example. In this example, the PWS is 8 bytes. After filtering, only matches 3 and 6 are kept.

works, we have also implemented match trimming functionality to Filter D. When an earlier match is about to be filtered for conflicting with a later one, the filter will attempt to trim the earlier match (down to a maximum of 3 bytes) so that both matches can be kept. An example showing all four of the match filters being used is shown in Fig. 3.4.

We will refer to the outputs from the 16 substring locations in the sliding window as “boxes”. Following filtering, each box will be classified as containing a matched literal, an unmatched literal, or a match (i.e., where the beginning of a match occurs) and will be prepared for the output to the Huffman encoder. The AXI-Stream TUSER signals are used to classify each box using a 2-bit flag: 0b00 for matched literals, 0b01 for unmatched literals, and 0b11 for matches. All of the boxes following a match, up to the match length, are the matched literals and are cleared and flagged so that they can be removed from the compressed data stream. The boxes containing matches are filled with the length and distance of the match while unmatched literal boxes are filled with the literal bytes from the sliding window, as shown in Fig. 3.5. In a length-distance pair, the length requires at least 4 bits (for values 3 to 16) and the distance requires at least 15 bits (for values 1 to 32,768), together occupying 19 bits. This means that each output box needs to be 3 bytes (24 bits) wide in order to fit a length-distance pair, requiring a total of 48 bytes (384 bits) for all 16 boxes. We do this so that each output box can be classified and filled simultaneously. The alternative is to fill these boxes one at a time and spread the length-distance pairs across multiple boxes (requiring fewer output bytes), but this requires sequentially filling each box. Packing the length-distance pairs into a single box as opposed to multiple boxes also allows the Huffman encoder to encode each box separately and concurrently instead of having to



Box	0	1	2	3	4	5	6	7
TUSER	00	01	01	10	00	00	10	00
TDATA	0	Lit	Lit	LD	0	0	LD	0

Figure 3.5: LZ77 encoder output window example that follows the previous example from Fig. 3.4 (with a PWS of 8).

obtain the length and distance bits from multiple boxes. Consequently, the interface between the LZ77 encoder and Huffman encoder (as shown in Fig. 3.1) is as follows: TDATA is composed of 16 3-byte boxes for a total width of 384 bits, TUSER is composed of 16 2-bit flags classifying each box, and TLAST signals the last transfer of the stream.

The most difficult challenge encountered when designing the LZ77 encoder was meeting clock timing after implementation and routing in Vivado. Even after deviating from our original design, as described in Section 3.1, this still proved to be a major problem. Vivado HLS is only aware of the logic synthesis stage of design and has no control over implementation and routing, which is performed by Vivado as a separate and subsequent step. As a result, Vivado HLS would synthesize designs that would easily meet timing from a logic delay standpoint, but would be impossible to meet timing after routing. The implemented designs would be prone to high routing congestion and large setup time violations. Even when performing high-effort implementation runs in Vivado that would attempt to reduce congestion and over-estimate net delays, timing still could not be met. The two main areas that seemed to be the largest contributors to the problem were the bank-access allocation stages and the best-match-length selection stages.

We ended up solving this timing problem using two main solutions. The first was the manual addition of extra pipeline stages to the design in Vivado HLS in order to reduce the routing delay between pipeline stages. As Vivado HLS is not aware of routing, it will only automatically add more pipeline stages when necessary to meet logic timing, not routing timing. The second solution was increasing the "clock uncertainty" parameter setting in Vivado HLS. This setting forces Vivado HLS to restrict the logic delay to a maximum portion of the clock period, allowing more room for routing delay later on. We ended up using a clock uncertainty of 0.8 ns, which restricted all synthesized logic delays to 3.2 ns. Using these two solutions, we were able to get the LZ77 encoder design to meet the target 4 ns (250 MHz) timing.

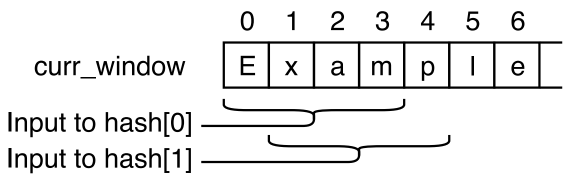
3.2.2 Hash Function

Since the LZ77 encoder design that we use performs match finding at a constant rate (and therefore more time cannot be spent searching for better matches), the only other design parameter that is left affecting the output compression ratio besides the dictionary architecture (i.e., the size and number of banks), the PWS value, and the type of Huffman encoding used is the hash function. Since we had to switch away from a hash dictionary design that was immune to hash conflicts, as explained in Section 3.1, we also have to be aware of how susceptible our chosen hash function is to hash conflicts as dropped matches seem to impact the compression ratio significantly. No hash function is able to prevent the occurrence of hash conflicts completely; however, we can aim to make the probability of conflicts as low as possible. The hash function should be provided enough information from the bytes in the sliding window to be able to find potential matching strings but not so much information that match finding becomes too strict and potential matches become too difficult to find. Out of all of the previous works examined in Section 2.3.1, only one, [27], describes the hash function that was used. This hash function is the following:

Algorithm 1: Hash Function from [27]

```
  Input: $curr\_window[2 \times PWS]$, array of 8-bit ints
  Output: $hash[PWS]$, array of 10-bit ints
1 for $i \leftarrow 0$ to $PWS\text{-}1$ do
2     $hash[i] = (curr\_window[i] \ll 2) \oplus (curr\_window[i+1] \ll 1) \oplus$
        $(curr\_window[i+2]) \oplus (curr\_window[i+3])$;
```

An example of how this hash function is used to generate hash signatures using the substrings within the sliding window is shown in Fig. 3.6. We used this function as the starting point for our hash function experimentation. Since our design uses 32 banks with 512 indexes each (for a total of 16,384 indexes), we used a 14-bit hash value. The top 5 bits of the hash value determine the bank while the bottom 9 bits determine the bank index. In order to reduce the number of bank conflicts, we want the top 5 bits to be as random as possible



$$
\begin{aligned}
\text{hash[0]} &= (\text{"E"} << 2) \oplus (\text{"x"} << 1) \oplus (\text{"a"}) \oplus (\text{"m"}) \\
&= (69 << 2) \oplus (120 << 1) \oplus 97 \oplus 109 \\
&= 276 \oplus 240 \oplus 97 \oplus 109 \\
&= \boxed{488 = \text{0b } 01\ 1110\ 1000} \\
\text{hash[1]} &= (\text{"x"} << 2) \oplus (\text{"a"} << 1) \oplus (\text{"m"}) \oplus (\text{"p"}) \\
&= (120 << 2) \oplus (97 << 1) \oplus 109 \oplus 112 \\
&= 480 \oplus 194 \oplus 109 \oplus 112 \\
&= \boxed{319 = \text{0b } 01\ 0001\ 1111}
\end{aligned}
$$

Figure 3.6: An example showing how the bytes from the substrings in the sliding window are used to calculate 10-bit hash signatures using the hash function from [27]. In this example, only the first two calculations are shown. With a PWS of 16, there would be 16 of these calculations.

so that hash values are distributed evenly across all of the banks.

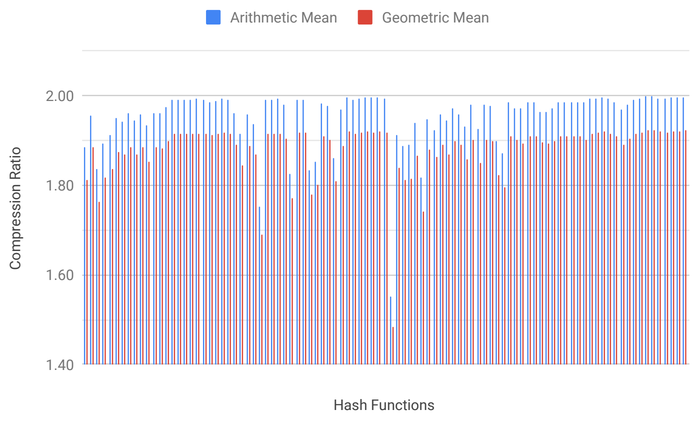
Our first attempts at improving the hash function involved varying the amount of shifting done on each byte before XORing them together. Since we wanted the upper 5 bits to receive the most amount of randomness, we shifted the 4 input bytes left far enough so that each one affected the upper 5 bits in some way. We then tried adding more input bytes to the hash function. We found that hashing 5 input bytes instead of 4 showed an improvement but hashing 6 bytes was worse. We believe this is because 6 bytes provides too much information resulting in the match finding becoming too strict. The reason for this is because when 6 bytes of a string are used to generate the hash value, match lengths of 3, 4, or 5 become harder to find, as each additional byte of information scrambles the hash value further. We also experimented with using two separate hash functions: a 5-bit function for the bank and a 9-bit hash function for the index, both utilizing the input bytes in different ways. We found, however, that these generally gave worse results than one 14-bit hash function. We believe this may be because of how bits were dropped from the 8-bit inputs when they were incorporated into the 5-bit hash value, resulting in poorer bank distribution.

Following this, we experimented with multiplication instead of shifting. In Vivado HLS, certain bit-wise operations are optimized automatically if more efficient operations can be performed instead. For example, multiplying a number by 31 is automatically optimized into "left-shift by 5 bits" and "subtract 1" operations. Taking the modulo of a "power of two" number simply selects the lower order bits of a number while division by powers of two performs right-shifts. This is generally an advantage of HLS but it can lead to confusion when attempting to understand the synthesized output, which is necessary when identifying areas where your design can be improved. We found that replacing the left-shift operations with multiplication operations that shifted by the same amount but also subtracted gave better results.

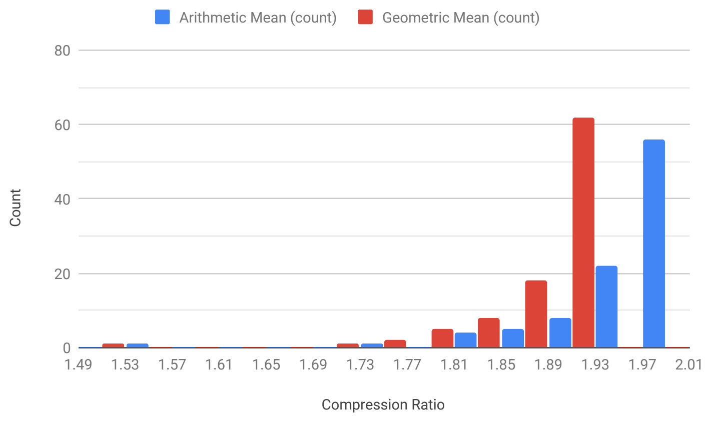
The next major change that we tried was combining the input bytes sequentially instead of concurrently. Up until now, the hash function involved manipulating the input bytes in various ways before XORing all of them together to get the final result. Instead, we tried performing something similar to a rolling hash function, where each input byte is incorporated into the hash one-at-a-time. From here, we experimented with shifting and multiplying by various values as well as XORing or adding the results to the hash. We then began performing various rotations on the hash function in-between incorporating the input bytes. We found that some hash functions worked better to improve the compression results of the non-ASCII data files, like geo and pic, while others work better on ASCII text data files. As a result, it was difficult to find a hash function that worked well across all of the different files in the Calgary corpus. In total, we tried about 100 different variations of hash functions. The compression ratio results from all of the different hash functions that we tried are shown in Fig. 3.7. As can be seen from this figure, most of the functions tested provided compression ratios within the range of 1.80 to 2.00, with a few outliers falling below this range. The best hash function tested provided arithmetic and geometric mean compression ratios of 2.00 and 1.92, respectively, on the Calgary corpus. This hash function is the following:

Algorithm 2: Our Chosen Hash Function

```
    Input: $curr\_window[2 \times PWS]$, array of 8-bit ints
    Output: $hash[PWS]$, array of 14-bit ints
1   for $i \leftarrow 0$ to $PWS\text{-}1$ do
2       $hash[i] = (curr\_window[i] \times 31) \oplus (curr\_window[i+1]);$
3       $hash[i].rrotate(4);$
4       $hash[i] = (hash[i] \times 3) \oplus curr\_window[i+2];$
5       $hash[i].rrotate(4);$
6       $hash[i] = (hash[i] \times 3) \oplus curr\_window[i+3];$
7       $hash[i].rrotate(4);$
8       $hash[i] = (hash[i] \times 3) \oplus curr\_window[i+4];$
```



(a) Column Chart



(b) Histogram

Figure 3.7: Compression ratio results from experimenting with various hash functions.

3.2.3 Huffman Encoder

Fig. 3.8 shows a diagram of the final pipelined Huffman encoder. Compared to the LZ77 encoder portion of the compressor design, the Huffman encoder is much simpler. Since only static Huffman codes are to be used in the design, the literals and length-distance pairs received from the LZ77 encoder can be quickly encoded using static codebook ROMs. Following this, the variable length codes just need to be packed together using a series of shift and OR operations. As mentioned above in Section 3.2.1, we designed the interface between the two encoders so that each box passed between the two can be encoded independently of the others. Each box from the LZ77 encoder is passed through a symbol encoder, which checks the type of symbol in the box (either a matched literal, unmatched literal, or length-distance pair) and encodes it accordingly. Unmatched literals and length-distance pairs are encoded using ROMs and their codes and accompanying extra bits are assembled and output. If a matched literal is received, it is removed from the stream by replacing it with a code and code length of 0. Once encoded, the codes as well as their bit-lengths are given to a window packer, which is responsible for shifting the codes and packing them into an encoded window of bits. The encoded window contains enough space for codes from each of the 16 boxes. The longest possible encoded size of a 16-box input window is 15 9-bit literal codes followed by a 26-bit length-distance code (7-bit length, 1 extra length

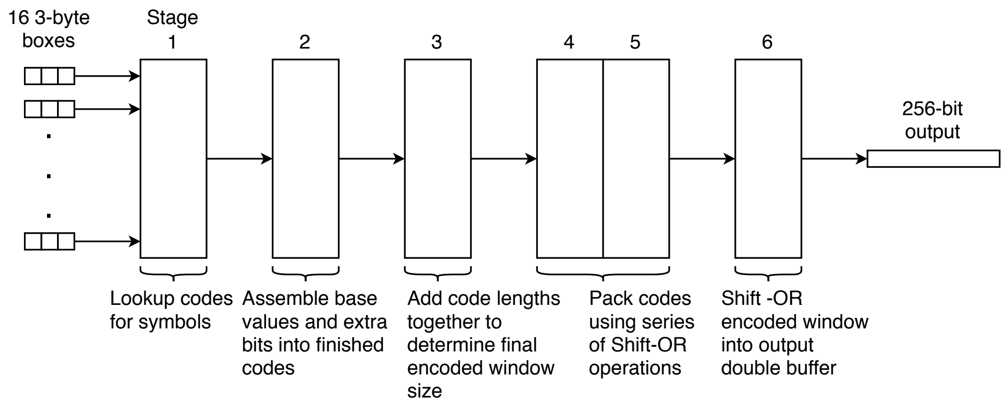

16 3-byte boxes
Stage 1
2
3
4
5
6
256-bit output
Lookup codes for symbols
Assemble base values and extra bits into finished codes
Add code lengths together to determine final encoded window size
Pack codes using series of Shift-OR operations
Shift -OR encoded window into output double buffer

Figure 3.8: Huffman encoder pipeline diagram.

bit, 5-bit distance, 13 extra distance bits) for a total of 161 bits. While it is impossible for there to be two consecutive 161-bit encoded windows (as the following window would have 15 matched literal boxes), it is possible for there to be two consecutive windows with more than 128 bits each. Therefore, the size of our output stream needed to be at least 256 bits wide to allow data to be written out in every clock cycle. At the end of the pipeline is an output window packer which contains a double buffer that is 512 bits wide. When the buffer contains more than half that amount, it writes the lower 256 bits to the output stream.

Our plan for the Huffman encoder design was to create a 16-stage pipeline in which each stage would take one box from the input window, encode it, and pack it into the encoded window. In every clock cycle, the input window and encoded window would be passed from one stage to the next. At the end of the pipeline is the output window packer which receives a fully encoded window in every cycle and writes it to the double buffer. With function inlining disabled in Vivado HLS, this is what ends up being synthesized. With function inlining enabled, however, Vivado HLS is able to optimize the operations performed and combine the pipeline stages together. What ends up being synthesized is a 6-stage pipeline where each input box is encoded and packed together simultaneously, as shown in Fig. 3.8. The ability for Vivado HLS to automatically optimize designs into more efficient forms is a powerful ability that gives it an advantage over traditional Register-Transfer Level (RTL) design. As with the LZ77 encoder, the Huffman encoder is pipelined with an II of 1.

3.3 Testing and Results

The two compressor cores were synthesized using Vivado HLS for a clock frequency of 250 MHz. Table 3.1 shows the FPGA resource utilization of the two cores when synthesized for a Xilinx Virtex UltraScale+ XCVU3P-FFVC1517 FPGA. The desired clock frequency and FPGA used were chosen by our industrial collaborator, Eideticom. We tested the compressor using the Calgary corpus benchmark files. A test bench was created in Vivado in which

Table 3.1: Compressor FPGA Resource Utilization

	LUTs	FFs	BRAM Tiles
LZ77 Encoder	62,227 (15.79%)	46,225 (5.86%)	240 (33.33%)
Huffman Encoder	6,887 (1.75%)	3,554 (0.45%)	20.5 (2.85%)
Total	69,114 (17.54%)	49,779 (6.31%)	260.5 (36.18%)

an AXI Direct Memory Access (DMA) core was used to stream the input data from memory to the compressor core, receive the compressed output data stream, and write it back to memory for verification. The compressed output files were decompressed and compared to their original forms in order to verify that the compressor was operating correctly. The resulting compression ratios were obtained by dividing the uncompressed size by the compressed size for each file and are shown in Table 3.2. Across the entire corpus, our compressor achieved an arithmetic mean (average) compression ratio of 2.00 and a geometric mean compression ratio of 1.92. Both compressor cores were synthesized with a pipeline II of 1, which, with an LZ77 encoder PWS of 16 bytes and a clock frequency of 250 MHz, provide an input compression throughput of 4.0 GB/s.

The results of our FPGA-based compressor design are compared to those from other previous works in Table 3.3. As mentioned in Section 2.3.1, the results from [5] are for a compressor with a PWS of 16 bytes and the results from [7] are for a single-core compressor only. By achieving a working clock frequency of 250 MHz, 50 MHz higher than the 200 MHz clock from [7], an extra 0.8 GB/s of throughput is obtained. This makes our design the fastest single core compressor out of those compared here. In terms of area, it is slightly difficult to make a direct comparison between our Xilinx-based FPGA design, with area utilization described in LUTs and FFs, and those designs using Intel/Altera-based FPGAs, which describe their area usage in ALMs. Compared to the designs that reported their memory usage, [28] and [27], our design uses about half as many BRAMs[2]. Compared to the other compressor design that was synthesized using Vivado HLS, [6], our design uses 20% fewer

[2]BRAM18Ks can store 18k bits while M20Ks can store 20k bits. Note that the actual number of bits stored in each will depend on the word width and depth used.

Table 3.2: Calgary Corpus Compression Results

File	Uncompressed Size (bytes)	Compressed Size (bytes)	Compression Ratio
bib	111,261	56,450	1.97
book1	768,771	482,048	1.59
book2	610,856	325,400	1.88
geo	102,400	96,448	1.06
news	377,109	218,688	1.72
obj1	21,504	14,093	1.53
obj2	246,814	129,327	1.91
paper1	53,161	28,636	1.86
paper2	82,199	45,891	1.79
pic	513,216	137,428	3.73
progc	39,611	20,556	1.93
progl	71,646	30,510	2.35
progp	49,379	21,534	2.29
trans	93,695	39,795	2.35
		Arithmetic Mean	2.00
		Geometric Mean	1.92

LUTs and FFs. Our compressor design has a total of 49 pipeline stages (43 for the LZ77 encoder and 6 for the Huffman encoder), which is fewer than the 58-stage compressor from [5] and the 87-stage compressor from [27] but more than the reported 17-stage compressor from [28]. In terms of compression ratio, our mean compression ratios (both arithmetic and geometric) are lower than all other designs besides [6].

Table 3.3: FPGA-based Compressor Design Comparison

Reference	Clock Frequency	Pipeline Depth	Input Throughput	Area Utilization	Compression Ratio
[28] (2013)	200 MHz	17	3.00 GB/s	105,624 ALMs, 1,152 M20Ks	?
[27] (2014)	193 MHz	87	2.84 GB/s	110,318 ALMs, 1,792 M20Ks	(Geo.) 2.17
[5] (2015)	175 MHz	58	2.80 GB/s	39,078 ALMs, ? M20Ks	2.05
[6] (2018)	200 MHz	?	2.80 GB/s	83,583 LUTs, 65,009 FFs, ? BRAM18Ks	(Geo.) 1.73
[7] (2018)	200 MHz	?	3.20 GB/s	38,297 ALMs, ? M20Ks	2.10
This work	250 MHz	49	4.00 GB/s	69,114 LUTs, 49,779 FFs, 521 BRAM18Ks	(Geo.) 1.92

3.4 Discussion

Despite the considerable effort that was made to improve the compression ratio of our design, we still weren't able to achieve higher ratios than previous works. Even with the addition of match-trimming to the match-selection stages of the LZ77 encoder, which to our knowledge is not done in other designs, our compression ratios were still lower. We found that the match-trimming improved the compression ratios by about 6% while adding 3 pipeline stages and increasing LUT usage by 3% and FF usage by 1%. Considering that we used the same 32-bank, 3-depth hash bank architecture as the design from [7], which achieved an average compression ratio of 2.10, we believe that our hash function may be one contributor, despite our extensive exploration into potential hash functions, and that there may still be room for improvement. The other thing that may account for the difference in the compression ratio is the number of hash indexes used. As mentioned above, our design used 512-index banks (see Fig. 3.3a) while the size of the banks in [7] is not disclosed. For comparison, the design from [5] used 65,536 hash indexes across 32 banks (2,048 indexes per bank), requiring a 16-bit hash function.

In order to verify the effect that the hash bank architecture has on the compression ratio and memory cost, we performed a series of design explorations, shown in Fig. 3.9. Throughout these tests, we kept the PWS value constant at 16 bytes and used the same hash function as was described in Section 3.2.2 (Algorithm 2). In Fig. 3.9a, the hash bank depth is increased while the number of banks and the total number of hash indexes are held constant, showing that the bank depth can be scaled up with linearly increasing memory costs. In Fig. 3.9b, the total number of hash indexes is increased while the number of banks and bank depth is held constant. Below 16,384 indexes, the number of BRAMs used does not decrease. This is due to how the BRAM18K configuration works: in order to accommodate a word width of 128-bits (16-byte strings), 4 BRAMs at minimum are required, no matter the number of memory indexes used. The 4 BRAMs have enough space to store 512 128-bit words, above which the number of required BRAMs doubles. Each hash bank also stores 512 32-bit positions, which completely fills another BRAM for a total of 5 BRAMs minimum per hash bank. Above 16,384 total hash indexes, the compression ratio barely increases while the memory costs increase exponentially. In Fig. 3.9c, the number of hash banks is increased while the number of total hash indexes is held constant. In this figure, the BRAMs are fully utilized up to 32 banks, after which the number of indexes per bank drops below 512. In Fig. 3.9d, the number of hash banks is increased while the number of indexes per bank is held constant at 512 (therefore the total number of indexes increases as well). This figure essentially shows the combined effects of Figs. 3.9b and 3.9c, with the number of memories required increasing exponentially. Despite the minimal increase in compression ratio that is obtained by increasing the total number of hash indexes, it makes more sense to do so while increasing the number of banks in order to fully utilize the BRAM storage space. If the increased memory cost can be accommodated, increasing the number of hash banks makes the most improvement on the compression ratio. Increasing the bank depth also provides some improvement to the compression ratio at a lower cost, but the returns are diminishing for depths greater than 3, as was also reported in [7].

The hash bank architecture of our compressor could be scaled up in order to increase the compression ratios achieved, but this would be difficult to get working as our design already struggles with meeting timing due to the large routing delay. The clock frequency (and therefore throughput) could be sacrificed though in order to allow more room for a larger scale design. As discussed in Section 2.3.1, increasing the PWS value is another way to increase both the compression ratio and the throughput, but the area requirements for PWS values greater than 16 bytes quickly become unsustainable and not worth the improvement as explored in [5] and shown in Fig. 2.7. Another idea for improving the compression ratio would be to implement multiple hash functions in the compressor. Two different hash functions could be used in a pipeline in order to give each substring two chances at accessing a bank instead of just one. Different hash functions that are tailored to different data types could be used as well. If the type of data being compressed could be communicated to

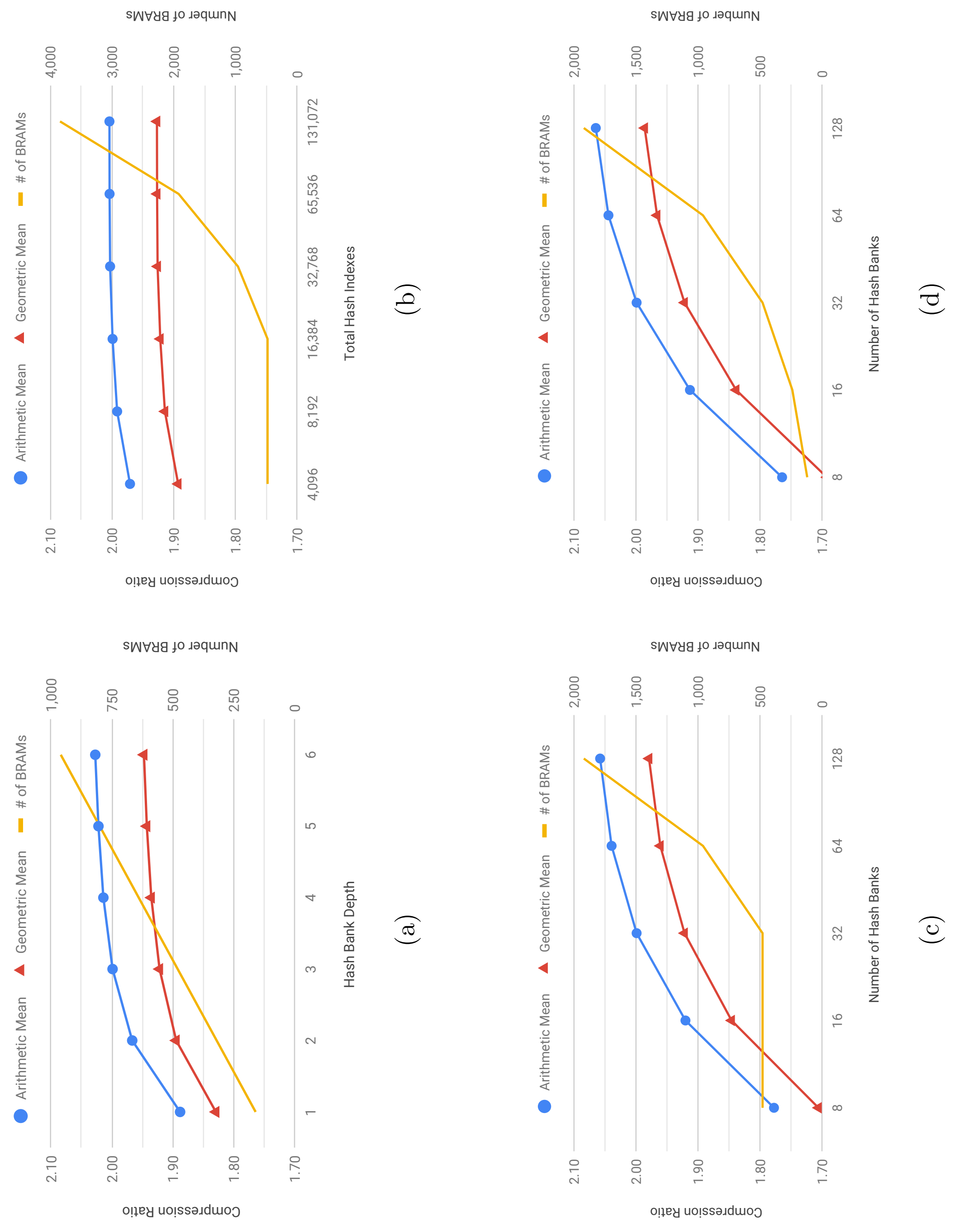


Figure 3.9: Effect of hash bank architecture on compression ratio and memory cost.
a) Varying hash bank depth with 32 hash banks and 16,384 total indexes.
b) Varying total hash indexes with 32 hash banks and 3 depth levels.
c) Varying number of hash banks with 16,384 total indexes and 3 depth levels.
d) Varying number of hash banks with 512 indexes per bank and 3 depth levels.

the compressor, then it could choose to use a hash function that is likely to be better suited for matching that type of data, which would on average increase the likelihood of better compression results. Another option for improving the compression ratios would be to utilize dynamic Huffman encoding as well as static. As mentioned in Section 2.3.1, however, this complicates the pipeline design and removes the fixed throughput rate, as an intermediate storage space would be required to hold the LZ77 encoded data before the Huffman tables could be constructed and used. This type of work was investigated in [29]. The other option to improve the compression ratios is to return to a traditional compressor design, like the ones used in software, which take more time to perform match finding, as these designs still are able to provide the highest compression ratios.

To increase the compression throughput further (besides increasing the PWS value), higher clock frequencies may be used or, as recommended by [7], multiple independent compressors could be utilized in parallel. Since we were able to achieve a clock frequency of 250 MHz while being restricted to high-level synthesis, it may be possible for hand-optimized VHDL or Verilog designs to meet even higher frequencies. As we discussed earlier, this may require sacrificing some compression ratio in order to scale the hash bank architecture down enough so that routing delays are more manageable. In terms of logic delay, the bank-access allocation stages and the best-match-length selection stage were the two largest critical path areas of the LZ77 encoder design. For our current FPGA-based compressor design, with a total BRAM usage of about 36%, it may be possible to fit two independent compressor cores on a single FPGA to operate in parallel. Together they would be able to provide a collective compression throughput of 8.0 GB/s, assuming they aren't bottlenecked by the FPGA interface.

3.5 Summary

In this chapter, we described our FPGA-based Deflate compressor design. This design followed the current state-of-the-art architecture, which utilizes a fixed-throughput pipeline and static Huffman encoding in order to rapidly perform Deflate compression. The use of Vivado HLS provided some challenges in creating a working design that was able to meet timing, but was also able to help improve the efficiency of our design through certain automatic optimizations. Despite the addition of match trimming functionality to the LZ77 encoder as well as extensive exploration on the hash function and hash bank architecture used, we weren't able to achieve higher compression ratios than previous works. Our compressor was able to achieve arithmetic and geometric mean compression ratios of 2.00 and 1.92, respectively, across the Calgary corpus benchmark. We were, however, able to meet clock timing with a clock frequency of 250 MHz, giving our compressor a throughput of 4.0 GB/s, which is higher than all other currently published single-compressor throughputs. We were also able to achieve this while being limited to using high-level synthesis.

As discussed in Section 3.4, we believe that we have still not hit the maximum limits of compression ratios achievable for these type of designs. Further exploration of possible hash functions may yield functions that are better able to prevent hash bank access conflicts from occurring. In terms of hash bank architecture, there are still many ways that compression ratios can be improved at the cost of additional area, though they may potentially require a shorter clock frequency (and therefore lower throughput) in order to meet post-route timing. Currently, instantiating multiple compressor cores in parallel is still the most efficient way to scale up compression throughput, though more efficient and higher frequency single-core compressor designs should also be achievable.

Chapter 4

Decompressor Design

Our goal in this chapter is to design a competitive FPGA-based decompressor core using Vivado HLS. The decompressor is to be fully compliant with the Deflate standard, to have the ability to process both static and dynamic Huffman-encoded blocks, and to not exploit any changes to the compressed format or make any limiting assumptions about the compressed data. In our design we focused on maximizing the decompression throughput by exploiting the parallel resources available in FPGAs. As we describe our initial and final decompressor designs, we will also mention the main limitations of Vivado HLS that we encountered and also explain how we were able to work around them in order to succeed in achieving our design goals.

4.1 Initial Design

Our initial decompressor architecture, shown in Fig. 4.1, utilizes speculative parallelization to perform the Huffman decoding operation in parallel in order to benefit from some potential speed-up. The architecture includes multiple Huffman decoder cores capable of processing static Huffman blocks only. In our design we chose to use four Huffman decoders, though any number of parallel decoders could be utilized. A "boundary detector" core at the beginning of the system pipeline scans the incoming data for the static EOB code in order to find potential block boundaries. The "data mover" core receives the data from the boundary detector (following temporary storage in a FIFO) and writes it to one of four local BRAM memories, one for each Huffman decoder. The data

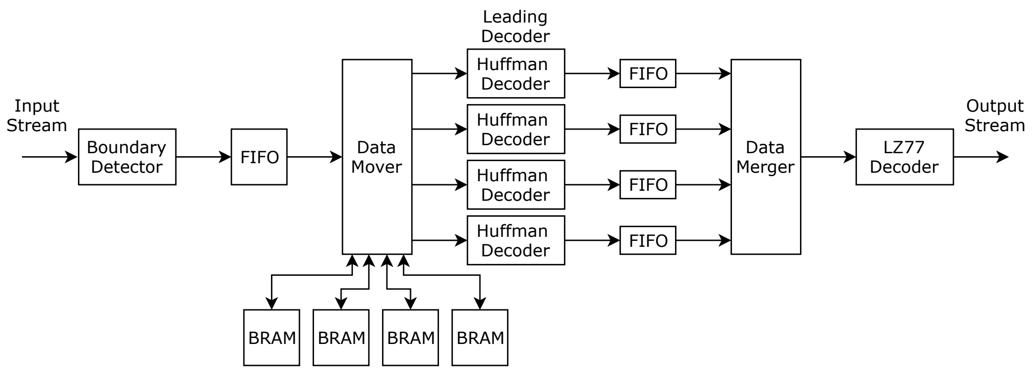


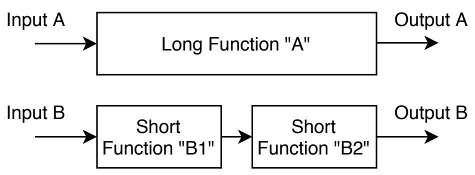
Figure 4.1: Initial speculative decompressor architecture.

mover has the ability to copy data from each of the BRAMs and stream it to one of the decoders simultaneously, in order to keep each decoder busy. The purpose of the BRAMs is to store the potentially parallelizeable data and, in the result of a false boundary being discovered (and data being decoded incorrectly), allows the data to be resent to another decoder in order to be processed correctly.

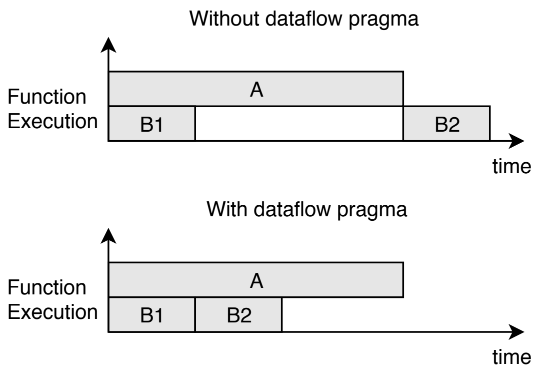
The first Huffman decoder to be given data will be referred to as the "leading decoder" and is guaranteed to correctly decode the first block. Whenever a potential block boundary in the incoming data is found, a copy of the data stream from that point is also sent to another Huffman decoder to run in parallel with the leading one. The outputs of each Huffman decoder feed into FIFOs for elastic storage so that the validity of the data can be verified later on (depending on if the prospective block boundary turns out to be valid or not). If the block boundary is valid, the Huffman decoded data is sent to the "data merger" core. The data merger merges the multiple parallel data streams and continuously streams the next set of valid data from one of the FIFOs to the LZ77 decoder core. If the prospective block boundary that a Huffman decoder was starting from is found to be invalid, the corresponding FIFO is cleared and its data is discarded.

The decoded data from the leading Huffman decoder is always valid as this is the decoder that is known to be performing correct sequential decoding from the start of the file or from a confirmed block boundary. When a prospective block boundary is found and confirmed to be valid, the following block is processed in parallel correctly and the Huffman decoder that processed it becomes the new leading decoder as it now knows the correct bit position from which to continue decoding the data stream. The output block from the previous leading decoder is kept and it will now be given potentially parallel data at the next opportunity. No matter how many false or positive potential block boundaries are found, one decoder is always kept running and the file will be decompressed sequentially in the worst case. More Huffman decoders could be employed in parallel if desired. The amount of parallel Huffman decoders that will be useful, though, will depend on the speed at which they can decode and the size of the Deflate blocks in the compressed data. If the blocks in the data are small and/or the decoders are fast enough, employing more parallel decoders will be pointless as they will likely never be used. Determining the optimal number of Huffman decoders to use will require design exploration and testing on various data types.

The two data passing cores, the data mover and data merger, were originally to be created using Vivado HLS. We found, however, that the parallel nature of their operation, made implementation in Vivado HLS difficult if not impossible. This is because of one crucial limitation of Vivado HLS which we have called “coarse function scheduling”. Different functions within an HLS core can be run in parallel, but they need to be scheduled to start at the same time, as illustrated in Fig. 4.2. This prevents the data mover and data merger cores from performing multiple asynchronous tasks in parallel as required. There is one way to overcome this limitation: the Vivado HLS “dataflow” pragma (a compiler directive) allows data to flow freely between function blocks; however, the use of this pragma imposes many strict constraints on the code to be synthesized. We were unable to meet these restrictions, which forbid the use of conditionally performed tasks as well as feedback between blocks. It may also be possible to split up the data passing cores into multiple HLS cores such that each operation can be carried out in parallel, but each of the sub-cores would need to be interfaced properly and be able to respond to interrupt signals. We chose to avoid designs that required complex



(a) Function data flow diagram



(b) Function execution timing

Figure 4.2: Diagram showing the coarse function scheduling limitation of Vivado HLS. a) Function chains A and B1/B2 have separate inputs and separate outputs and are completely independent from each other. b) Without the dataflow pragma, function B2 will not be scheduled to start until function A finishes. With the dataflow pragma, B2 executes as soon as B1 is finished.

interrupt handling. For these reasons, the data mover and data merger cores were written in structural level Verilog instead of using Vivado HLS, to allow us to have direct control over the structure of this part of the design.

As mentioned above, when operating on random data, the amount of Huffman decoders that would be utilized simultaneously would be dependent on the size of the Deflate blocks in the data. To compensate for this, our next plan was to modify the accompanying compressor to compress data into blocks of constantly known size which we could then take advantage of using the decompressor. This way, the data could be safely and consistently split into multiple blocks and all four (or possibly more) Huffman decoders could be fully utilized. Once the initial design was made to work, however, it was found that the downstream LZ77 decoder was unable to keep up even with only one Huffman decoder working. At this point, the decision was made to scrap the speculative parallelization architecture for the Huffman decoders and instead focus on optimizing the two main cores of the decompressor: one serial Huffman decoder and the following LZ77 decoder. The Huffman decoder was upgraded to include the ability to process dynamic Huffman blocks as well, which are significantly more complicated to process than static blocks. Our focus thus shifted from designing a fast, static-only Deflate decompressor that relied on fixed block sizes to designing a fast, fully Deflate-compliant decompressor capable of processing both static and dynamic blocks, and that made no assumptions about the block sizes in the compressed data it was given.

4.2 Final Design

A block diagram of the resulting final decompressor design is shown in Fig. 4.3. It is composed of two main cores: the Huffman decoder and the LZ77 decoder, as well as two byte-reordering cores, called the “literal stacker” and the “byte packer”. The purposes of these two byte-reordering cores are described below in Sections 4.2.2 and 4.2.4. All four of the cores share the same 250 MHz clock. A balanced and efficient design was sought where the output throughput of the Huffman decoder would match the input throughput of the LZ77 decoder. A data FIFO was instantiated in the middle of the chain in order to alleviate stalling between the two decoders by providing elastic storage, as explained further below. As with our compressor design (in Chapter 3), all four cores are specified in C++ and synthesized using Vivado HLS and are interconnected with standard AXI-Stream interfaces [42]. The data width (TDATA) of all of these interfaces is 32 bits to accommodate the passing of four bytes at a time, in little-endian order. The input data width of the Huffman decoder could be increased further but was chosen to be 4 bytes in order to match the rest of the system, which was determined by the 4-byte processing capabilities of the LZ77 decoder, as explained below. The data signals are accompanied by AXI-Stream sideband signals TKEEP, TUSER, and TLAST. The 4-bit TKEEP signal is used to identify which of the four data bytes are valid in the current transfer and the TLAST signal is used to identify the last transfer of the data stream. The TUSER signal is used to differentiate between AXI-Stream

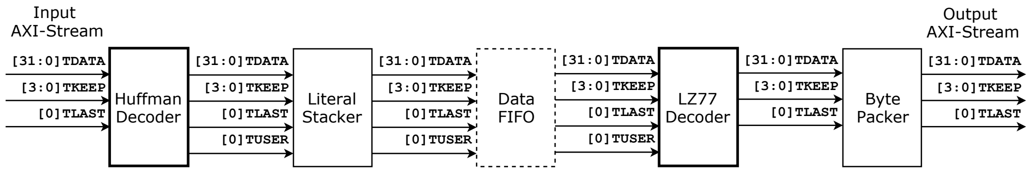


Figure 4.3: Final non-speculative decompressor architecture.

transfers containing literals and those containing length-distance pairs. The FIFO has a depth of 4096, which allows it to hold 4096 AXI-Stream transfers.

4.2.1 Huffman Decoder

The first core in the decompressor is the Huffman decoder, which is responsible for decoding the Huffman codes and outputting them as literals and length-distance pairs. This core differs from the others in that it uses a Finite-State Machine (FSM) architecture instead of a pipelined one. The state diagram is shown in Fig. 4.4. From the starting state, it begins by reading data from the input stream, identifying the Deflate block type from the block header, and then entering the appropriate block processing state: static, dynamic, or stored. Every time data is read from the input stream (i.e., 32 bits), it is placed into a 64-bit buffer called the accumulator. The Huffman decoder only reads new data from the input stream when it runs out of enough bits to decode the next code. Increasing the input stream data width to values larger than 32 bits would mean the Huffman decoder could go longer before reading more input data. The newly read 32 bits are appended to the left side of the accumulator (above any currently held bits) using a shift-OR operation while processed bits get shifted out the right side of the accumulator, as required. For example, after decoding a 9-bit Huffman code and determining that the code was 9 bits long, those 9 bits are then shifted out, all at once, from the accumulator. In the stored block state, the decoder will read the block length,

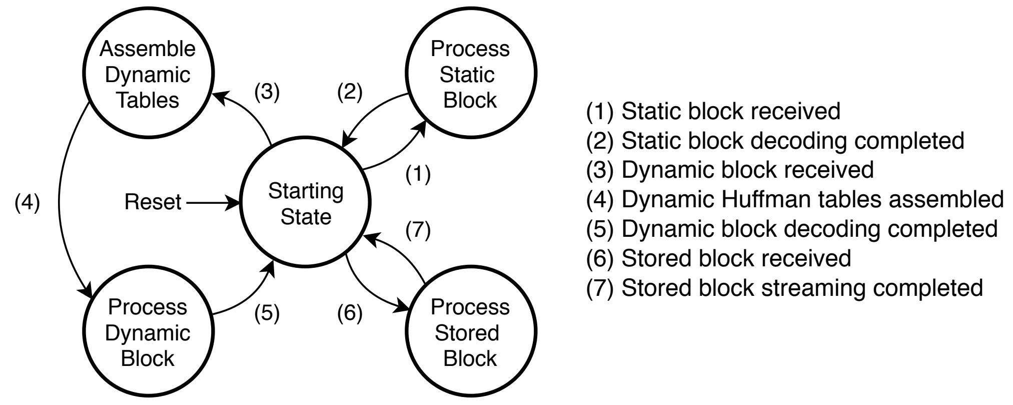
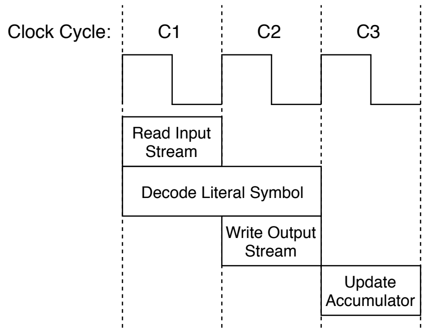

Assemble Dynamic Tables
Process Static Block
Starting State
Reset
Process Dynamic Block
Process Stored Block
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(1) Static block received
(2) Static block decoding completed
(3) Dynamic block received
(4) Dynamic Huffman tables assembled
(5) Dynamic block decoding completed
(6) Stored block received
(7) Stored block streaming completed

Figure 4.4: Huffman decoder state diagram.

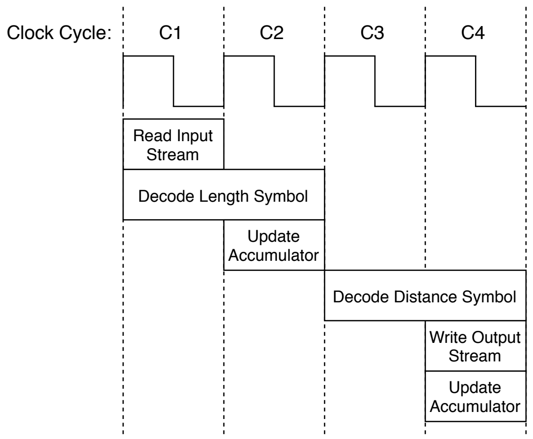
LEN, and block length complement, NLEN, and verify that the two numbers are complementary (See Fig. 2.3 for the stored block structure). It will then stream LEN bytes from the input to the output. In the static block state, the decoder will continually decode static codes using the static ROM tables until it encounters the EOB code, at which point it returns to the starting state to be ready to process the next Deflate block.

The process for decoding static codes is similar to the process described in Section 2.3.1 when describing reference [38]. From the accumulator, 9 bits are read and looked up in a 512-index literal/length LUT. The table returns the type of the decoded symbol (literal, length, or EOB), the number of bits in the code (7, 8, or 9), the number of extra bits following the code, and the value of the symbol (literal value or base length value). If the symbol is a literal, it is copied unchanged to the output. If the symbol is a length, then the associated extra bits are read from the accumulator as well as the 5-bit distance code that follows that. The distance code is looked up in a 32-index LUT, which returns the distance base value and the number of extra distance bits. The final 9-bit length and 16-bit distance values are calculated by adding the base values to the values of the extra bits and are written together to the 32-bit output stream. In order to tell literals apart from length-distance pairs, the 1-bit AXI-Stream TUSER signal is used. It takes the decoder 3 clock cycles to decode a static literal and 4 clock cycles to decode a static length-distance pair, as shown in Fig. 4.5. Note that these operations are not performed pipelined due to dependencies on the accumulator. In order to meet timing while pipelined, Vivado HLS synthesizes the pipeline with an II of 4 (input/output data being read/written every 4 clock cycles), which results in slower operation than a non-pipelined architecture.

When the decoder encounters a dynamic block, it processes it using two states. In the first state, it will read the sequence of code lengths and assemble the dynamic Huffman tables. Then, in the second state, it will use those code tables to decode the dynamic block. In order to build the dynamic tables, the decoder follows the five-step process described in Section 2.1. The table building process takes a variable amount of time that depends on the number



(a) Static literal processing



(b) Static length-distance pair processing

Figure 4.5: Huffman decoder static timing diagrams.

of code lengths in the code length sequence, the number of code lengths in the literal/length sequence, and how the literal/length code length sequence has been run-length encoded. Once the dynamic tables have been constructed and stored in BRAMs, the decoder enters the next state where it decodes the dynamic block.

As mentioned in Section 2.3.2, we used the Huffman table design from [40], which allows us to use a 286-index table for literal/length codes and a 32-index table for distance codes instead of two much larger 32,768-index tables. The process for decoding a dynamic Huffman code using this architecture is as follows: During Huffman table construction, the base values and base addresses for each of the 15 different code lengths are recorded. In order to decode a code, 15 bits are taken from the accumulator and reversed (since the Huffman codes are stored in reverse order). The code bits are then compared to each of the 15 base code values, which are left-aligned to see if they are greater-than or equal to them. An example of this process is shown in Fig. 4.6. These comparisons are all performed in parallel and the highest passing comparison indicates the length of the code. It is unlikely that all 15 different code lengths will be used in the same Dynamic Huffman table so only comparisons for the code lengths that are used are performed. Once the bit-length of the code is known, the base address for its length can be retrieved. The actual code index

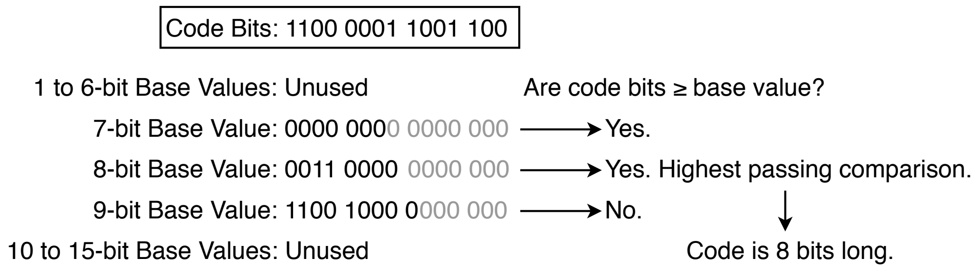
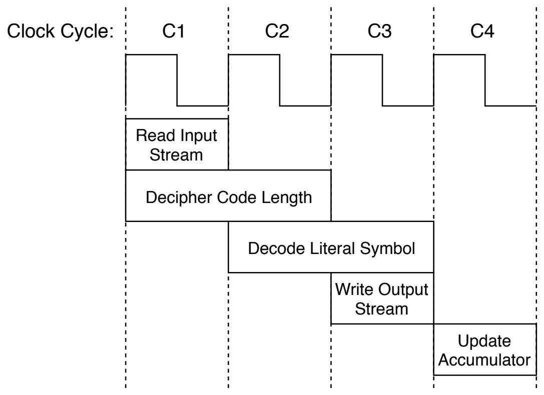

Code Bits: 1100 0001 1001 100
1 to 6-bit Base Values: Unused
Are code bits ≥ base value?
7-bit Base Value: 0000 0000 0000 000
Yes.
8-bit Base Value: 0011 0000 0000 000
Yes. Highest passing comparison.
9-bit Base Value: 1100 1000 0000 000
No.
10 to 15-bit Base Values: Unused
Code is 8 bits long.

Figure 4.6: An example of how the length of a code is deciphered during decoding using the area-efficient Huffman table memory design from [40]. In this example, the Deflate static codes (from Table 2.1) are used for simplicity. The resulting decoded symbol in this example is length symbol 281.

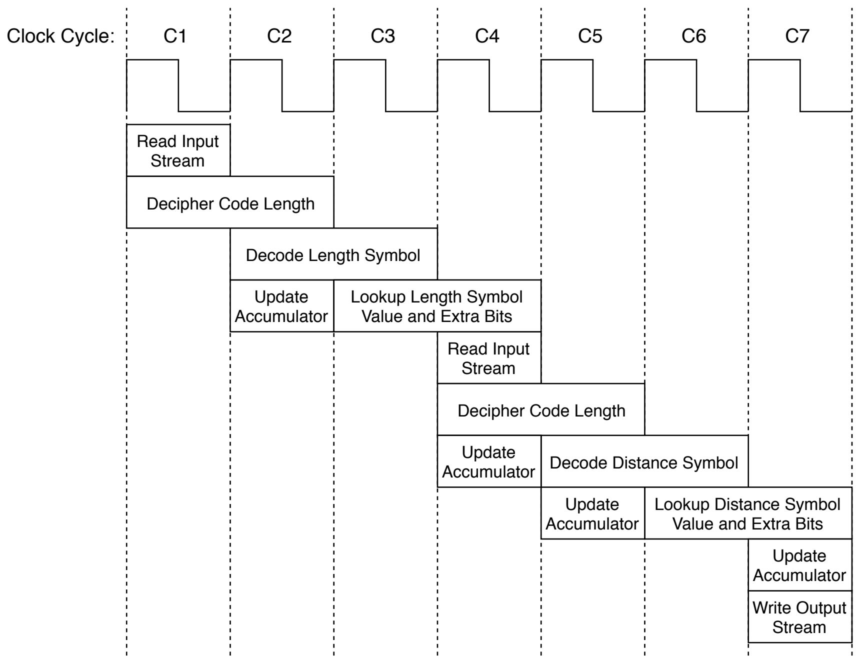
is calculated by adding an offset to the base address, as shown in Fig. 2.9. This offset is obtained by subtracting the code length base value from the code bits. Once the code has been decoded, the decoder follows the same procedure as it does during static decoding: If the symbol is a literal, the literal is copied unchanged to the output stream. If the symbol is a length, the extra bits are retrieved and then the distance code is decoded using the same above process. The Huffman decoder takes 4 clock cycles to decode a dynamic literal, and 7 clock cycles to decode a dynamic length-distance pair, as shown in Fig. 4.7. As with the static decoding operations, these operations are not pipelined due to dependencies on the accumulator.

4.2.2 Literal Stacker

As will be explained further in Section 4.2.3, the LZ77 decoder has the ability to read four bytes and write four bytes in one clock cycle. While the LZ77 decoder is processing a length-distance command, it will stop reading input bytes, which causes data to start piling up inside the preceding FIFO. In order to: a) more effectively utilize the FIFO's capacity, and b) help the LZ77 decoder catch up on any data backlog, the literal stacker is included in the system pipeline. This module takes the single-byte literals output by the Huffman decoder and consolidates them into stacks of up to four bytes, as shown in Fig. 4.8. Any length-distance pairs in the stream, which occupy



(a) Dynamic literal processing



(b) Dynamic length-distance pair processing

Figure 4.7: Huffman decoder dynamic timing diagrams.

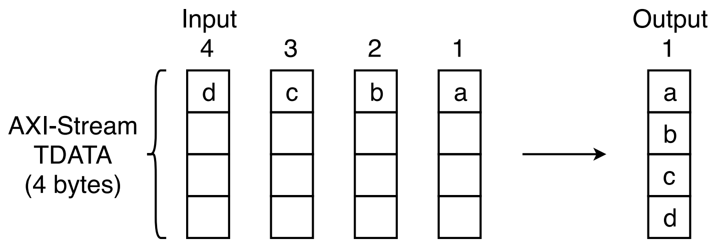


Figure 4.8: Literal stacker operation. In this example, four consecutive literal bytes occupying four consecutive AXI-Stream data transfers are combined to occupy one transfer.

all four bytes of the data width, will force the literal stacker to release its currently held stack of literals. The literal stacker uses the 4-bit AXI-Stream TKEEP signal, as it is intended to be used in the AXI-Stream standard [42], to indicate the number of literals in a stack, from one to four. The literal stacker is pipelined with an II of 1, meaning it can read new inputs in every clock cycle, and it has 2 stages.

4.2.3 LZ77 Decoder

The LZ77 decoder's main purpose is to resolve the length-distance pairs received from the Huffman decoder. It does this using a circular buffer BRAM that has 32,768 one-byte indexes, enough for storing the last 32,768 literal bytes, which is a necessary capacity since Deflate distances can back-reference up to that amount. When the LZ77 decoder receives literals, it records them in the circular buffer for future reference and also writes them out to the output (uncompressed) data stream. When it receives a length-distance pair, it looks back a "distance" number of bytes into the circular buffer and copies a "length" number of them to the head of the circular buffer as well as to the output. While copying the bytes of a length-distance pair, no more inputs are read, which usually causes incoming data to pile up safely in the FIFO before it.

As mentioned above, the LZ77 decoder has the ability to process four bytes in every clock cycle, meaning it can write one to four literals to the circular buffer at a time or copy one to four bytes of a length-distance pair at a time.

Originally, it was only able to process one byte per cycle, as done by the LZ77 decoder in [38]. We found that the LZ77 decoder was functioning too slowly to keep up with the Huffman decoder because length-distance pairs were taking longer than expected to resolve (e.g., a match length of 16 bytes would stall the LZ77 decoder for 16 cycles). In order to allow the LZ77 decoder to read and write multiple bytes to the circular buffer in every clock cycle, we used cyclic partitioning. By cyclically partitioning the circular buffer BRAM by a factor of four, we split it into four separate BRAMs where each BRAM holds every fourth byte, as shown in Fig. 4.9. This way, four consecutive indexes in the circular buffer can be read from and written to in every clock cycle (eight simultaneous BRAM accesses altogether). The BRAMs in our assumed Xilinx FPGA fabric are dual-port, meaning they can be read from and written to simultaneously in the same clock cycle.

In Vivado HLS, the partition pragma can be used on an array to automatically partition it into multiple BRAMs. When using this pragma, however, we found that Vivado HLS was unable to schedule the four reads and four writes simultaneously as it could not recognize that we were accessing four consecutive memory locations, one from each partition. Many different coding schemes were tried but the tool would only ever schedule the eight BRAM accesses across four clock cycles, with one read and one write per cycle. In the end, we ended up manually partitioning the array by simply using four

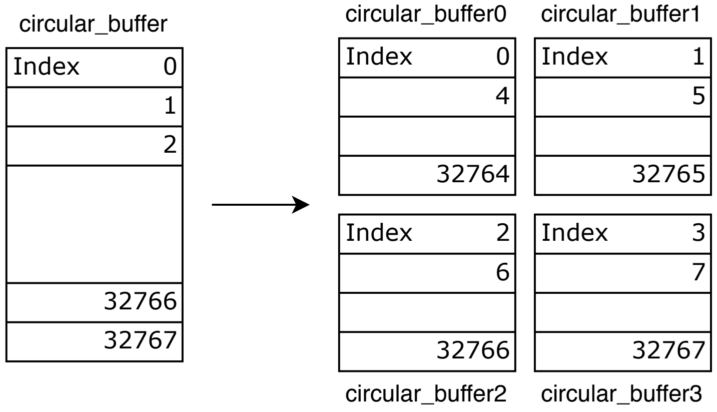


Figure 4.9: Cyclic partitioning of the LZ77 decoder circular buffer BRAM into four disjoint sub-buffers.

different arrays. This way, Vivado HLS was able to schedule the four reads and four writes to the circular buffer partitions in every clock cycle, as desired.

We will refer to the four consecutive addresses read or written (one address per memory partition) as the read and write “windows”. These are essentially just read and write pointers that span the width of four memory locations. During operation, these two windows will slide across the circular buffer as multiple bytes are read from the read-window location and written to the write-window location in every clock cycle. In the function pipeline, read and write operations are performed on the circular buffer simultaneously in every clock cycle. In terms of latency, however, the read operation takes place one stage before the write operation does in the pipeline, which can lead to memory access conflicts (i.e., when the same memory address is read from and written to simultaneously) in certain situations.

When the distance value of a length-distance pair is 5, 6, or 7, the read-window will collide with the write-window on the second iteration if it is allowed to access all four partitions. To prevent this, we programmed the read-window to keep track of how many bytes it is allowed to read before it must wrap-around to its starting location. The LZ77 decoder keeps track of both the number of length-bytes remaining (the total number of bytes to be copied) and the number of distance-bytes remaining (the number of bytes allowed to be read before the window is reset). Both of these numbers are used to determine how many reads and writes to perform in every cycle so that conflicts do not occur. An example of this is shown in Fig. 4.10.

When the distance value is from 1 to 4, the decoder enters a special state where only one read operation is performed. The decoder reads 1 to 4 bytes and then stores them in four registers, which are then used to write the bytes back to the circular buffer as many times as needed. If the distance is 1, for example, a single byte is read and copied to all four registers, allowing that byte to be written back to the circular buffer four times in every clock cycle instead of just once per cycle. An example of this is shown in Fig. 4.11. When the distance is 32,768, the read and write windows will point to the same location since the circular buffer only has 32,768 indexes. In order to prevent

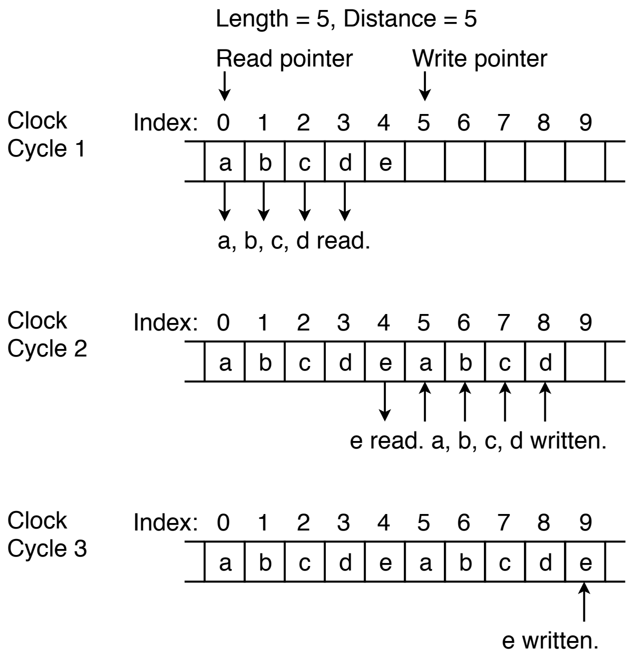


Figure 4.10: Example showing LZ77 circular buffer access behaviour with a length of 5 and a distance of 5.

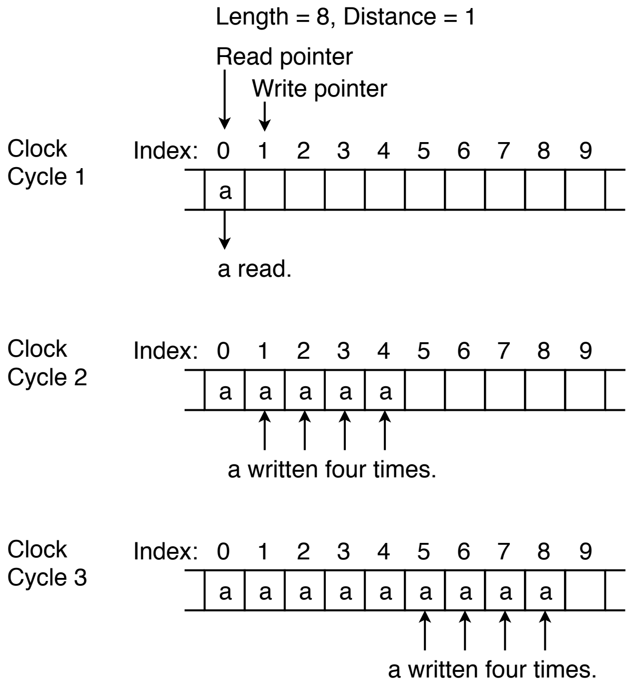


Figure 4.11: Example showing LZ77 circular buffer access behaviour with a length of 8 and a distance of 1.

the decoder from redundantly reading a set of bytes and writing them back to the same location, writes are prevented whenever this scenario is detected. The LZ77 decoder will simply read the bytes and write them to the output stream, while shifting the write-window forward.

The most difficult problem that we encountered when designing the LZ77 decoder was getting the pipeline to synthesize with an II of 1 with the four-byte version in Vivado HLS. With the original one-byte version, the entire decoder was able to be pipelined by Vivado HLS with an II of 1 without problems. The four-byte version, however, would not pipeline with an II of less than 2. Initially, we were able to pipeline certain portions of the function, like the length-distance pair copying loop, but never the entire function. The performance suffered in this case due to delays incurred from frequent switching between literal operation and length-distance pair operation. Many different code iterations were tried involving various different changes in order to experiment and determine how to get the code to synthesize as desired. These changes included: separating out parts of the code into their own functions (this seems to help Vivado HLS by breaking down the problem into smaller pieces since sub-functions are synthesized on their own first before being incorporated in the top-level function), experimenting with the number of different states in the code, rewriting the code in different but functionally-equivalent ways, and changing the function interfaces to different types. In the end, a combination of the above changes led to a version of the LZ77 decoder that was able to pass synthesis with a pipeline II of 1 and a depth of 5 stages.

4.2.4 Byte Packer

Since the LZ77 decoder can output any number of bytes from one to four, the output stream will be filled with null bytes. A null byte is defined to be a byte that is either empty or that contains non-useful information that occupies one byte location in an AXI-Stream transfer [42]. These null bytes must be removed from the stream in order to create a continuously aligned (i.e., fully occupied) output stream. The byte packer performs this operation by using the TKEEP signal to identify which bytes are null bytes to be filtered out from

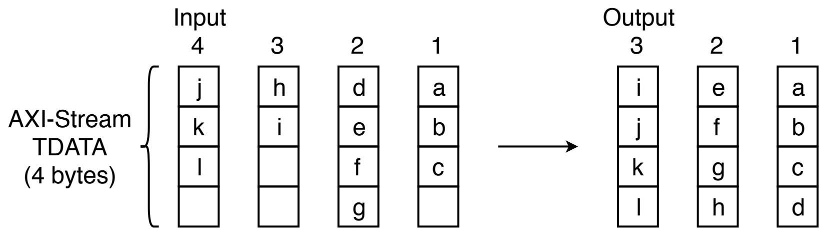


Figure 4.12: Byte packer operation. In this example, AXI-Stream transfers of various occupancies (3, 4, 2, and 3 bytes) are combined into a continuous aligned stream with full occupancies (4 bytes).

the stream. When a transfer of less than four bytes is received, it will align the bytes and hold on to them until more data is received, outputting only full four-byte transfers until the end of the stream is encountered, as shown in Fig. 4.12. The byte packer is pipelined with an II of 1 and a depth of 3 stages.

4.3 Testing and Results

The four decompressor cores were synthesized using Vivado HLS for a clock frequency of 250 MHz. As with our compressor design in Chapter 3, we synthesized our design for a Xilinx Virtex UltraScale+ XCVU3P-FFVC1517 FPGA, the same device used by our industrial collaborator, Eideticom. Table 4.1 shows the FPGA resource utilization of the four cores. We compressed the Calgary corpus benchmark files using the zlib software in order to create a test set to allow us to test the decompressor. As described in Section 2.2, two sets of files were created: one set compressed using static Huffman codes only, referred to as static files, and one set compressed using the default zlib settings

Table 4.1: Decompressor FPGA Resource Utilization

	LUTs	FFs	BRAM Tiles
Huffman Decoder	6,572 (1.67%)	4,975 (0.63%)	1.5 (0.21%)
Literal Stacker	261 (0.07%)	107 (0.01%)	0
FIFO	99 (0.03%)	139 (0.02%)	5 (0.69%)
LZ77 Decoder	2,243 (0.57%)	820 (0.10%)	8 (1.11%)
Byte Packer	1,013 (0.26%)	389 (0.05%)	0
Total	10,188 (2.59%)	6,430 (0.82%)	14.5 (2.01%)

(which used only dynamic Huffman codes), which we refer to as dynamic files. Before testing, the zlib headers and footers were removed from the files. As with our compressor design, we used a Vivado testbench containing an AXI DMA core, which was used to stream the compressed input data from memory to the decompressor core, receive the uncompressed output data stream, and then write it back to memory for verification. We measured the run time of the decompressor from the moment it receives the first input AXI-Stream transfer to the moment when it outputs the last uncompressed transfer. To obtain the input throughput, the compressed file size is divided by the decompression time. This is not an instantaneous throughput, but an average across the decompression time of the file. This value can be converted to an output throughput by multiplying it by the compression ratio of the file.

The results for the dynamic and static file decompression are shown in Tables 4.2 and 4.3, respectively. The maximum and minimum input throughput values in each table are shown in bold font. On the dynamic files, a maximum throughput value of 76.12 MB/s and a minimum throughput value of 62.74 MB/s were achieved, giving an overall average input throughput of 70.73 MB/s. On the static files, a maximum throughput of 151.64 MB/s and a minimum throughput of 101.16 MB/s were achieved, giving an average input throughput of 130.58 MB/s. In both sets of files, "book1" and "book2" gave the highest throughput values while "obj1" and "pic" gave the lowest throughput values. The fact that book1 decompressed at a faster rate than book2 when statically compressed but at a slower rate when dynamically compressed implies that the dynamic Huffman tables of book1 took longer to assemble than those from book2 did during dynamic decompression. The same pattern is seen between files obj1 and pic, with pic decompressing faster when dynamically compressed and obj1 decompressing faster when statically compressed. The input throughput values for the static files are about double those of the dynamic files. This is caused by both the delay incurred when assembling the dynamic Huffman tables as well as the fact that the Huffman decoder takes almost twice as long to decode a dynamic length-distance pair (static: 4 cycles, dynamic: 7 cycles). This large performance increase between

Table 4.2: Dynamically-Encoded Calgary Corpus Decompression Results

Compressed File	Compressed Size (bytes)	Compression Ratio	Decompression Time (μs)	Input Throughput (MB/s)
bib	35,222	3.16	471.948	74.63
book1	313,576	2.45	4197.576	74.70
book2	206,658	2.96	2714.984	(max) **76.12**
geo	68,427	1.50	1040.708	65.75
news	144,794	2.60	2017.096	71.78
obj1	10,311	2.09	164.356	(min) **62.74**
obj2	81,499	3.03	1155.584	70.53
paper1	18,552	2.87	255.952	72.48
paper2	29,754	2.76	402.156	73.99
pic	56,459	9.09	896.580	62.97
progc	13,337	2.97	191.152	69.77
progl	16,249	4.41	224.060	72.52
progp	11,222	4.40	160.564	69.89
trans	19,039	4.92	262.912	72.42

Table 4.3: Statically-Encoded Calgary Corpus Decompression Results

Compressed File	Compressed Size (bytes)	Compression Ratio	Decompression Time (μs)	Input Throughput (MB/s)
bib	40,931	2.72	292.220	140.07
book1	384,953	2.00	2538.596	(max) **151.64**
book2	243,843	2.51	1643.424	148.37
geo	80,949	1.26	668.172	121.15
news	168,375	2.24	1266.768	132.92
obj1	11,138	1.93	109.392	101.82
obj2	88,949	2.77	733.724	121.23
paper1	21,670	2.45	157.276	137.78
paper2	35,499	2.32	241.500	146.99
pic	67,529	7.60	667.552	(min) **101.16**
progc	15,365	2.58	117.756	130.48
progl	18,603	3.85	138.320	134.49
progp	12,771	3.87	97.792	130.59
trans	21,424	4.37	165.472	129.47

Table 4.4: FPGA-based Decompressor Design Comparison

Reference	Throughput	Area Utilization	Test Files
[38] (2007)	158.64 MB/s (avg., output)	387 LUTs, 128 FFs, 18 RAMB16s	Static Calgary corpus
[39] (2009)	125 MB/s (max, input)	20,596 LUTs, 22 BRAM18Ks	Unknown test files
[40] (2010)	300 MB/s (max)	Not disclosed	Unknown test files
[41] (2016)	375 MB/s (avg., output),	8,250 LUTs, 58 BRAM18Ks	Unknown dynamic test files
	495 MB/s (avg., output)	5,392 LUTs, 21 BRAM18Ks	Unknown static test files
This work	70.73 MB/s (avg., input),	10,188 LUTs, 6,430 FFs,	Dynamic Calgary corpus
	246.35 MB/s (avg., output),	29 BRAM18Ks	
	130.58 MB/s (avg., input),		Static Calgary corpus
	386.56 MB/s (avg., output)		

static and dynamic files suggests that static compression should be used when rapid decompression is required, assuming that the corresponding decrease in compression ratio is acceptable. This aligns with the current trend of FPGA-based compressors only performing static compression in order to maximize the compression throughput, as seen in other works (described in Section 2.3.1) and in our own compressor (described in Chapter 3).

The results of our work and of previous FPGA-based decompressor designs are summarized in Table 4.4. Due to the unspecified test files that were used to evaluate other published decompressor designs, it is difficult to make a fair comparison with most previous works. As shown by our results, the type of data and amount of compression it has undergone make a big impact on the decompression throughputs that are achievable. If we convert our average input throughput values of 70.73 MB/s (dynamic) and 130.58 MB/s (static) to output throughput values, we get 246.35 MB/s (dynamic) and 386.56 MB/s (static). These values are comparable to the 375 MB/s (dynamic) and 495 MB/s (static) average output throughputs of the proprietary Xilinx IP core [41]. Compared to the static average output throughput of 158.64 MB/s from [38], our design is about 2.44 times faster, which is a tribute to the capabilities of high-level synthesis. In terms of resource utilization, our design uses half as many LUTs but one third more BRAMs compared to [39]. Compared to the Xilinx IP core [41], our design uses about 23% more LUTs but half as many BRAMs. We can only speculate since no design details are shared in [41], however, the extra BRAMs may be used in some sort of speculatively parallelized architecture.

As mentioned above, the FIFO in our decompressor has a depth of 4096, meaning it can store up to 4096 transfers containing either literals or length-distance pairs. The FIFO count can be used as a measure of the amount of stalling done by the LZ77 decoder. For most static and dynamic files, the maximum FIFO count rarely went above 30, demonstrating the LZ77 decoder's ability to keep up with the Huffman decoder. The one exception to this is the benchmark file pic, which had a dynamic max FIFO count of 1441 while the static file maxed out the FIFO at 4096. The reason for this is due to the pic file's very high compression ratio which is a result of it being compressed using many length-258 length-distance pairs. Each of these length-distance pairs takes the LZ77 decoder 65 clock cycles to resolve with it copying four bytes per cycle. The LZ77 decoder could be improved by expanding the window width even further to be able to copy eight or sixteen bytes per cycle. However, the decompression throughput would not increase significantly, except for with highly compressed files like pic or unless the Huffman decoder was improved significantly as well. The purpose of the literal stacker module was to help the LZ77 decoder to catch up on any backlogged data in the FIFO. We found that the presence of the literal stacker was able to provide throughput improvements ranging from 0 to 4%, with the biggest benefit occurring while decompressing the pic file. This speedup was also mainly seen when processing static files, as the dynamic table construction phase gives the LZ77 decoder opportunities to catch up when processing dynamic files. Due to the small added latency (2 pipeline stages) and area overhead of the literal stacker, we believe this to be a worthwhile trade-off. This speedup benefit would increase even further with a higher-throughput Huffman decoder, which would put more pressure on the LZ77 decoder.

4.4 Discussion

From the timing diagrams in Figs. 4.5 and 4.7, we can identify areas where the performance of the Huffman decoder could be improved. The most obvious improvements are in the processing of static and dynamic literals in Fig.

4.5. The “Update Accumulator” operation should be able to be performed one cycle earlier in both cases. This highlights another limitation of Vivado HLS, when the way a piece of software code is written results in inefficient synthesis for non-obvious reasons. In this case, it appears that the presence of the length-distance pair processing code causes unnecessary caution for Vivado HLS in the synthesis process. We verified this by removing the length-distance pair code from the Huffman decoder and, as expected, it was then able to write static literals every 2 cycles and dynamic literals every 3 cycles. Unfortunately, some experimentation with the syntax of the C code may be required to overcome such synthesis problems like these in order to achieve the expected performance.

Another improvement would be to perform the lookup of the distance code at the same time as the length code, for static length-distance pairs, as the second lookup forces an additional cycle of latency. This would require multiple prospective lookups since the bit-length of the length code is unknown until after it has been decoded. Once known, however, a MUX could be used to select the correctly decoded distance. The performance of the Huffman decoder is not necessarily limited by the critical path delay, but rather by the lookup delay since a table lookup must be performed across two clock cycles (i.e., in the first clock cycle the address is given to the table, in the second cycle the data is received from the table). Therefore, a literal will never take fewer than 2 clock cycles to output due to how it is decoded using a table.

For the dynamic decoding processes, the code-length deciphering portion is currently the logical critical path. If the deciphering (which takes place once during cycles 1 and 2 in Figs. 4.7a and 4.7b, and a second time in cycles 4 and 5 in Fig. 4.7b) was able to be performed within the same clock cycle, the length and distance decoding lookups could be performed one cycle sooner, resulting in dynamic length-distance pairs being output two cycles faster. It may also be possible to pipeline the operation of the Huffman decoder if the architecture can be changed to remove the dependency on the accumulator. Optimizations of this nature may require a hand-optimized Verilog or VHDL structural-level implementation in order to achieve them; however, higher clock frequencies may also be possible this way. The prospective lookup of both length and distance codes simultaneously would be a lot more difficult to perform with dynamic codes since both codes can be from 1 to 15 bits long, meaning there are more than 15×15 (225) possibilities of code length combinations. This doesn't include the 0 to 5 extra bits that may fall in between the length and distance codes as well.

As mentioned above, the LZ77 decoder window width could be further expanded to copy more bytes of a length-distance pair at a time. Hypothetically, this could be done to the point where a length-258 length-distance pair (the longest match length possible) could be copied in a single iteration. In this case, it would require partitioning the circular buffer into 512 partitions of 64 indexes each (as 256 partitions wouldn't be enough). As explained before, though, only very highly compressed files would benefit from this type of architecture. Expanding the LZ77 decoder to be able to copy 16 bytes per cycle is probably the most effective width to expand to since the compressor-related works described in Section 2.3.1 reported that match lengths greater than 16 bytes are rare for most file types.

As shown in Table 4.1, the decompressor core occupies a relatively small area, using only 2.59% of the LUTs and 2.01% of the BRAMs on the FPGA. It is worth noting that this model of FPGA is the smallest of the Xilinx Virtex UltraScale+ product line. The compact core size allows for multiple decompressors to be utilized in parallel as part of a multi-file decompressor system. Hypothetically, a cluster of 32 decompressors would use about 83% of the LUTs and 64% of the BRAMs on the FPGA while being able to collectively provide average input decompression throughputs of 4.18 GB/s for statically encoded files and 2.26 GB/s for dynamically encoded files. This is assuming that enough LUTs remain to be able to interface with the 32 decompressors and that such a system could be routed properly.

A single decompressor core could also be improved by implementing multiple Huffman decoders in parallel as part of a speculatively parallel design, as in our original design (See Fig. 4.1). The benefits of this would be data-dependent and statistical, however, unless exploiting a fixed block size that happened to be present in the compressed data. A design incorporating multiple LZ77 decoders in parallel might be feasible but would also be complicated. The LZ77 decoders would need to be interconnected and have the ability to communicate with each other. When an LZ77 decoder encounters a back-reference to data held by another decoder, it would need to stop processing and request that data from the other decoder. A design that employs two-pass parallelization could also be created. Multiple parallel LZ77 decoders could perform a rapid first-pass on data while skipping over any back-references to unheld data. A final LZ77 decoder could collect the data from them and perform the second pass that resolves all of the remaining back-references. The final decoder would need to be fast enough to keep up with the first-pass decoders; otherwise, such a decoder would end up becoming the bottleneck. Any design incorporating multiple LZ77 decoders would also need to use multiple Huffman decoders in order to split the sequential data stream into multiple blocks.

4.5 Summary

In this chapter, we described two FPGA-based Deflate decompressor designs. As described in Chapter 2, the Deflate format makes acceleration using block-level parallelism difficult to achieve without altering the compressed format. Instead of exploiting block-level parallelism through speculation or using a fixed block size, our final decompressor design exploits micro-scale parallelism at the hardware level in order to accelerate the process. These exploits include pipelining at the system level (i.e., concurrently operating the Huffman and LZ77 decoders) and on the task level (i.e., the three pipelined cores operating with an II of 1). We also utilized BRAMs to perform quick decoding in the Huffman decoder and BRAM partitioning to copy multiple bytes at a time in the LZ77 decoder. By doing this, our decompressor design was able to achieve competitive throughputs with efficient hardware utilizations even while using high-level synthesis. Since most other designs don't specify the test files they used to get their results, it is difficult to make many fair comparisons. Compared to the design from [38], where the authors tested their design on a statically compressed Calgary corpus, our design is about 2.44 times faster. Our design is also comparable to the proprietary design sold by Xilinx [41], which has average output static and dynamic throughputs of 495 MB/s and 375 MB/s, respectively, while our design has average output static and dynamic throughputs of 386.56 MB/s and 246.35 MB/s, respectively.

As discussed in Section 4.4, we haven't yet fully reached the limits of exploitable parallelism in our design. Many different optimizations could be made to the Huffman decoder in order to reduce the latencies while the LZ77 decoder window width still has room to be scaled up. Multiple Huffman decoders could be implemented in a speculative parallelization architecture, but we believe this to generally be a poor trade-off. Architectures like this require extra area and provide statistical speed-ups that can't be relied on. This extra area is better spent implementing multiple independent decompressor cores in order to decompress multiple files at a time. Due to the small area of our decompressor design, a system like this would be easily achievable and would be able to achieve massive multi-file decompression throughputs, as explained in Section 4.4.

Chapter 5

Conclusion

5.1 Summary

This thesis research considered the design and implementation of FPGA-based hardware implementations of the Deflate lossless compression and decompression algorithms, using high-level synthesis. For the design of our FPGA-based compressor, we followed the current state-of-the-art architecture that many other designs have been using, which utilizes a fixed-throughput pipeline that performs static Huffman encoding only in order to rapidly compress data. Although these FPGA-based compressor designs provide lower compression ratios than software compressors, they are able to achieve compression throughputs that are a couple orders of magnitude greater. As shown in Table 3.3 and discussed in Section 2.3.1, FPGA-based compressors can achieve input throughputs ranging from 3 to 4 GB/s with mean compression ratios around 2.00, while software compressors typically compress around 50 MB/s with mean compression ratios around 3.00. Our FPGA-based compressor design was able to achieve arithmetic and geometric mean compression ratios of 2.00 and 1.92, respectively, across the Calgary corpus benchmark. Though these values are slightly lower than those achieved by other previous works, which were able to achieve values up to 2.17 [27], we were able to achieve a clock frequency of 250 MHz for our design, giving us a compression throughput of 4.0 GB/s, which is higher than the previous fastest design capable of compressing 3.20 GB/s [7]. These results were also obtained using high-level synthesis, which shows that Vivado HLS is capable of producing competitively performing designs.

In terms of Deflate decompression, the amount of published FPGA-based designs is much fewer. Despite this, we were able to find some designs with ideas that proved useful to include in our design. We also analyzed the technique of speculative parallelization, which has been performed by some software-based decompressors, and investigated how this may be performed on a hardware platform such as an FPGA. For the design of our FPGA-based decompressor, we set out to exploit the parallel hardware resources in FPGAs in order to achieve the fastest decompression throughputs possible. Doing this, we were able to achieve average input decompression throughputs of 130.58 MB/s and 70.73 MB/s for static and dynamic files from the Calgary corpus, respectively. These values correspond to output throughputs of 386.56 MB/s and 246.35 MB/s for static and dynamic files, respectively. Our design is about 2.44 times faster than the design from [38] and has comparable throughputs to the expertly-optimized proprietary IP decompressor sold by Xilinx [41], which has average output static and dynamic throughputs of 495 MB/s and 375 MB/s, respectively, though these results were achieved using unknown test files. Even though we used Vivado HLS to synthesize our decompressor core, we were still able to achieve a compact design that only utilizes 2.59% of the LUTs and 2.01% of the BRAMs on a low tier Xilinx Virtex UltraScale+ XCVU3P-FFVC1517 FPGA. On this FPGA, we could hypothetically deploy a cluster of 32 decompressor cores (while only using 83% of the LUTs and 64% of the BRAMs) and expect to achieve multi-file input decompression throughputs of 4.18 GB/s for statically compressed files and 2.26 GB/s for dynamically compressed files.

5.2 Evaluation of Vivado HLS

As part of this thesis, we evaluated the use of high-level synthesis, specifically Vivado HLS, as a tool for speeding up digital hardware design. From the results of our compressor and decompressor designs, which are summarized in the previous section, we have shown that Vivado HLS is able to create competitive designs in terms of both performance and area compared to designs specified in structural-level VHDL or Verilog. Achieving these competitive results, however, did not come easily due to some of the limitations that Vivado HLS has. In Section 3.2.1, we described the difficulties that we encountered when trying to meet post-route timing with our LZ77 encoder design. As Vivado HLS is only aware of (and therefore only has control over) the logic synthesis part of the FPGA design flow, sometimes extra effort is required in order to help a large and complex design (with significant routing delays) to meet timing following implementation. In Section 3.2.2, we explained a useful advantage of Vivado HLS with its ability to automatically optimize certain operations into simpler ones, like multiplication and division by powers of two automatically being converted into left-shift and right-shift operations. This can lead to faster and more area-efficient designs being synthesized without the user even being aware of these optimizations occurring. This was also shown in Section 3.2.3, where Vivado HLS was shown to be making design optimizations at the system pipeline level, resulting in our 16-stage Huffman encoder design being compacted down automatically into a more efficient 6-stage pipeline.

In Section 4.1, we described what we call the "coarse function scheduling" limitation of Vivado HLS. Due to how the tool operates, sub-functions within a synthesized core must be scheduled to start and finish on strict timing boundaries, preventing the asynchronous execution of multiple functions at a time. This limitation can be overcome to an extent using the Vivado HLS dataflow pragma, which allows each sub-function to operate itself independently of the others; however, this pragma itself imposes many constraints on the code to be synthesized. In Section 4.2.3, we described the difficulties that we encountered when using the partition pragma in order to automatically partition our LZ77 decoder circular buffer due to the fact that the tool could not recognize that we were accessing the memories safely without conflict. In the same section we also described the difficulties in achieving a pipeline II of 1 in our design. This difficulty was caused by the way our code was written, requiring us to rewrite our code in many functionally equivalent ways in order to find a syntax that the tool preferred. In Section 4.4, we gave another example of the same problem where the code describing our Huffman decoder has resulted in inefficient synthesis and the delay of certain operations being performed, even when it is clear that the delay is unnecessary.

Vivado HLS is a tool that comes with many useful advantages but also has many limitations. Although it enables RTL designs to be described in a high-level software programming language, it still requires a hardware engineering design mindset. The ability for Vivado HLS to automatically optimize certain parts of a design, without explanation, is both a blessing and a curse. It requires a very low-level understanding of what is expected to be synthesized and how hardware synthesis is performed in order to be able to understand some of the decisions that Vivado HLS makes in its operation. In general, the simpler a design, the easier it will be to synthesize efficiently using the tool. When attempting to synthesize large and complicated designs that realistically should be synthesizeable, it can be frustrating working around the quirks of Vivado HLS in order to get it to create what you want. In summary, Vivado HLS is a very powerful tool to aid in the design of digital hardware that comes with the cost of having a steep leaning curve in order to be able to use effectively.

5.3 Future Work

As discussed in Section 3.4, our compressor design still has room for improvement in terms of achieving greater compression ratios. First, the hash function still may have room to be improved in order to reduce the likelihood of bank access conflicts. Second, the hash bank architecture could be scaled up in multiple ways at the cost of area and throughput in order to increase the compression ratio. Other potential ideas for improving the compression ratio include using multiple hash functions for different data types or making the major design changes necessary to be able to perform dynamic Huffman encoding as well as static encoding. There is still room for improvement in the throughput of a single-core compressor as well, before resorting to scaling up the number of compressors working in parallel. Since we were able to achieve a clock frequency of 250 MHz using high-level synthesis, it is reasonable to expect that hand-coded RTL designs should be able to surpass this number.

As discussed in 4.4, our decompressor design still has not quite exploited all available parallelism. The Huffman decoder still has plenty of inefficiencies in its implementation that could be improved and the LZ77 decoder could be easily scaled up further in order to resolve large length-distance pairs faster. Once these limits of parallelism are reached, speculative parallelization is one way of potentially achieving even greater speed-ups at the cost of additional hardware area. Due to the inherently-serial Deflate compressed format, however, improving the decompression speeds much further than currently achievable numbers will be difficult and will likely never reach the values of more easily parallelizable compressed formats. This situation is an example of where a well-entrenched and widely-used standard is effectively limiting achievable improved performance.

References

[1] R. Koenen. (2002). Overview of the MPEG-4 standard, [Online]. Available: `https://web.archive.org/web/20100428021525/http://mpeg.chiariglione.org/standards/mpeg-4/mpeg-4.htm`.

[2] L. P. Deutsch. (1996). ‘‘RFC 1951: DEFLATE compressed data format specification version 1.3”, [Online]. Available: `https://www.w3.org/Graphics/PNG/RFC-1951`.

[3] D. Salomon, *Data Compression: The Complete Reference*, 4th ed. Springer Science & Business Media, 2006.

[4] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, “Massively-parallel lossless data decompression,” in *the 45th Int. Conf. Parallel Processing (ICPP)*, 2016, pp. 242–247.

[5] J. Fowers, J. Y. Kim, D. Burger, and S. Hauck, “A scalable high-bandwidth architecture for lossless compression on FPGAs,” in *IEEE 23rd Annu. Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, 2015, pp. 52–59.

[6] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu, “CPU-FPGA Coscheduling for big data applications,” English, *IEEE Design & Test*, vol. 35, no. 1, pp. 16–22, 2018.

[7] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C. F. Chang, and J. Cong, “High-throughput lossless compression on tightly coupled CPU-FPGA platforms,” in *IEEE 26th Annu. Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 37–44.

[8] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility,” *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[9] Amazon Web Services, Inc. (2019). *Amazon EC2 F1 Instances*, [Online]. Available: `https://aws.amazon.com/ec2/instance-types/f1/`.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture : A Quantitative Approach*, 5th ed. Burlington: Elsevier Science & Technology, May 2014, ISBN: 9780123704900.

[11] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[12] "IEEE standard VHDL language reference manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[13] "IEEE standard for Verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[14] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.

[15] Xilinx Inc. (2019). *Vivado High-Level Synthesis*, [Online]. Available: `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[16] PKWARE Inc. (2019). *PKZIP*, [Online]. Available: `https://www.pkware.com/pkzip`.

[17] J.-l. Gailly and M. Adler. (2018). *gzip*, [Online]. Available: `https://www.gzip.org/`.

[18] ——, (2017). *zlib*, [Online]. Available: `https://zlib.net/zlib.html`.

[19] M. Adler, *Answer to: How are zlib, gzip and zip related? what do they have in common and how are they different?* Dec. 2013. [Online]. Available: `https://stackoverflow.com/questions/20762094/how-are-zlib-gzip-and-zip-related-what-do-they-have-in-common-and-how-are-they/20765054%5C#20765054`.

[20] PKWARE Inc. (2019). ".ZIP file format specification", [Online]. Available: `https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT`.

[21] L. P. Deutsch. (1996). "RFC 1952: GZIP file format specification version 4.3", [Online]. Available: `https://tools.ietf.org/html/rfc1952`.

[22] T. Boutell. (1997). "RFC 2083: PNG (portable network graphics) specification version 1.0", [Online]. Available: `https://tools.ietf.org/html/rfc2083`.

[23] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Info. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[24] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[25] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Communications of the ACM*, vol. 7, no. 3, pp. 166–169, 1964.

[26] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 557–591, 1989.

[27] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL," in *Proc. Int. Workshop on OpenCL 2013 & 2014*, ser. IWOCL '14, Bristol, United Kingdom: ACM, 2014, pp. 4–9.

[28] A. Martin, D. Jamsek, and K. Agarawal, "FPGA-based application acceleration: Case study with gzip compression/decompression streaming engine," *ICCAD Special Session C*, vol. 7, 2013.

[29] Y. Kim, S. Choi, J. Jeong, and Y. H. Song, "Data dependency reduction for high-performance FPGA implementation of DEFLATE compression algorithm," *Journal of Systems Architecture*, vol. 98, pp. 41–52, 2019. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S1383762118306453`.

[30] V. Gopal, J. Guilford, W. Feghali, E. Ozturk, and G. Wolrich, *High performance DEFLATE compression on Intel® architecture processors*, 2011. [Online]. Available: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-deflate-compression-paper.pdf`.

[31] D. Harnik, E. Khaitzin, D. Sotnikov, and S. Taharlev, "A fast implementation of Deflate," in *2014 Data Compression Conference*, 2014, pp. 223–232.

[32] M. Chłopkowski and R. Walkowiak, "A general purpose lossless data compression method for GPU," *Journal of Parallel and Distributed Computing*, vol. 75, pp. 40–52, 2015.

[33] H. Jang, C. Kim, and J. W. Lee, "Practical speculative parallelization of variable-length decompression algorithms," *SIGPLAN Not.*, vol. 48, no. 5, pp. 55–64, Jun. 2013.

[34] Z. Wang, Y. Zhao, Y. Liu, Z. Chen, C. Lv, and Y. Li, "A speculative parallel decompression algorithm on Apache Spark," *Journal of Supercomputing*, vol. 73, no. 9, pp. 4082–4111, 2017.

[35] The Apache Software Foundation. (2018). *Apache Spark*, [Online]. Available: `https://spark.apache.org/`.

[36] M. Kerbiriou and R. Chikhi, "Parallel decompression of gzip-compressed files and random access to DNA sequences," May 2019. [Online]. Available: `https://arxiv.org/abs/1905.07224`.

[37] H. Li. (2008). "FASTQ format specification", [Online]. Available: `http://maq.sourceforge.net/fastq.shtml`.

[38] J. Lazaro, J. Arias, A. Astarloa, U. Bidarte, and A. Zuloaga, "Decompression dual core for SoPC applications in high speed FPGA," in *IECON 2007 - 33rd Annu. Conf. IEEE Industrial Electronics Society*, 2007, pp. 738–743.

[39] D. C. Zaretsky, G. Mittal, and P. Banerjee, "Streaming implementation of the ZLIB decoder algorithm on an FPGA," in *IEEE Int. Symp. Circuits and Systems*, 2009, pp. 2329–2332.

[40] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "FPGA implementation of GZIP compression and decompression for IDC services," in *IEEE Int. Conf. Field-Programmable Technology (FPT)*, IEEE, 2010, pp. 265–268.

[41] Xilinx and CAST Inc. (2016). GUNZIP/ZLIB/Inflate Data Decompression Core, [Online]. Available: `https://www.xilinx.com/products/intellectual-property/1-79drsh.html#overview`.

[42] ARM, *AMBA 4 AXI4-Stream Protocol*, 2010.

[43] Xilinx Inc., *UG902 - Vivado design suite user guide: High-level synthesis (v2019.1)*, Jul. 2019. [Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf`.

Appendix A

LZ77 Encoder Source File

LZ77_Encoder.cpp

```
#include "LZ77_Encoder.h"
#include <utils/x_hls_utils.h>

t_hash_size hash_func(
    ap_uint<8> in_byte0,
    ap_uint<8> in_byte1,
    ap_uint<8> in_byte2,
    ap_uint<8> in_byte3,
    ap_uint<8> in_byte4
){
  t_hash_size hash;

  hash = ((t_hash_size)in_byte0 * 31) ^ in_byte1;
  hash.rrotate(4);
  hash = (hash * 3) ^ in_byte2;
  hash.rrotate(4);
  hash = (hash * 3) ^ in_byte3;
  hash.rrotate(4);
  hash = (hash * 3) ^ in_byte4;

  return hash;
}

void bank_access_func(
    t_hash_size          hash[WIN_SIZE],               //Input
    ap_uint<WIN_SIZE*8>  substring_ints[WIN_SIZE],     //Input
    ap_uint<32>       data_pos,           //Input
    ap_uint<NUM_BANKS>  &bank_accessed_bits,      //Output
    ap_uint<WIN_SIZE>   &substring_access_bits,    //Output
    t_bank_values        associated_bank[WIN_SIZE],   //Output
    match_t           string_to_write[NUM_BANKS], //Output
    t_bank_size       string_address[NUM_BANKS]  //Output
){
#pragma HLS INLINE off
  t_num_banks desired_bank[WIN_SIZE];
#pragma HLS ARRAY_PARTITION variable=desired_bank complete dim=1
  t_bank_size bank_address[WIN_SIZE];
```

```
#pragma HLS ARRAY_PARTITION variable=bank_address complete dim=1
  t_win_size associated_substring[NUM_BANKS] = {0}; //Contains the substring number given to each bank
#pragma HLS ARRAY_PARTITION variable=associated_substring complete dim=1

  Bank_Access: for(int i = WIN_SIZE-1; i >= 0; i--){ //For each substring (starting from the bottom)
    desired_bank[i] = hash[i](HASH_BITS-1,HASH_BITS-NUM_BANKS_BITS); //The top 5 bits of the hash value indicate the desired bank to be accessed
    bank_address[i] = hash[i](BANK_SIZE_BITS-1,0);  //The bottom 9 bits are the bank address to access
    if(bank_accessed_bits[desired_bank[i]] == 0){ //If desired bank has not already been accessed
      bank_accessed_bits[desired_bank[i]] = 1; //Set bank as accessed
      substring_access_bits[i] = 1; //Record substring as granted access
      associated_bank[i] = desired_bank[i]; //Record bank accessed by substring
      associated_substring[desired_bank[i]] = i; //Record substring given to bank
    }
    else{
      associated_bank[i] = NUM_BANKS; //Associate with null bank
    }
  }

  //Pass 1 of 16 substrings to each of 32 banks (Not all banks will actually be accessed)
  String_to_Bank_MUX: for(int i = 0; i < NUM_BANKS; i++){//For each bank
    string_to_write[i].string = reg(substring_ints[associated_substring[i]]);
    string_to_write[i].position = data_pos + associated_substring[i];
    string_address[i] = bank_address[associated_substring[i]];
  }
}

void LZ77_Encoder(
    stream<in_strm>  &strm_in,
    stream<out_strm> &strm_out,
    int         input_size
){
#pragma HLS INTERFACE axis register both port=strm_in  //Input AXI Stream
#pragma HLS INTERFACE axis register both port=strm_out //Output AXI Stream

  in_strm input_buff;
  ap_uint<8> curr_window[2*WIN_SIZE]; //Current window of data to be compressed.
```

```
#pragma HLS ARRAY_PARTITION variable=curr_window complete dim=1
  t_hash_size hash[WIN_SIZE]; //Hash values of each substring in
    curr_window.
#pragma HLS ARRAY_PARTITION variable=hash complete dim=1
  static match_t dictionary[NUM_BANKS][BANK_DEPTH][BANK_SIZE]; //
    Dictionary partitioned into banks with each bank having
    multiple depth levels
#pragma HLS RESOURCE variable=dictionary core=RAM_S2P_BRAM
#pragma HLS ARRAY_PARTITION variable=dictionary complete dim=1
#pragma HLS ARRAY_PARTITION variable=dictionary complete dim=2
  ap_uint<32> data_pos = 0; //Marker to keep track of input data
    position
  ap_uint<WIN_SIZE> match_length_bits[NUM_BANKS][BANK_DEPTH]; //
    Bit-string of matching bytes between substring and potential
    matches
#pragma HLS ARRAY_PARTITION variable=match_length_bits complete
    dim=0
  t_match_size match_length[NUM_BANKS][BANK_DEPTH]; //Integer
    value of match length between substring and potential matches
#pragma HLS ARRAY_PARTITION variable=match_length complete dim=0
  t_match_size bank_best_length[NUM_BANKS]; //The value of the
    best match length from each bank
#pragma HLS ARRAY_PARTITION variable=bank_best_length complete dim
    =1
  ap_int<32> bank_best_position[NUM_BANKS]; //The position of the
    best match from each bank
#pragma HLS ARRAY_PARTITION variable=bank_best_position complete
    dim=1
  t_bank_depth bank_best_match[NUM_BANKS]; //The depth number
    containing the best match from each bank
#pragma HLS ARRAY_PARTITION variable=bank_best_match complete dim
    =1
  t_match_size window_best_length[WIN_SIZE]; //The value of the
    best match length for each substring
#pragma HLS ARRAY_PARTITION variable=window_best_length complete
    dim=1
  ap_int<32> window_best_position[WIN_SIZE]; //The position of the
     best match for each substring
#pragma HLS ARRAY_PARTITION variable=window_best_position complete
     dim=1
  t_win_size last_match; //Position of last match in current
    window. Used to calculate FVP
  t_win_size first_valid_pos = 0; //Marker to keep track of first
    valid match position in current window
  t_match_size reach[WIN_SIZE]; //How far a match extends within
    the current window
#pragma HLS ARRAY_PARTITION variable=reach complete dim=1
  ap_uint<32> offsets[WIN_SIZE]; //Offset of each best match found
#pragma HLS ARRAY_PARTITION variable=offsets complete dim=1
  ap_uint<WIN_SIZE> valid_matches_bits; //Bit-string for
    identifying valid best matches after filtering
  ap_uint<WIN_SIZE> matched_literals_bits; //Bit-string for
    locating matched literals in current window
  out_array output_array[WIN_SIZE];
```

```
#pragma HLS ARRAY_PARTITION variable=output_array complete dim=1
  out_strm output_buff = {0};
  int num_iterations;
  t_match_size bytes_conflicting; //Number of bytes conflicting between two potential matches
  ap_uint<NUM_BANKS> bank_accessed_bits; //Bit-string for recording which dictionary banks have been accessed for writing
  ap_uint<WIN_SIZE> substring_access_bits; //Bit-string for recording which substrings have been granted access to a bank
  t_bank_values associated_bank[WIN_SIZE]; //Contains the bank number accessed by each window substring
#pragma HLS ARRAY_PARTITION variable=associated_bank complete dim=1
  match_t string_to_write[NUM_BANKS]; //String + position to be written to each bank
#pragma HLS ARRAY_PARTITION variable=string_to_write complete dim=1
  t_bank_size string_address[NUM_BANKS]; //Address to write string and position to
#pragma HLS ARRAY_PARTITION variable=string_address complete dim=1
  match_t potential_match[NUM_BANKS][BANK_DEPTH]; //Potential matches read from each bank and depth level
#pragma HLS ARRAY_PARTITION variable=potential_match complete dim=0
  ap_uint<WIN_SIZE*8> substring_ints[WIN_SIZE]; //Integers of each substring in current window
#pragma HLS ARRAY_PARTITION variable=substring_ints complete dim=1

  //Dictionary Initialization
  Dict_Init_Loop_A: for(int i = 0; i < BANK_SIZE; i++){ //For each bank entry
    Dict_Init_Loop_B: for(int j = 0; j < NUM_BANKS; j++){ //For each bank
#pragma HLS UNROLL
      Dict_Init_Loop_C: for(int k = 0; k < BANK_DEPTH; k++){ //For each depth level
#pragma HLS UNROLL
        dictionary[j][k][i].string = 0;
        dictionary[j][k][i].position = -32769;
      }
    }
  }

  num_iterations = (input_size-1)/WIN_SIZE + 1; //Calculate number of iterations (Rounded up)
  //In the first iteration, current window needs to be loaded before shifting order to fill it completely
  strm_in >> input_buff;
  First_Load: for(int i = 0; i < WIN_SIZE; i++){ //Load window with data from input buffer
#pragma HLS UNROLL
    curr_window[i + WIN_SIZE] = input_buff.data((i*8)+7,i*8); //Convert from int to array (Little Endian)
  }

  if(!input_buff.last){ //If first read wasn't last
    //Main Function Loop
    Main_Loop: for(int n = 0; n < (num_iterations-1); n++){ //
   Perform all iterations except last in loop
#pragma HLS PIPELINE
#pragma HLS DEPENDENCE variable=dictionary inter false

      Window_Shift: for(int i = 0; i < WIN_SIZE; i++){ //Shift
   second half of window into front half of window
        curr_window[i] = curr_window[i + WIN_SIZE];
      }

      strm_in >> input_buff;
      Load_Window: for(int i = 0; i < WIN_SIZE; i++){ //Load
   window with data from input buffer
        curr_window[i + WIN_SIZE] = input_buff.data((i*8)+7,i*8);
   //Convert from int to array (Little Endian)
      }

      Hash: for(int i = 0; i < WIN_SIZE; i++){  //Hash each
   substring of window
        hash[i] = hash_func(curr_window[i],curr_window[i+1],
   curr_window[i+2],curr_window[i+3],
            curr_window[i+4]); //Hash first four bytes of each
   substring
      }

      Pull_Substrings_A: for(int i = 0; i < WIN_SIZE; i++){ //For
   each substring
        Pull_Substrings_B: for(int j = 0; j < WIN_SIZE; j++){ //
   For each character in substring
          substring_ints[i]((j*8)+7,j*8) = curr_window[i+j]; //
   Pull substrings from current window
        }
      }

      bank_accessed_bits = 0; //Reset bank access flag bits
      substring_access_bits = 0; //Reset substring access flag
   bits
      //Allocate bank access to substrings
      bank_access_func(hash, substring_ints, data_pos,
   bank_accessed_bits, substring_access_bits, associated_bank,
   string_to_write, string_address);

      Read_Matches_A: for(int i = 0; i < NUM_BANKS; i++){//For
   each bank
        Read_Matches_B: for(int j = 0; j < BANK_DEPTH; j++){//For
   each depth level
          if(bank_accessed_bits[i] == 1){ //If accessed by
   substring
            potential_match[i][j].string = dictionary[i][j][
   string_address[i]].string; //Read potential match string
```

```
        potential_match[i][j].position = dictionary[i][j][string_address[i]].position; //Read potential match position
      }
    }
  }

  Write_Substrings_A: for(int i = 0; i < NUM_BANKS; i++){//For each bank
    if(bank_accessed_bits[i] == 1){ //If accessed by substring
      dictionary[i][0][string_address[i]].string = string_to_write[i].string; //Write substring to depth 0
      dictionary[i][0][string_address[i]].position = string_to_write[i].position; //Write position to depth 0
      Write_Substrings_B: for(int j = 1; j < BANK_DEPTH; j++){//For each depth level below depth 0
        dictionary[i][j][string_address[i]].string = potential_match[i][j-1].string; //Write substring from depth above
        dictionary[i][j][string_address[i]].position = potential_match[i][j-1].position; //Write position from depth above
      }
    }
  }

  Match_Compare_A: for(int i = 0; i < NUM_BANKS; i++){//For each bank
    Match_Compare_B: for(int j = 0; j < BANK_DEPTH; j++){//For each depth level
      Match_Compare_C: for(int k = 0; k < WIN_SIZE; k++){//For each character
        match_length_bits[i][j][k] = (string_to_write[i].string((k*8)+7,k*8) == potential_match[i][j].string((k*8)+7,k*8)) ? 1 : 0; //Compare strings
      }
    }

  }

  Match_Length_A: for(int i = 0; i < NUM_BANKS; i++){ //For each bank
    Match_Length_B: for(int j = 0; j < BANK_DEPTH; j++){ //For each depth level
      match_length[i][j] = __builtin_ctz((0b1,~match_length_bits[i][j])); //Calculate integer length
    }
  }

  Reset_Best: for(int i = 0; i < NUM_BANKS; i++){ //For each bank
    bank_best_length[i] = 0; //Reset best match lengths from last iteration
  }

```

```
    Best_Match_A: for(int i = 0; i < NUM_BANKS; i++){ //For each bank
      Best_Match_B: for(int j = 0; j < BANK_DEPTH; j++){ //For each depth level
        if(match_length[i][j] > bank_best_length[i]){ //Compare matches to find best length
          bank_best_length[i] = match_length[i][j];
          bank_best_match[i] = j;
        }
      }
    }

    Best_Match_MUX: for(int i = 0; i < NUM_BANKS; i++){ //For each bank
      bank_best_position[i] = reg(potential_match[i][bank_best_match[i]].position); //Pass position of best match
    }

    Bank_to_String_MUX: for(int i = 0; i < WIN_SIZE; i++){//For each window substring
      if(substring_access_bits[i] == 1){ //If substring was granted bank access
        window_best_length[i] = bank_best_length[associated_bank[i]]; //Obtain best match length and position from associated bank
        window_best_position[i] = bank_best_position[associated_bank[i]];
      }
      else{ //Otherwise, set match length to 0
        window_best_length[i] = 0;
        window_best_position[i] = 0;
      }
    }

    Calc_Valid: for(int i = 0; i < WIN_SIZE; i++){ //For each best match
      valid_matches_bits[i] = (window_best_length[i] == 0) ? 0 : 1; //If the match length is not 0, the match is valid
    }

    Calc_Offset: for(int i = 0; i < WIN_SIZE; i++){ //For each substring
      offsets[i] = data_pos + i - window_best_position[i];
      if(offsets[i] > MAX_DISTANCE){ //Filter matches that are too far back
        valid_matches_bits[i] = 0;
      }
    }

    Filter_A: for(int i = 0; i < WIN_SIZE; i++){ //For each best match
      if(window_best_length[i] < MIN_LENGTH) //Filter matches with length less than 3
        valid_matches_bits[i] = 0;
    }

    //Filter B
    valid_matches_bits &= ((ap_uint<WIN_SIZE>)0xFFFF <<
first_valid_pos); //Filter matches covered by previous
iteration by clearing bits up to FVP

    last_match = (valid_matches_bits != 0) ? (31 - __builtin_clz
(valid_matches_bits)) : 0; //Calculate last match. If no valid
 matches, set to 0

    //Set matched_literals based on first valid position before
updating it
    matched_literals_bits = decoder[first_valid_pos](0,WIN_SIZE
-1);

    Calc_Reach: for(int i = 0; i < WIN_SIZE; i++){ //For each
best match
      reach[i] = i + window_best_length[i]; //Calculate reach of
 all matches
    }

    //Calculate first valid position of next iteration using the
 reach of the last match
    //If reach of last match is greater than 16, calculate FVP.
Otherwise FVP is 0.
    first_valid_pos = (reach[last_match][MATCH_SIZE_BITS-1] ==
1) ? reach[last_match](MATCH_SIZE_BITS-2,0) : 0;

    //Filter matches with same reach but lower length than
others
    Filter_C1: for(int i = 0; i < WIN_SIZE; i++){ //For each
match
      if(valid_matches_bits[i] != 0){ //If match is still valid
        Filter_C2: for(int j = i+1; j < WIN_SIZE; j++){ //
Compare to all valid matches 'below' this match
          if(reach[i] == reach[j]){ //If matches have same reach
, match below has lower length
            if(j == last_match){ //If last match is being
filtered for a longer match
              valid_matches_bits(WIN_SIZE-1,i+1) = 0; //Filter
all matches below match i
            }
            else{
              valid_matches_bits[j] = 0; //Filter match below
            }
          }
        }
      }
    }

    //Filter or trim matches that conflict with later matches (
last match must be kept)
```

```
  Filter_D1: for(int i = WIN_SIZE-1; i >= 0; i--){ //For each match (starting from the bottom)
    if(valid_matches_bits[i] != 0){ //If match is still valid
      Filter_D2: for(int j = 0; j < i; j++){ //Compare to all valid matches 'above' this match
        if(reach[j] > i){ //If above matches conflict with this match
          bytes_conflicting = reach[j] - i; //Calculate number of conflicting bytes
          if(window_best_length[j] - bytes_conflicting >= MIN_LENGTH){ //If match j can safely be trimmed (without decreasing length below 3)
            window_best_length[j] -= bytes_conflicting; //Trim match j
          }
          else{
            valid_matches_bits[j] = 0; //Filter match j
          }
        }
      }
    }
  }

  //Preparing output sequence
  Prepare_OutputA: for(int i = 0; i < WIN_SIZE; i++){ //For each box in window
    output_array[i].user = (valid_matches_bits[i] != 0) ? 0b10 : 0b01;
    //If box contains a valid match, set user flag to match. Otherwise set to unmatched literal
  }

  Prepare_OutputB: for(int i = 0; i < WIN_SIZE; i++){ //For each box in window
    if(valid_matches_bits[i] != 0){ //If match is valid
      matched_literals_bits |= (((ap_uint<WIN_SIZE>)decoder[window_best_length[i]] >> 1) << (WIN_SIZE - window_best_length[i] - i));
    } //Turn match length into bit-string and OR all together to find matched literals
  }

  Prepare_OutputC: for(int i = 0; i < WIN_SIZE; i++){ //For each box in window (each bit in window_matches)
    if(matched_literals_bits[WIN_SIZE-1-i] == 1){ //If the box contains a matched literal
      output_array[i].user = 0b00;
    } //Set user flag to 0 for all matched literals
  }

  Prepare_OutputD: for(int i = 0; i < WIN_SIZE; i++){
    switch(output_array[i].user){ //Based on status of output box, fill data accordingly
    case 0b00 : //Matched Literal
```

```
          output_array[i].data = 0; //Clear box (Only useful for debugging)
          break;

        case 0b01 : //Unmatched Literal
          output_array[i].data = curr_window[i]; //Write literal
          break;

        case 0b10 : //Match
          assert(window_best_length[i] >= MIN_LENGTH); // Assertions for C debugging
          assert(window_best_length[i] <= WIN_SIZE);
          assert(offsets[i] >= 1);
          assert(offsets[i] <= MAX_DISTANCE);

          output_array[i].data(23, 15) = window_best_length[i]; // Write Length in upper 9 bits
          output_array[i].data(14,  0) = offsets[i] - 1; //Write Distance in lower 15 bits
          break;
        }
      }

      Prepare_OutputE: for(int i = 0; i < WIN_SIZE; i++){ // Convert from array to int (Little Endian)
        output_buff.data((i*24)+23,i*24) = output_array[i].data;
        output_buff.user( (i*2)+ 1, i*2) = output_array[i].user;
      }

      //Write to output stream
      strm_out << output_buff;

      data_pos += WIN_SIZE; //Increment data marker
    } //For-num_iterations-loop
  }

  Last_Shift: for(int i = 0; i < WIN_SIZE; i++){ //Shift second half of window into front half of window
#pragma HLS UNROLL
    curr_window[i] = curr_window[i + WIN_SIZE];
  }

  //Calculate matched_literals one last time
  matched_literals_bits = decoder[first_valid_pos](0,WIN_SIZE-1);
  matched_literals_bits |= ~(((ap_uint<WIN_SIZE>)input_buff.keep(0,WIN_SIZE-1))); //OR with TKEEP to identify null bytes at end of input stream

  Last_OutputA: for(int i = 0; i < WIN_SIZE; i++){ //For each box in window (each bit in window_matches)
#pragma HLS UNROLL
    output_array[i].user = (matched_literals_bits[WIN_SIZE-1-i] == 1) ? 0b00 : 0b01;
  } //If the box contains a matched literal or null byte, set to 0
    b00

  Last_OutputB: for(int i = 0; i < WIN_SIZE; i++){
#pragma HLS UNROLL
    switch(output_array[i].user){ //Based on status of output box,
     fill data accordingly
    case 0b00 : //Matched Literal/Null Character
      output_array[i].data = 0; //Clear box (Only useful for
    debugging)
      break;

    case 0b01 : //Unmatched Literal
      output_array[i].data = curr_window[i]; //Write literal
      break;
    }
  }

  Last_OutputC: for(int i = 0; i < WIN_SIZE; i++){ //Convert from
    array to int (Little Endian)
#pragma HLS UNROLL
    output_buff.data((i*24)+23,i*24) = output_array[i].data;
    output_buff.user( (i*2)+ 1, i*2) = output_array[i].user;
  }
  output_buff.last = 1;
  strm_out << output_buff;
}
```

Appendix B

LZ77 Encoder Header File

LZ77_Encoder.h

```
#include <stdio.h>
#include <string.h>
#include <ap_int.h>
#include <hls_stream.h>
#include <assert.h>

using namespace hls;

#define WIN_SIZE      16                    //Compare window size.
    Sliding window is twice this
#define NUM_BANKS   32                  //Number of dictionary banks
#define BANK_DEPTH    3                     //Depth of each dictionary
     bank
#define BANK_INDEXES (512*NUM_BANKS)//16384    //Total number of
    hash bank indexes
#define BANK_SIZE   (BANK_INDEXES/NUM_BANKS) //Number of indexes in
     each bank
#define MIN_LENGTH    3                     //Minimum Deflate match
    length
#define MAX_DISTANCE 32768                //Maximum Deflate match
    distance

//Dynamic bit width definitions (Only work in C sim)
#ifndef __SYNTHESIS__
#define WIN_SIZE_BITS    (int)(log2(WIN_SIZE-1)+1)      //Number of
    bits required to store WIN_SIZE values
#define BANK_DEPTH_BITS (int)(log2(BANK_DEPTH-1)+1)    //Number of
    bits required to store BANK_DEPTH values
#define MATCH_SIZE_BITS (int)(log2(WIN_SIZE)+1)        //Number of
    bits required to store maximum match length (WIN_SIZE)
#define HASH_BITS        (int)(log2(BANK_INDEXES-1)+1) //Number of
    hash bits required to index BANK_INDEXES (Complete hash
    function)
#define NUM_BANKS_BITS   (int)(log2(NUM_BANKS-1)+1)     //Number of
    bits required to index NUM_BANKS values (Top bits of hash
    function)
```

```
#define BANK_SIZE_BITS  (int)(log2(BANK_SIZE-1)+1)    //Number of bits required to index BANK_SIZE values (Bottom bits of hash function)
#else
#define WIN_SIZE_BITS   4
#define BANK_DEPTH_BITS 2
#define MATCH_SIZE_BITS 5
#define HASH_BITS       14
#define NUM_BANKS_BITS  5
#define BANK_SIZE_BITS  9
#endif

typedef ap_uint<WIN_SIZE_BITS> t_win_size;
typedef ap_uint<BANK_DEPTH_BITS> t_bank_depth;
typedef ap_uint<MATCH_SIZE_BITS> t_match_size;
typedef ap_uint<HASH_BITS> t_hash_size;
typedef ap_uint<NUM_BANKS_BITS> t_num_banks;
typedef ap_uint<NUM_BANKS_BITS+1> t_bank_values; //1 extra bit to include the value NUM_BANKS
typedef ap_uint<BANK_SIZE_BITS> t_bank_size;

typedef struct{ //Input AXI-Stream structure
  ap_uint<WIN_SIZE*8> data; //Array of window characters (128 bits with 16 bytes)
  ap_uint<WIN_SIZE>   keep; //TKEEP Signals for each byte
  bool last;
} in_strm;

typedef struct{ //Output Array structure
  ap_uint<24> data; //3-byte box for one of the window characters from LZ77 encoder
  ap_uint<2> user;  //2-bit flag for identifying a box as a literal, length, distance, or matched literal
} out_array;

typedef struct{ //Output AXI-Stream structure
  ap_uint<3*WIN_SIZE*8> data; //3-byte 'boxes' for each of the window characters from LZ77 encoder
  ap_uint<2*WIN_SIZE> user; //2-bit flags for identifying each box as a literal, length, distance, or matched literal
  bool last;           //AXI-Stream TLAST signal
} out_strm;

typedef struct{ //Match Structure
  ap_uint<WIN_SIZE*8> string;   //16-byte string, int form
  ap_int<32>       position; //Position of string in history
} match_t;

//16-bit decoder. Given an integer index n, returns a bit string containing n ones.
const ap_uint<16> decoder[17] = {
    /* 0*/ 0b0000000000000000,
    /* 1*/ 0b0000000000000001,
    /* 2*/ 0b0000000000000011,
    /* 3*/ 0b0000000000000111,
    /* 4*/ 0b0000000000001111,
    /* 5*/ 0b0000000000011111,
    /* 6*/ 0b0000000000111111,
    /* 7*/ 0b0000000001111111,
    /* 8*/ 0b0000000011111111,
    /* 9*/ 0b0000000111111111,
    /*10*/ 0b0000001111111111,
    /*11*/ 0b0000011111111111,
    /*12*/ 0b0000111111111111,
    /*13*/ 0b0001111111111111,
    /*14*/ 0b0011111111111111,
    /*15*/ 0b0111111111111111,
    /*16*/ 0b1111111111111111
};
```

Appendix C

Huffman Encoder Source File

Huffman_Encoder.cpp

```
#include "Huffman_Encoder.h"
#include "code_tables.cpp"

void symbol_encoder(
    ap_uint<24>  data,        //Input (3 bytes)
    ap_uint<2>   type,        //Input
    ap_uint<26>  &code,       //Output (Can be from 0 to 26 bits
   long)
    ap_uint<5>   &code_length //Output
){
#pragma HLS INLINE
  ap_uint<9>     length;
  ap_uint<15>    distance;
  ap_uint<8>     length_code;
  distance_code  distance_symbol;
  ap_uint<13>    extra_distance_value;

  switch(type){
  case 0b00 : //Matched literal
    code = 0;
    code_length = 0;
    break;

  case 0b01 : //Unmatched Literal
    assert(data >= 0);
    assert(data <= 255);
    if(data <= 143){
      code_length = 8;
      data = data + 48; //Convert literal value to code
      code = data(0,7); //Mirror code and output it
    }
    else{
      code_length = 9;
      data = data + 256; //Convert literal value to code
      code = data(0,8);  //Mirror code and output it
    }
    break;

  case 0b10 : //Match
    length = data(23,15);
    distance = data(14,0);
    assert(length >= 3); //Assertions for C debugging
    assert(length <= 16);
    assert(distance >= 0); //Distance is subtracted by 1 to fit
   within 15 bits (Done by LZ77 Encoder)
    assert(distance <= 32767);

    //Length Encoding
    if(length <= 10){ //If length is 10 or less, no extra bits in
   code length
      code_length = 7;
    }
    else{
      code_length = 8; //7-bit length code + 1 extra bit
    }
    length -= 3; //Adjust length before looking up in table
    length_code = length_code_table[length]; //Length codes are
   pre-mirrored with extra bits included

    //Distance Encoding
    if(distance < 256){ //Distance is from 1 to 256
      distance_symbol = distance_code_table[distance];
    }
    else{ //Distance is from 257 to 32768
      distance_symbol = distance_code_table[distance(14,7) + 256];
    //Top 8 bits of distance + 256
    }
    extra_distance_value = distance - distance_symbol.base;

    //Concatenate codes as follows: extra distance bits, distance
   code (mirrored), length code (already mirrored)
    code = (extra_distance_value, distance_symbol.code(0,4),
   length_code(code_length-1,0));
    code_length += 5 + distance_symbol.bits; //Add distance bits
   to code length
    break;
  }
}

template <int instance>
void window_packer(
    in_stream  *input_window_in,  //Input: Window of 3-byte boxes
   to be encoded
    in_stream  *input_window_out,   //Output
    t_enc_win  *encoded_window_in,  //Input: Window for packing
   Huffman codes
    ap_uint<8> *encoded_bits_in,  //Input: Number of bits in
   encoded window
    t_enc_win  *encoded_window_out, //Output
    ap_uint<8> *encoded_bits_out  //Output
){
```

```
#pragma HLS INLINE

  ap_uint<26> code;       //Huffman code to be packed
  ap_uint<5>  code_length;  //Length of Huffman code
  t_enc_win   shifted_code;

  //Input window is Little Endian, process lower end first
  ap_uint<24> input_box =  input_window_in->data((instance*24)+23,
    instance*24);
  ap_uint<2>  input_type = input_window_in->user( (instance*2) +1,
     instance*2);

  symbol_encoder(input_box, input_type, code, code_length); //
    Encode box from input window

  *encoded_window_out = *encoded_window_in | (t_enc_win(code) << *
    encoded_bits_in); //Pack code in window
  *encoded_bits_out = *encoded_bits_in + code_length;  //Update
    bit count of window
  assert(*encoded_bits_out <= ENC_WIN_BITS); //C Debug: Check for
    encoded window overfill

  *input_window_out = *input_window_in;      //Pass input window to
     output
}

void output_packer(
    t_enc_win           encoded_window_in, //Input: Window for
    packing Huffman codes
    ap_uint<8>          encoded_bits_in,    //Input: Number of bits
     in encoded window
    ap_uint<OUT_WIN_BITS> &output_window,     //In/Out
    ap_uint<10>         &output_bits,       //In/Out
    stream<out_stream>    &output_strm        //Output
){
#pragma HLS INLINE
  out_stream out_buff = {0};

  output_window |= ap_uint<OUT_WIN_BITS>(encoded_window_in) <<
    output_bits; //Pack encoded window in output packer
  output_bits += encoded_bits_in; //Update bit count of packer
  assert(output_bits <= OUT_WIN_BITS); //C Debug: Check for output
     window overfill

  if(output_bits >= OUT_STRM_BITS){ //If packer is more than half
    full
    out_buff.data = output_window(OUT_STRM_BITS-1,0); //Write out
    bottom half of output packer
    out_buff.keep = 0xFFFFFFFF; //Set all TKEEP bits high
    out_buff.last = 0;
    output_strm << out_buff;
    output_window >>= OUT_STRM_BITS; //Shift out bottom half of
    output packer
    output_bits -= OUT_STRM_BITS; //Update bit count of packer
  }
}

//Pipelined Huffman encoder for encoding all symbols of a window
    in parallel
void huffman_encoder(
    stream<in_stream>  &strm_in,
    stream<out_stream> &strm_out
){
#pragma HLS INTERFACE axis off port=strm_out
#pragma HLS INTERFACE axis off port=strm_in
  in_stream input_window0;
  in_stream input_window1;
  in_stream input_window2;
  in_stream input_window3;
  in_stream input_window4;
  in_stream input_window5;
  in_stream input_window6;
  in_stream input_window7;
  in_stream input_window8;
  in_stream input_window9;
  in_stream input_window10;
  in_stream input_window11;
  in_stream input_window12;
  in_stream input_window13;
  in_stream input_window14;
  in_stream input_window15;
  in_stream input_window16;
  t_enc_win encoded_window0;
  t_enc_win encoded_window1;
  t_enc_win encoded_window2;
  t_enc_win encoded_window3;
  t_enc_win encoded_window4;
  t_enc_win encoded_window5;
  t_enc_win encoded_window6;
  t_enc_win encoded_window7;
  t_enc_win encoded_window8;
  t_enc_win encoded_window9;
  t_enc_win encoded_window10;
  t_enc_win encoded_window11;
  t_enc_win encoded_window12;
  t_enc_win encoded_window13;
  t_enc_win encoded_window14;
  t_enc_win encoded_window15;
  t_enc_win encoded_window16;
  ap_uint<8> encoded_bits0;
  ap_uint<8> encoded_bits1;
  ap_uint<8> encoded_bits2;
  ap_uint<8> encoded_bits3;
  ap_uint<8> encoded_bits4;
  ap_uint<8> encoded_bits5;
  ap_uint<8> encoded_bits6;
  ap_uint<8> encoded_bits7;
  ap_uint<8> encoded_bits8;
  ap_uint<8> encoded_bits9;
  ap_uint<8> encoded_bits10;
  ap_uint<8> encoded_bits11;
  ap_uint<8> encoded_bits12;
  ap_uint<8> encoded_bits13;
  ap_uint<8> encoded_bits14;
  ap_uint<8> encoded_bits15;
  ap_uint<8> encoded_bits16;

  //Initialize output_window
  ap_uint<OUT_WIN_BITS> output_window = 0b011; //Output buffer for packing encoded windows
  ap_uint<10> output_bits = 3; //Number of bits currently in output_packer
  out_stream out_buff = {0};

  do{
#pragma HLS PIPELINE
    encoded_window0 = 0; //Clear encoded window before packing
    encoded_bits0 = 0;

    strm_in >> input_window0; //Read an input window from the input stream

    window_packer<0>(&input_window0, &input_window1, &encoded_window0, &encoded_bits0, &encoded_window1, &encoded_bits1);
    window_packer<1>(&input_window1, &input_window2, &encoded_window1, &encoded_bits1, &encoded_window2, &encoded_bits2);
    window_packer<2>(&input_window2, &input_window3, &encoded_window2, &encoded_bits2, &encoded_window3, &encoded_bits3);
    window_packer<3>(&input_window3, &input_window4, &encoded_window3, &encoded_bits3, &encoded_window4, &encoded_bits4);
    window_packer<4>(&input_window4, &input_window5, &encoded_window4, &encoded_bits4, &encoded_window5, &encoded_bits5);
    window_packer<5>(&input_window5, &input_window6, &encoded_window5, &encoded_bits5, &encoded_window6, &encoded_bits6);
    window_packer<6>(&input_window6, &input_window7, &encoded_window6, &encoded_bits6, &encoded_window7, &encoded_bits7);
    window_packer<7>(&input_window7, &input_window8, &encoded_window7, &encoded_bits7, &encoded_window8, &encoded_bits8);
    window_packer<8>(&input_window8, &input_window9, &encoded_window8, &encoded_bits8, &encoded_window9, &encoded_bits9);
    window_packer<9>(&input_window9, &input_window10, &encoded_window9, &encoded_bits9, &encoded_window10, &encoded_bits10);
    window_packer<10>(&input_window10, &input_window11, &encoded_window10, &encoded_bits10, &encoded_window11, &encoded_bits11);
    window_packer<11>(&input_window11, &input_window12, &encoded_window11, &encoded_bits11, &encoded_window12, &encoded_bits12);
    window_packer<12>(&input_window12, &input_window13, &encoded_window12, &encoded_bits12, &encoded_window13, &encoded_bits13);
    window_packer<13>(&input_window13, &input_window14, &encoded_window13, &encoded_bits13, &encoded_window14, &encoded_bits14);
    window_packer<14>(&input_window14, &input_window15, &encoded_window14, &encoded_bits14, &encoded_window15, &encoded_bits15);
    window_packer<15>(&input_window15, &input_window16, &encoded_window15, &encoded_bits15, &encoded_window16, &encoded_bits16);

    output_packer(encoded_window16, encoded_bits16, output_window, output_bits, strm_out);

  }while(!input_window0.last);
  output_bits += 7; //Add EOB code to end of stream (7 zeroes)

  if(output_bits >= OUT_STRM_BITS){ //If packer is more than half full, perform two final writes to stream
    out_buff.data = output_window(OUT_STRM_BITS-1,0); //Write out bottom half of output packer
    out_buff.keep = 0xFFFFFFFF; //Set all TKEEP bits high
    strm_out << out_buff;
    output_window >>= OUT_STRM_BITS; //Shift out bottom half of output packer
    output_bits -= OUT_STRM_BITS; //Update bit count of packer
    out_buff.data = output_window(OUT_STRM_BITS-1,0); //Write out remaining data in output packer
    out_buff.keep = decoder32[(output_bits+7)/8]; //Set TKEEP using decoder. Number of bytes rounded up
    out_buff.last = true;
    strm_out << out_buff;
  }
  else{ //Otherwise, just one final write to stream
    out_buff.data = output_window(OUT_STRM_BITS-1,0); //Write out remaining data in output packer
    out_buff.keep = decoder32[(output_bits+7)/8]; //Set TKEEP using decoder. Number of bytes rounded up
    out_buff.last = true;
    strm_out << out_buff;
  }
}
```

Appendix D

Huffman Encoder Header File

Huffman_Encoder.h

```
#include <stdio.h>
#include <string.h>
#include <ap_int.h>
#include <hls_stream.h>
#include <assert.h>

using namespace hls;

#define WINDOW_SIZE   16  //Number of boxes in input window
#define WINDOW_BITS   WINDOW_SIZE*8
#define ENC_WIN_BITS  (WINDOW_SIZE-1)*9 + 26 //Bit width of
    encoded window. Longest possible enc_win is 15 9-bit literals
    and 1 26-bit LD pair = 161 bits.
#define OUT_STRM_BITS 2*WINDOW_BITS //Output stream bit width,
    must be power of 2 and larger than encoded window width
#define OUT_WIN_BITS  2*OUT_STRM_BITS //Bit width of output window
     packer, must be double that of output stream width

typedef ap_uint<ENC_WIN_BITS> t_enc_win;

typedef struct{ //Input AXI-Stream structure
  ap_uint<3*WINDOW_BITS> data;  //3-byte DATA 'boxes' for each of
    the window characters from LZ77 encoder
  ap_uint<2*WINDOW_SIZE> user;  //2-bit TUSER flags for
    identifying each box as a literal, length, distance, or
    matched literal
  bool last;              //TLAST signal
} in_stream;

typedef struct{ //Output AXI-Stream structure
  ap_uint<OUT_STRM_BITS>   data; //Compressed data output
  ap_uint<OUT_STRM_BITS/8> keep; //TKEEP signal for each byte of
    data
  bool last;              //TLAST signal
} out_stream;

typedef struct{ //Distance code table structure
```

```
  ap_uint<5>  code;
  ap_uint<15> base;
  ap_uint<4>  bits;
} distance_code;

//32-bit decoder. Given an integer index n, returns a bit string containing n ones.
const ap_uint<32> decoder32[33] = {
     /* 0*/ 0b00000000000000000000000000000000,
     /* 1*/ 0b00000000000000000000000000000001,
     /* 2*/ 0b00000000000000000000000000000011,
     /* 3*/ 0b00000000000000000000000000000111,
     /* 4*/ 0b00000000000000000000000000001111,
     /* 5*/ 0b00000000000000000000000000011111,
     /* 6*/ 0b00000000000000000000000000111111,
     /* 7*/ 0b00000000000000000000000001111111,
     /* 8*/ 0b00000000000000000000000011111111,
     /* 9*/ 0b00000000000000000000000111111111,
     /*10*/ 0b00000000000000000000001111111111,
     /*11*/ 0b00000000000000000000011111111111,
     /*12*/ 0b00000000000000000000111111111111,
     /*13*/ 0b00000000000000000001111111111111,
     /*14*/ 0b00000000000000000011111111111111,
     /*15*/ 0b00000000000000000111111111111111,
     /*16*/ 0b00000000000000001111111111111111,
     /*17*/ 0b00000000000000011111111111111111,
     /*18*/ 0b00000000000000111111111111111111,
     /*19*/ 0b00000000000001111111111111111111,
     /*20*/ 0b00000000000011111111111111111111,
     /*21*/ 0b00000000000111111111111111111111,
     /*22*/ 0b00000000001111111111111111111111,
     /*23*/ 0b00000000011111111111111111111111,
     /*24*/ 0b00000000111111111111111111111111,
     /*25*/ 0b00000001111111111111111111111111,
     /*26*/ 0b00000011111111111111111111111111,
     /*27*/ 0b00000111111111111111111111111111,
     /*28*/ 0b00001111111111111111111111111111,
     /*29*/ 0b00011111111111111111111111111111,
     /*30*/ 0b00111111111111111111111111111111,
     /*31*/ 0b01111111111111111111111111111111,
     /*32*/ 0b11111111111111111111111111111111
};
```

Appendix E

Code Tables File

code_tables.cpp

```
//Length Codes (and Code Lengths) for all possible Length Values.
//Pre-mirrored with extra bits added
const ap_uint<8> length_code_table[14] = {
    /*Length 3*/   0b1000000, /*7*/ //7-bit length codes with no
    extra bits
    /*Length 4*/   0b0100000, /*7*/
    /*Length 5*/   0b1100000, /*7*/
    /*Length 6*/   0b0010000, /*7*/
    /*Length 7*/   0b1010000, /*7*/
    /*Length 8*/   0b0110000, /*7*/
    /*Length 9*/   0b1110000, /*7*/
    /*Length 10*/  0b0001000, /*7*/
    /*Length 11*/ 0b01001000, /*8*/ //7-bit length codes with 1
    extra bit
    /*Length 12*/ 0b11001000, /*8*/
    /*Length 13*/ 0b00101000, /*8*/
    /*Length 14*/ 0b10101000, /*8*/
    /*Length 15*/ 0b01101000, /*8*/
    /*Length 16*/ 0b11101000  /*8*/
};

//Table for looking up the {Distance Symbol/Code, Base Value, and
    Extra Bits} of a Distance Value
//Since input distance is 1 less than the actual distance, the
    base values are pre-decremented by 1 as well
//This way the difference between the two is correct when
    calculating: extra_value = distance - base_value
const distance_code distance_code_table[512] = {
    {0, 0, 0},  {1, 1, 0},  {2, 2, 0},  {3, 3,  0}, {4, 4,  1},
    {4, 4,  1}, {5, 6,  1}, {5, 6,  1},
    {6, 8,  2}, {6, 8,  2}, {6, 8,  2}, {6, 8,  2}, {7, 12, 2},
    {7, 12, 2}, {7, 12, 2}, {7, 12, 2},
    {8, 16, 3}, {8, 16, 3}, {8, 16, 3}, {8, 16, 3}, {8, 16, 3},
    {8, 16, 3}, {8, 16, 3}, {8, 16, 3},
    {9, 24, 3}, {9, 24, 3}, {9, 24, 3}, {9, 24, 3}, {9, 24, 3},
    {9, 24, 3}, {9, 24, 3}, {9, 24, 3},
 {10, 32, 4}, {10, 32, 4}, {10, 32, 4}, {10, 32, 4}, {10, 32,
4}, {10, 32, 4}, {10, 32, 4}, {10, 32, 4},
 {10, 32, 4}, {10, 32, 4}, {10, 32, 4}, {10, 32, 4}, {10, 32,
4}, {10, 32, 4}, {10, 32, 4}, {10, 32, 4},
 {11, 48, 4}, {11, 48, 4}, {11, 48, 4}, {11, 48, 4}, {11, 48,
4}, {11, 48, 4}, {11, 48, 4}, {11, 48, 4},
 {11, 48, 4}, {11, 48, 4}, {11, 48, 4}, {11, 48, 4}, {11, 48,
4}, {11, 48, 4}, {11, 48, 4}, {11, 48, 4},
 {12, 64, 5}, {12, 64, 5}, {12, 64, 5}, {12, 64, 5}, {12, 64,
5}, {12, 64, 5}, {12, 64, 5}, {12, 64, 5},
 {12, 64, 5}, {12, 64, 5}, {12, 64, 5}, {12, 64, 5}, {12, 64,
5}, {12, 64, 5}, {12, 64, 5}, {12, 64, 5},
 {12, 64, 5}, {12, 64, 5}, {12, 64, 5}, {12, 64, 5}, {12, 64,
5}, {12, 64, 5}, {12, 64, 5}, {12, 64, 5},
 {12, 64, 5}, {12, 64, 5}, {12, 64, 5}, {12, 64, 5}, {12, 64,
5}, {12, 64, 5}, {12, 64, 5}, {12, 64, 5},
 {13, 96, 5}, {13, 96, 5}, {13, 96, 5}, {13, 96, 5}, {13, 96,
5}, {13, 96, 5}, {13, 96, 5}, {13, 96, 5},
 {13, 96, 5}, {13, 96, 5}, {13, 96, 5}, {13, 96, 5}, {13, 96,
5}, {13, 96, 5}, {13, 96, 5}, {13, 96, 5},
 {13, 96, 5}, {13, 96, 5}, {13, 96, 5}, {13, 96, 5}, {13, 96,
5}, {13, 96, 5}, {13, 96, 5}, {13, 96, 5},
 {13, 96, 5}, {13, 96, 5}, {13, 96, 5}, {13, 96, 5}, {13, 96,
5}, {13, 96, 5}, {13, 96, 5}, {13, 96, 5},
 {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14,
128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6},
 {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14,
128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6},
 {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14,
128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6},
 {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14,
128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6},
 {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14,
128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6},
 {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14,
128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6},
 {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14,
128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6},
 {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6}, {14,
128, 6}, {14, 128, 6}, {14, 128, 6}, {14, 128, 6},
 {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15,
192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6},
 {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15,
192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6},
 {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15,
192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6},
 {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15,
192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6},
 {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15,
192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6},
 {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15,
192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6},
 {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15,
192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6},
 {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6}, {15,
192, 6}, {15, 192, 6}, {15, 192, 6}, {15, 192, 6},
 { 0,   0, 0}, { 0,   0, 0}, {16, 256, 7}, {17, 384, 7}, {18,
512, 8}, {18, 512, 8}, {19, 768, 8}, {19, 768, 8},
 {20, 1024,  9}, {20, 1024,  9}, {20, 1024,  9}, {20, 1024,
9}, {21, 1536,  9}, {21, 1536,  9}, {21, 1536,  9}, {21, 1536,
  9},
 {22, 2048,  10}, {22, 2048, 10}, {22, 2048, 10}, {22, 2048,
10}, {22, 2048, 10}, {22, 2048, 10}, {22, 2048, 10}, {22,
2048, 10},
 {23, 3072,  10}, {23, 3072, 10}, {23, 3072, 10}, {23, 3072,
10}, {23, 3072, 10}, {23, 3072, 10}, {23, 3072, 10}, {23,
3072, 10},
 {24, 4096,  11}, {24, 4096, 11}, {24, 4096, 11}, {24, 4096,
11}, {24, 4096, 11}, {24, 4096, 11}, {24, 4096, 11}, {24,
4096, 11},
 {24, 4096,  11}, {24, 4096, 11}, {24, 4096, 11}, {24, 4096,
11}, {24, 4096, 11}, {24, 4096, 11}, {24, 4096, 11}, {24,
4096, 11},
 {25, 6144,  11}, {25, 6144, 11}, {25, 6144, 11}, {25, 6144,
11}, {25, 6144, 11}, {25, 6144, 11}, {25, 6144, 11}, {25,
6144, 11},
 {25, 6144,  11}, {25, 6144, 11}, {25, 6144, 11}, {25, 6144,
11}, {25, 6144, 11}, {25, 6144, 11}, {25, 6144, 11}, {25,
6144, 11},
 {26, 8192,  12}, {26, 8192, 12}, {26, 8192, 12}, {26, 8192,
12}, {26, 8192, 12}, {26, 8192, 12}, {26, 8192, 12}, {26,
8192, 12},
 {26, 8192,  12}, {26, 8192, 12}, {26, 8192, 12}, {26, 8192,
12}, {26, 8192, 12}, {26, 8192, 12}, {26, 8192, 12}, {26,
8192, 12},
 {26, 8192,  12}, {26, 8192, 12}, {26, 8192, 12}, {26, 8192,
12}, {26, 8192, 12}, {26, 8192, 12}, {26, 8192, 12}, {26,
8192, 12},
 {26, 8192,  12}, {26, 8192, 12}, {26, 8192, 12}, {26, 8192,
12}, {26, 8192, 12}, {26, 8192, 12}, {26, 8192, 12}, {26,
8192, 12},
 {27, 12288, 12}, {27, 12288, 12}, {27, 12288, 12}, {27, 12288,
 12}, {27, 12288, 12}, {27, 12288, 12}, {27, 12288, 12}, {27,
12288, 12},
 {27, 12288, 12}, {27, 12288, 12}, {27, 12288, 12}, {27, 12288,
 12}, {27, 12288, 12}, {27, 12288, 12}, {27, 12288, 12}, {27,
12288, 12},
 {27, 12288, 12}, {27, 12288, 12}, {27, 12288, 12}, {27, 12288,
 12}, {27, 12288, 12}, {27, 12288, 12}, {27, 12288, 12}, {27,
12288, 12},
 {27, 12288, 12}, {27, 12288, 12}, {27, 12288, 12}, {27, 12288,
 12}, {27, 12288, 12}, {27, 12288, 12}, {27, 12288, 12}, {27,
12288, 12},
 {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384,
 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28,
16384, 13},
 {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384,
 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28,
    16384, 13},
    {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384,
     13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28,
    16384, 13},
    {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384,
     13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28,
    16384, 13},
    {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384,
     13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28,
    16384, 13},
    {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384,
     13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28,
    16384, 13},
    {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384,
     13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28,
    16384, 13},
    {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384,
     13}, {28, 16384, 13}, {28, 16384, 13}, {28, 16384, 13}, {28,
    16384, 13},
    {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576,
     13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29,
    24576, 13},
    {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576,
     13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29,
    24576, 13},
    {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576,
     13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29,
    24576, 13},
    {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576,
     13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29,
    24576, 13},
    {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576,
     13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29,
    24576, 13},
    {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576,
     13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29,
    24576, 13},
    {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576,
     13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29,
    24576, 13},
    {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576,
     13}, {29, 24576, 13}, {29, 24576, 13}, {29, 24576, 13}, {29,
    24576, 13}
};
```

Appendix F

Huffman Decoder Source File

Huffman_Decoder.cpp

```
#include "Huffman_Decoder.h"
#include "symbol_tables.cpp"
#include "static_decoding_tables.cpp"

//Function for filling accumulator with bits from input stream. Input variable "num" is the number of bits to obtain.
//If accumulator already contains the number of needed bits, no more are read.
void fetch_bits(
    int num,              //Input
    stream<io_stream> &in_strm, //Input
    io_stream &in_buff,         //In/Out
    ap_uint<8> &bit_count,          //In/Out
    ap_uint<64> &acc            //In/Out
){
#pragma HLS INLINE
  //Inlining function automatically optimizes function usage to read one byte or two depending on input num
  if((bit_count < num) & !in_buff.last){ //If bit_count is still less than num and TLAST hasn't been asserted, read another byte
    in_strm >> in_buff; //Read 1 byte from input stream into buffer
    acc |= (ap_uint<64>)in_buff.data << bit_count;  //Append data to the left of current bits in acc. Assuming little endian data
    bit_count += 32; //Update number of bits in acc
  }
}

//Reset code length code length counts. Takes 1 cycle
void reset_CL_codelength_count(ap_uint<8> CL_codelength_count[MAX_CL_LENGTH+1]){
#pragma HLS INLINE off
  for(int i = 0; i < (MAX_CL_LENGTH+1); i++){
#pragma HLS UNROLL
    CL_codelength_count[i] = 0;
  }
}

//Reset code length code lengths. Takes 1 cycle
void reset_CL_codelengths(ap_uint<3> CL_codelengths[MAX_CL_CODES])
    {
#pragma HLS INLINE off
  for(int i = 0; i < MAX_CL_CODES; i++){
#pragma HLS UNROLL
    CL_codelengths[i] = 0;
  }
}

//Reset length and distance code length counts. Takes 1 cycle
void reset_LD_codelength_count(ap_uint<9> length_codelength_count[
    MAX_LEN_LENGTH+1], ap_uint<9> distance_codelength_count[
    MAX_DIS_LENGTH+1]){
#pragma HLS INLINE off
  for(int i = 0; i < (MAX_LEN_LENGTH+1); i++){
#pragma HLS UNROLL
    length_codelength_count[i] = 0;
    distance_codelength_count[i] = 0;
  }
}

//Reset length code lengths. Takes 29 cycles
void reset_length_codelengths(ap_uint<4> length_codelengths[
    MAX_LEN_CODES]){
#pragma HLS INLINE off
  for(int i = MIN_LEN_CODES; i < MAX_LEN_CODES; i++){ //Only last
     30 indexes need to be cleared, all others will always be
     filled
    //#pragma HLS UNROLL
    length_codelengths[i] = 0;
  }
}

//Reset distance code lengths. Takes 32 cycles
void reset_distance_codelengths(ap_uint<4> distance_codelengths[
    MAX_DIS_CODES]){
#pragma HLS INLINE off
  for(int i = 0; i < MAX_DIS_CODES; i++){
    //#pragma HLS UNROLL
    distance_codelengths[i] = 0;
  }
}

//Build length code table. Takes 286 cycles
void build_length_table(
    ap_uint<4> length_codelengths[MAX_LEN_CODES],     //Input
    ap_uint<9> length_code_table[MAX_LEN_CODES],      //Output
    ap_uint<9> next_length_address[MAX_LEN_LENGTH+1] //Input
){
#pragma HLS INLINE off
```

```
  ap_uint<4> length_codelength; //Code length of encoded Length/
    Literal
  L_Table_LoopC: for(int i = 0; i < MAX_LEN_CODES; i++){ //Build
    Length/Literal Table
#pragma HLS PIPELINE
     length_codelength = length_codelengths[i]; //Read Length/
    Literal code length
     if(length_codelength != 0){ //If the length of a code is not 0
       length_code_table[next_length_address[length_codelength]] =
    i; //At coded index, write symbol
       next_length_address[length_codelength]++;
     }
  }
}

//Build distance code table. Takes 32 cycles
void build_distance_table(
     ap_uint<4> distance_codelengths[MAX_DIS_CODES],
     ap_uint<8> distance_code_table[MAX_DIS_CODES],
     ap_uint<8> next_distance_address[MAX_DIS_LENGTH+1]
){
#pragma HLS INLINE off
  ap_uint<4> distance_codelength; //Code length of encoded
    Distance
  D_Table_LoopC: for(int i = 0; i < MAX_DIS_CODES; i++){ //Build
    Distance Table
#pragma HLS PIPELINE
     distance_codelength = distance_codelengths[i]; //Read Distance
     code length
     if(distance_codelength != 0){ //If the length of a code is not
     0
       distance_code_table[next_distance_address[
    distance_codelength]] = i; //At coded index, write symbol
       next_distance_address[distance_codelength]++;
     }
  }
}

void decode_static_block(
     stream<io_stream> &in_strm,   //Input
     io_stream &in_buff,         //In/Out
     ap_uint<8> &bit_count,          //In/Out
     ap_uint<64> &acc,           //In/Out
     bool &end_of_block,         //In/Out
     stream<lld_stream> &out_strm //Output
){
  ap_uint<9> static_code_bits; //9 bits containing a static length
    /distance code, actual code length may be 7 to 9 bits long
  static_symbol length_symbol; //Decoded Length/Literal Huffman
    symbol
  static_symbol distance_symbol; //Decoded Distance Huffman symbol
  ap_uint<9> match_length; //Match length variable
  ap_uint<16> match_distance; //Match distance variable
  ap_uint<5> extra_length; //Integer value of extra length
  ap_uint<13> extra_distance; //Integer value of extra distance
  lld_stream out_buff; //Buffer for writing to output stream

  Static_Block_Loop: while(!end_of_block){ //Continue processing a block until end-of-block code found
     fetch_bits(32, in_strm, in_buff, bit_count, acc);
     static_code_bits = acc(8,0); //Take in 9 bits from ACC
     length_symbol = static_length_table[static_code_bits]; //Decode in static literal/length table
     acc >>= length_symbol.bits;   //Shift out bits from acc
     bit_count -= length_symbol.bits; //Update bit count
     switch (length_symbol.type(7, 4)){ //Check upper four bits of type
     case 0 : //Literal
       out_buff.data = length_symbol.base; //Write Literal to output
       out_buff.keep = 0b0001; //Update TKEEP bits to indicate Literal
       out_buff.user = 0;
       out_buff.last = 0; //Assign TLAST
       out_strm << out_buff; //Write to output stream
       break;
     case 1: // Length/Distance
       extra_length = acc & ((1 << length_symbol.type(3,0)) - 1 ); //Fetch extra length bits from acc
       acc >>= length_symbol.type(3,0); //Discard extra length bits from acc
       bit_count -= length_symbol.type(3,0); //Update bit count accordingly
       match_length = length_symbol.base + extra_length; //Add base and extra to get length. Can be 3-258 (9 bits)

       distance_symbol = static_distance_table[acc(4,0)]; //Decode in static distance table
       acc >>= STATIC_DIS_LENGTH; //Shift out bits from acc
       bit_count -= STATIC_DIS_LENGTH; //Update bit count
       extra_distance = acc & ((1 << distance_symbol.type(3,0)) - 1 ); //Fetch extra distance bits from acc
       acc >>= distance_symbol.type(3,0); //Discard extra distance bits from acc
       bit_count -= distance_symbol.type(3,0); //Update bit count accordingly
       match_distance = distance_symbol.base + extra_distance; //Add base and extra to get distance. Can be 1-32768 (16 bits)

       out_buff.data(31,23) = match_length; //Write 9-bit match length to output
       out_buff.data(15, 0) = match_distance; //Write 16-bit match distance to output
       out_buff.keep = 0b1111; //Update TKEEP bits to indicate length-distance pair
       out_buff.user = 1;
       out_buff.last = 0; //Assign TLAST
       out_strm << out_buff; //Write to output stream
       break;
     case 6: //End-of-Block
       end_of_block = true; //Update EOB flag, exit Block loop and return to Main Loop
       break;
     }
  }
}

void decode_dynamic_block(
     stream<io_stream> &in_strm,    //Input
     io_stream &in_buff,          //In/Out
     ap_uint<8> &bit_count,           //In/Out
     ap_uint<64> &acc,            //In/Out
     bool &end_of_block,          //In/Out
     stream<lld_stream> &out_strm, //Output
     ap_uint<9> length_codelength_count[MAX_LEN_LENGTH+1],   //Input
     ap_uint<9> distance_codelength_count[MAX_DIS_LENGTH+1], //Input
     ap_uint<15> base_length_values[MAX_LEN_LENGTH+1],    //Input
     ap_uint<15> base_distance_values[MAX_DIS_LENGTH+1],  //Input
     ap_uint<9> base_length_address[MAX_LEN_LENGTH+1],    //Input
     ap_uint<8> base_distance_address[MAX_DIS_LENGTH+1],  //Input
     ap_uint<9> length_code_table[MAX_LEN_CODES],         //Input
     ap_uint<8> distance_code_table[MAX_DIS_CODES]        //Input
){
  ap_uint<15> code_bits; //15 bits containing a length/distance code, actual code length may be 0 to 15 bits long
  ap_uint<16> code_comparison; //Bit array for comparing code_bits to base_length_values
  ap_uint<4> code_length; //Deciphered length of currently held length code
  ap_uint<9> code_address; //Register for calculating length code address
  ap_uint<9> length_symbol; //Decoded Length/Literal Huffman symbol
  ap_uint<5> distance_symbol; //Decoded Distance Huffman symbol
  length_symbol_values length_symbol_extras;
  distance_symbol_values distance_symbol_extras;
  ap_uint<9> match_length; //Match length variable
  ap_uint<16> match_distance; //Match distance variable
  ap_uint<5> extra_length; //Integer value of extra length
  ap_uint<13> extra_distance; //Integer value of extra distance
  lld_stream out_buff; //Buffer for writing to output stream

  Dynamic_Block_Loop: while(!end_of_block){ //Continue processing a block until end-of-block code found
     fetch_bits(MAX_LEN_LENGTH, in_strm, in_buff, bit_count, acc); //Fetch 15 bits for Length/Literal code
     code_bits = acc(0,14); //Take in 15 bits from ACC and reverse them
     Length_Code_Compare_Loop: for(int i = 1; i <= MAX_LEN_LENGTH; i++){ //Compare bits to every length base_value to find code
    length of bits
#pragma HLS UNROLL
      code_comparison[i] = (length_codelength_count[i] != 0) ? (
    code_bits >= (base_length_values[i] << (MAX_LEN_LENGTH - i)))
    : 0;
     } //Comparisons with lengths that aren't used are disabled (
    They would always be 1 since base value would be 0)
     code_length = MAX_LEN_LENGTH - (__builtin_clz(code_comparison)
     - 16); //Code length is bit position of highest passing
    comparison. CLZ adds 16 bits
     code_bits >>= (MAX_LEN_LENGTH - code_length); //Shift bits out
     so only codelength remain
     code_address = base_length_address[code_length] + (code_bits -
     base_length_values[code_length]); //Address = base + offset
     length_symbol = length_code_table[code_address]; //Decode
    length code in lookup table
     acc >>= code_length;    //Shift out bits from acc
     bit_count -= code_length; //Update bit count
     if(length_symbol < 256){ //If decoded Length/Literal symbol is
     from 0 to 255, it is a Literal
       out_buff.data = length_symbol; //Write Literal to output
       out_buff.keep = 0b0001; //Update TKEEP bits to indicate
    Literal
       out_buff.user = 0;
       out_buff.last = 0; //Assign TLAST
       out_strm << out_buff; //Write to output stream
     }
     else if(length_symbol == 256){ //End-of-block symbol
       end_of_block = true; //Update EOB flag, exit Block loop and
    return to Main Loop
     }
     else{ //If length_symbol > 256, It is a Length
       length_symbol_extras = length_symbol_table[length_symbol-
    MIN_LEN_CODES]; //Lookup base length and extra bits in table
       fetch_bits(MAX_LEN_EXTRA_BITS+MAX_DIS_LENGTH, in_strm,
    in_buff, bit_count, acc); //Fetch 20 bits for Length code
    extra bits and distance
       extra_length = acc & ((1 << length_symbol_extras.bits) - 1 )
    ; //Fetch extra bits from acc
       acc >>= length_symbol_extras.bits; //Discard extra bits from
     acc
       bit_count -= length_symbol_extras.bits; //Update bit count
    accordingly
       match_length = length_symbol_extras.base + extra_length + 3;
     //Add base and extra to get length. Base Lengths 3-258 are
    encoded in 0-255 (8 bits) so we add 3

       code_bits = acc(0,14); //Take in 15 bits from ACC and
    reverse them
       Distance_Code_Compare_Loop: for(int i = 1; i <=
    MAX_DIS_LENGTH; i++){ //Compare bits to every distance
    base_value to find code length of bits
#pragma HLS UNROLL
```

```
            code_comparison[i] = (distance_codelength_count[i] != 0) ? (code_bits >= (base_distance_values[i] << (MAX_DIS_LENGTH - i))) : 0;
        } //Comparisons with lengths that aren't used are disabled (They would always be 1 since base value would be 0)
        code_length = MAX_DIS_LENGTH - (__builtin_clz(code_comparison) - 16); //Code length is bit position of highest passing comparison. CLZ adds 16 bits
        code_bits >>= (MAX_DIS_LENGTH - code_length); //Shift bits out so only code length remain
        code_address = base_distance_address[code_length] + (code_bits - base_distance_values[code_length]); //Address = base + offset
        distance_symbol = distance_code_table[code_address]; //Decode distance code in lookup table
        acc >>= code_length;     //Shift out bits from acc
        bit_count -= code_length; //Update bit count
        distance_symbol_extras = distance_symbol_table[distance_symbol]; //Lookup base distance and extra bits in table
        fetch_bits(MAX_DIS_EXTRA_BITS, in_strm, in_buff, bit_count, acc); //Fetch 13 bits for Distance code extra bits
        extra_distance = acc & ((1 << distance_symbol_extras.bits) - 1 ); //Fetch extra bits from acc
        acc >>= distance_symbol_extras.bits; //Discard extra bits from acc
        bit_count -= distance_symbol_extras.bits; //Update bit count accordingly
        match_distance = distance_symbol_extras.base + extra_distance; //Add base and extra to get distance. Can be 1-32768 (16 bits)

        out_buff.data(31,23) = match_length; //Write 9-bit match length to output
        out_buff.data(15, 0) = match_distance; //Write 16-bit match distance to output
        out_buff.keep = 0b1111; //Update TKEEP bits to indicate length-distance pair
        out_buff.user = 1;
        out_buff.last = 0; //Assign TLAST
        out_strm << out_buff; //Write to output stream
      }
    }
}

void build_dynamic_tables(
     stream<io_stream> &in_strm, //Input
     io_stream &in_buff,        //In/Out
     ap_uint<8> &bit_count,          //In/Out
     ap_uint<64> &acc,          //In/Out
     ap_uint<4> length_codelengths[MAX_LEN_CODES],       //Input
     ap_uint<4> distance_codelengths[MAX_DIS_CODES],      //Input
     ap_uint<9> length_codelength_count[MAX_LEN_LENGTH+1],    //Output
     ap_uint<9> distance_codelength_count[MAX_DIS_LENGTH+1], //Output
     ap_uint<15> base_length_values[MAX_LEN_LENGTH+1],     //Output
     ap_uint<15> base_distance_values[MAX_DIS_LENGTH+1],   //Output
     ap_uint<9> base_length_address[MAX_LEN_LENGTH+1],     //Output
     ap_uint<8> base_distance_address[MAX_DIS_LENGTH+1],   //Output
     ap_uint<9> length_code_table[MAX_LEN_CODES],          //Output
     ap_uint<8> distance_code_table[MAX_DIS_CODES]         //Output
){
    ap_uint<9> num_length_codelengths; //Number of Length/Literal code lengths in sequence, from 257 to 286
    ap_uint<8> num_distance_codelengths; //Number of Distance code lengths in sequence, from 1 to 32
    ap_uint<8> num_codelength_codelengths; //Number of Code_Length code lengths in sequence, from 4 to 19
    ap_uint<3> CL_codelength; //Code length of encoded Code Length. CLCLs can be from 0 to 7 bits long
    ap_uint<3> CL_codelengths[MAX_CL_CODES]; //Array for storing CL code lengths
#pragma HLS ARRAY_PARTITION variable=CL_codelengths complete dim=1
    ap_uint<8> CL_codelength_count[MAX_CL_LENGTH+1]; //Array for counting number of times each CLCL occurs
#pragma HLS ARRAY_PARTITION variable=CL_codelength_count complete dim=1
    ap_uint<7> base_CL_values[MAX_CL_LENGTH+1]; //Calculated base value for each CLCL
#pragma HLS ARRAY_PARTITION variable=base_CL_values complete dim=1
    ap_uint<5> base_CL_address[MAX_CL_LENGTH+1]; //Base address for each CLCL base value
#pragma HLS ARRAY_PARTITION variable=base_CL_address complete dim=1
    ap_uint<5> next_CL_address[MAX_CL_LENGTH+1]; //Contains the next address for each CLC symbol
#pragma HLS ARRAY_PARTITION variable=next_CL_address complete dim=1
    ap_uint<5> codelength_address; //Register for calculating code_length code address
    ap_uint<5> CL_code_table[MAX_CL_CODES]; //Table containing symbols for 19 CL Codes. Codes are indexed using base_CL_addresses
    ap_uint<9> codelengths_received; //For counting number of code lengths read from sequence
    ap_uint<7> codelength_bits; //7 bits containing a length/distance code length code, actual code length may be 0 to 7 bits long
    ap_uint<8> CL_comparison; //Bit array for comparing codelength_bits to base_CL_values
    ap_uint<3> codelength_length; //Deciphered length of currently held code length code
    ap_uint<5> CL_symbol; //Decoded Code Length symbol (Can be 0 to 18)
    ap_uint<9> length_symbol_counter; //Index counter for current Length/Literal symbol in sequence
    ap_uint<5> distance_symbol_counter; //Index counter for current Distance symbol in sequence
    ap_uint<9> next_length_address[MAX_LEN_LENGTH+1]; //Contains the next address for each Length/Literal symbol
#pragma HLS ARRAY_PARTITION variable=next_length_address complete dim=1
    ap_uint<8> next_distance_address[MAX_DIS_LENGTH+1]; //Contains the next address for each Distance symbol
#pragma HLS ARRAY_PARTITION variable=next_distance_address complete dim=1
    ap_uint<5> previous_CL; //Register for storing found_CL for next iteration
    ap_uint<8> repeat_value; //Used by code length symbols 16, 17, and 18 for storing number of repetitions to perform

    reset_CL_codelength_count(CL_codelength_count);
    reset_CL_codelengths(CL_codelengths);
    reset_LD_codelength_count(length_codelength_count, distance_codelength_count);

    //Read code length codes and build code length table
    fetch_bits(14, in_strm, in_buff, bit_count, acc);
    num_length_codelengths = acc(4,0) + 257; //There will always be at least 257 Length/Literal codes for the 256 Literals and the EOB code
    num_distance_codelengths = acc(9,5) + 1; //There will always be at least 1 distance code
    num_codelength_codelengths = acc(13,10) + 4; //There will always be at least 4 code length code lengths (for symbols 16, 17, 18, and 0)
    assert(num_length_codelengths < 287); //Max 286 Length/Literal code lengths
    assert(num_distance_codelengths < 33); //Max 32 Distance code lengths
    assert(num_codelength_codelengths < 20); //Max 19 Code Length code lengths
    acc >>= 14;      //Shift out bits from acc
    bit_count -= 14;  //Update bit count

    //Build code_length table
    //1: Count number of codes for each length
    CL_Table_LoopA: for(int i = 0; i < num_codelength_codelengths; i++){
#pragma HLS PIPELINE
     fetch_bits(3, in_strm, in_buff, bit_count, acc);
     CL_codelength = acc(2,0); //Read 3-bit code length
     CL_codelengths[permuted_order[i]] = CL_codelength; //Store code length in array in permuted order
     CL_codelength_count[CL_codelength]++; //Take count of number of code lengths
     acc >>= 3;       //Shift out bits from acc
     bit_count -= 3;    //Update bit count
    }
    //2: Calculate base value for each code length
    CL_codelength_count[0] = 0; //Set back to 0 before calculating base values (Codes with length of 0 are unused)
    base_CL_values[0] = 0;
    base_CL_address[0] = 0;
    CL_Table_LoopB: for(int i = 1; i <= MAX_CL_LENGTH; i++){ //For each possible code length
#pragma HLS PIPELINE
     base_CL_values[i] = (base_CL_values[i-1] + CL_codelength_count[i-1]) << 1; //Base value for a code length is based on number of previous code lengths
     base_CL_address[i] = base_CL_address[i-1] + CL_codelength_count[i-1]; //Base address is "number of code lengths" away from previous base address
     next_CL_address[i] = base_CL_address[i]; //Initialize "next address" to base address for this length
    }
    //3: Assign consecutive values (addresses) to each code for all code lengths
    CL_Table_LoopC: for(int i = 0; i < MAX_CL_CODES; i++){ //For all 19 code length symbols
#pragma HLS PIPELINE
     CL_codelength = CL_codelengths[i]; //Read CL code length
     if(CL_codelength != 0){ //If the length of a code is not 0
       CL_code_table[next_CL_address[CL_codelength]] = i;
       next_CL_address[CL_codelength]++;
     }
    }

    //Read encoded code length sequence and decode it using that table
    codelengths_received = 0; //Reset code lengths received
    length_symbol_counter = 0; //Reset Length/Literal array index pointer
    distance_symbol_counter = 0; //Reset Distance array index pointer
    LD_Table_LoopA: while(codelengths_received < (num_length_codelengths+num_distance_codelengths)){ //Retrieve all code lengths in sequence
     //Sequence can be from 258 to 318 code lengths long
     fetch_bits(7, in_strm, in_buff, bit_count, acc);
     codelength_bits = acc(0,6); //Take in 7 bits from ACC and reverse them
     CL_Comparison_Loop: for(int i = 1; i <= MAX_CL_LENGTH; i++){ //Compare bits to every length base_value to find code length of bits
#pragma HLS UNROLL
       CL_comparison[i] = (CL_codelength_count[i] != 0) ? (codelength_bits >= (base_CL_values[i] << (MAX_CL_LENGTH - i))) : 0;
     } //Comparisons with lengths that aren't used are disabled (They would always be 1 since base value would be 0)
     codelength_length = MAX_CL_LENGTH - (__builtin_clz(CL_comparison) - 24); //Code length is bit position of highest passing comparison. CLZ adds 24 bits
     codelength_bits >>= (MAX_CL_LENGTH - codelength_length); //
 Shift bits out so only codelength remain
     codelength_address = base_CL_address[codelength_length] + (
 codelength_bits - base_CL_values[codelength_length]);
     CL_symbol = CL_code_table[codelength_address]; //Decode code
 length in lookup table
     acc >>= codelength_length;      //Shift out bits from acc
     bit_count -= codelength_length; //Update bit count
     if(CL_symbol <= 15){ //If decoded code length symbol is from 0
     to 15
       if(codelengths_received < num_length_codelengths){ //If
 Length code length
         length_codelengths[length_symbol_counter] = CL_symbol; //
 Current symbol in sequence has code length of found_CL
         length_codelength_count[CL_symbol]++; //Count code length
         length_symbol_counter++; //Increment symbol pointer
       }
       else{ //If Distance code length
         distance_codelengths[distance_symbol_counter] = CL_symbol;
     //Current symbol in sequence has code length of found_CL
         distance_codelength_count[CL_symbol]++; //Count code
 length
         distance_symbol_counter++;
       }
       codelengths_received++; //Increment code length counter
       previous_CL = CL_symbol; //Save code length for next
 iteration
     }
     else{
       switch(CL_symbol){
       case 16 : //Symbol 16: Copy previous code length 3 to 6
 times
         fetch_bits(2, in_strm, in_buff, bit_count, acc); //Fetch 2
     extra bits containing repeat value
         repeat_value = acc(1,0) + 3;
         acc >>= 2;      //Shift out bits from acc
         bit_count -= 2; //Update bit count
         Copy_Previous_CL_Loop: for(int i = 0; i < repeat_value; i
 ++){ //For 3 to 6 times
           //#pragma HLS PIPELINE
           if(codelengths_received < num_length_codelengths){ //If
 Length code length
             length_codelengths[length_symbol_counter] =
 previous_CL; //Current symbol in sequence has code length of
 previous_CL
             length_codelength_count[previous_CL]++; //Count code
 length
             length_symbol_counter++; //Increment symbol pointer
           }
           else{ //If Distance code length
             distance_codelengths[distance_symbol_counter] =
 previous_CL; //Current symbol in sequence has code length of
 previous_CL
```

```
          distance_codelength_count[previous_CL]++; //Count code length
          distance_symbol_counter++;
        }
        codelengths_received++; //Increment code length counter
      }
      break;
    case 17 : //Symbol 17: Repeat code length of zero 3 to 10 times
      fetch_bits(3, in_strm, in_buff, bit_count, acc); //Fetch 3 extra bits containing repeat value
      repeat_value = acc(2,0) + 3;
      acc >>= 3;     //Shift out bits from acc
      bit_count -= 3; //Update bit count
      Repeat_Zero_CL_LoopA: for(int i = 0; i < repeat_value; i++){ //For 3 to 10 times
        //#pragma HLS PIPELINE
        if(codelengths_received < num_length_codelengths){ //If Length code length
          length_codelengths[length_symbol_counter] = 0; //Current symbol in sequence has code length of 0
          length_symbol_counter++; //Increment symbol pointer
        }
        else{ //If Distance code length
          distance_codelengths[distance_symbol_counter] = 0; //Current symbol in sequence has code length of 0
          distance_symbol_counter++; //Increment symbol pointer
        }
        codelengths_received++; //Increment code length counter
      }
      break;
    case 18 : //Symbol 18: Repeat code length of zero 11 to 138 times
      fetch_bits(7, in_strm, in_buff, bit_count, acc); //Fetch 7 extra bits containing repeat value
      repeat_value = acc(6,0) + 11;
      acc >>= 7;     //Shift out bits from acc
      bit_count -= 7; //Update bit count
      Repeat_Zero_CL_LoopB: for(int i = 0; i < repeat_value; i++){ //For 11 to 138 times
        //#pragma HLS PIPELINE
        if(codelengths_received < num_length_codelengths){ //If Length code length
          length_codelengths[length_symbol_counter] = 0; //Current symbol in sequence has code length of 0
          length_symbol_counter++; //Increment symbol pointer
        }
        else{ //If Distance code length
          distance_codelengths[distance_symbol_counter] = 0; //Current symbol in sequence has code length of 0
          distance_symbol_counter++; //Increment symbol pointer
        }
        codelengths_received++; //Increment code length counter
      }
```

```
        break;
      default :
        //Shouldn't occur, misread codes should be mostly 0s, could be other values from previous iterations if tables arent cleared.
        break;
      }
    }
  }

  //When all length and distance code lengths are received
  length_codelength_count[0] = 0; //Set back to 0 before calculating base values (Codes with length of 0 are unused)
  distance_codelength_count[0] = 0;
  base_length_values[0] = 0;
  base_length_address[0] = 0;
  base_distance_values[0] = 0;
  base_distance_address[0] = 0;
  //Calculate Base Values and Base Addresses
  LD_Table_LoopB: for(int i = 1; i <= MAX_LEN_LENGTH; i++){ //For each possible code length
#pragma HLS PIPELINE
    base_length_values[i] = (base_length_values[i-1] + length_codelength_count[i-1]) << 1;
    base_length_address[i] = base_length_address[i-1] + length_codelength_count[i-1];
    next_length_address[i] = base_length_address[i];
    base_distance_values[i] = (base_distance_values[i-1] + distance_codelength_count[i-1]) << 1;
    base_distance_address[i] = base_distance_address[i-1] + distance_codelength_count[i-1];
    next_distance_address[i] = base_distance_address[i];
  }

  //Build code tables from code lengths
  //Separated as functions to allow both to execute simultaneously
  build_length_table(length_codelengths, length_code_table, next_length_address);
  build_distance_table(distance_codelengths, distance_code_table, next_distance_address);
}

ap_uint<2> full_huffman_decoder(
    stream<io_stream> &in_strm,
    stream<lld_stream> &out_strm
){
#pragma HLS INTERFACE ap_ctrl_hs register port=return
#pragma HLS INTERFACE axis off port=out_strm
#pragma HLS INTERFACE axis off port=in_strm
  ap_uint<64> acc = 0; //Accumulator for storing up to 32 bits at a time
  ap_uint<8> bit_count = 0; //Number of bits currently in the acc
  bool block_final = false; //Flag for BFINAL block header
  ap_uint<2> block_type; //Flag for BTYPE block header
  io_stream in_buff = {0}; //Buffer for reading input stream. It is passed to functions so they can check TLAST before reading from stream.
  bool end_of_block; //End-of-block flag
  ap_uint<16> block_length;
  ap_uint<16> block_Nlength;
  static ap_uint<4> length_codelengths[MAX_LEN_CODES] = {0}; //Array of code lengths for each Length/Literal symbol
  static ap_uint<4> distance_codelengths[MAX_DIS_CODES] = {0}; //Array of code lengths for each Distance symbol
  ap_uint<9> length_codelength_count[MAX_LEN_LENGTH+1]; //Array for counting number of times each code length occurs
#pragma HLS ARRAY_PARTITION variable=length_codelength_count complete dim=1
  ap_uint<9> distance_codelength_count[MAX_DIS_LENGTH+1]; //Array for counting number of times each code length occurs
#pragma HLS ARRAY_PARTITION variable=distance_codelength_count complete dim=1
  ap_uint<15> base_length_values[MAX_LEN_LENGTH+1]; //Calculated base value for each Length/Literal code length
#pragma HLS ARRAY_PARTITION variable=base_length_values complete dim=1
  ap_uint<15> base_distance_values[MAX_DIS_LENGTH+1]; //Calculated base value for each Distance code length
#pragma HLS ARRAY_PARTITION variable=base_distance_values complete dim=1
  ap_uint<9> base_length_address[MAX_LEN_LENGTH+1]; //Base address for each Length/Literal code length base value
#pragma HLS ARRAY_PARTITION variable=base_length_address complete dim=1
  ap_uint<8> base_distance_address[MAX_DIS_LENGTH+1]; //Base address for each Distance code length base value
#pragma HLS ARRAY_PARTITION variable=base_distance_address complete dim=1
  ap_uint<9> length_code_table[MAX_LEN_CODES]; //For looking up Length/Literal symbol of a code.
  ap_uint<8> distance_code_table[MAX_DIS_CODES]; //For looking up Distance symbol of a code.
  lld_stream out_buff = {0}; //Buffer for writing to output stream

  Main_Loop: while(!block_final){ //Continue processing Deflate blocks until final block is finished
    end_of_block = false;       //Reset end-of-block flag
    fetch_bits(3, in_strm, in_buff, bit_count, acc);
    block_final = acc[0];    //Retrieve BFINAL bit
    block_type = acc(2, 1); //Retrieve BTYPE bits
    acc >>= 3;       //Shift out bits from acc
    bit_count -= 3;    //Update bit count

    switch(block_type){
    case 0b00 : //Stored Block
      acc >>= (bit_count % 8);       //Discard 0 to 7 remaining bits to next byte boundary in acc
      bit_count -= (bit_count % 8); //Update bit count

      fetch_bits(32, in_strm, in_buff, bit_count, acc);
      block_length(15,8)  = acc(7,0); //Read upper byte of LEN
      block_length(7,0)   = acc(15,8); //Read lower byte of LEN
      block_Nlength(15,8) = acc(23,16); //Read upper byte of NLEN
      block_Nlength(7,0)  = acc(31,24); //Read lower byte of NLEN
      if(block_length != ~block_Nlength){ //Check if LEN is 1s
    compliment of NLEN
        return 1;
      }
      else{
        acc >>= 32;      //Shift out bits from acc
        bit_count -= 32;    //Update bit count
        Stream_Stored_Block_Loop: for(int i = 0; i < block_length
    -4; i+=4){ //Pass LEN number of bytes through
#pragma HLS PIPELINE
          in_strm >> in_buff;
          out_buff.data = in_buff.data;
          out_buff.keep = 0b1111;
          out_buff.user = 0;
          out_buff.last = 0;
          out_strm << out_buff;
        }
        fetch_bits(32, in_strm, in_buff, bit_count, acc);

        switch(block_length % 4){ //Write remaining amount of
    bytes on last iteration (1-4)
        case 0 : //4 remaining bytes
          out_buff.data = acc(31,0);
          out_buff.keep = 0b1111;
          acc >>= 32;    //Shift out bits from acc
          bit_count -= 32; //Update bit count
          break;
        case 1 : //1 remaining byte
          out_buff.data = acc(7,0);
          out_buff.keep = 0b0001;
          acc >>= 8;    //Shift out bits from acc
          bit_count -= 8; //Update bit count
          break;
        case 2 : //2 remaining bytes
          out_buff.data = acc(15,0);
          out_buff.keep = 0b0011;
          acc >>= 16;    //Shift out bits from acc
          bit_count -= 16; //Update bit count
          break;
        case 3 : //3 remaining bytes
          out_buff.data = acc(23,0);
          out_buff.keep = 0b0111;
          acc >>= 24;    //Shift out bits from acc
          bit_count -= 24; //Update bit count
          break;
        }
        out_buff.user = 0;
        out_buff.last = 0; //Assign TLAST
        out_strm << out_buff;
      }
      break;

    case 0b01 : //Static Compressed Block
      decode_static_block(in_strm, in_buff, bit_count, acc,
    end_of_block, out_strm);
      break;

    case 0b10 : //Dynamic Compressed Block
      build_dynamic_tables( in_strm, in_buff, bit_count, acc,
          length_codelengths,     distance_codelengths,
          length_codelength_count, distance_codelength_count,
          base_length_values,     base_distance_values,
          base_length_address,    base_distance_address,
          length_code_table,      distance_code_table);
      decode_dynamic_block( in_strm, in_buff, bit_count, acc,
    end_of_block, out_strm,
          length_codelength_count, distance_codelength_count,
          base_length_values,     base_distance_values,
          base_length_address,    base_distance_address,
          length_code_table,      distance_code_table
      );
      reset_length_codelengths(length_codelengths); //Reset code
    length arrays while dynamic decoding takes place
      reset_distance_codelengths(distance_codelengths);
      break;

    case 0b11 : //Reserved (Error)
      return 2;
      break;
    }
  } //After last block has been processed
  out_buff.keep = 0; //Lower TKEEP signal
  out_buff.last = 1; //Raise TLAST signal
  out_strm << out_buff;
  return 0;
}
```

Appendix G

Huffman Decoder Header File

Huffman_Decoder.h

```
#include <stdio.h>
#include <string.h>
#include <ap_int.h>
#include <hls_stream.h>
#include <assert.h>

using namespace hls;

#define MAX_LEN_CODES 286 //Max 286 Length/Literal code lengths
#define MIN_LEN_CODES 257 //Min 257 Length/Literal code lengths
    (256 Literals and 1 End_of_block code)
#define MAX_DIS_CODES 32  //Max 32 Distance code lengths
#define MAX_CL_CODES  19  //Max 19 Code Length code lengths
#define MAX_LEN_LENGTH 15 //Max length of Length/Literal code is
    15 bits
#define MAX_DIS_LENGTH 15 //Max length of Distance code is 15 bits
#define MAX_CL_LENGTH  7  //Max length of Code Length code is 7
    bits
#define STATIC_MAX_LEN_LENGTH 9 //Max length of Static Length/
    Literal code is 9 bits
#define STATIC_MIN_LEN_LENGTH 7 //Min length of Static Length/
    Literal code is 7 bits
#define STATIC_DIS_LENGTH 5 //Static Distance codes are fixed
    length 5-bit codes
#define MAX_LEN_EXTRA_BITS 5 //Max number of extra bits following
    a Length code
#define MAX_DIS_EXTRA_BITS 13 //Max number of extra bits following
     a Distance code

typedef struct{ //Input AXI-Stream structure
  ap_uint<32> data; //Four bytes of data
  ap_uint<4> keep;  //TKEEP Signals for each byte
  bool last;      //TLAST AXI-Stream signal
} io_stream;

typedef struct{ //Output (Literal, Length/Distance Pair) AXI-
    Stream structure
  ap_uint<32> data; //Four bytes of data
  ap_uint<4> keep;  //TKEEP Signals for each byte
  bool user;     //TUSER Signal to identify Literal(0) or Length-
    Distance pair(1)
  bool last;     //TLAST AXI-Stream signal
} lld_stream;

typedef struct{ //Structure for Length symbol table
  ap_uint<3> bits; //Number of extra bits following code. Can be
    from 0 to 5 for lengths.
  ap_uint<8> base; //Base length value - 3. Lengths 3-258 are
    encoded in 0-255
} length_symbol_values;

typedef struct{ //Structure for Distance symbol table
  ap_uint<4> bits; //Number of extra bits following code. Can be
    from 0 to 13 for distances.
  ap_uint<15> base; //Base distance value from 0 to 24577
} distance_symbol_values;

typedef struct{ //Structure for looking up static codes in tables
  ap_uint<8> type; //End-of-block (0110 0000), Literal (0000 0000)
    , or Length/Distance (0001 XXXX) (Where XXXX is number of
    extra bits)
  ap_uint<8> bits; //Number of bits in code
  ap_uint<16> base; //Base length/distance, or literal value
} static_symbol;
//The above structure was adapted from the 'code' structure used
    by zlib (defined in "inftrees.h")
```

Appendix H

Symbol Tables File

symbol_tables.cpp

```
//The permuted order found when reading Code Length code length
    sequence
const ap_uint<8> permuted_order[MAX_CL_CODES] =
    {16,17,18,0,8,7,9,6,10,5,11,4,12,3,13,2,14,1,15};

//Each index (symbol) contains that symbols base value and extra
    bits
//Structure: {Extra bits, Base Value-3}
const length_symbol_values length_symbol_table[29] = { //Length
    Symbols 257 to 285 are indexed here from 0 to 28
     /* 0*/  {0, 0},   {0, 1},   {0, 2},   {0, 3},
     /* 4*/  {0, 4},   {0, 5},   {0, 6},   {0, 7},
     /* 8*/  {1, 8},   {1, 10},  {1, 12},  {1, 14},
     /*12*/  {2, 16},  {2, 20},  {2, 24},  {2, 28},
     /*16*/  {3, 32},  {3, 40},  {3, 48},  {3, 56},
     /*20*/  {4, 64},  {4, 80},  {4, 96},  {4, 112},
     /*24*/  {5, 128}, {5, 160}, {5, 192}, {5, 224},
     /*28*/  {0, 255}
};

//Structure: {Extra bits, Base value}
const distance_symbol_values distance_symbol_table[30] = {
     /* 0*/  {0, 1},      {0, 2},      {0, 3},      {0, 4},
     /* 4*/  {1, 5},      {1, 7},      {2, 9},      {2, 13},
     /* 8*/  {3, 17},     {3, 25},     {4, 33},     {4, 49},
     /*12*/  {5, 65},     {5, 97},     {6, 129},    {6, 193},
     /*16*/  {7, 257},    {7, 385},    {8, 513},    {8, 769},
     /*20*/  {9, 1025},   {9, 1537},   {10, 2049},  {10, 3073},
     /*24*/  {11, 4097},  {11, 6145},  {12, 8193},  {12, 12289},
     /*28*/  {13, 16385}, {13, 24577}
};
```

Appendix I

Static Decoding Tables File

static_decoding_tables.cpp

```
//The following tables are from "inffixed.h" in the zlib library

static const static_symbol static_length_table[512] = {
    /*  0*/{96,7,0}, {0,8,80},
    {0,8,16},{20,8,115},{18,7,31},{0,8,112},{0,8,48},{0,9,192},
    /*  8*/{16,7,10},{0,8,96}, {0,8,32},{0,9,160}, {0,8,0},
    {0,8,128},{0,8,64},{0,9,224},
    /* 16*/{16,7,6}, {0,8,88}, {0,8,24},{0,9,144},
    {19,7,59},{0,8,120},{0,8,56},{0,9,208},
    /* 24*/{17,7,17},{0,8,104},{0,8,40},{0,9,176}, {0,8,8},
    {0,8,136},{0,8,72},{0,9,240},
    /* 32*/{16,7,4}, {0,8,84},
    {0,8,20},{21,8,227},{19,7,43},{0,8,116},{0,8,52},{0,9,200},
    /* 40*/{17,7,13},{0,8,100},{0,8,36},{0,9,168}, {0,8,4},
    {0,8,132},{0,8,68},{0,9,232},
    /* 48*/{16,7,8}, {0,8,92}, {0,8,28},{0,9,152},
    {20,7,83},{0,8,124},{0,8,60},{0,9,216},
    /* 56*/{18,7,23},{0,8,108},{0,8,44},{0,9,184}, {0,8,12},
    {0,8,140},{0,8,76},{0,9,248},
    /* 64*/{16,7,3}, {0,8,82},
    {0,8,18},{21,8,163},{19,7,35},{0,8,114},{0,8,50},{0,9,196},
    /* 72*/{17,7,11},{0,8,98}, {0,8,34},{0,9,164}, {0,8,2},
    {0,8,130},{0,8,66},{0,9,228},
    /* 80*/{16,7,7}, {0,8,90}, {0,8,26},{0,9,148},
    {20,7,67},{0,8,122},{0,8,58},{0,9,212},
    /* 88*/{18,7,19},{0,8,106},{0,8,42},{0,9,180}, {0,8,10},
    {0,8,138},{0,8,74},{0,9,244},
    /* 96*/{16,7,5}, {0,8,86}, {0,8,22},{64,8,0},
    {19,7,51},{0,8,118},{0,8,54},{0,9,204},
    /*104*/{17,7,15},{0,8,102},{0,8,38},{0,9,172}, {0,8,6},
    {0,8,134},{0,8,70},{0,9,236},
    /*112*/{16,7,9}, {0,8,94}, {0,8,30},{0,9,156},
    {20,7,99},{0,8,126},{0,8,62},{0,9,220},
    /*120*/{18,7,27},{0,8,110},{0,8,46},{0,9,188}, {0,8,14},
    {0,8,142},{0,8,78},{0,9,252},
    /*128*/{96,7,0}, {0,8,81},
    {0,8,17},{21,8,131},{18,7,31},{0,8,113},{0,8,49},{0,9,194},
    /*136*/{16,7,10},{0,8,97}, {0,8,33},{0,9,162}, {0,8,1},
    {0,8,129},{0,8,65},{0,9,226},
    /*144*/{16,7,6}, {0,8,89}, {0,8,25},{0,9,146},
    {19,7,59},{0,8,121},{0,8,57},{0,9,210},
    /*152*/{17,7,17},{0,8,105},{0,8,41},{0,9,178}, {0,8,9},
    {0,8,137},{0,8,73},{0,9,242},
    /*160*/{16,7,4}, {0,8,85},
    {0,8,21},{16,8,258},{19,7,43},{0,8,117},{0,8,53},{0,9,202},
    /*168*/{17,7,13},{0,8,101},{0,8,37},{0,9,170}, {0,8,5},
    {0,8,133},{0,8,69},{0,9,234},
    /*176*/{16,7,8}, {0,8,93}, {0,8,29},{0,9,154},
    {20,7,83},{0,8,125},{0,8,61},{0,9,218},
    /*184*/{18,7,23},{0,8,109},{0,8,45},{0,9,186}, {0,8,13},
    {0,8,141},{0,8,77},{0,9,250},
    /*192*/{16,7,3}, {0,8,83},
    {0,8,19},{21,8,195},{19,7,35},{0,8,115},{0,8,51},{0,9,198},
    /*200*/{17,7,11},{0,8,99}, {0,8,35},{0,9,166}, {0,8,3},
    {0,8,131},{0,8,67},{0,9,230},
    /*208*/{16,7,7}, {0,8,91}, {0,8,27},{0,9,150},
    {20,7,67},{0,8,123},{0,8,59},{0,9,214},
    /*216*/{18,7,19},{0,8,107},{0,8,43},{0,9,182}, {0,8,11},
    {0,8,139},{0,8,75},{0,9,246},
    /*224*/{16,7,5}, {0,8,87}, {0,8,23},{64,8,0},
    {19,7,51},{0,8,119},{0,8,55},{0,9,206},
    /*232*/{17,7,15},{0,8,103},{0,8,39},{0,9,174}, {0,8,7},
    {0,8,135},{0,8,71},{0,9,238},
    /*240*/{16,7,9}, {0,8,95}, {0,8,31},{0,9,158},
    {20,7,99},{0,8,127},{0,8,63},{0,9,222},
    /*248*/{18,7,27},{0,8,111},{0,8,47},{0,9,190}, {0,8,15},
    {0,8,143},{0,8,79},{0,9,254},
    /*256*/{96,7,0}, {0,8,80},
    {0,8,16},{20,8,115},{18,7,31},{0,8,112},{0,8,48},{0,9,193},
    /*264*/{16,7,10},{0,8,96}, {0,8,32},{0,9,161}, {0,8,0},
    {0,8,128},{0,8,64},{0,9,225},
    /*272*/{16,7,6}, {0,8,88}, {0,8,24},{0,9,145},
    {19,7,59},{0,8,120},{0,8,56},{0,9,209},
    /*280*/{17,7,17},{0,8,104},{0,8,40},{0,9,177}, {0,8,8},
    {0,8,136},{0,8,72},{0,9,241},
    /*288*/{16,7,4}, {0,8,84},
    {0,8,20},{21,8,227},{19,7,43},{0,8,116},{0,8,52},{0,9,201},
    /*296*/{17,7,13},{0,8,100},{0,8,36},{0,9,169}, {0,8,4},
    {0,8,132},{0,8,68},{0,9,233},
    /*304*/{16,7,8}, {0,8,92}, {0,8,28},{0,9,153},
    {20,7,83},{0,8,124},{0,8,60},{0,9,217},
    /*312*/{18,7,23},{0,8,108},{0,8,44},{0,9,185}, {0,8,12},
    {0,8,140},{0,8,76},{0,9,249},
    /*320*/{16,7,3}, {0,8,82},
    {0,8,18},{21,8,163},{19,7,35},{0,8,114},{0,8,50},{0,9,197},
    /*328*/{17,7,11},{0,8,98}, {0,8,34},{0,9,165}, {0,8,2},
    {0,8,130},{0,8,66},{0,9,229},
    /*336*/{16,7,7}, {0,8,90}, {0,8,26},{0,9,149},
    {20,7,67},{0,8,122},{0,8,58},{0,9,213},
    /*344*/{18,7,19},{0,8,106},{0,8,42},{0,9,181}, {0,8,10},
    {0,8,138},{0,8,74},{0,9,245},
    /*352*/{16,7,5}, {0,8,86}, {0,8,22},{64,8,0},
    {19,7,51},{0,8,118},{0,8,54},{0,9,205},
    /*360*/{17,7,15},{0,8,102},{0,8,38},{0,9,173}, {0,8,6},
    {0,8,134},{0,8,70},{0,9,237},
    /*368*/{16,7,9}, {0,8,94}, {0,8,30},{0,9,157},
    {20,7,99},{0,8,126},{0,8,62},{0,9,221},
    /*376*/{18,7,27},{0,8,110},{0,8,46},{0,9,189}, {0,8,14},
    {0,8,142},{0,8,78},{0,9,253},
    /*384*/{96,7,0}, {0,8,81},
    {0,8,17},{21,8,131},{18,7,31},{0,8,113},{0,8,49},{0,9,195},
    /*392*/{16,7,10},{0,8,97}, {0,8,33},{0,9,163}, {0,8,1},
    {0,8,129},{0,8,65},{0,9,227},
    /*400*/{16,7,6}, {0,8,89}, {0,8,25},{0,9,147},
    {19,7,59},{0,8,121},{0,8,57},{0,9,211},
    /*408*/{17,7,17},{0,8,105},{0,8,41},{0,9,179}, {0,8,9},
    {0,8,137},{0,8,73},{0,9,243},
    /*416*/{16,7,4}, {0,8,85},
    {0,8,21},{16,8,258},{19,7,43},{0,8,117},{0,8,53},{0,9,203},
    /*424*/{17,7,13},{0,8,101},{0,8,37},{0,9,171}, {0,8,5},
    {0,8,133},{0,8,69},{0,9,235},
    /*432*/{16,7,8}, {0,8,93}, {0,8,29},{0,9,155},
    {20,7,83},{0,8,125},{0,8,61},{0,9,219},
    /*440*/{18,7,23},{0,8,109},{0,8,45},{0,9,187}, {0,8,13},
    {0,8,141},{0,8,77},{0,9,251},
    /*448*/{16,7,3}, {0,8,83},
    {0,8,19},{21,8,195},{19,7,35},{0,8,115},{0,8,51},{0,9,199},
    /*456*/{17,7,11},{0,8,99}, {0,8,35},{0,9,167}, {0,8,3},
    {0,8,131},{0,8,67},{0,9,231},
    /*464*/{16,7,7}, {0,8,91}, {0,8,27},{0,9,151},
    {20,7,67},{0,8,123},{0,8,59},{0,9,215},
    /*472*/{18,7,19},{0,8,107},{0,8,43},{0,9,183}, {0,8,11},
    {0,8,139},{0,8,75},{0,9,247},
    /*480*/{16,7,5}, {0,8,87}, {0,8,23},{64,8,0},
    {19,7,51},{0,8,119},{0,8,55},{0,9,207},
    /*488*/{17,7,15},{0,8,103},{0,8,39},{0,9,175}, {0,8,7},
    {0,8,135},{0,8,71},{0,9,239},
    /*496*/{16,7,9}, {0,8,95}, {0,8,31},{0,9,159},
    {20,7,99},{0,8,127},{0,8,63},{0,9,223},
    /*504*/{18,7,27},{0,8,111},{0,8,47},{0,9,191}, {0,8,15},
    {0,8,143},{0,8,79},{0,9,255}
};
static const static_symbol static_distance_table[32] = {
    /* 0*/{16,5,1},{23,5,257},{19,5,17},{27,5,4097}, {17,5,5},
    {25,5,1025},{21,5,65}, {29,5,16385},
    /* 8*/{16,5,3},{24,5,513},{20,5,33},{28,5,8193}, {18,5,9},
    {26,5,2049},{22,5,129},{64,5,0},
    /*16*/{16,5,2},{23,5,385},{19,5,25},{27,5,6145}, {17,5,7},
    {25,5,1537},{21,5,97}, {29,5,24577},
    /*24*/{16,5,4},{24,5,769},{20,5,49},{28,5,12289},{18,5,13},
    {26,5,3073},{22,5,193},{64,5,0}
};
```

Appendix J

Literal Stacker Source File

literal_stacker.cpp

```
#include "literal_stacker.h"

void literal_stacker(
    stream<lld_stream> &in_strm,
    stream<lld_stream> &out_strm
){
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS PIPELINE enable_flush
#pragma HLS INTERFACE axis off port=out_strm //For individual core
     synthesis
#pragma HLS INTERFACE axis off port=in_strm
  static lld_stream in_buff;
  static lld_stream data_regA; //Data register to be filled with
    literals
  static ap_uint<2> state = 0;

  if(state == 0){ //Length-Distance State: Continue streaming LD
    pairs through until Literals are encountered
    in_strm >> in_buff;
    if(in_buff.last == 1){ //If last pack (No data)
      out_strm << in_buff; //Stream out data pack with TLAST high
    }
    else{
      if(in_buff.user == 1){ //If data is Length-Distance pair
        out_strm << in_buff; //Stream out LD pair
      }
      else{ //If data is composed of literals
        if(in_buff.keep == 0b1111){ //If no literals needed
          out_strm << in_buff; //Stream out full data pack
        }
        else{ //Literals needed to fill this data pack
          data_regA = in_buff; //Store in RegA
          state = 1; //Change to literal state
        }
      }
    }
  }
  else if(state == 1){ //Literal State: Stack literals with more literals from the stream. Remain here until LD pair read
    in_strm >> in_buff; //Stream in another data pack
    if(in_buff.last == 1){ //If last pack (No data)
      out_strm << data_regA; //Flush out datapack in RegA
      state = 2; //Go to flush state
    }
    else{
      if(in_buff.user == 1){ //If data is Length-Distance pair
        out_strm << data_regA; //Flush out datapack in RegA
        state = 2; //Go to flush state
      }
      else{ //If data is composed of literals
        if(data_regA.keep == 0b1111){ //If no literals needed (Should never happen)
          out_strm << data_regA; //Stream out full data pack
          data_regA = in_buff; //Move new data to regA
        }
        else if(data_regA.keep == 0b0111){ //1 literal needed by RegA
          if(in_buff.keep == 0b0001){ //If number of literals in in_buff is exactly 1
            data_regA.data(31,24) = in_buff.data(7,0); //Take 1 literal from it
            data_regA.keep = 0b1111;           //Update RegA TKEEP signal
            out_strm << data_regA;             //Stream out full data pack
            state = 0;                   //Return to LD state
          }
          else if(in_buff.keep == 0b0000){ //If data pack is empty (Should never happen)
            //Do nothing
          }
          else{ //If number of literals is from 2 to 4 (more than needed)
            data_regA.data(31,24) = in_buff.data(7,0); //Take 1 literal from it
            data_regA.keep = 0b1111;           //Update RegA TKEEP signal
            out_strm << data_regA;             //Stream out full data pack
            data_regA.data = in_buff.data >> 8;    //Shift in_buff data and store in RegA
            data_regA.keep = in_buff.keep >> 1;    //Shift in_buff TKEEP signal and store in RegA
          }
        }
        else if(data_regA.keep == 0b0011){ //2 literals needed by RegA
          if(in_buff.keep == 0b0011){ //If number of literals in in_buff is exactly 2
            data_regA.data(31,16) = in_buff.data(15,0); //Take 2 literals
```

```
                data_regA.keep = 0b1111; //Update RegA TKEEP signal
                out_strm << data_regA; //Stream out full data pack
                state = 0;  //Return to LD state
            }
            else if(in_buff.keep == 0b0001){ //If only 1 literal in in_buff (less than needed)
                data_regA.data(23,16) = in_buff.data(7,0); //Take 1 literal
                data_regA.keep = 0b0111; //Update TKEEP
            }
            else if(in_buff.keep == 0b0000){ //If data pack is empty (Should never happen)
                //Do nothing
            }
            else{ //If number of literals is 3 or 4 (more than needed)
                data_regA.data(31,16) = in_buff.data(15,0); //Take 2 literals
                data_regA.keep = 0b1111; //Update RegA TKEEP signal
                out_strm << data_regA; //Stream out full data pack
                data_regA.data = in_buff.data >> 16; //Shift in_buff data and store in RegA
                data_regA.keep = in_buff.keep >> 2;  //Shift in_buff TKEEP signal and store in RegA
            }
        }
        else if(data_regA.keep == 0b0001){ //3 literals needed by RegA
            if(in_buff.keep == 0b0111){ //If number of literals in in_buff is exactly 3
                data_regA.data(31,8) = in_buff.data(23,0); //Take 3 literals
                data_regA.keep = 0b1111;
                out_strm << data_regA;
                state = 0;  //Return to LD state
            }
            else if(in_buff.keep == 0b0011){ //If only 2 literals in in_buff (less than needed)
                data_regA.data(23,8) = in_buff.data(15,0); //Take 2 literals
                data_regA.keep = 0b0111; //Update TKEEP
            }
            else if(in_buff.keep == 0b0001){ //If only 1 literal in in_buff (less than needed)
                data_regA.data(15,8) = in_buff.data(7,0); //Take 1 literal
                data_regA.keep = 0b0011; //Update TKEEP
            }
            else if(in_buff.keep == 0b0000){ //If data pack is empty (Should never happen)
                //Do nothing
            }
            else{ //If number of literals is 4 (more than needed)
            data_regA.data(31,8) = in_buff.data(23,0); //Take 3 literals
            data_regA.keep = 0b1111; //Update RegA TKEEP signal
            out_strm << data_regA; //Stream out full data pack
            data_regA.data = in_buff.data >> 24; //Shift in_buff data and store in RegA
            data_regA.keep = in_buff.keep >> 3; //Shift in_buff TKEEP signal and store in RegA
          }
        }
        else if(data_regA.keep == 0b0000){ //RegA is empty (Should never happen)
          data_regA = in_buff; //Move new data to regA
        }
      }
    }
  }
  else if(state == 2){ //Flush state (Needed to space out writes to output stream)
    out_strm << in_buff; //Stream out LD pair
    state = 0; //Return to LD state
  }
}
```

Appendix K

Literal Stacker Header File

literal_stacker.h

```
#include <stdio.h>
#include <string.h>
#include <ap_int.h>
#include <hls_stream.h>
#include <assert.h>

using namespace hls;

//Stream Format:
//TDATA: | 31 - Byte3 - 24 | 23 - Byte2 - 16 | 15 - Byte1 - 8 | 7
    - Byte0 - 0 |
//TKEEP: |      TKEEP3      |     TKEEP2    |      TKEEP1     |
    TKEEP0     |
typedef struct{ //Literal, Length/Distance stream structure
  ap_uint<32> data; //Four bytes of data
  ap_uint<4> keep;  //TKEEP Signals for each byte
  bool user;     //TUSER Signal to identify Literal(0) or Length-
    Distance pair(1)
  bool last;     //TLAST AXI-Stream signal
} lld_stream;
```

Appendix L

LZ77 Decoder Source File

LZ77_Decoder.cpp

```
#include "LZ77_Decoder.h"

#define BUFF_SIZE 32768

void read_cb(
    ap_uint<3> n, //Number of bytes to read (1-4)
    ap_uint<2> read_config, //Read configuration
    ap_uint<15> address0,
    ap_uint<15> address1,
    ap_uint<15> address2,
    ap_uint<15> address3,
    ap_uint<8> &literal0, //Output Literals
    ap_uint<8> &literal1,
    ap_uint<8> &literal2,
    ap_uint<8> &literal3,
    ap_uint<8> (&circ_buff0)[BUFF_SIZE/4],
    ap_uint<8> (&circ_buff1)[BUFF_SIZE/4],
    ap_uint<8> (&circ_buff2)[BUFF_SIZE/4],
    ap_uint<8> (&circ_buff3)[BUFF_SIZE/4]
){
#pragma HLS INLINE
  switch(n){
  case 4 :{
    switch(read_config){
    case 0 :
      literal0 = circ_buff0[address0]; //Read 4 literals from buffer
      literal1 = circ_buff1[address1];
      literal2 = circ_buff2[address2];
      literal3 = circ_buff3[address3];
      break;
    case 1 :
      literal0 = circ_buff1[address0];
      literal1 = circ_buff2[address1];
      literal2 = circ_buff3[address2];
      literal3 = circ_buff0[address3];
      break;
    case 2 :
      literal0 = circ_buff2[address0];
      literal1 = circ_buff3[address1];
      literal2 = circ_buff0[address2];
      literal3 = circ_buff1[address3];
      break;
    case 3 :
      literal0 = circ_buff3[address0];
      literal1 = circ_buff0[address1];
      literal2 = circ_buff1[address2];
      literal3 = circ_buff2[address3];
      break;
    }
    break;}
  case 3 :{
    switch(read_config){
    case 0 :
      literal0 = circ_buff0[address0]; //Read 3 literals from
     buffer
      literal1 = circ_buff1[address1];
      literal2 = circ_buff2[address2];
      break;
    case 1 :
      literal0 = circ_buff1[address0];
      literal1 = circ_buff2[address1];
      literal2 = circ_buff3[address2];
      break;
    case 2 :
      literal0 = circ_buff2[address0];
      literal1 = circ_buff3[address1];
      literal2 = circ_buff0[address2];
      break;
    case 3 :
      literal0 = circ_buff3[address0];
      literal1 = circ_buff0[address1];
      literal2 = circ_buff1[address2];
      break;
    }
    break;}
  case 2 :{
    switch(read_config){
    case 0 :
      literal0 = circ_buff0[address0];
      literal1 = circ_buff1[address1];
      break;
    case 1 :
      literal0 = circ_buff1[address0];
      literal1 = circ_buff2[address1];
      break;
    case 2 :
      literal0 = circ_buff2[address0];
      literal1 = circ_buff3[address1];
      break;
    case 3 :
      literal0 = circ_buff3[address0];
      literal1 = circ_buff0[address1];
      break;
    }
    break;}
  case 1 :{
    switch(read_config){
    case 0 : literal0 = circ_buff0[address0]; break;
    case 1 : literal0 = circ_buff1[address0]; break;
    case 2 : literal0 = circ_buff2[address0]; break;
    case 3 : literal0 = circ_buff3[address0]; break;
    }
    break;}
  }
}

void write_cb(
    ap_uint<3> n, //Number of bytes to write (1-4)
    ap_uint<2> write_config, //Write configuration
    ap_uint<15> address0,
    ap_uint<15> address1,
    ap_uint<15> address2,
    ap_uint<15> address3,
    ap_uint<8> literal0, //Input Literals
    ap_uint<8> literal1,
    ap_uint<8> literal2,
    ap_uint<8> literal3,
    ap_uint<8> (&circ_buff0)[BUFF_SIZE/4],
    ap_uint<8> (&circ_buff1)[BUFF_SIZE/4],
    ap_uint<8> (&circ_buff2)[BUFF_SIZE/4],
    ap_uint<8> (&circ_buff3)[BUFF_SIZE/4]
){
#pragma HLS INLINE
  switch(n){
  case 4 :{
    switch(write_config){
    case 0 :
      circ_buff0[address0] = literal0; //Print 4 literals to
   buffer head
      circ_buff1[address1] = literal1;
      circ_buff2[address2] = literal2;
      circ_buff3[address3] = literal3;
      break;
    case 1 :
      circ_buff1[address0] = literal0;
      circ_buff2[address1] = literal1;
      circ_buff3[address2] = literal2;
      circ_buff0[address3] = literal3;
      break;
    case 2 :
      circ_buff2[address0] = literal0;
      circ_buff3[address1] = literal1;
      circ_buff0[address2] = literal2;
      circ_buff1[address3] = literal3;
      break;
    case 3 :
      circ_buff3[address0] = literal0;
      circ_buff0[address1] = literal1;
      circ_buff1[address2] = literal2;
      circ_buff2[address3] = literal3;
      break;
    }
    break;}
  case 3 :{
    switch(write_config){
    case 0 :
      circ_buff0[address0] = literal0;
      circ_buff1[address1] = literal1;
      circ_buff2[address2] = literal2;
      break;
    case 1 :
      circ_buff1[address0] = literal0;
      circ_buff2[address1] = literal1;
      circ_buff3[address2] = literal2;
      break;
    case 2 :
      circ_buff2[address0] = literal0;
      circ_buff3[address1] = literal1;
      circ_buff0[address2] = literal2;
      break;
    case 3 :
      circ_buff3[address0] = literal0;
      circ_buff0[address1] = literal1;
      circ_buff1[address2] = literal2;
      break;
    }
    break;}
  case 2 :{
    switch(write_config){
    case 0 :
      circ_buff0[address0] = literal0;
      circ_buff1[address1] = literal1;
      break;
    case 1 :
      circ_buff1[address0] = literal0;
      circ_buff2[address1] = literal1;
      break;
    case 2 :
      circ_buff2[address0] = literal0;
      circ_buff3[address1] = literal1;
      break;
    case 3 :
      circ_buff3[address0] = literal0;
      circ_buff0[address1] = literal1;
      break;
    }
    break;}
  case 1 :{
    switch(write_config){
    case 0 : circ_buff0[address0] = literal0; break;
    case 1 : circ_buff1[address0] = literal0; break;
    case 2 : circ_buff2[address0] = literal0; break;
    case 3 : circ_buff3[address0] = literal0; break;
    }
    break;}
  }
}

//LZ77 Decoder: Takes in LZ77 commands and outputs decompressed
    data (4-byte version)
void LZ77_Decoder(
    stream<lld_stream> &in_strm,
    stream<io_stream> &out_strm
){
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS INTERFACE axis off port=in_strm //Registers off
#pragma HLS INTERFACE axis off port=out_strm
  lld_stream in_buff;
  lld_stream literal_buffer;
  io_stream out_buff = {0};
  ap_uint<3> state = 0;
  ap_uint<9> length;
  ap_uint<16> distance;
  ap_uint<8> circ_buff0[BUFF_SIZE/4]; //32 kiB cicular buffer
    split into 4 cyclical partitions
  ap_uint<8> circ_buff1[BUFF_SIZE/4];
  ap_uint<8> circ_buff2[BUFF_SIZE/4];
  ap_uint<8> circ_buff3[BUFF_SIZE/4];
  ap_uint<2> read_config; //Points to the circ_buff partition that
    literal0 will be read from
  ap_uint<2> write_config = 0; //Points to the circ_buff partition
    that literal0 will be written to
  ap_uint<8> copy_array[3][4];
#pragma HLS ARRAY_PARTITION variable=copy_array complete dim=0
  ap_uint<2> copy_pointer;
  ap_uint<15> write_addr0 = 0;
  ap_uint<15> write_addr1 = 1;
  ap_uint<15> write_addr2 = 2;
  ap_uint<15> write_addr3 = 3;
  ap_uint<15> read_addr0;
  ap_uint<15> read_addr1;
  ap_uint<15> read_addr2;
  ap_uint<15> read_addr3;
  ap_uint<15> first_read_addr;
  ap_uint<9> length_remaining; //Length_remaining is the total
    number of bytes to be written
  ap_uint<9> distance_remaining; //Distance_remaining is the
    number of bytes we can write before the sliding window must be
    reset
  ap_uint<8> literal0;
  ap_uint<8> literal1;
  ap_uint<8> literal2;
  ap_uint<8> literal3;

  Main_Loop: while(1){
#pragma HLS PIPELINE
#pragma HLS DEPENDENCE variable=circ_buff0 inter false
#pragma HLS DEPENDENCE variable=circ_buff1 inter false
#pragma HLS DEPENDENCE variable=circ_buff2 inter false
#pragma HLS DEPENDENCE variable=circ_buff3 inter false
    if(state == 0){ //No Literal Held State
      in_buff = in_strm.read();
      if(in_buff.last){
        break;
      }
      else{
        if(in_buff.user == 0){ //If data is a literal
          literal_buffer = in_buff;
          state = 1;
        }
        else{ //Length/Distance pair
          length = in_buff.data(31,23);
          distance = in_buff.data(15,0);
          assert(length >= 3); //DEFLATE min allowed length is 3
          assert(length <= 258); //DEFLATE max allowed length is
    258
          assert(distance >= 1); //DEFLATE min allowed distance is
     1
          assert(distance <= 32768); //DEFLATE max allowed
    distance is 32768
          read_addr0 = write_addr0 - distance;
          read_addr1 = read_addr0 + 1;
          read_addr2 = read_addr0 + 2;
          read_addr3 = read_addr0 + 3;
          read_config = read_addr0 & 3; //Modulo the read_address
    to find which buffer partition is read first
          first_read_addr = read_addr0;
          length_remaining = length;
          if(distance <= 4){ //If Distance is 1, 2, 3, or 4
            state = 3;
            distance_remaining = length; //Distance remaining
    equals length, no wrap-around
          }
          else if(distance <= 7){ //If Distance is 5, 6, or 7
            state = 2;
            distance_remaining = distance; //Distance remaining
    before wrap-around is distance
          }
          else{ //If Distance is 8 or greater
            state = 2;
            distance_remaining = length;
          }
        }
      }
    }
    else if(state == 1){ //Have_Literal State
      switch(literal_buffer.keep){
      case 0b0001 :
        write_cb(1, write_config, write_addr0/4, 0, 0, 0, literal_buffer.data(7,0), 0, 0, 0, circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        write_config++;
        write_addr0++; write_addr1++; write_addr2++; write_addr3++;
        break;
      case 0b0011 :
        write_cb(2,write_config,write_addr0/4,write_addr1/4,0,0,literal_buffer.data(7,0),literal_buffer.data(15,8),
            0,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        write_config += 2;
        write_addr0 += 2; write_addr1 += 2; write_addr2 += 2; write_addr3 += 2;
        break;
      case 0b0111 :
        write_cb(3,write_config,write_addr0/4,write_addr1/4,write_addr2/4,0,literal_buffer.data(7,0),literal_buffer.data(15,8),
            literal_buffer.data(23,16),0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        write_config += 3;
        write_addr0 += 3; write_addr1 += 3; write_addr2 += 3; write_addr3 += 3;
        break;
      case 0b1111 :
        write_cb(4,write_config,write_addr0/4,write_addr1/4,write_addr2/4,write_addr3/4,literal_buffer.data(7,0),literal_buffer.data(15,8),
            literal_buffer.data(23,16),literal_buffer.data(31,24),circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        write_addr0 += 4; write_addr1 += 4; write_addr2 += 4; write_addr3 += 4;
        break;
      }
      out_buff.data = literal_buffer.data;
      out_buff.keep = literal_buffer.keep;
      out_strm << out_buff;
      in_buff = in_strm.read();
      if(in_buff.last){
        break;
      }
      else{
        if(in_buff.user == 0){ //If data is a literal
          literal_buffer = in_buff;
        }
        else{ //Length/Distance pair
          length = in_buff.data(31,23);
          distance = in_buff.data(15,0);
          assert(length >= 3); //DEFLATE min allowed length is 3
          assert(length <= 258); //DEFLATE max allowed length is 258
```

```
          assert(distance >= 1); //DEFLATE min allowed distance is 1
          assert(distance <= 32768); //DEFLATE max allowed distance is 32768
          read_addr0 = write_addr0 - distance;
          read_addr1 = read_addr0 + 1;
          read_addr2 = read_addr0 + 2;
          read_addr3 = read_addr0 + 3;
          read_config = read_addr0 & 3; //Modulo the read_address to find which buffer partition is read first
          first_read_addr = read_addr0;
          length_remaining = length;
          if(distance <= 4){ //If Distance is 1, 2, 3, or 4
            state = 3;
            distance_remaining = length; //Distance remaining equals length, no wrap-around
          }
          else if(distance <= 7){ //If Distance is 5, 6, or 7
            state = 2;
            distance_remaining = distance; //Distance remaining before wrap-around is distance
          }
          else{ //If Distance is 8 or greater
            state = 2;
            distance_remaining = length;
          }
        }
      }
    }
    else if(state == 2){ //Regular Length-Distance State
      if(length_remaining >= 4){
        if(distance_remaining >= 4){
          read_cb(4,read_config,read_addr0/4,read_addr1/4,read_addr2/4,read_addr3/4,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
          if(distance != 32768){
            write_cb(4,write_config,write_addr0/4,write_addr1/4,write_addr2/4,write_addr3/4,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
          }
          out_buff.data(7,0)   = literal0; //Write 4 literals to output buffer
          out_buff.data(15,8)  = literal1;
          out_buff.data(23,16) = literal2;
          out_buff.data(31,24) = literal3;
          write_addr0 += 4; write_addr1 += 4; write_addr2 += 4; write_addr3 += 4;
          out_buff.keep = 0b1111;
          if(length_remaining == 4){ //If only 4 bytes were left
            state = 0; //Return to Literal state
          }
          else{
            length_remaining -= 4;
            if(distance_remaining == 4){
              distance_remaining = distance; //Reset sliding
    window
              read_addr0 = first_read_addr; //Shift addresses back
     to start
              read_addr1 = first_read_addr + 1;
              read_addr2 = first_read_addr + 2;
              read_addr3 = first_read_addr + 3;
            }
            else{
              read_addr0 += 4; read_addr1 += 4; read_addr2 += 4;
    read_addr3 += 4;
              distance_remaining -= 4;
            }
          }
        }
        else if(distance_remaining == 3){
          read_cb(3,read_config,read_addr0/4,read_addr1/4,
    read_addr2/4,0,literal0,literal1,literal2,literal3,circ_buff0,
    circ_buff1,circ_buff2,circ_buff3);
          if(distance != 32768){
            write_cb(3,write_config,write_addr0/4,write_addr1/4,
    write_addr2/4,0,literal0,literal1,literal2,0,circ_buff0,
    circ_buff1,circ_buff2,circ_buff3);
          }
          write_config += 3;
          write_addr0 += 3; write_addr1 += 3; write_addr2 += 3;
    write_addr3 += 3;
          out_buff.data(7,0)   = literal0; //Write 3 literals to
    output buffer
          out_buff.data(15,8)  = literal1;
          out_buff.data(23,16) = literal2;
          out_buff.keep = 0b0111;
          length_remaining -= 3;
          distance_remaining = distance; //Reset sliding window
          read_addr0 = first_read_addr; // Shift addresses back to
     start
          read_addr1 = first_read_addr + 1;
          read_addr2 = first_read_addr + 2;
          read_addr3 = first_read_addr + 3;
        }
        else if(distance_remaining == 2){
          read_cb(2,read_config,read_addr0/4,read_addr1/4,0,0,
    literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,
    circ_buff2,circ_buff3);
          if(distance != 32768){
            write_cb(2,write_config,write_addr0/4,write_addr1
    /4,0,0,literal0,literal1,0,0,circ_buff0,circ_buff1,circ_buff2,
    circ_buff3);
          }
          write_config += 2;
          write_addr0 += 2; write_addr1 += 2; write_addr2 += 2;
    write_addr3 += 2;
          out_buff.data(7,0)   = literal0; //Write 2 literals to
    output buffer
```

```
        out_buff.data(15,8)  = literal1;
        out_buff.keep = 0b0011;
        length_remaining -= 2;
        distance_remaining = distance; //Reset sliding window
        read_addr0 = first_read_addr; // Shift addresses back to start
        read_addr1 = first_read_addr + 1;
        read_addr2 = first_read_addr + 2;
        read_addr3 = first_read_addr + 3;
      }
      else if(distance_remaining == 1){
        read_cb(1,read_config,read_addr0/4,0,0,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        if(distance != 32768){
          write_cb(1,write_config,write_addr0/4,0,0,0,literal0,0,0,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        }
        write_config++;
        write_addr0++; write_addr1++; write_addr2++; write_addr3++;
        out_buff.data(7,0)   = literal0; //Read literal from buffer
        out_buff.keep = 0b0001;
        length_remaining -= 1;
        distance_remaining = distance; //Reset sliding window
        read_addr0 = first_read_addr; // Shift addresses back to start
        read_addr1 = first_read_addr + 1;
        read_addr2 = first_read_addr + 2;
        read_addr3 = first_read_addr + 3;
      }
    }
    else if(length_remaining == 3){
      if(distance_remaining >= 3){
        read_cb(3,read_config,read_addr0/4,read_addr1/4,read_addr2/4,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        if(distance != 32768){
          write_cb(3,write_config,write_addr0/4,write_addr1/4,write_addr2/4,0,literal0,literal1,literal2,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        }
        out_buff.data(7,0)   = literal0; //Write 3 literals to output buffer
        out_buff.data(15,8)  = literal1;
        out_buff.data(23,16) = literal2;
        write_config += 3;
        write_addr0 += 3; write_addr1 += 3; write_addr2 += 3; write_addr3 += 3;
        out_buff.keep = 0b0111;
        state = 0; //Return to Literal state
      }
      else if(distance_remaining == 2){
        read_cb(2,read_config,read_addr0/4,read_addr1/4,0,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        if(distance != 32768){
          write_cb(2,write_config,write_addr0/4,write_addr1/4,0,0,literal0,literal1,0,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        }
        write_config += 2;
        write_addr0 += 2; write_addr1 += 2; write_addr2 += 2; write_addr3 += 2;
        out_buff.data(7,0)   = literal0; //Write 2 literals to output buffer
        out_buff.data(15,8)  = literal1;
        out_buff.keep = 0b0011;
        length_remaining -= 2;
        distance_remaining = distance; //Reset sliding window
        read_addr0 = first_read_addr; // Shift addresses back to start
        read_addr1 = first_read_addr + 1;
        read_addr2 = first_read_addr + 2;
        read_addr3 = first_read_addr + 3;
      }
      else if(distance_remaining == 1){
        read_cb(1,read_config,read_addr0/4,0,0,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        if(distance != 32768){
          write_cb(1,write_config,write_addr0/4,0,0,0,literal0,0,0,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        }
        write_config++;
        write_addr0++; write_addr1++; write_addr2++; write_addr3++;
        out_buff.data(7,0)   = literal0; //Read literal from buffer
        out_buff.keep = 0b0001;
        length_remaining -= 1;
        distance_remaining = distance; //Reset sliding window
        read_addr0 = first_read_addr; // Shift addresses back to start
        read_addr1 = first_read_addr + 1;
        read_addr2 = first_read_addr + 2;
        read_addr3 = first_read_addr + 3;
      }
    }
    else if(length_remaining == 2){
      if(distance_remaining >= 2){
        read_cb(2,read_config,read_addr0/4,read_addr1/4,0,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        if(distance != 32768){
          write_cb(2,write_config,write_addr0/4,write_addr1/4,0,0,literal0,literal1,0,0,circ_buff0,circ_buff1,circ_buff2,
circ_buff3);
          }
          write_config += 2;
          write_addr0 += 2; write_addr1 += 2; write_addr2 += 2; write_addr3 += 2;
          out_buff.data(7,0)   = literal0; //Write 2 literals to output buffer
          out_buff.data(15,8)  = literal1;
          out_buff.keep = 0b0011;
          state = 0; //Return to Literal state
        }
        else if(distance_remaining == 1){
          read_cb(1,read_config,read_addr0/4,0,0,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
          if(distance != 32768){
            write_cb(1,write_config,write_addr0/4,0,0,0,literal0,0,0,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
          }
          write_config++;
          write_addr0++; write_addr1++; write_addr2++; write_addr3++;
          out_buff.data(7,0)   = literal0; //Read literal from buffer
          out_buff.keep = 0b0001;
          length_remaining -= 1;
          distance_remaining = distance; //Reset sliding window
          read_addr0 = first_read_addr; // Shift addresses back to start
          read_addr1 = first_read_addr + 1;
          read_addr2 = first_read_addr + 2;
          read_addr3 = first_read_addr + 3;
        }
      }
      else if(length_remaining == 1){
        read_cb(1,read_config,read_addr0/4,0,0,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        if(distance != 32768){
          write_cb(1,write_config,write_addr0/4,0,0,0,literal0,0,0,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        }
        write_config++;
        write_addr0++; write_addr1++; write_addr2++; write_addr3++;
        out_buff.data(7,0)   = literal0; //Read literal from buffer
        out_buff.keep = 0b0001;
        state = 0; //Return to Literal state
      }
      out_strm << out_buff; //Write to stream
    }
    else if(state == 3){ //Distance <= 4, Single-Read State before Write State
```

```
      switch(distance){ //Set up copy array based on distance value
      case 1 :
        read_cb(1,read_config,read_addr0/4,0,0,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        copy_array[0][0] = literal0; //Store in copy array
        copy_array[0][1] = literal0;
        copy_array[0][2] = literal0;
        copy_array[0][3] = literal0;
        break;
      case 2 :
        read_cb(2,read_config,read_addr0/4,read_addr1/4,0,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        copy_array[0][0] = literal0; //Store in copy array
        copy_array[0][1] = literal1;
        copy_array[0][2] = literal0;
        copy_array[0][3] = literal1;
        break;
      case 3 :
        read_cb(3,read_config,read_addr0/4,read_addr1/4,read_addr2/4,0,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        copy_array[0][0] = literal0; //Store 0120 in copy array 0
        copy_array[0][1] = literal1;
        copy_array[0][2] = literal2;
        copy_array[0][3] = literal0;
        copy_array[1][0] = literal1; //Store 1201 in copy array 1
        copy_array[1][1] = literal2;
        copy_array[1][2] = literal0;
        copy_array[1][3] = literal1;
        copy_array[2][0] = literal2; //Store 2012 in copy array 2
        copy_array[2][1] = literal0;
        copy_array[2][2] = literal1;
        copy_array[2][3] = literal2;
        break;
      case 4 :
        read_cb(4,read_config,read_addr0/4,read_addr1/4,read_addr2/4,read_addr3/4,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        copy_array[0][0] = literal0; //Store in copy array
        copy_array[0][1] = literal1;
        copy_array[0][2] = literal2;
        copy_array[0][3] = literal3;
        break;
      }
      copy_pointer = 0;
      state = 4;
     }
     else if(state == 4){ //Distance <= 4, Write State
      if(length_remaining >= 4){
        literal0 = copy_array[copy_pointer][0];
        literal1 = copy_array[copy_pointer][1];
        literal2 = copy_array[copy_pointer][2];
        literal3 = copy_array[copy_pointer][3];
        if(distance == 3){ //If distance is 3, shift copy array pointer
          copy_pointer = (copy_pointer == 2) ? (ap_uint<2>)0b00 : (ap_uint<2>)(copy_pointer + 1); //Increment copy pointer or set it back to 0
        }
        write_cb(4,write_config,write_addr0/4,write_addr1/4,write_addr2/4,write_addr3/4,literal0,literal1,literal2,literal3,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        write_addr0 += 4; write_addr1 += 4; write_addr2 += 4; write_addr3 += 4;
        out_buff.data(7,0)   = literal0; //Write 4 literals to output buffer
        out_buff.data(15,8)  = literal1;
        out_buff.data(23,16) = literal2;
        out_buff.data(31,24) = literal3;
        out_buff.keep = 0b1111;
        if(length_remaining == 4){ //If only 4 bytes were left
          state = 0; //Return to Literal state
        }
        else{
          length_remaining -= 4;
        }
      }
      else if(length_remaining == 3){
        literal0 = copy_array[copy_pointer][0];
        literal1 = copy_array[copy_pointer][1];
        literal2 = copy_array[copy_pointer][2];
        write_cb(3,write_config,write_addr0/4,write_addr1/4,write_addr2/4,0,literal0,literal1,literal2,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        write_config += 3;
        write_addr0 += 3; write_addr1 += 3; write_addr2 += 3; write_addr3 += 3;
        out_buff.data(7,0)   = literal0; //Write 3 literals to output buffer
        out_buff.data(15,8)  = literal1;
        out_buff.data(23,16) = literal2;
        out_buff.keep = 0b0111;
        state = 0; //Return to Literal state
      }
      else if(length_remaining == 2){
        literal0 = copy_array[copy_pointer][0];
        literal1 = copy_array[copy_pointer][1];
        write_cb(2,write_config,write_addr0/4,write_addr1/4,0,0,literal0,literal1,0,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        write_config += 2;
        write_addr0 += 2; write_addr1 += 2; write_addr2 += 2; write_addr3 += 2;
        out_buff.data(7,0)   = literal0; //Write 2 literals to output buffer
```

```
        out_buff.data(15,8)  = literal1;
        out_buff.keep = 0b0011;
        state = 0; //Return to Literal state
      }
      else if(length_remaining == 1){
        literal0 = copy_array[copy_pointer][0];
        write_cb(1,write_config,write_addr0/4,0,0,0,literal0
  ,0,0,0,circ_buff0,circ_buff1,circ_buff2,circ_buff3);
        write_config++;
        write_addr0++; write_addr1++; write_addr2++; write_addr3
  ++;
        out_buff.data(7,0)   = literal0; //Read literal from
  buffer
        out_buff.keep = 0b0001;
        state = 0; //Return to Literal state
      }
      out_strm << out_buff; //Write to stream
    }
  }
  out_buff.keep = 0b0000; //Set TKEEP signal
  out_buff.last = 0b1; //Set TLAST signal
  out_strm << out_buff;
}
```

Appendix M

LZ77 Decoder Header File

LZ77_Decoder.h

```

#include <stdio.h>
#include <string.h>
#include <ap_int.h>
#include <hls_stream.h>
#include <assert.h>

using namespace hls;

typedef struct{ //Input AXI-Stream structure
  ap_uint<32> data; //Four bytes of data
  ap_uint<4> keep;  //TKEEP Signals for each byte
  bool user;      //TUSER Signal to identify Literal(0) or Length-
    Distance pair(1)
  bool last;      //TLAST AXI-Stream signal
} lld_stream;

typedef struct{ //Output AXI-Stream structure
  ap_uint<32> data; //Four bytes of data
  ap_uint<4> keep;  //TKEEP Signals for each byte
  bool last;      //TLAST AXI-Stream signal
} io_stream;
```

Appendix N

Byte Packer Source File

byte_packer.cpp

```
#include "byte_packer.h"

unsigned char align_bytes(
    io_strm &data_reg_in,
    io_strm &data_reg_out
){
#pragma HLS INLINE
  unsigned char bytes_needed;

  switch(data_reg_in.keep){
  case 0b0000 : //All bytes Null
    data_reg_out.data = data_reg_in.data; //Pass data through anyway
    data_reg_out.keep = 0b0000;
    bytes_needed = 4;
    break;
  case 0b0001 :
    data_reg_out.data = data_reg_in.data;
    data_reg_out.keep = 0b0001;
    bytes_needed = 3;
    break;
  case 0b0010 :
    data_reg_out.data = data_reg_in.data >> 8; //Shift all bytes right by 8 bits
    data_reg_out.keep = 0b0001;
    bytes_needed = 3;
    break;
  case 0b0011 :
    data_reg_out.data = data_reg_in.data;
    data_reg_out.keep = 0b0011;
    bytes_needed = 2;
    break;
  case 0b0100 :
    data_reg_out.data = data_reg_in.data >> 16; //Shift all bytes right by 16 bits
    data_reg_out.keep = 0b0001;
    bytes_needed = 3;
    break;
  case 0b0101 :
    data_reg_out.data(15,8) = data_reg_in.data(23,16); //Shift
   second byte right by 8 bits
    data_reg_out.data(7,0)  = data_reg_in.data(7,0);   //Copy
   bottom byte
    data_reg_out.keep = 0b0011;
    bytes_needed = 2;
    break;
  case 0b0110 :
    data_reg_out.data = data_reg_in.data >> 8; //Shift all bytes
   right by 8 bits
    data_reg_out.keep = 0b0011;
    bytes_needed = 2;
    break;
  case 0b0111 :
    data_reg_out.data = data_reg_in.data;
    data_reg_out.keep = 0b0111;
    bytes_needed = 1;
    break;
  case 0b1000 :
    data_reg_out.data = data_reg_in.data >> 24;
    data_reg_out.keep = 0b0001;
    bytes_needed = 3;
    break;
  case 0b1001 :
    data_reg_out.data(15,8) = data_reg_in.data(31,24); //Shift
   upper byte right by 16 bits
    data_reg_out.data(7,0)  = data_reg_in.data(7,0);   //Copy
   bottom byte
    data_reg_out.keep = 0b0011;
    bytes_needed = 2;
    break;
  case 0b1010 :
    data_reg_out.data(7,0)  = data_reg_in.data(15,8);  //Shift
   third byte right by 8 bits
    data_reg_out.data(15,8) = data_reg_in.data(31,24); //Shift
   upper byte right by 16 bits
    data_reg_out.keep = 0b0011;
    bytes_needed = 2;
    break;
  case 0b1011 :
    data_reg_out.data(23,16) = data_reg_in.data(31,24); //Shift
   upper byte right by 8 bits
    data_reg_out.data(15,0)  = data_reg_in.data(15,0);  //Copy
   bottom two bytes
    data_reg_out.keep = 0b0111;
    bytes_needed = 1;
    break;
  case 0b1100 :
    data_reg_out.data = data_reg_in.data >> 16; //Shift all bytes
   right by 16 bits
    data_reg_out.keep = 0b0011;
    bytes_needed = 2;
    break;
  case 0b1101 :
    data_reg_out.data(23,8) = data_reg_in.data(31,16); //Shift
   upper two bytes right by 8 bits
    data_reg_out.data(7,0)  = data_reg_in.data(7,0);   //Copy
   bottom byte
    data_reg_out.keep = 0b0111;
    bytes_needed = 1;
    break;
  case 0b1110 :
    data_reg_out.data = data_reg_in.data >> 8; //Shift all bytes
   right by 8 bits
    data_reg_out.keep = 0b0111;
    bytes_needed = 1;
    break;
  case 0b1111 :
    data_reg_out.data = data_reg_in.data;
    data_reg_out.keep = 0b1111;
    bytes_needed = 0;
    break;
  }
  data_reg_out.last = data_reg_in.last; //Pass TLAST signal

  return bytes_needed;
};

//byte_packer:
//Takes in 4 byte wide (little-endian) AXI-Stream and checks
    accompanying TKEEP signals for presence of null bytes.
//Will remove null bytes from stream and realign bytes into 4 byte
     wide continuous output stream.
//If TLAST is asserted, will output currently held bytes
    regardless of having 4 bytes or not.
void byte_packer(
    stream<io_strm> &in_strm,
    stream<io_strm> &out_strm
){
#pragma HLS PIPELINE enable_flush
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis register both port=in_strm
#pragma HLS INTERFACE axis register both port=out_strm
  static io_strm in_buff = {0};   //Axi-Stream input buffer
  static io_strm data_regA = {0}; //First data register to be
    filled
  static io_strm data_regB = {0}; //Second data register to be
    filled, used to fill RegA
  static unsigned char bytes_neededA = 4; //Number of bytes needed
     to fill the data pack currently held in data_regA
  static unsigned char bytes_neededB = 4;
  static bool state = 0; //Current machine state, can be 1 or 0

  if(state == 0){ //Pass-through State: Continue streaming full
   data packs through until null bytes are encountered
```

```
      in_strm >> in_buff;
      bytes_neededA = align_bytes(in_buff, data_regA); //Align data pack and find bytes_needed
      if(bytes_neededA == 0){    //If no bytes needed, data already aligned
        out_strm << data_regA; //Stream out full data pack
      }
      else if(bytes_neededA == 4){   //If data pack is all null bytes
        if(data_regA.last){         //And TLAST is high
          out_strm << data_regA; //Stream out empty data pack with TLAST high
        }                 //Otherwise, discard empty datapack
      }                             //If bytes_needed is 1, 2, or 3
      else if(data_regA.last){   //And TLAST is high
        out_strm << data_regA; //Stream out data pack whether it is full or not
      }
      else{ //Bytes needed to fill this data pack
        state = 1; //Change to fill state
      }
    }
    else{ //Fill State: Replace null bytes with true data bytes from the stream. Remain here as long as stream is discontinuous
      in_strm >> in_buff; //Stream in another data pack
      bytes_neededB = align_bytes(in_buff, data_regB);   //Align it and store in Register B
      switch(bytes_neededA){//Check if data is needed in RegA
      case 0 : //Reg A is already full of good data (Should never happen)
        out_strm << data_regA;           //Stream out full data pack
        data_regA.data = data_regB.data >> 8; //Move RegB data to RegA
        data_regA.keep = data_regB.keep >> 1; //Move RegB TKEEP to RegA
        bytes_neededA = bytes_neededB;      //Update number of bytes needed by RegA
        break;
      case 1 : //1 byte needed by RegA
        if(bytes_neededB == 3){ //If number of bytes in RegB is exactly 1
          data_regA.data(31,24) = data_regB.data(7,0); //Take 1 byte from it
          data_regA.keep = 0b1111;           //Update RegA TKEEP signal
          data_regA.last = data_regB.last;       //Pass TLAST signal
          out_strm << data_regA;             //Stream out full data pack
          state = 0;                      //Return to pass-through state
        }
        else if(bytes_neededB <= 2){ //If number of bytes in RegB is 2 or more (more than needed)
          data_regA.data(31,24) = data_regB.data(7,0); //Take 1 byte from it
          data_regA.keep = 0b1111;           //Update RegA TKEEP signal
          out_strm << data_regA;             //Stream out full data pack
          data_regA.data = data_regB.data >> 8;    //Shift RegB data and store in RegA
          data_regA.keep = data_regB.keep >> 1;    //Shift RegB TKEEP signal and store in RegA
          bytes_neededA = bytes_neededB + 1;       //Update number of bytes needed by RegA
        }
        else{ //RegB has no useable bytes.
          if(data_regB.last){ //If RegB is last
            data_regA.last = 1;
            out_strm << data_regA; //Flush out datapack in RegA
            state = 0;         //Return to pass-through state
          } //Otherwise, read another data pack into RegB next iteration
        }
        break;
      case 2 : //2 bytes needed by RegA
        if(bytes_neededB == 2){ //If number of bytes in RegB is exactly 2
          data_regA.data(31,16) = data_regB.data(15,0); //Take 2 bytes
          data_regA.keep = 0b1111;
          data_regA.last = data_regB.last;
          out_strm << data_regA;
          state = 0;  //Return to pass-through state
        }
        else if(bytes_neededB == 3){ //If only 1 byte in RegB (less than needed)
          data_regA.data(23,16) = data_regB.data(7,0); //Take 1 byte
          data_regA.keep = 0b0111; //Update TKEEP (in case TLAST is 1)
          if(data_regB.last){ //If TLAST is high
            data_regA.last = 1;
            out_strm << data_regA; //Write out regA
            state = 0;         //Return to pass-through state
          }
          else{ //Otherwise
            bytes_neededA = 1; //Update number of bytes needed by RegA
          }
        }
        else if(bytes_neededB <= 1){ //If number of bytes in RegB is 3 or more (more than needed)
          data_regA.data(31,16) = data_regB.data(15,0); //Take 2 bytes
          data_regA.keep = 0b1111;            //Update RegA TKEEP signal
          out_strm << data_regA;              //Stream out full data
           pack
          data_regA.data = data_regB.data >> 16;      //Shift RegB
         data and store in RegA
          data_regA.keep = data_regB.keep >> 2;     //Shift RegB
         TKEEP signal and store in RegA
          bytes_neededA = bytes_neededB + 2;        //Update number
         of bytes needed by RegA
        }
        else{ //RegB has no useable bytes
          if(data_regB.last){ //If RegB is last
            data_regA.last = 1;
            out_strm << data_regA; //Flush out datapack in RegA
            state = 0;         //Return to pass-through state
          } //Otherwise, read another data pack into RegB next
         iteration
        }
        break;
      case 3 : //3 bytes needed by RegA
        if(bytes_neededB == 1){ //If number of bytes in RegB is
         exactly 3
          data_regA.data(31,8) = data_regB.data(23,0); //Take 3
         bytes
          data_regA.keep = 0b1111;
          data_regA.last = data_regB.last;
          out_strm << data_regA;
          state = 0;  //Return to pass-through state
        }
        else if(bytes_neededB == 2){ //If only 2 bytes in RegB (less
          than needed)
          data_regA.data(23,8) = data_regB.data(15,0); //Take 2
         bytes
          data_regA.keep = 0b0111; //Update TKEEP (in case TLAST is
         1)
          if(data_regB.last){
            data_regA.last = 1;
            out_strm << data_regA; //Write out regA
            state = 0;         //Return to pass-through state
          }
          else{
            bytes_neededA = 1;
          }
        }
        else if(bytes_neededB == 3){ //If only 1 byte in RegB (less
         than needed)
          data_regA.data(15,8) = data_regB.data(7,0); //Take 1 byte
          data_regA.keep = 0b0011; //Update TKEEP (in case TLAST is
         1)
          if(data_regB.last){
            data_regA.last = 1;
            out_strm << data_regA; //Write out regA
            state = 0;         //Return to pass-through state
          }
          else{
            bytes_neededA = 2;
          }
        }
        else if(bytes_neededB == 0){ //If number of bytes in RegB is 4 (more than needed)
          data_regA.data(31,8) = data_regB.data(23,0); //Take 3 bytes
          data_regA.keep = 0b1111;            //Update RegA TKEEP signal
          out_strm << data_regA;              //Stream out full data pack
          data_regA.data = data_regB.data >> 24;      //Shift RegB data and store in RegA
          data_regA.keep = data_regB.keep >> 3;     //Shift RegB TKEEP signal and store in RegA
          bytes_neededA = bytes_neededB + 3;        //Update number of bytes needed by RegA
        }
        else{ //RegB has no useable bytes
          if(data_regB.last){ //If RegB is last
            data_regA.last = 1;
            out_strm << data_regA; //Flush out datapack in RegA
            state = 0;          //Return to pass-through state
          } //Otherwise, read another data pack into RegB next iteration
        }
        break;
      case 4 : //RegA is empty (Should never happen)
        data_regA.data = data_regB.data >> 8; //Move RegB data to RegA
        data_regA.keep = data_regB.keep >> 1; //Move RegB TKEEP to RegA
        bytes_neededA = bytes_neededB;      //Update number of bytes needed by RegA
        break;
      }
    }
  }
```

Appendix O

Byte Packer Header File

byte_packer.h

```
#include <stdio.h>
#include <string.h>
#include <ap_int.h>
#include <hls_stream.h>

using namespace hls;

template<int D>
struct axi_strm{
  ap_uint<D> data;    //D bits of data
  ap_uint<D/8> keep; //TKEEP signals for each byte
  bool last;          //TLAST AXI-Stream signal
};
//Stream Format:
// | 31 - Byte3 - 24 | 23 - Byte2 - 16 | 15 - Byte1 - 8 | 7 -
    Byte0 - 0 |
// |    TKEEP3    |    TKEEP2     |    TKEEP1   |   TKEEP0    |

typedef axi_strm<32> io_strm;
```