FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms

Suzanne Rigler
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
Email: slrigler@uwaterloo.ca

William Bishop
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
Email: wdbishop@uwaterloo.ca

Andrew Kennings
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
Email: akennings@uwaterloo.ca

***Abstract*— Lossless data compression algorithms are widely used by data communication systems and data storage systems to reduce the amount of data transferred and stored. GZIP is a popular, patent-free compression program that delivers good compression ratios. This paper presents hardware implementations for the LZ77 encoders and Huffman encoders that form the basis for a full hardware implementation of a GZIP encoder. The designs have been implemented as state machines in VHDL in such a way that they are suitable for implementation using either FPGA or ASIC technologies. Performance metrics and resource utilization results obtained for a prototype implementation running on an Altera DE2 board are presented. Ultimately, the goal is to utilized the LZ77 encoders and Huffman encoders described in this paper to build a fully-functional, hardware design for a GZIP encoder that could be used in data communication systems and data storage systems to boost overall system performance[1].**

I. Introduction

The evolution of computing has created a rapid expansion in the volume of data to be stored on hard disks and sent over the Internet. This growth has led to a need for data compression (i.e., the ability to reduce the amount of storage or Internet bandwidth required to handle data). Data compression can be either lossy or lossless. Lossy compression works on the assumption that data doesn't need to be stored perfectly. For example, photographs can be stored using lossy compression techniques without noticeable degradation. Lossless data compression is used when data cannot afford to be lost or degraded. Text files are typically stored using lossless techniques since the loss of a single character can be dangerously misleading in some situations. For the purpose of archiving original images, videos, and audio files, lossless compression is often used.

To meet the demand for lossless compression, GZIP [1] was introduced. GZIP compresses data to create a smaller and less demanding representation for storing and transferring data. It was designed as a replacement for COMPRESS. When compared to COMPRESS, GZIP delivers better compression ratios and GZIP is patent-free. GZIP is based on the DEFLATE algorithm [2] which is a lossless data compression algorithm that uses a combination of LZ77 and Huffman encoding. GZIP has long been a popular tool for the compression and storage of data files on hard-drives. Recently, GZIP has also become a popular tool to compress and decompress websites. One key advantage of the GZIP algorithm over alternatives is that GZIP is fast. It is often faster to compress data using GZIP, communicate the compressed file, and decompress the data using GZIP than to simply send the original, uncompressed data. The USWeb website reported that page load times decrease by up to 40% [3] when GZIP is used.

GZIP compression software has become a standardized part of the HTTP protocol and most web browsers have built-in GZIP decompression software. The HTTP/1.1 protocol allows for clients to optionally request the compression of content from the server [4]. The standard itself specifies two compression methods: GZIP and DEFLATE. Both are supported by many HTTP client libraries and almost all modern browsers.

This paper presents hardware implementations for the LZ77 encoders and the Huffman encoders that form the basis for a full hardware implementation of GZIP. The motivation for a hardware implementation of GZIP is clear. A hardware implementation of GZIP offers the possibility of off-loading the compression and decompression of data files from a CPU to a co-processor. A hardware implementation of GZIP on a single chip (e.g., a low cost FPGA) also offers the interesting possibility that GZIP compression could be embedded into storage area networks, web servers, and other network equipment to simplify processing requirements.

This paper is organized as follows. Section II provides background on LZ77 and Huffman encoding and introduces related work. Section IV and Section III present the LZ77 encoders and the Huffman encoders, respectively. Experimental results are presented in Section V. Conclusions are drawn in Section VI.

II. Background

A. LZ Encoding Algorithms

LZ (Lempel-Ziv) encoding algorithms are generic, dictionary-based lossless compression algorithms that exploit regularities in data [5]. LZ77, LZSS, LZ78, and LZW are the most common variations of LZ encoding algorithms. GZIP utilizes LZ77 compression.

[1]This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant #203763-03.

0840-7789/07/$25.00 ©2007 IEEE

LZ77 processes data from left to right, inserting every string into the dictionary. Quite often, the dictionary is limited by available memory, so a sliding dictionary is used. A sliding dictionary maintains a list of the most recently used strings. If a string does not occur in the dictionary, the string is emitted as a literal sequence of bytes. If a match is found, the string is replaced by a pointer to the matching string in the form of distance-length pair. The distance-length pair is composed of two parts: the distance from the current string to the start of the matching string and the length of the match. The newly compressed information is also preceded by a flag bit to distinguish literals from distance-length pairs. Flag bits can be packed together into one byte to conserve memory. Efficent access to the dictionary is key so most programs, including GZIP, implement the dictionary using hashing functions.

Hardware implementations of LZ77 encoding algorithms typically use either Content Addressable Memories (CAMs) or systolic arrays. Although CAMs lead to high throughput, CAMs are costly in terms of hardware requirements. Systolic arrays require less hardware and deliver improved testability. Both technologies tend to be complex and dependent upon features provided the hardware architecture.

B. Huffman Encoding Algorithms

Huffman coding [6] is based on the concept of mapping an alphabet to a different representation composed of strings of variable size such that symbols with a high probability of occurring have a smaller representation than those that occur less often. There are two variations of Huffman encoding: static and dynamic. Static Huffman encoding uses a look-up table that stores a pre-defined frequency for each character in the alphabet. Unlike static Huffman encoding, dynamic Huffman encoding calculates the actual frequency of characters based on the data in the file to be compressed. Once the frequency data has been determined, the two variations are identical. The two elements with the lowest weights are selected. These two elements are inserted as leaf nodes of a node with two branches. The frequencies of the two elements selected are then added together and this value becomes the frequency for the new node (see Fig. 1). The algorithm continues selecting two elements at a time until a Huffman tree is complete with the root node having a frequency of 100%.

Classic Huffman encoding is non-deterministic. It is possible for a data set to be represented by more than one possible tree. GZIP applies two additional rules to ensure that each data set has at most one possible tree representation. First, elements with shorter codes are placed to the left in the Huffman tree. In Figure 1, D and E have the longest codes so they appear in the right branch of the tree. Second, elements with codes of the same length appear in the order in which they are first encountered. In Figure 1 D and E are the only elements with a code length of four. Since D appears first in the data set, D is assigned to the left branch and E is assigned to the right branch. These two restrictions ensure that at most one possible tree representation exists for every set of elements and their respective code lengths. The uniqueness of the tree implies that the entire tree can be regenerated if the length of each code is known.

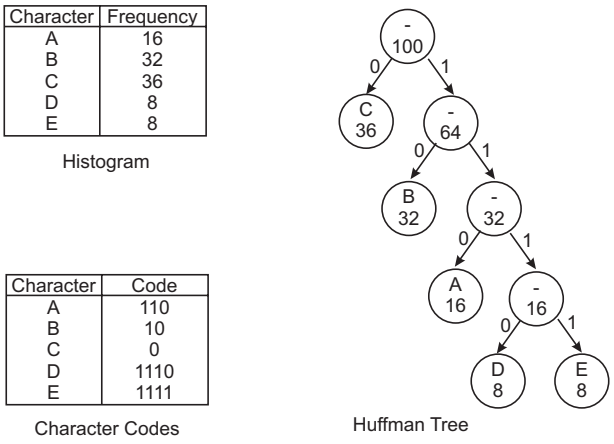

Fig. 1. An example of huffman encoding.

Numerous hardware implementations of static Huffman encoders can be found [7]–[11]. However, a full hardware implementation of GZIP requires both static and dynamic Huffman encoders. Existing hardware implementations of dynamic Huffman encoders [12], [13] do not implement the two additional rules required by GZIP to ensure that the Huffman tree is deterministic.

III. Huffman Hardware Implementation

A. Data Structures

The GZIP encoder requires three dynamic Huffman trees to compress a block of data. All three Huffman trees use the same data structures but the trees are different sizes since each tree uses a different alphabet.

The main feature of a dynamic Huffman encoder is that the frequency data is calculated for the input data and not read from a pre-defined look-up table. A RAM named `freq` is used to store the frequencies of each character. Initially, all locations in the RAM are initialized to zero. As characters are observed, the corresponding location in the RAM is incremented by one.

Two RAMs are used to store the values in the Huffman tree. A RAM named `parent` is used to keep track of every node's parent in the tree. Similarly, a RAM named `depth` is used to keep track of the depth of each node in the Huffman tree.

Perhaps the most important step in the actual Huffman algorithm is to choose the two elements with the smallest frequencies. This is accomplished using a min heap. A RAM named `heap` is used to store the heap data.

Once the Huffman algorithm is complete, the lengths of each character in the tree are computed. This requires a RAM named `lengths`. After the code lengths have been calculated, the codes can be determined and stored in a RAM named `codes`. The `lengths` and `codes` RAMs are dual-ported to allow for the Huffman tree to be retrieved efficiently outside the block.

Finally a small RAM named nextCount is used when determining the code lengths and the codes. This RAM is a fixed size of 16×15 bits.

B. Implementation

The Huffman encoder consists of six stages. The stages are described in detail in the following sections.

1) Initialization: The initialization stage performs two tasks. First, it initializes all variables. Second, it assigns values of zero to all locations in freq, codes, lengths.

2) Reading Input: This stage reads the data and updates the frequencies for the dynamic Huffman tree. This is accomplished by incrementing the appropriate location in freq each time a character is processed. This loop creates the histogram used by the dynamic Huffman encoder.

3) Heap Setup: The third stage creates a min heap that is used to retrieve the smallest element to be processed. Initially, all elements where the corresponding location in freq does not equal zero are inserted into heap and depth is assigned a value of zero for each element. Once all elements in freq have been processed, the frequencies are updated and the heap is rebalanced accordingly.

4) Huffman Algorithm: The fourth stage implements the most difficult and time consuming task of the implementation. This stage begins by removing the smallest element from the balanced heap, assigning it to a variable p, and re-balancing the heap. Then, the next smallest element is removed from the heap, assigned to the variable q, and the heap is re-balanced. The freq RAM is then written the value of freq(p)+freq(q) and depth is written the value of max(depth(p)+depth(q))+1. The value of parent is also assigned at p and q to the next available location in heap. Next, heap at location one is assigned the new value and a re-balancing is performed. This loop continues while the heap length is greater than or equal to two.

5) Calculating Code Lengths: The fifth stage calculates code lengths. Once the Huffman tree has been computed and all the pointers in parent have been assigned, computing the code lengths is simple. For each value in heap, the length and parent are retrieved from their respective RAMs. The child node's length is then updated by assigning length(parent) the value of the parent's length+1. This routine is similar to traversing a tree. Every child node has a length equal to one plus its parent's length.

6) Calculating Codes: The sixth stage calculates code words based based on the code lengths. This requires two separate loops. The first loop iterates through each possible code length assigning nextCount the first possible value for each code based on the number of codes with the same length. The second loop assigns all the values to the codes RAM. This is accomplished by iterating through all the values in lengths not equal to zero and reading the value in nextCount. This value is then stored in codes and nextCount is incremented. This loop effectively reads nextCount to get the first available code for that code length, assigns it to the tree and then increments it for the next code.

IV. LZ Hardware Implementation

The LZ77 implementation consists of 4 RAMs and a state machine. Two RAMs, head and prev (both of size 32 k×32 bits) implement a chaining hash table. The RAM head stores the most recent string hashed into the corresponding location in the hash table while prev contains previous strings hashed to the corresponding location in the hash table. With this design, the first string matched in the hash table is always the closest to the current data. A RAM named window (of size 32 k×8 bits) stores the input data[2]. A RAM named encodeData (of size 15×16 bits) stores distance-length pairs and literals to be written as output[3].

A variable, lookAhead, is used to store the number of characters left to be processed and a variable strStart is used to index window. Other important variables include flag and flagPos which are used to detect flag bits and to select when to output data. Finally, a variable, matchAvailable, is used as a flag to determine when a match should be processed during the LZ77 encoding.

A. Implementation Details

The implementation initializes all variables and RAMs accordingly. RAMs head and prev are initialized to values of zero and RAM window is populated with the data to be compressed. An incremental hash function is used. This allows the hash value to be incrementally updated for each string rather than recalculated minMatch times for every string, where minMatch=3. To begin the LZ77 algorithm, the hash value h must be updated for the first minMatch-1 characters.

The hash calculation, updateHash, is completed using the pseudocode is in Figure 2, where c is the character to be processed from window.

```
Procedure: updateHash(h,c)
begin
h = ((h << d) xor c) and hMask;
    where hMask = hSize - 1 and d = 5
end
```

Fig. 2. Pseudocode for Updating the Hash Value

If lookAhead is greater than zero, the main loop of the algorithm is entered. Otherwise, the algorithm is complete. The first task inside the while loop is the insertion of a string of length minMatch into the hash table. The hash value is updated by calling updateHash(h,window(strStart+minMatch-1)). Once the hash value has been updated, the insertion is performed (see Fig. 3). It can be seen that insertions are quick and simple, and deletions are unnecessary since a sliding dictionary is used and older strings will be replaced.

After the string of length minMatch has been inserted in the hash table, the previous match needs to be

[2]The block size of 32 k equals the block size used in GZIP.

[3]The FPGA prototype described in Section V off-loads window, head and prev to on-board SRAM while encodeData uses RAM internal to the FPGA. This partitioning is due to the memory limitations imposed by the chosen FPGA platform.

```
Procedure: InsertString(strStart)
begin
h = updateHase(h,window(strStart + minMatch - 1));
prev(strStart and wMask) = head(h);
head(h) = hashHead = strStart;
    where wMask = dSize - 1;
end
```

Fig. 3. Pseudocode for Inserting a String into the Dictionary

saved. If the value of `hashHead` which was computed in `insertString` does not equal zero, the `longestMatch` routine is entered. Otherwise, the function is skipped. The `longestMatch` routine iterates through each value on the hash chain beginning at `hashHead` and searches for the longest match. This is accomplished by comparing two locations in `window` to find a match and searching for any previous matches found.

After exiting the `longestMatch` routine, a comparison is performed to determine what, if anything, should be stored for output. If a better match was found in the previous iteration of the loop, the distance-length pair of the previous match is processed. This is accomplished by storing the output to `encodedData`, adjusting the `flag` to indicate that a distance-length pair has been encoded, decrementing `lookAhead` by the previous match length and assigning `matchAvailable` a value of zero. An insert is also performed for all the substrings of length `minMatch` into the hash table by calling `insertString` several times and incrementing `strStart`. If the first statement is not satisfied, a check is performed to see if `matchAvailable` equals one. If the condition holds true, then no match was found during the last iteration and a better match was not found so a literal is stored in `encodedData`, `lookAhead` is decremented and `strStart` is incremented. If neither of the previous conditions hold true, the algorithm waits for the next iteration of the loop to determine what to do. This is accomplished by assigning one to `matchAvailable`, decrementing `lookAhead`, and incrementing `strStart`.

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

V. FPGA Resource Utilization

The FPGA resource utilizations for the hardware implementations of the LZ77 encoder and the three Huffman encoders are outlined in Table I. Further optimizations to the implementations are possible.

TABLE I
FPGA Resource Utilization

	HUFF 1	**HUFF 2**	**HUFF 3**	**LZ77**
Logic Elements	2066	1980	1842	2077
Registers	573	533	502	734
Memory Bits	40234	4410	2969	8192
9-bit Multipliers	10	10	6	0

VI. Conclusion

This paper has described hardware implementations of LZ77 encoders and dynamic Huffman encoders suitable for a hardware implementation of GZIP. The encoders have been written in VHDL and tested using an Altera DE2 development board. These hardware implementations provide the basis for a GZIP encoder in hardware. Eventually, it will be possible to merge these designs to build an encoder that is capable of generating compressed files that may be decompressed using a standard implementation of GZIP.

References

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

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

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

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

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

[6] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[7] N. Dandalis and V. Prasanna, "Configuration compression for FPGA-based embedded systems," *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.

[8] T. Lee and J. Park, "Design and implementation of a static Huffman encoding hardware using a parallel shifting algorithm," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5, pp. 2073–2080, October 2004.

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

[10] S. Mathen, "Wavelet transform based adaptive image compression on FPGA," Master's thesis, Unviersity of Calicut, 1996, http://www.ittc.ku.edu/projects/ACS/documents/sarin_presentation.pdf.

[11] J. Nikara, "Parallel Huffman decoder with an optimize look up table option on FPGA," Ph.D. dissertation, Tampere University of Technology, 2004.

[12] M. W. E. Jamro and K. Wiatr, "FPGA implementation of the dynamic Huffman encoder," *Proceedings of IFAC Workshop on Programmable Devices and Embedded Systems*, 2006.

[13] S. Sun and S. Lee, "A JPEG chip for image compression and decompression," *Journal of VLSI Signal Processing Systems*, vol. 35, no. 1, pp. 43–60, 2003.