VLSI implementation of a high speed LZW data compressor

by Robert Lyle Wall

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering
Montana State University

Abstract:
The growing volume of data stored and transmitted by computer is creating an increasing demand for efficient means of reducing the size of this data, while retaining all or most of its information content. This process is known as data compression, and it is frequently classified by whether the data recovered by the decompression process is always exactly the same as the original (lossless) or is allowed to vary from the original somewhat (lossy). This discussion will concentrate on lossless data compression methods.

Today, most lossless data compression is still being performed in software. There are a small number of integrated circuits available which implement compression algorithms directly in hardware, but their execution speed is fairly limited. A design is presented for a VLSI integrated circuit which will perform lossless data compression at speeds roughly an order of magnitude greater than those currently available. It is based on the Lempel-Ziv-Welch (LZW) algorithm and relies on a high-speed content-addressable memory (also known as associative memory) to provide its performance increase.

The process by which this algorithm has been subdivided into hardware modules is described, and the implementation of various modules using VLSI standard cell design techniques is presented. The similarities between standard cell circuit design and software development are examined, and the applicability of established software development methodologies to this type of design process is considered.

Simulation and timing analysis of the modules suggests that the fully assembled circuit will be capable of compressing data at the rate of ten million bytes per second, which is nearly five times that of commercially available hardware, and the decompression rate will be only slightly less.

VLSI IMPLEMENTATION OF A HIGH SPEED LZW DATA COMPRESSOR

by

Robert Lyle Wall

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

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

August 1991

N378
W1495

APPROVAL

of a thesis submitted by

Robert Lyle Wall

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

August 8, 1991	[illegible]
Date	Chairperson, Graduate Committee

Approved for the Major Department

	[illegible] 8-8-91
Date	Head, Major Department

Approved for the College of Graduate Studies

August 9, 1991	Henry L Parsons
Date	Graduate Dean

STATEMENT OF PERMISSION TO USE

Signature Robert L. Wall

Date Aug. 9, 1991

ACKNOWLEDGEMENTS

I would especially like to thank Professor Kel Winters for the guidance, suggestions, and inspiration he has provided. I also wish to thank Reza Massarat for his work on the project. Thanks also to Jaye Mathisen, for his help with the OCT tools and UNIX problems, and to Rick Spickelmier, Andrea Cassotto, and Luciano Lavagno of the University of California at Berkeley for their support of OCT. Finally, thanks to Pat Owsley and others at Advanced Hardware Architectures for funding this research project.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

Data Compression Definitions 1
Entropy 5
Common Data Compression Algorithms 6
Huffman Coding 6
Arithmetic Coding 8
Dictionary-Based Algorithms 11
Other Data Compression Algorithms 12
Choosing an Algorithm for Hardware Implementation 13

2. LZW COMPRESSION AND DECOMPRESSION ALGORITHMS 15

The Basic LZ78 Compression Algorithm 15
The LZW Algorithms 17
Compression Algorithm 21
Decompression Algorithm 24
Adaptation for High-Speed Hardware Implementation 27
Compressor String Table Search 28
Decompressor String Reversal 29
Hardware Design Requirements 29

3. CONTENT-ADDRESSABLE MEMORY 31

Basic Content-Addressable Memory Characteristics 31
Commercially Available CAMs 32
Custom Dynamic CAM 34
Synchronous Mode-sensitive Operation 35
Predefined String Initialization 36
Reset Operation 37
Refresh Operation 37
Possible Performance Enhancements 38

4. STRING REVERSAL MECHANISM 40

String Reversal Algorithm 40
Physical Implementation Considerations 45

5. INPUT AND OUTPUT BUFFERING 49

Character Buffers 49
Code Buffers 51

TABLE OF CONTENTS--Continued

Page

6. HARDWARE DESIGN PROCEDURE 53

Target Technology 54
CAD Tools 54
Bdsyn 54
MisII 56
Bdnet 56
Wolfe 57
Musa 58
Design Methodology 58
External Reference Specifications 59
RTL Descriptions 59
BDS and Bdnet Input Files 61
The Logic Generation Process 62
Logic Simulation 63
Naming Conventions 64
Finite Field Sequencers 64
Problem Decomposition 66

7. THE COMPRESSOR MODULE 68

Module Design for Normal DCAM Operation 68
External Reference Specifications 69
RTL Description 69
Module Generation 72
Module Testing 74
Module Design for Enhanced DCAM Operation 75
External Reference Specifications 75
RTL Description 75
Module Generation 77
Module Testing 79

8. THE DECOMPRESSOR MODULE 80

Module Design to Accompany First Compressor 80
External Reference Specifications 80
RTL Description 81
Module Generation 84
Module Testing 86
Module Design to Accompany Single-Cycle Compressor 86
External Reference Specifications 87
RTL Description 87
Module Generation 88
Module Testing 90

TABLE OF CONTENTS--Continued

Page

9. MERGING THE COMPRESSOR AND DECOMPRESSOR 91

Additional Functionality 93
Halt Signal Handling 93
String Length Limiting 95
Complete Merged Module 96

10. THE STRING REVERSAL MODULE 97

Module Controller Design 97
External Reference Specifications 97
RTL Description 98
Module Generation 99
Module Testing 99
Addition of Musa RAM Models 99

11. THE MASTER CONTROLLER MODULE 103

Module Design 103
External Reference Specifications 103
RTL Description 104
Module Generation 106
Module Testing 106
System Integration and Test 106

12. MANAGING THE DESIGN PROCESS 107

13. CIRCUIT PERFORMANCE ESTIMATES 110

Timing Extraction Using MisII 110
Timing Information for LZW Modules 112
Compression Efficiency Measurement 115

14. CONCLUSIONS 117

Further Research 118

REFERENCES CITED 120

APPENDICES 124

Appendix A - Complete Set of Documentation for
Merged_2 Module 125
Appendix B - BDS and Bdnet Source for All Modules 146

LIST OF TABLES

Table Page

1. Merged_2 Timing Data 114

2. Comparison of Compression Ratios 116

LIST OF FIGURES

Figure Page

1. Example of a Dictionary Trie 18
2. Modified Dictionary Trie 19
3. String Table Representation of a Dictionary 20
4. LZW Compression Algorithm 21
5. Compression Example 22
6. Basic LZW Decompression Algorithm 24
7. Correct LZW Decompression Algorithm 26
8. Decompression Example 27
9. Block Diagram of String Reversal Mechanism 44
10. String Reversal Algorithm 45
11. Implementing Registers with BDS and Bdnet 62
12. Linear Feedback Shift Register 66
13. System Block Diagram 67
14. Normal Mode Compressor RTL 70
15. Compress Module Timing Diagram 73
16. BDS Finite Field Sequencer 73
17. Single-cycle Compressor RTL 76
18. Compress_2 Module Timing Diagram 78
19. Normal Mode Decompressor RTL 82
20. Decompress Timing Diagram 85
21. Enhanced Decompressor RTL 89
22. Decompress_2 Timing Diagram 90

LIST OF FIGURES--Continued

Figure Page

23. RTL for Addition of Halt States 94
24. String Reversal Timing Diagram 100
25. Master Controller RTL 105
26. Sample of MisII Timing Calculations 113
27. Sample ERS Form 126
28. Merged_2.RTL 128
29. Merged_2.BDS 133
30. Merged_2.bdnet 141
31. M2_comp.musa 144
32. Compress.BDS 147
33. Compress.bdnet 151
34. Compress_2.BDS 154
35. Compress_2.bdnet 157
36. Decompress.BDS 159
37. Decompress.bdnet 164
38. Decompress_2.BDS 167
39. Decompress_2.bdnet 171
40. Stack.RTL 174
41. Stack.BDS 179
42. Stack.bdnet 186
43. Stackram.bdnet 189
44. Stackram.musa 192

LIST OF FIGURES--Continued

Figure Page

45. Control.BDS 195

46. Control.bdnet 200

47. Makefile 202

ABSTRACT

The growing volume of data stored and transmitted by computer is creating an increasing demand for efficient means of reducing the size of this data, while retaining all or most of its information content. This process is known as data compression, and it is frequently classified by whether the data recovered by the decompression process is always exactly the same as the original (lossless) or is allowed to vary from the original somewhat (lossy). This discussion will concentrate on lossless data compression methods.

Today, most lossless data compression is still being performed in software. There are a small number of integrated circuits available which implement compression algorithms directly in hardware, but their execution speed is fairly limited. A design is presented for a VLSI integrated circuit which will perform lossless data compression at speeds roughly an order of magnitude greater than those currently available. It is based on the Lempel-Ziv-Welch (LZW) algorithm and relies on a high-speed content-addressable memory (also known as associative memory) to provide its performance increase.

The process by which this algorithm has been subdivided into hardware modules is described, and the implementation of various modules using VLSI standard cell design techniques is presented. The similarities between standard cell circuit design and software development are examined, and the applicability of established software development methodologies to this type of design process is considered.

Simulation and timing analysis of the modules suggests that the fully assembled circuit will be capable of compressing data at the rate of ten million bytes per second, which is nearly five times that of commercially available hardware, and the decompression rate will be only slightly less.

CHAPTER 1

INTRODUCTION

The growing volume of data being stored and transmitted by digital computers has enhanced interest in data reduction techniques. The amount of digitally stored text, electronic mail, image data, and executable binary images being accessed, archived, and transferred over bandwidth-limited channels has consistently outpaced technological improvements in data storage and communication capacity, demanding some means of reducing storage space and transmission time requirements. Recently, personal computers have shrunk dramatically in size, while the software packages which run on them continue to swell. However, advances in disk technology (specifically the amount of physical volume required to store a given amount of data) are not keeping up, so a demand is developing for some means of compacting data to be stored on relatively small disks and enlarging it when it is accessed. This data reduction or compaction is usually accomplished by some form of *data compression*.

Data Compression Definitions

Data compression is "the process of *encoding* a body of data D into a smaller body of data $\Delta(D)$. It must be possible for $\Delta(D)$ to be *decoded* back to D or some acceptable approximation of D." ([STORER88], p. 1). The objective of compressing a message is to minimize the number of symbols (typically binary digits, or bits) required to represent it, while still allowing it to be reconstructed. Compression techniques exploit the redundancy of a message, representing redundant portions in a form that requires fewer bits than the original. This reduced message is stored or transmitted, and when the original message is required, a corresponding reverse transformation, the decompression technique, is applied to recover the original information or an approximation thereof.

Compression techniques can be separated into several subdivisions. This paper will concentrate exclusively on techniques which are applicable to digital data processing, as opposed to signal encoding techniques studied in communications. These digital techniques include text compression and compression of digitally sampled analog data (although the two are not mutually exclusive; text compression algorithms are often successfully applied to two-dimensional image data). The primary difference between digital compression techniques (especially text compression) and compression of communications signals is that digital compressors typically do not have a well-defined statistical model of the data source to be compressed which can be tuned to optimize performance. It is thus necessary for the compression method to determine a model of the data source and compute probability distributions for the symbols in each data message. This task is essentially equivalent to finding the redundancy in the message.

There are several types of redundancy present in data typically encountered on computer systems. Four common types which have been identified are redundant character distribution (where some characters are used more frequently than others), character repetition, high-usage patterns (strings that are frequently used within blocks of data), and positional redundancy (where certain characters occur consistently at predictable places within the data) [WELCH]. Most blocks of data will exhibit one or more of these types of redundancy to some extent. An efficient data compression method should be able to exploit all types of redundancy as fully as possible.

A common measure of a compression method's efficiency (the amount by which it reduces the size of a data message) is the *compression ratio*. This is typically defined as the ratio of the number of fixed-length units (typically bytes) input by the compressor to the number of those units produced as compressed output. Obviously, the larger this number is, the better the compression performance. An alternative measure is the inverse of this ratio, the amount of space required by the compressed data over the amount required by the original data. The closer this number is to zero, the higher the compression efficiency.

One of the most important subdivisions of compression techniques is into *lossless* and *lossy* methods. As the name suggests, lossless methods allow the exact reconstruction of the original message from the compressed data, while lossy methods do not. Lossless methods are most appropriate, and usually essential, for text compression applications, where it is not acceptable to restore approximately the same data. Lossy methods are more typically used on digitally sampled analog data, where a good approximation is often sufficient. An example would be the compression of a digitized audio signal. Due to the imperfections of the human ear, the loss of a small amount of information would probably be unnoticeable. Another example is the new Joint Photographic Expert Group's (JPEG) proposed standard for compressing two dimensional image data. Since the restriction on exact recovery of the data has been relaxed, lossy techniques usually achieve greater amounts of compression. For example, lossless text compression methods typically reduce English text to between 40% and 70% of its original size, with some schemes approaching 25%; the best reduction possible is estimated to be no less than about 12% ([BELL], p. 13). JPEG, on the other hand, can achieve reduction to 2% or smaller without severely degrading the quality of the decompressed image [WALLACE]. This paper will focus exclusively on lossless methods, since they are applicable to a more general class of data storage and communication applications.

Another distinction frequently made is between *static* and *adaptive* techniques. This refers to the method used to determine the redundancy characteristics of the data. The basis of all compression algorithms is essentially the determination of the probability of a symbol or string of symbols appearing in the source message, and the replacement of high probability symbols or strings with short code representations and low probability symbols or strings with longer code representation. Static (non-adaptive) algorithms assume an a priori probability distribution of the symbols in order to assign codes to them. These types of algorithms typically suffer badly in compression ratio if the input data doesn't fit well with the assumed probability distributions. Adaptive algorithms either make an initial assumption about the symbol probabilities and adjust that assumption as the message is processed, or make an initial pass over the data to extract accurate probability information for the compression of that data. This probability information is then fixed for the compression of that message. The latter methods suffer in execution speed, since two passes must be made through the data. The former methods require only one pass through the data and will adjust to messages with probability distributions different than those originally assumed. A small percentage of the optimum compression efficiency is sacrificed as the compressor adjusts its assumptions to match the data, but this is much more robust than a static algorithm.

Another classification of algorithms is into *dictionary-based* and *statistical* schemes. In the former, each recognized string of input symbols is replaced by a reference to a previous occurrence of that string. The latter methods construct variable-length codes to represent input symbols based on the probability distribution of those symbols. This distinction will be discussed in more detail in the next section.

Entropy

A concept that is fundamental to data compression is that of *entropy*, which is a measure of the information content in any message produced by some source. This is directly related to the randomness of the source; for example, if a machine is being examined that produces messages that are always a single binary zero, there is very little information contained in these messages. This is because the source is not random, so it is known in advance what it will do.

The formal definition of entropy was developed in the late 1940's by C. E. Shannon. Given a *source alphabet* S (a set of n symbols $\{s_i\}$) produced by a message source which is characterized by the corresponding symbol probabilities $P = \{p_i\}$ (where $\sum p_i = 1$), the entropy of the source is defined as

$$H_r(S) = \sum p_i \log_r (1 / p_i)$$

Typically, if the radix *r* is not defined (only *H*(*S*) is given), it is assumed to be two; this indicates that the information content is measured in bits.

A related theorem, the *fundamental source-coding theorem*, also introduced by Shannon, states that the average length of any encoding of the symbols of *S* cannot exceed the entropy of *S*. That is, if a set of codes $\{C_i\}$ with lengths $\{l_i\}$ (measured in bits) is assigned to replace the symbols of *S*, the resulting average length $\sum l_i \, p_i$ will asymptotically approach *H*(*S*), but cannot exceed it. The theoretical maximum efficiency of any *first-order* encoding of a source (an encoding that assigns a code to each source symbol) is thus the entropy of that source divided by the length of the original source symbols (in bits).

Common Data Compression Algorithms

Huffman Coding

There are several data compression techniques which are widely used today. One of the oldest and most widespread is Huffman coding, introduced by D. A. Huffman in 1952 ([HUFFMAN]). It is a fairly straightforward statistical coding scheme in which the probabilities of each of the possible symbols in *S* are determined, and output codes of varying bit length are assigned to those symbols such that frequently used symbols are represented by shorter codes than those assigned to less common symbols. For example, in typical English text the letters **e** and **t** appear quite frequently, while **q** and **z** are seldom used. If each character in a text message is represented using the standard ASCII binary code, seven bits are required for each character. However, a Huffman code might assign two-bit codes to **e** and **t** and ten-bit codes to **q** and **z**, so that the overall length of the encoded message will be shorter than that of the original.

More precisely, given an alphabet S of n symbols, a simple binary code would require $\lceil \log_2 n \rceil$ bits to represent each symbol. If the corresponding symbol probabilities P are known, the Huffman algorithm will assign a set of codes $\{C_i\}$ to the symbols of S such that the resulting average length is less than $\lceil \log_2 n \rceil$. Any message D from S whose symbol distribution (the number of occurrences of each s_i divided by the total number of symbols in D) is roughly the same as P can thus be encoded using fewer bits than the binary coding of D. However, if the symbol distribution is much different from P, the encoded message may actually expand. For instance, in the English text example above, a block of 100 **q**s would require 700 bits using the ASCII code, but would need 1000 bits using the Huffman code. For details on the construction of a Huffman code given P, see [HUFFMAN], [STORER88] (p. 39), or [BELL] (pp. 105-107), among others.

Implementations of the Huffman algorithm can be either static or adaptive. If the characteristics of the data to be compressed are well known in advance, a static implementation will perform well. For instance, if only typical English text is of interest, it is possible to generate a code based on published standard character distributions. Since a great deal of research has been devoted to determining the characteristics of English text, these distributions are fairly accurate (see [STORER88], Appendix A.1 for distribution tables and further references). It has been shown that if the message to be compressed does match the probability distributions used to generate the codes, the Huffman code is optimal; that is, no other first-order technique which will produce better average compression ([HUFFMAN]). In fact, if the source symbol probabilities are all integral powers of one half, the average length of the Huffman code is equal to the entropy of the source. However, if the symbol distribution is unknown, or if messages with a variety of characteristics are to be compressed, an adaptive method must be used.

There are two means of making Huffman coding adaptive. The first and simplest is to scan the data once to determine the symbol distributions, then to build the code and encode the data during a second scan. This greatly reduces the usefulness of the algorithm, since it can no longer be used for stream-oriented data (such as data passing through a modem or a streaming tape controller). In addition, since the decompressor does not know the symbol probabilities in advance to generate the code, the compressor must transmit the code along with the message, decreasing the compression efficiency. Another method is to begin with a standard symbol distribution and corresponding code, and to update the distribution as the message is processed. Thus, after each source symbol or block of symbols is processed and the distribution updated, the code is regenerated. The decompressor can now update the code in the same order that the compressor does, without requiring the transmission of the code with the encoded message. However, significant overhead is required for both the compressor and decompressor to regenerate the code at specified intervals. For more details on this adaptive technique and other improvements to the basic Huffman algorithm, see [STORER88], pp. 40-46.

Arithmetic Coding

Another variable-length coding scheme which is becoming increasingly popular is arithmetic coding. Arithmetic codes have been studied by many people, with much of the initial research into practical implementations for data compression purposes conducted by Jorma Rissanen and Glen G. Langdon Jr. ([RISSANEN], [LANGDON82], [LANGDON84]). This research was later refined and a straightforward algorithm for the implementation of arithmetic coding presented by Ian H. Witten, Radford M. Neal, and John G. Cleary in 1987 ([WITTEN]). Arithmetic coding is currently one of the most active topics in data compression.

The principal behind arithmetic coding is the mapping of any source message onto the real numbers in the interval [0, 1). As the input message becomes longer, the portion of this interval that it represents narrows, and more bits are required to represent it. The interval is reduced as each symbol from the source message is processed according to the probability of occurrence of that symbol (the $\{p_i\}$ described above), the fundamental idea being that high-probability symbols will narrow the interval less than low-probability symbols, so fewer bits will be required to represent that reduction.

One of the primary advantages of arithmetic coding is that it very clearly separates the compression mechanism into an *encoder*, which accepts an event (typically an input symbol) and its associated probability information and produces a compressed data stream, and a *modeler*, which accepts the input symbols and produces corresponding events and their probabilities. Static and adaptive modelers are discussed in depth in [RISSANEN], [WITTEN], [ABRAHAM], [BELL], and [KWAN], among others. Kwan shows that the LZW algorithm (to be discussed in the next chapter) can be represented as a model for an arithmetic encoder, although its execution is very slow ([KWAN]).

The event and probability outputs from the modeler could be encoded using either a Huffman or an arithmetic encoder. As was stated in the previous section, the Huffman encoder is frequently described as producing the optimal coding, given a set of probabilities. An argument against this assumption is given by Witten, et. al.

> A message can be coded with respect to a model using either Huffman or arithmetic coding. The former method is frequently advocated as the best possible technique for reducing the encoded data rate. It is not. Given that each symbol in the alphabet must translate into an integral number of bits in the encoding, Huffman coding indeed achieves "minimum redundancy". In other words, it performs optimally if all symbol probabilities are integral powers of ½. But this is not normally the case in practice; indeed, Huffman coding can take up to one extra bit per symbol. The worst case is realized by a source in which one symbol has probability approaching unity. Symbols emanating from such a source convey negligible information on average, but require at least one bit to transmit. Arithmetic coding dispenses with the restriction that each symbol must translate into an integral number of bits, thereby coding more efficiently. [WITTEN]

The separation of the compressor into a modeler and an encoder is advantageous because the encoder can be constructed to produce a compressed message of the minimum possible length given the probabilities from the modeler, and attention can then be turned to perfecting one or more modelers to handle various input messages. Details of arithmetic encoders are given in [LANGDON84], [WITTEN], and [BELL] (Chapter 5). One particular encoder of interest is the **binary arithmetic coder** (BAC) described in [LANGDON82] and [LANGDON84], which is designed to encode a binary source; i.e. the source alphabet is {0, 1}. It is a relatively straightforward algorithm, and could be implemented in hardware rather easily, requiring relatively simple logic (only two registers and an integer ALU). This assumes that an appropriate modeler or set of modelers could be designed and implemented as well.

Dictionary-Based Algorithms

The principal idea behind dictionary encoding compression schemes is the replacement of strings of input symbols by references to previous occurrences of those strings. If the number of bits required to represent these references is shorter than the average length of repeated strings, compression can be achieved. There are two major classes of these methods, both proposed by Jacob Ziv and Abraham Lempel.

The first scheme was introduced in 1977, and is commonly referred to as LZ77 ([ZIV77]). The algorithm keeps the last *n* input symbols in a buffer, effectively sliding an *n*-symbol *window* over the input data. When a string of symbols is encountered that has occurred previously in this window, it is encoded as a pair of values corresponding to the string's position in the window and its length. The description of the method given in [ZIV77] is highly theoretical, and a usable algorithm implementing it, commonly referred to as LZSS, was presented by James Storer and Thomas Szymanski in 1978 (see [STORER82], [STORER88], Chapter 3, and [BELL], Chapter 8 for details). Several variants on this technique have been proposed, creating a family of algorithms, each reflecting different decisions in the implementation of the algorithm. Many common enhancements include the use of some sort of statistical coding (dynamic Huffman, Shannon-Fano, etc.) to further compress the (position, length) pairs.

The second scheme was introduced in 1978, and is commonly referred to as LZ78 ([ZIV78]). This technique is based on the construction of a table or dictionary of symbol strings encountered in the input. When a string is encountered subsequently, the corresponding dictionary index is transmitted instead of the string. Again, Ziv and Lempel's presentation is highly theoretical, and another practical algorithm implementing it was developed by Terry Welch in 1984 ([WELCH]). This algorithm will be dealt with in detail in the remainder of this paper. Again, a number of modifications have since been proposed, resulting in another family of algorithms.

Both types of LZ algorithms are fairly simple to implement in software and are amenable to hardware implementations. They are both adaptive with only one pass over the data. Unlike the variable-length coding schemes, the modelling and encoding functions are not cleanly separated. For some types of input, the LZ algorithms' compression efficiency is very good, and over a range of data characteristics they perform reasonably well, but they cannot be expected to match the performance of arithmetic codes over a wide range of input data. Their choice as a compression algorithm would be a tradeoff between compression efficiency and ease of implementation.

Other Data Compression Algorithms

The text compression techniques listed above are probably the most popular and widely used today, but there are a great number of other methods available. These range from other variable-length coding schemes, such as Shannon-Fano codes, and dictionary-based schemes, such as splay trees, to continuing enhancements of existing algorithms, such as the Q-coder, an implementation of arithmetic coding, and LZRW, a highly optimized variant of LZ77 [WILLIAMS]. The interested reader is referred to [BELL] for a good overview of various text compression methods, and an excellent bibliography of related information.

Choosing an Algorithm for Hardware Implementation

Software to perform various text compression algorithms is readily available on a variety of computer platforms. From the original public-domain compression programs *SQ* and *USQ* (squeeze and unsqueeze) for CP/M-based machines to today's sophisticated compression and archival utilities, such as *PKZIP*, *ARC*, *LHARC*, *ZOO*, etc., for the MS-DOS operating system and *compress* for UNIX and its derivatives, data compression software is a commonly accepted and widely used feature of many computer systems [VAUGHAN]. However, more demanding applications, such as high-speed data communications networks, streaming tape controllers, and disk drive interfaces, require levels of performance that probably cannot be provided by the execution of software on a general-purpose processor, short of the dedicated use of today's extremely fast RISC processors. The solution to this problem is the direct implementation of appropriate compression and decompression algorithms as VLSI circuits.

There are very few commercially available integrated circuits which perform data compression. Two which are currently available achieve data rates approaching two million bytes per second; one uses an undisclosed compression algorithm [INFOCHIP], and the other uses an LZ77 variant [STAC]. Another slightly faster chip utilizes a modified LZW technique known as DCLZ, and approaches compression rates of 2.5 million bytes per second [AHA]. The compression ratio of all three chips is in the vicinity of two to one for a wide range of input data types, which is probably acceptable. While these data rates are far faster than most software implementations, they are not sufficient to meet the demands of many high speed applications. There is thus a great deal of motivation for research into alternative circuits.

The algorithm chosen for implementation should have relatively limited complexity and should not require an exorbitant amount of computing resources. It should also be appropriate for the target applications; a lossless technique is certainly required, and it must be adaptive as well, since the characteristics of the input data will be totally unknown. Several of the applications are stream-oriented, so a single-pass algorithm is necessary. The implementation must be able to achieve data rates substantially greater than one million bytes per second, and should deliver compression ratio of around 2.0 or better for most input data.

Of the algorithms discussed, the most likely choices are arithmetic coding and one of the dictionary-based algorithms. The implementation of an arithmetic encoder would be reasonably simple and should provide the desired performance, but then the issue of choosing a modeler arises. Of the dictionary-based techniques, LZW appears to be one of the most amenable to a high-speed hardware implementation (since it was designed with that in mind). This is the algorithm that has been chosen for a VLSI implementation. The remainder of this paper will describe the design of this integrated circuit and the methodology used in that design.

CHAPTER 2

LZW COMPRESSION AND DECOMPRESSION ALGORITHMS

As described briefly in the previous chapter, LZW is a variation on a dictionary encoding method proposed by Ziv and Lempel in 1978, known as LZ78. Terry Welch presented the technique in 1984 as a realization of the algorithm which was suitable to hardware implementation, for application in high-speed disk controllers ([WELCH]). Since that time it has become one of the most well-known text compression algorithms, due primarily to its robustness and simplicity. It is used in the UNIX *compress* utility and the MS-DOS *PKZIP* and *ARC* archival utilities and is widely approved as a reasonable software compression method. This wide-spread familiarity should provide for relatively easy acceptance of hardware implementing the algorithm.

The Basic LZ78 Compression Algorithm

The basic principle of LZ78's operation is that the stream of input symbols is parsed into strings, where each string consists of the longest matching string seen thus far in the previous input plus the one symbol that makes it different from prior strings. Each of these strings is then added to a *dictionary* and coded as the index of the previous, or *prefix*, string plus the normal binary representation of the extra symbol. One prefix code is reserved as a **null** (zero-length) string, for transmission before new input symbols. The output stream from the compressor thus alternates between prefix codes and symbols from the input. This process of parsing the input and adding entries continues until the dictionary is full, at which time it is reset and started over again.

Note that as the number of dictionary entries grows, the number of bits required to represent each prefix increases as well. This can be handled in one of two ways. Since the maximum size of the dictionary, N, is known in advance, each prefix or index can just be represented using $\lceil \log_2 N \rceil$ bits. Alternatively, after p strings have been added to the dictionary, the index can be represented using $\lceil \log_2 p \rceil$ bits, and this will increase to $\lceil \log_2 N \rceil$ as the dictionary fills.

The decompression scheme is very simple. The decompressor begins with an empty dictionary, just as the compressor did. As it receives (prefix, symbol) pairs from the input stream, it can recreate a dictionary which will be an exact image of the one used by the compressor when it generated that pair. Each (prefix, symbol) pair can then be expanded into the full string of symbols using the dictionary entry for that prefix code.

One of the advantages of the LZ78 family of algorithms is that it is unnecessary to know or estimate any of the a priori symbol probabilities. Another advantage is its quick adaptability to any kind of input, as long as it contains repeating strings of symbols. Also, it has been proven that if the input text is generated by a stationary, ergodic source, compression is asymptotically optimal as the length of the input increases ([ZIV78], Theorem 4). A source is ergodic if any sequence which it produces becomes entirely representative of the source characteristics as the sequence grows longer, so it would appear that LZ78 should be an ideal compression method for all messages generated by such sources. The drawback is that, while it is asymptotically optimal, it converges to this limit relatively slowly. Short inputs compress very poorly, if at all. The reason for LZ78's popularity is not for its compression efficiency, but for the efficient means by which several of its variants can be implemented.

The LZW Algorithms

Since Welch's goal was a compression algorithm which could be used in the channel between a computer and a disk drive, where high speed is essential, LZW was derived from LZ78 to be as fast as possible. The first modification is the elimination of the alternating prefix codes and input symbols. By initializing the dictionary to contain all the single-symbol strings from the source alphabet, it is possible to transmit each string as just a prefix code. Strings are parsed as before into the maximum length prefix and a terminating symbol, but rather than transmitting this symbol, it is instead encoded as the first symbol of the next string. Another modification is fixing the length of the dictionary at a power of two and using fixed-length prefix codes. These two modifications together greatly simplify the generation and processing of the coded data stream.

The other major specification of the LZW algorithm is the means by which the dictionary is represented. The initial presentation did not specify the means by which strings would be stored so that the input could be parsed into prefixes. One possibility that allows for relatively efficient parsing is the storage of all strings in a *trie* data structure. A trie is just a multiway tree (each node can have up to *n* children, where *n* is the number of symbols in the source alphabet) where each branch from a node is labelled with a symbol from the source alphabet, and each node represents the string obtained by traversing the trie from the root to that node. For example, the trie shown in Figure 1 contains the strings "a", "b", "c", "aa", "ba", and "bc".

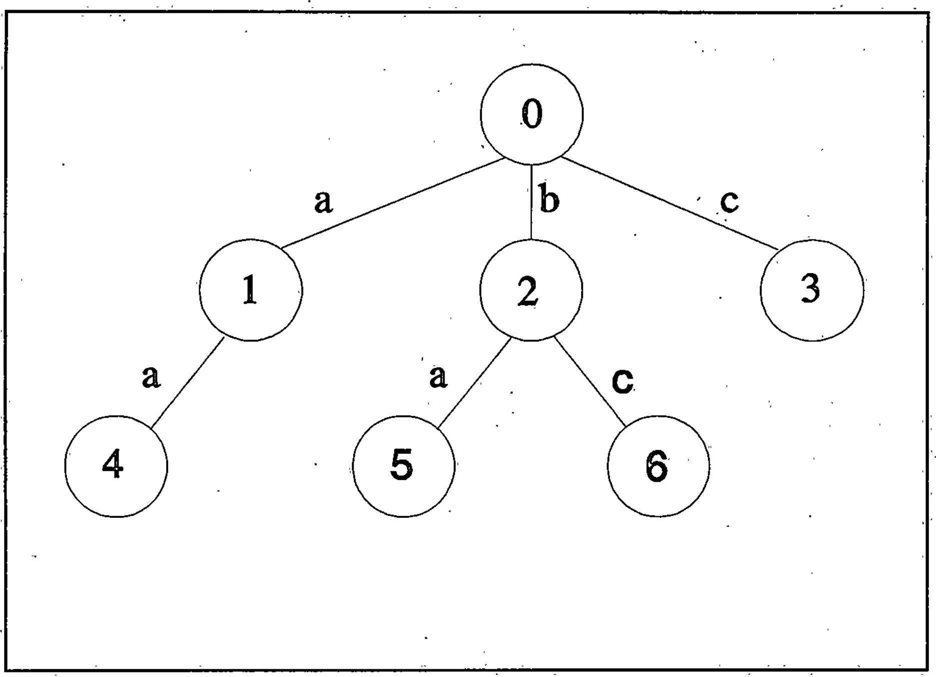

Figure 1 - Example of a Dictionary Trie

If each node (except the root, which represents the **null** string) is labelled with the index of the corresponding string's dictionary entry, parsing input data is very straightforward. The trie is traversed from the root, following the branch labelled with the next input symbol, until a required branch cannot be located. The index stored in the last node is output as the prefix; a new node is added to the trie, labelled with the next available dictionary index, and connected to the last node by a branch labelled with the last input symbol; and the traversal is begun again from the root, using the last input symbol to traverse the first branch. For example, if the trie shown in Figure 1 has been constructed and the input string "bca" is being parsed from the input, after the symbol "a" has been read, the prefix code 6 would be output, the string "bca" would be added to the dictionary as entry 7, the trie would be modified as shown in Figure 2, and the symbol "a" would be used to traverse the trie to node 1. Parsing of the input stream would continue from this point.

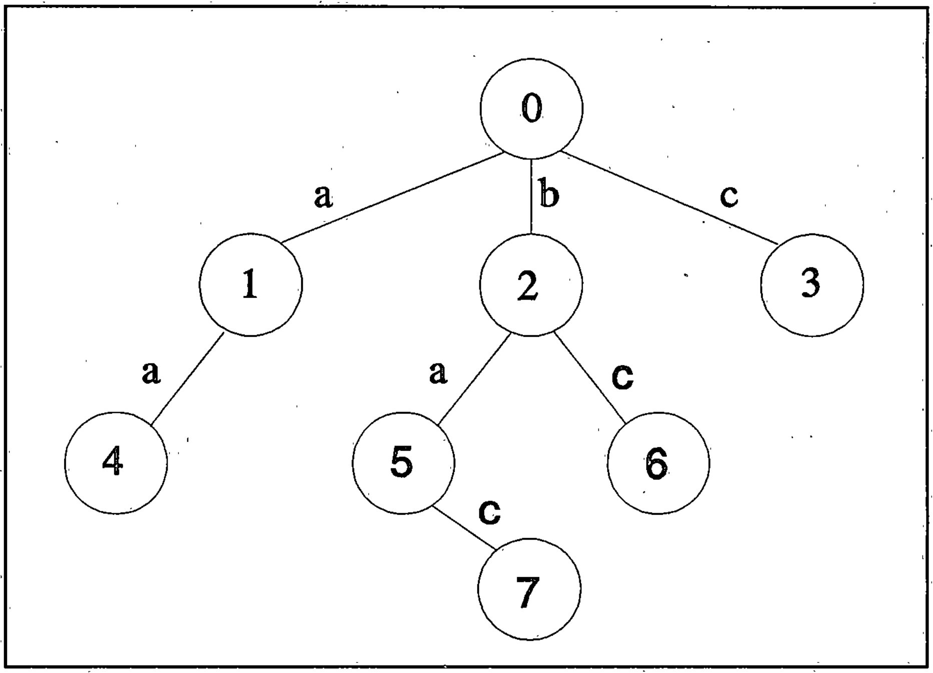

Figure 2 - Modified Dictionary Trie

This is a fairly effective means of parsing the input into strings, but it is not sufficient for regenerating strings given their prefix codes. In order to allow this, some means would be required to locate a node in the trie given a prefix code, then to traverse the tree upward from that node to the root to accumulate the symbols in the string. Note that the symbols from the string would be encountered in reverse order, so it would be necessary to provide some means of reversing them.

The approach implemented by LZW uses a similar idea. The dictionary is stored as a fixed-length string table, where each entry represents a string from the dictionary and consists of the index of the prefix for the string and the last symbol of the string. One index can be reserved to represent a **null** prefix. For example, the dictionary represented by the trie in Figure 2 would be represented as shown in Figure 3 on the next page, if entry 0 in the table is unused and that index is reserved as the **null** prefix.

Entry Index	Last Symbol	Prefix String
1	a	0
2	b	0
3	c	0
4	a	1
5	a	2
6	c	2
7	a	6

Figure 3 - String Table Representation of a Dictionary

This scheme allows symbol strings to be easily generated from compression codes by simply using the codes as indices into the string table and following the prefix codes until a **null** prefix is encountered, accumulating the last symbol of each entry. This also generates the string in backward order, as discussed above, so a string reversal mechanism is still required. However, parsing input symbols is no longer as straightforward as the trie implementation. Given the index of a prefix string and the next input symbol, it is no longer immediately apparent what the index of that dictionary entry would be. This problem is addressed in the following section on the detailed compression algorithm.

Compression Algorithm

The basic LZW algorithm requires a register, *Omega*, to hold the code representing the accumulated prefix string. For a dictionary (typically referred to as the string table) containing N entries, this *Omega* must be $\lceil \log_2 N \rceil$ bits in length. Another register, K, is used to hold the next symbol from the input data stream. The algorithm assumes there is a simple method to empty out the string table and initialize it to contain only single-symbol strings. The pseudo-code description of the algorithm is shown in Figure 4, where the + operator represents string concatenation.

```
Initialize string table to contain all single-symbol strings
Prefix code Omega ← index of single-symbol string formed by first input symbol
While more input data available
        K ← next input symbol
        If string Omega + K is in string table
                Omega ← index of string Omega + K
        Else
                Output prefix code Omega
                Omega ← index of single-symbol string K
                Add string Omega + K into string table
        End else
End while
Output code for last accumulated string, Omega
```

Figure 4 - LZW Compression Algorithm

An example of the execution of this algorithm is shown in Figure 5. The source alphabet for this example is {a, b, c}, so the string table initially contains these strings in entries 1, 2, and 3. *Omega* is initially set to 1 (the string for the single symbol 'a'). When the first 'b' is encountered, and entry '1 b' is not found in the table, code 1 is output, code '1 b' is added to the first empty table location, and *Omega* is reset to 2 (the string for the single symbol 'b').

Input Symbols	a	b	c	a	b	c	a	c	a	c	b	
Output Codes		1	2	3		4		6			8	2
New Table		4		6				8				
Entries			5			7					9	

Final String Table:

Entry Number	1	2	3	4	5	6	7	8	9
Last Symbol	a	b	c	b	c	a	c	c	b
Prefix	0	0	0	1	2	3	4	6	8

Figure 5 - Compression Example

The algorithm is very simple, with an almost trivial implementation. Each string table entry will contain an *Omega* prefix value and a symbol *K*, so adding a string to the table is simply a matter of keeping track of the next unused entry, and writing the values of *Omega* and *K* into that entry when a string search is unsuccessful.

It can be seen that the only non-trivial portion of this algorithm is the search for strings *Omega* + *K* in the string table. A simple means of locating strings would be a linear search of the table, but this would slow compression unacceptably. Instead, typical software implementations of LZW use hashing techniques to index the table, rather than indexing it directly with the prefix code (most textbooks on algorithms contain a presentation of the use of hashing techniques to search tables; for a very detailed description, see [KNUTH], section 6.4). Determining a suitable hash function typically requires a great deal of experimentation, and most hashing techniques require that the string table have extra unused entries in order to function effectively. The computational overhead of access to the string table using hashing is the primary bottleneck in software implementations. Possible alternatives to hash functions are discussed in the subsection on adaptation of the algorithm for hardware implementation.

The only unresolved issue involves handling the string table when all entries have been filled. The initial development of LZW did not address this problem, assuming that once the table was full, it would be frozen and new strings would just be discarded. This can severely impact the compression efficiency if the redundancy characteristics of the data change after the table is frozen. Several of the LZ78 variants attempt to handle the full table in a more reasonable manner to improve compression. The algorithm used in the UNIX *compress* utility, known as LZC, monitors the compression ratio, and if it begins to deteriorate (decrease), the string table is reset to contain only single-symbol strings before compression continues. Other implementations attempt even more sophisticated management of the table. The most popular technique is Least Recently Used (LRU) replacement of strings when the table becomes full. As the name suggests, the entry in the table which has been accessed least recently is discarded and overwritten with a new string as required. This will improve the compression ratio, but at the cost of substantially complicated string table manipulation. A practical adaptation of LZW which performs LRU-type table management is described in [BUNTON].

Another possible enhancement to improve the compression ratio is to reintroduce the variable-length output codes originally proposed in LZ78. LZC uses this technique, incrementing the number of bits n in the output codes whenever the number of entries in the string table exceeds 2^n. Other authors have suggested using arithmetic coding (see [PERL] for one description of cascading LZW and arithmetic coders) or similar phased-in binary codes (as described in [HORSPOOL]) to relax the requirement that the number of bits per code be an integer.

The implementation described in this paper just uses the dictionary freeze technique and fixed-length output codes, in order to maintain simplicity. The addition of various table management and output coding techniques to the basic compressor and their benefits could be examined in the future.

Decompression Algorithm

The LZW decompressor utilizes the same string table as the compressor, including the capability of initializing it to contain only single-symbol strings. It uses the same *K* register, along with three code registers of the same length as the *Omega* register used in the compression algorithm. The basic decompression algorithm is shown in Figure 6.

```
Initialize string table to contain all single-symbol strings
OldCode ← first input code
K ← StringTable [OldCode].LastSymbol
Output K
While more input data available
        InCode ← Code ← next input code
        While StringTable [Code].Prefix ≠NULL
                K ← StringTable [Code].LastSymbol
                Output K
                Code ← StringTable [Code].Prefix
        End while
        K ← StringTable [Code].LastSymbol
        Output K
        Add string OldCode + K to string table
        OldCode ← InCode
End while
```

Figure 6 - Basic LZW Decompression Algorithm

As noted by Welch, there are two basic problems with this algorithm: the first is that output symbols are produced in reverse order, and the second is that there is a special input case in which the compressor will output a code which the decompressor will not have in its string table when it is encountered. The first problem is easily addressed by pushing the symbols onto a stack, then popping them off when the end of the string is reached. However, this requires that the decompressor halt while the string is being removed from the stack, and this becomes the decompression bottleneck. An alternate string reversal scheme is presented in the next section. Note that, since the stack (or any string reversal mechanism) will have finite capacity, it is necessary to limit the length of strings to be reversed. That is, the compressor must be constrained to accumulate symbols into strings only up to some maximum length.

The abnormal input condition occurs because, although the decompressor is creating a string table identical to the compressor's, it is doing it one step behind the compressor. The problem arises if an input of the form $K\omega K\omega KL$ is encountered, where K and L are single input symbols, ω is a string, and $K\omega$ is already in the string table. When the compressor encounters the second K, it will send the code for $K\omega$, add the string $K\omega K$ to the string table, and start over with the string K. It will then parse the input until it comes to L, at which time it will send the code for $K\omega K$, which was the last one added to the table. When the decompressor receives this code, it will not yet be in the string table. However, the only strings which cause this problem are of the form shown, where the second string is just a one-symbol extension of the first string, and this symbol is identical to the first symbol of the string. In that case, if the decompressor encounters a code that is not in the string table yet, it knows the last symbol must be the same as the first symbol of the previous string, and the remainder of the string is identical to the previous string. Thus, if the first symbol of each string produced (the last symbol to be reversed) is stored, and the previous string is available in the *OldCode* register, the input code can be replaced by this combination and decompressed. The modified algorithm is shown in Figure 7, with the addition of the *FirstSymbol* register to hold the initial symbol in each string and commands to reverse the string using a simple stack.

Note that Welch's presentation of this algorithm is incorrect; he does not correctly handle the case where the input code is not in the string table yet. He writes the final character (*FinChar*) of the last string out, instead of pushing it on the stack, and then tries to look up the missing string in the table. The following algorithm rectifies these problems.

```
Initialize string table to contain all single-symbol strings
OldCode ← first input code
FirstSymbol ← K ← StringTable [OldCode].LastSymbol
Output K
While more input data available
        InCode ← Code ← next input code
        If Code not in string table
                Push FirstSymbol on stack
                Code ← OldCode
        End if
        While StringTable [Code].Prefix ≠ NULL
                K ← StringTable [Code].LastSymbol
                Push K on stack
                Code ← StringTable [Code].Prefix
        End while
        FirstSymbol ← K ← StringTable [Code].LastSymbol
        Output K
        While stack not empty
                Pop top symbol off stack and output
        End while
        Add string OldCode + K to string table
        OldCode ← InCode
End while
```

Figure 7 - Correct LZW Decompression Algorithm

Note that, although the decompressor uses the same string table as the compressor, it does not need to search the table for strings, so the hashing function is not needed. Since the decompressor is essentially only using the string table as a RAM to retrieve pointer chains, the decompression process is typically much faster than compression in software implementations.

An example of decompression is shown in Figure 8, for the codes generated in Figure 5. Note that when code 8 is encountered, it is not in the table, so the last symbol of the previous string ('c'), contained in *FinChar*, is pushed as the final symbol of the string and the last input code, 6, contained in *OldCode*, is substituted for code 8 and traced backward to produce the string. When the string has been reversed, code 8 is added to the table just as it should be. Also note that the string table is identical to that produced by the compressor, after the entire input has been processed.

Input Codes	1	2	3	4	6	8	2
	↓	↓	↓	↓	↓	↓	↓
	a	b	c	1 b	3 a	6 c	b
				↓	↓	↓	
				a	c	3 a	
						↓	
						c	
Output Symbols	a	b	c	ab	ca	cac	b
New Table Entries							
Index		4	5	6	7	8	9
Last Symbol		b	c	a	c	b	b
Prefix		1	2	3	4	6	8

Figure 8 - Decompression Example

Adaptation for High-Speed Hardware Implementation

The LZW compression and decompression algorithms as presented are very amenable to software implementation. However, the performance of a hardware implementation can be increased by utilizing specialized circuits to perform the critical functions which decrease the algorithms' performance. These problem areas are the string table search in the compressor and the string reversal in the decompressor.

<u>Compressor String Table Search</u>

The string table search is ideally suited to the use of a *content-addressable memory* (CAM), also known as an *associative memory*. Rather than reading a CAM like a normal RAM, by specifying an address and retrieving the data stored there, it is possible to specify a data pattern for which to search the entire memory; a match signal is generated if the pattern occurs in the CAM, and if a match is found, the address of the matching memory location can be returned. This is a completely parallel operation, so the entire CAM can be searched in roughly the amount of time it would take to read one memory location from a normal RAM. This totally eliminates the compressor bottleneck.

One difficulty which may be encountered in CAM searches is multiple occurrences of the search pattern in the memory. If this happens, it is necessary to develop some sort of scheme to decide which of the matching addresses will be reported. However, examination of the compression algorithm reveals that string table entries are guaranteed to be unique, so this is not an issue for the LZW application. However, one requirement is that the CAM also perform as a standard RAM, so the decompressor can use it for string table storage as well. The next chapter discusses the CAM requirements and capabilities in greater detail.

Decompressor String Reversal

In order to maintain decompressor operation at as constant a level as possible, it is necessary to allow it to produce symbols of a string to be reversed at the same time that the stack mechanism is reversing the previous string and writing symbols to the output. This could be achieved by using two stacks instead of just one. Thus, while one stack is being filled with a string to reverse, the other stack can be reversing the previous string. However, this does not fully alleviate the problem of the decompressor being required to wait for the stack. For example, if an input code representing a long string is followed by a series of codes representing short strings, the decompressor will be able to push the long string onto one stack and the short string onto the other, but it must then wait until the long string is fully reversed before a stack is available to hold the next string. Some advantage is gained from having both "stacks", but the problem is not completely solved.

This situation can be eliminated by using a single ring buffer to hold a series of strings and maintaining a set of pointers to the start and end of those strings. Providing the ring buffer is large and there are enough pairs of pointers available, the decompressor should not have to wait on the reversal mechanism. This string reversal scheme is discussed in more detail in Chapter 4.

Hardware Design Requirements

The background on LZW data compression has been presented in previous sections, along with a number of design options. In order to maintain simplicity and overall system speed, the VLSI implementation which is presented in the remainder of this document will be designed to meet the following specifications. This is not a comprehensive list of design issues, but rather a set of constraints placed on the implementation.

The input alphabet will be the full set of 256 eight-bit bytes.

The number of system clock cycles required by the compressor to process each input symbol and by the decompressor to produce each output symbol must be minimized.

The dictionary or string table will contain 4096 entries. The compressed codes will thus be 12 bits in length. It will be implemented using a content-addressable memory to increase compression speed. This will require a 4096 word by 20 bit CAM. Additionally, some mechanism must be provided for easily initializing the first 256 CAM entries to hold single-symbol strings.

Code 4095 (hexadecimal fff) will be reserved for the **null** string code. That string table entry will therefore be unused.

In order to simplify input and output buffering, fixed length codes will be used; the length of codes generated by the compressor will not increase as the string table is filled. When the string table is filled, it will be frozen, with new entries generated after that time being discarded.

The decompressor will write decompressed bytes to a string reversal mechanism, which will accumulate them until the end of the string is reached, at which time it will begin writing them to the output buffer in reverse order. It must provide the capability to write one byte into the reversal buffer and output another to the output buffer without forcing the decompressor to wait for buffer space.

The maximum string length the compressor will be allowed to accumulated will be 128 bytes, to limit the size of the reversal buffer required.

Byte and code input and output buffers will be provided to interface the circuit to external equipment. The code buffers will convert between a standard eight-bit data stream interface and the twelve-bit code stream used internally.

CHAPTER 3

CONTENT-ADDRESSABLE MEMORY

The content-addressable memory (CAM) plays a key role in both the compressor and decompressor implementation, since it will be used to hold the string table. Therefore, before the controller logic for those two modules can be designed, it is necessary to determine the functionality of the CAM and its interface characteristics.

Basic Content-Addressable Memory Characteristics

Typically, content-addressable memories are very similar to static random-access memories (RAMs). An *M*-word by *N*-bit CAM will have a $\log_2 M$-bit **address** bus, an *N*-bit bidirectional **data** bus, and **read** and **write** control signals, just as a RAM would. It provides normal random access data storage and retrieval functions asynchronously (without requiring a clock signal to control timing), and will store all data written to it as long as it is connected to a power supply (as opposed to a dynamic RAM, which requires that each memory location be periodically *refreshed*, by reading it and immediately rewriting the retrieved data, in order to maintain its contents). However, the CAM also has a **search** control signal, a **match** output signal, and a bidirectional **address** bus. Once data has been written to the CAM, the entire memory can be searched for a desired data word simply by placing the word on the **data** bus and asserting **search.** The search is conducted on all memory words in parallel, and if the data pattern is not found, the **match** signal is not asserted. However, if the pattern is contained in some location, **match** is asserted, and the address of the matching location is placed on the **address** bus.

In order to prevent unused memory locations from possibly generating erroneous matches, CAMs typically have an **empty** bit associated with each data word. This bit is set for all words when a **reset** control signal is asserted, and if it is set, the corresponding memory word is prevented from generating a match signal. When a word is written, the bit is reset, allowing it to be included in searches.

The possibility of multiple locations matching the search pattern poses a problem. It is necessary for the encoder that returns the matching location's address to decide which match to report, using some *priority encoding* scheme to choose between multiple locations. One simple method would be to just report the location with the smallest binary address; however, for certain applications, this might be undesirable. Fortunately, as mentioned in Chapter 2, the LZW string table contains only unique entries, so a search can never match more than one location.

A transistor-level description of RAM and CAM memory cells is beyond the scope of this presentation. Most texts on VLSI design include a section on memories; see for example [WESTE], section 8.5, or [SHOJI], sections 7.20 through 7.24, for a description of the different types of memory cells.

Commercially Available CAMs

General-purpose CAMs are not readily available. One of the few ICs currently available is the **Am99C10A**, manufactured by Advanced Micro

Devices. It is a 256 word by 48 bit CAM optimized for use in address decoding and bridging for Ethernet and FDDI local area network applications, where it can be used as an address filter. It provides all the functionality previously described, although it is a register-based interface; for example, to read a word from the CAM, the address is written to an address register in the CAM and a read command is written to a control register, then the requested data can be read from another register. It provides a single-cycle reset command to clear the contents of all 256 words simultaneously, and it includes a priority encoder using the simple scheme described above (choosing the match location with the smallest address). In addition, it includes a 48 bit **mask** register, which can be used to selectively disable certain bits from the search operation. Also, each word includes a **skip** bit in addition to the **empty** bit. When a search results in multiple matches, **skip** can be set for the matching word chosen by the priority encoder, and subsequent searches for the same data pattern will not match that location. This allows an application to find all of the matches in the CAM. Further details of the IC's functionality are given in [AMD].

This CAM provides the required functionality, although its interface would be difficult to deal with. However, the memory density is too small to be very useful for this application. It would require 16 ICs to hold the full 4096-entry string table, and only 20 of the 48 bits in each word would actually be used. Alternatively, each 48-bit word could be divided into two 24-bit words, effectively doubling the memory density. This would greatly complicate the controller logic, since each search would require masking off one half of each word, searching for the pattern, then masking off the other half and searching again, but it could be done to half the number of ICs required.

Even if only eight ICs are required to store the string table, this would substantially complicate the system design. In addition, they are very expensive; one distributor priced eight 100 ns ICs at over $60.00 each, and 16 at over $45.00 each. Prices for 70 ns ICs were $95.00 each for eight and $65.00 each for 16. Aside from the obvious expense, their access time is not fast enough, especially if they are split and searched in halves, to provide a significant speed increase over existing data compression ICs. The obvious alternative is a custom-built CAM designed to meet the requirements of this application.

Custom Dynamic CAM

It is desirable for the final circuit, including the string table, to fit into one IC. The primary area requirement would be for the CAM. The string table will require 80 Kbits (4096 words times 20 bits per word), which would be a fairly large memory, especially in view of the fact that CAM memory cells are larger than static RAM cells (a normal CMOS RAM cell requires six transistors, while the corresponding CMOS CAM cell requires nine).

In order to decrease area requirements, a dynamic CMOS CAM, currently being designed by Professor Kel Winters of Montana State University, will be used. The basic memory cell requires only six transistors, so the total area required by the CAM should be on the order of that required for an 80 Kbit static RAM. However, this introduces the necessity for periodic refresh of every memory location in the CAM.

The dynamic CAM (DCAM) will provide all the functionality described in the previous sections, including generation of the matching location's address on a successful search. It will not include any priority encoding mechanism for multiple matches, since this is guaranteed to not occur. It will contain 4096 20-bit words, organized into a roughly square array of M rows by N columns (where M and N are both powers of two).

The interface will consist of the following signals:

Inputs -	20-bit **InputData** bus, 12-bit **InputAddress** bus, **Read**, **Write**, **Search**, **Reset**, **Refresh**, **Compress**, **Clock**
Outputs -	20-bit **OutputData** bus, 12-bit **OutputAddress** bus, **Match**

The purpose of most of these signals is as described earlier. Note that there are separate input and output data and address busses; since the DCAM will be on the same IC as the controller, it is not necessary to consolidate them into bidirectional busses to save on interface connections. Enhancements to normal CAM operations made specifically for this application are detailed in the following subsections.

Synchronous Mode-sensitive Operation

Note the addition of the **Clock** and **Compress** control signals. Unlike typical memories, which are asynchronous devices, the DCAM is synchronous. The system clock controls the timing of all read, write, and search operations. Also, the fundamental operation of the DCAM is different during search operations than read operations. For reads, the internal data lines are precharged to the supply voltage during the first half of the clock cycle, then are forced to the values to be stored in the selected memory location during the second half. However, for searches the data lines are predischarged to the ground voltage during the first half of the clock cycle, then the search pattern is placed on them and the match signals are evaluated during the second half. For write operations, the data lines can be precharged to either a high or low voltage before they are set to the value to be written. Since the compressor performs only searches and writes, and the decompressor performs only reads and writes, the logic that precharges the data lines can be simplified if it is known ahead of time whether to precharge high (if the **Compress** signal is de-asserted) or precharge low (if **Compress** is asserted).

Predefined String Initialization

Rather than attempt to write the first 256 single-byte strings into the DCAM each time it is initialized, the special characteristics of these strings can be exploited to hard-wire them into the DCAM. Each string in locations 0 through 255 will consist of a single byte (with the same value as its address), followed by the **null** prefix code. These table entries will never be written, only read and searched. Therefore, rather than create CAM or even ROM words to hold them, simple combinational logic can be added to simulate the DCAM operation for these locations.

Specifically, whenever the address of a read operation is less than 256, the lower eight bits of the address can be returned as the data value. Likewise, whenever a search is requested for an entry containing the **null** prefix code, the search is automatically successful, and the data byte from the search pattern is padded out with zeroes and returned as the matching address. Writes to addresses less than 256 should never occur, but should be ignored in any event. Adding this logic greatly simplifies the initialization process (table resets now involve only marking actual DCAM words as empty), and eliminates 256 words from the DCAM.

Reset Operation

Assertion of the **Reset** signal will cause the DCAM to reset all words to empty status as described above. It has not yet been determined how memory cells will be marked empty. The two possibilities are the inclusion of an empty bit in each word, which would just be a static latch which could be set by the **Reset** signal and reset by a write operation, or the use of a data pattern that will not occur in any search operation to fill all memory locations. Examination of the LZW algorithm reveals that no string table entry after the first 256 hard-wired single-byte strings will contain the **null** prefix code (hexadecimal value FFF), and the logic emulating the first 256 entries will intercept any search operation containing the **null** prefix anyway. This pattern can therefore be written into all memory locations to prevent them from erroneously matching a search. Note that this requires the logic for "searching" the first 256 entries to not only provide the match signal and address, but to also inhibit the normal search on the remainder of the DCAM, since this search could potentially match an empty location.

If the latter mechanism is used, it might not be possible to mark every location in the entire DCAM empty in one clock cycle. The controller logic design assumes that it will be possible; if this is not the case, it will be necessary to add a short state sequence during the reset process to allow the DCAM time to finish the operation.

Refresh Operation

The DCAM will be refreshed in much the same way as a standard dynamic RAM. Periodically, the control logic should halt the normal flow of operations and generate a refresh cycle. This will consist of placing the address of a row of the DCAM to be refreshed on the **InputAddress** bus and asserting the **Refresh** signal; during this cycle, the **Read**, **Write**, and **Search** signals should not be asserted, and the value on the **InputData** bus will be ignored. The DCAM will refresh the contents of every bit in that row simultaneously. In addition to regenerating cell contents during the refresh cycle, the capability will also be provided to refresh an entire row whenever a word in that row is written.

Note that although the **InputAddress** bus is 12 bits wide, the number of bits actually required to address a row of the DCAM is $\log_2 M$, where M is the number of rows in the DCAM. Therefore, the control logic should cycle through each of the possible values from 0 to $M - 1$ repetitively, padding them out $\log_2 N$ zeros (where N is the number of words per row) as the least significant bits to generate a 12-bit address. The DCAM will ignore the low-order $\log_2 N$ bits of the address during refresh. It will also ignore refreshes for the rows containing the first 256 hard-wired table entries (since N is a power of two, there will be no rows which contain both hard-wired entries and regular CAM words).

Possible Performance Enhancements

The design of the DCAM has not been completed yet, but the basic DCAM cell has been stabilized. Reza Massarat has run extensive SPICE simulations of the cell, and initial indications are that it can generate the match signal on a word level in less than 10 nsec, and read and write operations are faster. Assuming that the word matches can be accumulated and encoded to form the match address in a similar amount of time, the DCAM should be capable of continuous search operation at 20 MHz (since the evaluation is done in only half the clock cycle, with the other half being used to precharge the data lines).

A concern expressed by Professor Winters during the design is the difficulty of performing a write operation on the clock cycle immediately following a search at the clock frequency required. The procedure followed by the compressor would be to request a search for a string during one clock cycle, then to write a new table entry on the following clock cycle if the search was unsuccessful. The design of the DCAM is such that it may not be possible to get the new write address decoded and the data to be written onto the data lines in time to actually write the addressed word if the DCAM is recovering from a search operation.

To eliminate this problem, Professor Winters proposed the following modification to the search operation: when a search is requested, the address of the next unused table entry is also placed on the **InputAddress** bus, and the search pattern is automatically written to that location while the remainder of the DCAM is being searched. The compressor now needs to decide only whether to update the location of the next unused table entry (if the search failed) or to maintain the current value. It may be necessary to decrease the clock frequency to allow the DCAM time to complete this operation, but the compressor now only requires one clock cycle to process an input byte, rather than two, so the net result is faster operation. It has not yet been determined whether the DCAM will implement this proposed scheme. The controller design will be separated into two distinct paths in order to accommodate either option. Both are described in detail in the following chapters.

CHAPTER 4

STRING REVERSAL MECHANISM

Another key block of the complete IC that must be defined before the controller logic can be designed is the string reversal mechanism. Like the DCAM, much of this hardware module will be a custom design (as compared to the controller modules, which will be generated using logic synthesis tools). The throughput of the decompressor will rely heavily on this module's capability to simultaneously accept strings to be reversed from the decompressor and write reversed strings to the output buffer without halting the decompressor.

String Reversal Algorithm

As mentioned in Chapter 2, typical implementations of the LZW algorithm use a simple stack to reverse the output strings. Obviously, once a string has been pushed onto the stack, the decompressor must wait until that string has been completely popped off and written to the output before it can begin pushing the next string. This problem can be alleviated somewhat by using a dual-ended stack. That is, a string can be pushed onto the stack from one end, and while it is being popped off, another string can be pushed from the other end. In this manner, a stack large enough to hold one maximum-length string can be used to process two strings concurrently, provided it maintains two stack heads and has the capability of performing simultaneous pushes and pops.

Unfortunately, this does not completely solve the problem. For instance, if a very long string is pushed onto the stack, then as it is being reversed a single-byte string is pushed onto the other end, the decompressor must still wait until the first string is completely popped before continuing. However, the idea of using two stacks can be logically extended to an entire set of stacks. If there are enough of them available, the decompressor would never need to wait to push a string to be reversed.

Since the output strings to be reversed have been limited in length, the number of stacks required can be easily determined. The most demanding case the decompressor could produce would be a string of maximum length, followed by a series of single-byte strings. Since the first string has been limited to 128 bytes in length, by the time the 128th single-byte string has been pushed, the first string will have been completely popped, and that stack will be available for use. Any other set of string lengths will require fewer stacks before the first one is emptied, so the maximum number required is 129 (or one more than the maximum string length).

Actually implementing 129 stacks would require 16,512 bytes of memory, since each stack must be capable of storing a maximum-length string. Most of this memory would be unused at any given time, so a more efficient means of storing the strings is desirable. Rather than viewing the reversal buffer as a stack or set of stacks, it can be represented as a circular queue, or ring buffer. Instead of keeping a pointer to the top of each stack, a pair of pointers can be kept to locate the start and end of each string in the queue (the *tail* and *head* pointers, respectively).

When the first byte of a string to be reversed is written to the string reversal module, the next available pair of pointers will be set to the next available byte in the ring buffer, and the byte will be written to that location. Then, as subsequent bytes of the string are received, the *head* pointer will be incremented, and the byte will be stored in that location. When the end of the string is received, a new pair of pointers can be assigned to the next byte in the buffer, and the current pair will locate the start and end of the string. To reverse the string, the byte addressed by the *head* pointer will be written to the output buffer and the *head* pointer decremented until the byte addressed by the *tail* pointer has been written. At this point, the entire string has been reversed, and the pair of pointers is available for use again.

The size of the ring buffer required to implement this scheme is substantially less than the number of bytes required to implement the multiple-stack technique. The most demanding condition produced by the decompressor is the generation of a maximum-length string followed by a series of strings of any length. The ring buffer must contain 256 bytes to accommodate two maximum-length strings. The first string will be written into the first 128 bytes of the buffer, and as the second string is being written, the first will be reversed. By the time the second string has been fully written into the second 128 bytes of the buffer, the first will have been entirely reversed, and the first 128 bytes will be available for use again. Note that this is true for a series of shorter strings as well. By the time 128 bytes have been written, the space used by the initial string is available again, so 256 bytes should be sufficient for all cases.

The issue remaining is the management of the *head* and *tail* pointers. These can also be stored as pairs in another circular queue, the *string* queue. A pair of pointers is maintained for the string currently being written into the ring buffer (the *insertion* pointers), and another pair for the string currently being reversed (the *removal* pointers). When the end of a string is written, the *insertion* pair is added to the head of the *string* queue and they are reset to point to the next available space in the ring buffer. When the reversal of a string is completed, the next pair is removed from the tail of the *string* queue and placed into the *removal* pair. If a maximum-length string is generated by the decompressor, followed by a series of single-byte strings, 128 pairs of pointers would be added to the *string* queue before the first string is reversed. This is the maximum number of entries required; any other combination of strings following the maximum-length one will produce fewer additions to the *string* queue before the first string is reversed and its pointers are available for use.

Since the ring buffer is 256 bytes long, each pointer to it must be eight bits long. The total amount of memory required to implement this string reversal mechanism is 512 bytes (256 for the ring buffer, and another 256 bytes for the 128 pairs of one-byte pointers in the *string* queue). Two pairs of eight-bit registers are also required for the *insertion* and *removal* pointers, and a pair of seven-bit registers is required for the *head* and *tail* pointers for the *string* queue. Note that, since the length of each queue is a power of two, implementing ring buffers is simply a matter of ignoring the carry generated from the most significant bit when a pointer is incremented. When a pointer reaches the end of the buffer, it will automatically wrap around to the beginning the next time it is incremented. A block diagram is shown below.

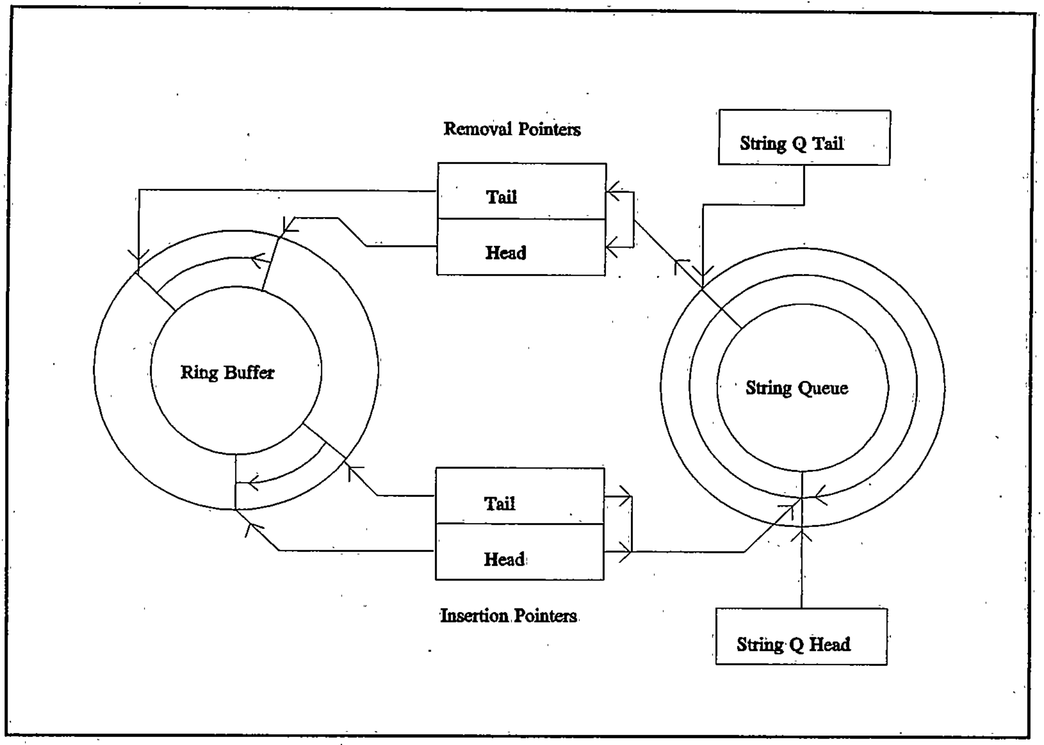

Figure 9 - Block Diagram of String Reversal Mechanism

Pseudo-code for the string reversal algorithm using the data structures described above is shown in Figure 10. **InsertHead** and **InsertTail** are the pointers to the string currently being written into the ring buffer, **RingBuff**, and **RemoveHead** and **RemoveTail** are the pointers to the string currently being reversed. **RemoveUsed** is a flag to indicate whether there is a string being reversed, and **StringQHead** and **StringQTail** are the head and tail pointers for the string queue, which is divided into **String-Head** and **StringTail**. Note that if a head and tail pointer are equal, the string or queue is empty, and that the head pointer always points to the next position available to insert an element.