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

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
Date

Chairperson, Graduate Committee

Approved for the Major Department

Date

8-8-91
Head, Major Department

Approved for the College of Graduate Studies

August 9, 1991
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his/her absence, by the Dean of Libraries when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

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

TABLE OF CONTENTS--Continued

TABLE OF CONTENTS--Continued

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES--Continued

LIST OF FIGURES--Continued

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".

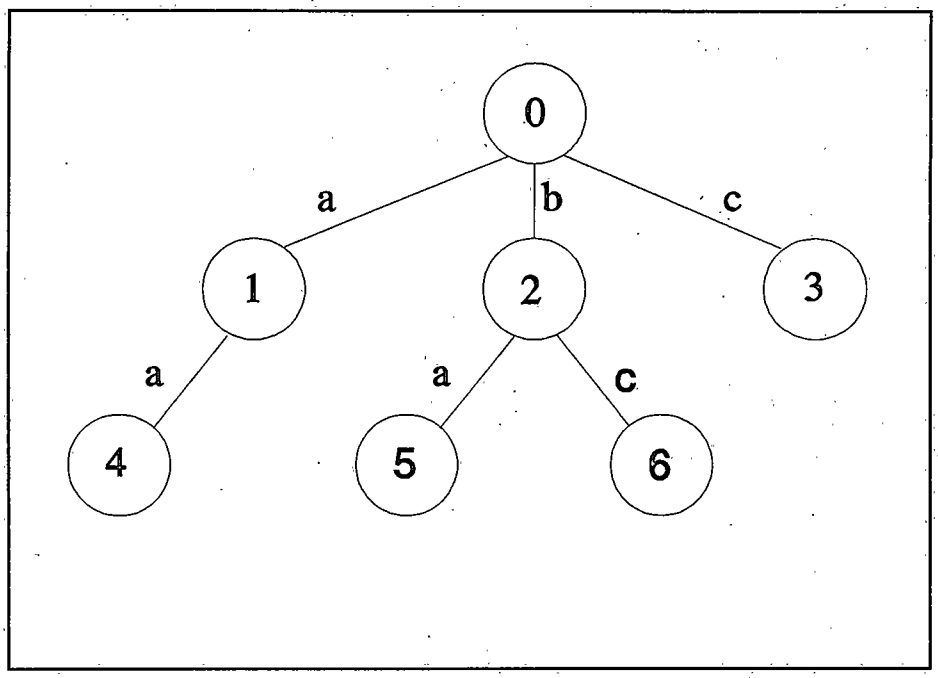


Figure 1 - Example of a Dictionary Trie

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.

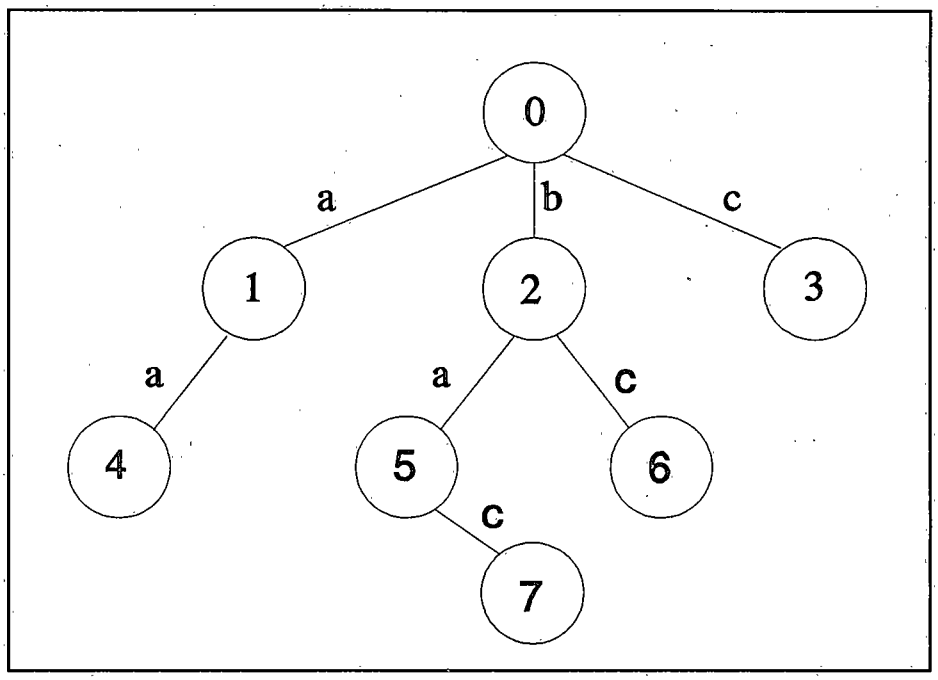


Figure 2 - Modified Dictionary Trie

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 Entries		4	5	6		7		8			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.

Compressor String Table Search

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.

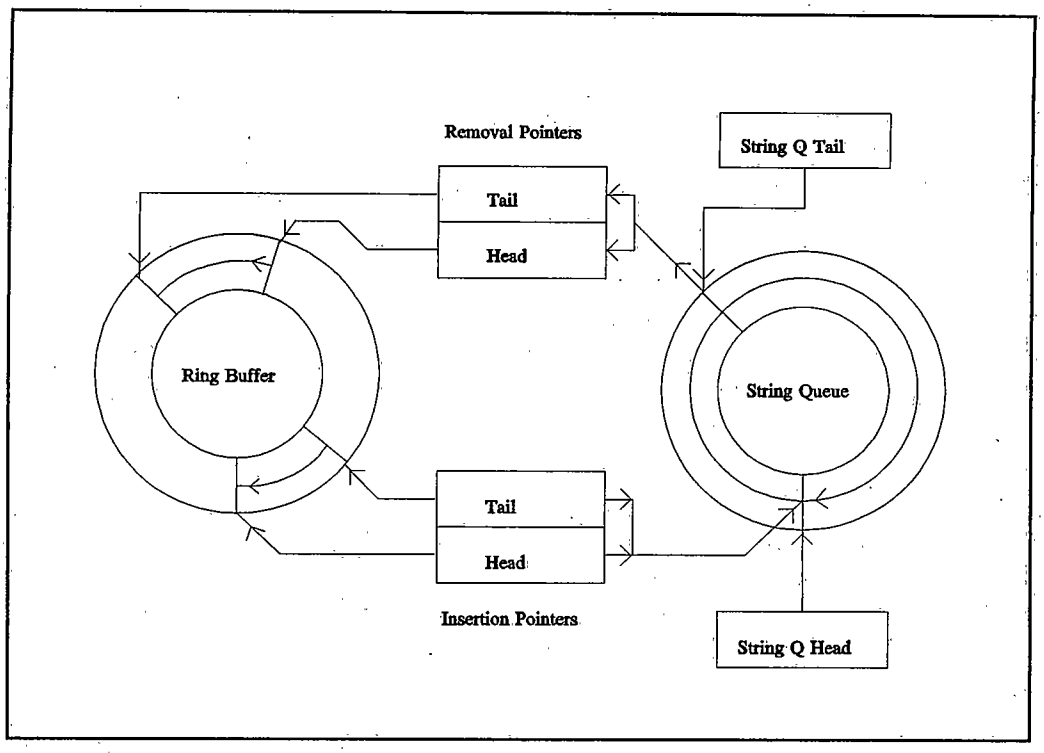


Figure 9 - Block Diagram of String Reversal Mechanism

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 **StringHead** 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.

```
If (Reset)
    StringQHead, StringQTail  ←  0
    InsertHead,  InsertTail   ←  0
    RemoveUsed  ←  FALSE
Else loop
    If input byte ready
          RingBuff [InsertHead]  ←  input byte
          Increment InsertHead, wrapping from 0xff to 0
          If byte is end of string
                StringHead [StringQHead]  ←  InsertHead
                StringTail [StringQHead]  ←  InsertTail
                Increment StringQHead, wrapping from 0x7f to 0
                InsertTail  ←  InsertHead

    If (RemoveUsed)
        Decrement RemoveHead, wrapping from 0 to 0xff
          Output RingBuff [RemoveHead]
          If (RemoveHead == RemoveTail)
                If (StringQHead == StringQTail)
                    RemoveUsed  ←  FALSE
                Else
                    RemoveHead  ←  StringHead [StringQTail]
                    RemoveTail  ←  StringTail [StringQTail]
                    Increment StringQTail, wrapping from 0x7f to 0
                    RemoveUsed  ←  TRUE
    Else if (StringQHead ≠ StringQTail)
          RemoveHead  ←  StringHead [StringQTail]
          RemoveTail  ←  StringTail [StringQTail]
          Increment StringQTail, wrapping from 0x7f to 0
          RemoveUsed  ←  TRUE
End loop
```

Figure 10 - String Reversal Algorithm

Note that the input and output sections of the algorithm are meant to be executed in the same system clock cycle. A typical software implementation would not benefit from this scheme, because the read and write operations could not be executed in parallel. However, this is relatively easily accomplished in hardware. This is the feature that provides the performance increase for the decompressor; as it is writing out strings to be reversed, previously written strings are concurrently being reversed and written to the output buffer.

Physical Implementation Considerations

The string reversal module interface will consist of the following signals (note that the module is still referred to as a stack, for compatibility with original algorithm):

Inputs - 8-bit **StackData** bus, **Reset**, **WriteStack**, **EndOfString**, and **OutByteFIFOFull** control signals

Outputs - 8-bit **OutputByte** bus, **WriteByte** and **StackFull** control signals

Reset is the global reset and initialization signal. **StackData**, **WriteStack**, and **EndOfString** are the inputs from the decompressor. When **WriteStack** is asserted, the byte present on the **StackData** bus will be latched as the next input. If **EndOfString** is also asserted, the byte will be marked as the last in the current string and processed accordingly. **OutputByte** and **WriteByte** are the interface to the output buffer. When a byte is to be output, it will be placed on the **OutputByte** bus and the **WriteByte** signal will be asserted. It is expected that on the next clock edge, the output buffer will latch the value.

Two additional control signals, **StackFull** and **OutByteFIFOFull**, which were not previously mentioned have been added. The algorithm described above assumes that it will always be possible to write a byte to the output buffer. The actual implementation must provide flow control between the output buffer and string reversal module. When **OutByteFIFO-Full** is asserted, bytes cannot be written to the **OutputByte** bus. However, it will still be possible for bytes to be received from the decompressor. This will make it possible for either the ring buffer or the string queue to fill up. Additional logic must be added to detect this condition and assert **StackFull**, until the output buffer can accept bytes again and there is room in the ring buffer and string queue for new entries. It is expected that the assertion of this signal will halt the decompressor output, and any attempts to write additional bytes while **StackFull** is asserted will be ignored. This additional logic is detailed in Chapter

10, in which a standard-cell implementation of the string reversal module controller is described.

The control logic design has been completed assuming that the string queue will be large enough to hold all 128 pairs of pointers required to allow the decompressor to run without interruption. However, if this is not feasible due to area limitations, the size of the queue could be reduced somewhat without dramatically reducing its benefits to decompressor throughput. The controller will already be able to handle string queue overflows and generate a halt signal, so no additional logic will be necessary. Software simulations of the decompression algorithm on various compressed files could be run to determine how small the queue could be made before the decompressor is forced to pause too frequently.

The details of the actual implementation of the memory used for the ring buffer and string queue are not fully decided yet. It is assumed that it will be static RAM, but if area is severely constrained, it may be necessary to use dynamic RAM. Another unresolved issue is the means by which the memories can be written and read in the same clock cycle. Since the memory will be a custom module, one possibility is the use of dual-ported RAM. This RAM would have separate read and write data and address busses and be capable of physically reading and writing different memory locations concurrently. Another alternative would be to gate the read and write signals using opposite phases of the system clock, and to use the clock to select which data and address busses to connect to the RAM. That is, the write operation would be performed in the first half of the clock cycle, and the read operation would be performed in the second half (so that the output data from the RAM could be latched by the control logic).

There are several difficult timing issues involved with using the clock in this manner, so this option is less attractive than the dual-ported RAM. Also, the RAM would have to have half the access time of the dual-port memory, since it would be required to do two operations in a single clock cycle. However, dual-ported RAM is typically larger than normal static RAM, so the second alternative may be necessary. A solution to the timing problems introduced by using both phases of the clock would be to use a double frequency clock and perform reads and writes on alternate clock cycles.

Regardless of which method is chosen for the RAM implementation, the control logic will treat it as dual-ported RAM, with separate read and write address and data busses. Read and write operations may both be requested concurrently during the same clock cycle, and the controller will make no assumptions about the portion of the clock cycle during which they will take place. It will only assume that the data requested from a read ill still be on the read data bus at the end of the clock cycle to be latched on the next clock edge.

CHAPTER 5

INPUT AND OUTPUT BUFFERING

As mentioned previously, I/O buffers will be provided between the compressor and decompressor and the IC's external interface. These buffers are intended to smooth the flow of data into and out of the IC and to allow for an external interface which is not compatible with that expected by the compressor and decompressor; for example, the buffers might provide direct memory access (DMA) data transfers to and from the external busses. However, the external IC interface has not yet been defined, so the design of the I/O buffers must be deferred.

It is expected that these buffers will be FIFOs similar to the circular queues used in the string reversal mechanism, utilizing a similar RAM access scheme to allow read and write operations during the same system clock cycle. They will be divided into eight-bit character buffers, for the compressor input and decompressor/string reversal output, and twelve-bit code buffers, for the decompressor input and compressor output. The internal interface for each buffer is specified below.

Character Buffers

The character buffers should be very straightforward, assuming the external interface is not excessively complex. The simplest scheme would be similar to that used for the string queue in the string reversal mechanism; the external bus would assert a write signal and put characters into

a ring buffer, while the compressor would assert a read signal and remove characters from the buffer, or vice versa for the decompressor's buffer.

The compressor's input character buffer, **InputCharFIFO**, will have the following internal interface:

Inputs - **ReadCharFIFO** and **Reset** control signals

Outputs - eight-bit **InCharFIFO** data bus, **InCharFIFOEmpty** and **InCharDataEnd** control signals

The system **Reset** signal will reset the FIFO status to empty and turn off the **InCharFIFOEmpty** and **InCharDataEnd** signals. Whenever there is data in the ring buffer, the next available character will be placed on the **InCharFIFO** bus. In the clock cycle when the compressor latches the value, it should assert **ReadCharFIFO**; on the next clock cycle, the next available character will be placed on the bus. The FIFO will assert **InCharFIFOEmpty** (after data has been initially written into it) whenever its ring buffer is empty. It is assumed that, in addition to a **WriteCharFIFO** signal from the external interface, there will be a control signal that will be asserted when the last byte of input is written into the FIFO. After this signal has been asserted, when the ring buffer is emptied, the **InCharDataEnd** signal will be asserted to signify that the compressor has processed all input.

The decompressor's output buffer, **OutputCharFIFO**, will have the following internal interface:

Inputs - eight-bit **OutCharFIFO** data bus, **WriteCharFIFO** and **Reset** control signals

Outputs - **OutCharFIFOFull** control signal

The system **Reset** signal will empty the FIFO and reset all of its status flags. When the string reversal mechanism needs to output a byte,

it should place that byte on the **OutCharFIFO** bus and assert **WriteCharFIFO**. Both signals should be maintained until the next negative clock edge, when the FIFO will latch the byte and place it into the ring buffer. **OutCharFIFOFull** will be asserted when there is no more room in the ring buffer; additional write attempts while it is asserted will be ignored.

If the external interface is symmetric (character input and output use the same mechanism), it should be possible to combine both FIFOs into one. The logic for managing the ring buffer should be the same in both cases, with the only difference being that bytes are coming from the external bus in the compression case and the internal bus in the decompression case. A **Compress** control input could be used to select the direction of data flow through the FIFO.

Code Buffers

The code buffers will be very similar to the character buffers, and should have an identical interface.

The decompressor's input code buffer, **InputCodeFIFO**, will have the following internal interface:

Inputs - **ReadCodeFIFO** and **Reset** control signals

Outputs - eight-bit **InCodeFIFO** data bus, **InCodeFIFOEmpty** and **InCodeDataEnd** control signals

The compressor's output code buffer, **OutputCodeFIFO**, will have the following internal interface:

Inputs - eight-bit **OutCodeFIFO** data bus, **WriteCodeFIFO**, **FlushCode**, and **Reset** control signals

Outputs - **OutCodeFIFOFull** control signal

The only new signal is **FlushCode**, which should be asserted after the compressor has output the last code for the data stream. This will force the output FIFO to dump its last byte, even if it was only half-filled by the last code.

The code buffers will be complicated somewhat by the requirement that they convert the twelve-bit internal data stream into an eight-bit external data stream. This conversion could take place either before or after the data is written to the ring buffer. Again, if the external read and write operations are symmetric, the two FIFOs could be combined into one, sharing the RAM and much of the control logic and providing a substantial area savings. Also, if future enhancements require that variable-length output codes are handled, that functionality could be integrated directly into the code buffer(s), and the compressor and decompressor could continue to assume a simple twelve-bit code stream.

The determination of the size of the ring buffers used in the character and code buffers will be deferred until the external interface has been finalized. The information presented here is sufficient to proceed with the design of the controller logic.

CHAPTER 6

HARDWARE DESIGN PROCEDURE

The preceding chapters have specified the requirements of this hardware implementation and enumerated several constraints placed on its design. The procedure which will be followed to complete the design and realize the circuit must now be detailed.

The design to be completed will involve only the control logic for the LZW IC. The custom DCAM and RAM modules described in Chapters 3 and 4 will not be included in this process. The DCAM interface was defined in Chapter 3, and the RAM interface to the string reversal module was described in Chapter 4 and will be defined in detail in Chapter 10. In addition, the input and output buffer modules will not be included in the design. Their requirements and interface were discussed in Chapter 5.

The remainder of this document describes the design of the controller, including its subdivision into modules, the interface requirements and functional specification of each module, the high-level description of the logic required to implement that functionality, the process for synthesizing that logic, and the procedure used to verify that the logic thus generated implements the specified functionality. This chapter describes the framework in which the design will be carried out, including the methodology used to define the modules and the CAD tools used to synthesize and verify them, and presents the decomposition of the controller into modules.

Target Technology

All control logic will be implemented using standard cell design techniques. The logic for each module will be described using a Hardware Description Language (HDL), converted to logic equations and optimized, then mapped into the Mississippi State University standard cell library (revision 2.2). The cells in this library are drawn for the MOSIS scalable CMOS N-well (SCN) process, with two metal layers and a two micron minimum feature size. The resulting mapping will be placed using automated place-and-route tools to form the completed logic modules.

CAD Tools

The CAD tool set to be used for this implementation is the **OCT** tool suite (Version 4.0) distributed by the University of California at Berkeley. **OCT** runs in the UNIX environment and is distributed in source code form, so it can be ported to a variety of host hardware platforms. It provides a fairly complete set of tools for doing custom and semi-custom circuit designs, including a logic synthesis package (with standard cell, PLA, and gate matrix generation capabilities), physical and symbolic layout editors, tools for creating and managing circuit hierarchies, a logic-level circuit simulator, and automated place-and-route tools. The individual **OCT** tools that have been used for this implementation are **bdsyn, misII, bdnet, wolfe,** and **musa.**

Bdsyn

Bdsyn is a hardware description translator. Its input is a "program" written in **BDS** which describes the functionality of a block of

combinational logic, and its output is a multi-level logic representation of the described function, in the Berkeley Logic Interface Format (**BLIF**). The **BLIF** representation is then used as input to optimization and technology mapping tools.

BDS is a behavioral description language first introduced as part of Digital Equipment Corporation's hardware simulation system **DECSIM**. A subset of **BDS** was adopted as the HDL for **OCT**. It provides a means for quickly describing and implementing relatively complex logic functions. Its syntax is derived from the Pascal programming language, and it supports a number of high-level language constructs, including statement blocks, subroutines, if-then-else statements, case (select) statements, and for loops (with constant index limits). It also recognizes complex operators such as integer addition, subtraction, multiplication, and division, bit shifts, and logical comparisons, and generates the logic necessary to implement these constructs and operations. Note that only combinational logic can described by **BDS**; that is, the logic generated uses no clocking or signal latching of any kind. The generation of sequential systems such as state machines requires the addition of latches using other tools.

The combinational logic for each of the LZW modules is described in **BDS**, and **bdsyn** is used to convert this high-level description into the corresponding **BLIF** logic equations for the module, which are used in further stages of the implementation. Detailed information on the **BDS** language and **bdsyn**'s operation is provided as part of the online documentation distributed with **OCT** (the file *bdsyn.doc* in the *~octtools/doc* directory). A summary of **bdsyn**'s capabilities and options is also

available in the online UNIX **man** pages (the UNIX help facility), provided the help files that are distributed with **OCT** have been installed.

MisII

MisII is a multi-level logic synthesis and minimization program. It accepts a description of a combinational logic function in a variety of formats, including **BLIF**, and produces an optimized set of logic equations which will implement the function. In addition to synthesizing a network realizing those equations, it provides a technology mapping step which will map them into a user-specified cell library (in this case, the Mississippi State University library). Following this mapping step, **misII** can perform timing analysis on the resulting network, given the timing information for each cell in the library. Information obtainable includes the arrival time, required time, and slack for each node in the network. This information can also be sorted in order of critical values, so the critical timing paths in the network can be identified.

This is the tool that actually synthesizes and minimizes the logic described in the **BLIF** files produced by **bdsyn** and generates a collection of standard cells to implement that logic. A fairly comprehensive overview of **misII**'s capabilities is presented in the online UNIX **man** pages.

Bdnet

Bdnet is a net-list translator which allows the manipulation of networks of devices stored in **OCT**'s internal format and provides the capability of forming network hierarchies. A script written in a special language can specify the creation of a new **OCT** cell, define the input and output terminals of the cell, allow the instancing (logical placement) of

existing **OCT** cells within the new cell, and describe the logical interconnection of these instances. For example, once all the LZW modules have been created as **OCT** cells, **bdnet** would be used to create a new macro-cell containing all the modules and to specify how the input and output terminals on the various modules interconnect and how they connect to the macro-cell's inputs and outputs. Note that these would only be logical netlist interconnections; actual physical interconnections would have to be created using either manual or automatic placement tools.

In addition to assembling modules into a hierarchy, **bdnet** is used to add memory elements (such as flip-flops) to the combinational logic generated by **misII** to form synchronous circuits. Note that each time **bdnet** is used to connect cells, another level of hierarchy is created. The **octflatten** tool is provided to reduce this hierarchy. For instance, if a controller module is generated using **misII** and flip-flops are added using **bdnet**, **octflatten** can be used to compress the resulting cell into a single hierarchical level. Information on both **bdnet** and **octflatten**, including the syntax of **bdnet**'s input language, is available in the online UNIX **man** pages.

Wolfe

Wolfe performs standard cell placement and routing of **OCT** cells. It uses the **TimberWolfSC** utility to perform cell placement and global routing, and the **YACR** (**Y**et **A**nother **C**hannel **R**outer) utility to perform detailed routing. In addition, it routes power and ground busses for the cell. Its placement criterion is the minimization of overall wire length.

This is the tool that creates the actual physical circuits for each module. Information on its operation and options is available in the

online UNIX **man** pages; more detailed information on **TimberWolfSC** and **YACR** can be found in the files *TimberWolfSC4-0.doc* and *yacr.doc* included in the **OCT** distribution (in the *~octtools/doc* directory).

Musa

Musa is a multi-level logic simulator. It is primarily used to perform switch-level simulation of MOS transistors, but it has been extended to allow simulation of higher-level **OCT** constructs, including latches and memory cells. It is an interactive program, but will also accept scripts specifying a sequence of commands to control simulations. It allows the user to set inputs and propagate the changes through the network, then examine the logic level of nodes throughout the network and verify the state of outputs. It allows several logic levels, including driven high, low, or undetermined by the circuit or set high, low, or undetermined by the user.

This tool provides the primary means of verifying the module designs. Information on its operation and options, including the syntax of its input language, is available in the online UNIX **man** pages.

Design Methodology

The design of each module proceeds through a sequence of steps very similar to those typically encountered in software design. The software design cycle is often divided into the following stages: module requirements and functional specifications, high-level (algorithmic) design, coding, compilation and linking, and module testing. Each of these stages has an equivalent phase in the standard cell design methodology used for this implementation.

External Reference Specifications

The module requirements and functional specifications are represented by the **External Reference Specification** (**ERS**) form. This document identifies (names) the module, details its interface, identifying each input and output, and includes a functional description. This description is not presented at a detailed level (such as identifying how many registers will be required or specifying what type of adder circuits should be used), but instead defines the *black-box* functionality. That is, given a set of inputs, it identifies what the outputs should be. This can be presented in the form of a truth table or a more complex representation, such as an algorithmic description. The **ERS** should also include design constraints, such as maximum area requirements and critical timing specifications for outputs, if they are known.

An example of the standard **ERS** form used at Montana State University is included as Figure 27 of Appendix A. Note that it also includes information about the module that will be determined at future points in the design cycle, including performance data, power requirements, input and output terminal parameters, and a verification checklist. It has been designed to not only present the module requirements, but to track the design process.

RTL Descriptions

The high-level module design is documented using a **Register Transfer Level** (**RTL**) description. **RTL** is analogous to pseudo-code in software designs, and is a commonly used method of describing synchronous circuits such as microprocessor controller modules. Much like pseudo-code, there

is no accepted definition of **RTL** syntax; any conventions which clearly denote the intended operations are acceptable.

An example of a complete **RTL** module description is shown in Figure 28 of Appendix A. Much of the syntax is borrowed from the C programming language, including the equality and inequality comparison operators (== and !=), the logical **and** and **or** operators (&& and ||), the addition and multiplication operators, and the notation for hexadecimal numbers (the **0x** suffix). Some additional symbology used includes:

<- denotes a synchronous register assignment; that is, the D inputs to the flip-flops are set to the indicated value, and on the next negative clock edge, it will be latched into the register

<= denotes an asynchronous assignment; that is, a temporary variable or output is immediately assigned the value indicated

. indicates concatenation; for example, 0x00 . 0x10 is 0x0010

! marks the beginning of a comment, which extends to the end of the line (unless it is part of the inequality operator !=)

<a:b> denotes a range of bits in a register or signal bus; for example, **x<5:3>** would select bits 3 through 5 (inclusive) of register **x** (note that the least significant bit is always numbered 0)

Each **RTL** description includes a preliminary section describing the inputs and outputs of the module, the registers used and their lengths, and the variables used. These variables are basically intermediate values used in combinational logic operations, and can only be assigned values using the asynchronous assignment operator, **<=**. Following this declaration section is the logic description, in a block-structured format. Indentation is used to indicate statement blocking; there are no explicit block start and end markers. Other notation should be self-explanatory.

The level of detail in the **RTL** description is fairly arbitrary. Operations should not be detailed down to the bit level; for instance, if the contents of two registers are to be added and placed in a third register, it is not necessary to detail the combinational logic required to perform the addition; this is a detail that should be deferred until the HDL is written. However, enough detail must be present in the **RTL** description to provide an unambiguous description of all operations that must be performed internally by the module in order to satisfy the requirements specified in the **ERS**.

BDS and Bdnet Input Files

The **BDS** and **bdnet** input files are the equivalent of source code in a software implementation. The **BDS** file must describe the design given in the **RTL** in sufficient detail that **bdsyn** and **misII** can generate multi-level logic equations to implement the functionality. The **bdnet** file is used to connect flip-flops for registers, and to assemble sub-modules if necessary.

A convention has been adopted to describe registers in **BDS**, since **bdsyn** does not directly support them. The output declarations of the **BDS** file will include an output for each register specified in the **RTL**, with the name *register*_**D**, and the input declarations will include a corresponding input with the name *register*_**Q**. Thus values to be stored in a register will be assigned to *register*_**D**, and the value stored in a register is accessed by referencing *register*_**Q**. For example, the **RTL** shown in Figure 11 would have the **BDS** implementation also shown in that figure. Once the **BDS** file has been successfully processed by **bdsyn**, the flip-flops required to create the registers are added using **bdnet**, as shown in Figure 11.

Sample **RTL** description

```
INPUT     A <3:0>, B <3:0>, C <3:0>
OUTPUT    D <4:0>
REGISTER  Sum <5:0>

Sum  <-  A + B
D    <=  Sum + C
```

BDS equivalent, which will create test:logic

```
MODEL Test
D     <5:0>,                  !   Module output
Sum_D <4:0>,                  !   Output to register
=
A <3:0>, B <3:0>, C <3:0>,    !   Module inputs
Sum_Q <4:0>;                  !   Input from register

ROUTINE TestRoutine;
        Sum_D  =  A + B;      !   Sum  <-  A + B
        D      =  Sum_Q + C;  !   D    <=  Sum + C
ENDROUTINE TestRoutine;
ENDMODEL Test;
```

Corresponding **bdnet** input, which will create test:unplaced

```
MODEL      test:unplaced;
TECHNOLOGY scmos;

OUTPUT     D <5:0>;
INPUT      A <3:0>, B <3:0>, C <3:0>;
CLOCK      Clk;
SUPPLY     Vdd;
GROUND     GND;

INSTANCE test:logic PROMOTE;

ARRAY I% FROM 0 TO 4 OF
        INSTANCE "d_flip_flop":physical
                D : Sum_D <I%>;
                Q : Sum_Q <I%>;
                Q_BAR : UNCONNECTED;
                CLK : Clk;
                Vdd : Vdd;
                GND : GND;
```

Figure 11 - Implementing Registers with BDS and Bdnet

The Logic Generation Process

The use of the CAD tools to process the **BDS** and **bdnet** input files is analogous to the compilation/assembly/linkage process in a software environment. The translation of a **BDS** file to **BLIF** is the equivalent of compiling a program written in a high level language into assembly code, while the mapping into logic performed by **misII** is much like assembling

that code into machine language. **bdnet** performs a function similar to that of the linker, joining modules together and linking in system libraries (in this case, flip-flops and other library cells).

The execution of **bdsyn**, **misII**, and **bdnet** is typically performed iteratively, initially to remove syntax errors from the input files, and subsequently to remove logic errors and add new functionality. A method for automatically controlling this process is described in Chapter 12. Once these steps have been completed successfully, **wolfe** is run to generate the placed and routed module. **Chipstats**, an **OCT** utility, can be run on the **OCT** cell created by **wolfe** to determine the number of standard cells used in the implementation and the dimensions of the resulting layout.

Logic Simulation

After the module has been created, it must be tested to verify that it does implement the functionality specified in the **ERS**. When the **BDS** description is created, an accompanying set of **musa** scripts should also be written to test it. Much like a set of test inputs for a program module, the scripts should be devised to verify normal modes of operation and to check behavior for input boundary conditions and unusual cases. A full set of *black-box* tests should be performed on the module before it is ready for integration with other modules. A *black-box* test is one in which only the inputs and outputs are available to the tester, so only the overall functionality of the module can be tested (as opposed to assuring that all possible paths of execution through the module have been checked for correctness).

If **octflatten** is executed on the module before the simulations are run, all of its internal nodes are accessible to **musa**. This allows **musa**

to be used to debug the module and perform *white-box* testing as well. A *white-box* test is one in which the tester has access to the internals of the module, and can verify that the test cases do indeed cover all paths through the logic.

Naming Conventions

All design documentation associated with each module will be kept as simple ASCII text files. The following naming conventions will be used for files created during the design of a given *module*:

module.ers - the External Reference Specification (ERS) form
module.rtl - the Register Transfer Level (RTL) description
module.bds - the **bdsyn** input file
module.blif - the **BLIF** output of **bdsyn**
module.bdnet - the **bdnet** source file
module.musa - the initial **musa** source script, in which all macros and vectors are created for testing the module. Additional scripts will be named *module*_1.musa, *module*_2.musa, etc.

An example of each of these files is shown in Appendix A, Figures 27 through 31 (except the **BLIF** file). The following naming conventions will be used for **OCT** views created during the creation of the placed and routed module:

module:logic - output from **misII**; logic without flip-flops
module:unplaced- output from **bdnet**; *logic* with flip-flops added
module:flat - output from **octflatten**; *unplaced* without hierarchy
module:placed - output from **wolfe**; *flat* placed and routed

Finite Field Sequencers

Several functions of the controller require the generation of fixed-length sequences, such as the refresh address generator and string length counter. These functions can easily be implemented using binary counters; however, there is an alternate method that requires far fewer

logic gates to implement. Since it is not important that each value in the sequence be one more than the previous one, Galois field (or finite field) based linear feedback shift registers can be used to generate these sequences. These finite field sequence generators have the property that the sequence repeats in a regular fashion after a number of cycles equal to $2^n - 1$, where *n* is the number of bits in the field (the value 0 is never generated in the sequence). This is ideal for the generation of refresh addresses, and works well for string length counting, since intermediate values are not important and only the value corresponding to the maximum string length needs to be detected.

These sequence generators are based on a finite field generating polynomial. For example, the polynomial $P(x) = x^4 + x + 1$ represents the linear feedback shift register shown in Figure 12, where the + operator is a finite field adder. Finite field addition is similar to normal binary addition, except that no carries are generated, so this operation is equivalent to a simple exclusive-or. Given a non-zero initial value, the register will generate each value from 1 to 15 once and only once before repeating the initial value.

Given the number of bits *n* that are required for the values in a sequence, an irreducible primitive polynomial of order *n* must be chosen for the generator. Tables of minimum weight irreducible polynomials have been published; these polynomials have the minimum number of non-zero coefficients possible. This implies that the shift registers implementing these polynomials will require the minimum number of exclusive-or gates. For a table of minimum weight irreducible polynomials of order three to 34, and a more in-depth discussion of finite-field sequencers and their

applications, see [WINTERS]. The savings realized by using finite field sequencers instead of binary counters are substantial; for example, a 10-bit incrementer generated using **bdsyn** and **misII** requires 26 standard cells. The minimum weight polynomial of order 10 is $x^{10} + x^3 + 1$, which requires only a single two input exclusive-or gate, which is available in the Mississippi State cell library. In fact, examination of the minimum weight polynomial table shows that for orders up to 34, no more than three exclusive-or gates are ever required.

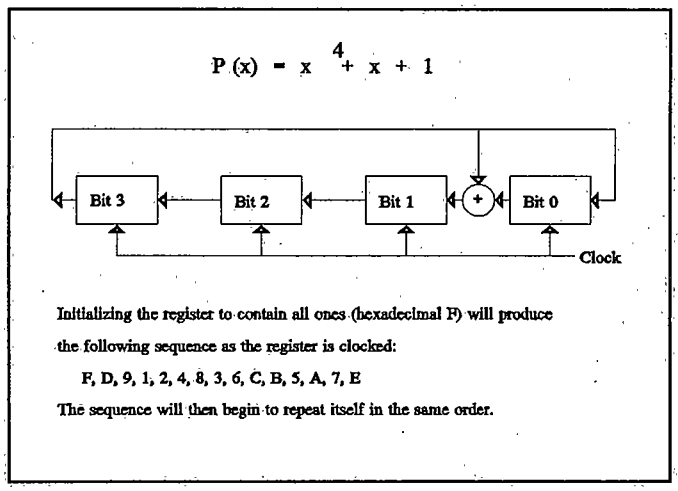


Figure 12 - Linear Feedback Shift Register

Problem Decomposition

The controller has been decomposed into the following modules: the compressor, the decompressor, the DCAM, the string reversal mechanism, the master controller, and the I/O FIFOs. The DCAM and the I/O FIFOs have been described in Chapters 3 and 5, respectively, and will not be developed further. Each of the other modules is described in a subsequent chapter. The compressor and decompressor are further subdivided into the

original modules and the modules updated to function with the DCAM that performs an automatic write during each search operation. After the generation and verification of the compressor and decompressor modules, their functionality is merged into a single module (in order to share registers and logic). A block diagram of the system is shown in Figure 13. The signals will be explained in the following chapters.

For each of the modules, the **ERS** and **RTL** are presented, followed by a summary of the **BDS** and **bdnet** files, a review of the **musa** simulations performed, and a report on the size of the complete module.

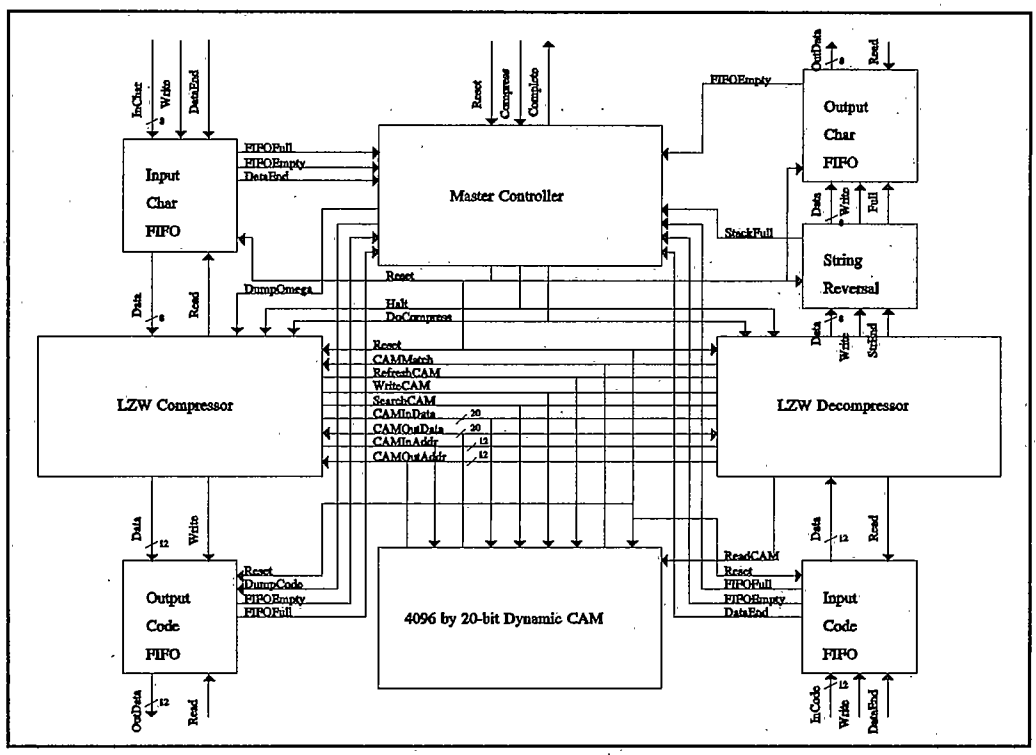


Figure 13 - System Block Diagram

CHAPTER 7

THE COMPRESSOR MODULE

The design of the compressor module is key to the success of the IC. This is where substantial performance improvements must be made over existing technology. Use of the DCAM should provide an advantage over the hashing methods typically used to search the string table, as long as the number of clock cycles required to perform each loop through the algorithm is minimized.

In Chapter 3, two different modes of operation of the custom DCAM are detailed. The first is a standard mode, in which the DCAM is searched for a string table entry; if the entry is not found, the compressor writes a new table entry. The second is a specialized mode, in which the DCAM automatically writes the table entry it is searching for into the next available location in the table. The compressor design has been split into two separate modules: **compress**, which is designed for the standard mode of operation, and **compress_2**, which is designed to take advantage of the DCAM modification. Each module is described separately.

Module Design for Normal DCAM Operation

The normal mode compressor is presented in this section. Note that the basic design does not include logic to allow the master controller to halt the compressor when an I/O wait occurs or to limit the length of accumulated strings. This functionality will be added in Chapter 9.

External Reference Specifications

The **compress** module will have the following interface:

Inputs - 8-bit **InCharFIFO** bus, 12-bit **CAMInAddr** bus, **Reset** and **CAMMatch** control signals

Outputs - 12-bit **OutCodeFIFO** bus, 20-bit **CAMOutData** bus, 12-bit **CAMOutAddr** bus, **ReadCharFIFO**, **WriteCodeFIFO**, **SearchCAM**, **WriteCAM**, and **DoRefresh** control signals

The signal names are fairly self-explanatory. The interface to the DCAM assumes that the master controller module will reset the DCAM at the same time the compressor is reset, and that it will provide the **Compress** signal if compression is being performed. The interface to the input and output FIFOs also assumes that they will be reset by the master controller. Note that the FIFO full and empty signals are not present in the compressor interface. It will be the responsibility of the master controller to handle all input and output buffering conditions and to halt the compressor as necessary.

The functionality of the compressor is straightforward: it must implement the LZW compression algorithm described in Chapter 2, and it must provide sufficient refresh cycles to maintain the DCAM's contents.

RTL Description

The RTL for the compressor is shown in Figure 14. It has been divided into three states, a reset and initialization state, and a two-state cycle that implements the basic compression loop. It is important to note that the outputs to the DCAM are latched; that is, the actual output signals are connected to the outputs of a set of registers. This pipelining operation may not be absolutely necessary, but is provided to ensure that the address, data, and control signals to the DCAM remain

```
INPUT     InCharFIFO  < 7:0>, CAMInAddr <11:0>, Reset, CAMMatch

OUTPUT    ReadCharFIFO, WriteCodeFIFO,
          !   Following outputs are latched.
          OutCodeFIFO <11:0>, CAMOutData <19:0>, CAMOutAddr <11:0>,
          SearchCAM, WriteCAM, DoRefresh

REGISTER  State, Omega <11:0>, K <7:0>, NextCode <11:0>, Refresh <11:0>

If Reset                                  !  Initialization stage

    SearchCAM, WriteCAM, DoRefresh  <-  0
    WriteCodeFIFO  <=  0
    ReadCharFIFO <=  1                    !   Read 1st byte to initialize Omega
    Omega          <-  0 . InCharFIFO
    NextCode       <-  0x100
    Refresh        <-  0x3ff
    State          <-  0

If (State == 0) and (!Reset)              !   First half of compression cycle

    ReadCharFIFO <=  1                    !   Read next byte into K
    K              <-  InCharFIFO

    SearchCAM      <-  1                  !   Enable a CAM search
    CAMOutData     <-  Omega . InCharFIFO !   Omega . K to search for

    Omega          <-  Omega              !   Retain register contents
    WriteCodeFIFO  <=  0

    If (DoRefresh)                        !   Doing a refresh - get next addr
        Refresh      <-  NextValue (Refresh)
    else
        Refresh      <-  Refresh          !   Retain last register value
    DoRefresh      <-  0                  !   Reset flag, regardless of value

    If (WriteCAM)                         !   Wrote a new entry - get new addr
        NextCode     <-  NextCode + 1
    Else
        NextCode     <-  NextCode         !   Retain last register value
    WriteCAM       <-  0                  !   Reset flag, regardless of value

    State          <-  1

If (State == 1) and (!Reset)              !   Second half of compression cycle

    ReadCharFIFO <=  0                    !   Done, turn off control bits
    SearchCAM      <-  0
    K              <-  K                  !   Retain last register values

    If (CAMMatch)                         !   Omega . K was in the table
        Omega          <-  CAMInAddr      !   Omega  <-  addr (Omega . K)
        WriteCodeFIFO <=  0               !   Don't generate output
        WriteCAM       <-  0              !     or a new table entry
        DoRefresh      <-  1              !   Refresh next row of CAM
        CAMOutAddr     <-  Refresh . 0    !   Pad Refresh with 0s to 12 bits
    Else
        Omega          <-  0 . K          !   Omega gets single-char string K
        DoRefresh      <-  0              !   Can't do that, going to write
        WriteCAM       <-  1              !   Enable both outputs
        WriteCodeFIFO <=  1
        CAMOutAddr     <-  NextCode       !   Put new Omega . K into table
        CAMOutData     <-  Omega . K
        OutCodeFIFO    <=  Omega          !   Output last Omega

    State          <-  0
```

Figure 14 - Normal Mode Compressor RTL

constant throughout the entire clock cycle. Because of this pipelining, the actual DCAM search is set up in state 0, but is performed in state 1. During state 1, the DCAM outputs are set up to either write the pattern from the failed search or to refresh the DCAM, depending on the state of the **CAMMatch** signal. During state 0, the requested write or refresh operation is actually performed. This demonstrates the reason for requiring that the DCAM refresh a row of words if a word in that row is written; it is possible for the controller to enter into a loop where it will add a new entry to the table on every other clock cycle (for instance, if the input consists of the sequence of bytes from 0 to 255) and be unable to perform a refresh. However, either a write or a refresh is guaranteed. Note that the write operations will be performed on sequential words in the DCAM, while refreshes are generated for rows using a finite field sequencer (the **NextValue** subroutine, whose implementation details have been left to the coding stage). However, the combination of the two refresh modes should guarantee that all rows of the DCAM are refreshed frequently.

This design will process input bytes at a constant rate of one per two clock cycles, provided that the input and output FIFOs can maintain this rate. The module assumes that the input FIFO will always have a byte ready, and that the output FIFO will always have room for another code. These assumptions can remain in place, provided the master controller handles the I/O buffer overflow and underrun conditions and generates a signal to halt the compressor. The addition of the halt capability is presented in Chapter 9.

One slight deficiency of the **RTL** is that the final accumulated code is never output. When the end of the data has been processed, the code remaining in the **Omega** register is the last code to output. This can be corrected by adding a signal to request writing the **Omega** register to the output FIFO during the halt state. This addition is included in the halt state detailed in Chapter 9.

Note that every register is assigned a value in each state, unless it is allowable for the register to lose its contents. If a new value is not being assigned, the old output value is fed back to the register inputs. This is to assure that the logic that selects the register inputs will have no uncovered input combinations, which could potentially lead to garbage data (such as all zeroes) being written into the register.

A timing diagram detailing the compressor signals is shown in Figure 15. The input is "abcab", and the output is 061 062 063, with the final code, 100, left in the *Omega* register.

Module Generation

The **BDS** and **bdnet** files for this module are a fairly straightforward translation of the **RTL**. The complete source files for the module are included as Figures 32 and 33 in Appendix B. The only unspecified detail is the construction of the finite field sequencer. The actual generation of the sequence depends on the number of bits required, which in turn depends on the number of words in each row of the DCAM. This will not be determined until the DCAM has been laid out. However, some initial assumptions can be made. For example, if it is assumed that there are four words in each row of the DCAM, ten bits are required to address each row. Consultation of the polynomial table given in [WINTERS] gives a

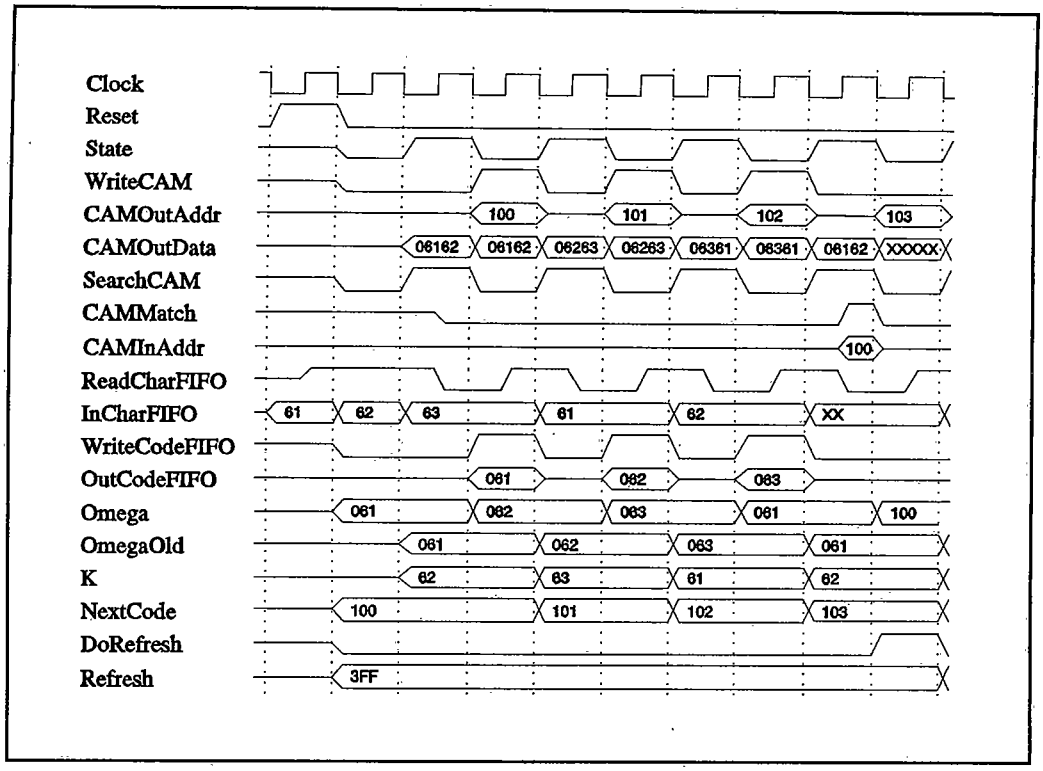


Figure 15 - Compress Module Timing Diagram

suitable generator polynomial $P(x) = x^{10} + x^3 + 1$. This can be implemented and used with the **BDS** subroutine and code fragment shown in Figure 16.

```
ROUTINE NextRefresh<9:0> (CurrentRefresh<9:0>)
    STATE  Temp<9:0>, I<>;

    Temp<0> = CurrentRefresh<9>;
    Temp<1> = CurrentRefresh<0>;
    Temp<2> = CurrentRefresh<1>;
    Temp<3> = CurrentRefresh<2> XOR CurrentRefresh<9>;

    For I from 4 to 9 do
        Temp<I> = CurrentRefresh<I - 1>;

    return Temp;

ENDROUTINE NextRefresh;

...
Refresh_D = NextRefresh (Refresh_Q);
```

Figure 16 - BDS Finite Field Sequencer

Any non-zero value can be chosen to initialize the **Refresh** register. Refresh addresses of different lengths can be easily changed by updating the length of the **Refresh** register and modifying the **NextRefresh** subroutine to use the appropriate generator polynomial.

The module has been successfully generated using the CAD tools. The **chipstats** analysis of the placed and routed module reports 302 standard cell instances, 327 nets, and a resulting area of 1432 by 2147 microns (3074504 square microns, or just slight over three square millimeters). Of this, 1.423 square millimeters are instance area (46.3 percent) and 1.651 square millimeters are routing and empty area (53.7 percent).

Module Testing

Musa test scripts have been generated to simulate the module for the input strings "aaaaaaaaaaaa" and "abaabacabcabcabaaa". Execution of the algorithm by hand determines that the code output sequence should be

061 100 101 102 100

(as twelve-bit hexadecimal values) for the first case and

061 062 061 100 061 063 100 105 062 107 102 061

for the second case. The test scripts verify that the module does indeed produce these output codes for the specified inputs.

Manual generation of the test scripts for various input cases is a tedious task. Only the two cases shown above were tested in this manner. In order to generate an appropriate set of test cases for other conditions (such as a case where the string table is filled up), a separate utility program has been written which executes the LZW algorithm on an input file and produces both the compressed output and a corresponding **musa** script simulating the module for that input. This program was used on a variety

of inputs, including the two previous examples, a file of 16 Kbytes of the character 'a', the source code file for the utility, and the executable file for the utility (the latter two of which did cause the string table to fill up). The module has verified correctly for all input cases tried.

Module Design for Enhanced DCAM Operation

The changes required to take advantage of the DCAM enhancement are fairly extensive, but they greatly simplify the resulting circuit.

External Reference Specifications

The **compress_2** module will have the following interface:

Inputs - 8-bit **InCharFIFO** bus, 12-bit **CAMInAddr** bus, **Reset** and **CAMMatch** control signals

Outputs - 12-bit **OutCodeFIFO** bus, 20-bit **CAMOutData** bus, 12-bit **CAMOutAddr** bus, **ReadCharFIFO**, **WriteCodeFIFO**, and **SearchCAM** control signals

The interface is identical to that of **compress**, except for the removal of the **WriteCAM** and **DoRefresh** control signals. The same assumptions are made about the interface; in addition, it is assumed that whenever **SearchCAM** is asserted, in addition to performing the normal search operation, the DCAM will also write the value on the **CAMOutData** bus to the address specified by **CAMOutAddr**. The module is only required to implement the LZW compression algorithm. The refresh cycle is now assumed to be controlled by the master control module, which will halt the compressor when a refresh is required.

RTL Description

The RTL for **compress_2** is shown in Figure 17. There are now only two states, the reset state and the compression (non-reset) state. Also

note that the pipelining on all the signals to the DCAM has been removed, except for the latches on **CAMOutAddr**. It is initialized to the address of the next empty table entry (256), and each time a match fails, it is immediately incremented, so the new value is ready by the next clock cycle when a new search will be requested.

It is important to insure that the addition of the automatic write feature does not adversely affect the execution of the algorithm. The most obvious potential problem is the possibility of a search pattern now matching more than one DCAM location. However, close examination of the

```
INPUT       Reset, CAMMatch, InCharFIFO <11:0>, CAMInAddr <11:0>;

OUTPUT      ReadCharFIFO, WriteCodeFIFO, OutCodeFIFO <11:0>, SearchCAM, CAMOutData <19:0>,
            CAMOutAddr <11:0>,          !  Latched

REGISTER    Omega <11:0>, NextCode <11:0>;

If Reset                                !  Initialization stage
    SearchCAM     <=  FALSE
    WriteCodeFIFO <=  FALSE
    ReadCharFIFO  <=  TRUE              !  Read 1st byte to initialize Omega
    Omega         <-  0 . InCharFIFO
    NextCode      <-  0x100             !  First empty table entry

Else
    ReadCharFIFO  <=  TRUE              !  Get next input byte ready
    SearchCAM     <=  TRUE              !  Enable a CAM search
    CAMOutData    <=  Omega.InCharFIFO  !  Omega . K to search for

    !  Note that every time a search is done, the search pattern (Omega . K)
    !  is written to address NextCode.  Thus, if a match fails, the new
    !  string has already been added to the table, and all that is necessary
    !  is to update NextCode to point to the next empty table entry.
    If (CAMMatch)                       !  Omega . K was in the table
        Omega         <-  CAMInAddr     !  Omega  <-  addr (Omega . K)
        WriteCodeFIFO <=  FALSE         !  Don't generate output
        NextCode      <-  NextCode
        CAMOutAddr    <-  NextCode

    Else                                !  It wasn't in the table
        Omega         <-  0.InCharFIFO  !  Omega gets single-char string K
        WriteCodeFIFO <=  TRUE          !  Enable output
        OutCodeFIFO   <=  Omega         !  Output last Omega

        If (NextCode < 0xfff)           !  Update next table address
            NextCode      <-  NextCode + 1
            CAMOutAddr    <-  NextCode + 1

        Else                            !  Table full
            NextCode      <-  NextCode
            CAMOutAddr    <-  NextCode
```

Figure 17 - Single-cycle Compressor RTL

algorithm reveals that a search pattern will never be repeated on two consecutive clock cycles, unless the first search results in a mismatch (the table entry is not located). If the pattern matches, it will be present in the DCAM in two different locations (the original one and the next free entry), but on the following clock cycle, a different search pattern will be used, so a multiple match cannot result. If the match fails, the entry added to the table will be unique, and the search on the following clock cycle can still only match one location.

This design will process input bytes at a constant rate of one per clock cycle, again provided that the input and output FIFOs can maintain this rate. This is a significant performance advantage, unless it is necessary to substantially reduce the clock rate to allow the DCAM time to perform the enhanced search operation. Note that the same assumptions are made regarding the FIFO capacities that were made in the design of the **compress** module, and they are also valid providing a halt state is added.

A timing diagram detailing the compressor signals is shown in Figure 18. The input is "abcab", as in the previous example, and the output is 061 062 063, with the final code, 100, left in the *Omega* register. Note the significant decrease in complexity and in the number of clock cycles required to compress the input.

Module Generation

The **BDS** and **bdnet** files for this module are a simple translation of the **RTL**. The complete source files are included as Figures 34 and 35 in Appendix B. Note that the finite field sequencer is no longer required (it will instead be implemented in the master controller module). The

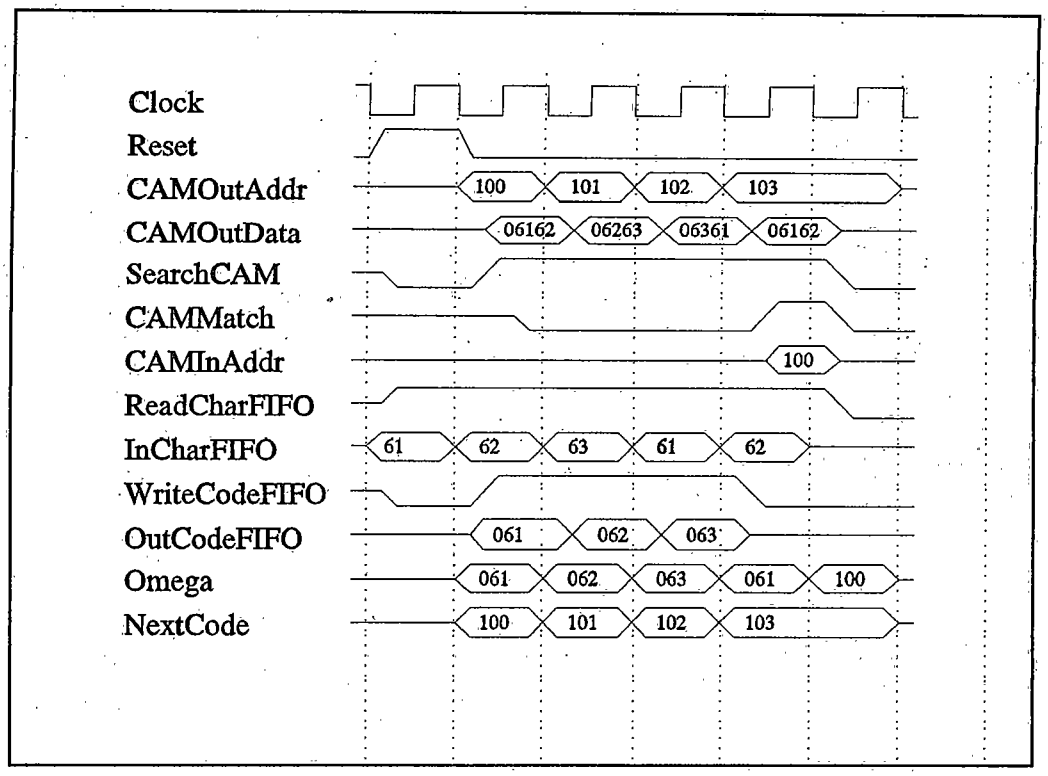


Figure 18 - Compress_2 Module Timing Diagram

module has been successfully generated using the CAD tools. The **chipstats** analysis of the placed and routed module reports 160 standard cell instances, 185 nets, and a resulting area of 936 by 1275 microns (1193400 square microns, or just slightly over one square millimeter). Of this, 0.551 square millimeters are instance area (46.2 percent) and 0.642 square millimeters are routing and empty area (53.8 percent).

The reduction of the module's area requirements to one third of **compress**'s is due primarily to the elimination of several flip-flops (the latches on the DCAM interface and several registers), the reduction from three states to two (Reset and the main loop), and the removal of the refresh control logic. The reduced area requirements and increased

performance provide a clear incentive for implementing a DCAM capable of the enhanced mode of operation.

Module Testing

The module has been tested using the same test cases as **compress**; the scripts were just updated to correctly check the execution paths through the reduced logic. In addition, the **musa** script generator has been updated to produce scripts for the new logic and run on the same set of inputs described above. The module verifies correctly for all inputs tried.

CHAPTER 8

THE DECOMPRESSOR MODULE

The design of the decompressor module is similar to that of the compressor. It is also divided into two separate modules; the first is the standard module **decompress**, designed to accompany **compress**. It also produces the required refresh cycles for the DCAM. Although there is no change necessary in the decompressor to accommodate the DRAM enhancement, since this only affects the search operation and the decompressor utilizes only read and write operations, the **decompress_2** module takes advantage of the fact that the master controller is now responsible for generating refresh signals to significantly simplify the logic.

Module Design to Accompany First Compressor

The normal mode decompressor is presented in this section. Note that the basic design does not include logic to allow the master controller to halt the decompressor when an I/O wait occurs or to take into account limitations on the length of accumulated strings imposed by the compressor. This functionality will be added in Chapter 9.

External Reference Specifications

The **decompress** module will have the following interface:

Inputs - 12-bit **InCodeFIFO** bus, 20-bit **CAMInData** bus, **Reset**

Outputs - 8-bit **StackData** bus, 20-bit **CAMOutData** bus, 12-bit **CAMOutAddr** bus, **WriteStack**, **EndOfString**, **ReadCAM**, **WriteCAM**, and **DoRefresh** control signals

As in the compressor modules, the interface to the DCAM assumes that the master controller module will reset the DCAM at the same time the compressor is reset, and that it will not assert the **Compress** signal if decompression is being performed. The interface to the input and output FIFOs also assumes that they will be reset by the master controller. Note that the FIFO full and empty signals are not present in the decompressor interface; it will be the responsibility of the master controller to handle all input and output buffering conditions and to halt the decompressor as necessary.

The module is required to implement the LZW decompression algorithm described in Chapter 2, and it must also provide sufficient refresh cycles to maintain the DCAM's contents.

RTL Description

The RTL for the decompressor is shown in Figure 19. It has been divided into five states: two reset and initialization states, a state to check for the special decompressor condition where the input code is not in the table, a state that implements the basic decompression loop, and a state to perform the DCAM refresh. It is important to note that the outputs to the DCAM are latched, for the same reasons they were pipelined in the compressor; that is, to ensure that the address, data, and control signals to the DCAM remain constant throughout the entire clock cycle. Because of this pipelining, the reads that are requested in states 01 and 10, the write in state 10, and the refresh in state 11 are not actually performed until the following clock cycle.

The refresh sequencer is identical to the one used in the compressor. Note that a refresh is generated once for every input code that is

```
INPUT           CAMInData <19:0>, InputCodeFIFO <11:0>, Reset

OUTPUT      StackData <7:0>, WriteStack, EndOfString,
                ! Following outputs are latched.
                CAMOutData <19:0>, CAMOutAddr <11:0>, ReadCAM, WriteCAM, DoRefresh

REGISTER        FinChar <7:0>, InCode <11:0>, OldCode <11:0>, NextCode <11:0>,
                State <1:0>

If Reset                                    !  Initialization stage

    WriteCAM, DoRefresh  <-  0
    WriteStack, EndOfString  <=  0

    ReadCodeFIFO <=  1                      !  Read 1st code to initialize Omega
    ReadCAM      <-  1                      !  Read char for that code

    CAMOutAddr   <-  InputCodeFIFO          !     from the string table
    OldCode      <-  InputCodeFIFO

    NextCode     <-  0x100
    Refresh      <-  0x3ff
    State        <-  00

Else if (State == 00)                       !  Second half of initialization

    FinChar      <-  CAMInData <7:0>        !  Character portion of Omega . K
    WriteStack   <=  1
    EndOfString  <=  1
    StackData    <=  CAMInData <7:0>        !  Output first K
    ReadCodeFIFO <=  1                      !  Get next input code to start
    InCode       <-  InputCodeFIFO          !    the decompression loop

    ReadCAM, WriteCAM, DoRefresh  <-  0

    NextCode     <-  NextCode               !  Maintain register contents
    Refresh      <-  Refresh
    OldCode      <-  OldCode
    State        <-  01

Else if (State == 01)                       !  Setup for decompression loop
    If InCode == NextCode                   !  Special case - InCode not in table
        WriteStack  <=  1
        StackData   <=  FinChar
        CAMOutAddr  <-  OldCode
    Else                                    !  Normal mode - InCode in table
        WriteStack  <=  0
        CAMOutAddr  <-  InCode

    ReadCAM      <-  1
    WriteCAM     <-  0
    EndOfString  <=  0

!   The next statement is to get the next refresh address.  The only time
!   that this state doesn't follow State 11, when refresh is done, is on
!   initialization, and it is not crucial that Refresh have the value 0x3ff
!   on the first refresh cycle, so there is no conditional preceding it.

    Refresh      <-  NextValue (Refresh)
    DoRefresh    <-  0

    NextCode     <-  NextCode               !  Maintain register contents
    OldCode      <-  OldCode
    InCode       <-  InCode
    ReadCodeFIFO <=  0

    State        <-  10
```

Figure 19 - Normal Mode Decompressor RTL (cont. on next page)

```
Else if (State == 10)                    !  Main decompression loop
    WriteStack   <=  1                   !  Push chars from table as the
    StackData    <=  CAMInData <7:0>     !    string is being reversed

    If CAMInData <19:8> != NULL_PREFIX   !  Haven't reached start of string
        ReadCAM      <-  1               !  Continue tracing string back
        CAMOutAddr   <-  CAMInData <19:8>

        EndOfString  <=  0
        WriteCAM     <-  0
        OldCode      <-  OldCode         !  Maintain register contents
        State        <-  10              !  Continue in loop

    Else                                 !  Reached head of string
        ReadCAM      <-  0
        EndOfString  <=  1               !  Can start outputting string now
        FinChar      <-  CAMInData <7:0> !    Record first char of string
        OldCode      <-  InCode

        If NextCode != 0xfff             !  String table not full yet
            WriteCAM     <-  1           !  Place new entry in table
            CAMOutData   <-  OldCode . CAMInData <7:0>
            CAMOutAddr   <-  NextCode
        Else                             !  No room for new entry
            WriteCAM     <-  0

        State            <-  11          !  Done with loop - go do refresh

    ReadCodeFIFO, DoRefresh  <-  0
    NextCode     <-  NextCode            !  Maintain register contents
    Refresh      <-  Refresh
    InCode       <-  InCode

Else if (State == 11)                    !  DCAM Refresh state
    DoRefresh    <-  1
    CAMOutAddr   <-  Refresh . 00

    ReadCodeFIFO <=  1                   !  Get next input code to set up
    InCode       <-  InputCodeFIFO       !    for decompression loop

    WriteStack, EndOfString  <=  0
    ReadCAM      <-  0

    If WriteCAM                          !  Added string to table last state,
        NextCode     <-  NextCode + 1    !    update the next code value
    Else
        NextCode     <-  NextCode        !  Maintain register contents
    WriteCAM     <-  0

    Refresh      <-  Refresh
    OldCode      <-  OldCode
    FinChar      <-  FinChar             !  Maintain register contents
    State        <-  01                  !  Go back up and do it again
```

Figure 19 (cont.) - Normal Mode Decompressor RTL

decompressed. Decompression of the code takes one cycle to check for the special input condition, and one cycle for each byte in the decompressed string. Since the strings will be limited to 128 bytes in length (the logic to perform this is described in Chapter 9), this guarantees that a refresh will performed at least every 130th clock cycle.

This design will write output bytes to the string reversal module at the rate of one per clock cycle, with a pause for two cycles for each input code processed, provided that the input and output FIFOs can maintain this rate. The module assumes that the input FIFO will always have a code ready, and that the string reversal module will always have room for another byte. As in the compressor, these assumptions are valid provided the master controller handles the I/O buffer overflow and underrun conditions and generates a signal to halt the decompressor. The addition of the halt capability is presented in Chapter 9.

As in the compressor, every register is assigned a value in each state (unless it is acceptable for the register to lose its contents) to assure that garbage data is not written into the register.

A timing diagram detailing the decompressor signals is shown in Figure 20. The input is 061 100 062 101 061, and the output is "aaabaab", although each substring is output in reverse order.

Module Generation

The **BDS** and **bdnet** files implementing this **RTL** are included as Figures 36 and 37 of Appendix B. The finite field sequencer subroutine is identical to the one used in the **compressor** module. One problem that arose in the process of creating the module was the generation of the logic to perform the comparison operation

If InCode_Q EQL NextCode_Q then

to check the special condition where the input code is not in the table. **Bdsyn** easily generates comparisons of an input with a constant value by exclusive-oring the input with the value and checking the result for zero. This is the same procedure that is used to compare two inputs; however,

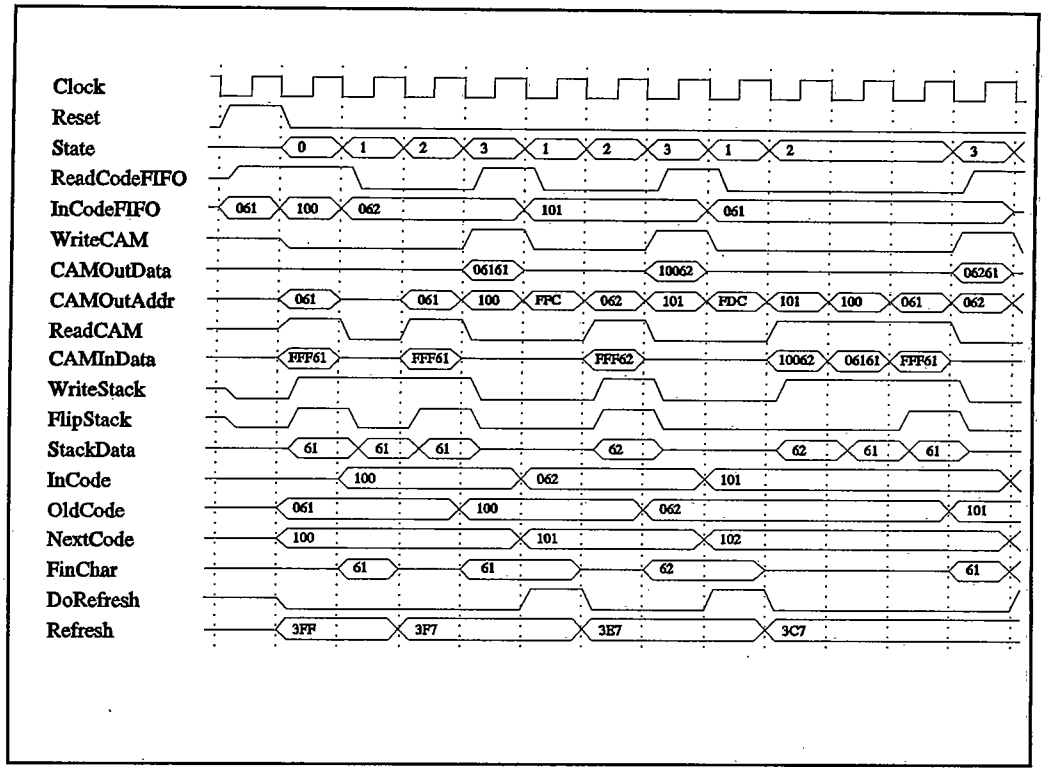


Figure 20 - Decompress Timing Diagram

misII attempts to optimize the exclusive-or equations. This process takes an inordinate amount of time to perform for inputs of more than about six bits, and increases rapidly as the number of bits in the inputs increases.

In order to force **misII** to leave the comparison in an unoptimized form, a special comparator is added with **bdnet** (at the same as the flip-flops) which simply connects each bit of the two inputs to be compared to the inputs of an exclusive-or gate. The output of this exclusive-or array is then checked for equality to zero in the **BDS** file. For instance, the comparison shown above would be replaced by the following **BDS** statement:

```
If Comparator EQL 0 then     !   NextCode_Q == InCode_Q
```

where Comparator is a twelve-bit input that is connected to the output of

an array of exclusive-or gates in the **bdnet** file. An example of this is shown in the **bdnet** file in Figure 37. This modifications allows the generation of the module in a reasonable amount of time.

The module has been successfully generated using the CAD tools. The **chipstats** analysis of the placed and routed module reports 425 standard cell instances, 461 nets, and a resulting area of 1648 by 2627 microns (4329296 square microns, over four square millimeters). Of this, 1.788 square millimeters are instance area (41.3 percent) and 2.541 square millimeters are routing and empty area (58.7 percent).

Module Testing

Musa test scripts have been generated to simulate the module for the input code streams "061 100 101 102 100" and "061 062 061 100 061 063 100 105 062 107 102 061" (the outputs that were produced in the compressor tests). The tests verify that the module does correctly produce the strings "aaaaaaaaaaaa" and "abaabacabcabcabaaa", although each substring is generated in reverse order (for output to the string reversal module).

The utility written to generate **musa** scripts for the **compress** module also generates the corresponding scripts for **decompress**. The program has been used to decompress all of the files compressed during the testing of **compress**, and the resulting scripts have been used to verify that **decompress** properly performs the decompression of all those inputs (including the two that cause the string table to fill up).

Module Design to Accompany Single-Cycle Compressor

As mentioned above, it is not absolutely necessary to modify the decompressor to deal with the search enhancement to the DCAM. However,

the compressor now requires that the master control module perform refreshes, so the refresh generation logic can be removed from the accompanying decompressor. A further simplification involves elimination of DCAM reads if the address to be read is less than 256. It is guaranteed that these reads will return the **null** prefix and the low-order byte of the string as the table entry, so there is no need to perform the read. Note that this eliminates the need for the **null** prefix; however, it will still be reserved, in case it is required to mark DCAM words as empty.

External Reference Specifications

The **decompress_2** module will have the following interface:

Inputs - 12-bit **InCodeFIFO** bus, 20-bit **CAMInData** bus, **Reset**

Outputs - 8-bit **StackData** bus, 20-bit **CAMOutData** bus, 12-bit **CAMOutAddr** bus, **WriteStack**, **EndOfString**, **ReadCAM**, and **WriteCAM** control signals

The interface is identical to that of **decompress_2**, except for the removal of the **DoRefresh** control signal. The same assumptions are made about the DCAM and I/O buffer interfaces. The module is only required to implement the LZW decompression algorithm. The refresh cycle is now assumed to be controlled by the master control module, which will halt the decompressor when a refresh is required.

RTL Description

The RTL for **decompress_2** is shown in Figure 21. There are now only three states: the reset state, a loop setup state which handles the problem input condition, and the main decompression loop (which traces the string backward through the string table and produces bytes to be reversed). Note that pipelining of the signals to the DCAM is still in

place, to insure that they are asserted for a sufficient length of time during the clock cycle.

This module does not have a significant performance advantage over **decompress**. The only effective timing difference is the elimination of the refresh state, so that the output to the string reversal module is only interrupted for one cycle for each input code processed. However, it does improve a critical timing path in state 1 significantly. In **decompress**, when the head of the string is reached, the final byte returned by the DCAM must immediately be latched into both the **CAMOutData** register (so the new string entry can be written on the following clock cycle) and the **FinChar** register. Now, since the DCAM is not even read to determine the last byte in the string and the data to be written to the DCAM is already present in registers, this timing path is relaxed considerably.

A timing diagram detailing the decompressor signals is shown in Figure 22, for the same inputs as in the previous example.

Module Generation

The **BDS** and **bdnet** files for this module are a simple translation of the **RTL**. They are included as Figures 38 and 39 of Appendix B. Note that the finite field sequencer is no longer required (it will instead be implemented in the master controller module). The module has been successfully generated using the CAD tools. The **chipstats** analysis of the placed and routed module reports 326 standard cell instances, 362 nets, and a resulting area of 1432 by 2379 microns (3406728 square microns, or over three square millimeters). Of this, 1.427 square millimeters are instance area (41.9 percent) and 1.980 square millimeters are routing and empty area (58.1 percent).

```
INPUT     Reset, InCodeFIFO <11:0>, CAMInData <19:0>;
OUTPUT    ReadCodeFIFO, OutCodeFIFO <11:0>, WriteStack, EndOfString, StackData <7:0>,
             !   Latched outputs.  ReadCAM & CAMOutAddr also used as registers.
          ReadCAM, WriteCAM, CAMOutAddr <11:0>, CAMOutData <19:0>
REGISTER  State, FinChar <7:0>, InCode <11:0>, OldCode <11:0>,
          NextCode <11:0>;

If Reset                                  !  Initialization stage
    ReadCodeFIFO  <-  TRUE                !   Read 1st code to initialize OldCode
    NextCode      <-  0x100
    OldCode       <-  InCodeFIFO
    FinChar       <-  InCodeFIFO <7:0>
    StackData     <=  InCodeFIFO <7:0>    !   Output the character portion of it
    WriteStack, EndOfString, ReadCodeFIFO  <=  TRUE
    ReadCAM, WriteCAM  <-  FALSE
    State         <-  0

Else if (State == 0)
    If (InCodeFIFO == NextCode)           !   Special case - InCode not in table
        WriteStack  <=  TRUE
        StackData   <=  FinChar
        CAMOutAddr  <-  OldCode
    Else                                  !   Normal mode - InCode in table
        WriteStack  <=  FALSE
        CAMOutAddr  <-  InCodeFIFO
    If (CAMOutAddr_D <11:8> == 0)         !   Single character string - the
        ReadCAM     <-  FALSE             !     byte is just the last 8 bits
    Else                                  !     of the address
        ReadCAM     <-  TRUE
    InCode        <-  InCodeFIFO          !   Latch input code for later use
    ReadCodeFIFO  <=  TRUE                !   and get the next one ready
    EndOfString <=  FALSE
    WriteCAM      <-  FALSE
    NextCode      <-  NextCode            !   Maintain register contents
    OldCode       <-  OldCode
    State         <-  1

Else if (State == 1)                      !   Main decompression loop
    WriteStack  <=  TRUE                  !   Push next char from string
    If (ReadCAM)                          !   Haven't reached start of string
        StackData   <=  CAMInData <7:0>
        CAMOutAddr  <-  CAMInData <19:8>
        If (CAMOutAddr_Q <11:8> == 0)     !   Same as in State 0
            ReadCAM     <-  FALSE         !   Reached end of string
        Else
            ReadCAM     <-  TRUE          !   Continue tracing string back
        EndOfString   <=  FALSE
        WriteCAM      <-  FALSE
        OldCode       <-  OldCode         !   Maintain register contents
        NextCode      <-  NextCode        !   Maintain register contents
        State         <-  1               !   Continue in loop
    Else                                  !   Reached head of string
        StackData   <=  CAMOutAddr <7:0>
        ReadCAM     <-  FALSE
        EndOfString <=  TRUE              !   Can start outputting string now
        FinChar     <-  CAMOutAddr <7:0>  !    Record first char of string
        OldCode     <-  InCode
        If (NextCode != 0xfff)            !   String table not full yet
            WriteCAM    <-  TRUE          !   Place new entry in table
            CAMOutData  <-  OldCode . CAMOutAddr <7:0>
            CAMOutAddr  <-  NextCode
            NextCode    <-  NextCode + 1  !    Move on to next empty entry
        Else                              !   No room for new entry
            WriteCAM    <-  FALSE
            NextCode    <-  NextCode
        State       <-  0                 !   Done with loop - go to next input
    ReadCodeFIFO  <=  FALSE
    InCode        <-  InCode              !   Maintain register contents
```

Figure 21 - Enhanced Decompressor RTL

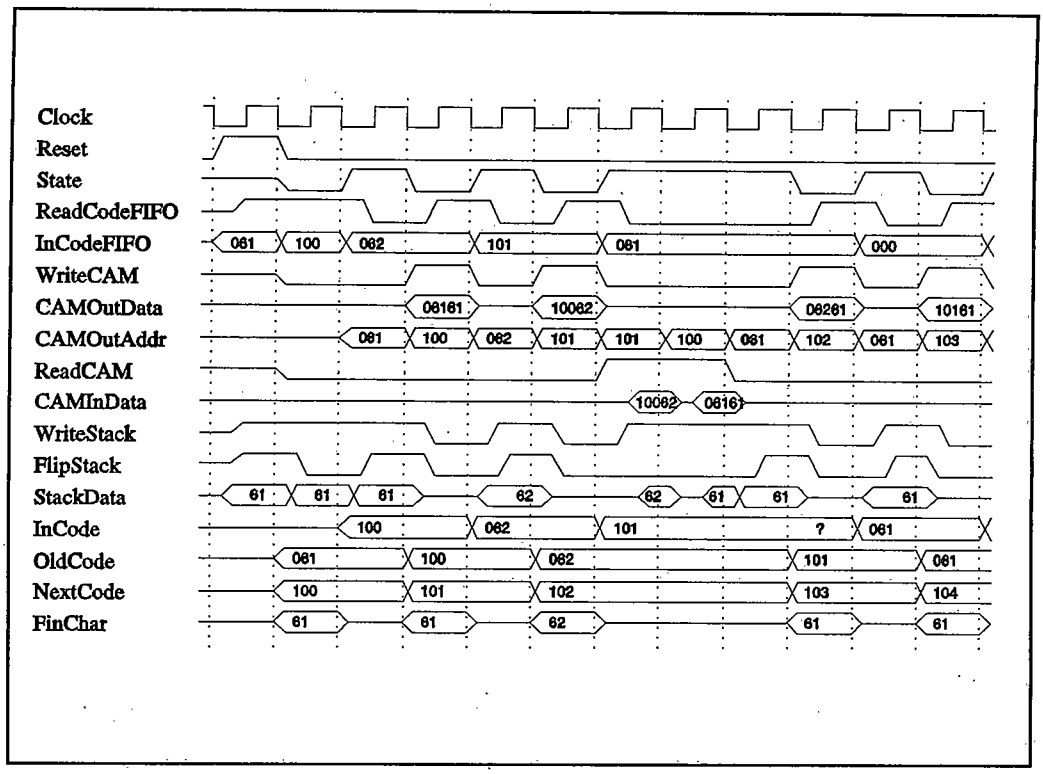


Figure 22 - Decompress_2 Timing Diagram

The reduction of the module's area requirements to roughly three quarters of **decompress**'s is not dramatic, but the logic is much cleaner and the critical timing paths are improved, so this module is more desirable for the final implementation.

Module Testing

The module has been tested using the same test cases as **decompress**; the scripts were just updated to correctly check the execution paths through the reduced logic. In addition, the **musa** script generation utility has been updated to produce scripts for the new logic and has been run on the same set of test cases. The module verified correctly for all input cases tried.

CHAPTER 9

MERGING THE COMPRESSOR AND DECOMPRESSOR

Since the compressor and decompressor can never run concurrently (due to contention for the string table), it is possible to merge the two modules. This should allow the sharing of several registers and logic functions (such as the refresh address sequencer). It should also allow **misII** to perform additional global optimizations.

The **merged** module is formed by combining **compress** and **decompress**. The **ERS** consists of the combination of the **ERS**es for the individual modules, with the addition of a **DoCompress** input which allows the master controller module to specify which function of the module should be executed. The **RTL** likewise consists of a combination of the two individual **RTL**s. The *input* and *output* sections are just the consolidation of those sections from the two component modules. The **compress** *register* section remains intact, and is merged with the **decompress** registers. The following translations are made to register names in the decompressor in order to share registers: **InCode** becomes **Omega**, and **FinChar** becomes **K**. The functional descriptions from the two individual modules are just patched together with the conditional statement

```
If (DoCompress)
      ! do compressor functions
Else
      ! do decompressor functions
```

The **BDS** is formed in a similar fashion. The input and output terminal lists are merged (with the addition of the **DoCompress** input), and

the main routines from the two modules are included as separate **Compressor** and **Decompressor** routines. A main routine is created that just calls one routine or the other, depending on the state of **DoCompress**.

The resulting module has been created using the CAD tools. It contains 590 standard cell instances and 648 nets, and has a resulting area of 1856 by 3467 microns (6434752 square microns, nearly six and one half square millimeters). Of this, 2.283 square millimeters are instance area (35.5 percent) and 4.151 square millimeters are routing and empty area (64.5 percent). This is a definite savings over including both of the original modules; they have a combined total of 727 instances, 788 nets, and an area of 7.4 square millimeters. This is an area savings of 13 percent.

The module has been successfully simulated with each of the hand-generated compression and decompression cases developed for the two original modules. In addition, the script generation utility has been updated to generate test scripts for **merged**, and it has been verified on a number of input files.

The **merged_2** module is formed by combining **compress_2** and **decompress_2** using an identical process. The only register that is renamed in the decompressor is **InCode**, which is merged with **Omega**. However, since the compressor expects the output data bus to the DCAM to be unlatched, while the decompressor expects it to be latched, a multiplexer is added that selects the appropriate signal for output based on the state of **DoCompress**. In addition, in each of the routines the control signals that are not used by that routine are disabled. For instance, the compressor disables the **ReadCAM** output, and the decompressor disables **SearchCAM**.

The resulting module contains 518 standard cell instances and 576 nets, and has an area of 1688 by 3163 microns (5339144 square microns). Of this, 1.925 square millimeters are instance area (36.1 percent), and 3.414 square millimeters are routing and empty area. The combined total for the original modules is 486 instances, 547 nets, and an area of 4.6 square millimeters. The merged module has increased in size somewhat, but it does contain additional logic not in the original modules. Also, it is still 17 percent smaller than **merged**. Because of the area savings and functional advantages, this module has been chosen for the addition of the logic to allow a halt state and to limit the length of compressed strings.

Again, **merged_2** has been simulated with each of the test cases, and the script generation utility has been updated to produce appropriate test scripts. The module has successfully passed all of the tests attempted.

Additional Functionality

Halt Signal Handling

Chapters 7 and 8 briefly discuss the addition of a halt signal to the compressor and decompressor to allow the master control module to temporarily suspend their execution, so that input FIFOs can fill up, output FIFOs can empty out, and DCAM refreshes can be generated. This capability has been added to both the compressor and decompressor portions of **merged_2**.

The goal is to add a halt state similar to the reset state. When the **Halt** signal is asserted, the compressor or decompressor should immediately (asynchronously) enter this state. All outputs should be disabled (such as FIFO read and write control signals), and the contents of all

registers should be maintained, so that when the **Halt** signal is disabled, the state that was interrupted can be re-entered and execution can continue. In addition, the compressor should also include a **DumpOmega** signal; if the halt state is entered and it is asserted, the current contents of the **Omega** register should be written to the output code FIFO. This allows the master controller to flush the last code accumulated out of the compressor when the end of the input stream has been processed.

The **RTL** required to add the halt state is shown in Figure 23. Note that, since the outputs to the DCAM are latched and also used as registers in the decompressor, they cannot just be disabled. The master controller module will be responsible for disconnecting them from the DCAM in order to perform refreshes (see Chapter 11 for details).

```
If DoCompress
    If Reset
        ...
    Else if Halt
        SearchCAM, ReadCharFIFO  <= FALSE
        If (DumpOmega)
            WriteCodeFIFO   <=  TRUE
            OutCodeFIFO     <=  Omega
        Else
            WriteCodeFIFO   <=  FALSE
        Omega          <-  Omega
        CAMOutAddr     <-  CAMOutAddr
        StrLen         <-  StrLen
    Else
        ...
Else
    If Reset
        ...
    Else if Halt
        WriteStack, FlipStack, ReadCharFIFO  <=  FALSE
        WriteCAM     <-  WriteCAM
        ReadCAM      <-  ReadCAM
        CAMOutAddr   <-  CAMOutAddr
        D_CAMOutData <-  D_CAMOutData
        NextCode     <-  NextCode
        OldCode      <-  OldCode
        InCode       <-  InCode
        FinChar      <-  FinChar
        OmitNextStr  <-  OmitNextStr
        StrLen       <-  StrLen
        State        <-  State
    Else if (State == 0)
        ...
```

Figure 23 - RTL for Addition of Halt States

String Length Limiting

As discussed previously, it is necessary to limit the length of strings accumulated in the compressor, in order to bound the size of the string reversal mechanism. In order to implement this, a length counter must be added to the compressor and incremented each time a search of the string table is successful and a new byte is added to the accumulated string. When the string's length reaches 128 bytes, its code should be written to the output code FIFO, and the string should be reinitialized to the single-byte string formed from the current input byte. This will prevent any strings of more than 128 bytes from being added to the string table or being output.

A corresponding length counter must be added to the decompressor as well, since it must recreate the string table exactly as the compressor has. The decompressor should count the length of each string that it is generating; if the total length is 128 bytes, a flag is set, and the end of the string is processed normally. However, at the end of the next decompressed string, if that flag is set a new entry is not added to the string table. This pipelined operation must be done because the decompressor is one step behind the compressor while creating the string table.

As mentioned previously, the length counter is implemented using an eight-bit finite-field sequencer. Its value is reset to 255 (hexadecimal ff) at the beginning of each string, and "incremented" in the normal shift register fashion as the string grows. It is then checked against the 128th value in the sequence (35, or hexadecimal 23, for the minimum-weight polynomial x^8 + x^4 + x^3 + x^2 + 1) to see if the maximum string length has been reached.

Complete Merged Module

The complete set of source files for the **merged_2** module (**RTL**, **BDS**, **bdnet**, and **musa**), with the additional functionality described above, is included in Appendix A. The module has been successfully created using the CAD tools; it contains 630 instances, 690 nets, and has an area of 1896 by 3835 microns (7271160 square microns). Of this, 2.314 square millimeters is instance area (31.8 percent), and 4.956 square millimeters is routing and empty area (68.2 percent). Even with the additional logic, the module is smaller than the total area of **compress** and **decompress**.

The **musa** script generation utility has been updated to produce test scripts for this module. The module verifies correctly for each test input attempted. Assuming the DCAM will be implemented with the automatic write on search enhancement, this is the module that will be included in the IC.

CHAPTER 10

THE STRING REVERSAL MODULE

Although the RAMs used as the ring buffers in the string reversal mechanism are custom modules, the control logic is implemented using standard cells. The module design assumes that the ring buffers are implemented using dual-ported RAM, capable of concurrent read and write operations. Other options for the RAM implementation are discussed in Chapter 4.

Module Controller Design

External Reference Specifications

The **stack** module will have the following interface:

Inputs - 8-bit **StackData** bus, **WriteStack**, **EndOfString**, **Reset**, **OutCharFIFOFull**

Outputs - 8-bit **OutCharFIFO** bus, **WriteCharFIFO**, **StackOverRun**, **StackEmpty**

In addition, it will have the following internal interface with the RAMs used for the ring buffers:

Inputs - 8-bit **RStackInData**, **StrHeadInData**, and **StrTailInData** busses

Outputs - 8-bit **RStackOutData**, **StrHeadOutData**, **StrTailOutData**, **RStackReadAddr** and **RStackWriteAddr** busses, 7-bit **StringQReadAddr** and **StringQWriteAddr** busses, **ReadRStack**, **WriteRStack**, **ReadStringQ**, and **WriteStringQ** control signals

The module must implement the string reversal algorithm given in Chapter 4. It will provide the **StackOverRun** and **StackEmpty** flow control

signals for use by the master controller module to control data coming from the decompressor. The decompressor interface is fairly simple; when the decompressor has a byte to output, it will place it on **StackData** and assert **WriteStack**. If it is the last character in the string, **EndOfString** will also be asserted. It is expected that the stack module will latch the byte on the next negative clock edge. The interface to the output byte buffer is similar, without the **EndOfString** control signal. If the FIFO asserts **OutCharFIFOFull**, it assumes that the stack will not attempt to write further data; any write requests will be ignored. The stack assumes the same procedure if it asserts **StackOverRun**. **StackEmpty** will be asserted whenever there are no bytes remaining in its ring buffer.

RTL Description

Due to its length, the **RTL** description of the module is included as Figure 40 in Appendix B. Note that bytes output to the stack are latched into **NewChar** before they are processed by the stack. This stage of pipelining is present because the decompressor will always write a byte to the stack during the reset cycle, before the stack data structures are initialized and ready to handle the data. In addition, this pipeline helps ease the critical timing path in the decompressor; a write request must only meet the setup time of the **NewChar** register, with no additional switching delays added by the stack.

The non-reset state is broken up into two sections: one to handle insertions into the ring buffer (write requests from the decompressor), and another to handle removals from the buffer (write requests to the output FIFO). Note that when the stack fills up, insertions are halted, but outputs to the FIFO continue. Also, if the output byte buffer fills

up, outputs to the FIFO are halted, but insertions are still allowed (until the ring buffer or string queue fills up).

A timing diagram detailing the operation of **stack**'s signals is shown in Figure 24.

Module Generation

The **BDS** and **bdnet** source for the module are shown in Figures 41 and 42 of Appendix B. Note that several comparators are created using **bdnet** to avoid the **misII** optimization problem described in Chapter 8.

The module has been successfully generated using the CAD tools. It contains 459 standard cell instances, 498 nets, and occupies an area of 1504 by 2611 microns (3926944 square microns). Of that, 1.616 square millimeters are instance area (41.1 percent), and 2.311 square millimeters are routing and empty area (58.9 percent).

Module Testing

The module has been tested fairly extensively with **musa**. Boundary cases that were examined include two 128-byte strings in succession and a 128-byte string followed by 128 one-byte strings. The same cases, and a series of 129 one-byte strings, were also tested with **OutCharFIFOFull** asserted to verify that the **StackOverRun** was asserted properly. It properly handles both ring buffer and string queue full conditions. All test cases were executed correctly.

Addition of Musa RAM Models

As mentioned in the description of **musa** in Chapter 6, it is primarily used as a switch-level simulator for MOS transistors, but it also

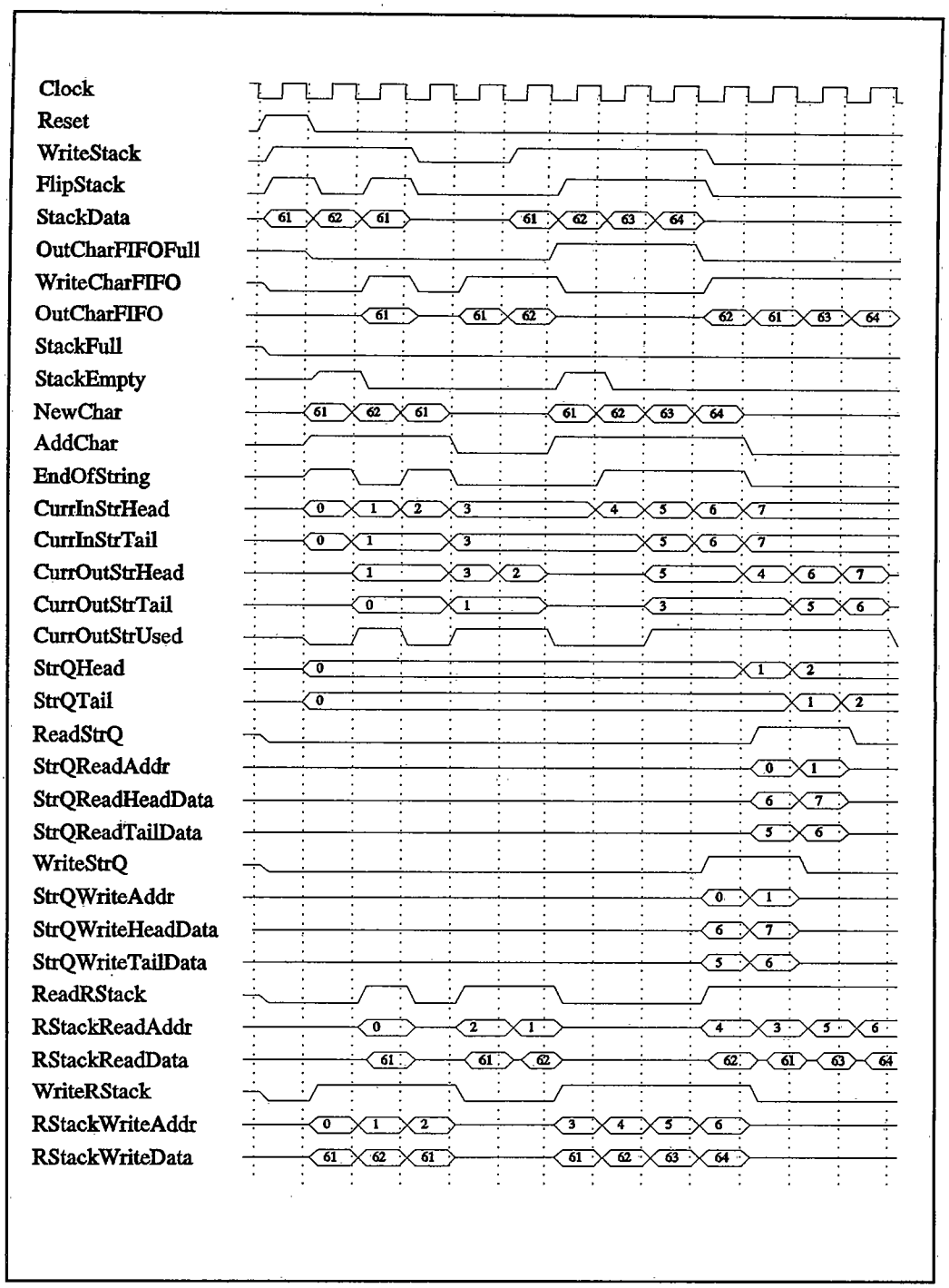


Figure 24 - String Reversal Timing Diagram

includes enhancements to simulate high-level devices, including RAMs. In order to perform more complete tests of the stack module, a set of these

RAMs has been added to the stack. This allows the simulation to ignore the RAM interface lines and concentrate on the decompressor and output FIFO interfaces. The **bdnet** file included as Figure 43 of Appendix B is used to add the RAMs to **stack**. Note that these RAMs are only functional models, not actual physical circuits. As such, they are only useful for **musa** simulations.

Several difficulties were encountered in the interface of the RAM to the controller logic. In order to simulate a dual-ported RAM using the single-port models, the read and write signals were initially gated using the clock and inverted clock signals, respectively. Also, the read and write address and data busses were selected using a multiplexer controlled by the clock. This would supposedly allow the write operations to be executed during the first half clock cycle and the reads to be executed during the second half. However, the timing path to switch the address and data busses was shorter than the one that disabled the write and enabled the read, so as the clock transition was simulated, data was erroneously written to the next read location before write was disabled.

In order to correct this problem, three additional signals, used only for simulation, were introduced into the bdnet file: **ReadEnable**, **WriteEnable**, and **AddressSelect**. The sequence required to simulate one clock cycle is shown in the **musa** file in Figure 44 of Appendix B. Basically, **AddressSelect** is set to select the write address, **WriteEnable** is asserted, **WriteEnable** is deasserted, **AddressSelect** is deasserted to select the read address, **ReadEnable** is asserted, **ReadEnable** is deasserted, then the clock is toggled high then low to produce the negative edge required by all the flip-flops in the module. Every time a signal is changed, the

changes are propagated through the network before another signal is changed. This process allowed successful use of the RAM modules.

With the RAMs in place, **stack** was simulated with several other test inputs, all of which executed correctly. The **stack** module is ready for integration with the custom RAM modules as soon as they are designed.

CHAPTER 11

THE MASTER CONTROLLER MODULE

Module Design

External Reference Specifications

The final module is the master controller. The **control** module will have the following interface:

Inputs - **ResetIn, DoCompressIn, InCharFIFOEmpty, InCharFIFOFull, InCharDataEnd, InCodeFIFOEmpty, InCodeFIFOFull, InCodeDataEnd, OutCodeFIFOEmpty, OutCodeFIFOFull, OutCharFIFOEmpty, StackFull, CAMOutAddrIn <11:0>, ReadCAMIn, WriteCAMIn**

Outputs - **Halt, DumpOmega, DoRefresh, OperationComplete, CAMOutAddr <11:0>, ReadCAM, WriteCAM, Reset, DoCompress, DumpOutCode**

This module is responsible for interfacing the external and internal control signals, handling flow control between the compressor and decompressor and the I/O buffers, and generating the refresh cycles for the DCAM. The external interface currently consists only of the **ResetIn**, **DoCompressIn**, and **OperationComplete** signals. **ResetIn** should be asserted for at least one clock cycle by the external circuitry. During this clock cycle, **DoCompressIn** should be asserted if compression is desired, or deasserted if decompression is desired. On the negative clock edge following the assertion of **ResetIn**, it and **DoCompressIn** will be latched into the global signals **Reset** and **DoCompress**. **Reset** will remain asserted for one clock cycle, and should reset the other modules. **DoCompress** will remain constant until the next reset.

Inputs - **ResetIn, DoCompressIn, InCharFIFOEmpty, InCharDataEnd, InCodeFIFOEmpty, InCodeDataEnd, OutCodeFIFOEmpty, OutCodeFIFOFull, OutCharFIFOEmpty, StackFull, CAMOutAddrIn <11:0>, ReadCAMIn, WriteCAMIn**

Outputs - **Halt, DumpOmega, DoRefresh, OperationComplete, CAMOutAddr <11:0>, ReadCAM, WriteCAM, Reset, DoCompress, DumpOutCode**

The refresh cycle will be generated periodically. During normal operation, the **ReadCAMIn**, **WriteCAMIn**, and **CAMOutAddrIn** signals will be connected directly to **ReadCAM**, **WriteCAM**, and **CAMOutAddr**. During a refresh cycle, **ReadCAM** and **WriteCAM** will be deasserted, **CAMOutAddr** will contain the address of the next row of the DCAM to refresh, and the **Halt** and **DoRefresh** signals will be asserted.

The I/O flow control has been designed assuming the I/O buffer interface presented in Chapter 5. If **InCharFIFOEmpty**, **InCodeFIFOEmpty**, **OutCodeFIFOFull**, or **StackFull** is asserted, the **Halt** signal is immediately (asynchronously) asserted, and remains asserted until all four signals are deasserted. If a halt is executed and **InCharFIFOEmpty** and **InCharDataEnd** are asserted, **DumpOutCode** will be asserted, and **DumpOmega** will be asserted for one clock cycle. In addition, if **InCharFIFOEmpty**, **InCharDataEnd**, and **OutCodeFIFOEmpty** are all asserted, or **InCodeFIFOEmpty**, **InCodeDataEnd**, and **OutCharFIFOEmpty** are all asserted, **OperationComplete** is asserted.

RTL Description

The **RTL** description of the module is shown in Figure 25. It uses finite field sequencers to count clock cycles to determine when a refresh should be done and to generate the refresh addresses. **REF_LEN** will be determined by the number of rows in the DCAM, and **CNT_LEN** will be determined by the frequency at which it must be refreshed.

```
INPUT       ResetIn, DoCompressIn, InCharFIFOEmpty, InCharDataEnd,
            InCodeFIFOEmpty, InCodeDataEnd, OutCodeFIFOEmpty, OutCodeFIFOFull,
            OutCharFIFOEmpty, StackFull, CAMOutAddrIn <11:0>, ReadCAMIn,
            WriteCAMIn

OUTPUT      Halt, DumpOmega, DoRefresh, OperationComplete, CAMOutAddr, ReadCAM,
            WriteCAM,
            !   The following outputs are latched
            Reset, DoCompress, DumpOutCode

REGISTER    RefreshCt <CNT_LEN>, RefreshAddr <REF_LEN>

If ResetIn
    Reset       <-  TRUE
    DoCompress  <-  DoCompressIn
    DumpOutCode <-  FALSE
    Halt        <=  FALSE
    DumpOmega   <=  FALSE
    ReadCAM     <=  FALSE
    WriteCAM    <=  FALSE
    OperationComplete   <=  FALSE
    DoRefresh   <=  FALSE
    RefreshCt   <-  CNT_INIT            !   Finite field sequencer dependent
    RefreshAddr <-  REF_INIT

Else
    Reset       <-  FALSE
    DoCompress  <-  DoCompress

    If (RefreshCt == CNT_DONE)
        Halt        <=  TRUE
        DumpOmega   <=  FALSE
        DumpOutCode <-  DumpOutCode
        DoRefresh   <=  TRUE
        CAMOutAddr  <=  RefreshAddr . 0 !   Enough 0s to fill 12-bit buss
        ReadCAM     <=  FALSE
        WriteCAM    <=  FALSE
        RefreshAddr <-  NextAddr(RefreshAddr)

    Else
        DoRefresh   <=  FALSE
        CAMOutAddr  <=  CAMOutAddrIn
        ReadCAM     <=  ReadCAMIn
        WriteCAM    <=  WriteCAMIn
        RefreshAddr <-  RefreshAddr
        If (InCharFIFOEmpty || InCodeFIFOEmpty || OutCodeFIFOFull || StackFull)
            Halt        <=  TRUE

            If (InCharFIFOEmpty && InCharDataEnd && !DumpOutCode)
                DumpOmega   <=  TRUE
                DumpOutCode <-  TRUE
            Else
                DumpOmega   <=  FALSE
                DumpOutCode <-  DumpOutCode

        Else
            Halt        <=  FALSE
            DumpOmega   <=  FALSE
            DumpOutCode <-  DumpOutCode

    RefreshCt   <-  NextCt (RefreshCt)

    If ((InCharFIFOEmpty && InCharDataEnd && OutCodeFIFOEmpty) ||
        (InCodeFIFOEmpty && InCodeDataEnd && OutCharFIFOEmpty))
        OperationComplete   <=  TRUE
    Else
        OperationComplete   <=  FALSE
```

Figure 25 - Master Controller RTL

Module Generation

The **BDS** and **bdnet** source for the module are shown in Figures 45 and 46 of Appendix B. The module has been successfully generated using the CAD tools. It contains 101 standard cell instances, 130 nets, and occupies an area of 712 by 951 microns (677112 square microns). Of that, 0.370 square millimeters are instance area (54.7 percent), and 0.307 square millimeters are routing and empty area (45.3 percent).

Module Testing

The module has been tested for a variety of cases using **musa**. More complete tests will be required when the modules are assembled and the interface signals are all connected.

System Integration and Test

Once the IC's external interface has been fully defined, the controllers for each of the I/O buffers can be designed and generated. At this time, **musa** RAM models can be attached to serve as the ring buffers in each of the FIFOs and all the modules can be connected using **bdnet**. This should allow complete black-box testing of the entire IC, except for the DCAM. It will be necessary to update the **musa** script generation utility to produce test scripts for the IC, in order to simulate the operation of the DCAM. After a complete suite of tests has been conducted, the logic simulation of the IC should be complete.

CHAPTER 12

MANAGING THE DESIGN PROCESS

The similarity between the standard cell design methodology used in this development and typical software development cycles was mentioned in Chapter 6. This similarity can be exploited to take advantage of the body of techniques used in software development to manage large, complex projects.

One method which has been utilized herein is the structured design life-cycle: requirements specification, high-level functional description, code generation and compilation, and testing. The set of documentation and source files developed for each module fits into this framework well. This structured design methodology is frequently used to manage VLSI projects; for example, see the discussion in Chapter 6 of [WESTE]. For the most part, it can be applied to all facets of VLSI design, not just standard cell implementations.

Another tool that can be borrowed from software development that is not so widely acknowledged is the use of automated source code control and configuration management tools. For example, the UNIX tools **make** and **rcs** (or the similar package **sccs**) can be used to greatly simplify the module generation portion of this process.

Make is a utility for controlling the automatic recompilation of source code files. A control file is created which specifies the dependency hierarchy of source code, and **make** can automatically regenerate all

portions of that hierarchy that depend on a file that has been updated. For example, in a C software project, a source file might include a number of header files. This file therefore depends on those header files, and **make** can be used to automatically recompile the file if any of the header files is changed. Another example is the generation of the **:unplaced OCT** view of a module. This view typically depends on the **:logic** view and a **bdnet** file. **make** can automatically rebuild the **:unplaced** view if either the **:logic** view or the **bdnet** file is modified. An example of a **Makefile** (the control file) is shown in Figure 47 of Appendix B. Simply by changing the definition of the module name at the top of the file, the same **Makefile** was used to control the generation of all the modules described in previous chapters. Information on the use of **make** and the syntax of **Makefiles** is available in the online UNIX **man** pages.

Rcs (short for **R**evision **C**ontrol **S**ystem) is a set of tools which aid in revision and configuration management. It allows the storage of multiple versions of a source file in a single archive, and automatically generates revision histories for the archive. Any version of the file can be retrieved, and multiple modification branches can be started from a single revision. After these branches grow as successive modifications are made in each branch, it is possible to merge the new branches together to reform a main branch containing all the revisions from all the sub-branches. A symbolic revision name or number can be assigned to a revision level across a set of source archives, allowing the user to "freeze" a given software configuration for future reference. The use of **rcs** can greatly simplify the task of maintaining revisions of **BDS** files as new functionality is added to modules, or errors are removed. Information on

the usage and syntax of the various **rcs** utilities is available in the online UNIX **man** pages.

These are just two examples of UNIX software development tools which adapt well to the standard cell design methodology in the **OCT** environment. There are a number of other utilities which could be used to assist in the rapid generation and verification of VLSI designs.

CHAPTER 13

CIRCUIT PERFORMANCE ESTIMATES

Timing Extraction Using MisII

In addition to **misII's** normal batch execution mode, which is used to optimize logic and map it into a standard cell library, it provides an interactive mode which can be used to obtain timing information. A library file is required that describes the logic function and timing characteristics of each cell in the standard cell library; Luciano Lavagno at the University of California has recently completed two of these library files for the Mississippi State University cell library, one containing nominal cell parameters and the other containing worst case values. These files contain timing data only for the combinational logic cells in the library.

The procedure followed to obtain a timing profile of a combinational logic circuit involves starting **misII** with no arguments, reading in the **OCT :logic** view to be profiled, loading the library file for the cell library and performing the mapping, then printing out delay information for the circuit. This delay information is presented in the form of the rising edge and falling edge *arrival time*, *required time*, and *slack time* for each node in the circuit, and it can be sorted to produce a list of the nodes with the most critical values. The delay is divided into rising and falling edge information because the cells are not typically symmetrical; that is, they can make one transition faster than the other.

The cell parameters in the library file include the following values for each of the cell's input pins: the phase of the input (**INV**erting, **NONINV**erting, or **UNKNOWN**); the input load (capacitance) of the pin, in arbitrary floating-point units; the rise-time parameters, given as a block delay in nanoseconds and a fanout delay in nanoseconds per unit load (the same units used to define the input load); and the fall-time parameters, as block delay and fanout delay. The input phase is required because an inverting input essentially interchanges the rise time and fall time of the incoming signal.

Calculation of arrival time is performed by assuming that all inputs to the circuit initially arrive at time zero, with an initial input drive value (essentially the strength with which the input is being driven). Arrival times are then propagated to the first level of nodes in the network by multiplying the drive value by the input load of each input pin being driven to determine the initial arrival time at that pin, adding the pin's block delay (either rising or falling), then adding the pin's fanout delay multiplied by the sum of the input loads of all connected nodes. Note that this does not take into account any additional load introduced by wiring in the circuit. As mentioned previously, if the pin is inverting, the rising and falling edge times are interchanged before the block delays are added. This process of accumulating block delays and fanout delays is continued until the arrival times have propagated through the network to the output nodes.

Required times are calculated in a similar manner. The required times of all output nodes in the network are initially assumed to be time *t*. These times are then propagated backward through the network to the

input nodes. Given the required times (for both rising and falling edges) for a node, the required times of an input pin of a connected cell are calculated by subtracting the node's load value times the input pin's fanout delay, then subtracting the pin's block delay. This is just the reverse of the process used to calculate arrival times. Again, inverting input pins interchange the rising and falling edge times. The resulting times will all be less than *t*. Note that if an input pin fans out to multiple nodes, the worst required time of all connected nodes is used to calculate the pin's required time.

The slack time of a node is just defined as the difference between its required time and arrival time. It represents amount of excess time available in the longest timing path from any input to any output that passes through the node. A negative slack time indicates that a signal cannot propagate from the input and arrive at a node in time to meet the required time of at least one output that is fed from that node.

An example of this timing delay profiling is shown in Figure 26. In this case, the output required times are all just left at zero, so all the nodes' required times are negative, and the slack time for each node is just the maximal delay time for a path from any input to any output which passes through the node.

Timing Information for LZW Modules

This procedure was used to extract timing information from the **compress_2**, **decompress_2**, and **merged_2** logic views, using both nominal and worst case values. The resulting longest-path timing data is shown in Table 1.

Sample BDNET file to generate a network - just a two-level NOR-NOR.

```
MODEL           test:logic;
TECHNOLOGY      scmos;
VIEWTYPE        SYMBOLIC;
EDITSTYLE       SYMBOLIC;
OUTPUT          E;
INPUT           A, B, C, D;
SUPPLY          Vdd;
GROUND          GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/norf201":physical
        A1: A;
        B1: B;
        O:Temp1;
        "Vdd!":Vdd;
        "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/norf201":physical
        A1: C;
        B1: D;
        O:Temp2;
        "Vdd!":Vdd;
        "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/norf201":physical
        A1: Temp1;
        B1: Temp2;
        O:E;
        "Vdd!":Vdd;
        "GND!":GND;

ENDMODEL;
```

Cell parameters from library file (nominal values). Since **misII** optimizes for area, it will reduce the three NOR gates into a four-input AND-OR-INVERT followed by an inverter.

```
GATE "invf101:physical" 16 O=!A1;
PIN  A1 INV 0.00821 999 0.68 2.04 0.22 2.5

GATE "oaif2201:physical" 40 O=!((A1+B1)*(C2+D2));
PIN  A1 INV 0.08032 999 1.73 3.1 1.68 2.76
PIN  B1 INV 0.08032 999 1.73 3.1 1.68 2.76
PIN  C2 INV 0.08032 999 1.71 3.11 1.68 2.76
PIN  D2 INV 0.08032 999 1.71 3.11 1.68 2.76
```

MisII delay timing output for sample circuit. Note that the network is mapped to the cell library while **script.msu** is sourced.

```
UC Berkeley, MIS Release #2.2 (compiled 18-Dec-90 at 3:57 PM)
misII> read_oct test:logic
misII> source script.msu
WARNING: uses as primary input arrival time the value (0.00,0.00)
WARNING: uses as primary input drive the value (2.04,2.50)
WARNING: uses as primary output load the value 0.00
WARNING: uses as primary output required time the value (0.00,0.00)
misII> print_delay
 ... using library delay model
A         : arrival=( 0.16  0.20) required=(-2.38 -1.97) slack=(-2.55 -2.17)
B         : arrival=( 0.16  0.20) required=(-2.38 -1.97) slack=(-2.55 -2.17)
C         : arrival=( 0.16  0.20) required=(-2.38 -1.95) slack=(-2.55 -2.15)
D         : arrival=( 0.16  0.20) required=(-2.38 -1.95) slack=(-2.55 -2.15)
[361]     : arrival=( 1.95  1.87) required=(-0.22 -0.68) slack=(-2.17 -2.55)
{E}       : arrival=( 2.55  2.17) required=( 0.00  0.00) slack=(-2.55 -2.17)
```

Figure 26 - Sample of MisII Timing Calculations

		Nominal value	Worst case
compress_2	worst arrival time	31.06	81.80
	worst required time	-28.87	-77.34
	worst slack time	-31.06	-81.80
decompress_2	worst arrival time	37.86	93.63
	worst required time	-33.38	-84.28
	worst slack time	-37.86	-93.63
merged_2	worst arrival time	53.89	130.24
	worst required time	-49.84	-119.59
	worst slack time	-53.89	-130.24

Table 1 - Merged_2 Timing Data

Note that these values are only for the combinational logic, and do not include the setup and hold times of the flip-flops. Also, the timing of the DCAM, stack, and FIFOs is still unknown. If the library files for the cell library are correct, this data would indicate that even for nominal values, if the flip-flop timing is taken into account, the maximum clock frequency for **merged_2** would be between 10 and 15 MHz. If the worst case values are used, the maximum frequency would only be about 7 MHz.

Before these values are taken as absolutes, the library files should be verified. If they are correct, it appears that in order to achieve performance in the range of 10 to 20 MHz, it will be necessary to use a faster standard cell library. Note that this analysis has only focussed on the paths within the combinational logic. If one of the nodes with a critical arrival or required time is an interface signal to the DCAM, for example, the clock rate would be even more severely constrained. Initial examination of the critical node lists did not suggest that this is the case, but more in-depth analysis should be done after the delay data has been verified.

Compression Efficiency Measurement

The efficiency of the compressor has been measured using the utility which executes the algorithm and generates test scripts. This program is executing the identical algorithm, and produces output files whose size can be compared to those of the input files to measure the compression ratio.

The tests have been performed on a set of files which have been assembled to demonstrate the strengths and weaknesses of various text compression algorithms. This set of files, known as the Calgary compression corpus, was first presented in [BELL], Appendix B. It is available via anonymous FTP from the Internet address *fsa.cpsc.ucalgary.ca* and has become a standard benchmark for text compression programs.

The data in Table 2 represents the performance of the algorithm on each of the files in the corpus, where the compression ratio is measured in output bits per input byte (in order to maintain compatibility with results presented in [BELL]). The compression ratio of input bytes to output bytes can be obtained by dividing eight by this number. The table also shows the compression ratios for the **compress** program (available under version 4.2 of the **Ultrix** operating system) and for **PKZIP** (version 1.1 of the MS-DOS program) for comparison.

It can be seen that these programs both outperform the LZW algorithm in all cases. This is to be expected, since **compress** uses a modified LZW with a 16-bit string table, variable length output codes, and logic to reset the table if the compression ratio begins to decrease, and **PKZIP** uses an LZ77 algorithm with an 8 Kbyte sliding window, followed by a

Shannon-Fano coder (a statistical encoder similar to a Huffman encoder). However, the LZW algorithm's compression efficiency is not unreasonable, especially in light of the expected compression speeds the circuit will provide. It provides an compression ratio of 4.33 bits per byte, which is roughly a 1.85 ratio of input to output bytes. For several of the files, it performed substantially better than this; if the file **obj2** is ignored, the average compression ratio is 3.86 bits per byte, or an absolute ratio of 2.07. This is a reasonable average value.

		compress		**PKZIP**		**LZW**	
Source	Input Size	Output Size	Comp. Ratio	Output Size	Comp. Ratio	Output Size	Comp. Ratio
bib	111,261	46,528	3.35	41,354	2.97	53,849	3.87
book1	768,771	332,056	3.46	350,560	3.65	390,815	4.07
book2	610,856	250,759	3.28	232,589	3.05	346,535	4.54
geo	102,400	77,777	6.08	76,172	5.95	78,759	6.15
news	377,109	182,121	3.86	157,326	3.34	232,815	4.94
obj1	21,504	14,048	5.23	10,546	3.92	16,931	6.30
obj2	246,814	128,659	4.17	90,130	2.92	302,547	9.81
paper1	53,161	25,077	3.77	20,041	3.02	31,187	4.69
paper2	82,199	36,161	3.52	32,867	3.20	41,670	4.06
pic	513,216	62,215	0.97	63,805	1.00	70,617	1.10
progc	39,611	19,143	3.87	14,161	2.86	24,471	4.94
progl	71,646	27,148	3.03	17,255	1.93	34,920	3.90
progp	49,379	19,209	3.11	11,877	1.92	23,292	3.77
trans	93,695	38,240	3.27	23,135	1.98	50,549	4.32
Total	3,141,622	1,259,141	3.21	1,141,818	2.91	1,698,957	4.33

Table 2 - Comparison of Compression Ratios

CHAPTER 14

CONCLUSIONS

The design presented herein is fairly complete; all of the standard cell control logic required for the IC has been subdivided into modules, and each module has been designed, generated, and simulated (using a logic-level simulator). Initial performance estimates of the merged compression and decompression modules have been presented. The assumptions made regarding the custom modules required to complete the IC have been enumerated.

If the enhanced dynamic content-addressable memory (DCAM) capable of automatically writing the data pattern during each CAM search can be built, so that the **merged_2** module can be used, the total area required for the compressor/decompressor, the string reversal module, and the master controller is roughly 11.88 square millimeters. If the nominal timing values for the Mississippi State University standard cell library are correct, the nominal maximum clock rate for the controller logic will be in excess of 10 MHz.

If an IC capable of operation at a clock rate of 15 MHz can be created, it will substantially outperform existing data compression ICs. The current state of the art is about 2.5 Mbytes/sec average compression speed, with slightly higher decompression rate. However, since **merged_2** compresses at one byte per clock cycle and decompresses at nearly that rate, the IC would be six times as fast as any competitor's.

As noted, the average compression ratio for the LZW algorithm is not extremely high, but it is probably sufficient for most applications. It should operate with a compression ratio in excess of 2.0 for all but the most difficult data.

Several similarities between the standard cell design methodology used for this project and the normal software development cycle have been noted, and some software development tools have been adapted for use in the standard cell design.

Further Research

There is a fair amount of follow-up research related to this project. The design and implementation of the DCAM, the ring buffers used in the string reverser, and the I/O buffers must be completed, and system level integration and testing must be done. The timing data for the standard cell library must be verified. If it is correct, an alternative standard cell library might be required in order to achieve the desired performance.

In addition to just finishing up this design, a number of enhancements to the basic algorithm could be investigated to improve compression performance. For example, the capacity to handle variable length compression codes could be implemented in the input and output code FIFOs. Various string table management algorithms should be evaluated. Run length limiting could be added to the byte stream.

Finally, an investigation of other compression techniques could be conducted to determine their applicability to standard cell designs. LZW is certainly not the most efficient compression technique available, even

though it is ideally suited to hardware implementation. However, arithmetic coding seems to hold a great deal of promise as a highly efficient compression algorithm, if high-speed hardware could be developed to implement it.

REFERENCES CITED

[ABRAHAM] Abrahamson, David M. *An Adaptive Dependency Source Model for Data Compression*. Communications of the ACM, Vol. 32, No. 1, January, 1989, pp. 77-83.

[AHA] Advanced Hardware Architectures, Inc. AHA3101 Data Compression Coprocessor IC Product Specifications. Advanced Hardware Architectures, Moscow, ID, Publication #PS3101-0691, June, 1991.

[AMD] Advanced Micro Devices, Inc. Am99C10A 256 x 48 Content Addressable Memory Data Sheet. Advanced Micro Devices, Sunnyvale, CA, Publication #08125, February, 1990.

[BELL] Bell, Timothy C., John G. Cleary, and Ian H. Witten. Text Compression. Prentice Hall Advanced Reference Series, Englewood Cliffs, NJ, 1990.

[BUNTON] Bunton, Suzanne, and Gaetano Borriello. *Practical Dictionary Management for Hardware Data Compression*. Washington Research Foundation Technical Publication #02-90-09.

[HORSPOOL] Horspool, R. Nigel. *Improving LZW*. Proceedings of the Data Compression Conference 1991, James A. Storer and John H. Reif, editors, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 332-341.

[HUFFMAN] Huffman, D. A. *A Method for the Construction of Minimum-Redundancy Codes*. Proceedings of the Institute of Electrical and Radio Engineers, Vol. 40, No. 9, September, 1952, pp. 1098-1101.

[INFOCHIP] InfoChip Systems, Inc. InfoChip Systems IC-105 Data Compression Coprocessor Data Sheet (Preliminary). InfoChip Systems, Santa Clara, CA, Publication #18-000062, April, 1990.

[KNUTH] Knuth, Donald E. The Art of Computer Programming, Vol. 3: Searching and Sorting. Addison Wesley, Reading, MA, 1973.

[KWAN] Kwan, Reggie C. Universal Coding with Different Modelers in Data Compression. Master's Thesis, Department of Computer Science, Montana State University, Bozeman, MT, July, 1987.

[LANGDON82] Langdon, Glen G. Jr., and Jorma Rissanen. *A Simple General Binary Source Code*. IEEE Transactions on Information Theory, Vol. 28, No. 5, September, 1982, pp. 800-803.

[LANGDON84] Langdon, Glen G. Jr. *An Introduction to Arithmetic Coding*. IBM Journal of Research and Development, Vol. 28, No. 2, March, 1984, pp. 135-149.

[PERL] Perl, Yehoshua, Venkat Maram, and Nageshwar Kadakuntla. *The Cascading of the LZW Compression Algorithm with Arithmetic Coding.* Proceedings of the Data Compression Conference 1991, James A. Storer and John H. Reif, editors, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 277-286.

[RISSANEN] Rissanen, Jorma, and Glen G. Langdon Jr. *Universal Modeling and Coding.* IEEE Transactions on Information Theory, Vol. 27., No. 1, January, 1981, pp. 12-23.

[SHOJI] Shoji, Masakazu. CMOS Digital Circuit Technology. Prentice Hall, Englewood Cliffs, NJ, 1988.

[STAC] Stac Electronics. 9704 Data Compression Coprocessor Data Sheet. Stac Electronics, Carlsbad, CA, Publication #PRS-0006, February, 1990.

[STORER82] Storer, James A., and Tim G. Szymanski. *Data Compression via Textual Substitution.* Journal of the ACM, Vol. 29, No. 4, October, 1982, pp. 928-951.

[STORER88] Storer, James A. Data Compression: Methods and Theory. Computer Science Press, Rockville, MD, 1988.

[VAUGHAN] Vaughan-Nichols, Steven J. *Saving Space.* Byte, Vol. 15, No. 3, March, 1990, pp. 237-243.

[WALLACE] Wallace, Gregory K. *The JPEG Still Picture Compression Standard.* Communications of the ACM, Vol. 34, No. 4, April, 1991, pp. 30-45.

[WELCH] Welch, Terry A. *A Technique for High-Performance Data Compression.* IEEE Computer, Vol. 17, No. 6, June, 1984, pp. 8-19.

[WESTE] Weste, Neil, and Kamran Eshraghian. Principles of CMOS VLSI Design. Addison Wesley, Reading, MA, 1985.

[WILLIAMS] Williams, Ross N. *An Extremely Fast Ziv-Lempel Data Compression Algorithm.* Proceedings of the Data Compression Conference 1991, James A. Storer and John H. Reif, editors, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 362-371.

[WINTERS] Winters, Kel. Galois Field Based State Assignment for Programmable Logic Array Controllers. Original Manuscript.

[WITTEN] Witten, Ian H., Radford M. Neal, and John G. Cleary. *Arithmetic Coding for Data Compression.* Communications of the ACM, Vol. 30, No. 6, June, 1987, pp. 520-540.

[ZIV77] Ziv, Jacob, and Abraham Lempel. *A Universal Algorithm for Sequential Data Compression.* IEEE Transactions on Information Theory, Vol. 23, No. 3, May, 1977, pp. 337-343.

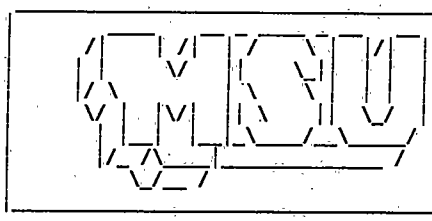
[ZIV78] Ziv, Jacob, and Abraham Lempel. *Compression of Individual Sequences via Variable Rate Coding.* IEEE Transactions on Information Theory, Vol. 24, No. 5, September, 1978, pp. 530-536.

APPENDICES

Appendix A

Complete Set of Documentation for

Merged_2 Module



Montana State University
Dept. of Electrical Engineering
Bozeman, Montana 59717

LZW Data Compression Project

External Reference Specification

GENERAL:

Cell Name: merged_2
Short Description:
Combined compress_2 and decompress_2 modules (LZW compressor and decompressor).
Pathname: ~icsg8038/thesis/merged/merged_2
Revision History:

[X] check if LEAFCELL. If not,
Cells Instanced:

Circuit Design by: Bob Wall
Layout by: Standard cell generation in OCT
Email contact: icsg8038@caesar.cs.montana.edu Bob Wall

Process: MOSIS SCN (N-well scalable CMOS)
LAMBDA value(s) in microns:
Height (in lambdas):
Width (in lambdas):

OCT Revision: 3.5
Quickic Revision: 6
Tekspice Revision: 1E

FUNCTIONAL DESCRIPTION:
For details on LZW compressor and decompress, see compress_2.ers and decompress_2.ers.

BDNET INSTANCE FORM:

LOGIC EQUATIONS:

TRUTH TABLE:

INPUT SPECIFICATIONS for LAMBDA=1um:

Terminal:	Load Cap:	Tsetup(min):	Thold(min):

OUTPUT SPECIFICATIONS for LAMBDA=1um:

Terminal:	Tdelay(min):	Tdelay(max):	Load Cap:

CLOCK RATE for LAMBDA=1um:

Fclock(max):
Fclock(min):

Figure 27 - Sample ERS Form (cont. on next page)

POWER REQUIREMENTS for LAMBDA=1um:

AC Power:	Fclock:

USAGE and TILING NOTES:

OCT VIEWS:

Viewname:	Description:
physical	physical artwork
symbolic	symbolic artwork
schematic	simulation schematic (musa)
schemdoc	documentation schematic
logic	output of misII - logic without flip-flops
unplaced	output of bdnet - logic with flip-flops added
flat	output of octflatten - 'unplaced' without hierarchy
placed	output of wolfe - 'flat', placed and routed

OTHER FILES:

Filename:	Description:
bdnet	Oct bdnet netlist
bds	Oct bds behavioral description
cif	CIF artwork
ers	External Reference Specification document
musa	Oct musa simulation script
rtl	Register Transfer Level description
sim	Mextra extraction netlist (sim format)
spice	Spice simulation file
tspice	Tekspice simulation file

VERIFICATION:

	Engineer:	Tool:	Date:
Logic Simulation:	Bob Wall	musa	7/01/91
Circuit Simulation:			
DRC:			
Extract & Simulate:			
Connectivity Check:			

Figure 27 (cont.) - Sample ERS Form

```
!=============================================================================
!
!   File:    merged_2.rtl    (/user1/icsg8038/thesis/merged_2.rtl)
!   Author:  Bob Wall
!   Date:    6/30/91
!
!   Description:
!       Simplified RTL description of the LZW compressor/decompressor opera-
!       tion, with the single clock cycle compressor and optimized decompres-
!       sor.  This is essentially a boiled-down version of merged_2.bds, and
!       will hopefully be a little easier to read.
!
!   Notes:
!       The '.' between two expressions represents concatenation.
!       '<-' represents a clocked (synchronous) assignment, while '<=' is
!       a combinational (asynchronous) assignment.
!
!       The compressor portion is essentially just compress_2.rtl, and the
!       decompressor is just decompress_2.rtl.  They are stuck together with
!       an "If DoCompress ... Else ...".
!
!       The logic to perform refreshes of the DCAM has been removed.  The
!       controller will now periodically halt the compressor or decompressor
!       and refresh a row of the DCAM.
!
!       Now have string length limiting in place, as well as halt signals from
!       the controller.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/merged/RCS/merged_2.rtl,v 1.3 91/07/15
20:17:15 icsg8038 Exp $
!       $Log:   merged_2.rtl,v $
!       Revision 1.3  91/07/15  20:17:15  icsg8038
!       Added DumpOmega capability.
!
!       Revision 1.2  91/07/14  14:33:32  icsg8038
!       Added string length limiting, halt signal processing.
!
!       Revision 1.1  91/07/08  15:47:40  icsg8038
!       Initial revision
!
!=============================================================================

!-----------------------------------------------------------------------------
!   Variable declarations

INPUT       Reset,
            Halt,
            DumpOmega,
            CAMMatch,
            InCharFIFO  < 7:0>,
            InCodeFIFO  <11:0>,
            CAMInData   <19:0>,
            CAMInAddr   <11:0>;

OUTPUT      ReadCharFIFO,
            ReadCodeFIFO,
            WriteCodeFIFO,
            OutCodeFIFO <11:0>,
            WriteStack,
            FlipStack,
            StackData   < 7:0>,
            SearchCAM,
            ReadCAM,                    !   Latched, output fed back
            WriteCAM,                   !   Latched
            CAMOutData  <19:0>,
            CAMOutAddr  <11:0>;         !   Latched, output fed back
```

Figure 28 - Merged_2.RTL (cont. on next page)

```
REGISTER    State,
            FinChar      < 7:0>,
            Omega        <11:0>,
            InCode       <11:0>,
            OldCode      <11:0>,
            NextCode     <11:0>,
            D_CAMOutData<19:0>,        !   Latched version for decompressor
            StrLen       < 7:0>,
            OmitNextStr;


CONSTANT    TRUE            1,
            FALSE           0,
            INIT_STR_LEN    0xff,
            LAST_STR_LEN    0x23;      !   Gives 128-character max. length


!   The following is a simple Galois finite-field sequencer to generate 8-bit
!   sequences.  The minimum-weight polynomial is x^8 + x^4 + x^3 + x^2 + 1.
MACRO       NextStrLen(StrLen)
                TempLen<0>  <=  StrLen<7>;
                TempLen<1>  <=  StrLen<0>;
                TempLen<2>  <=  StrLen<1> XOR StrLen <7>;
                TempLen<3>  <=  StrLen<2> XOR StrLen <7>;
                TempLen<4>  <=  StrLen<3> XOR StrLen <7>;
                TempLen<5>  <=  StrLen<4>;
                TempLen<6>  <=  StrLen<5>;
                TempLen<7>  <=  StrLen<6>;
                Return (TempLen)


!   Note that in the compressor, instead of using NextCode (as would normally
!   be done), the CAMOutAddr register is used.  This is because during com-
!   pression, it is identical to the NextCode register, so only one of them
!   need be used.  Use CAMOutAddr, since its output is connected to the address
!   buss out to the DCAM (hence the name!).

!-------------------------------------------------------------------------------
!   Begin Merged
!
!   First comes the compression section.
!

If DoCompress

    !   Turn off the outputs used only by the decompressor.

    ReadCAM, WriteCAM <-  FALSE
    WriteStack, FlipStack, ReadCodeFIFO  <=  FALSE

    !   Select the unlatched version of CAMOutData to place on the buss
    !   to the DCAM.

    CAMOutData  <=  C_CAMOutData

    If Reset                            !   Initialization stage

        SearchCAM     <=  FALSE
        WriteCodeFIFO <=  FALSE

        ReadCharFIFO  <=  TRUE          !   Read 1st byte to initialize Omega
        Omega         <-  0 . InCharFIFO
        CAMOutAddr    <-  0x100         !   First emtpy table entry
        StrLen        <-  INIT_STR_LEN

    Else if Halt
        SearchCAM, ReadCharFIFO  <=  FALSE
```

Figure 28 (cont.) - Merged_2.RTL (cont. on next page)

```
        If (DumpOmega)
            WriteCodeFIFO   <=  TRUE
            OutCodeFIFO     <=  Omega
        Else
            WriteCodeFIFO   <=  FALSE

        Omega           <-  Omega
        CAMOutAddr      <-  CAMOutAddr
        StrLen          <-  StrLen

    Else
        !   Precalculate the value of CAMOutAddr + 1, so it will be ready
        !   by the time CAMMatch is back from the DCAM.  If the table is
        !   full already, just set it to CAMOutAddr.
        !   Also precalculate new string length.

        If (CAMOutAddr != 0xfff)        !   Update next table address
            TempCode  <=  CAMOutAddr + 1
        Else                            !   Table full
            TempCode  <=  CAMOutAddr

        TempStrLen    <=  NextStrLen (StrLen)
        ReadCharFIFO  <=  TRUE          !   Get next input byte ready

        If (StrLen == LAST_STR_LEN)     !   Accumulated maximum length string

            SearchCAM     <=  FALSE     !   Dump accumulated code and restart
            WriteCodeFIFO <=  TRUE
            OutCodeFIFO   <=  Omega
            Omega         <-  0 . InCharFIFO
            CAMOutAddr    <-  CAMOutAddr
            StrLen        <-  INIT_STR_LEN

        Else

            SearchCAM     <=  TRUE      !   Enable a CAM search
            C_CAMOutData  <=  Omega.InCharFIFO  !   Omega . K to search for

            !   Note that every time a search is done, the search pattern
            !   (Omega . K) is written to address CAMOutAddr.  Thus, if a match
            !   fails, the new string has already been added to the table, and
            !   all that is necessary is to update CAMOutAddr to point to the
            !   next empty table entry.

            If (CAMMatch)               !   Omega . K was in the table
                Omega         <-  CAMInAddr !   Omega  <-  addr (Omega . K)
                WriteCodeFIFO <=  FALSE     !   Don't generate output
                CAMOutAddr    <-  CAMOutAddr
                StrLen        <-  TempStrLen

            Else                        !   It wasn't in the table
                Omega         <-  0 . InCharFIFO    !   Omega <- 1-char string K
                WriteCodeFIFO <=  TRUE      !   Enable output

                OutCodeFIFO   <=  Omega     !   Output last Omega
                CAMOutAddr    <-  TempCode  !   Update next table entry address
                StrLen        <-  INIT_STR_LEN

!-------------------------------------------------------------------------
!   Decompression section
!

Else

    !   Turn off the outputs used only by the compressor.

    SearchCAM, ReadCharFIFO, WriteCodeFIFO  <=  FALSE
```

Figure 28 (cont.) - Merged_2.RTL (cont. on next page)

```
!   Select the latched version of CAMOutData to place on the buss
!   to the DCAM.

CAMOutData  <=  D_CAMOutData

If Reset                              !  Initialization stage
    !   This step takes advantage of the fact that the first input code is
    !   guaranteed to represent a single character string, and that char-
    !   acter will just be the last eight bits of the code.

    NextCode    <-  0x100
    OldCode     <-  InCodeFIFO
    FinChar     <-  InCodeFIFO <7:0>
    StackData   <=  InCodeFIFO <7:0>    !   Output character portion of it

    WriteStack, FlipStack, ReadCodeFIFO  <=  TRUE
    ReadCAM, WriteCAM  <-  FALSE

    StrLen      <-  INIT_STR_LEN
    OmitNextStr <-  FALSE
    State       <-  0

Else if Halt

    WriteStack, FlipStack, ReadCharFIFO  <=  FALSE
    WriteCAM    <-  WriteCAM
    ReadCAM     <-  ReadCAM
    D_CAMOutData  <-  D_CAMOutData
    NextCode    <-  NextCode
    OldCode     <-  OldCode
    InCode      <-  InCode
    FinChar     <-  FinChar
    OmitNextStr <-  OmitNextStr
    StrLen      <-  StrLen
    State       <-  State

Else if (State == 0)

    If (InCodeFIFO == NextCode)     !   Special case - InCode not in table
        WriteStack  <=  TRUE
        StackData   <=  FinChar
        CAMOutAddr  <-  OldCode
        StrLen      <-  NextStrLen (StrLen)

    Else                            !   Normal mode - InCode in table
        WriteStack  <=  FALSE
        CAMOutAddr  <-  InCodeFIFO
        StrLen      <-  StrLen

    !   If the top four bits of the address to be read are 0, then it is a
    !   single character string, and there is no need to read the CAM after
    !   all (the character is just the last eight bits of the address).

    If (CAMOutAddr_D <11:8> == 0)
        ReadCAM     <-  FALSE
    Else
        ReadCAM     <-  TRUE

    InCode      <-  InCodeFIFO      !   Latch input code for later use
    ReadCodeFIFO  <=  TRUE          !   and get the next one ready
    FlipStack   <=  FALSE
    WriteCAM    <-  FALSE

    NextCode    <-  NextCode        !   Maintain register contents
    OldCode     <-  OldCode
    OmitNextStr <-  OmitNextStr
    State       <-  1
```

Figure 28 (cont.) - Merged_2.RTL (cont. on next page)

```
Else if (State == 1)                    !  Main decompression loop

    WriteStack  <=  TRUE                !  Push next char from string

    If (ReadCAM)                        !  Haven't reached start of string
        StackData   <=  CAMInData <7:0>

        CAMOutAddr  <-  CAMInData <19:8>

        If (CAMOutAddr_Q <11:8> == 0)   !  Same as in State 0
            ReadCAM     <-  FALSE       !  Reached end of string
        Else
            ReadCAM     <-  TRUE        !  Continue tracing string back

        FlipStack   <=  FALSE
        WriteCAM    <-  FALSE

        OldCode     <-  OldCode         !  Maintain register contents
        NextCode    <-  NextCode
        OmitNextStr <-  OmitNextStr

        StrLen      <-  NextStrLen (StrLen)

        State       <-  1               !  Continue in loop

    Else                                !  Reached head of string
        StackData   <=  CAMOutAddr <7:0>
        ReadCAM     <-  FALSE
        FlipStack   <=  TRUE            !  Can start outputting string now

        FinChar     <-  CAMOutAddr <7:0>    !  Record first char of string
        OldCode     <-  InCode

        If (NextCode != 0xfff && !OmitNextStr)
            !  String table not full yet, and it's OK to add new string
            WriteCAM    <-  TRUE        !  Place new entry in table
            D_CAMOutData <-  OldCode . CAMOutAddr <7:0>
            CAMOutAddr  <-  NextCode
            NextCode    <-  NextCode + 1    !  Move on to next empty entry

        Else                            !  No room for new entry
            WriteCAM    <-  FALSE
            NextCode    <-  NextCode

        If (StrLen == LAST_STR_LEN)
            OmitNextStr <-  TRUE
        Else
            OmitNextStr <-  FALSE

        StrLen      <-  INIT_STR_LEN

        State       <-  0               !  Done with loop - go to next input

    ReadCodeFIFO  <=  FALSE
    InCode      <-  InCode              !  Maintain register contents
```

Figure 28 (cont.) - Merged_2.RTL

```
!=============================================================================
!
!   File:    merged_2.bds    (/user1/icsg8038/thesis/merged_2.bds)
!   Author:  Bob Wall
!   Date:    7/01/91
!
!   Description:
!       BDS description of merged LZW compressor/decompressor core (no FIFOs
!       or CAM).  Uses the single clock cycle compress (compress_2) and the
!       improved decompressor (decompress_2).
!
!   Notes:
!       This is a simplified version - it assumes that the input FIFO will
!       always have data available, and that the output FIFO will never be
!       full.
!
!       This algorithm assumes that the CAM is pre-initialized with all the
!       single-character strings - i.e. the first 256 entries are ROM
!       instead of CAM.
!
!       Since BDS does not support memory (i.e. flip-flops), all the regi-
!       sters and flags are set up as separate output and input signals,
!       which can be connected to D flip-flops using BDNET.
!
!       Note that some registers are "refreshed" in certain states.  A
!       register must be assigned some value for every state during every
!       clock cycle, unless it is acceptable for it to be filled with
!       possible garbage from the input logic.  If it does not get assigned
!       a new value, it should be reassigned its output to maintain that
!       value.
!
!       The last entry in the CAM (with address 0xfff) is not used, since
!       this is used as the NULL pointer for the decompressor.  Thus, if
!       NextCode reaches 0xfff, the table is full.
!
!       The logic to perform refreshes of the DCAM has been removed.  The
!       controller will now periodically halt the compressor or decompressor
!       and refresh a row of the DCAM.
!
!   NOTE:
!       This file was created by merging compress_2.bds and decompress_2.bds,
!       with few modifications (other than the signal name substitutions
!       noted in the descriptions of the Compress and Decompress routines).
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/merged/RCS/merged_2.bds,v 1.3 91/07/15
20:17:12 icsg8038 Exp $
!       $Log:   merged_2.bds,v $
!       Revision 1.3  91/07/15  20:17:12  icsg8038
!       Added DumpOmega capability.
!
!       Revision 1.2  91/07/14  14:32:46  icsg8038
!       Added string length limiting, halt signal processing.
!
!       Revision 1.1  91/07/08  15:47:22  icsg8038
!       Initial revision
!
!=============================================================================

MODEL Merged

!   Circuit outputs and connections to all register and flag D inputs.
    FinChar_D    < 7:0>,        !   Last char of last output string
    Omega_D      <11:0>,        !   Ptr. to accumulated string / input code
    OldCode_D    <11:0>,        !   Previous input code
    NextCode_D   <11:0>,        !   Next unused comp. code / next table entry
    StrLen_D     < 7:0>,        !   String length counter
```

Figure 29 - Merged_2.BDS (cont. on next page)

```
    State_D,                    !   State variable
    OmitNextStr_D,              !   Flag to prevent decompressor string add
    WriteCAM_D,                 !   Flag, output to signal CAM data write
    ReadCAM_D,                  !   Flag, output to signal CAM data read
    SearchCAM,                  !   Output to signal CAM data search
    ReadCharFIFO,               !   Output to signal input char FIFO read
    ReadCodeFIFO,               !   Output to signal input code FIFO read
    WriteCodeFIFO,              !   Output to signal output FIFO write
    FlipStack,                  !   Output to signal stack str. reversal
    WriteStack,                 !   Output to signal stack data write
    StackData   < 7:0>,         !   Output to stack char. input
    OutCodeFIFO <11:0>,         !   Output to output FIFO input
    CAMOutAddr_D<11:0>,         !   Output to CAM address bus (latched)
    CAMOutData  <19:0>,         !   Output to CAM data bus
    C_CAMOutData<19:0>,
    D_CAMOutData_D <19:0>       !   Latched value for decompressor
=
!   Circuit inputs and connections to all register and flag Q outputs.
    FinChar_Q   < 7:0>,
    Omega_Q     <11:0>,
    OldCode_Q   <11:0>,
    NextCode_Q  <11:0>,
    StrLen_Q    < 7:0>,
    State_Q,
    OmitNextStr_Q,
    ReadCAM_Q,                  !   Output that is fed back and reused
    CAMOutAddr_Q<11:0>,         !       "
    D_CAMOutData_Q <19:0>,
    Reset,                      !   Input to initialize and start operation
    HaltIn,                     !   Input to temporarily suspend operation
    DumpOmega,                  !   Input to output last code accumulated
    DoCompress,                 !   Input to signal compression / decompression
    CAMMatch,                   !   Input from CAM search match result
    InCharFIFO  < 7:0>,         !   Input from input char FIFO output
    InCodeFIFO  <11:0>,         !   Input from input code FIFO output
    CAMInAddr   <11:0>,         !   Input from CAM address bus
    CAMInData   <19:0>,         !   Input from CAM data bus
    Comparator  <11:0>;         !   NextCode_Q XOR InCodeFIFO

CONSTANT
    TRUE = 1, FALSE = 0;

!=============================================================================
!
!   Routine to generate the next string length using a Galois-field based
!   sequencer.
!
!   Invocation: StrLen = NextStrLen (StrLen);
!   Input args: StrLen      RepLen-bit current refresh address
!
!   Notes:
!       The logic for the finite-field sequencer is very dependent on the
!       number of bits in the field (an irreducible polynomial of the correct
!       length is needed), so if the maximum string length is changed, this
!       routine MUST be modified.
!
!       For an eight-bit sequencer, a suitable minimum-weight polynomial is
!       0x11d, which corresponds to X^8 + X^4 + X^3 + X^2 + 1.
!
!       Since 0 is a meta-stable state of the sequencer, the StrLen register
!       should never be allowed to by all 0's.  It would be simple enough to
!       add a check in here - if CurrentStrLen = H000 THEN Return 0x3ff.
!
!       For more information on the subject of Galois-field based sequencers,
!       with a table of minimum-weight irreducible polynomials, see the paper
!       "Galois-Field Based State Assignment for PLA Controllers" by K. Winters.
!
!=============================================================================
```

Figure 29 (cont.) - Merged_2.BDS (cont. on next page)

```
CONSTANT
    INIT_STR_LEN = 255,     !   HFF
    LAST_STR_LEN = 35;      !   H23 - limit strings to 128 characters in length

ROUTINE NextStrLen<7:0> (CurrentStrLen<7:0>);

    STATE   Temp<7:0>,
            I<>;

    Temp<0> = CurrentStrLen<7>;
    Temp<1> = CurrentStrLen<0>;
    Temp<2> = CurrentStrLen<1> XOR CurrentStrLen<7>;
    Temp<3> = CurrentStrLen<2> XOR CurrentStrLen<7>;
    Temp<4> = CurrentStrLen<3> XOR CurrentStrLen<7>;

    for I FROM 5 TO 7 DO
        Temp<I> = CurrentStrLen<I - 1>;

    return Temp;

ENDROUTINE NextStrLen;

!=============================================================================
!
!   Compress routine - contains the body of compress_2.bds, plus logic to
!   allow halt and to limit maximum string length.
!
!   Note that CAMOutAddr is used as the NextCode register.
!
!=============================================================================

ROUTINE  Compress;

    STATE   TempCode<11:0>,             !   Computes CAMOutAddr + 1
            TempStrLen<7:0>;            !   Computes NextStrLen (StrLen)

    !   Turn off the outputs used only by the decompressor.

    ReadCAM_D     = FALSE;
    WriteCAM_D    = FALSE;
    WriteStack    = FALSE;
    FlipStack     = FALSE;
    ReadCodeFIFO  = FALSE;


    !   If a reset occurs, reset all the variables and prepare for compression
    If Reset EQL TRUE then
        Begin
        CAMOutAddr_D  = 100#16;
        SearchCAM     = FALSE;
        WriteCodeFIFO = FALSE;
        ReadCharFIFO  = TRUE;           !   Read first word to initialize Omega
        Omega_D       = 0#16 & InCharFIFO;
        StrLen_D      = INIT_STR_LEN;
        End

    !   If a halt occurs, just turn off all the outputs and maintain the values
    !   in all the registers, so execution can be picked up where it left off.
    !   If this is the end of the input stream, the controller will assert
    !   DumpOmega for one clock cycle during the Halt, so slap whatever's left
    !   in Omega into the output code FIFO.
    Else if HaltIn EQL TRUE then
        Begin
        SearchCAM     = FALSE;
        ReadCharFIFO  = FALSE;
```

Figure 29 (cont.) - Merged_2.BDS (cont. on next page)

```
    If DumpOmega EQL TRUE then
        Begin
        WriteCodeFIFO   = TRUE;
        OutCodeFIFO     = Omega_Q;
        End
    Else
        WriteCodeFIFO = FALSE;

    Omega_D       = Omega_Q;
    CAMOutAddr_D  = CAMOutAddr_Q;
    StrLen_D      = StrLen_Q;
    End

!   Otherwise, proceed with compression loop.  It is just one clock cycle
!   now.
Else
    Begin
    !   Precalculate the value of CAMOutAddr + 1, so it will be ready
    !   by the time CAMMatch is back from the DCAM.  Also precalculate
    !   the next string length.

    If CAMOutAddr_Q NEQ fff#16 then !   Room in the table .
        TempCode  = CAMOutAddr_Q + 1
    Else                            !   All full, sorry
        TempCode  = CAMOutAddr_Q;   !   Retain current CAMOutAddr value

    TempStrLen    = NextStrLen(StrLen_Q);
    ReadCharFIFO  = TRUE;           !   Read the next char from the input

    If StrLen_Q EQL LAST_STR_LEN then   !   Accumulated max length string
        Begin                           !   so dump Omega and restart
        SearchCAM     = FALSE;
        WriteCodeFIFO = TRUE;
        OutCodeFIFO   = Omega_Q;
        Omega_D       = 0#16 & InCharFIFO;
        CAMOutAddr_D  = CAMOutAddr_Q;
        StrLen_D      = INIT_STR_LEN;
        End

    Else
        Begin
        SearchCAM     = TRUE;       !   Initiate search for input string
        C_CAMOutData  = Omega_Q & InCharFIFO;

        !   Now, hang out and "wait" for the results of the search to come
        !   back.  They should be done in sufficient time to finish proces-
        !   sing in this clock cycle.
        !   NOTE:  Once the CAM decides whether the search matched or not,
        !   it must hold the CAMMatch signal at that level through the end
        !   of the clock cycle.

        If CAMMatch EQL TRUE then   !   Need to update Omega to new string
            Begin
            Omega_D   = CAMInAddr;  !   Latch match address as new string
            WriteCodeFIFO = FALSE;  !   Not generating an output this time
            CAMOutAddr_D  = CAMOutAddr_Q;   !   Retain value in register
            StrLen_D  = TempStrLen; !   Added a new char to string
            End

        Else                        !   Output code, reset Omega to K
            Begin
            !   Produce the next output code (just the value of Omega).
            OutCodeFIFO   = Omega_Q;
            WriteCodeFIFO = TRUE;

            !   Reset Omega to the single-character string K.
            Omega_D   = 0#16 & InCharFIFO;
```

Figure 29 (cont.) - Merged_2 BDS (cont. on next page)

```
                If DumpOmega EQL TRUE then
                !   The new string (Omega . K) has already been added to the
                !   table.  Just need to update the place where the next string
                !   will be written, if the table has not filled up already.
                !   Note that the last entry is unused.

                    CAMOutAddr_D  = TempCode;

                StrLen_D      = INIT_STR_LEN;   !   Restart length count

                End;    !   Else (!CAMMatch)
            End;    !   Else (StrLen_Q != LAST_STR_LEN)
        End;    !   Else (!Reset && !HaltIn)

ENDROUTINE Compress;

!=============================================================================
!
!   Decompress routine - contains the body of decompress_2.bds, plus halt
!   logic and string length limiting.
!
!   The following substitution was made from the original decompressor:
!       InCode -> Omega
!
!=============================================================================

ROUTINE  Decompress;

    !   Turn off the outputs used only by the decompressor.

    SearchCAM     = FALSE;
    ReadCharFIFO  = FALSE;
    WriteCodeFIFO = FALSE;

    !   If a reset occurs, reset all the variables and get the first input
    !   code to start decompression.
    !
    If Reset EQL TRUE then
        Begin
        NextCode_D   = 100#16;      !   Register initialization stuff

        !   Read the first code; since it is guaranteed to represent a single
        !   character string, the character is just the last eight bits.
        !   Write that string to the stack.

        OldCode_D    = InCodeFIFO;
        StackData    = InCodeFIFO<7:0>;
        FinChar_D    = InCodeFIFO<7:0>;

        WriteStack   = TRUE;
        FlipStack    = TRUE;
        ReadCodeFIFO = TRUE;
        WriteCAM_D   = FALSE;
        ReadCAM_D    = FALSE;

        StrLen_D     = INIT_STR_LEN;
        OmitNextStr_D = FALSE;

        State_D      = 0;
        End     !   If Reset

    !   If a halt occurs, just turn off all the outputs (except ReadCAM,
    !   WriteCAM) and maintain the values in all the registers, so execution
    !   can be picked up where it left off.  ReadCAM and WriteCAM must be
    !   maintained because the Halt might be caused by a Refresh, so the
    !   operation will have to be completed next clock cycle.  ReadCAM is used
    !   as a register as well.
```

Figure 29 (cont.) - Merged_2.BDS (cont. on next page)

```
Else if HaltIn EQL TRUE then
    Begin
    WriteStack   = FALSE;
    FlipStack    = FALSE;
    ReadCharFIFO = FALSE;

    ReadCAM_D    = ReadCAM_Q;
    WriteCAM_D   = WriteCAM_D;
    D_CAMOutData_D = D_CAMOutData_Q;

    NextCode_D   = NextCode_Q;
    OldCode_D    = OldCode_Q;
    Omega_D      = Omega_Q;
    FinChar_D    = FinChar_Q;
    CAMOutAddr_D = CAMOutAddr_Q;
    OmitNextStr_D = OmitNextStr_Q;
    StrLen_D     = StrLen_Q;
    State_D      = State_Q;
    End

!   Proceed with decompression loop.  The new input code should be ready
!   in InCodeFIFO.  Read it and latch into Omega, then determine whether
!   the code is in the table and prepare to create decompressed string.
!
Else if State_Q EQL 0 then
    Begin
    If Comparator EQL 000#16 then   !   InCodeFIFO == NextCode_Q
        Begin                       !   Special case - Omega not in table
        WriteStack     = TRUE;      !   Push FinChar
        StackData      = FinChar_Q;
        CAMOutAddr_D   = OldCode_Q; !   And assume code is the prev. one
        StrLen_D       = NextStrLen(StrLen_Q);
        End

    Else
        Begin
        WriteStack     = FALSE;     !   Don't need to push anything yet
        CAMOutAddr_D   = InCodeFIFO;!   Read table entry for new code
        StrLen_D       = StrLen_Q;
        End;

    If CAMOutAddr_D<11:8> EQL 0 then    !   Just a single character string
        ReadCAM_D      = FALSE      !   Don't need to read it from DCAM
    Else
        ReadCAM_D      = TRUE;      !   Get the appropriate table entry.

    Omega_D      = InCodeFIFO;
    ReadCodeFIFO = TRUE;

    FlipStack    = FALSE;
    WriteCAM_D   = FALSE;           !   Turn off unneeded flags

    NextCode_D   = NextCode_Q;      !   Maintain register contents
    OldCode_D    = OldCode_Q;
    OmitNextStr_D = OmitNextStr_Q;

    State_D      = 1;               !   Get into main decompress loop
    End     !   Else if State == 0

!   The meat of the decompressor - assumes that the CAM is returning the
!   string table entry for a given code.  A code is decompressed by recur-
!   sively pushing the character portion of the table entry on the stack,
!   and reading the entry for the code portion of the entry (the link to
!   the remainder of the string).  The recursion terminates when a code
!   less than 0x100 is reached.  The last character is just the lower
!   eight bits of this code.  It is pushed, the stack is signalled to
!   begin popping the string, and a new code is added to the string table.
```

Figure 29 (cont.) - Merged_2.BDS (cont. on next page)

```
    Else if State_Q EQL 1 then
        Begin
        WriteStack   = TRUE;            !   Push character from table entry

        If ReadCAM_Q EQL 1 then
            Begin                       !   Haven't reached start of string yet
            StackData    = CAMInData< 7:0>;
            CAMOutAddr_D = CAMInData<19:8>;

            If CAMOutAddr_D<11:8> EQL 0 then    !   Just a one character string
                ReadCAM_D      = FALSE  !   Don't need to read it from DCAM
            Else
                ReadCAM_D    = TRUE;    !   Get the appropriate table entry.

            FlipStack    = FALSE;       !   Don't reverse string yet
            WriteCAM_D   = FALSE;

            OldCode_D    = OldCode_Q;   !   Maintain register contents
            NextCode_D   = NextCode_Q;
            OmitNextStr_D = OmitNextStr_Q;

            StrLen_D     = NextStrLen(StrLen_Q);

            State_D      = 1;           !   Remain in this state
            End     !   If ReadCAM

        Else                            !   Found the head of the string
            Begin
            StackData    = CAMOutAddr_Q< 7:0>;
            ReadCAM_D    = FALSE;       !   Stop tracing links

            FlipStack    = TRUE;        !   Can start outputting string now

            FinChar_D    = CAMOutAddr_Q< 7:0>;  !   Record last char in string
            OldCode_D    = Omega_Q;     !   Remember last code

            If NextCode_Q NEQ fff#16 AND OmitNextStr_Q NEQ TRUE then
                Begin                   !   Table not full yet, and add is OK
                WriteCAM_D   = TRUE;    !   Add new entry
                D_CAMOutData_D = OldCode_Q & CAMOutAddr_Q< 7:0>;
                CAMOutAddr_D = NextCode_Q;
                NextCode_D   = NextCode_Q + 1;
                End
            Else
                Begin
                WriteCAM_D   = FALSE;   !   No room for new entry
                NextCode_D   = NextCode_Q;  !   Maintain register contents
                End;

            If StrLen_Q EQL LAST_STR_LEN then
                OmitNextStr_D = TRUE
            Else
                OmitNextStr_D = FALSE;

            StrLen_D     = INIT_STR_LEN;

            State_D      = 0;           !   Done with loop - clear outta here
            End;    !   Else

        ReadCodeFIFO = FALSE;           !   Turn off unneeded flags
        Omega_D      = Omega_Q;     !   Maintain register contents
        End;        !   Else if State == 1

ENDROUTINE Decompress;
```

Figure 29 (cont.) - Merged_2.BDS (cont. on next page)

```
!=============================================================================
!
!   Main routine - contains all the logic for sequencing through both the
!   compression and decompression algorithms.
!
!   Calls:  Compress (), Decompress ()
!
!=============================================================================

ROUTINE Merged;

    If DoCompress then
        Compress ()
    Else
        Decompress ();

    If DoCompress then
        CAMOutData = C_CAMOutData
    Else
        CAMOutData = D_CAMOutData_Q;

ENDROUTINE Merged;

ENDMODEL Merged;
```

Figure 29 (cont.) - Merged_2.BDS

```
!============================================================================
!
!   File:    merged_2.bdnet    (/user1/icsg8038/thesis/merged_2.bdnet)
!   Author:  Bob Wall
!   Date:    7/01/91
!
!   Description:
!       BDNET description of merged LZW compressor/decompressor core (no
!       FIFOs or CAM) as described in merged_2.bds.  This file will place
!       the flip-flops needed to hold register and flag values in the
!       (de)compressor, and will connect up SUPPLY and GROUND nodes.
!
!   Notes:
!       The dfnf311 instanced is a negative edge-triggered D flip-flop with
!       Q and Q_bar outputs.
!
!       The number of bits in the StrLen register must be changed if the
!       maximum string length is modified in merged_2.bds.
!
!   Revision History:
!      $Header: /n/dali/u1/icsg8038/thesis/merged/RCS/merged_2.bdnet,v 1.3 91/07/15 20:17:01
icsg8038 Exp $
!       $Log:   merged_2.bdnet,v $
!       Revision 1.3  91/07/15  20:17:01  icsg8038
!       Added DumpOmega capability.
!
!       Revision 1.2  91/07/14  14:32:25  icsg8038
!       Added string length limiting, halt signal processing.
!
!       Revision 1.1  91/07/08  15:46:59  icsg8038
!       Initial revision
!
!============================================================================

MODEL       merged_2:unplaced;
TECHNOLOGY  scmos;
VIEWTYPE    SYMBOLIC;
EDITSTYLE   SYMBOLIC;

OUTPUT
    OutCodeFIFO<11:0> : OutCodeFIFO<11:0>,  !   Output to output FIFO input
    CAMOutAddr<11:0>  : CAMOutAddr_Q<11:0>, !   Output to CAM address bus
    CAMOutData<19:0>  : CAMOutData<19:0>,   !   Output to CAM data bus
    StackData<7:0>,                         !   Output to character stack
    WriteCAM          : WriteCAM_Q<0>,      !   Latch data into CAM
    ReadCAM           : ReadCAM_Q<0>,       !   Read CAM as RAM
    SearchCAM         : SearchCAM<0>,       !   Initiate associative search
    WriteCodeFIFO     : WriteCodeFIFO<0>,   !   Latch next output code
    ReadCharFIFO      : ReadCharFIFO<0>,    !   Input latched - get next input byte
    ReadCodeFIFO      : ReadCodeFIFO<0>,    !   Input latched - get next input code
    WriteStack        : WriteStack<0>,      !   Output char to stack
    FlipStack         : FlipStack<0>;       !   Begin string reversal

INPUT
    DoCompress : DoCompress<0>, !   Choose compress or decompress
    Reset      : Reset<0>,      !   Reinit vars, start (de)compression
    HaltIn     : HaltIn<0>,     !   Suspend execution temporarily
    DumpOmega  : DumpOmega<0>,  !   Output last code accumulated
    InCharFIFO< 7:0>,           !   Input from input character FIFO output
    InCodeFIFO<11:0>,           !   Input from input code FIFO output
    CAMInAddr<11:0>,            !   Input from CAM address bus
    CAMInData<19:0>,            !   Input from CAM data bus
    CAMMatch   : CAMMatch<0>;   !   Input from CAM match line

CLOCK
    CLK;                        !   System clock

SUPPLY      Vdd;   GROUND      GND;
```

Figure 30 - Merged_2.bdnet (cont. on next page)

```
INSTANCE    merged_2:logic PROMOTE;

!   Place flip-flops for flags first.

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
    DATA1:WriteCAM_D<0>;
    CLK2:CLK;
    Q:WriteCAM_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
    DATA1:ReadCAM_D<0>;
    CLK2:CLK;
    Q:ReadCAM_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

!   Now place flip-flops to form all the registers.

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
    DATA1:State_D<0>;
    CLK2:CLK;
    Q:State_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
    DATA1:OmitNextStr_D<0>;
    CLK2:CLK;
    Q:OmitNextStr_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

ARRAY %I FROM 0 TO 7 OF             !   Number of chars in current string
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
        DATA1:StrLen_D<%I>;
        CLK2:CLK;
        Q:StrLen_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   Next code symbol to generate
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
        DATA1:NextCode_D<%I>;
        CLK2:CLK;
        Q:NextCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   Pointer to accumulated string
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
        DATA1:Omega_D<%I>;
        CLK2:CLK;
        Q:Omega_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;
```

Figure 30 (cont.) - Merged_2.bdnet (cont. on next page)

```
ARRAY %I FROM 0 TO 7 OF              !   Next input character
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
        DATA1:FinChar_D<%I>;
        CLK2:CLK;
        Q:FinChar_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF             !   Last Omega searched for
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
        DATA1:OldCode_D<%I>;
        CLK2:CLK;
        Q:OldCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 19 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
        DATA1:D_CAMOutData_D<%I>;
        CLK2:CLK;
        Q:D_CAMOutData_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

!   Place flip-flops for latches on output busses.

ARRAY %I FROM 0 TO 11 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
        DATA1:CAMOutAddr_D<%I>;
        CLK2:CLK;
        Q:CAMOutAddr_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

!   Generate 12-bit XOR for Comparator.

ARRAY %I FROM 0 TO 11 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_0/xorf201":physical
        A1:NextCode_Q<%I>;
        B1:InCodeFIFO<%I>;
        O:Comparator<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

ENDMODEL;
```

Figure 30 (cont.) - Merged_2.bdnet

```
!=============================================================================
!
!   File:    m2_comp.musa       (/user1/icsg8038/thesis/m2_comp.musa)
!   Author:  Bob Wall
!   Date:    7/01/91
!
!   Description:
!       MUSA source file to exercise the compressor portion of the logic
!       described in merged_2.bds.  This file defines a set of vectors for the
!       various inputs and outputs of the compressor, as well as for internal
!       registers of interest.
!       It also defines macros to toggle the clock (to produce the negative
!       edge required by the flip-flops in the registers).
!
!   Notes:
!       This is essential compress_2.musa.
!
!   Revision History:
!       $Header$
!       $Log$
!
!=============================================================================

!   Define vectors to make setting and dumping registers and flags easier.
!   All the xxxIn vectors are the Q outputs of the corresponding registers,
!   and all the xxxOut vectors are the D inputs of those registers.
mv NextCodeIn   NextCode_Q<11:0>
mv OmegaIn      Omega_Q<11:0>
mv NextCodeOut  NextCode_D<11:0>
mv OmegaOut     Omega_D<11:0>

!   The next vectors are for the input and output signals for the compressor,
!   inputs first.
mv InCharFIFO   InCharFIFO<7:0>
mv OutCodeFIFO  OutCodeFIFO<11:0>
mv CAMInAddr    CAMInAddr<11:0>
mv CAMOutAddr   CAMOutAddr<11:0>
mv CAMOutData   CAMOutData<19:0>

!   Define collections of the input and output flags, for easy display.

mv  WriteCodeFIFO   WriteCodeFIFO<0>
mv  ReadCharFIFO    ReadCharFIFO<0>
mv  SearchCAM   SearchCAM<0>
mv  Reset       Reset<0>
mv  CAMMatch    CAMMatch<0>
mv  Inputs  Reset CAMMatch


!   Macros - first one sets the clock on, then off to produce the negative
!       edge required to trigger the flip-flops in the registers and flags.

ma  clock
se CLK 1
ev
se CLK 0
ev
$end

!   This one just displays the values in all the registers and flags.
ma showregs
sh NextCodeOut OmegaOut
sh InCharFIFO CAMInAddr
sh OutCodeFIFO CAMOutAddr CAMOutData
sh "Outputs - " WriteCodeFIFO ReadCharFIFO SearchCAM
sh "Inputs  (Reset CAMMatch) - %b\n" Inputs
$end
```

Figure 31 - M2_comp.musa (cont. on next page)

```
!   Start of compression simulation - first, just get the thing reset.
se DoCompress 1

se Reset 1
se InCharFIFO H55
ev
showregs
clock

se Reset 0
ev
showregs
clock

set CAMMatch 0
ev
showregs
clock

!   End of m2_comp.musa - all basic vectors and macros created.
!
```

Figure 31 (cont.) - M2_comp.musa

Appendix B

BDS and Bdnet Source for All Modules

```
!=============================================================================
!
!   File:    compress.bds    (/user1/icsg8038/thesis/compress.bds)
!   Author:  Bob Wall
!   Date:    8/08/90
!
!   Description:
!       BDS description of LZW compressor core (no FIFOs or CAM).
!
!   Notes:
!       This is a simplified version - it assumes that the input FIFO will
!       always have data available, and that the output FIFO will never be
!       full.  Also, the logic to limit the length of accumulated strings is
!       not implemented yet.
!
!       This algorithm assumes that the CAM is pre-initialized with all the
!       single-character strings - i.e. the first 256 entries are ROM
!       instead of CAM.
!
!       Since BDS does not support memory (i.e. flip-flops), all the regi-
!       sters and flags are set up as separate output and input signals,
!       which can be connected to D flip-flops using BDNET.
!
!       Note that some registers are "refreshed" in certain states.  A
!       register must be assigned some value for every state during every
!       clock cycle, unless it is acceptable for it to be filled with
!       possible garbage from the input logic.  If it does not get assigned
!       a new value, it should be reassigned its output to maintain that
!       value.
!
!       The last entry in the CAM (with address 0xfff) is not used, since
!       this is used as the NULL pointer for the decompressor.  Thus, if
!       NextCode reaches 0xfff, the table is full.
!
!   Revision History:
!       8/21/90  RLW  changed Data register to Omega_Old, left lower 8 bits
!                     in the K register.  Refresh K and Omega_Old on State = 1
!
!       $Header: /n/dali/u1/icsg8038/thesis/compress/RCS/compress.bds,v 1.2 1991/06/25
01:16:14 icsg8038 Exp $
!       $Log: compress.bds,v $
!       Revision 1.2  1991/06/25  01:16:14  icsg8038
!       Renamed InputFIFO to InCharFIFO, OutputFIFO to OutCodeFIFO,
!       ReadFIFO to ReadCharFIFO, and WriteFIFO to WriteCodeFIFO (to
!       match merged logic names).  Removed latch on ReadCharFIFO
!       (so it goes straight out to InCharFIFO controller now), and
!       deleted unnecessary OmegaOld register.
!
!       The latch on ReadCharFIFO caused the wrong input character
!       to be latched into K on the State 0 immediately following a
!       Reset.
!
!       Revision 1.1  91/04/28  23:09:23  icsg8038
!       Initial revision
!
!
!=============================================================================

MACRO REF_LEN  = 9      $ENDMACRO;  !   # bits in a refresh address - 1
MACRO REF_INIT = 111111111#2   $ENDMACRO;  !   Initializer of REF_LEN 1 bits
MACRO REF_EXT  = 00#2   $ENDMACRO;  !   LSBs to extend to a 12-bit address

MODEL Compressor

!   Circuit outputs and connections to all register and flag D inputs.
    Refresh_D   <REF_LEN:0>,   !   Register for next refresh address for CAM
    NextCode_D  <11:0>,        !   Register for next unused comp. code
```

Figure 32 - Compress.BDS (cont. on next page)

```
    Omega_D     <11:0>,         !   Register for ptr. to accumulated string
    K_D         < 7:0>,         !   Register for last input character
    State_D,                    !   Register for state variable
    WriteCAM_D,                 !   Flag, output to signal CAM data write
    SearchCAM_D,                !   Flag, output to signal CAM data search
    WriteCodeFIFO_D,            !   Flag, output to signal output FIFO write
    ReadCharFIFO,               !   Flag, output to signal input FIFO read
    DoRefresh_D,                !   Flag, output to signal CAM refresh cycle
    OutCodeFIFO <11:0>,         !   Output to output FIFO input
    CAMOutAddr  <11:0>,         !   Output to CAM address bus
    CAMOutData  <19:0>          !   Output to CAM data bus
=
!   Circuit inputs and connections to all register and flag Q outputs.
    Refresh_Q   <REF_LEN:0>,
    NextCode_Q  <11:0>,
    Omega_Q     <11:0>,
    K_Q         < 7:0>,
    State_Q,
    WriteCAM,
    DoRefresh,
    Reset,                      !   Input to initialize and start compressor
    InCharFIFO  < 7:0>,         !   Input from input FIFO output
    CAMInAddr   <11:0>,         !   Input from CAM address bus
    CAMMatch;                   !   Input from CAM match line

CONSTANT
    TRUE  = 1,
    FALSE = 0;

!=============================================================================
!
!   Routine to generate the next refresh address using a Galois-field based
!   sequencer.
!
!   Invocation: Refresh = NextRefresh (Refresh);
!
!   Input args: Refresh     RepLen-bit current refresh address
!
!   Notes:
!       The logic for the finite-field sequencer is very dependent on the
!       number of bits in the field (an irreducible polynomial of the correct
!       length is needed), so if the length of the refresh address is changed,
!       this routine MUST be modified.
!
!       For a ten-bit sequencer, a suitable minimum-weight polynomial is
!       0x409, which corresponds to X^10 + X^3 + 1.
!
!       Since 0 is a meta-stable state of the sequencer, the refresh register
!       should never be allowed to by all 0's.  It would be simple enough to
!       add a check in here - if CurrentRefresh = H000 THEN Return 0x3ff.
!
!       For more information on the subject of Galois-field based sequencers,
!       along with a table of minimum-weight irreducible polynomials, see the
!       paper "Galois-Field Based State Assignment for PLA Controllers" by
!       Kel Winters.
!
!=============================================================================

ROUTINE NextRefresh<REF_LEN:0> (CurrentRefresh<REF_LEN:0>);

    STATE   Temp<REF_LEN:0>,
            I<>;

    Temp<0> = CurrentRefresh<9>;
    Temp<1> = CurrentRefresh<0>;
    Temp<2> = CurrentRefresh<1>;
    Temp<3> = CurrentRefresh<2> XOR CurrentRefresh<9>;
```

Figure 32 (cont.) - Compress.BDS (cont. on next page)

```
    for I FROM 4 TO 9 DO
        Temp<I> = CurrentRefresh<I - 1>;

    return Temp;

ENDROUTINE NextRefresh;


!============================================================================
!
!   Main compressor routine - contains all the logic for sequencing through
!   the compression algorithm.
!
!   Calls:  NextRefresh ()
!
!============================================================================

ROUTINE  Compress;

    !   If a reset occurs, reset all the variables and prepare for compression
    If Reset EQL TRUE then
        Begin
        Refresh_D     = REF_INIT;
        NextCode_D    = 100#16;
        DoRefresh_D   = FALSE;
        WriteCAM_D    = FALSE;
        SearchCAM_D   = FALSE;
        WriteCodeFIFO_D = FALSE;
        ReadCharFIFO = TRUE;     !   Read first word to initialize Omega
        Omega_D       = 0#16 & InCharFIFO;
        State_D       = 0;
        End

    !   Otherwise, proceed with compression loop.  Check State to see if
    !   this is the first or second clock cycle of the loop.
    !
    !   On the first clock cycle, get the next input character, append it
    !   to the accumulated string, and search the CAM for the new string.
    Else if State_Q EQL 0 then
        Begin
        ReadCharFIFO = TRUE;         !   Read the next char from the input
        K_D          = InCharFIFO;

        Omega_D      = Omega_Q;      !   Refresh Omega
        WriteCodeFIFO_D = FALSE;     !   Turn off output write, if it was on
        SearchCAM_D  = TRUE;         !   Initiate search for input string
        CAMOutData   = Omega_Q & InCharFIFO;

        !   If the CAM was written last clock cycle, we need to go to the
        !   next code/address value this cycle.  In either case, WriteCAM
        !   should be turned off (or refreshed, if it was already off).
        If WriteCAM EQL TRUE then
            NextCode_D  = NextCode_Q + 1
        Else
            NextCode_D  = NextCode_Q;    !   Refresh NextCode
        WriteCAM_D   = FALSE;

        !   Similar to above.  If a refresh was performed, get the next
        !   refresh address this cycle.  Turn DoRefresh off regardless.
        If DoRefresh EQL TRUE then
            Refresh_D   = NextRefresh (Refresh_Q)
        Else
            Refresh_D   = Refresh_Q;     !   Refresh Refresh register
        DoRefresh_D  = FALSE;

        State_D      = 1;        !   Move on to next state
        End     !   Else if State_Q EQL 0
```

Figure 32 (cont.) - Compress.BDS (cont. on next page)

```
        !   On the second clock cycle, either update Omega to the new string,
        !   if the search was successful, or write the output code and start
        !   Omega over, if the search was unsuccessful.
        Else if State_Q EQL 1 then
            Begin
            NextCode_D    = NextCode_Q;  !   Refresh NextCode
            Refresh_D     = Refresh_Q;   !   Ditto for Refresh register
            K_D           = K_Q;         !       and K register (last data)

            SearchCAM_D   = FALSE;       !   Turn off CAM search enable
            ReadCharFIFO  = FALSE;       !       and input read enable

            If CAMMatch EQL TRUE then    !   Need to update Omega to new string
                Begin
                Omega_D     = CAMInAddr;     !   Latch match address as new string

                !   Since there was a match, we won't be writing a new string to
                !   the CAM.  Take this opportunity to refresh a row of the CAM.
                CAMOutAddr  = Refresh_Q & REF_EXT;
                DoRefresh_D = TRUE;
                WriteCAM_D  = FALSE;

                WriteCodeFIFO_D = FALSE;     !   Not generating an output code this time
                End

            Else                         !   Output code, reset Omega to K
                Begin
                !   Produce the next output code (just the value of Omega).
                OutCodeFIFO = Omega_Q;
                WriteCodeFIFO_D = TRUE;

                !   Reset Omega to the single-character string K.
                Omega_D     = 0#16 & K_Q;

                DoRefresh_D = FALSE;     !   No refresh while writing the CAM

                !   Need to add a new string to the table (Omega), if it is not
                !   filled up already.  Note that last entry is unused.
                If NextCode_Q NEQ fff#16 then   !   Room in the table
                    Begin
                    WriteCAM_D  = TRUE;
                    CAMOutAddr  = NextCode_Q;
                    CAMOutData  = Omega_Q & K_Q;    !   Write last Omega & K
                    End
                Else                     !   All full, sorry
                    WriteCAM_D  = FALSE;

                End;    !   Else

            State_D         = 0;         !   Back to the first state again

            End;    !   Else if State_Q EQL 1

    ENDROUTINE Compress;


    ENDMODEL Compressor;
```

Figure 32 (cont.) - Compress.BDS

```
!=============================================================================
!
!   File:    compress.bdnet    (/user1/icsg8038/thesis/compress.bdnet)
!   Author:  Bob Wall
!   Date:    8/13/90
!
!   Description:
!       BDNET description of LZW compressor core (no FIFOs or CAM).  This
!       file will place the flip-flops needed to hold register and flag
!       values in the compressor, and will connect up SUPPLY and GROUND
!       nodes.
!
!   Notes:
!       The dfnf311 instanced is a negative edge-triggered D flip-flop with
!       Q and Q_bar outputs.
!
!       The Refresh register length is given by the macro REF_LEN.  If it is
!       changed in compress.bds, it must be changed here.
!
!   Revision History:
!         $Header: /n/dali/u1/icsg8038/thesis/compress/RCS/compress.bdnet,v 1.5 1991/07/27
00:36:19 icsg8038 Exp $
!       $Log: compress.bdnet,v $
!       Revision 1.5  1991/07/27  00:36:19  icsg8038
!       Updated to version 2_2 of standard cell library.
!
!       Revision 1.4  1991/06/26  01:49:27  icsg8038
!       Cosmetics - cleaned up some comments.
!
!       Revision 1.3  91/06/26  00:45:43  icsg8038
!       Added latches on the CAMOutData, CAMOutAddr, and OutCodeFIFO busses.
!
!       Revision 1.2  91/06/24  23:05:06  icsg8038
!       Renamed InputFIFO to InCharFIFO, OutputFIFO to OutCodeFIFO,
!       ReadFIFO to ReadCharFIFO, and WriteFIFO to WriteCodeFIFO (to
!       match merged logic names).  Removed latch on ReadCharFIFO,
!       and deleted unnecessary OmegaOld register.
!
!       Revision 1.1  91/04/28  23:09:06  icsg8038
!       Initial revision
!
!
!=============================================================================

MACRO       REF_LEN =   9   $ENDMACRO;  !   Refresh addr length - 1


MODEL       compress:unplaced;
TECHNOLOGY  scmos;
VIEWTYPE    SYMBOLIC;
EDITSTYLE   SYMBOLIC;

OUTPUT
    OutCodeFIFO<11:0> : OutCodeFIFO_Q<11:0>,    !   Output to output FIFO input
    CAMOutAddr<11:0>  : CAMOutAddr_Q<11:0>,     !   Output to CAM address bus
    CAMOutData<19:0>  : CAMOutData_Q<19:0>,     !   Output to CAM data bus
    WriteCAM    : WriteCAM<0>,    !   Latch data into CAM
    SearchCAM   : SearchCAM<0>,   !   Initiate associative search
    WriteCodeFIFO : WriteCodeFIFO<0>,   !   Latch next output code
    ReadCharFIFO  : ReadCharFIFO<0>,    !   Input latched - get next input byte
    DoRefresh   : DoRefresh<0>;   !   Generate DCAM refresh, next ref. addr.

INPUT
    Reset       : Reset<0>,       !   Reinit vars, start compression
    InCharFIFO< 7:0>,             !   Input from input FIFO output
    CAMInAddr<11:0>,              !   Input from CAM address bus
    CAMMatch    : CAMMatch<0>;    !   Input from CAM match line
```

Figure 33 - Compress.bdnet (cont. on next page)

```
CLOCK
    CLK;                        !   System clock

SUPPLY      Vdd;
GROUND      GND;


INSTANCE    compress:logic PROMOTE;

!   Place flip-flops for flags first.

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:State_D<0>;
    CLK2:CLK;
    Q:State_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:WriteCAM_D<0>;
    CLK2:CLK;
    Q:WriteCAM<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:SearchCAM_D<0>;
    CLK2:CLK;
    Q:SearchCAM<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:WriteCodeFIFO_D<0>;
    CLK2:CLK;
    Q:WriteCodeFIFO<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:DoRefresh_D<0>;
    CLK2:CLK;
    Q:DoRefresh<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;


!   Now place flip-flops to form all the registers.

ARRAY %I FROM 0 TO REF_LEN  OF      !   Next DCAM refresh address
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:Refresh_D<%I>;
        CLK2:CLK;
        Q:Refresh_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;
```

Figure 33 (cont.) - Compress.bdnet (cont. on next page)

```
ARRAY %I FROM 0 TO 11 OF            !   Next code symbol to generate
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:NextCode_D<%I>;
        CLK2:CLK;
        Q:NextCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   Pointer to accumulated string
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:Omega_D<%I>;
        CLK2:CLK;
        Q:Omega_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF             !   Next input character
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:K_D<%I>;
        CLK2:CLK;
        Q:K_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;


!  Place flip-flops to latch the output busses.

ARRAY %I FROM 0 TO 19 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CAMOutData<%I>;
        CLK2:CLK;
        Q:CAMOutData_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CAMOutAddr<%I>;
        CLK2:CLK;
        Q:CAMOutAddr_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:OutCodeFIFO<%I>;
        CLK2:CLK;
        Q:OutCodeFIFO_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;


ENDMODEL;
```

Figure 33 (cont.) - Compress.bdnet

```
!============================================================================
!
!   File:    compress_2.bds    (/user1/icsg8038/thesis/compress_2.bds)
!   Author:  Bob Wall
!   Date:    6/23/91
!
!   Description:
!       BDS description of LZW compressor core (no FIFOs or CAM).  This is
!       a single-clock cycle version of the compressor, adapted from the
!       one described in compress.bds.  The difference between the two is
!       that it is now assumed that when a search operation is performed on
!       the DCAM, the search pattern will automatically be written into the
!       location specified by CAMInAddr.  Thus, if a match fails, the new
!       string has already been added to the table, and it is not necessary
!       to take another clock cycle to do that.
!
!   Notes:
!       This is a simplified version - it assumes that the input FIFO will
!       always have data available, and that the output FIFO will never be
!       full.  Also, the logic to limit the length of accumulated strings is
!       not implemented yet.
!
!       This algorithm assumes that the CAM is pre-initialized with all the
!       single-character strings - i.e. the first 256 entries are ROM
!       instead of CAM.
!
!       Since BDS does not support memory (i.e. flip-flops), all the regi-
!       sters and flags are set up as separate output and input signals,
!       which can be connected to D flip-flops using BDNET.
!
!       Note that some registers are "refreshed" in certain states.  A
!       register must be assigned some value for every state during every
!       clock cycle, unless it is acceptable for it to be filled with
!       possible garbage from the input logic.  If it does not get assigned
!       a new value, it should be reassigned its output to maintain that
!       value.
!
!       The last entry in the CAM (with address 0xfff) is not used, since
!       this is used as the NULL pointer for the decompressor.  Thus, if
!       NextCode reaches 0xfff, the table is full.
!
!       NOTE:  The output of the NextCode register is also the compressor's
!       CAMOutAddr buss.
!
!       The logic to refresh the DCAM which was contained in compress.bds
!       has now been deferred to the controller - the compressor should
!       be periodically halted and the controller should refresh a row of
!       the DCAM.
!
!   Revision History:
!         $Header: /n/dali/u1/icsg8038/thesis/compress/RCS/compress_2.bds,v 1.3 1991/07/01
23:20:20 icsg8038 Exp $
!       $Log: compress_2.bds,v $
!       Revision 1.3  1991/07/01  23:20:20  icsg8038
!       Removed latches on SearchCAM, WriteCodeFIFO, CAMOutData,
!       and OutCodeFIFO.
!
!       Revision 1.2  91/06/26  01:14:05  icsg8038
!       Removed latch on ReadCharFIFO control signal.
!
!       Revision 1.1  91/06/25  22:04:36  icsg8038
!       Initial revision
!
!
!============================================================================

MODEL Compressor
```

Figure 34 - Compress_2.BDS (cont. on next page)

```
    !   Circuit outputs and connections to all register and flag D inputs.
    NextCode_D  <11:0>,         !   Register for next unused comp. code
    Omega_D     <11:0>,         !   Register for ptr. to accumulated string
    WriteCodeFIFO,              !   Output to signal output FIFO write
    SearchCAM,                  !   Output to signal CAM data search
    ReadCharFIFO,               !   Output to signal input FIFO read
    OutCodeFIFO <11:0>,         !   Output to output code FIFO input
    CAMOutData  <19:0>          !   Output to CAM data bus

!   The CAMOutAddr output is tied directly to NextCode_Q (the latched
!   NextCode) in compress_2.bdnet.

=

!   Circuit inputs and connections to all register and flag Q outputs.
    NextCode_Q  <11:0>,
    Omega_Q     <11:0>,
    Reset,                      !   Input to initialize and start compressor
    InCharFIFO  < 7:0>,         !   Input from input byte FIFO output
    CAMInAddr   <11:0>,         !   Input from CAM address bus
    CAMMatch;                   !   Input from CAM match line

CONSTANT
    TRUE  = 1,
    FALSE = 0;


!=============================================================================
!
!   Main compressor routine - contains all the logic for sequencing through
!   the compression algorithm.
!
!   Calls:  !  NextRefresh ()
!
!=============================================================================

ROUTINE  Compress;

    !   If a reset occurs, reset all the variables and prepare for compression
    If Reset EQL TRUE then
        Begin
        NextCode_D    = 100#16;
        SearchCAM     = FALSE;
        WriteCodeFIFO = FALSE;
        ReadCharFIFO  = TRUE;           !   Read first word to initialize Omega
        Omega_D       = 0#16 & InCharFIFO;
        End

    !   Otherwise, proceed with compression loop.  It is just one clock cycle
    !   now.
    Else
        Begin
        ReadCharFIFO  = TRUE;           !   Read the next char from the input

        SearchCAM     = TRUE;           !   Initiate search for input string
        CAMOutData    = Omega_Q & InCharFIFO;

        !   Now, hang out and "wait" for the results of the search to come
        !   back.  They should be done in sufficient time to finish processing
        !   in this clock cycle.
        !   NOTE:  Once the CAM decides whether the search matched or not, it
        !   must hold the CAMMatch signal at that level through the end of the
        !   clock cycle.
```

Figure 34 (cont.) - Compress_2.BDS (cont. on next page)

```
        If CAMMatch EQL TRUE then       !   Need to update Omega to new string
            Begin
            Omega_D    = CAMInAddr;     !   Latch match address as new string
            WriteCodeFIFO = FALSE;      !   Not generating an output this time
            NextCode_D  = NextCode_Q;   !   Retain value in NextCode register
            End

        Else                            !   Output code, reset Omega to K
            Begin
            !   Produce the next output code (just the value of Omega).
            OutCodeFIFO = Omega_Q;
            WriteCodeFIFO = TRUE;

            !   Reset Omega to the single-character string K.
            Omega_D     = 0#16 & InCharFIFO;

            !   The new string (Omega . K) has already been added to the table.
            !   Just need to update the place where the next string will be
            !   written, if the table has not filled up already.  Note that the
            !   last entry is unused.
            !   Note that CAMOutAddr will be the same as NextCode_Q, if it is
            !   latched.

            If NextCode_Q NEQ fff#16 then   !   Room in the table
                NextCode_D  = NextCode_Q + 1
            Else                        !   All full, sorry
                NextCode_D  = NextCode_Q;   !   Retain current NextCode value

            End;    !   Else (!CAMMatch)

        End;    !   Else (!Reset)

ENDROUTINE Compress;


ENDMODEL Compressor;
```

Figure 34 (cont.) - Compress_2.BDS

```
!=============================================================================
!
!   File:    compress_2.bdnet    (/user1/icsg8038/thesis/compress_2.bdnet)
!   Author:  Bob Wall
!   Date:    6/23/91
!
!   Description:
!       BDNET description of LZW compressor core (no FIFOs or CAM).  This
!       file will place the flip-flops needed to hold register and flag
!       values in the compressor, and will connect up SUPPLY and GROUND
!       nodes.
!       This is a single-clock cycle version of the compressor, adapted from
!       the original two cycle per input byte design.
!
!   Notes:
!       The dfnf311 instanced is a negative edge-triggered D flip-flop with
!       Q and Q_bar outputs.
!
!       The NextCode register's output is also connected to the compressor's
!       CAMOutAddr buss.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/compress/RCS/compress_2.bdnet,v 1.4 1991/07/27
00:36:19 icsg8038 Exp $
!       $Log: compress_2.bdnet,v $
!       Revision 1.2  1991/07/01  23:19:53  icsg8038
!       Removed latches on SearchCAM, WriteCodeFIFO, CAMOutData,
!       ReadCodeFIFO, and OutCodeFIFO.
!
!       Revision 1.1  91/06/25  22:04:22  icsg8038
!       Initial revision
!
!
!=============================================================================

MODEL       compress_2:unplaced;
TECHNOLOGY  scmos;
VIEWTYPE    SYMBOLIC;
EDITSTYLE   SYMBOLIC;

OUTPUT
    CAMOutAddr<11:0>  : NextCode_Q<11:0>,   !   Output to CAM address bus
    CAMOutData<19:0>  : CAMOutData<19:0>,   !   Output to CAM data bus
    OutCodeFIFO<11:0> : OutCodeFIFO<11:0>,! Output to output FIFO input
    SearchCAM     : SearchCAM<0>,       !   Initiate associative search
    WriteCodeFIFO : WriteCodeFIFO<0>,   !   Latch next output code
    ReadCharFIFO  : ReadCharFIFO<0>;    !   Input latched - get next input byte

INPUT
    Reset      : Reset<0>,      !   Reinit vars, start compression
    InCharFIFO<7:0>,            !   Input from input FIFO output
    CAMInAddr<11:0>,            !   Input from CAM address bus
    CAMMatch   : CAMMatch<0>;   !   Input from CAM match line

CLOCK
    CLK;                        !   System clock

SUPPLY      Vdd;
GROUND      GND;

INSTANCE    compress_2:logic PROMOTE;

!   Place flip-flops to form all the registers.
```

Figure 35 - Compress_2.bdnet (cont. on next page)

```
ARRAY %I FROM 0 TO 11 OF            !   Next code symbol to generate
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:NextCode_D<%I>;
        CLK2:CLK;
        Q:NextCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   Pointer to accumulated string
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:Omega_D<%I>;
        CLK2:CLK;
        Q:Omega_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ENDMODEL;
```

Figure 35 (cont.) - Compress_2.bdnet

```
!=============================================================================
!
!   File:    decompress.bds    (/user1/icsg8038/thesis/decompress.bds)
!   Author:  Bob Wall
!   Date:    10/13/90
!
!   Description:
!       BDS description of LZW decompressor core (no FIFOs, CAM, stack).
!
!   Notes:
!       This is a simplified version - it assumes that the input FIFO will
!       always have data available, and that the output FIFO will never be
!       full.  Also, the logic to handle length-limited compression strings
!       is not implemented yet.
!
!       This algorithm assumes that the CAM is pre-initialized with all the
!       single-character strings - i.e. the first 256 entries are ROM
!       instead of CAM.
!
!       Since BDS does not support memory (i.e. flip-flops), all the regi-
!       sters and flags are set up as separate output and input signals,
!       which can be connected to D flip-flops using BDNET.
!
!       Note that some registers are "refreshed" in certain states.  A
!       register must be assigned some value for every state during every
!       clock cycle, unless it is acceptable for it to be filled with
!       possible garbage from the input logic.  If it does not get assigned
!       a new value, it should be reassigned its output to maintain that
!       value.
!
!       The last entry in the CAM (with address 0xfff) is not used, since
!       this is used as the NULL pointer for the decompressor.  Thus, if
!       NextCode reaches 0xfff, the table is full.
!
!       It is expected that the "stack" used by the decompressor will
!       actually consist of two or more stacks.  Characters will be pushed
!       on the current stack on every WriteStack signal until FlipStack
!       is asserted, then those characters will be popped off the stack
!       and passed to the output FIFO while new characters are pushed on
!       the next available stack.
!
!       Still need to add capability for controller to put decompressor
!       in wait state while freeing up a stack or waiting for input.
!
!       misII is choking on a 12-bit comparator, such as the one used in
!       State 10, comparing NextCode_Q to InCode_Q.  It is apparently evalu-
!       ating every combination of the 24 input bits to try to simplify the
!       comparator.  In order to bypass this, had to add an extra 12-bit
!       input, Comparator, and assign it NextCode_Q XOR InCode_Q.  This result
!       is then compared to 0 to see if the two are equal.  The Comparator is
!       generated in decompress.bds.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/decompress/RCS/decompress.bds,v 1.3 1991/07/01
00:38:51 icsg8038 Exp $
!       $Log: decompress.bds,v $
!       Revision 1.3  1991/07/01  00:38:51  icsg8038
!       Removed the Temp output, added Comparator input.  Same thing - generate
!       a 12-bit XOR with bdnet instead of internally.
!
!       Revision 1.2  91/06/26  10:46:28  icsg8038
!       Removed latch on ReadCodeFIFO control signal.
!
!       Revision 1.1  91/04/28  23:26:46  icsg8038
!       Initial revision
!
!=============================================================================
```

Figure 36 - Decompress.BDS (cont. on next page)

```
MACRO REF_LEN  = 9      $ENDMACRO;  !   # bits in a refresh address - 1
MACRO REF_INIT = 111111111#2   $ENDMACRO;  !   Initializer of REF_LEN 1 bits
MACRO REF_EXT  = 00#2   $ENDMACRO;  !   LSBs to extend to a 12-bit address
MACRO NULL_PREFIX = fff#16      $ENDMACRO;  !   NULL link for strings

MODEL Decompressor

!   Circuit outputs and connections to all register and flag D inputs.
    Refresh_D   <REF_LEN:0>,    !   Register for next refresh address for CAM
    NextCode_D  <11:0>,         !   Register for next unused comp. code
    OldCode_D   <11:0>,         !   Register for previous input code
    InCode_D    <11:0>,         !   Register for newest input code
    FinChar_D   < 7:0>,         !   Register for last char of previous string
    State_D     < 1:0>,         !   Register for state variable
    WriteCAM_D,                 !   Flag, output to signal CAM data write
    ReadCAM_D,                  !   Flag, output to signal CAM data read
    ReadCodeFIFO,               !   Flag, output to signal input FIFO read
    WriteStack,                 !   Flag, output to signal stack data write
    FlipStack_D,                !   Flag, output to signal stack reversal
    DoRefresh_D,                !   Flag, output to signal CAM refresh cycle
    StackData   < 7:0>,         !   Output to string reversal stack
    CAMOutAddr  <11:0>,         !   Output to CAM address bus
    CAMOutData  <19:0>          !   Output to CAM data bus

=

!   Circuit inputs and connections to all register and flag Q outputs.
    Refresh_Q   <REF_LEN:0>,
    NextCode_Q  <11:0>,
    OldCode_Q   <11:0>,
    InCode_Q    <11:0>,
    FinChar_Q   < 7:0>,
    State_Q     < 1:0>,
    WriteCAM,
!   DoRefresh,                  !   Flip-flop outputs are not currently
!   ReadCAM,                    !     needed, since they are never referenced
!   ReadCodeFIFO,               !     in the code.
!   WriteStack,
!   FlipStack,
!   DoRefresh,
    Reset,                      !   Input to initialize and start compressor
    InCodeFIFO  <11:0>,         !   Input from input code FIFO output
    CAMInData   <19:0>,         !   Input from CAM data bus
    Comparator  <11:0>;         !   External XOR for 12-bit compare.

CONSTANT
    TRUE  = 1,
    FALSE = 0;


!=============================================================================
!
!   Routine to generate the next refresh address using a Galois-field based
!   sequencer.
!
!   Invocation: Refresh = NextRefresh (Refresh);
!
!   Input args: Refresh     RepLen-bit current refresh address
!
!   Notes:
!       The logic for the finite-field sequencer is very dependent on the
!       number of bits in the field (an irreducible polynomial of the correct
!       length is needed), so if the length of the refresh address is changed,
!       this routine MUST be modified.
!
!       For a ten-bit sequencer, a suitable minimum-weight polynomial is
!       0x409, which corresponds to X^10 + X^3 + 1.
```

Figure 36 (cont.) - Decompress.BDS (cont. on next page)

```
!       Since 0 is a meta-stable state of the sequencer, the refresh register
!       should never be allowed to by all 0's.  It would be simple enough to
!       add a check in here - if CurrentRefresh = H000 THEN Return 0x3ff.
!       For more information on the subject of Galois-field based sequencers,
!       along with a table of minimum-weight irreducible polynomials, see the
!       paper "Galois-Field Based State Assignment for PLA Controllers" by
!       Kel Winters.
!
!=============================================================================

ROUTINE NextRefresh<REF_LEN:0> (CurrentRefresh<REF_LEN:0>);

    STATE   Ref_Temp<REF_LEN:0>,
            I<>;

    Ref_Temp<0> = CurrentRefresh<9>;
    Ref_Temp<1> = CurrentRefresh<0>;
    Ref_Temp<2> = CurrentRefresh<1>;
    Ref_Temp<3> = CurrentRefresh<2> XOR CurrentRefresh<9>;
    for I FROM 4 TO 9 DO
        Ref_Temp<I> = CurrentRefresh<I - 1>;

    return Ref_Temp;

ENDROUTINE NextRefresh;

!=============================================================================
!
!   Main decompression routine - contains all the logic for sequencing through
!   the decompression algorithm.
!
!   Calls:  NextRefresh ()
!
!=============================================================================

ROUTINE  Decompress;

    !   If a reset occurs, reset all the variables and get the first input
    !   code to start decompresion.
    !
    If Reset EQL TRUE then
        Begin
        Refresh_D    = REF_INIT;        !   Register initialization stuff
        NextCode_D   = 100#16;
        DoRefresh_D  = FALSE;
        WriteCAM_D   = FALSE;
        WriteStack   = FALSE;
        FlipStack_D  = FALSE;
        ReadCodeFIFO = TRUE;        !   Read first word to initialize OldCode
        OldCode_D    = InCodeFIFO;
        ReadCAM_D    = TRUE;        !   Get the character for the first code
        CAMOutAddr   = InCodeFIFO;

        State_D  = 00#2;
        End     !   If Refresh

    !   Second half of initialization - get the resulting character from
    !   the CAM and write it out, then get ready to do main decompression
    !   loop.
    !
    Else if State_Q EQL 00#2 then
        Begin
        FinChar_D    = CAMInData<7:0>;  !   Record as last char of string
        WriteStack   = TRUE;        !   Write out single-character string
        FlipStack_D  = TRUE;
        StackData    = CAMInData<7:0>;
        ReadCodeFIFO = TRUE;        !   Get next input code to start decomp.
```

Figure 36 (cont.) - Decompress.BDS (cont. on next page)

```
    InCode_D     = InCodeFIFO;
    ReadCAM_D    = FALSE;        !   Turn off unneeded flags
    WriteCAM_D   = FALSE;
    DoRefresh_D  = FALSE;
    NextCode_D   = NextCode_Q;   !   Maintain register contents
    Refresh_D    = Refresh_Q;
    OldCode_D    = OldCode_Q;

    State_D  = 01#2;             !   Start main decompression loop
    End      !   Else if State == 00

!   Otherwise, proceed with decompression loop.  The new input code should
!   already be in InCode.  Determine whether the code is in the table, and
!   prepare to create the decompressed string.
!
Else if State_Q EQL 01#2 then
    Begin
    If Comparator EQL 000#16 then   !   InCode_Q == NextCode_Q
        Begin                       !   Special case - InCode not in table
        WriteStack     = TRUE;      !   Push FinChar
        StackData      = FinChar_Q;
        CAMOutAddr     = OldCode_Q; !   And assume code is the prev. one
        End
    Else
        Begin
        WriteStack     = FALSE;     !   Don't need to push anything yet
        CAMOutAddr     = InCode_Q;  !   Read table entry for new code
        End;

    ReadCAM_D    = TRUE;            !   Get the appropriate table entry.
    WriteCAM_D   = FALSE;           !   Turn off unneeded flags
    FlipStack_D  = FALSE;
    ReadCodeFIFO = FALSE;

    !   Get the next refresh address.  The only time that this state
    !   doesn't follow state 11#2, when the refresh is done, is on init-
    !   ialization, and it is not crucial that Refresh have any particular
    !   value on the first refresh cycle, so don't worry about getting an
    !   unnecessary NextRefresh () value.
    !
    Refresh_D    = NextRefresh(Refresh_Q);
    DoRefresh_D  = FALSE;
    NextCode_D   = NextCode_Q;      !   Maintain register contents
    OldCode_D    = OldCode_Q;
    InCode_D     = InCode_Q;

    State_D  = 10#2;                !   Get into string decompress loop
    End      !   Else if State == 01

!   The meat of the decompressor - assumes that the CAM is returning the
!   string table entry for a given code.  A code is decompressed by recur-
!   sively pushing the character portion of the table entry on the stack,
!   and reading the entry for the code portion of the entry (the link to
!   the remainder of the string).  The recursion terminates when a NULL
!   link (value 0xfff) is reached.  The last character is pushed, the
!   stack is signalled to begin popping the string, and a new code is
!   added to the string table.
!
Else if State_Q EQL 10#2 then
    Begin
    WriteStack   = TRUE;            !   Push character from table entry
    StackData    = CAMInData<7:0>;

    If CAMInData<19:8> NEQ NULL_PREFIX then
        Begin                       !   Haven't reached start of string yet
        ReadCAM_D    = TRUE;        !   Continue tracing string back
        CAMOutAddr   = CAMInData<19:8>;
```

Figure 36 (cont.) - Decompress.BDS (cont. on next page)

```
            FlipStack_D  = FALSE;        !  Don't reverse string yet
            WriteCAM_D   = FALSE;
            OldCode_D    = OldCode_Q;    !  Maintain register contents

            State_D  = 10#2;             !  Remain in this state
            End      !  If PrefixPtr != NULL_PREFIX

        Else                             !  Found the head of the string
            Begin
            ReadCAM_D    = FALSE;        !  Stop tracing links
            FlipStack_D  = TRUE;         !  Can start outputting string now
            FinChar_D    = CAMInData<7:0>; !  Record last character in string
            OldCode_D    = InCode_Q;     !  Remember last code

            If NextCode_Q NEQ fff#16 then   !  Table is not full yet
                Begin
                WriteCAM_D   = TRUE;     !  Add new entry
                CAMOutData   = OldCode_Q & CAMInData<7:0>;
                CAMOutAddr   = NextCode_Q;
                End
            Else
                WriteCAM_D   = FALSE;    !  No room for new entry

            State_D  = 11#2;             !  Done with loop - clear outta here
            End;     !  Else

        ReadCodeFIFO = FALSE;            !  Turn off unneeded flags
        DoRefresh_D  = FALSE;

    !   Mod  3/05/91  RLW  moved InCode_D down here from inside IF above,
    !                      since it needs to be "refreshed" in either case.
        InCode_D     = InCode_Q;         !  Maintain register contents
        NextCode_D   = NextCode_Q;
        Refresh_D    = Refresh_Q;
        End      !  Else if State == 10

    !   A clean-up state, added just so the refresh could get done.  This re-
    !   lies on the fact that compressed string lengths will be limited to some
    !   relatively small number of characters.  Does a refresh and gets the
    !   next code to set up the decompression loop again.
    !
    Else if State_Q EQL 11#2 then
        Begin
        DoRefresh_D  = TRUE;             !  Slap the next refresh addr on bus
        CAMOutAddr   = Refresh_Q & REF_EXT;
        ReadCodeFIFO = TRUE;             !  Get next input code
        InCode_D     = InCodeFIFO;
        WriteStack   = FALSE;            !  Reset unneeded flags
        FlipStack_D  = FALSE;
        ReadCAM_D    = FALSE;

    !   Mod  3/05/91  RLW  use WriteCAM (input), not WriteCam_D (output)
        If WriteCAM EQL TRUE then        !  Added string to table last state
            NextCode_D = NextCode_Q + 1 !  Need to update NextCode value
        Else
            NextCode_D = NextCode_Q;     !  Table full - maintain register

        WriteCAM_D   = FALSE;
        Refresh_D    = Refresh_Q;        !  Maintain register contents
        OldCode_D    = OldCode_Q;
        FinChar_D    = FinChar_Q;

        State_D  = 01#2;                 !  Return to top of decompression loop
        End;     !  Else if State == 11

ENDROUTINE Decompress;
ENDMODEL Decompressor;
```

Figure 36 (cont.) - Decompress.BDS

```
!=============================================================================
!
!   File:    decompress.bdnet    (/user1/icsg8038/thesis/decompress.bdnet)
!   Author:  Bob Wall
!   Date:    10/15/90
!
!   Description:
!       BDNET description of LZW decompressor core (no FIFO, CAM, or stack).
!       This file will place the flip-flops needed to hold register and flag
!       values in the decompressor, and will connect up SUPPLY and GROUND
!       nodes.
!
!   Notes:
!       The dfnf311 instanced is a negative edge-triggered D flip-flop with
!       Q and Q_bar outputs.
!
!       The Refresh register length is given by the macro REF_LEN.  If it is
!       changed in decompress.bds, it must be changed here.  These should be
!       the same as the values in compress.bds.
!
!       The Comparator function generated here is just a 12-bit XOR of the
!       NextCode_Q and InCodeFIFO signals.  It is used internally in the
!       decompressor to check if the two signals are equal.  The xorf201
!       is a simple two-input xor gate.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/decompress/RCS/decompress.bdnet,v 1.4 1991/07/27
00:46:39 icsg8038 Exp $
!       $Log: decompress.bdnet,v $
!       Revision 1.4  1991/07/27  00:46:39  icsg8038
!       Update from version 2.0 to version 2.2 of standard cell library
!
!       Revision 1.3  1991/07/01  00:37:57  icsg8038
!       Added Comparator - NextCode_Q XOR InCode_Q - generated to
!       replace the 12-bit XOR in decompress.bds.
!
!       Revision 1.2  91/06/26  10:46:04  icsg8038
!       Removed latch on ReadCodeFIFO control signal.  Also added
!       latches on the OutCodeFIFO, CAMOutAddr, and CAMOutData busses.
!
!       Revision 1.1  91/04/28  23:26:35  icsg8038
!       Initial revision
!
!=============================================================================

MACRO       REF_LEN =   9   $ENDMACRO;  !   Refresh addr length - 1


MODEL       decompress:unplaced;
TECHNOLOGY  scmos;
VIEWTYPE    SYMBOLIC;
EDITSTYLE   SYMBOLIC;

OUTPUT
    StackData<7:0>,                 !   Output to stack data bus
    CAMOutAddr<11:0> : CAMOutAddr_Q<11:0>,  !   Output to CAM address bus
    CAMOutData<19:0> : CAMOutData_Q<19:0>,  !   Output to CAM data bus
    WriteCAM     : WriteCAM<0>,     !   Latch data into CAM
    ReadCAM      : ReadCAM<0>,      !   Read CAM as a RAM
    ReadCodeFIFO : ReadCodeFIFO<0>, !   Latched input code - get new one
    WriteStack   : WriteStack<0>,   !   Push next char. on stack
    FlipStack    : FlipStack<0>,    !   Switch stacks, start reversing string
    DoRefresh    : DoRefresh<0>;    !   Generate DCAM refresh, next ref. addr.

INPUT
    Reset        : Reset<0>,        !   Reinit vars, start compression
    InCodeFIFO<11:0>,               !   Input from code FIFO output
    CAMInData<19:0>;                !   Input from CAM data bus
```

Figure 37 - Decompress.bdnet (cont. on next page)

```
CLOCK
    CLK;                            !   System clock

SUPPLY      Vdd;
GROUND      GND;

INSTANCE    decompress:logic PROMOTE;

!   Place flip-flops for flags first.
INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:WriteCAM_D<0>;
    CLK2:CLK;
    Q:WriteCAM<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:ReadCAM_D<0>;
    CLK2:CLK;
    Q:ReadCAM<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:FlipStack_D<0>;
    CLK2:CLK;
    Q:FlipStack<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:DoRefresh_D<0>;
    CLK2:CLK;
    Q:DoRefresh<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

!   Now place flip-flops to form all the registers.

ARRAY %I FROM 0 TO REF_LEN  OF      !   Next DCAM refresh address
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:Refresh_D<%I>;
        CLK2:CLK;
        Q:Refresh_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   Next code symbol to generate
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:NextCode_D<%I>;
        CLK2:CLK;
        Q:NextCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   Previous input code
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:OldCode_D<%I>;
        CLK2:CLK;
        Q:OldCode_Q<%I>;
        Q_b:UNCONNECTED;
```

Figure 37 (cont.) - Decompress.bdnet (cont. on next page)

```
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   New input code
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:InCode_D<%I>;
        CLK2:CLK;
        Q:InCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:State_D<0>;
    CLK2:CLK;
    Q:State_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:State_D<1>;
    CLK2:CLK;
    Q:State_Q<1>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

ARRAY %I FROM 0 TO 7 OF             !   Final character of last string
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:FinChar_D<%I>;
        CLK2:CLK;
        Q:FinChar_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

!   Place flip-flops to latch the CAM busses.
ARRAY %I FROM 0 TO 19 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CAMOutData<%I>;
        CLK2:CLK;
        Q:CAMOutData_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CAMOutAddr<%I>;
        CLK2:CLK;
        Q:CAMOutAddr_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

!   Generate 12-bit XOR for Comparator.
ARRAY %I FROM 0 TO 11 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/xorf201":physical
        A1:NextCode_Q<%I>;
        B1:InCode_Q<%I>;
        O:Comparator<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

ENDMODEL;
```

Figure 37 (cont.) - Decompress.bdnet

```
!============================================================================
!
!   File:    decompress_2.bds    (/user1/icsg8038/thesis/decompress_2.bds)
!   Author:  Bob Wall
!   Date:    6/30/91
!
!   Description:
!       BDS description of LZW decompressor core (no FIFOs, CAM, stack).
!       Modified substantailly from original decompress.bds - the refresh
!       operation is now being handled by the controller, so the additional
!       state required for refresh has been eliminated.  In addition, the
!       two state required for initialization have been combined.
!
!   Notes:
!       This is a simplified version - it assumes that the input FIFO will
!       always have data available, and that the output FIFO will never be
!       full.  Also, the logic to handle length-limited compression strings
!       is not implemented yet.
!
!       This algorithm assumes that the CAM is pre-initialized with all the
!       single-character strings - i.e. the first 256 entries are ROM
!       instead of CAM.
!
!       Since BDS does not support memory (i.e. flip-flops), all the regi-
!       sters and flags are set up as separate output and input signals,
!       which can be connected to D flip-flops using BDNET.
!
!       Note that some registers are "refreshed" in certain states.  A
!       register must be assigned some value for every state during every
!       clock cycle, unless it is acceptable for it to be filled with
!       possible garbage from the input logic.  If it does not get assigned
!       a new value, it should be reassigned its output to maintain that
!       value.
!
!       The last entry in the CAM (with address 0xfff) is not used, since
!       this is used as the NULL pointer for the decompressor.  Thus, if
!       NextCode reaches 0xfff, the table is full.
!
!       It is expected that the "stack" used by the decompressor will
!       actually consist of two or more stacks.  Characters will be pushed
!       on the current stack on every WriteStack signal until FlipStack
!       is asserted, then those characters will be popped off the stack
!       and passed to the output FIFO while new characters are pushed on
!       the next available stack.
!
!       Still need to add capability for controller to put decompressor
!       in wait state while freeing up a stack or waiting for input.
!
!       Stupid !@@#@??!@#$ misII is choking on a 12-bit comparator, such as
!       the one used in State 0, comparing NextCode_Q to InCodeFIFO.  It is
!       apparently evaluating every combination of the 24 input bits to try
!       to simplify the comparator - this naturally causes a few page
!       faults during logic minimization.  In order to bypass this, had to add
!       an extra 12-bit input, Comparator, which is just a 12-bit XOR of
!       NextCode_Q and InCodeFIFO generated in decompress_2.bdnet.
!       This input is just compared to 0 to see if the two are equal.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/decompress/RCS/decompress_2.bds,v 1.1 1991/07/16
00:41:51 icsg8038 Exp $
!       $Log: decompress_2.bds,v $
!       Revision 1.1  1991/07/16  00:41:51  icsg8038
!       Initial revision
!
!============================================================================

MODEL Decompressor
```

Figure 38 - Decompress_2.BDS (cont. on next page)

```
!   Circuit outputs and connections to all register and flag D inputs.
    NextCode_D  <11:0>,         !   Register for next unused comp. code
    OldCode_D   <11:0>,         !   Register for previous input code
    InCode_D    <11:0>,         !   Register for newest input code
    FinChar_D   < 7:0>,         !   Register for last char of previous string
    State_D,                    !   Register for state variable
    WriteCAM_D,                 !   Flag, output to signal CAM data write
    ReadCAM_D,                  !   Flag, output to signal CAM data read
    ReadCodeFIFO,               !   Flag, output to signal input FIFO read
    WriteStack,                 !   Flag, output to signal stack data write
    FlipStack,                  !   Flag, output to signal stack reversal
    StackData   < 7:0>,         !   Output to string reversal stack
    CAMOutAddr_D<11:0>,         !   Output to CAM address bus
    CAMOutData_D<19:0>          !   Output to CAM data bus
=
!   Circuit inputs and connections to all register and flag Q outputs.
    NextCode_Q  <11:0>,
    OldCode_Q   <11:0>,
    InCode_Q    <11:0>,
    FinChar_Q   < 7:0>,
    State_Q,
    ReadCAM_Q,                  !   Latched CAM Read signal
    CAMOutAddr_Q<11:0>,         !   Latched CAM output address
    Reset,                      !   Input to initialize and start compressor
    InCodeFIFO  <11:0>,         !   Input from input code FIFO output
    CAMInData   <19:0>,         !   Input from CAM data bus
    Comparator  <11:0>;         !   NextCode_Q XOR InCodeFIFO


CONSTANT
    TRUE  = 1,
    FALSE = 0;


!===============================================================================
!
!   Main decompression routine - contains all the logic for sequencing through
!   the decompression algorithm.
!
!   Calls:  None
!
!===============================================================================

ROUTINE  Decompress;

    !   If a reset occurs, reset all the variables and get the first input
    !   code to start decompression.
    !
    If Reset EQL TRUE then
        Begin
        NextCode_D   = 100#16;      !   Register initialization stuff

        !   Read the first code; since it is guaranteed to represent a single
        !   character string, the character is just the last eight bits.
        !   Write that string to the stack.
        OldCode_D    = InCodeFIFO;
        StackData    = InCodeFIFO<7:0>;
        FinChar_D    = InCodeFIFO<7:0>;
        WriteStack   = TRUE;
        FlipStack    = TRUE;
        ReadCodeFIFO = TRUE;
        WriteCAM_D   = FALSE;
        ReadCAM_D    = FALSE;

        State_D      = 0;
        End     !   If Reset
```

Figure 38 (cont.) - Decompress_2.BDS (cont. on next page)

```
!   Proceed with decompression loop.  The new input code should be ready
!   in InCodeFIFO.  Read it and latch into InCode, then determine whether
!   the code is in the table and prepare to create decompressed string.
!
Else if State_Q EQL 0 then
    Begin
    If Comparator EQL 000#16 then   !   InCodeFIFO == NextCode_Q
        Begin                       !   Special case - InCode not in table
        WriteStack      = TRUE;     !   Push FinChar
        StackData       = FinChar_Q;
        CAMOutAddr_D    = OldCode_Q; !  And assume code is the prev. one
        End

    Else
        Begin
        WriteStack      = FALSE;    !   Don't need to push anything yet
        CAMOutAddr_D    = InCodeFIFO;!  Read table entry for new code
        End;

    If CAMOutAddr_D<11:8> EQL 0 then    !   Just a single character string
        ReadCAM_D       = FALSE     !   Don't need to read it from DCAM
    Else
        ReadCAM_D       = TRUE;     !   Get the appropriate table entry.

    InCode_D      = InCodeFIFO;
    ReadCodeFIFO = TRUE;
    FlipStack     = FALSE;
    WriteCAM_D    = FALSE;          !   Turn off unneeded flags
    NextCode_D    = NextCode_Q;     !   Maintain register contents
    OldCode_D     = OldCode_Q;
    State_D       = 1;              !   Get into main decompress loop
    End     !   Else if State == 0

!   The meat of the decompressor - assumes that the CAM is returning the
!   string table entry for a given code.  A code is decompressed by recur-
!   sively pushing the character portion of the table entry on the stack,
!   and reading the entry for the code portion of the entry (the link to
!   the remainder of the string).  The recursion terminates when a code
!   less than 0x100 is reached.  The last character is just the lower
!   eight bits of this code.  It is pushed, the stack is signalled to
!   begin popping the string, and a new code is added to the string table.

Else if State_Q EQL 1 then
    Begin
    WriteStack    = TRUE;           !   Push character from table entry

    If ReadCAM_Q EQL 1 then
        Begin                       !   Haven't reached start of string yet
        StackData     = CAMInData< 7:0>;
        CAMOutAddr_D = CAMInData<19:8>;

        If CAMOutAddr_D<11:8> EQL 0 then    !   Just a one character string
            ReadCAM_D       = FALSE !   Don't need to read it from DCAM
        Else
            ReadCAM_D     = TRUE;   !   Get the appropriate table entry.

        FlipStack     = FALSE;      !   Don't reverse string yet
        WriteCAM_D    = FALSE;
        OldCode_D     = OldCode_Q;  !   Maintain register contents
        NextCode_D    = NextCode_Q;
        State_D       = 1;          !   Remain in this state
        End     !   If ReadCAM

    Else                            !   Found the head of the string
        Begin
        StackData     = CAMOutAddr_Q< 7:0>;
        ReadCAM_D     = FALSE;      !   Stop tracing links
```

Figure 38 (cont.) - Decompress_2.BDS (cont. on next page)

```
            FlipStack     = TRUE;          !   Can start outputting string now

            FinChar_D     = CAMOutAddr_Q< 7:0>;  !   Record last char in string
            OldCode_D     = InCode_Q;      !   Remember last code

            If NextCode_Q NEQ fff#16 then   !   Table is not full yet
                Begin
                WriteCAM_D    = TRUE;       !   Add new entry
                CAMOutData_D = OldCode_Q & CAMOutAddr_Q< 7:0>;
                CAMOutAddr_D = NextCode_Q;
                NextCode_D    = NextCode_Q + 1;
                End
            Else
                Begin
                WriteCAM_D    = FALSE;      !   No room for new entry
                NextCode_D    = NextCode_Q; !   Maintain register contents
                End;

            State_D       = 0;             !   Done with loop - clear outta here
            End;    !   Else

        ReadCodeFIFO = FALSE;               !   Turn off unneeded flags
        InCode_D     = InCode_Q;            !   Maintain register contents
        End;        !   Else if State == 1

ENDROUTINE Decompress;


ENDMODEL Decompressor;
```

Figure 38 (cont.) - Decompress_2.BDS

```
!=============================================================================
!
!   File:    decompress_2.bdnet   (/user1/icsg8038/thesis/decompress_2.bdnet)
!   Author:  Bob Wall
!   Date:    6/30/91
!
!   Description:
!       BDNET description of LZW decompressor core (no FIFO, CAM, or stack).
!       This file will place the flip-flops needed to hold register and flag
!       values in decompressor_2, and will connect up SUPPLY and GROUND
!       nodes.
!
!   Notes:
!       The dfnf311 instanced is a negative edge-triggered D flip-flop with
!       Q and Q_bar outputs.
!
!       The Comparator function generated here is just a 12-bit XOR of the
!       NextCode_Q and InCodeFIFO signals.  It is used internally in the
!       decompressor to check if the two signals are equal.  The xorf201
!       is a simple two-input xor gate.
!
!   Revision History:
!      $Header: /n/dali/u1/icsg8038/thesis/decompress/RCS/decompress_2.bdnet,v 1.2 1991/07/27
00:46:39 icsg8038 Exp $
!       $Log: decompress_2.bdnet,v $
!       Revision 1.2  1991/07/27  00:46:39  icsg8038
!       Update from version 2.0 to version 2.2 of standard cell library
!
!       Revision 1.1  1991/07/16  00:41:34  icsg8038
!       Initial revision
!
!=============================================================================

MODEL       decompress_2:unplaced;
TECHNOLOGY  scmos;
VIEWTYPE    SYMBOLIC;
EDITSTYLE   SYMBOLIC;

OUTPUT
    StackData<7:0>,                 !   Output to stack data bus
    CAMOutAddr<11:0> : CAMOutAddr_Q<11:0>,  !   Output to CAM address bus
    CAMOutData<19:0> : CAMOutData_Q<19:0>,  !   Output to CAM data bus
    WriteCAM     : WriteCAM_Q<0>,   !   Latch data into CAM
    ReadCAM      : ReadCAM_Q<0>,    !   Read CAM as a RAM
    ReadCodeFIFO : ReadCodeFIFO<0>, !   Latched input code - get new one
    WriteStack   : WriteStack<0>,   !   Push next char. on stack
    FlipStack    : FlipStack<0>;    !   Switch stacks, start reversing string

INPUT
    Reset        : Reset<0>,        !   Reinit vars, start compression
    InCodeFIFO<11:0>,               !   Input from code FIFO output
    CAMInData<19:0>;                !   Input from CAM data bus

CLOCK
    CLK;                            !   System clock

SUPPLY      Vdd;
GROUND      GND;

INSTANCE    decompress_2:logic PROMOTE;

!   Place flip-flops for latched flags first.
INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:WriteCAM_D<0>;
    CLK2:CLK;
    Q:WriteCAM_Q<0>;
    Q_b:UNCONNECTED;
```

Figure 39 - Decompress_2.bdnet (cont. on next page)

```
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:ReadCAM_D<0>;
    CLK2:CLK;
    Q:ReadCAM_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

!   Now place flip-flops to form all the registers.

INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:State_D<0>;
    CLK2:CLK;
    Q:State_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   Next code symbol to generate
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:NextCode_D<%I>;
        CLK2:CLK;
        Q:NextCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   Previous input code
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:OldCode_D<%I>;
        CLK2:CLK;
        Q:OldCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 11 OF            !   New input code
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:InCode_D<%I>;
        CLK2:CLK;
        Q:InCode_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF             !   Final character of last string
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:FinChar_D<%I>;
        CLK2:CLK;
        Q:FinChar_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

!   Place flip-flops to latch the CAM busses.

ARRAY %I FROM 0 TO 19 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CAMOutData_D<%I>;
        CLK2:CLK;
        Q:CAMOutData_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;
```

Figure 39 (cont.) - Decompress_2.bdnet (cont. on next page)

```
ARRAY %I FROM 0 TO 11 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CAMOutAddr_D<%I>;
        CLK2:CLK;
        Q:CAMOutAddr_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

!   Generate 12-bit XOR for Comparator.

ARRAY %I FROM 0 TO 11 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/xorf201":physical
        A1:NextCode_Q<%I>;
        B1:InCodeFIFO<%I>;
        O:Comparator<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

ENDMODEL;
```

Figure 39 (cont.) - Decompress_2.bdnet

```
!=============================================================================
!
!   File:    stack.rtl    (/user1/icsg8038/thesis/stack.rtl)
!   Author:  Bob Wall
!   Date:    7/10/91
!
!   Description:
!       Simplified RTL description of the LZW string reversal mechanism.
!       This is essentially a boiled-down version of stack.bds, and will
!       hopefully be a little easier to read.
!
!       Here's the scoop on the string reversal technique - a ring buffer of
!       characters large enough to hold two maximum length strings is kept
!       (RStack).  In addition to this ring buffer, which is treated somewhat
!       like a dequeue, another pair of ring buffers are used as a regular
!       queue to hold pointers to the beginning and one past the end of each
!       string (StringHead, StringTail).  StringQHead and StringQTail are the
!       insertion and deletion pointers for these queues (both use the same
!       head and tail).  There are enough entries to hold MAX_STR_LEN strings.
!
!       The string currently being constructed is pointed to by CurrentInStr-
!       Head and CurrentInStrTail.  When the end of the string is encountered,
!       those head and tail values are added to the StringHead and StringTail
!       queues, the current tail is moved up to the previous head, and a new
!       string is begun.
!
!       The string currently being reversed is pointed to be CurrentOutStrHead
!       and CurrentOutStrTail.  If they point to a valid string, the Current-
!       OutStrUsed flag will be set to TRUE.  Each clock cycle, CurrentOutStr-
!       Head is moved back one character, and the character it points to is
!       output.  When the string is fully reversed (CurrentOutStrHead has been
!       moved back to CurrentOutStrTail), if there is a new string ready from
!       CurrentInStr, it is moved into CurrentOutStr; otherwise, if there is
!       a string in the StringQ (StringQHead != StringQTail), it is moved into
!       CurrentOutStr; otherwise, CurrentOutStrUsed is set to FALSE.
!
!       The head pointer of a string always points to the next available place
!       to add a character, and the tail pointer points to the first character
!       in the string.  Thus, when reversing the string, the head pointer must
!       be decremented before a character can be fetched.
!
!       When the decompressor signals a write, the character is latched into
!       NewChar, and the AddChar and EndOfString flags are set as appropriate.
!       The character is then processed on the following clock cycle.
!
!   Notes:
!       The '.' between two expressions represents concatenation.
!       '<-' represents a clocked (synchronous) assignment, while '<=' is
!       a combinational (asynchronous) assignment.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/stack/RCS/stack.rtl,v 1.1 91/07/16
00:18:52 icsg8038 Exp $
!       $Log:   stack.rtl,v $
!       Revision 1.1  91/07/16  00:18:52  icsg8038
!       Initial revision
!
!=============================================================================

!-----------------------------------------------------------------------------
!   Variable declarations
INPUT       Reset
            WriteStack
            FlipStack
            StackData <7:0>
            OutCharFIFOFull
            CLK
```

Figure 40 - Stack.RTL (cont. on next page)

```
OUTPUT      WriteCharFIFO
            OutCharFIFO <7:0>
            StackOverRun
            StackEmpty

REGISTER    CurrentInStrHead  <7:0>
            CurrentInStrTail  <7:0>
            CurrentOutStrHead <7:0>
            CurrentOutStrTail <7:0>
            CurrentOutStrUsed
            StringQHead <6:0>
            StringQTail <6:0>
            NewChar <7:0>
            AddChar
            EndOfString

RAM         RStack [256] <7:0>
            ReadRStack
            WriteRStack
            RStackData <7:0>
            RStackAddr <7:0>

            StringQ [128] [2] <7:0>
            ReadStringQ
            WriteStringQ
            StringQAddr <6:0>
            StrHeadData <7:0>
            StrTailData <7:0>

VARIABLE    TempInStrHead  <7:0>
            TempOutStrHead <7:0>
            NeedOutStr


!-------------------------------------------------------------------------
!   Begin Stack


!   Check to see if there is anything in the stack - if there is, either
!   a current input or current output string, or both, will be in progress.

If !CurrentOutStrUsed && CurrentInStrHead == CurrentInStrTail
    StackEmpty      <=  TRUE
Else
    StackEmpty      <=  FALSE

!   Precalculate the new values of the input and output string head pointers.

TempInStrHead   <=  CurrentInStrHead + 1

If CurrentOutStrUsed
    TempOutStrHead  <=  CurrentOutStrHead + 1
Else
    TempOutStrHead  <=  CurrentOutStrHead

If Reset                                !   Initialization stage
    StringQHead         <-  0           !   Set the string queue to empty,
    StringQTail         <-  0
    CurrentInStrHead    <-  0           !   and the current string to add
    CurrentInStrTail    <-  0
    CurrentOutStrUsed   <-  FALSE       !   Nothing in current remove string
    WriteCharFIFO       <=  FALSE       !   Turn off all the control signals
    StackOverRun        <=  FALSE
    ReadRStack          <=  FALSE
    WriteRStack         <=  FALSE
    ReadStringQ         <=  FALSE
    WriteStringQ        <=  FALSE
```

Figure 40 (cont.) - Stack.RTL (cont. on next page)

```
    !   Check for a character coming into the stack - if one is there, latch it
    !   and record whether it is the last character in the string.
    If WriteStack
        NewChar         <-  StackData
        AddChar         <-  TRUE
        If (FlipStack)
            EndOfString     <-  TRUE
        Else
            EndOfString     <-  FALSE
    Else
        AddChar         <-  FALSE

Else
    !   First, attempt to add a new character to the stack, if necessary.
    !   This is not possible if the stack is full.

    If (CurrentOutStrUsed && CurrentInStrHead == CurrentOutStrTail && AddChar)
        !   If a string is being removed, and the next place to insert into the
        !   string space is the same as the end of the oldest string (InStrHead
        !   == OutStrTail), or there is no more room in the string queue, sig-
        !   nal an overrun condition and just maintain the appropriate register
        !   contents.

        StackOverRun        <=  TRUE

        StringQHead         <-  StringQHead
        CurrentInStrHead    <-  CurrentInStrHead
        CurrentInStrTail    <-  CurrentInStrTail
        NewChar             <-  NewChar
        AddChar             <-  AddChar
        EndOfString         <-  EndOfString

        WriteRStack         <=  FALSE
        WriteStringQ        <=  FALSE

    Else                                !   Add chars to stack, if needed
        StackOverRun        <=  FALSE

        If WriteStack
            NewChar         <-  StackData
            AddChar         <-  TRUE
            If (FlipStack)
                EndOfString     <-  TRUE
            Else
                EndOfString     <-  FALSE
        Else
            AddChar         <-  FALSE

        If AddChar
            If !CLK         !   RStack [CurrentInStrHead] = NewChar
                WriteRStack     <=  TRUE
                RStackData      <=  NewChar
                RStackAddr      <=  CurrentInStrHead
            Else
                WriteRStack     <=  FALSE

            CurrentInStrHead    <-  TempInStrHead

            If EndOfString              !   Move current str into str queue
                CurrentInStrTail    <-  TempInStrHead

                If NeedOutStr           !   OutStr empty - refill it
                    CurrentOutStrHead   <-  TempInStrHead
                    CurrentOutStrTail   <-  CurrentInStrTail
                    CurrentOutStrUsed   <-  TRUE
                    StringQHead         <-  StringQHead
                    WriteStringQ        <=  FALSE
```

Figure 40 (cont.) - Stack.RTL (cont. on next page)

```
            Else
                If !CLK     !   StringHead [StringQHead] = TempInStrHead
                    WriteStringQ    <=  TRUE
                    StringQAddr     <=  StringQHead
                    StrHeadData     <=  TempInStrHead
                            !   StringTail [StringQHead] = CurrentInStrTail
                    StrTailData     <=  CurrentInStrTail
                Else
                    WriteStringQ    <=  FALSE

                StringQHead     <-  StringQHead + 1 !   Truncate to 7 bits

        Else
            StringQHead         <-  StringQHead
            CurrentInStrTail    <-  CurrentInStrTail

            If NeedOutStr
                CurrentOutStrUsed <-  FALSE

            WriteStringQ        <=  FALSE

    Else
        StringQHead         <-  StringQHead
        CurrentInStrHead    <-  CurrentInStrHead
        CurrentInStrTail    <-  CurrentInStrTail

        If NeedOutStr
            CurrentOutStrUsed   <-  FALSE

        WriteRStack         <=  FALSE
        WriteStringQ        <=  FALSE

!   Now try to write a character to the output FIFO, if there is one to
!   write and room in the FIFO.
If CurrentOutStrUsed && !OutCharFIFOFull
    If CLK          !   OutCharFIFO  =  Stack [TempOutStrHead]
        ReadRStack      <=  TRUE
        RStackAddr      <=  TempOutStrHead
        OutCharFIFO     <=  RStackData
    Else
        ReadRStack      <=  FALSE

    WriteCharFIFO   <=  TRUE

    If (TempOutStrHead == CurrentOutStrTail)    !   Wrote all chars in str
        If (StringQHead == StringQTail) !   No string available to reverse
            ReadStringQ     <=  FALSE
            NeedOutStr      <=  TRUE
            StringQTail     <-  StringQTail

        Else
            NeedOutStr      <=  FALSE

            If CLK
                !   CurrentOutStrHead  =  StringHead [StringQTail]
                ReadStringQ     <=  TRUE
                StringQAddr     <=  StringQTail
                CurrentOutStrHead   <=  StrHeadData

                !   CurrentOutStrTail  =  StringTail [StringQTail]
                CurrentOutStrTail   <=  StrTailData

            Else
                ReadStringQ     <=  FALSE

            StringQTail     <-  StringQTail + 1
            CurrentOutStrUsed   <-  TRUE
```

Figure 40 (cont.) - Stack.RTL (cont. on next page)

```
    Else                                    ! Still chars left in output str
        NeedOutStr          <=  FALSE
        ReadStringQ         <=  FALSE
        CurrentOutStrUsed   <-  TRUE
        CurrentOutStrHead   <-  TempOutStrHead
        CurrentOutStrTail   <-  CurrentOutStrTail
        StringQTail         <-  StringQTail

Else
    If CurrentOutStrUsed
        NeedOutStr          <=  FALSE
        CurrentOutStrUsed   <-  TRUE
        CurrentOutStrHead   <-  CurrentOutStrHead
        CurrentOutStrTail   <-  CurrentOutStrTail
    Else
        NeedOutStr          <=  TRUE

    ReadRStack          <=  FALSE
    ReadStringQ         <=  FALSE
    WriteCharFIFO       <=  FALSE

    StringQTail         <-  StringQTail
```

Figure 40 (cont.) - Stack.RTL

```
!=============================================================================
!
!   File:    stack.bds    (/user1/icsg8038/thesis/stack.bds)
!   Author:  Bob Wall
!   Date:    7/10/91
!
!   Description:
!       BDS description of string reversal mechanism for LZW compressor.
!       Details of algorithm in the Stack routine below.
!
!   Notes:
!       The '.' between two expressions represents concatenation.
!       '<-' represents a clocked (synchronous) assignment, while '<=' is
!       a combinational (asynchronous) assignment.
!
!       Assumes the availability of three RAMs, one 256 x 8 bits and the
!       other two 128 x 8 bits.  They are assumed to have an address buss,
!       separate read and write signals, and a bidirectional data buss.
!
!       Since BDS does not support memory (i.e. flip-flops), all the regi-
!       sters and flags are set up as separate output and input signals,
!       which can be connected to D flip-flops using BDNET.
!
!       Note that some registers are "refreshed" in certain states.  A
!       register must be assigned some value for every state during every
!       clock cycle, unless it is acceptable for it to be filled with
!       possible garbage from the input logic.  If it does not get assigned
!       a new value, it should be reassigned its output to maintain that
!       value.
!
!       The Comparator inputs used below are so misII doesn't choke generating
!       the various 8-bit == operations.  They are wired up in stack.bdnet as
!       follows:
!           Comparator1 == 0  <-->  CurrentInStrHead_Q == CurrentInStrTail_Q
!           Comparator2 == 0  <-->  CurrentInStrHead_Q == CurrentOutStrTail_Q
!           Comparator3 == 0  <-->  TempOutStrHead     == CurrentOutStrTail_Q
!           Comparator4 == 0  <-->  StringQHead_Q      == StringQTail_Q
!           Comparator5 == 0  <-->  StringQHead_D      == StringQTail_D
!
!   Revision History:
!        $Header: /n/dali/u1/icsg8038/thesis/stack/RCS/stack.bds,v 1.1 1991/07/16 00:18:26
icsg8038 Exp $
!       $Log: stack.bds,v $
!       Revision 1.1  1991/07/16  00:18:26  icsg8038
!       Initial revision
!
!=============================================================================

MODEL StringReversal

!   Circuit outputs and connections to all register and flag D inputs.
    CurrentInStrHead_D  <7:0>,          !   Registers
    CurrentInStrTail_D  <7:0>,
    CurrentOutStrHead_D <7:0>,
    CurrentOutStrTail_D <7:0>,
    CurrentOutStrUsed_D,
    StringQHead_D       <6:0>,
    StringQTail_D       <6:0>,
    StringQUsed_D,
    NewChar_D           <7:0>,
    AddChar_D,
    EndOfString_D,

!   RAM control signals.  There are three RAMs - RStack, StrHead, and
!   StrTail.  The last two are always addressed identically and read/
!   written simultaneously, so they share the StringQ read, write, and
!   address signals.
```

Figure 41 - Stack.BDS (cont. on next page)

```
    ReadRStack,
    WriteRStack,
    ReadStringQ,
    WriteStringQ,
    RStackReadAddr      <7:0>,          !   RAM address busses
    RStackWriteAddr     <7:0>,
    StringQReadAddr     <6:0>,
    StringQWriteAddr    <6:0>,
    RStackDataOut       <7:0>,          !   RAM data busses
    StrHeadDataOut      <7:0>,
    StrTailDataOut      <7:0>;
    StackOverRun,                       !   Control outputs
    StackEmpty,
    WriteCharFIFO,
    OutCharFIFO         <7:0>,          !   Data to output FIFO
    TempOutStrHead      <7:0>
=
!   Circuit inputs and connections to all register and flag Q outputs.
    CurrentInStrHead_Q  <7:0>,          !   Registers
    CurrentInStrTail_Q  <7:0>,
    CurrentOutStrHead_Q <7:0>,
    CurrentOutStrTail_Q <7:0>,
    CurrentOutStrUsed_Q,
    StringQHead_Q       <6:0>,
    StringQTail_Q       <6:0>,
    StringQUsed_Q,
    NewChar_Q           <7:0>,
    AddChar_Q,
    EndOfString_Q,
    RStackDataIn        <7:0>,          !   RAM data busses
    StrHeadDataIn       <7:0>,
    StrTailDataIn       <7:0>,
    Reset,                              !   Circuit inputs
    WriteStack,
    FlipStack,
    StackData           <7:0>,
    OutCharFIFOFull,
    Comparator1         <7:0>,          !   CurInStrHead   == CurInStrTail
    Comparator2         <7:0>,          !   CurInStrHead   == CurOutStrTail
    Comparator3         <7:0>,          !   TempOutStrHead == CurOutStrTail
    Comparator4         <6:0>,          !   StringQHead    == StringQTail
    Comparator5         <6:0>;          !   StringQHead_D  == StringQTail_D

STATE
    TempInStrHead       <7:0>,          !   Temporary variables
    NeedOutStr,
    StringAdded,
    StringRemoved;

CONSTANT
    TRUE  = 1,
    FALSE = 0;


!=============================================================================
!
!   String reversal routine:
!
!       Here's the scoop on the string reversal technique - a ring buffer of
!       characters large enough to hold two maximum length strings is kept
!       (RStack).  In addition to this ring buffer, which is treated somewhat
!       like a dequeue, another pair of ring buffers are used as a regular
!       queue to hold pointers to the beginning and one past the end of each
!       string (StringHead, StringTail).  StringQHead and StringQTail are the
!       insertion and deletion pointers for these queues (both use the same
!       head and tail).  There are enough entries to hold MAX_STR_LEN strings.
!
```

Figure 41 (cont.) - Stack.BDS (cont. on next page)

```
!       The string currently being constructed is pointed to by CurrentInStr-
!       Head and CurrentInStrTail.  When the end of the string is encountered,
!       those head and tail values are added to the StringHead and StringTail
!       queues, the current tail is moved up to the previous head, and a new
!       string is begun.
!
!       The string currently being reversed is pointed to be CurrentOutStrHead
!       and CurrentOutStrTail.  If they point to a valid string, the Current-
!       OutStrUsed flag will be set to TRUE.  Each clock cycle, CurrentOutStr-
!       Head is moved back one character, and the character it points to is
!       output.  When the string is fully reversed (CurrentOutStrHead has been
!       moved back to CurrentOutStrTail), if there is a new string ready from
!       CurrentInStr, it is moved into CurrentOutStr; otherwise, if there is
!       a string in the StringQ (StringQHead != StringQTail), it is moved into
!       CurrentOutStr; otherwise, CurrentOutStrUsed is set to FALSE.
!
!       The head pointer of a string always points to the next available place
!       to add a character, and the tail pointer points to the first character
!       in the string.  Thus, when reversing the string, the head pointer must
!       be decremented before a character can be fetched.
!
!       When the decompressor signals a write, the character is latched into
!       NewChar, and the AddChar and EndOfString flags are set as appropriate.
!       The character is then processed on the following clock cycle.
!
!
!   Calls:  None
!
!=============================================================================

ROUTINE  Stack;

    !   Check to see if there is anything in the stack - if there is, either
    !   a current input or current output string, or both, will be in progress,
    !   and the stack is not empty.

    If (CurrentOutStrUsed_Q EQL FALSE AND Comparator1 EQL 0) then
!               CurrentInStrHead_Q EQL CurrentInStrTail_Q) then
        StackEmpty = TRUE
    Else
        StackEmpty = FALSE;

    !   Precalculate new values of input and output string head pointers.

    TempInStrHead  =  CurrentInStrHead_Q + 1;

    If CurrentOutStrUsed_Q EQL TRUE then
        TempOutStrHead  =  CurrentOutStrHead_Q - 1
    Else
        TempOutStrHead  =  CurrentOutStrHead_Q;

    !   On Reset, just set all the necessary variables to their initial
    !   values.

    If Reset then
        Begin
        StringQHead_D       =  0;          !   Set the string queue and the
        StringQTail_D       =  0;
        CurrentInStrHead_D  =  0;          !   current input string to empty
        CurrentInStrTail_D  =  0;
        CurrentOutStrUsed_D =  FALSE;      !   Nothing in current out string
        StringQUsed_D       =  FALSE;      !   of string queue
        WriteCharFIFO       =  FALSE;      !   Turn off all control signals
        StackOverRun        =  FALSE;
        ReadRStack          =  FALSE;
        WriteRStack         =  FALSE;
        ReadStringQ         =  FALSE;
        WriteStringQ        =  FALSE;
```

Figure 41 (cont.) - Stack.BDS (cont. on next page)

```
    !   Check for a character coming into the stack - if one is there,
    !   latch it and record whether it is the last character in the string.

    If WriteStack EQL TRUE then
        Begin
        NewChar_D       =  StackData;
        AddChar_D       =  TRUE;
        If FlipStack EQL TRUE then
            EndOfString_D   =  TRUE
        Else
            EndOfString_D   =  FALSE;
        End

    Else
        AddChar_D       =  FALSE;

    End !   If Reset

Else
    Begin

    !-------------------------------------------------------------------
    !   First, try to write a character to the output FIFO, if there is one
    !   to write and room in the FIFO.  This has to come before the logic
    !   to process the incoming characters, because it generates the flag
    !   NeedOutStr, and BDSYN needs to see the statements that assign
    !   the value before the statements that use it.

    If CurrentOutStrUsed_Q EQL TRUE AND OutCharFIFOFull EQL FALSE then
        Begin
        !   OutCharFIFO =  Stack [TempOutStrHead]
        ReadRStack      =  TRUE;
        RStackReadAddr  =  TempOutStrHead;
        OutCharFIFO     =  RStackDataIn;

        WriteCharFIFO   =  TRUE;

        !   If all the characters in the current output string have been
        !   or will be written, try to load another pair of pointers from
        !   either the input string or the string queue.
        !
        !   If (TempOutStrHead == CurrentOutStrTail)

        If Comparator3 EQL 0 then
            If StringQUsed_Q EQL 0 then
                Begin                   !   No string available to reverse
                ReadStringQ     =  FALSE;
                NeedOutStr      =  TRUE;
                StringQTail_D   =  StringQTail_Q;
                StringRemoved   =  FALSE;
                End

            Else
                Begin
                NeedOutStr      =  FALSE;

                !   CurrentOutStrHead  =  StringHead [StringQTail]
                ReadStringQ     =  TRUE;
                StringQReadAddr =  StringQTail_Q;
                CurrentOutStrHead_D =  StrHeadDataIn;

                !   CurrentOutStrTail  =  StringTail [StringQTail]
                CurrentOutStrTail_D =  StrTailDataIn;
                CurrentOutStrUsed_D =  TRUE;
                StringQTail_D       =  StringQTail_Q + 1;
                StringRemoved       =  TRUE;

                End     !   String available to reverse
```

Figure 41 (cont.) - Stack.BDS (cont. on next page)

```
    Else
        Begin
        NeedOutStr      =  FALSE;
        ReadStringQ     =  FALSE;
        CurrentOutStrUsed_D =  TRUE;
        CurrentOutStrHead_D =  TempOutStrHead;
        CurrentOutStrTail_D =  CurrentOutStrTail_Q;
        StringQTail_D       =  StringQTail_Q;
        StringRemoved       =  FALSE;

        End;    !   CurrentOutString still had something in it

    End     !   Chars in OutString to reverse and output FIFO ready

Else
    Begin
    If CurrentOutStrUsed_Q EQL TRUE then
        Begin
        NeedOutStr          =  FALSE;
        CurrentOutStrUsed_D =  TRUE;
        CurrentOutStrHead_D =  CurrentOutStrHead_Q;
        CurrentOutStrTail_D =  CurrentOutStrTail_Q;
        End

    Else
        NeedOutStr      =  TRUE;

    ReadRStack      =  FALSE;
    ReadStringQ     =  FALSE;
    WriteCodeFIFO   =  FALSE;
    StringQTail_D   =  StringQTail_Q;
    StringRemoved   =  FALSE;

    End;    !   No OutString ready or output FIFO full

!-----------------------------------------------------------------------
!   Next, attempt to add a new character to the stack, if necessary.
!   This is not possible if the stack is full.
!
!   If a string is being output and the next place to insert into the
!   string space is the same as the end of the oldest string (InStrHead
!   == OutStrTail) and there is a character to add, or if the queue of
!   string pointers is already full and there is a string to add,
!   signal an overrun condition and just maintain the appropriate
!   register contents.  The condition is
!
!   If (CurrentOutStrUsed &&
!       ((CurrentInStrHead == CurrentOutStrTail && AddChar) ||
!        (StringQUsed && StringQHead == StringQTail && EndOfString)))
!

If (CurrentOutStrUsed_Q EQL TRUE AND
    ((Comparator2 EQL 0 AND AddChar_Q EQL TRUE) OR
     (StringQUsed_Q EQL TRUE AND Comparator4 EQL 0 AND
                EndOfString_Q EQL TRUE))) then
    Begin
    StackOverRun        =  TRUE;
    StringQHead_D       =  StringQHead_Q;
    StringAdded         =  FALSE;
    CurrentInStrHead_D  =  CurrentInStrHead_Q;
    CurrentInStrTail_D  =  CurrentInStrTail_Q;
    NewChar_D           =  NewChar_Q;
    AddChar_D           =  AddChar_Q;
    EndOfString_D       =  EndOfString_Q;
    WriteRStack         =  FALSE;
    WriteStringQ        =  FALSE;

    End     !   Stack full
```

Figure 41 (cont.) - Stack.BDS (cont. on next page)

```
!   Otherwise, check to see if there is a new character being written
!   and if so, latch it.  Meanwhile, if a character has been latched,
!   stuff it into the stack.

Else
    Begin
    StackOverRun        =  FALSE;

    If WriteStack EQL TRUE then
        Begin
        NewChar_D       =  StackData;
        AddChar_D       =  TRUE;

        If FlipStack EQL TRUE then
            EndOfString_D   =  TRUE
        Else
            EndOfString_D   =  FALSE;
        End

    Else
        AddChar_D       =  FALSE;

    If AddChar_Q EQL TRUE then
        Begin
        !   RStack [CurrentInStrHead] = NewChar
        WriteRStack     =  TRUE;
        RStackDataOut   =  NewChar_Q;
        RStackWriteAddr =  CurrentInStrHead_Q;

        CurrentInStrHead_D  =  TempInStrHead;

        If EndOfString_Q EQL TRUE then
            Begin               !   Move current str into str queue
            CurrentInStrTail_D  =  TempInStrHead;

            If NeedOutStr EQL TRUE then
                Begin           !   OutStr empty - refill it
                CurrentOutStrHead_D =  TempInStrHead;
                CurrentOutStrTail_D =  CurrentInStrTail_Q;
                CurrentOutStrUsed_D =  TRUE;

                StringQHead_D       =  StringQHead_Q;
                StringAdded         =  FALSE;
                WriteStringQ        =  FALSE;
                End

            Else
                Begin
                !   StringHead [StringQHead] = TempInStrHead
                WriteStringQ        =  TRUE;
                StrHeadDataOut      =  TempInStrHead;
                StringQWriteAddr    =  StringQHead_Q;

                !   StringTail [StringQHead] = CurrentInStrTail
                StrTailDataOut      =  CurrentInStrTail_Q;

                StringQHead_D   =  StringQHead_Q + 1;
                StringAdded     =  TRUE;

                End;    !   !NeedOutStr
            End     !   EndOfString

        Else
            Begin
            StringQHead_D       =  StringQHead_Q;
            StringAdded         =  FALSE;
            CurrentInStrTail_D  =  CurrentInStrTail_Q;
```

Figure 41 (cont.) - Stack.BDS (cont. on next page)

```
                    If NeedOutStr EQL TRUE then
                        CurrentOutStrUsed_D =  FALSE;
                        !   It doesn't matter what CurrentOutStrHead and
                        !   CurrentOutStrTail are assigned, since they
                        !   will be assigned new values when they become
                        !   available anyway.

                    WriteStringQ        =  FALSE;

                    End;    !   !EndOfString

                End     !   AddChar

            Else
                Begin
                StringQHead_D       =  StringQHead_Q;
                StringAdded         =  FALSE;

                CurrentInStrHead_D  =  CurrentInStrHead_Q;
                CurrentInStrTail_D  =  CurrentInStrTail_Q;

                WriteRStack         =  FALSE;
                WriteStringQ        =  FALSE;

                If NeedOutStr EQL TRUE then
                    CurrentOutStrUsed_D = FALSE;

                End;    !   !AddChar

            End;    !   Stack not full


        !---------------------------------------------------------------------
        !   Check to see what the status of StringQUsed should be.
        !

        If StringAdded EQL TRUE then
            StringQUsed_D = TRUE
        Else if StringRemoved EQL TRUE then
            If Comparator5 EQL 0 then       !   StringQHead_D == StringQTail_D
                StringQUsed_D = FALSE
            Else
                StringQUsed_D = TRUE
        Else
            StringQUsed_D = StringQUsed_Q;

        End;    !   Not Reset

ENDROUTINE Stack;


ENDMODEL StringReversal;
```

Figure 41 (cont.) - Stack.BDS

```
!=============================================================================
!
!   File:    stack.bdnet    (/user1/icsg8038/thesis/stack.bdnet)
!   Author:  Bob Wall
!   Date:    7/11/91
!
!   Description:
!       BDNET description of string reversal mechanism described in stack.bds.
!       This file will place the flip-flops needed to hold register and flag
!       values in the compressor, and will connect up SUPPLY and GROUND
!       nodes.
!
!   Notes:
!       The dfnf311 instanced is a negative edge-triggered D flip-flop with
!       Q and Q_bar outputs.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/stack/RCS/stack.bdnet,v 1.1 1991/07/16
00:18:08 icsg8038 Exp $
!       $Log: stack.bdnet,v $
!       Revision 1.1  1991/07/16  00:18:08  icsg8038
!       Initial revision
!
!=============================================================================

MODEL       stack:unplaced;
TECHNOLOGY  scmos;
VIEWTYPE    SYMBOLIC;
EDITSTYLE   SYMBOLIC;

OUTPUT
    StackOverRun    :   StackOverRun<0>,
    StackEmpty      :   StackEmpty<0>,
    WriteCharFIFO   :   WriteCharFIFO<0>,
    OutCharFIFO<7:0>,
    ReadRStack      :   ReadRStack<0>,  !   RAM control signals
    WriteRStack     :   WriteRStack<0>,
    RStackReadAddr<7:0>,
    RStackWriteAddr<7:0>,
    RStackDataOut<7:0>,
    ReadStringQ     :   ReadStringQ<0>,
    WriteStringQ    :   WriteStringQ<0>,
    StringQReadAddr<6:0>,
    StringQWriteAddr<6:0>,
    StrHeadDataOut<7:0>,
    StrTailDataOut<7:0>;

INPUT
    Reset           :   Reset<0>,
    WriteStack      :   WriteStack<0>,
    FlipStack       :   FlipStack<0>,
    StackData<7:0>,
    OutCharFIFOFull :   OutCharFIFOFull<0>,
    RStackDataIn<7:0>,
    StrHeadDataIn<7:0>,
    StrTailDataIn<7:0>;

CLOCK
    CLK;                        !   System clock
SUPPLY      Vdd;
GROUND      GND;

INSTANCE    stack:logic PROMOTE;

!   Place flip-flops for flags first.
INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:CurrentOutStrUsed_D<0>;
```

Figure 42 - Stack.bdnet (cont. on next page)

```
    CLK2:CLK;
    Q:CurrentOutStrUsed_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:StringQUsed_D<0>;
    CLK2:CLK;
    Q:StringQUsed_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:AddChar_D<0>;
    CLK2:CLK;
    Q:AddChar_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:EndOfString_D<0>;
    CLK2:CLK;
    Q:EndOfString_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;


!   Now place flip-flops to form all the registers.

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CurrentInStrHead_D<%I>;
        CLK2:CLK;
        Q:CurrentInStrHead_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CurrentInStrTail_D<%I>;
        CLK2:CLK;
        Q:CurrentInStrTail_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CurrentOutStrHead_D<%I>;
        CLK2:CLK;
        Q:CurrentOutStrHead_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:CurrentOutStrTail_D<%I>;
        CLK2:CLK;
        Q:CurrentOutStrTail_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;
```

Figure 42 (cont.) - Stack.bdnet (cont. on next page)

```
ARRAY %I FROM 0 TO 6 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:StringQHead_D<%I>;
        CLK2:CLK;
        Q:StringQHead_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 6 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:StringQTail_D<%I>;
        CLK2:CLK;
        Q:StringQTail_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:NewChar_D<%I>;
        CLK2:CLK;
        Q:NewChar_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

!   Put in XOR gates for the comparators.
ARRAY %I FROM 0 TO 7 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/xorf201":physical
        A1:CurrentInStrHead_Q<%I>;
        B1:CurrentInStrTail_Q<%I>;
        O:Comparator1<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/xorf201":physical
        A1:CurrentInStrHead_Q<%I>;
        B1:CurrentOutStrTail_Q<%I>;
        O:Comparator2<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/xorf201":physical
        A1:TempOutStrHead<%I>;
        B1:CurrentOutStrTail_Q<%I>;
        O:Comparator3<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 6 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/xorf201":physical
        A1:StringQHead_Q<%I>;
        B1:StringQTail_Q<%I>;
        O:Comparator4<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 6 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_2/xorf201":physical
        A1:StringQHead_D<%I>;
        B1:StringQTail_D<%I>;
        O:Comparator5<%I>;
        "Vdd!":Vdd;
        "GND!":GND;
ENDMODEL;
```

Figure 42 (cont.) - Stack.bdnet

```
!============================================================================
!
!   File:    stackram.bdnet    (/user1/icsg8038/thesis/stackram.bdnet)
!   Author:  Bob Wall
!   Date:    7/13/91
!
!   Description:
!       String reversal mechanism, with addition of RAMs for the stack and
!       string queues.
!
!   Notes:
!       The dfnf311 instanced is a negative edge-triggered D flip-flop with
!       Q and Q_bar outputs.
!
!   Revision History:
!       $Header$
!       $Log$
!
!============================================================================

MODEL       stackram:logic;
TECHNOLOGY  scmos;
VIEWTYPE    SYMBOLIC;
EDITSTYLE   SYMBOLIC;

OUTPUT
    StackOverRun,
    StackEmpty,
    WriteCharFIFO,
    OutCharFIFO<7:0>;

INPUT
    Reset,
    WriteStack,
    FlipStack,
    StackData<7:0>,
    OutCharFIFOFull;

CLOCK
    CLK;                        !   System clock

SUPPLY      Vdd;
GROUND      GND;

INSTANCE    stack:flat PROMOTE;

!============================================================================
!   In order to simulate a dual-ported RAM, we will play a few games with the
!   clock to enable writes on the first half clock cycle (when CLK is low)
!   and reads on the second half clock cycle (when CLK is high).  I know it is
!   bad to gate the clock, but this is just to simulate a dual-ported RAM.
!
!----------------------------------------------------------------------------
!   Generate the gated Read and Write signals.

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/nanf201":physical
    A1:WriteRStack;
    B1:WriteEnable;
    O:temp1;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/invf101":physical
    A1:temp1;
    O:writerstack;
    "Vdd!":Vdd;
    "GND!":GND;
```

Figure 43 - Stackram.bdnet (cont. on next page)

```
INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/nanf201":physical
    A1:WriteStringQ;
    B1:WriteEnable;
    O:temp2;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/invf101":physical
    A1:temp2;
    O:writestringq;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/nanf201":physical
    A1:ReadRStack;
    B1:ReadEnable;
    O:temp3;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/invf101":physical
    A1:temp3;
    O:readrstack;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/nanf201":physical
    A1:ReadStringQ;
    B1:ReadEnable;
    O:temp4;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/invf101":physical
    A1:temp4;
    O:readstringq;
    "Vdd!":Vdd;
    "GND!":GND;

!-------------------------------------------------------------------------
!   Tristate the address busses, and enable them on opposite phases of the
!   clock.
ARRAY %I FROM 0 TO 7 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/muxf201":physical
        A1:RStackWriteAddr<%I>;
        B2:RStackReadAddr<%I>;
        SEL3:AddrSelect;
        O:rstackaddr<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 6 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/muxf201":physical
        A1:StringQWriteAddr<%I>;
        B2:StringQReadAddr<%I>;
        SEL3:AddrSelect;
        O:stringqaddr<%I>;
        "Vdd!":Vdd;
        "GND!":GND;

!-------------------------------------------------------------------------
!   Finally, connect all this mess up to the RAMs and hope they work correctly.
INSTANCE "RAM/ram256x8":logic          !   RStack
    read:readrstack;
    write:writerstack;
    address<7:0>:rstackaddr<7:0>;
    indata<7:0>:RStackDataOut<7:0>;
    outdata<7:0>:RStackDataIn<7:0>;
```

Figure 43 (cont.) - Stackram.bdnet (cont. on next page)

```
INSTANCE "RAM/ram128x8":logic          !   StringHead
    read:readstringq;
    write:writestringq;
    address<6:0>:stringqaddr<6:0>;
    indata<7:0>:StrHeadDataOut<7:0>;
    outdata<7:0>:StrHeadDataIn<7:0>;

INSTANCE "RAM/ram128x8":logic          !   StringTail
    read:readstringq;
    write:writestringq;
    address<6:0>:stringqaddr<6:0>;
    indata<7:0>:StrTailDataOut<7:0>;
    outdata<7:0>:StrTailDataIn<7:0>;

ENDMODEL;
```

Figure 43 (cont.) - Stackram.bdnet

```
!============================================================================
!
!   File:    stackram.musa      (/user1/icsg8038/thesis/stackram.musa)
!   Author:  Bob Wall
!   Date:    7/13/91
!
!   Description:
!       stack.musa, set up to use stackram:flat, which has the RAM added,
!       so it is not necessary to play with the RAM control signals.  Also,
!       all the internal signals are invisible, so only the external inter-
!       face can be manipulated.
!
!   Notes:
!
!   Revision History:
!       $Header$
!       $Log$
!
!============================================================================

!   The next vectors are for the input and output signals for the stack,
!   inputs first.
mv StackData            StackData<7:0>
mv OutCharFIFO          OutCharFIFO<7:0>
mv RSReadAddr           RStackReadAddr<7:0>
mv RSWriteAddr          RStackWriteAddr<7:0>
mv RSDataOut            RStackDataOut<7:0>
mv RSDataIn             RStackDataIn<7:0>
mv SQReadAddr           StringQReadAddr<6:0>
mv SQWriteAddr          StringQWriteAddr<6:0>
mv SQDataOut            StrHeadDataOut<7:0> StrTailDataOut<7:0>
mv SQDataIn             StrHeadDataIn<7:0> StrTailDataIn<7:0>

!   Macros - first one sets the clock on, then off to produce the negative
!       edge required to trigger the flip-flops in the registers and flags.

ma  clock
se AddrSelect 1
ev
se WriteEnable 1
ev
se WriteEnable 0
ev
se AddrSelect 0
ev
se ReadEnable 1
ev
se ReadEnable 0
ev
se CLK 1
ev
se CLK 0
ev
$end

ma ShowOutputs
ev
se AddrSelect 1
ev
se WriteEnable 1
ev
se WriteEnable 0
ev
se AddrSelect 0
ev
se ReadEnable 1
ev
```

Figure 44 - Stackram.musa (cont. on next page)

```
sh ReadRStack WriteRStack RSReadAddr RSWriteAddr RSDataOut RSDataIn
sh ReadStringQ WriteStringQ SQReadAddr SQWriteAddr
sh StackOverRun StackEmpty WriteCharFIFO OutCharFIFO
se ReadEnable 0
ev
se CLK 1
ev
se CLK 0
ev
$end

ma VerifyOut
ev
se AddrSelect 1
ev
se WriteEnable 1
ev
se WriteEnable 0
ev
se AddrSelect 0
ev
se ReadEnable 1
ev
se CLK 1
ev
ve WriteCharFIFO 1
ve OutCharFIFO $1
se ReadEnable 0
ev
se CLK 0
ev
$end

ma VerifyNoOut
ev
se AddrSelect 1
ev
se WriteEnable 1
ev
se WriteEnable 0
ev
se AddrSelect 0
ev
se ReadEnable 1
ev
se CLK 1
ev
ve WriteCharFIFO 0
se ReadEnable 0
ev
se CLK 0
ev
$end

!   Try a little quickie - reverse the string "abc".
se Reset 1
se WriteStack 1
se FlipStack 0
se StackData H61
se OutCharFIFOFull 0
ev
ShowOutputs

se Reset 0
se StackData H62
ev
ShowOutputs
```

Figure 44 (cont.) - Stackram.musa (cont. on next page)

```
se StackData H63
se FlipStack 1
ev
ShowOutputs

se WriteStack 0
ev
ShowOutputs

ShowOutputs

ShowOutputs

ShowOutputs

ShowOutputs

ShowOutputs

ShowOutputs

!   End of stackram.musa - all basic vectors and macros created.
!
```

Figure 44 (cont.) - Stackram.musa

```
!=============================================================================
!
!   File:    control.bds    (/user1/icsg8038/thesis/control.bds)
!   Author:  Bob Wall
!   Date:    7/15/91
!
!   Description:
!       BDS description of controller for the LZW compression engine.
!
!       Note that the controller inserts itself in the address buss between
!       the compressor / decompressor and the DCAM.  It does the same with the
!       ReadCAM and WriteCAM control signals.  This is to allow it to assert
!       the appropriate address and controls when it is time to do a refresh.
!       At all other times, it just passes the signals through.
!
!   Notes:
!       The '.' between two expressions represents concatenation.
!       '<-' represents a clocked (synchronous) assignment, while '<=' is
!       a combinational (asynchronous) assignment.
!
!       Since BDS does not support memory (i.e. flip-flops), all the regi-
!       sters and flags are set up as separate output and input signals,
!       which can be connected to D flip-flops using BDNET.
!
!       Note that some registers are "refreshed" in certain states.  A
!       register must be assigned some value for every state during every
!       clock cycle, unless it is acceptable for it to be filled with
!       possible garbage from the input logic.  If it does not get assigned
!       a new value, it should be reassigned its output to maintain that
!       value.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/periph/RCS/control.bds,v 1.1 91/07/15
19:48:15 icsg8038 Exp $
!       $Log:   control.bds,v $
!       Revision 1.1  91/07/15  19:48:15  icsg8038
!       Initial revision
!
!=============================================================================


MODEL Controller

!   Circuit outputs and connections to all register and flag D inputs.
    RefreshCt_D     < 7:0>,             !   Registers - make sure lengths are
    RefreshAddr_D   < 7:0>,             !     same as CNT_LEN and REF_LEN.
    Reset_D,                            !   Latched control outputs
    DoCompress_D,
    DumpOutCode_D,
    HaltOut,                            !   Unlatched control outputs
    DumpOmega,
    DoRefresh,
    OperationComplete,
    ReadCAM,                            !   DCAM control and address signals
    WriteCAM,
    CAMOutAddr      <11:0>

=

!   Circuit inputs and connections to all register and flag Q outputs.
    RefreshCt_Q     < 7:0>,             !   Registers
    RefreshAddr_Q   < 7:0>,
    Reset_Q,                            !   Latched control outputs
    DoCompress_Q,
    DumpOutCode_Q,
    ResetIn,                            !   Circuit inputs
    DoCompressIn,
```

Figure 45 - Control.BDS (cont. on next page)

```
    InCharFIFOEmpty,                    !   Flow control signals from I/O FIFOs
    InCharFIFOFull,
    InCharDataEnd,
    InCodeFIFOEmpty,
    InCodeFIFOFull,
    InCodeDataEnd,
    OutCodeFIFOEmpty,
    OutCodeFIFOFull,
    OutCharFIFOEmpty,
    StackFull,
    ReadCAMIn,                          !   DCAM control and address signals
    WriteCAMIn,                         !   from compressor/decompressor
    CAMOutAddrIn    <11:0>;

CONSTANT
    TRUE  = 1,
    FALSE = 0,

    !   Values for Galois-field sequencers.  REF_LEN is determined by the
    !   number of rows in the DCAM, and REF_EXT is enough zeroes to fill
    !   the number of bits out to a 12-bit DCAM address; CNT_LEN is determined
    !   by the system clock speed and how often each row needs to be refreshed.
    REF_LEN  = 8,
    REF_INIT = FF#16,
    REF_EXT  = 0#16,
    CNT_LEN  = 8,
    CNT_INIT = FE#16,
    CNT_DONE = FF#16;

!============================================================================
!
!   Routine to generate the next clock cycle count using a Galois-field based
!   sequencer.
!
!   Invocation: Count = NextCount (Count);
!
!   Input args: Count       CNT_LEN-bit current refresh address
!
!   Notes:
!       The logic for the finite-field sequencer is very dependent on the
!       number of bits in the field (an irreducible polynomial of the correct
!       length is needed), so if the maximum string length is changed, this
!       routine MUST be modified.
!
!       For an eight-bit sequencer, a suitable minimum-weight polynomial is
!       0x11d, which corresponds to X^8 + X^4 + X^3 + X^2 + 1.
!
!       Since 0 is a meta-stable state of the sequencer, the StrLen register
!       should never be allowed to by all 0's.  It would be simple enough to
!       add a check in here - if CurrentStrLen = H000 THEN Return 0x3ff.
!
!       For more information on the subject of Galois-field based sequencers,
!       along with a table of minimum-weight irreducible polynomials, see the
!       paper "Galois-Field Based State Assignment for PLA Controllers" by
!       Kel Winters.
!
!============================================================================

ROUTINE NextCount<7:0> (CurrentCount<7:0>);

    STATE   CountTemp<7:0>,
            I<>;
    CountTemp<0> = CurrentCount<7>;
    CountTemp<1> = CurrentCount<0>;
    CountTemp<2> = CurrentCount<1> XOR CurrentCount<7>;
    CountTemp<3> = CurrentCount<2> XOR CurrentCount<7>;
    CountTemp<4> = CurrentCount<3> XOR CurrentCount<7>;
```

Figure 45 (cont.) - Control.BDS (cont. on next page)

```
    for I FROM 5 TO 7 DO
        CountTemp<I> = CurrentCount<I - 1>;

    return CountTemp;

ENDROUTINE NextCount;


!=============================================================================
!
!   Routine to generate the next refresh address using a Galois-field based
!   sequencer.
!
!   Invocation: Refresh = NextRefresh (Refresh);
!
!   Input args: Refresh     REF_LEN-bit current refresh address
!
!   Notes:
!       The logic for the finite-field sequencer is very dependent on the
!       number of bits in the field (an irreducible polynomial of the correct
!       length is needed), so if the length of the refresh address is changed,
!       this routine MUST be modified.
!
!       For a ten-bit sequencer, a suitable minimum-weight polynomial is
!       0x409, which corresponds to X^10 + X^3 + 1.
!
!=============================================================================

ROUTINE NextRefresh<REF_LEN:0> (CurrentRefresh<REF_LEN:0>);

    STATE   RefTemp<REF_LEN:0>,
            I<>;

    RefTemp<0> = CurrentRefresh<7>;
    RefTemp<1> = CurrentRefresh<0>;
    RefTemp<2> = CurrentRefresh<1> XOR CurrentRefresh<7>;
    RefTemp<3> = CurrentRefresh<2> XOR CurrentRefresh<7>;
    RefTemp<4> = CurrentRefresh<3> XOR CurrentRefresh<7>;

    for I FROM 5 TO 7 DO
        RefTemp<I> = CurrentRefresh<I - 1>;

    return RefTemp;

ENDROUTINE NextRefresh;

!=============================================================================
!
!   Main LZW system controller.
!
!   Calls:  NextCount, NextRefresh
!
!=============================================================================

ROUTINE  Control;

    If ResetIn EQL TRUE then
        Begin
        Reset_D         = TRUE;
        DoCompress_D    = DoCompressIn;
        DumpOutCode_D   = FALSE;
        HaltOut         = FALSE;
        DumpOmega       = FALSE;
        ReadCAM         = FALSE;
        WriteCAM        = FALSE;
        OperationComplete = FALSE;
        DoRefresh       = FALSE;
```

Figure 45 (cont.) - Control.BDS (cont. on next page)

```
    RefreshCt_D     = CNT_INIT;
    RefreshAddr_D   = REF_INIT;
    End

Else
    Begin
    Reset_D         = FALSE;
    DoCompress_D    = DoCompress_Q;

    If RefreshCt_Q EQL CNT_DONE then
        Begin                       !   Have counted enough cycles -
        HaltOut     = TRUE;         !     do a refresh
        DumpOmega   = FALSE;
        DumpOutCode_D = DumpOutCode_Q;

        DoRefresh   = TRUE;
        CAMOutAddr  = RefreshAddr_Q & REF_EXT;

        ReadCAM     = FALSE;        !   Turn off other DCAM control signals
        WriteCAM    = FALSE;

        RefreshAddr_D = NextRefresh(RefreshAddr_Q);
        End

    Else
        Begin
        DoRefresh   = FALSE;

        CAMOutAddr  = CAMOutAddrIn; !   Allow DCAM control to pass through
        ReadCAM     = ReadCAMIn;
        WriteCAM    = WriteCAMIn;
        RefreshAddr_D = RefreshAddr_Q;

        If (InCharFIFOEmpty EQL TRUE OR InCodeFIFOEmpty EQL TRUE OR
            OutCodeFIFOFull EQL TRUE OR StackFull EQL TRUE) then
            Begin
            HaltOut = TRUE;         !   I/O wait required

            If (InCharFIFOEmpty EQL TRUE AND InCharDataEnd EQL TRUE AND
                DumpOutCode_Q EQL FALSE) then
                Begin               !   End of compressor input - tell it
                DumpOmega   = TRUE; !     to jettison its last code
                DumpOutCode_D = TRUE;
                End

            Else
                Begin
                DumpOmega   = FALSE;
                DumpOutCode_D = DumpOutCode_Q;
                End;

            End     !   I/O wait required

        Else
            Begin
            HaltOut     = FALSE;
            DumpOmega   = FALSE;
            DumpOutCode_D = DumpOutCode_Q;
            End;

        End;    !  No refresh necessary

    RefreshCt_D = NextCount(RefreshCt_Q);

    !   Check to see whether everything is wrapped up - if the input
    !   FIFO is processed and the corresponding output is all gone,
    !   we are finished.
```

Figure 45 (cont.) - Control.BDS (cont. on next page)

```
        If ((InCharFIFOEmpty EQL TRUE AND InCharDataEnd EQL TRUE AND
                        OutCodeFIFOEmpty EQL TRUE)  OR
            (InCodeFIFOEmpty EQL TRUE AND InCodeDataEnd EQL TRUE AND
                        OutCharFIFOEmpty EQL TRUE)) then
            OperationComplete = TRUE
        Else
            OperationComplete = FALSE;

        End;    !   Not ResetIn

ENDROUTINE Control;


ENDMODEL Controller;
```

Figure 45 (cont.) - Control.BDS

```
!============================================================================
!
!   File:    control.bdnet    (/user1/icsg8038/thesis/control.bdnet)
!   Author:  Bob Wall
!   Date:    7/11/91
!
!   Description:
!       BDNET description of LZW controller described in control.bds.
!       This file will place the flip-flops needed to hold register and flag
!       values in the compressor, and will connect up SUPPLY and GROUND
!       nodes.
!
!   Notes:
!       The dfnf311 instanced is a negative edge-triggered D flip-flop with
!       Q and Q_bar outputs.
!
!   Revision History:
!       $Header: /n/dali/u1/icsg8038/thesis/periph/RCS/control.bdnet,v 1.1 91/07/15
19:48:35 icsg8038 Exp $
!       $Log:   control.bdnet,v $
!       Revision 1.1  91/07/15  19:48:35  icsg8038
!       Initial revision
!
!============================================================================

MODEL       control:unplaced;
TECHNOLOGY  scmos;
VIEWTYPE    SYMBOLIC;
EDITSTYLE   SYMBOLIC;

OUTPUT
    Reset               :   Reset_Q<0>,
    DoCompress          :   DoCompress_Q<0>,
    DumpOutCode         :   DumpOutCode_Q<0>,
    HaltOut             :   HaltOut<0>,
    DumpOmega           :   DumpOmega<0>,
    DoRefresh           :   DoRefresh<0>,
    OperationComplete   :   OperationComplete<0>,
    ReadCAM             :   ReadCAM<0>,
    WriteCAM            :   WriteCAM<0>,
    CAMOutAddr  <11:0>;

INPUT
    ResetIn             :   ResetIn<0>,
    DoCompressIn        :   DoCompressIn<0>,
    InCharFIFOEmpty     :   InCharFIFOEmpty<0>,
    InCharFIFOFull      :   InCharFIFOFull<0>,
    InCharDataEnd       :   InCharDataEnd<0>,
    InCodeFIFOEmpty     :   InCodeFIFOEmpty<0>,
    InCodeFIFOFull      :   InCodeFIFOFull<0>,
    InCodeDataEnd       :   InCodeDataEnd<0>,
    OutCodeFIFOEmpty    :   OutCodeFIFOEmpty<0>,
    OutCodeFIFOFull     :   OutCodeFIFOFull<0>,
    OutCharFIFOEmpty    :   OutCharFIFOEmpty<0>,
    StackFull           :   StackFull<0>,
    ReadCAMIn           :   ReadCAMIn<0>,
    WriteCAMIn          :   WriteCAMIn<0>,
    CAMOutAddrIn    <11:0>;

CLOCK
    CLK;                        !   System clock
SUPPLY      Vdd;
GROUND      GND;

INSTANCE    control:logic PROMOTE;

!   Place flip-flops for flags first.
```

Figure 46 - Control.bdnet (cont. on next page)

```
INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:Reset_D<0>;
    CLK2:CLK;
    Q:Reset_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:DoCompress_D<0>;
    CLK2:CLK;
    Q:DoCompress_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
    DATA1:DumpOutCode_D<0>;
    CLK2:CLK;
    Q:DumpOutCode_Q<0>;
    Q_b:UNCONNECTED;
    "Vdd!":Vdd;
    "GND!":GND;

!   Now place flip-flops to form all the registers.

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:RefreshCt_D<%I>;
        CLK2:CLK;
        Q:RefreshCt_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ARRAY %I FROM 0 TO 7 OF
    INSTANCE "/cad/lib/technology/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1:RefreshAddr_D<%I>;
        CLK2:CLK;
        Q:RefreshAddr_Q<%I>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;

ENDMODEL;
```

Figure 46 (cont.) - Control.bdnet

```
#*****************************************************************************
#
#  File name      :  Makefile
#  Written by     :  Bob Wall
#  Date written   :  4/25/91
#
#  Description    :  Makefile to generate the various LZW module views
#                    using the tools in the OCT suite.  Can be used to generate
#                    any of the LZW modules (i.e. compress, decompress, or
#                    merged) by changing the "MODULE = " line below.
#
#  Other info     :  Adopted from Jaye Mathisen's makefile for the btpx project.
#
#  Modifications :
#       $Header: /n/dali/u1/icsg8038/thesis/merged/RCS/Makefile_2,v 1.1 1991/07/08
15:43:44 icsg8038 Exp $
#       $Log: Makefile_2,v $
#       Revision 1.1  1991/07/08  15:43:44  icsg8038
#       Initial revision
#
#*****************************************************************************

#*****************************************************************************
#   Define the name of the module, and the various options for the OCT tools.
#*****************************************************************************

MODULE = merged_2
STATS_FILE = m2_stats
IC = $(MODULE)_ic
MISIILIB = ~octtools/lib/misII/lib/script.msu
WOLFEOPTS = -h # -t ./wolfe.rules
MOSAICOOPTS = -c -s
PADFAMILY = sc2
PUPPYGRAPHICS = -g
#CADBIN = ~octtools/cad/bin
CADBIN = /cad/bin/
#CADBIN = ~octtools/bin
#TIME = /bin/time

#*****************************************************************************
#   Define the dependencies required by the various OCT views:
#       logic     - the output of bdsyn; just describes the combinational
#                   logic in BLIF format.
#       unplaced  - the output of bdnet; combinational logic with additional
#                   flip-flops attached to form registers.
#       flat      - output of octflatten; combination logic + flip-flops with
#                   hierarchy removed.
#       placed    - output of wolfe; flattened view placed and routed.
#
#   Other views are concerned with putting the placed view inside a pad ring.
#*****************************************************************************
LOGIC= $(MODULE)/logic/interface; $(MODULE)/logic/contents;
LOGICDONE = $(MODULE)/logic/.done
UNPLACED= $(MODULE)/unplaced/contents; $(MODULE)/unplaced/interface;
UNPLACEDDONE = $(MODULE)/unplaced/.done
FLAT= $(MODULE)/flat/contents; $(MODULE)/flat/interface;
FLATDONE = $(MODULE)/flat/.done
PLACED= $(MODULE)/placed/contents; $(MODULE)/placed/interface;
PLACEDDONE = $(MODULE)/placed/.done
CHIPBDNET = $(IC)/unplaced/contents; $(MODULE)/unplaced/interface;
CHIPBDNETDONE = $(IC)/unplaced/.done
PADSDONE = $(IC)/withpads/.done
PUPPYDONE = $(IC)/placed/.done
CURRENTDONE = .currentdone
PROPDONE = .propdone
MOSAICODONE = $(IC)/placed/.mosaico
RINGDONE = $(IC)/ringed/.done
```

Figure 47 - Makefile (cont. on next page)

```
#*****************************************************************************
#   OS commands that will come in handy down the road.
#*****************************************************************************
RM    = /bin/rm -f
RMR   = /bin/rm -rf
SHELL = /bin/csh
TOUCH = /usr/bin/touch -f

#*****************************************************************************
#   If the goal is to make the entire IC, with pads and all, the following
#   line should be first.  However, for right now, the desired goal is just
#   the placed view.  The commands for making the basics are in the reverse
#   order from the normal execution sequence.  Just typing make will ensure
#   that the placed view is up-to-date.
#*****************************************************************************
#   ic: $(RINGDONE)
$(PLACEDDONE): $(FLATDONE)
    $(TIME) $(RM)  $(PLACEDDONE)
    $(TIME) $(CADBIN)wolfe $(WOLFEOPTS) -o $(MODULE):placed $(MODULE):flat \
            2> wolfe.error
    $(TIME) $(CADBIN)chipstats $(MODULE):placed > $(STATS_FILE).placed
    $(TIME) $(TOUCH) $(PLACEDDONE)

$(FLATDONE): $(UNPLACEDDONE)
    $(TIME) $(RM)  $(FLATDONE)
    $(TIME) $(CADBIN)octflatten -t LEAF -o $(MODULE):flat $(MODULE):unplaced
    $(TIME) $(CADBIN)chipstats $(MODULE):flat > $(STATS_FILE).flat
    $(TIME) $(TOUCH) $(FLATDONE)

$(UNPLACEDDONE): $(LOGICDONE) $(MODULE).bdnet
    $(TIME) $(RM)  $(UNPLACEDDONE)
    $(TIME) $(CADBIN)bdnet $(MODULE).bdnet
    $(TIME) $(TOUCH) $(UNPLACEDDONE)

$(LOGICDONE): $(MODULE).blif
    $(TIME) $(RM)  $(LOGICDONE)
    $(TIME) $(CADBIN)misII -f $(MISIILIB) -T oct -o $(MODULE):logic \
            $(MODULE).blif
    $(TIME) $(CADBIN)chipstats $(MODULE):logic > $(STATS_FILE).logic
    $(TIME) $(TOUCH) $(LOGICDONE)

$(MODULE).blif: $(MODULE).bds
    $(TIME) $(CADBIN)bdsyn $(MODULE).bds > $(MODULE).blif

#*****************************************************************************
#   Following commands are all for making the chip - putting the placed
#   view inside a pad ring.
#*****************************************************************************
$(CHIPBDNETDONE): $(PLACEDDONE)
    $(TIME) $(RM)  $(CHIPBDNETDONE)
    $(TIME) $(CADBIN)bdnet $(IC).bdnet
    $(TIME) $(TOUCH) $(CHIPBDNETDONE)

$(PADSDONE): $(CHIPBDNETDONE)
    $(TIME) $(RM)  $(PADSDONE)
    $(TIME) $(CADBIN)padplace -P -u $(PADFAMILY) -o $(IC):withpads \
            $(IC):unplaced
    $(TIME) $(TOUCH) $(PADSDONE)

$(PUPPYDONE): $(PADSDONE)
    $(TIME) $(RM)  $(PUPPYDONE)
    $(TIME) $(CADBIN)puppy $(PUPPYGRAPHICS) -o $(IC):placed $(IC):withpads
    $(TIME) $(TOUCH) $(PUPPYDONE)

$(CURRENTDONE): $(PUPPYDONE)
    $(TIME) $(RM)  $(CURRENTDONE)
    $(TIME) $(CADBIN)bdnet $(MODULE).current.bdnet
    $(TIME) $(TOUCH) $(CURRENTDONE)
```

Figure 47 (cont.) - Makefile (cont. on next page)

```
$(PROPDONE): $(CURRENTDONE)
    $(TIME) $(RM)  $(PROPDONE)
    $(TIME) $(CADBIN)bdnet $(IC).prop.bdnet
    $(TIME) $(TOUCH) $(PROPDONE)

$(MOSAICODONE): $(PROPDONE)
    $(TIME) $(RM)  $(MOSAICODONE)
    $(TIME) $(CADBIN)mosaico $(MOSAICOOPTS) $(IC):placed
    $(TIME) $(TOUCH) $(MOSAICODONE)

$(RINGDONE): $(MOSAICODONE)
    $(TIME) $(RM)  $(RINGDONE)
    $(TIME) $(CADBIN)padplace -R -o $(IC):ringed $(IC):placed.spaced
    $(TIME) $(TOUCH) $(RINGDONE)

#*****************************************************************************
#   Clean up the various time-keeping and unnecessary files and views left
#   laying around.
#*****************************************************************************
clean:
    $(TIME) $(RM)  $(LOGICDONE) $(UNPLACEDDONE) $(PLACEDDONE) $(FLATDONE)
    $(TIME) $(RM)  $(CHIPBDNETDONE) $(PADSDONE) $(PUPPYDONE) $(CURRENTDONE)
    $(TIME) $(RM)  $(MOSAICODONE) $(PROPDONE) $(RINGDONE)
    $(TIME) $(RM)  $(MODULE).blif
    $(TIME) $(RMR) $(IC) $(IC).placed.spaced $(PROC)
    $(TIME) $(RMR) $(IC).ioleft $(IC).ioright $(IC).iotop $(IC).iobottom
```

Figure 47 (cont.) - Makefile