A VLSI Architecture For a Data Compression Engine In a Communications Network

by

Brian Ta-Cheng Hou

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements for the Degrees of

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1989

Signature of Author: [illegible]
Department of Electrical Engineering and Computer Science
January 6, 1989

Certified by: [illegible]
Professor Jon Allen
Thesis Supervisor

Approved by: [illegible]
Mickey Gutman
Company Supervisor, Codex Corporation

Approved by: [illegible]
Jay Pasco-Anderson
Company Supervisor, Codex Corporation

Accepted by: [illegible]
Professor Arthur C. Smith, Chairman
Departmental Committee on Graduate Students

A VLSI Architecture For a Data Compression Engine In a Communications Network

by

Brian Ta-Cheng Hou

Submitted to the Department of Electrical Engineering and Computer Science on January 6, 1989 in Partial Fulfillment of the Requirements for the Degrees of Master of Science and Bachelor of Science in Electrical Engineering and Computer Science

ABSTRACT

In the past, data compression (DC) algorithms used in modems or statistical multiplexers have been implemented in general purpose microprocessors which are already burdened with other tasks such as operating systems or data I/O. With the advent of high bandwidth communication media such as fiber optics, which demands high processing throughput, the traditional von Neumann model approach clearly is insufficient. In order to apply data compression to high speed communications networks, a VLSI data compression engine incorporating a specialized architecture, state-of-the-art VLSI technology, and high performance DC algorithms is needed to offload the microprocessor.

This thesis identifies an architecture which breaks the DC processing bottleneck. In order to achieve this goal, a specialized architecture which supports a single compression algorithm was favored over a general architecture which could be designed to support several algorithms. Specifically, an adaptive, string-matching compression algorithm is chosen as the basis for architectural design. A highly concurrent structure called content addressable memory is exploited to drastically reduce the encoding time and increase the throughput.

Thesis Supervisor: Jon Allen

Title: Professor of Electrical Engineering and Computer Science

Acknowledgement

I truly appreciate Mickey and Jay for always being supportive and patient and for the freedom they gave me to pursue whatever interests me. In particular, I wish to thank them for letting me borrow their Macs whenever I needed them; otherwise the thesis writeup would not have gone so smoothly.

I would also like to thank Prof. Allen for his encouragement and understanding, and for his sharing his interesting trip to Sweden.

Many thanks go to Craig Holt and Lloyd Hasley, both invaluable mentors. Their suggestions have contributed tremendously to the final design of the Data Compression Engine.

I wish to thank Craig Cohen for always being a great company; for the inspiring discussion on the thesis; for the sharing of knowledge in general; for his putting up with me in the last seven months and the past three summers; and for the professional attitude he has transcended on me.

Finally, I would like to dedicate this thesis to my family for their constant love and support, even though they still don't know what I was doing in the last seven months.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 PROBLEMS

Lossless data compression is a coding technique which minimizes data redundancy in order to reduce the offerred load to the communications network. Losslessness means that the exact original data can be recovered at some later time. By encoding data according to specific rules which aim at reducing the average number of bits per message, more information can be transmitted over channel in the same amount of time, thus the effective bandwidth of the communication channels is increased, and the response time, network loading, and probability of transmission errors may be reduced. Furthermore, the compressed codes can provide limited security against illicit monitoring. Data compression has been incorporated into statistical multiplexers and modems to attain improved throughput. Figure 1.1 shows a conceptual data compression system model.

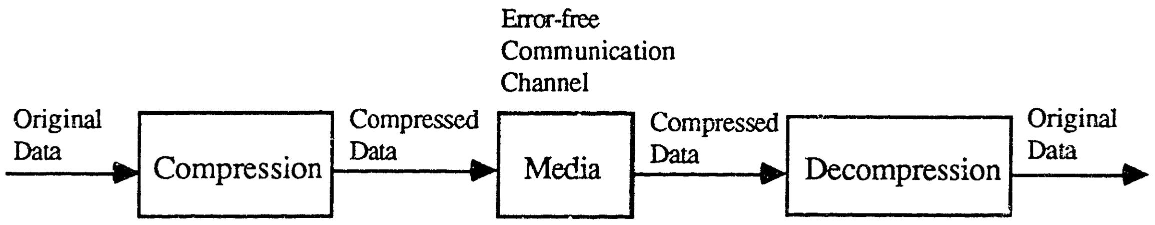


Figure 1.1 Data Compression System Model

However, several problems have prevented the widespread use of data compression in communications networks, especially those of high bandwidth. First, the data compression encoding and decoding algorithms implemented with software load down the microprocessor which has to perform other tasks such as operating system, data routing, and flow control. This constrains data compression to be applied only to low bandwidth communications networks, such as dial modems. The loading on the microprocessor also prevents the use of more sophisticated data compression algorithms which would produce better compression better compression results. The second problem is that most data compression techniques perform favorably with only certain types of data redundancy. For example, some algorithms work better for numeric data while others are more suited for text files. Since most data traffic is heterogeneous, either the algorithms need to adapt to different data types or several semantic-dependent algorithms are used in order to achieve optimal compression result [2].

1.2 OBJECTIVES

The aforementioned problems suggest that insufficient computing resources and memories are the major constraints for widespread data compression applications over high bandwidth communication channels. One of the solutions is to develop a VLSI (Very Large Scale Integration) data compression engine to handle data compression processing and free the microprocessor to concentrate on other duties. It is hoped that the use of such a dedicated engine will allow more complicated algorithms to be devised and facilitate the implementation of data compression algorithms in new network applications.

With the steady advance in VLSI technology, which allows thousands of transistors to fit on a single chip, special purpose chips that function as peripheral devices to a microprocessing unit are fast becoming an industry trend as a viable solution to enhance the system throughput. The high logic density of a VLSI chip also permits multiple processing elements to reside on the same chip, thus providing the opportunities to implement concurrent operations. As a result, mapping algorithms into special VLSI architectures to improve performance has been an active research area in recent years. The pattern matching chip developed at Carnegie Melon University is a famous example [38]. First, a concurrent algorithm for pattern matching was developed; then the systolic array which consisted of many small processing elements capable of parallel computation was implemented to carry out the algorithm. This approach is a radical departure from the traditional von Neumann model of computation in which a single processor executes a single instruction at a time.

Similarly, the goal of this thesis is to explore the possibilities for mapping the compression algorithms into specific VLSI structures. We would also like to investigate the maximum throughput of a specific, high performance data compression engine. This will determine if data compression can be applied to high-bandwidth communication networks such as T1 backbone (1.54 Mbps in North America) or local area networks (10 Mbps for Ethernet). In order to achieve this goal, a specialized architecture, which supports a single data compression algorithm with reasonable compression ratio, is favored over the general architecture which can be designed to support several types of algorithms. Based on the architecture, an area estimate on the major blocks of the chip will be made to reflect the cost.

1.3 APPROACHES

The ZL77 algorithm chosen in this work was developed by Ziv and Lempel [7]. It is a string matching algorithm which encodes variable-length input string to fixed-length codeword. ZL77 is selected as the basis for the specialized architecture for two reasons. First, it has the potential for VLSI implementation. Second, it is a proven high performance algorithm. The following outlines the sequence of investigations and procedures to complete the thesis.

The first step is to study the ZL77 algorithm in detail and completely understand the operations required. This will reveal the implementation bottlenecks which must be solved. The second step is to research existing data structures being proposed to implement the algorithm. These two steps constitute the first half of the thesis and will indicate the critical sequences of operations, possibilities for concurrent processing, data memory requirements, and instruction sequences.

The second half of the thesis consists of investigating and developing a VLSI architecture suitable for ZL77 algorithm implementation. Functional blocks and interconnecting buses will be proposed. Several hardware components will be designed to process frequent data compression related operations or to replace traditional software operations which perform housekeeping tasks. Finally, technological bottlenecks, throughput, silicon area, and interface mechanism with an external central processing unit (CPU) will be identified.

1.4 ORGANIZATION

Chapter 2 describes the general concepts of data compression. The ZL77 algorithm and several published data structures which support it are surveyed and their complexities are summarized. In Chapter 3, special VLSI structures are developed to implement the critical operations. Their throughputs and silicon areas are estimated. Chapter 4 proposes one data compression engine chip architecture. A generic system environment in which the data compression engine operates is described. (An interface mechanism with an external microprocessor is assumed for concreteness.) A finite state machine controlled ZL77 encoder and decoder are fully implemented and their throughputs and areas estimated. Chapter 5 concentrates on the VERILOG behavioral simulation of the proposed data compression engine. Chapter 6 presents the future extensions to the current designs, and finally, Chapter 7 concludes by summarizing the highlights of the research.

CHAPTER 2

ZL77 DATA COMPRESSION ALGORITHM

The first section of this chapter covers the basics of data compression and its practical applications. Section 2.2 concentrates on the basic ZL77 encoding algorithm and its variations. Then, the data structures that have been proposed to implement the algorithm are presented and their merits and shortcoming are analyzed in Section 2.3. Finally, Section 2.4 identifies the critical operations for ZL77 implementations.

2.1 FUNDAMENTAL CONCEPTS ON DATA COMPRESSION

The primary objective of data compression is to minimize the amount of data to be transmitted while preserving the information which the original data contains. Thus, data compression is concerned with transforming the source messages in one representation (such as ASCII or EBCDIC codes) into a new string of codewords that has shorter average length but carries the same information.

This section provides the general background on data compression and briefly describes relevant definitions and concepts. Next, the applications of data compression are discussed. Finally, the focus on the specific type of data compression application and algorithm for the thesis are presented.

2.1.1 Definitions

In information theory, a *message* usually consists of one or more binary symbols (0 or 1) called *bits*. The bit pattern that represents a message is a *codeword*. For example, the codeword for message 'a' in ASCII is binary 01100001. A *Message ensemble*, on the other hand, is a sequence of messages [2]. For instance, the string "Hello, world" is a message ensemble consisting of twelve messages (including the space that follows the comma). In the thesis, *character* and *symbol* are synonomous with *message* while *string* and *message ensemble* are used interchangeably.

The process of transforming source ensemble into a sequence of codewords is called *encoding*; the process of reversing the above operation is called *decoding*. The entities that carry out the encoding and decoding processes are the *encoder* and *decoder*, respectively.

From the information theory point of view [1], there is a probability distribution associated with a message, since it is one realizationof a random variable. As a result, some messages are more probable than others in a given message ensemble, such as the message '*l*' in "Hello, world". Therefore, one compression approach is to map the more probable messages into the shorter codewords and the less likely ones into the longer codewords, thus reducing the average codeword length. This encoding scheme is categorized as *fixed-variable* (F-V) because the message length is fixed while the codeword length varies from character to character. *Huffman coding* is representative of the F-V class of compression algorithms [4-6].

String matching, or*variable-fixed*, is another class of encoding method which exploits redundancy due to repeating string and other types of redundancy. *Ziv-Lempel coding* is an example of string matching compression method, and there are many variations of this algorithm [7-10]. In the ZL77 scheme, variable length strings are replaced with fixed length codewords which point to earlier occurrences of the same strings. Therefore, both the encoder and the decoder are required to keep and build a *history buffer*, a collection of all of the character strings being processed up to the present. In essence, the encoder and decoder are able to *learn* or *adapt to* the source characteristics.

Compression ratio (CR) is a measure of compression; it has two well-known definitions [2]. CR may be defined as (average source message length)/(average codeword length) or as (average codeword length)/(average source message length). For example, if the average source message length is 8 bits and the average codeword length is 4 bits, then CR is 2 for the first definition and 0.5 for the second definition. This thesis uses the first definition when referring to compression ratio.

2.1.2 Applications of Data Compression

Data compression has two major areas of applications: data storage and data transmission. The first area of application involves compressing the data before it is stored in the digital storage media. As more offices are computerized, the volumes of data to be stored also become very large. If the file size could be reduced, the effective capacity of the storage medium would be increased. At the same time, the input/output traffic of a computer system could be decreased.

The second area of application concerns the real-time compression of data before it is transmitted over the communication links, such as phone lines, satellite channels, or cables that connect local/wide area networks. The proliferation of the communication networks has resulted in massive data traffic over the communication links. If the data were compressed to half its size, the effective bandwidth of the communication channels would be doubled.

In the past, there has been a tradeoff between the benifits of data compression and the computational cost associated with the encoding and decoding processes. However, the prices of microprocessors and custom VLSI chips have lowered steadily, so data compression is more popular now as the needs to store and/or transmit large volumes of data grow at a rapid pace and the savings achieved by data compression in storage or communication costs become more significant. Application to data transmission is especially attractive because the communication costs now dominate over the costs of memory and processing power.

2.1.3 Focus On The Thesis

Data storage and data transmission both require encoding and decoding of data. However, the speed in data storage application is not as critical as that of data transmission. For example, it is tolerable to wait for a few seconds before a program is loaded from a disk to a system's random access memory. Furthermore, the connection between the storage media and the computer system is usually local and free, unlike a telephone line which is charged by the amount of time the line is active. As a result, the compression algorithms for this type of application can afford to take the two-pass approach. In the first pass, the source data is scanned and statistics are gathered to determine the character frequency which is used to map or construct the codewords. In the second pass, the characters are encoded and stored. The important concept for this application is that speed is not the primary concern, and the path between the storage element and the host is free.

Data transmission has different requirements. For example, too much transmission delay in most communication networks is undesirable. Secondly, sessions are indefinite; they go on forever. This suggests that data compression and decompression have to be performed in real time, or at least not much slower than the channel bandwidth; this makes the two-pass approach unviable. Therefore, compression algorithms geared for this type of application usually are one-pass only with continuous encoding and decoding at high speed as the data are transmitted and received.

This thesis only considers the algorithms for data transmission application, which requires high throughput, low delay, and continuous one-pass compression and decompression.

2.2 ZIV-LEMPEL '77 ALGORITHM

Ziv-Lempel '77 is one of several variable-length input and fixed-length output (V-F) class of data compression algorithms. It was proposed by Ziv and Lempel in 1977, and is widely known as ZL77 for short. Other V-F class algorithms are ZLSS, ZLW, ZL78, and so on [7-10]. The following sections discuss the basic ZL77 algorithm and the modified algorithm.

2.2.1 The Basic Algorithm

ZL77 algorithm is based on the concept that in the continuous data stream, some string patterns occur more than once. Therefore, if we keep a history of the data, we can find the longest match of the incoming character string from the history buffer and then encode the string with a pointer to an earlier occurrence of the string. The pointer, or codeword, consists of Index, Length, and Innovation Character. Index shows how far back from the current input string the match starts; Length shows the length of the match; and finally, Innovation Character is the first input character not included in the match. Data compression is achieved if the number of bits required to represent the codeword is less than the number of bits required to represent the string. Figure 2.1(a) illustrates the basic ZL77 algorithm.

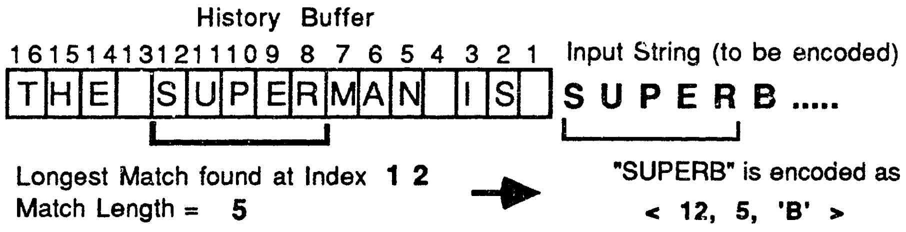


Figure 2.1 (a) The Basic ZL77 Encoding Algorithm

Let us take a snapshot of the encoding process. Assume that at time t = 0, the history buffer contains the character string "THE SUPERMAN IS " which has already been encoded and just became history. The incoming data "SUPERB...." is to be encoded next. By observation, the longest match is "SUPER", which can be found by counting 12 characters to the left from the current input string. The match length is five and the character 'B' is the Innovation Character since it is the first input character not belonging to the longest match. Therefore, the string "SUPERB" is encoded as shown in Figure 2.1(a).

In short, the characters which have been encoded before now become part of a history buffer. The encoder examines the current incoming string and searches the history buffer to find the longest match and replaces it with a codeword.

It is impossible keep all of the messages because only finite memory is available. Furthermore, when the history buffer gets really large, the codeword length required to identify the history buffer also becomes too large to have any compression benefit. One feasible alternative is to store only a fixed, reasonably large number of characters in the history buffer. As new strings are encoded, they enter the history buffer and the oldest members have to leave so that the buffer size remains constant. In essence, the history buffer acts as a sliding window, moving from left to right as new strings are encoded. For example, in Figure 2.1(b), the history buffer moves forward to cover the newly encoded "SUPERB" while the oldest characters "THE SU" drop out and are no longer in the history buffer. If the character string follows "SUPERB" starts with "THE..." then a match will not be found. This clearly illustrates why the history buffer should be reasonably large. If it is too small, such as the one in Figure 2.1, then the chance of finding a matching string is too small to have any compression benefit.

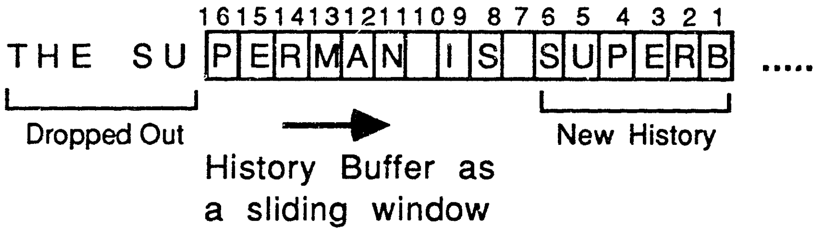


Figure 2.1 (b) Encoder History Buffer After Encoding "SUPERB"

In summary, ZL77 takes in a variable-length string and produces a fixed-length codeword. If N is the size of the history buffer, then $\log_2 N$ bits are necessary to uniquely specify every Index. If L is the number of bits for the Length field, then the maximum match length is $(2^L - 1)$. Finally, assume that each Innovation Character is 8-bit wide, then the size of the codeword will be $(\log_2 N + L + 8)$ bits. On average, if the bit length of the codeword is less than the bit length of the string which it represents, then data compression is achieved.

Sometimes, ZL77 encoding actually expands the data. This penalty occurs whenever the string bit length is shorter than the codeword bit length. An extreme case is when there is no match at all. The single 8-bit character is then encoded as $(\log_2 N + L + 8)$-bit codeword, an expansion of $(\log_2 N + L)$ bits. Depending on the values of N and L, expansion could happen if the match length is only one or two. Nonetheless, ZL77 usually produces satisfactory compression ratio of about 2 to 1.

ZL77 decoding is relatively straightforward. The decoder keeps the same history buffer as the encoder and updates the history buffer the same way. To decode, the Index field is extracted from the codeword and used as a pointer into the history buffer. The Length field is used to determine how many characters should be read off from the buffer. Finally, the Innovation Character is taken as it is. Then the decoder inserts the decoded characters, including the Innovation Character, into the history buffer and deletes the same number of the oldest characters from the history buffer. Therefore, as long as the codewords arrive in order and the encoder and decoder updates the history buffer appropriately, the compressed data will be correctly decompressed.

Figure 2.2 (a) and (b) show the ZL77 decoding process.

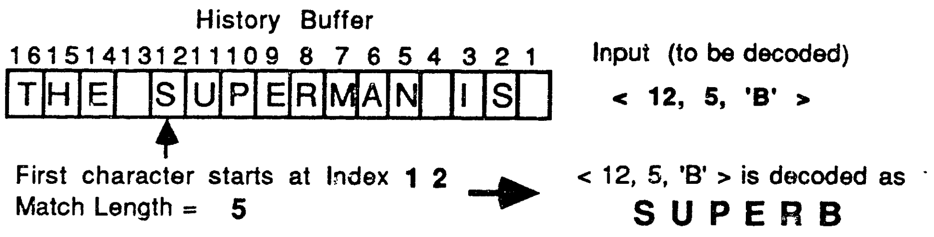


Figure 2.2 (a) The Basic ZL77 Decoding Algorithm

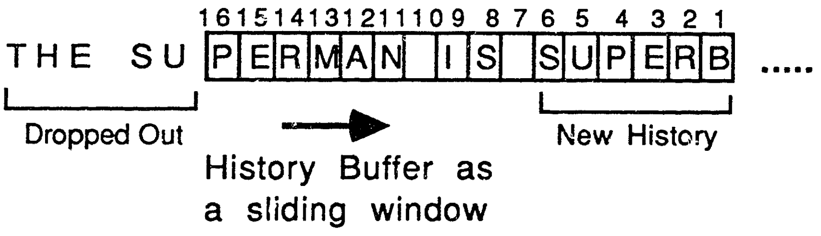


Figure 2.2 (b) Decoder History Buffer After Decoding

2.2.2 The Modified Algorithm

The modified ZL77 scheme differs from the basic algorithm in two aspects. First, the Innovation Character is dropped from the codeword. Second, the absolute index, rather than relative index, is used. The basic ZL77 principle is not violated by these modifications.

The first modification is similar to the ZLSS scheme proposed by Storer and Szymanki [9]. In ZLSS, the codeword normally consists of Index and Length without the Innovation Character; but in the case when the codeword is longer than the characters it encodes, the characters are transmitted as a codeword instead. Clearly this scheme attempts to optimize compression ratio; but in doing so, an extra bit is required to tell the decoder which one (character or pointer) is transmitted. In the modified algorithm used in the thesis, the codeword is always made up of Index and Length. In the case of no match (i.e. Length = 0), the encoder puts the 8-bit raw character into the Index field and places a 0 in the Length field. The decoder, upon detecting a Length of 0, will take the raw character from the Index field without going into the history buffer.

The second modification is simply a different way the Index portion of the codeword is interpreted. The following example should help clarify these modifications.

2.2.3 An Example

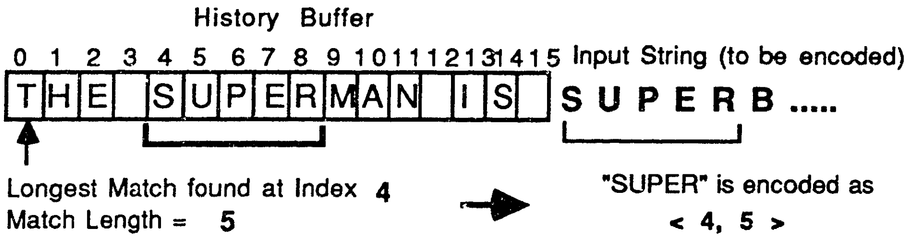


Figure 2.3 (a) The Modified ZL77 Encoding Algorithm

Let us use the same encoding example for the basic ZL77 algorithm to illustrate the differences as shown in Figure 2.3 (a). Notice how the history buffer is indexed. In the basic algorithm, the Index portion of the codeword tells the decoder how many characters back from the end of history buffer the longest match can be found. For example, in Figure 2.1 (b), the string "SUPER" is recovered by counting back 12 characters from the newest character in the decoder history buffer. This is called relative indexing. The decoder needs to keep a pointer which points to the end of the history buffer. The decoder then subtracts the Index from the pointer to uncover the string.

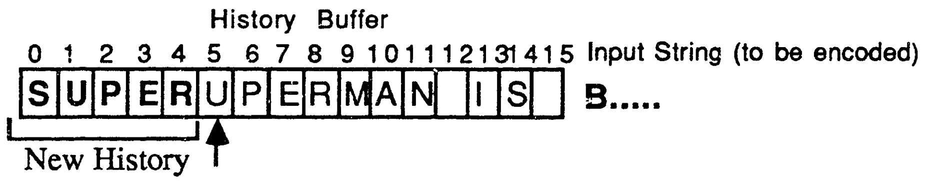


Figure 2.3 (b) Encoder History Buffer After Encoding "SUPER"

In the modified encoding algorithm, the Index portion of the codeword specifies the exact location in the history buffer where the match lies. In other words, the decoder simply takes it as a pointer into history buffer without having to perform the subtraction. This is called absolute indexing. The Data Compression Engine developed in the thesis encodes and decodes with absolute indexing.

Another distinction is that the codeword no longer includes the Innovation Character. The character 'B' is not encoded yet; it will be the first character of the next input string to be encoded as shown in Figure 2.3 (b).

Both the encoder and the decoder have to keep a pointer that points to the end of the history buffer for update purpose. The pointer is incremented modulo the size of the history buffer, so that after the maximum index is reached, the pointer will point to location 0 the next time. In Figure 2.3 (a), there is an arrow below position 0. This means that position 15 contains the most recently encoded character and that position 0 is where the new character should be inserted. This is demonstrated in Figure 2.3 (b). After "SUPER" is encoded, position 0 to 4 are overwritten with the newly encoded string, and the arrow now points to position 5.

2.2.4 ZL77 Implementation Parameters

In the Data Compression Engine architecture, the size of the history buffer N is 1024 characters; L, the number of bits for the Length field, is 4, which implies that a maximum match length of 15 is allowed (zero is reserved to indicate a no match).

Therefore, the codeword length is $(\log_2 N + L) = (\log_2 1024 + 4) = 14$ (bits).

2.3 DATA STRUCTURES

In ZL77 encoding, the most time-consuming operation is to find the longest match for the input string from the history buffer, which contains the N most recently encoded characters. This is a difficult process as a naive approach will easily take $O(N^2)$ searches. Various data structures have been proposed to reduce the search time, each with tradeoffs between memory requirements and processor loading. Three of the data structures have been surveyed and analyzed: binary search tree, hashing, and systolic arrays. They are summarized in the following sections.

2.3.1 Binary Search Tree

Binary search tree data structure was proposed by Bell [8]. In a binary search tree, for any node n, all node values on its left subtree are less than n, and all node values in its right subtree are greater than n. Figure 2.4 shows an example of a binary search tree.

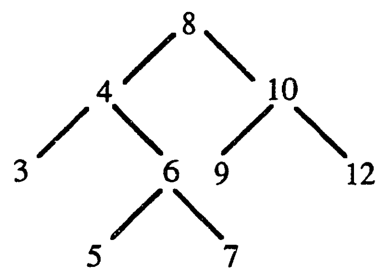


Figure 2.4 An Example of a Binary Search Tree

In the applications for ZL77 encoding, Bell suggested that all possible strings of fixed length in the history buffer be organized lexicographically in a binary search tree to reduce the number of searches for the longest match. For example, let the history buffer contain 7 characters and input string be "bacd" as shown in Figure 2.5.

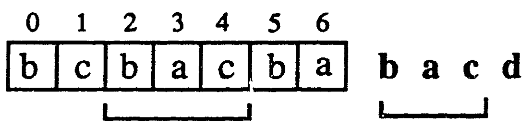


Figure 2.5 History Buffer and Input String

Let the maximum match length to be four. Let S(i) denote the four-character string whose first character starts at index i of the history buffer. Notice that it is a *circular buffer*, so that the four-character string which starts at Index 6 would include characters from positions 6, 0, 1, and 2. We then have the following set of strings possible in the history buffer:

S(0) = bcba S(1) = cbac S(2) = bacb S(3) = acba
S(4) = cbab S(5) = babc S(6) = abcb

The binary search tree is constructed as shown in Figure 2.6. For any given node i, all the strings on its left subtree are lexicographically lower than S(i), and all the strings on its right subtree are lexicographically higher than S(i). The input string, of course, is I = "bacd". To find the longest match for the input string, it is necessary to traverse from the root. Bell argued that in the process of updating the history buffer, the input string has to be inserted into the binary search tree as well; it is a dynamic tree as some strings disappear when the characters associated with them are deleted and new strings appear after they are encoded. In order to maintain the binary search tree property, the input string has to traverse down the tree to find out where it should be inserted. Bell recognized that the longest matching string will be ON the traversing path of the input string; therefore, finding the longest match actually becomes a by-product of updating the binary search tree!

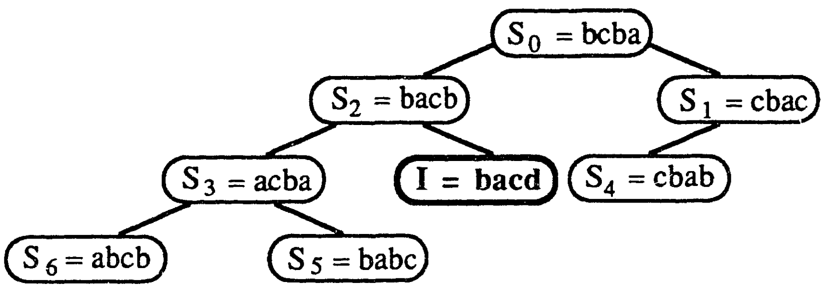


Figure 2.6 Binary Search Tree for the Given History Buffer

Bell argued that the longest match has to be one of two nodes: the parent node onto which the input string is inserted as a son, or the node where the traversal last turned the direction different from the direction the input string is inserted. For example, if the input string is inserted as a righthand child, then the other candidate for the longest match is the node in which the most recent left turn is made. On the other hand, if the input string is inserted as a left child, then the other candidate wold be the node where the most recent right turn is made. Therefore, the match length can be found by just comparing the match lengths of those two nodes.

In this particular example, the two candidate strings for the longest match are S(0) and S(2). S(2) is the parent of the input string I, and S(0) is the node where the input string last turns left (the new string is inserted as the right son of S(2)). S(2) turns out to be the node that contains the longest matching string.

For each L characters match, L insertions and deletions are required. Each insertion needs an average of $O(\log_2 N)$ string search, where N is the number of nodes, or strings. Within each string search an average of two character comparisons are required. For deletion, it is not necessary to search because we always know which node to delete next by keeping a pointer to the array of N nodes. However, in order to maintain the binary search tree property, the nodes around the deleted node must be adjusted, and this could be time-consuming. Appendix A shows the C codes for binary search tree updates.

The memory requirement for the encoding operation consists of a history buffer of size N and the data structures for an array of N nodes. Each node requires three pointers: two pointers for sons and one pointer for a parent. The parent pointer is necessary because during deletion, the parent of the node to be deleted must be identified so that a link for the new son can be made. The character strings themselves need not be stored in the node data structures because each node number i implicitly points to S(i). For example, the string associated with node 1 is "abbc", or S(1). Finally, the son and parent pointers are actually indexes into the array of N nodes. Figure 2.7 shows the node data structures which store the binary search tree of Figure 2.6.

Node #	Left Son	Right Son	Parent
0	1	2	0
1	4	Null	0
2	3	Null	0
3	6	5	2
4	Null	Null	1
5	Null	Null	3
6	Null	Null	3

Figure 2.7 Node Data Structures for Binary Search Tree

If each character is represented with eight bits, then the total encoding memory required will be $[8N + 3N (\log_2 N)]$ bits for the binary search tree [2].

This data structure only needs $O(\log_2 N)$ search time if the tree is balanced. By organizing all of the possible strings in the history buffer lexicographically in a binary search tree, only a subset of strings need to be searched; this is a great improvement over brute force search, in which $O(N^2)$ search time is required.

2.3.2 Hashing

Hashing is another technique to reduce the processing complexity of finding the longest match. In a N-character history buffer, there are potentially N different strings from which to search in a brute force implementation. Suppose we calculate the hash value[1] for each possible string and chain the strings that have the same hash values in a linked list. To search for the longest match, the hash value of the input string is first calculated, then only the strings that have the same hash value are compared to the input string. Since we only have to compare a subset of all possible strings, the search time is greatly reduced. Figure 2.8 illustrates one approach to hashing and its data structures.

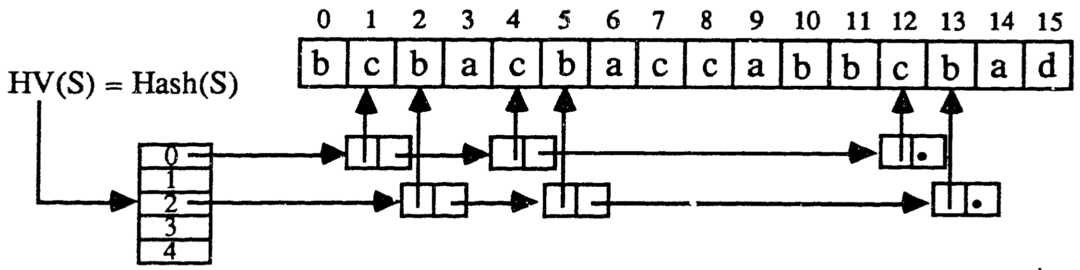


Figure 2.8 The Hashing Data Structure

Figure 2.8 shows that strings that start with "ba" all have a hash value of 2, and those that start with "cb" have hash value 0. Each link entry has a pointer into the history buffer where the string starts and a pointer to the next link entry which has the same hash value. The black dots denote the end of the linked lists. Hash(S) is the hash function that performs on the first few characters of a string S, and HV(S) denote the hash value of a string S. Depending on the hash function chosen and thus the number of distinct hash values possible, different strings with different initial characters could result in the same hash values.

Clearly, the search time is directly proportional to the length of the linked list; it is undesirable if the linked list gets too long. Therefore, the idea is to devise a hash function to have a distribution of hash values that minimizes the expected number of searches, i.e., a hash function that minimizes $\Sigma(L_{HV} * P[HV(S)])$, where L_{HV} denotes the length of a list with hash value HV, and P[HV(S)] denotes the probability of a hash value HV. This is impossible, however, because the input string characteristics are unpredictable. Another alternative is to set an upper bound on the number of searches in a linked list, so that the worst case search time can be deterministic. However, the penalty is that the longest matching string might be near the bottom of the linked list and not found.

[1] A hash value is produced by manipulating the input of a hash function in a certain way. For example, a hash function for a string could be (4*C1+C2), which means a hash value is obtained by shifting the first character two positions to the left, then adds the result to the second character.

Updating the hash tables takes a constant amount of time. Before inserting the new characters into the history buffer, the oldest characters must be deleted from the hash table. Therefore, the hash values for the strings that start with those characters must be calculated to locate the linked lists to which they belong. Then the pointers to those old characters are removed from the linked lists. This accomplishes the delete operation. Next, the new characters are inserted into the history buffer. For each input character, a hash value for the string that starts with the character has to be calculated. The pointer that points to this character in the history buffer is then inserted into the end of respective linked list.

2.3.3 Systolic Arrays

Parallel algorithms for data compression using the systolic arrays approach were proposed by Storer [9]. While binary search tree and hashing implementations require sequential operations and extensive memory accesses based on a von Neumann model of computations, a systolic arrays is a VLSI structure with distributed and parallel computing capabilities.

In systolic arrays, the idea is to lay out a regular pattern of identical processing elements, each capable of carrying out simple tasks. In addition, these processing elements are to have simple interconnections. For example, each processing element only connects with adjacent elements; a global communication line does not exist. This attribute minimizes signals propagation delay so that a faster clock can be used to attain better performance. With many processing elements available, parallel computation is possible.

Storer proposed two parallel algorithms for string substitution data compression methods: static dictionary model and sliding dictionary model [9]. The first scheme uses a static dictionary of strings; input characters are compressed by replacing the substrings with pointers to matching strings in the dictionary. The decoder on the receiving end then decompresses the data by using the pointers to read off the strings from the dictionary. In the second scheme, the dictionary is constantly updated so that it only contains the N most recent characters. The sliding dictionary model is more relevant to the thesis so it is discussed in a greater detail below.

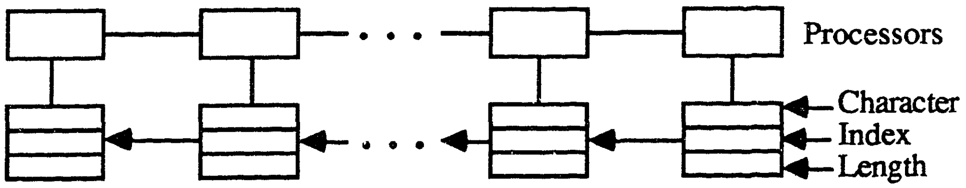


Figure 2.9 The Systolic Pipe

In the sliding dictionary model, an array of 2n processors is used to keep the 2n most recently encoded characters to process a block of n input characters. The reason that 2n processors are required is that each character has to be compared with each of the n characters that precede it. In order for the comparisons to take place in parallel for every n input characters, a window of 2n most recent characters must be kept. The following description should clarify this requirement.

Let every processing element have a local memory and a comparator. Each input character, along with its index and match length values, goes through the systolic pipe from the right, as shown in Figure 2.9. In order to synchronize the processors, two clock pulses called pulse 1 and pulse 2 are used. A pulse 2 occurs after n pulse 1's. Assume that the characters $X_1,\ldots,X_n$ have already been encoded, and while they were being processed, characters $X_{n+1},\ldots,X_{2n}$ were shifted in from the right. Figure 2.10 (a) illustrates this situation. Since n pulse 1's have passed, a pulse 2 occurs. At this moment, every processor stores the character below it into its local memory. Figure 2.10 (b) shows the consequence of this action.[2]

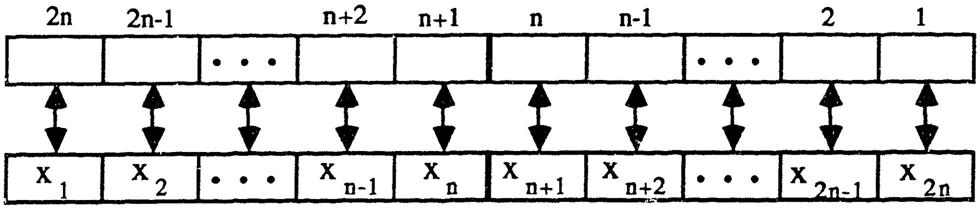


Figure 2.10 (a) Right Before a Pulse 2

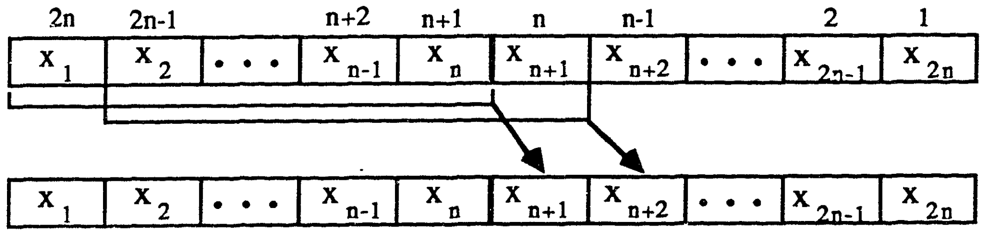


Figure 2.10 (b) Right After a Pulse 2

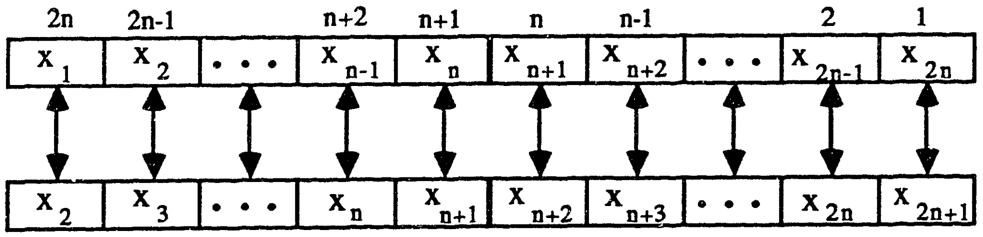


Figure 2.10 (c) After the first Pulse 1

[2] These figures are similar to those of Storer's in [9].

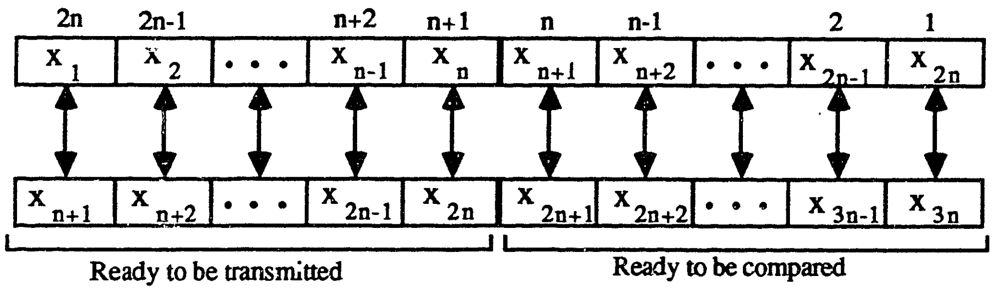


Figure 2.10 (d) After n Pulss 1's

At each subsequent pulse 1, the characters $X_{n+1},...,X_{2n}$ are shifted to the left and examined by the processors above them. These comparisons take place simultaneously. Within the next n pulse 1's, character X_{n+1} will be compared with $X_n,..X_1$ by processors $P_{n+1},..., P_{2n}$ and character X_{n+2} will be compared with $X_{n+1},..., X_2$ by processors $P_n ,..., P_{2n-1}$ as these characters are shifted to the left one slot at each pulse 1. Figure 2.10 (c) shows the systolic pipe after the first pulse 1 since the last pulse 2.

If a processor has a match, it will look at its left-hand neighbor and right-hand neighbor. If its left-hand neighbor does not have a match, it will generate a left bracket. If its right hand neighbor does not have a match, it will generate a right bracket. Then it waits for the higher level processing elements which will pair the brackets to determine the current match length. If it is greater than the match length associated with the current character, the match length and index will be updated to the new match length and current processor number. Otherwise, nothing is changed, and the characters continue shifting to the left. After n pulse 1's, character X_{n+1} would be under processor P_{2n}, as shown in Figure 2.5 (d). This processor is responsible for transmitting the codeword, or index and match length values. For example, if the match length associated with character X_{n+1} is three, then processor P_{2n} will transmit the match length and the index associated with X_{n+1} and ignore the codewords associated with the characters X_{n+2} and X_{n+3} since they are part of the three-character string being encoded.

As can be seen, the input characters enter the systolic arrays at a constant rate, i.e., a new character enters the systolic pipe at every pulse 1. After 2n pulse 1's, the codeword is ready to be transmitted. Therefore, the major advantage of the systolic arrays approach is that the throughput is independent of the array size. However, the latency for each character is directly proportional to the array size. In this case, each character has to stay in the pipe for a period of 2n pulse 1's before leaving the pipe, and this could be undesirable for a data communication network which is intolerable of too much delay.

2.3.4 ZL77 Data Structures Implementation Summary

The memory requirements and the average search time for the three data structures implementing ZL77 encoding are summarized in the following table. N is the size of the history buffer.

Data Structure	Memory Requirement	Avg Search Time
Binary Search Tree	$N + [3N * \log_2(N)]/8$	$O(\log_2 N)$
Hash Table	$N + [(N+42) * \log_2(N)]/8$	$O(N/\#$ of hash values$)$
Systolic Array	$2N*[21+2 * \log_2(2N)]/8$	$O(1)$

Note: the memory requirement for Systolic Array is for regular RAM storage only and does not include the processing elements

Figure 2.11 shows the memory requirements for each data structure as a function of history buffer size.

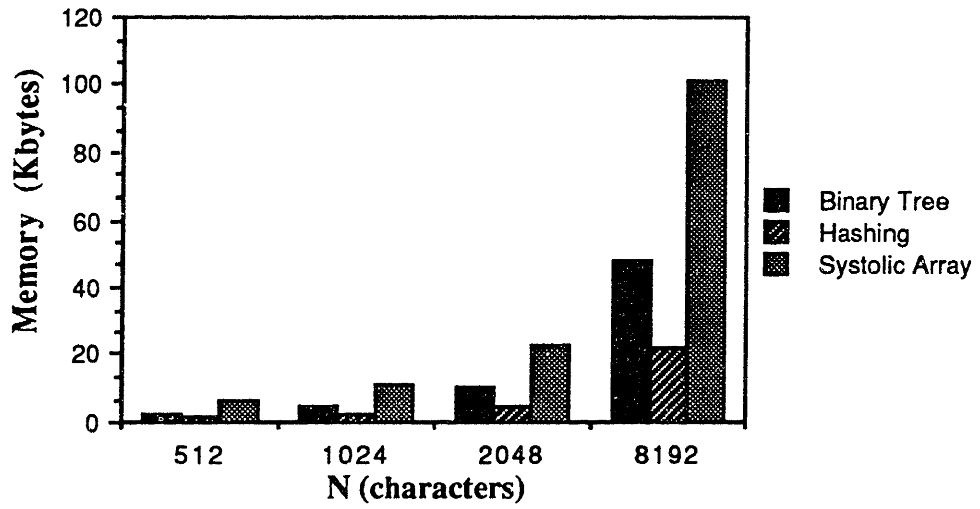


Figure 2.11 Memory Requirement Comparison (Encoder only)

The following summarizes each data structure.

Binary Search Tree

- The worst case search time is O(N) but it rarely happens.
- It guarantees finding the longest matching string.
- Critical operation is the tree update in which deletion and insertion take place for each input character. Tree update complexity depends on the neighborhood of the node to be deleted.
- Might need to implement tree balancing algorithm to keep average search time at $O(\log_2 N)$. This is an added complexity.

Hashing

- The worst case search time can be preset, i.e. the search can be terminated after certain number of searches in a linked list are reached. As a result, sometimes the matching length found is not the maximum.
- The most critical operation is the character comparisons in the history buffer. The table update process, in which deletion and insertion take place for each input character, takes only O(1) time because the pointers to the first and last elements of the linked list can be kept in the hash pointer table to facilitate the update process. Another critical operation is that hash values have to be calculated twice for each incoming character: once for insertion, and once for deletion.
- Overall a very efficient implementation because both memory requirement and processing complexity are relatively low compared to binary search tree.

Systolic Arrays

- There is a constant search time. A constant O(2N) delay in encoding and decoding time is one of the major drawbacks.
- Brute force comparisons are done in parallel. Essentially, each character is compared with its N previous characters. In each clock cycle, a block of N characters are compared.
- Suitable for VLSI implementation. However, at least 2N processing elements in the Storer implementation are required for encoding. Each processing element requires substantial logic and therefore will take a lot of area. This is likely to limit the history window size.
- At least three different clocks must be used.
- 6N memory is required for the decoder in the Storer implementation.
- 3N processors are required for the decoder. However, each processing element is simpler than its encoding counterpart.

2.4 CRITICAL OPERATIONS

From the data structures surveyed in the previous sections, the single most frequent operation is character comparison. In binary search tree and hashing implementations, each prospective string from the history buffer is compared with the input string character by character to find out which one has the longest match. In the systolic array implementation, multiple character comparisons take place simultaneously, but each input character has to go through N comparisons before leaving the systolic pipe. The former two data structures have the tremendous advantage of only having to compare a subset of strings and ignore the majority others, while the latter data structure allows character comparisons in parallel. It seems that in order to dramatically reduce the processing time, we must somehow combine those two attributes.

Other common operations for the two software data structures are numerous memory accesses and indirect addressing. Since memory accesses are usually the processing bottleneck, it would be nice to avoid them if possible. If any breakthrough in throughput is to be made, we must think of other solutions to find the longest match.

Regardless of which data structures are used, there are always needs for bit packing and unpacking. Bit packing refers to the bundling of normally non-byte size codewords into byte or word quantities. Bit unpacking is the inverse operation of extracting the codewords from a sequence of bytes. These operations are not as complex as finding the longest match, but they are so common that any speed-up will improve the overall throughput.

In short, finding the longest match, bit packing, and bit unpacking are identified as the operations critical to ZL77 encoding and decoding. Special VLSI structures will be developed to support these operations in the next Chapter. Of these operations, finding the longest match is the bottleneck in implementing the algorithm, and breaking this bottleneck will be a major effort of the thesis.

CHAPTER 3

SPECIAL VLSI STRUCTURES

3.1 VARIABLE LENGTH STRING MATCHER

Variable Length String Matcher (VLSM) is a dedicated VLSI structure which performs the longest string search, the most time intensive component of ZL77 algorithm. VLSM is an example of how the concurrency inherent in VLSI structure gives rise to an algorithm which is free of the traditional von Neuman bottleneck. In our case, the algorithm for finding the longest match exploits the parallelism provided in VLSM and enables the search time to be independent of the history buffer size. A discussion on the algorithm follows.

3.1.1 Algorithm For The Longest String Search

The algorithm is best explained through an example. Suppose eight simple processors, numbered from 0 to 7, are used as a history buffer. Each processor, also called a cell, has a comparator and a local memory that can hold a character. Furthermore, assume that each one is able to see the data on a global bus simultaneously. Figure 3.1 shows the history buffer and its contents. Let the input string be "abcd", and we would like to find the longest match for the input string and replace it with a ZL77 codeword. By inspection, we are sure that the longest match is "abc", so it should be encoded as <1, 3>. Let us see how the VLSM algorithm works to determine the codeword, and hence the longest match.

Suppose we start by forcing each cell to have a match, or hit. This step is the INIT cycle shown in Figure 3.1 as every cell has an arrow below it. Next, we present the first input character 'a' to every cell, which we call the COMPARE 'a' cycle. In order for a cell to get a hit, two criteria must be satisfied. First, the cell's content must match the input character; second, the left neighbor of the cell must have a hit in the previous COMPARE cycle. The second criterion is very important.

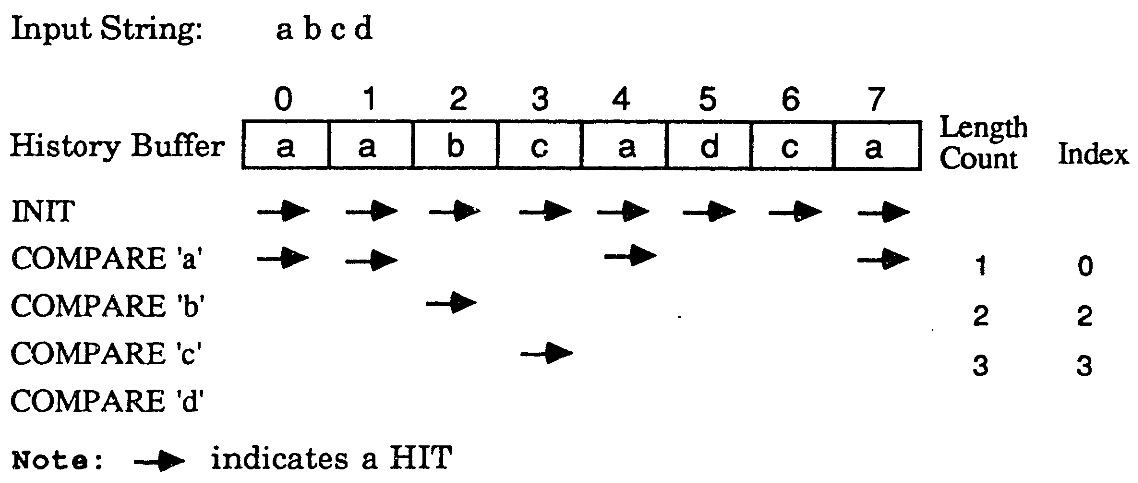


Figure 3.1 An Example For VLSM Algorithm

As can be seen, cells 0, 1, 4, and 7 indicate hits because their contents match the character 'a' and their left neighbors had hits in the previous cycle (i.e., the INIT cycle in which every cell was forced to have a hit). We then increment a length counter which keeps track of the match length. At the same time, the index of the cell which has a hit is recorded. Since multiple cells have hits, we need some kind of arbitration scheme. In this case we decide to record only the one that has the lowest index.

Next, 'b' is compared; only cell 2 has a hit. We again increment the length counter and record the index. We proceed to compare 'c', and as shown in Figure 3.1, only cell 3 indicates a hit even though cell 6's content is also 'c'. This is because cell 5, the left neighbor of cell 6, did not have a hit in the previous COMPARE cycle, so the second criterion is not satisfied. At any rate, we increment the counter and record the index. Finally, 'd' is compared; no more hit is indicated. This signals that we have found the longest match! The following explains how.

We can know that the match length is 3 by looking at the length counter. The index where the longest match begins can be found by subtracting the match length from the last index we recorded and then add one to it. In this example, index = 3 - 3 + 1 = 1. Therefore, the string "abc" can be encoded as <1, 3>, which is exactly what we expect.

Clearly, by enforcing the two criteria for a match, we can isolate the strings that have potential to be the longest match. This algorithm has the significant property in that the search time does not depend on history buffer size. Instead, it depends on the match length. For example, if L is the match length, then we only need to compare (L + 1) characters to find the longest match. This is a tremendous improvement from the search times for hashing or binary search tree data structures. In contrast to the systolic array implementation which also has high throughput, this algorithm has noticeably lower latency because all cells are comparing their contents with the same input character; the codeword can be produced as soon as the longest match is found. In systolic array, the latency is always proportional to the size of the history buffer, as was mentioned in Section 2.3.3.

In order to implement this searching algorithm, we must provide a structure capable of parallel comparisons similar to the systolic arrays arrangement yet with a globally distributed data bus. The prospective structure has been identified as content addressable memory. It is discussed in the next section.

3.1.2 Content Addressable Memory

Content addressable memory (CAM), also known as associative memory, is defined to be a device consisting of a number of cells which can store data and be accessed by their contents [32]. As opposed to random access memory (RAM), which during the read cycle takes an address as input and outputs the data stored in that memory location, CAM takes the data as input and then outputs the address which contains the data.

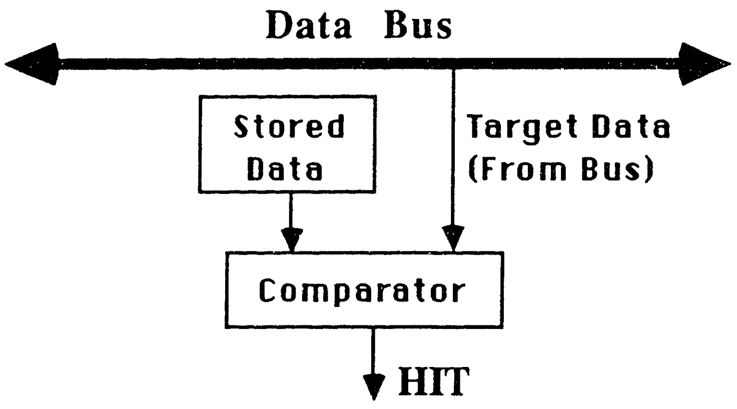


Figure 3.2 A Conceptual Model For a CAM Cell

In the classical CAM structure, each CAM cell has both a memory element and a comparator, as shown in Figure 3.2. As in RAM, each cell can be addressed for independent access. A global data bus is connected to every CAM cell and there is no other interconnection between adjacent cells. When a CAM cell contains the presented data, a HIT signal is asserted to indicate a match. Thus, in this fully-parallel CAM configuration, it is possible to know whether a particular datum is present in memory in just one clock cycle. Figure 3.3 illustrates the CAM structure.

The basic principles of CAM have been in existence since the 1950's. Despite its tremendous searching capability, CAM never flourished in the computer systems because its hardware complexity limited its use only to special roles such as small buffer memories

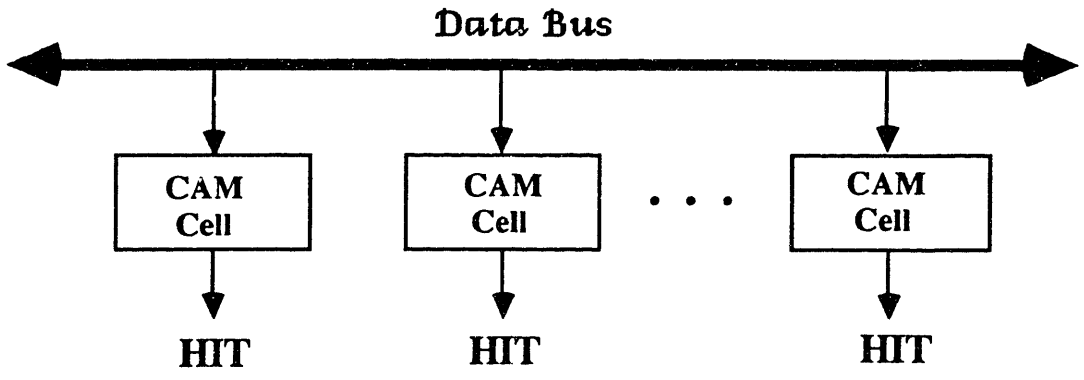


Figure 3.3 A CAM Structure

or control units [19]. However, with the development of VLSI semiconductor circuits, logic density has increased so dramatically that it becomes practical to transfer some processing capabilities to memories. Furthermore, VLSI technology makes the production of larger CAM economically feasible, thus increasing its applications arena. For example, Advanced Micro Device, Inc. has marketed a Content Addressable Data Manager which contains 1 Kbytes of CAM and internal logic to perform sorting and searching operations with 100 ns cycle time. Moreover, the chips are cascadable to 256 Kbytes RAM. The impact of CAM on computer architectures is witnessed by the active research in content addressable processors, noticeably a new content addressable parallel processor design for picture processing by Foster [33]; Titanic, a VLSI based content addressable parallel array processor by Weems et al [34]; and vector associative processor by Berkovich and Pullen [35]. The performance of a highly parallel system has been measured by Parkinson and Liddel [19, 36].

The fully parallel CAM configuration allows one character at a time to be placed on the global data bus; each cell then compares its content with the character simultaneously. This is essential to the implementation of the longest match searching algorithm discussed in the previous section. However, a fixed-sized string or character search is insufficient; we are interested in the consecutive matches from consecutive locations that form a variable-length match. Therefore, we need to preserve the match result of the previous compare cycle, so that each CAM cell can look back one cycle to see if its left neighbor had a match. This requirement brings about the modification to the basic CAM cell.

3.1.3 Byte Associative Content Addressable Memory

In byte associative CAM, there are two distinctions from the basic CAM. The first is that each CAM cell passes the delayed match result (via a flip flop) to its right-hand neighbor and receives the delayed match result from the left-hand neighbor. The second difference is that the HIT signal is the logical AND of the content match and the match result of the left neighbor in the previous cycle. Figure 3.4 shows the change in the basic CAM cell model. As can be seen, $HIT_{n-1,t-1}$ is the match result of the left neighbor in the previous compare cycle, and $HIT_{n,t-1}$ is the match result of the CAM cell in the previous compare cycle. The Delay is a structure that takes an input signal and outputs the same signal a clock cycle later.

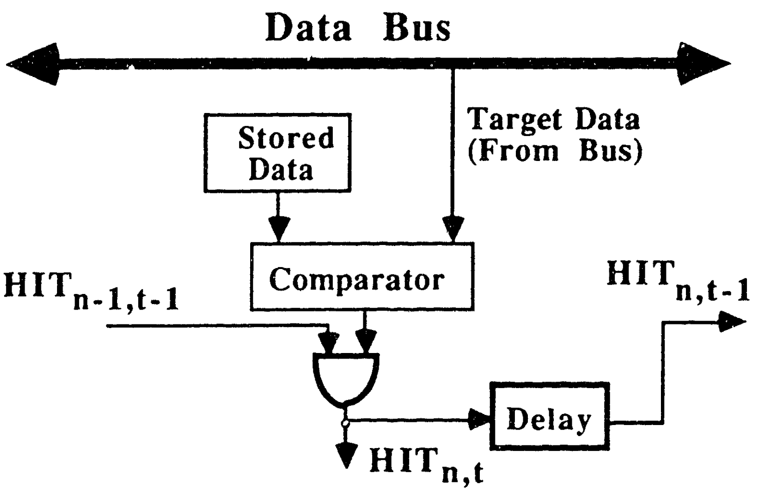


Figure 3.4 The Modified CAM Cell Model

Figure 3.5 shows the change in the CAM structure. Now there is interconnection between two neighboring CAM cells. The cell-level HIT signals look the same; the logic that generates them changes and is hidden inside. These modifications only implement the two criteria for a match; more control logic is required to find the longest match.

The idea of byte associativity is similar to the general purpose CAM architecture proposed by Adams [18]. In that design, an Address Selector feature allows only certain set of the CAM cells to participate in the matching activities. For instance, it is possible to specify that only the even-numbered cells be engaged in the search. The match results of the participating CAM cells are then stored in the Match bit cells in each CAM cell. Next, all the odd-numbered cells are allowed to compare. In the Linked Associative Mode in this example, only if the Match bit of the even-numbered cell were set and the odd-numbered cell's content matched the presented data would the Match bit for that odd-numbered cell be set. For address priority encoding, the Match bits of the even-numbered cells must be explicitly turned off before encoding can take place. This associative capability makes multiple-word pattern matching possible. However, the word width needs to be fixed and known in advance so that appropriate Address Selector can be set and the Match bits be cleared. For ZL77 encoding, we do not and can not know the match length beforehand, nor do we know where the match starts, so Adams' general purpose CAM architecture can not be used to find the longest match.

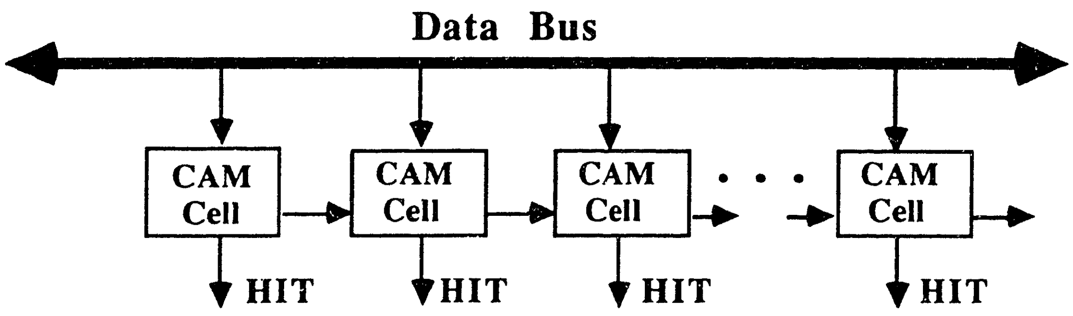


Figure 3.5 The Modified CAM Structure

3.1.4 VLSM Functional Description

The VLSM, as mentioned in Section 3.1, is a structure that utilizes the modified CAM to encode ZL77 codewords. It is helpful to treat the VLSM on a functional level before going into the detail of implementation. In ZL77 encoding, we are interested in obtaining a sequence of fixed-length codewords as we parse the input strings. This task can be accomplished efficiently with the VLSM algorithm as shown in Section 3.1.1. In addition to searching, we would like to update the history buffer such that it always includes the N most recent characters before a new string is encoded. Intuitively, four discrete operations are required for the encoding and updating processes using the byte associative CAM, namely INIT, COMPARE, OUTPUT, and UPDATE.

INIT is used only once for each string encoding. If one recalls the example from Section 3.1.1, the INIT command is used to force a hit from every CAM cell to set up for the first COMPARE cycle; it does not clear the cell contents. In a COMPARE cycle, a character of the input string is put on the data bus for every CAM cell to check. If at least one match occurs, the COMPARE cycle continues with the next character of the string. The codeword which consists of Index and Length components is output during the OUTPUT cycle. Finally, a character is inserted into the history buffer by an UPDATE command. Therefore, the VLSM is conceived to have the set of inputs and outputs as shown in Figure 3.6.

The inputs to this functional block consist of an 8-bit data bus DATA[7:0], a 10-bit address bus ADDRESS[9:0], an ENABLE line and two select lines S1, S0. DATA[7:0] contains the character to be compared or updated. ADDRESS[9:0] specifies the history buffer location into which the character is inserted. ADDRESS[9:0] is 10-bit wide because the thesis uses a 1024-character history buffer as mentioned in Section 2.2.4. Therefore, 10 bits are required to uniquely address any of the 1024 locations.

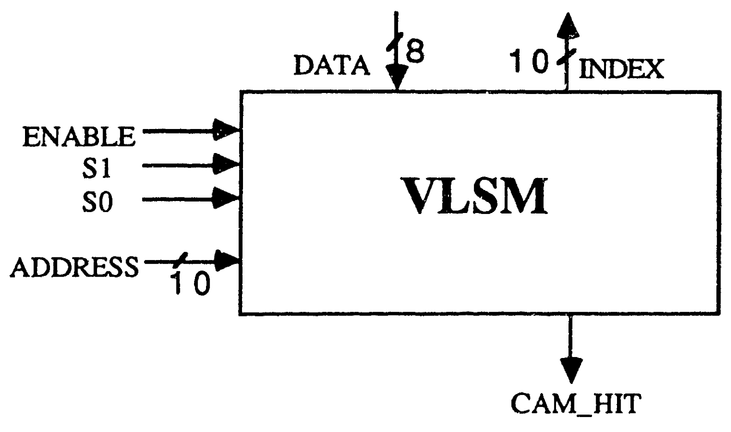


Figure 3.6 VLSM Functional Block Diagram

When ENABLE is asserted, the combination of the two function mode select signals S1 and S0 will generate appropriate control signals inside the functional block to execute one of four possible commands. Table 3.1 lists the VLSM functional mode control.

ENABLE	S1	S0	Function
0	x	x	NOP
1	0	0	INIT
1	0	1	COMPARE
1	1	0	OUTPUT
1	1	1	UPDATE

Table 3.1 VLSM Functional Modes

The outputs consist of INDEX[9:0] and CAM_HIT. INDEX[9:0] is one of the addresses of CAM words which have a match. It is normally latched inside the functional block in Figure 3.6 and is output-enabled when OUTPUT mode is selected. CAM_HIT is asserted by the VLSM when at least one CAM word has a match. This output signal is monitored by an external controller to determine if the COMPARE cycle should be continued. From now on we assume that the external controller is implemented with a finite state machine (FSM).

The control of the VLSM is straightforward. Let the 1024-character history buffer be stored in the byte associative CAM inside the VLSM. To find the longest match, the FSM first selects the INIT command. At the next cycle, the first character of the input string is driven onto the data bus, and the COMPARE mode is selected. If CAM_HIT is asserted, the next character will be driven onto the data bus and another COMPARE command is issued. The FSM repeats this step until CAM_HIT is not asserted after a COMPARE cycle. By then, the index for the of the last character of the longest matching string in the CAM is ready, so the FSM would issue OUTPUT to fetch the index. To update the history buffer, the FSM drives the character onto DATA[7:0] and selects the UPDATE mode. Note that CAM_HIT also clocks a small counter which keeps count of the match length. Figure 3.7 is an extension to Figure 3.1 and should clarify the sequence of operations required to carry out the encoding and updating procedures.

In this example, it is assume that the history buffer pointer points to location 5. Therefore, the characters that were matched in the previous set of COMPARE cycles are removed from an input FIFO and are inserted into the history buffer starting at this location during the UPDATE cycles. The changes in history buffer are shown by the bold characters.

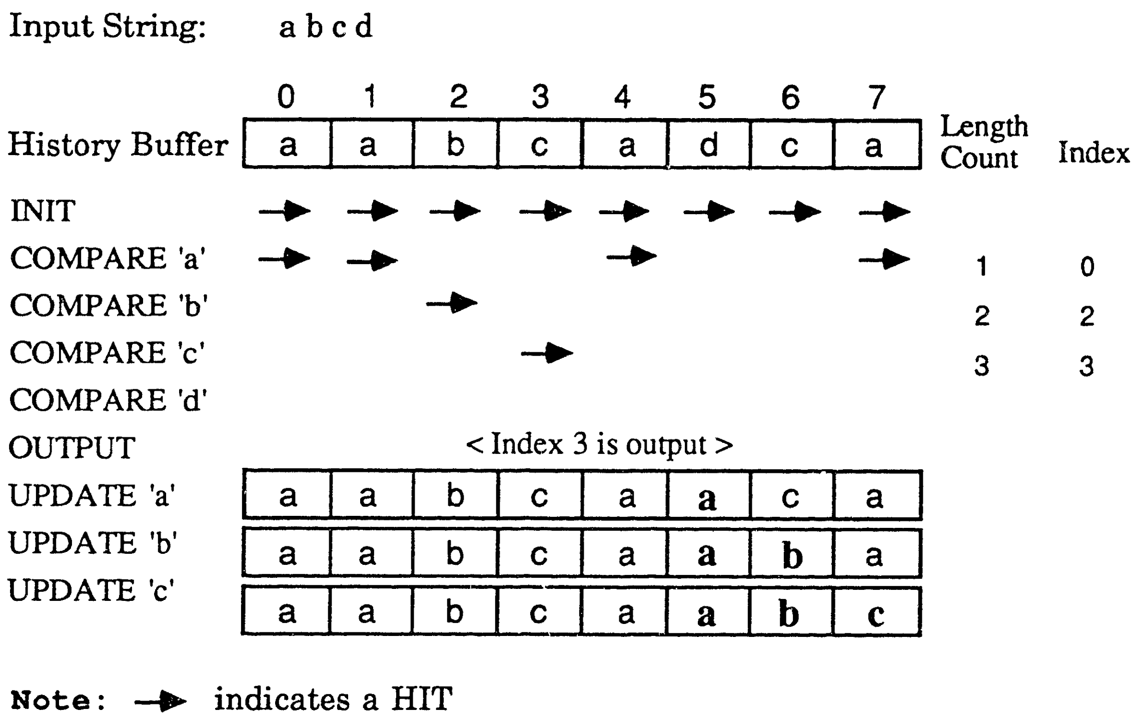


Figure 3.7 Encoding and Updating Processes

Finally, Table 3.2 shows the states of the inputs and outputs of the VLSM functional block corresponding to each cycle.

INSTRUCTION	S1	S0	DATA	CAM_HIT	INDEX	ADDRESS
INIT	0	0	x	x	x	x
COMPARE 'a'	0	1	'a'	1	x	x
COMPARE 'b'	0	1	'b'	1	x	x
COMPARE 'c'	0	1	'c'	1	x	x
COMPARE 'd'	0	1	'd'	0	x	x
OUTPUT	1	0	x	x	3	x
UPDATE 'a'	1	1	'a'	x	x	5
UPDATE 'b'	1	1	'b'	x	x	6
UPDATE 'c'	1	1	'c'	x	x	7

x means no valid value

Table 3.2 VLSM Functional Block I/O States

Implementation of the VLSM

Figure 3.8 discloses the internal structures of the VLSM functional block. The byte associative CAM array, called CAM Word Array for short, is the major component. It stores the history buffer and performs parallel comparison. Other important blocks are the address encoder, address decoder, data buffer, and supporting logic such as the flip flops, decoder, and multiplexors.

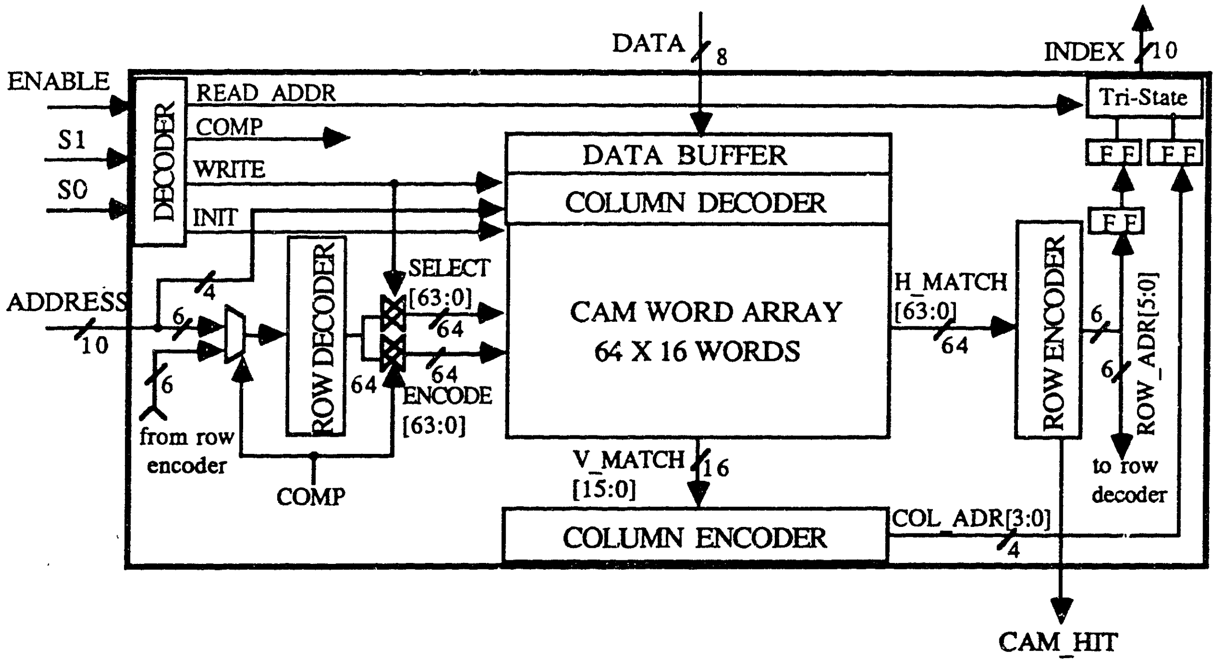


Figure 3.8 VLSM Internal Block Diagram

The Function Mode Decoder on the top-left corner, when enabled by ENABLE, takes S1 and S2 as inputs and asserts one of four lines: INIT, COMPARE, WRITE, or

READ_ADDR. INIT goes through every CAM word in the CAM Word Array. As its name suggests, this mode is selected once per codeword generation.

COMP is asserted in the COMPARE cycle. It selects the ENCODE lines to take on the outputs of row decoder to select a row for address encoding purpose. Section 3.1.7 will explain address priority encoding in greater detail. COMP also multiplex the inputs to the Row Decoder between ADDRESS[9:4] and the output of Row Encoder. WRITE is asserted in the UPDATE cycle. It enables the SELECT lines and Column Decoder to address one CAM word for a data write. Finally, READ_ADDR serves as the output-enable for the tristate in the top right corner of Figure 3.8 in the OUTPUT cycle.

Not shown in Figure 3.8 are two non-overlapping clock signals Phi1 and Phi2 which go through each CAM word.

ADDRESS[9:0] is broken down into 4-bit going into Column Decoder and 6-bit going into Row Decoder. The Row Decoder, when enabled, asserts one of 64 lines. If WRITE is asserted, these lines will be connected to 64 SELECT lines which act as row enable during the write cycle. If COMP is asserted, the 64 ENCODE lines will assume the outputs of the Row Decoder.

CAM Word Array produces 64 horizontal H_MATCH lines and 16 vertical V_MATCH lines which go into Row Encoder and Column Encoder, respectively. The Encoders generate six and four address lines which are eventually combined to form a 10-bit Index. The Row Encoder is also responsible for generating the CAM_HIT signal.

3.1.5 Byte Associative CAM Word Array

Figure 3.9 shows an example of byte associative CAM word cells arrangement and interconnection in the CAM Word Array. There are 64 x 16 (= 1024) word cells in the proposed implementation, but only 3 x 3 (= 9) case is shown in Figure 3.9 due to space constraint. This two-dimensional arrangement is desirable for a compact VLSI layout, especially for a large number of word cells.

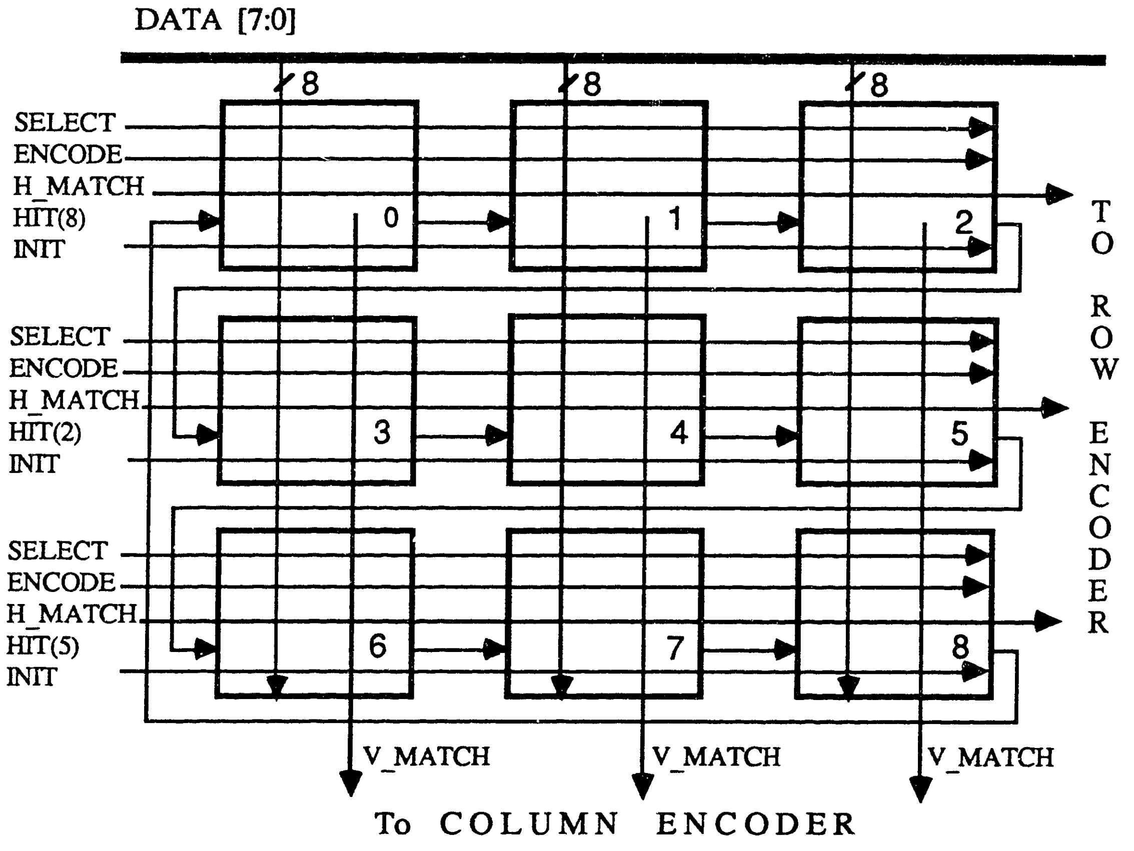


Figure 3.9 CAM Word Array

As can be seen, DATA[7:0] reaches every word cell. The SELECT lines are used only during the UPDATE cycle and are high impedance during other times. The SELECT lines are connected to the outputs of the Row Address Decoder during UPDATE cycle to enable one row for data write. The INIT is the output of Function Mode Decoder and is fed into every word cell.

The H_MATCH lines for each row and V_MATCH lines for each column are wire-ORed with each CAM word on the same row or column and are precharged high during phi1. Each H_MATCH line is pulled low if at least one CAM word in that row has a match during phi2. Section 3.1.6 will explain exactly how H_MATCH, V_MATCH, and ENCODE lines are used to perform priority address encoding.

The unusual interconnection is the HIT lines. As can be seen from Figure 3.9, the HIT output from one CAM word is the input to its right-hand neighbor. In addition, the HIT from the highest-numbered word cell is an input to the lowest-numbered word cell. For example, the HIT output of word cell 8 is connected to the HIT input of word cell 0. In order to maintain consistent addressing, the HIT from the last word in the row has to be routed to the first word in the next row; for a large array, the wire could be very long, resulting in undesirable propagation delay. However, the wire length can be minimized if the array can be "folded" horizontally.

3.1.5.1 CAM Bit Cell

The most primitive CAM structure is the CAM Bit Cell shown in Figure 3.10. It has a standard six-transistor static RAM cell topology in the upper portion, with three additional transistors that carry out the equivalence logic function in the bottom half. If high density is desired, the CAM Bit Cell can be implemented with a four-transistor dynamic RAM topology patented by Mundy and improved by Wade [29-31].

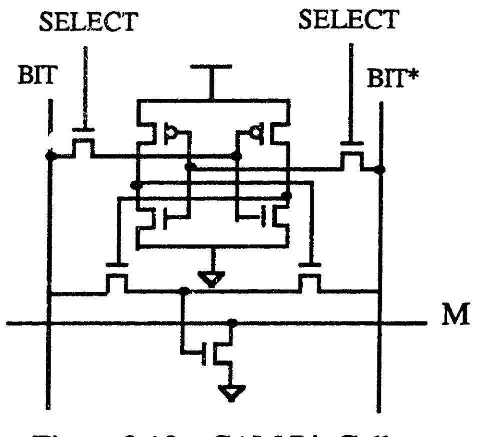


Figure 3.10 CAM Bit Cell

In the proposed implementation, the wire M is precharged high during phi1. During a match cycle, if the data on the Bit line is the same as the bit value stored in the cell, M will remain high; otherwise, M will be pulled low to indicate a mismatch. The SELECT line, when asserted, will cause the value on the BIT and BIT* lines to be written into the storage element.

Next, eight of these Bit Cells are grouped to form a CAM Match Cell as shown in Figure 3.11. Notice that the line M is wire-ANDed in the 8-bit construct. Therefore, if at least one Bit Cell detects a mismatch, the whole M line will be pulled low.

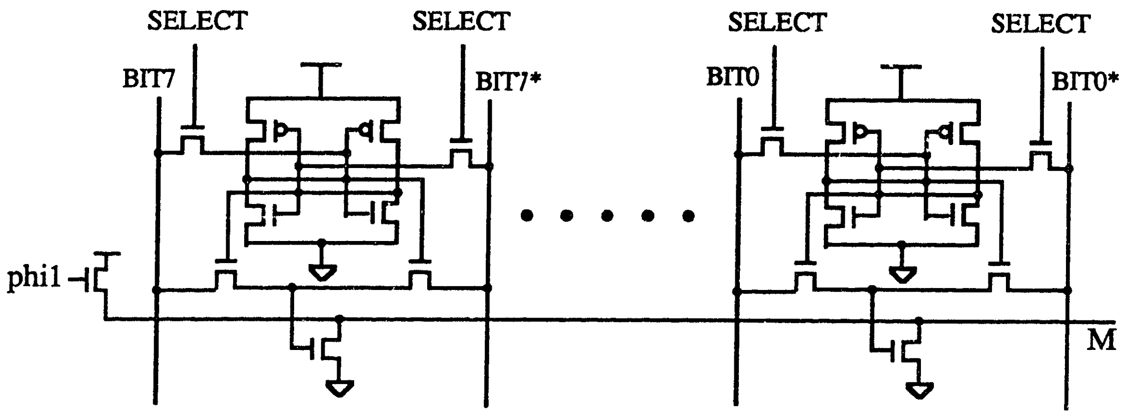


Figure 3.11 CAM Match Cell

3.1.5.2 CAM Word Cell

The CAM Word Cell, which makes up the CAM Word Array, is the next higher level of abstraction. Figure 3.12 shows the CAM Word Cell's internal structure. As expected, the CAM Match Cell is the largest component of the Word Cell; the rest are random logic and control lines.

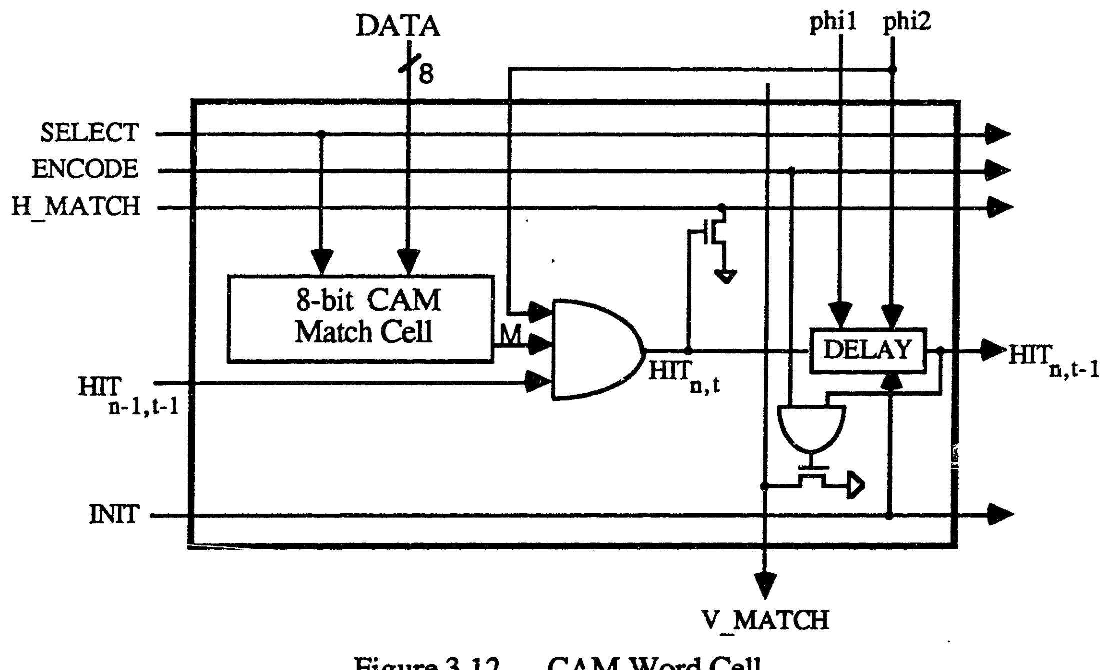


Figure 3.12 CAM Word Cell

As the logic shows, $HIT_{n,t}$ is the AND of phi2, M and $HIT_{n-1,t-1}$ which is the match result of the left Word Cell in the previous COMPARE cycle. $HIT_{n,t}$, if positive, will pull the

H_MATCH line down, indicating a match for this Word Cell. $HIT_{n,t-1}$ is the one-cycle-delayed $HIT_{n,t}$ signal; in finding the longest matching string, a current match is allowed only if a CAM Word Cell's left neighbor had a match in the previous COMPARE cycle and the Cell's content is the same as the data. Remember that the input string is presented on the data bus only one character at a time. Somehow we need to create byte associativity by using the flip flop to save the previous match result.

However, the first character match of the input string does not have to obey the left-neighbor-had-a-match rule because no COMPARE cycle has taken place yet. This is why the INIT command has to be issued first whenever a new string search begins. Figure 3.13 shows the inside of the Delay block. When INIT line is asserted high, the high value is fed into the flip flop rather than the value of $HIT_{n,t}$ in the Word Cell. In essence, this forces the $HIT_{n,t-1}$ lines of each Word Cell to be high by the time the COMPARE cycle takes place, so that all words that have matches can indicate so.

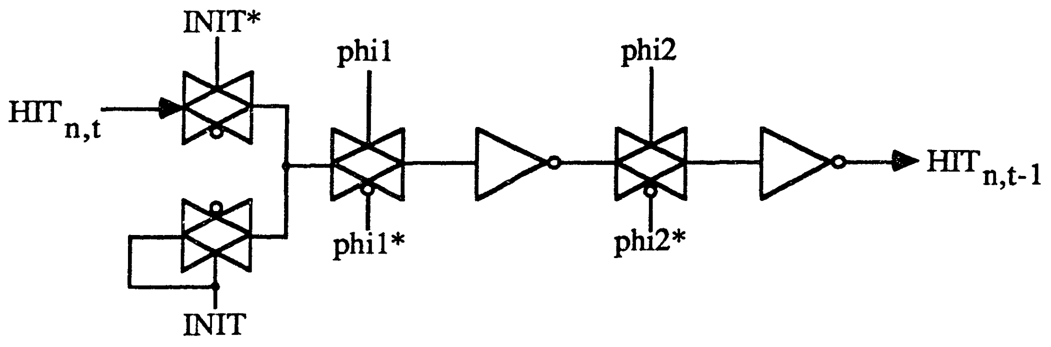


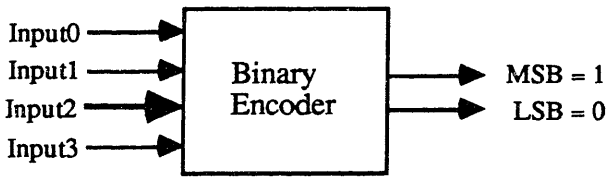
Figure 3.13 The Delay Structure

3.1.6 Address Priority Encoder

This section describes the problem of address priority encoding and surveys the methods in which people have used in applications with different speed/cost requirements. A novel, proposed implementation of the Address Priority Encoder will be presented in the end.

3.1.6.1 Address Priority Encoding

Address priority encoding is fundamental to VLSM and CAM in general because outputing a single address where the same data is found is one of the CAM's attributes. In the digital world, an address usually takes a binary representation. For example, an address of six in a four-bit binary representation will be 0110. For N possible addresses, $\log_2 N$ bits are necessary to specify each address. It is trivial to encode a binary address if only one out of N inputs is active, as shown in Figure 3.14 (a). This example shows that Input 2 is active, so binary 10 (= decimal 2) is encoded and output. The encoder can simply be implemented with random logic. However, it becomes not so straightforward if more than one inputs are active at a time, because conflicts need to be resolved first before binary encoding can occur. In other words, a prioritizer is needed. Figure 3.14 (b) shows a model for a address priority encoder.



N INPUTS ==> (LOG_2 N) OUTPUTS

Figure 3.14 (a) Normal Binary Address Encoding Model

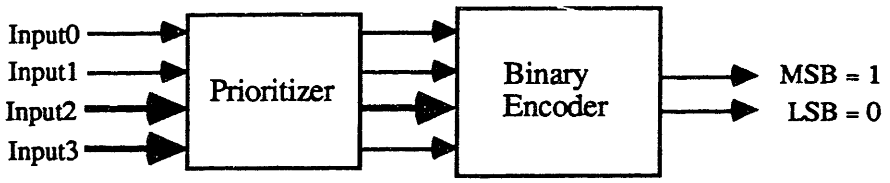


Figure 3.14 (b) Priority Address Encoding Model

One way to prioritize the inputs is by their locations. For example, the prioritizer can be designed to allow the input that is the closest to the top or the bottom to pass the prioritizer and inhibit the others from going through. Thus, only one of the outputs of the priori er is active, and a single address can then be encoded.

One common and the most cost effective solution to implement the prioritizer is cascaded ripple or daisy chain through N OR or AND gates, where N is the number of inputs, or respondants, to the encoder. Figure 3.15 illustrates this method. The outputs of the higher gates inhibit the lower gates. The advantage is that only N identical two-input gates plus N inverters are required; since the interconnection pattern repeats, this arrangement is great for a compact VLSI layout. However, the propagation delay is equal to the sum of N gate delays. If a system requires high throughput, this ripple inhibit chaining method will not be ideal.

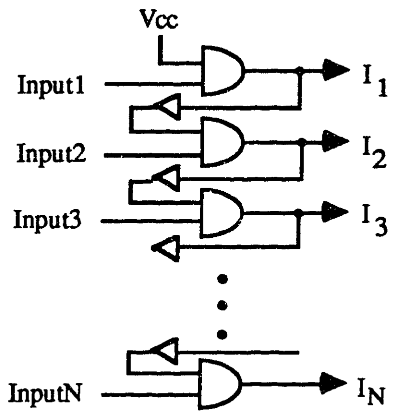


Figure 3.15 Ripple Chain Prioritizer

The other extreme is the use of lookahead, whereby the top gates directly inhibits all the lower gates. This means that the topmost gate uses no gate; the next one uses a two-input gate; the gate below uses a three input gate; and the lowest gate will have N inputs! This scheme is very fast because the latency is the only the gate delay of the N-input lowest gate. However, it is very expensive and not ideal for VLSI layout.

Priority Encoding of Two-Dimensional Array

There is an additional complication for priority encoding of a two-dimensional array. Suppose we have 16 CAM cells arranged in a 4x4 fashion shown in Figure 3.17. Those that are marked with 'X' indicate that they have a match. Therefore, the match lines associated with those cells are activated as indicated by the bold lines in the Figure. Let us encode the cell which has the smallest address among the matching cells. By inspection, cell 2 should have been encoded.

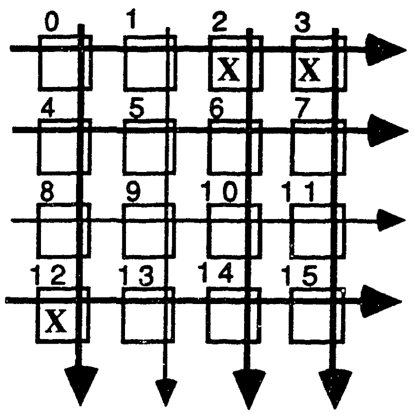


Figure 3.16 Two-Dimensional Array

In order to uniquely address a cell, we need to specify the row and column numbers. In other words, we encode row and column addresses separately. In this case, cell 2 belongs to row 0 and column 2. By looking at the row match lines, we find that rows 0, 1, and 3 have matches. According to the priority rule, row 0 will be encoded. At the same time, we look at the column match lines. We find that columns 0, 1, 2, and 3 have matches. Therefore, column 0 is encoded. When we combine them, the result is wrong! Cell 0 does not have a hit, yet the address for it is encoded. This unfortunately implies that the row and column address can not be encoded simultaneously; some restrictions must be applied to ensure that the correct address is encoded.

Let us change the procedure. Suppose we encode the row address first, so we get row address of 0, as before. Now, suppose we allow only the column matches that occur in row 0 to enter the column encoder while inhibit the column matches in other rows. In row 0, columns 2 and 3 have matches, and column encoder should generate an address 2. When we combine row and column addresses, an address of 2 is formed, which is correct.

The procedure described above explains the presence of the ENCODE lines found in Figures 3.9 and 3.12. In Figure 3.12, the H_MATCH lines enter the Row Encoder during the COMPARE cycle. The encoded row address is then routed to the Row Decoder. In the same cycle, the outputs of Row Decoder are connected to the ENCODE lines, thereby only one ENCODE line is asserted (i.e., one row is enabled). The V_MATCH in a Word Cell is pulled low only if the ENCODE line for that row is asserted and HITn is also asserted. Without this restriction, the correct lowest-numbered Word Cell address can not be encoded. Figure 3.17 summarizes the address encoding sequence.

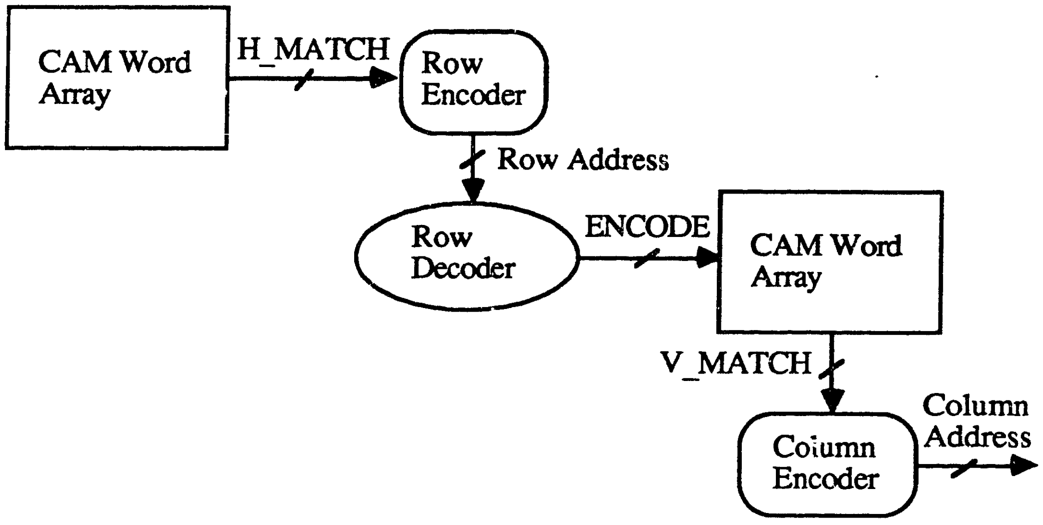


Figure 3.17 Priority Encoding Datapath in the VLSM

Just a quick reminder: each horizontal H_MATCH line is precharged high during phi1 and is wire-ORed with the word cells on the same row. It is the same for each vertical V_MATCH line except that it is wire-ORed with the Word Cells on the same column.

3.1.6.2 The Implementation

In order to uniquely and accurately encode an address, row address encoding has to take place first. The resulting row address is then routed to the Row Decoder, whose outputs are routed to the ENCODE lines which enable a row to propagate its vertical V_MATCH lines down to the Column Address Encoder to encode the vertical address. The Address Encoder is responsible for resolving multiple matches in order to encode an unique address. There are several ways to arbitrate simultaneous responses; the one implemented in the design is priority encoding, i.e., the CAM word that has the lowest address is encoded.

The resulting encoder is constructed in a grid fashion as shown in Figure 3.18. Its regularity is particularly suitable for VLSI implementation. In addition, it is a reasonably fast circuit, although it takes more area than the ripple chain approach. The tradeoff is worthwhile performance takes the higher priority than area cost. The design shown in Figure 3.18 is innovative in that it combines prioritization and binary encoding in the same structure.

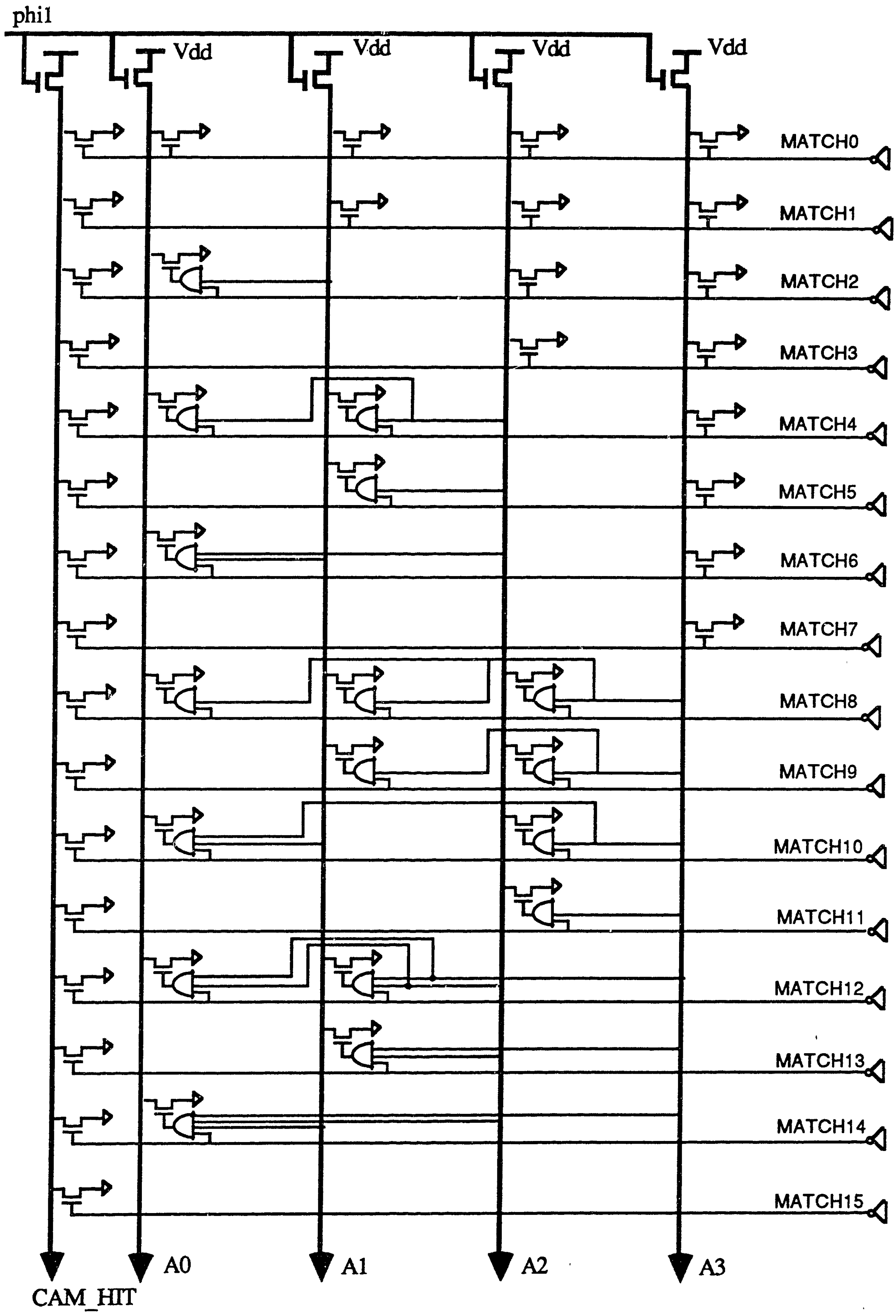


Figure 3.18 Priority Address Encoder

The ADDR lines are running vertically and are precharged high during phi1. In order to exploit the idiosynchrocy of the binary representation, the most significant bit (MSB) of the address (i.e., A3 in Figure 3.18) is encoded first. Then the random logic can be designed to depend on the value of the more significant address lines. For example, the logic MATCH14 on A0 depends on A3, A2, A1, and MATCH14. A more general case is that if any of the top eight MATCH lines (encoding binary addresses ranged from 0 to 7, whose MSB's, i.e., A3, in the four-bit address are 0) is asserted, the logic for the bottom eight MATCH lines will be turned off since it is ANDed with A3. With the appropriate placement of logic, the construct is indeed able to encode the lowest address in the case of multiple responses.

Note that the MATCH lines from the CAM Word Array are buffered by inverters before they are evaluated. There are two reasons for doing so. First, the MATCH lines are wired-ANDed, so they need buffers to sustain their logic levels. Second, for the encoding logic to work, the MATCH lines are assumed positive logic. But in the CAM Array, MATCH lines are pulled low if matches occur. The use of inverters will reverse them to the correct polarity.

The leftmost column in Figure 3.18 is responsible for generating the CAM_HIT signal. It is again a wire-ORed structure precharged high during phi1. If any one of the MATCH lines is high, CAM_HIT will be pulled low. This structure is placed in the Row Address Encoder ONLY.

3.1.7 Timing Analysis

Of the four operation modes: INIT, COMPARE, OUTPUT, and UPDATE, the COMPARE cycle has the highest latency and thus it governs the clock frequency under which the VLSM can run. This is not surprising at all since both the matching and address priority encoding, the core processes of the VLSM, are accomplished in this cycle. Let us examine the events that occur in this stage and see if the latency can be reduced.

First, there is a delay for each CAM Match Cell to generate a valid M signal (refer to Figure 3.11) which is logic high if the cell content matches the presented data, and logic low otherwise. Next, $Hit_{n-1,t-1}$[1], M, and phi2 are ANDed to determine the signal X (refer to Figure 3.12) which, when positive, will pull down the H_MATCH line[2] that is wire-ANDed with every cell in the same row. For a large array, the line capacitance is approximately the sum of each pulldown gate capacitance, so it could take a relatively long time to discharge.

Next, the Row Address Encoder takes these H_MATCH lines as inputs and produces a binary address corresponding to the lowest-numbered H_MATCH line that was pulled down. The address then enters the Row Decoder. After certain delay, the ENCODE lines are connected with the outputs of the Row Decoder. Note that only one ENCODE line is asserted. The COMPARE cycle continues with the generation of V_MATCH lines. On the row enabled by ENCODE, the match result is ANDed with ENCODE. If positive, the V_MATCH line for that column will be pulled low. Finally, the column address is encoded after valid V_MATCH lines enter the Column Address Encoder. Figure 3.17 shows the sequence of events described in this paragraph, and Figure 3.19 summarizes the propagation delays for the COMPARE cycle.

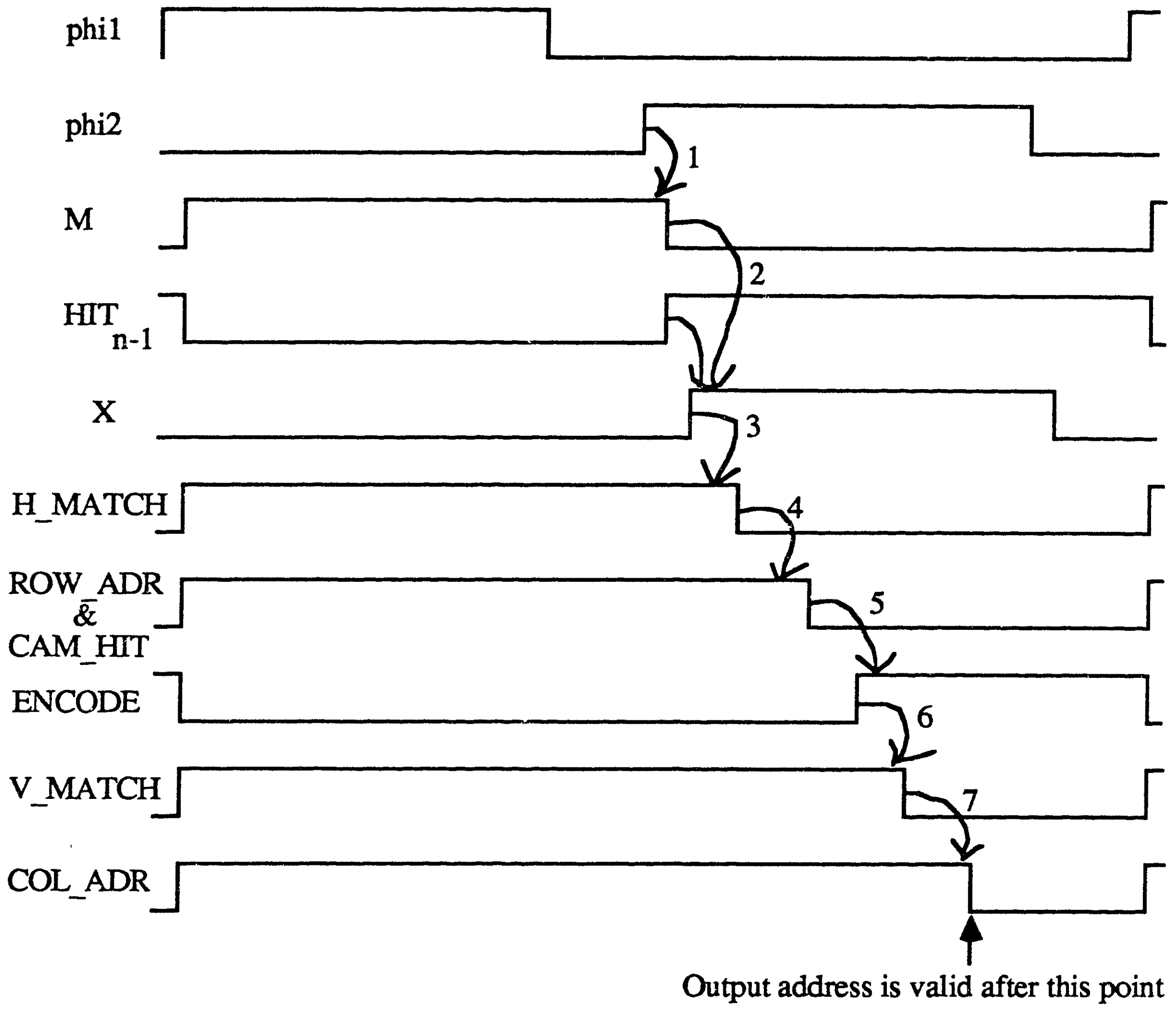
It is clear that address priority encoding consumes the most time. If we take a step back and recall the VLSM algorithm, we realize that we don't really need to find out the address until we are finished with the COMPARE cycles. We encode the address during each COMPARE cycle just because we are not sure when the a MISS will occur. Therefore, we can take advantage of this fact to reduce the latency by pipelining the address encoding process. For example, we can encode the row and column addresses in two different clock cycles instead of in the same cycle. We can start another COMPARE cycle (if necessary) as soon as the row address has been encoded and CAM_HIT asserted. The generation of ENCODE and subsequent column address encoding can take place in the next COMPARE cycle, at the same

[1] This is the match result of the left cell in the previous cycle.

[2] Remember that H_MATCH lines are precharged high during phi1. Evaluation occurs during phi2.

time the new row address is being generated. The events can be broken in half as shown in the bottom of Figure 3.19.

Note: M, H_MATCH, ROW_ADR, CAM_HIT, V_MATCH, and COL_ADR lines are precharged high during phi1. ENCODE is normally low.



First Half
1. M line pull-down delay
2. AND gate delay
3. Horizontal MATCH line pull-down delay
4. Row Address Encoder delay

Second Half
5. Row Address Decoder delay
6. AND gate + Vertical MATCH lines pull-down delay
7. Column Address Encoder delay

Delays 4, 5, and 7 are believed to be more significant.

Figure 3.19 COMPARE Cycle Timing Diagram

Figure 3.20 shows an example of the pipelining process. After COMPARE cycle 1, we learn that CAM_HIT is asserted, so we continue with another COMPARE cycle. After Cycle 3, CAM_HIT is not asserted. Therefore, we issue an OUTPUT command to read out the address in Cycle 4. Note that the address we want is the address of the CAM cell that had a hit in Cycle 2, not Cycle 3. As a result, we must buffer at least two most recent row addresses. This explains why there are two layers of flip flops (FF's) for row address in the upper righthand corner of Figure 3.8. Column address only has a single layer of FF because the row address is always produced one clock cycle ahead of the column address.

Another requirement to make pipelining work is to buffer the match results. This is reflected in Figure 3.12 in which $HIT_{n,t-1}$, the delayed match result, rather than HIT_n, is used as an input to the AND gate that produces the signal to pull down V_MATCH. This does not incur any extra hardware cost because the FF is there already.

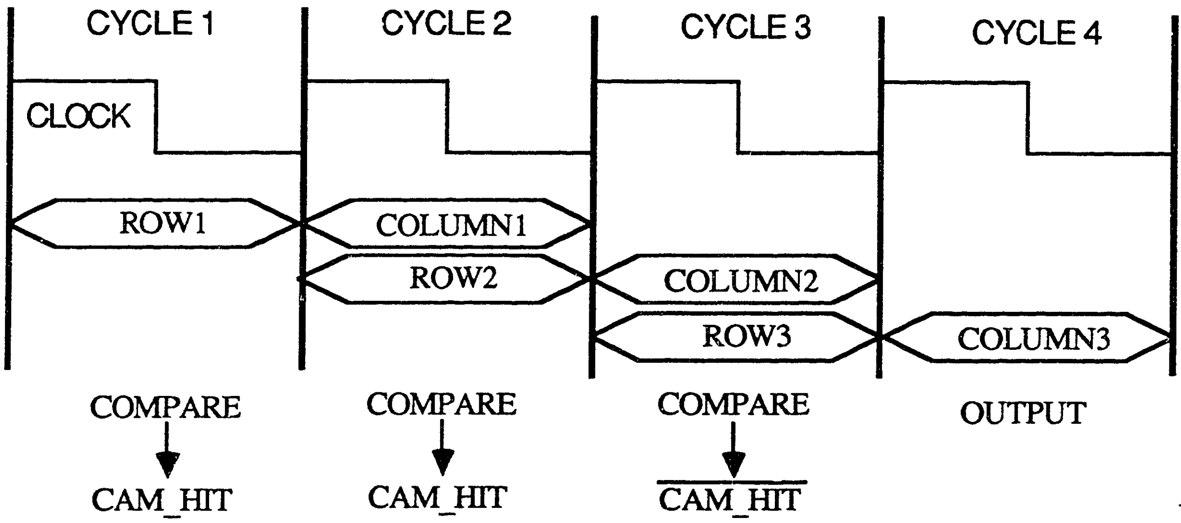


Figure 3.20 Pipelining of Row and Column Address Encoding

In short, pipelining the address encoding process effectively cuts the latency of the COMPARE cycle in half. As Figure 3.20 shows, the row address encoding and of a given cycle and the column address encoding of the previous cycle can take place at the same time. Surprisingly, no additional hardware is required except the extra set of FF's to buffer the row address.

Other Blocks

The Row and Column Address Decoders, Data Buffers, and Function Mode Decoder are standard logic so they will not be discussed here.

3.1.8 Design Evaluation

Since this thesis concentrates on the architectural and logic level designs only, we can only estimate the area and throughput. While the numbers will not be exact, they at least give us some insight on the cost and performance of the VLSM device. We will compare the area and throughput against those of the hashing data structure because it takes the least amount of area among the data structures surveyed in Chapter 2.

3.1.8.1 VLSM Area Estimate

In estimating the areas of the major blocks of the VLSM, a process is assumed. Figure 3.21 shows the floor plan and the dimension of the VLSM.

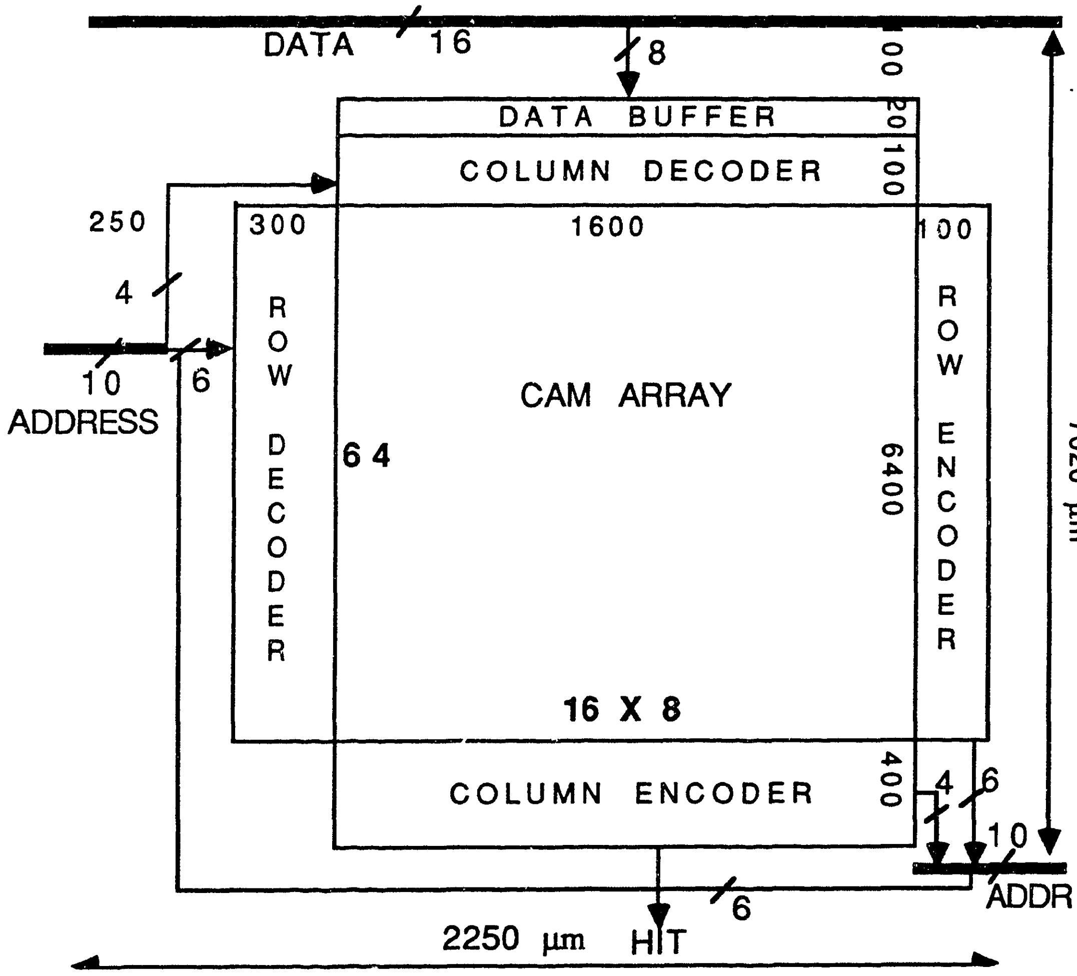


Figure 3.21 VLSM Floor Plan and Area Estimate

The layout of the basic components is similar to that of a RAM, with Data Buffer, Row Decoder and Column Decoder on the sides. However, sense amplifiers are not included since reading from the CAM cells is not necessary for the ZL77 encoding application. The CAM Word Array is arranged in 64 by 16 fashion for a total of 1024 Word Cells; naturally it is the largest component of the VLSM.

The non-bold numbers in Figure 3.21 are dimensions in μm. Note that 1 mil = 25.4 μm. All in all, the VLSM structure is estimated to take 276 mil x 88 mil = 24,482 mil^2. This is a little more than a 4 Kbytes static RAM would take[1]! Recall from Chapter 2 that the hashing method needs about 5.25 Kbytes RAM for the same 1 Kbytes history buffer. Therefore, as long as the performance of the VLSM, which we will analyze in the next section, is better than hashing, then the VLSM structure is feasible and more efficient on an area basis.

6.1.8.2 VLSM Throughput Estimate

Using the VLSM, the number of cycles required to produce a codeword for each string is directly proportional to the match length. The following summarizes the cycles necessary per string, where n is the match length of any given string:

1 INIT + (n+1) COMPARE + 1 OUTPUT + max(n,1) UPDATE= max{(2n+3), 4} cycles.

For example, to encode a string whose longest match in the history buffer is 3, only 9 cycles are required. Of these cycles, INIT, OUTPUT, and one COMPARE cycles are fixed overhead; this is how the 3 comes about in the expression. We need the extra COMPARE cycle to find out that there is a miss. The overhead becomes significant in the case of no match, as 4 cycles are consumed to encode a character. An UPDATE is still necessary even there is no match; this is why the expression max(n,1) is there. As the match length increases, the encoding speed will approach two cycles per input character.

Some assumptions must be made before the throughput can be estimated. First, we assume that the compression ratio is 2. Since the codeword length used in the implementation is 14 bits, it is reasonable to say that the VLSM on average replaces every 28 bits of source data with a 14-bit codeword. 28 bits are equivalent to 3.5 8-bit characters, so we assume that the average match length is 3.5 characters. Putting this number for n in the above formula, 10 cycles are required to encode 3.5 input characters. We can extend this to any number of data samples. In general, let N be the number of source characters. Then the number of cycles needed to encode N input characters follows the formula: (N/3.5) * 10. For example, to encode 100 input characters, it will need (100/3.5) * 10 = 285 cycles. Based on these

[1] Assuming 1μm, two-layer metal CMOS technology, 4 Kbytes SRAM will take about 1160 mil x 140 mil = 22,400 mil^2

assumptions, the plot in Figure 3.22 (a) is derived as a function of the VLSM cycle time, or clock period:

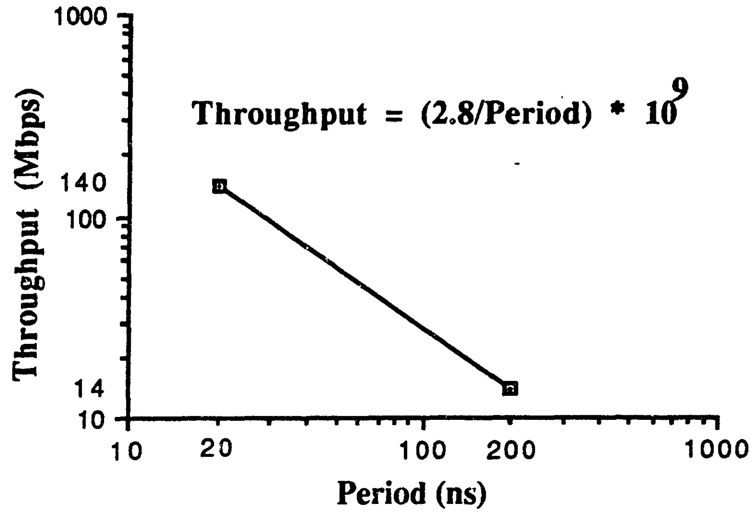


Figure 3.22 (a) VLSM input throughput vs. VLSM cycle time

The plot suggests that even if the cycle time is 200 ns, the VLSM can still sustain 14 megabits per second (Mbps) input rate!! Initial estimate based on the design reveals that the VLSM cycle time is about 40 ns. This will provide a 70 Mbps input throughput. In other words, the VLSM can sustain an input rate of 8.75 million characters per second. These valuse are derived from the maximum possible performance available from the VLSM device iteself. The actual throughput of the Data Compression Engine depends on applications and the ability to keep the VLSM active.

On the other hand, the hashing method requires about 225 cycles per character. Using a 40 ns cycle time, this translates into 0.11 million input characters per second. Therefore, the VLSM is about 81 times faster hashing. Appendix A lists the code from which the hashing cycle time estimate is derived.

It is interesting to see how changes in compression ratio will affect the VLSM input throughput. Figure 3.22 (b) shows a plot for cycle time fixed to 50ns and 100ns, respectively, while the compression ratio varies from 1.5 to 2.3. Fortunately, the curves are relatively flat; this demonstrates that even under low compression ratio, the input throughput is still very high. However, the curves suggest that higher compression ratio results in higher throughput.

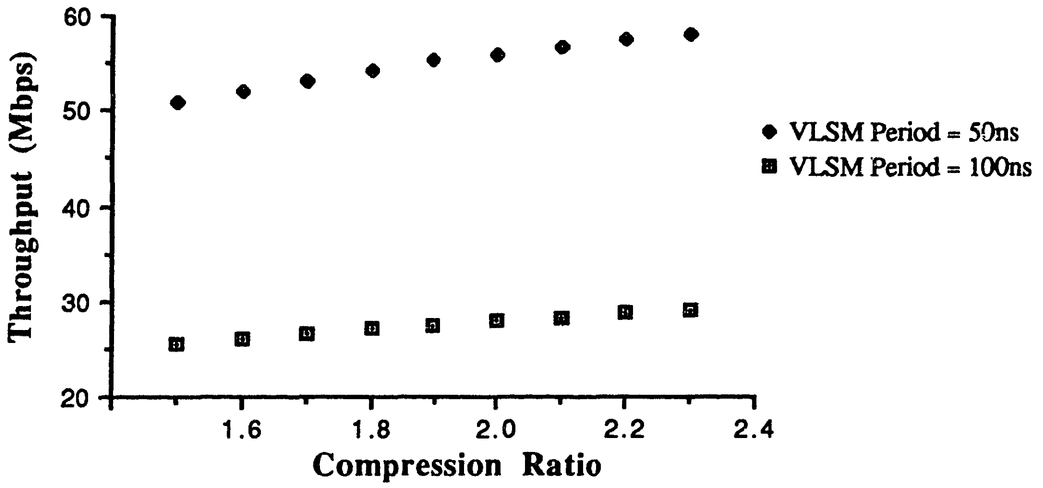


Figure 3.22 (b) VLSM input throughput vs. compression ratio

By combining the VLSM with other elements of a DC Engine on the same chip, the input throughput derived in this section may be achieved. These elements are discussed in the following sections.

3.2 THE BIT PACKER

The goal of the Bit Packer is to pack an arbitrary number of bits into a byte or multi-byte quantity. Bit packing normally is accomplished in software using the SHIFT and OR instructions. Another alternative would be making the Bit Packer a hardware peripheral to offload the central processing unit (CPU). In the more extreme case, if the CPU is entirely absent, the Bit Packer would have to be a standalone device controlled by a finite state machine. This is the framework under which the Bit Packer is designed.

There are parallel and serial approaches to implement this device with tradeoffs in hardware complexity and speed. The major component in the parallel approach will be the barrel shifter, which can shift parallel bits in one cycle. However, it takes a lot of area and has a noticeable propagation delay relative to a regular shift register. Therefore, the serial approach is favored in the thesis and is discussed in the following sections. It is believed that although the serial approach needs time proportional to the number of input bits to be packed, the simplicity of the logic required allows very high speed clocking, thus the overall performance should be comparable to that of the parallel approach.

3.2.1 Functional Description

Figure 3.23 sketches the Bit Packer functional block. For the purpose of the thesis, the number of input bits is the fixed codeword length, 14, and the output is in byte, or 8 bits. The input control signals are shown in Figure 3.23. Bit Packer is normally in standby mode. When BP is asserted, the bit packing process begins. BP_RESET will clear the internal states of the Bit Packer and send it to the standby mode. FLUSH is a signal that when asserted will force the Bit Packer to output a byte, whether it's packed or not. Finally, WR_RDY is an input signal from an external storage device that indicates if the packed data can be stored away. The storage device is assumed to be a first-in-first-out (FIFO) memory.

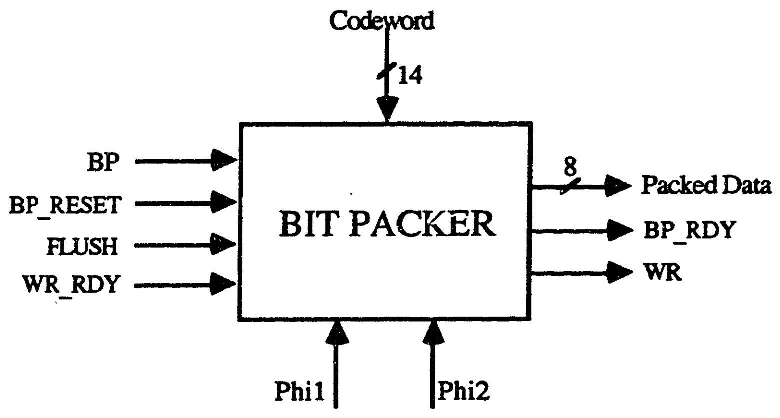


Figure 3.23 Bit Packer Functional Block

The outputs consist of Packed Data[7:0], BP_RDY, and WR. BP_RDY indicates that the Bit Packer is ready to take in another codeword. WR handshakes with the external storage device to which the packed data is output. Finally, two non-overlapping clock signals phi1 and phi2 are used to clock the internal circuit.

3.2.2 Implementation

The Bit Packer is made up of counters, shift registers, tri-stated latch, and a simple finite state machine (FSM). Figure 3.24 shows the architecture. Conceptually, the codeword is loaded into a codeword shift register, which is connected with another 8-bit static shift register, also known as the Byte Template. The arrow indicates the direction of shift. A Countdown Counter preset to binary 14 is used to track the number of codeword bits that has been shifted and is decremented by one with each shift. The Countdown Counter has its inputs hard-wired to a tri-stated binary constant 1110 (14 decimal). When this counter eventually reaches zero, all 14 bits have been shifted, so no more shifting will take place until a new codeword arrives. A Count-to-8 Counter, on the other hand, is used to detect when 8 bits have been shifted into the Byte Template. When it is full, WR is asserted as the 8-bit quantity latched into the tri-stated latch is ready for output.

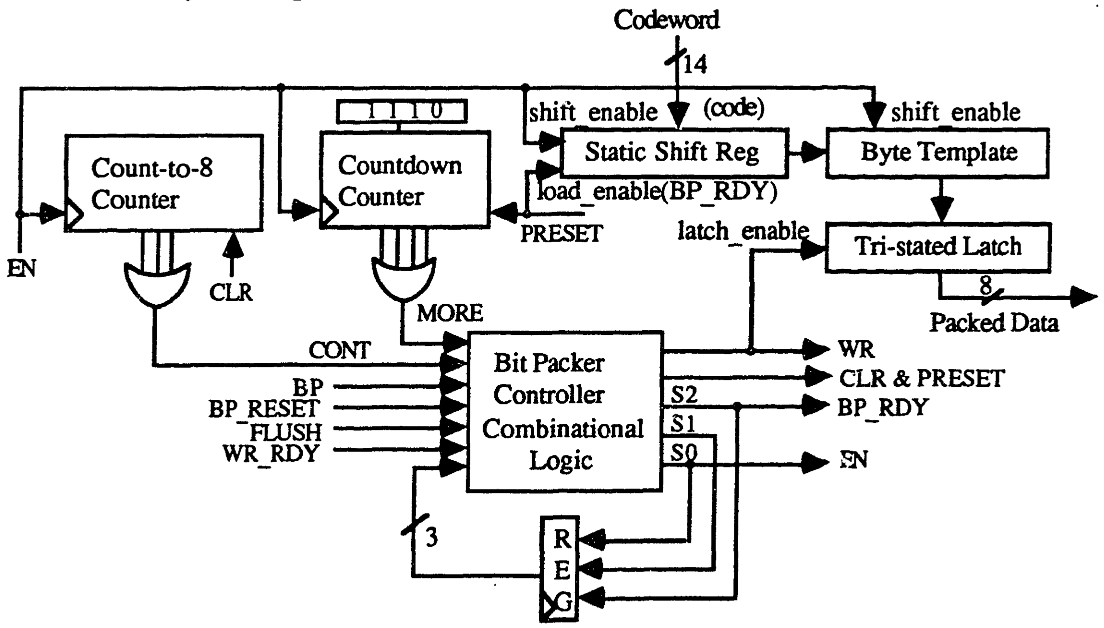


Figure 3.24. Bit Packer Block Diagram

Figure 3.25 shows the state diagram for the Bit Packer Controller. It has six states, each denoted by the state variables (S2 S1 S0). The Bit Packer is idle in (1 0 0) as BP_RDY is asserted. Note that BP_RDY is also used as the parallel-load-enable for the codeword shift register, output-enable for the binary constant, and preset for the Countdown Counter.

The transition to state (0 0 1) from state (1 0 0) is caused by the assertion of BP which is the output of some global FSM. In this new state, the codeword shift register has the codeword and EN is asserted. EN is the signal that clocks the counters and shift-enables the shift registers, as shown in Figure 3.24. This state repeats until either the Countdown or Count-to-8 Counter reaches zero. If Count-to-8 Counter reaches zero (indicated by CONT*) and WR_RDY is asserted, a transition from (0 0 1) to (0 1 1) will take place. However, if WR_RDY is not asserted, then a (0 1 0) will be the new state, which is basically a wait state. It will jump to (0 1 1) as soon as WR_RDY becomes active. If during (0 0 1) Countdown Counter reaches zero (indicated by MORE*) and Count-to-8 Counter is non-zero, a transition to state (1 0 0) will occur.

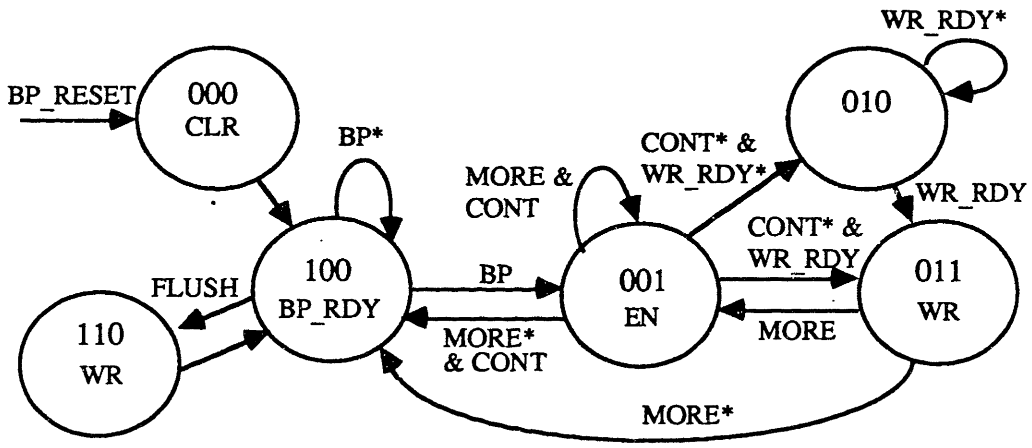


Figure 3.25 State Diagram for the Bit Packer Controller

In state (0 1 1), WR is asserted. It serves as the latch-enable of the tri-stated latch which latches the content of the Byte Template. The 8-bit data by design is valid in this state and ready to be stored away. WR also output-enables the tri-stated latch. Finally, it signals to the external device that the data is ready. The next state transition depends on the value of MORE, as can be seen in Figure 3.25.

Inputs to the FSM are CONT, MORE, BP, BP_RESET, FLUSH, WR_RDY, and three state variables S2, S1 and S0. When BP_RESET is asserted, state (0 0 0) will be the next state regardless of the current state. CLR is asserted during this state. In state (1 0 0), if FLUSH is asserted, a transition to state (1 1 0) will take place. The assertion of WR will force the content of the Byte Template to be written into the external FIFO. FLUSH is asserted by some global FSM in response to the end of packet or any other special circumstances.

Current implementation of the Bit Packer is not flexible, i.e., it can only bit-pack 14-bit codeword into 8-bit. However, it can easily be modified to accomodate variable-length codewords by explicitly loading in the length of the codeword to the Countdown register instead of hardwiring its inputs to 1110 (decimal 14). The state machine does not even have to be changed. On the other hand, if the packed data width needs to go up to 16, hardware modifications are necessary. Simply, a Count-to-16 counter is required, and the Byte Template shift register, packed data bus, and tri-stated latch have to be expanded to 16 bits. The state machine remains unchanged.

3.2.3 Design Evaluation

3.2.3.1 Area Estimate

An area estimate based on the transistor counts, interconnections, and FSM logic reveals that about 600 mil^2 is required by the Bit Packer.

3.2.3.2 Throughput Estimate

It can be determined that the Bit Packer needs 15 cycles in this implementation to pack a 14-bit codeword. This is assuming that the wait state is unnecessary, i.e., WR_RDY is always asserted. One of the cycles is used to output the Byte Template so it can be stored away. The VLSM will output a codeword as frequent as every four cycles[1]; therefore, the Bit Packer requires a clock of four times the frequency of the VLSM clock in order to handle the throughput of the VLSM without delay. However, the use of BP_RDY as handshake will ensure proper flow control. The Bit Packer clocking requirement does not really have to be met unless the full throughput of the VLSM is needed.

The throughput of the Bit Packer can be degraded if the i/o device becomes the bottleneck. For example, the output FIFO might be full already while the packed data is produced by the Bit Packer. It will have to wait until the FIFO has room. This is reflected by the use of wait state, when WR_RDY is not asserted. In short, the throughput of the Bit Packer also depends on the design of the output storage device.

[1] This is the minimum number of cycles each input string encoding will need, specifically in the case of no match at all.

3.3 BIT UNPACKER

This section discusses the design detail of the Bit Unpacker, which, as the name suggests, reverses the Bit Packer operations.

3.3.1 Functional Description

Figure 3.26 shows the Bit Unpacker Functional Block, which takes in 8-bit data and outputs a 14-bit codeword. Obviously, it needs to read two 8-bit data from the external storage device before the 14-bit unpacked data can be produced. Bit Unpacker uses BUP and BUP_RDY to handshake with the outside world. When BUP_RDY is asserted, the external master is allowed to assert BUP to obtain an unpacked data. RD and RD_RDY are the two interface signals between the Bit Unpacker and external storage device. If the latter asserts RD_RDY, then the former can assert RD to fetch a byte from the FIFO. Finally, DUP_RESET is used to reset the Bit Unpacker.

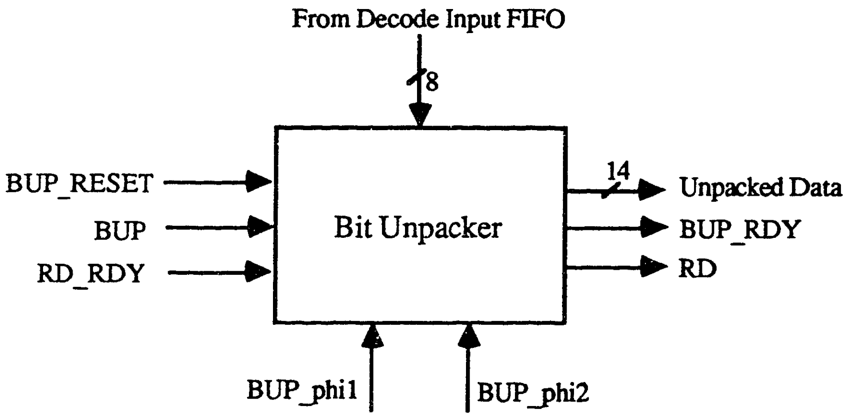


Figure 3.26 · Bit Unpacker Functional Block

3.3.2 Implementation

Figure 3.27 illustrates the major blocks of the Bit Unpacker. Its components are remarkably similar to those of the Bit Packer, but the operation is exactly the opposite. For example, the Bit Packer takes in 14-bit codewords and bundles them into 8-bit quantities, while the Bit Unpacker takes in bundles of 8-bit data and produces 14-bit codewords.

In order to provide the 14-bit codeword immediately upon the request from the external master, the Bit Unpacker is designed such that it always has a 14-bit codeword ready for output. This requires that the Bit Unpacker prefetches the bundles of 8-bit data from the FIFO and serially shifts them into the static shift register, as shown in Figure 3.27.

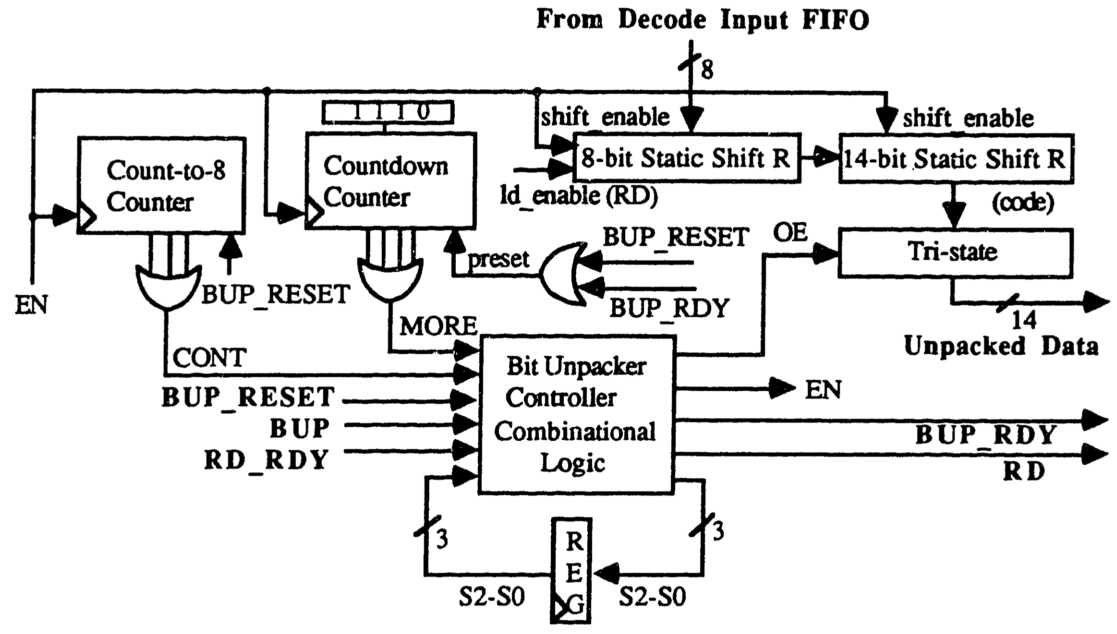


Figure 3.27 Bit Unpacker Block Diagram

The Countdown Counter can be preset to 14, the length of the codeword, by either BUP_RESET or BUP_RDY. As each bit is shifted into the 14-bit static shift register, the Counter is decremented. When it reaches zero, MORE* is asserted. The Count-to-8 Counter, on the other hand, is used to decide when a new 8-bit data should be fetched from the FIFO. It is incremented every time a bit is shifted into the 14-bit static shift register. The counter generates CONT* every eight shifts. Finally, the counters and shift registers are clocked and shift-enabled, respectively, by the rising edge of the signal EN.

Figure 3.28 displays the Bit Unpacker Controller state diagram. There are six states, so three state variables (S2, S1, S0) are required. In the default state (0 0 0), BUP_RDY is asserted to indicate that a 14-bit codeword is ready for output upon the assertion of BUP. As soon as BUP is true, a transition to state (0 0 1) takes place, as the codeword contained in the static shift register is output to the data bus by the assertion of OE, which output-enables the tri-state. The Bit Unpacker then enters state (0 1 1) and starts the process of refilling the 14-bit static shift register by shifting in the remaining bits of the 8-bit static shift register. When the data is exhausted, as indicated by CONT*, a new byte is read from the FIFO. The Bit Unpacker FSM has to make sure that RD_RDY is true before asserting RD to get a byte. RD also load-enables the 8-bit static shift register. The Bit Unpacker returns to the default state when MORE* is asserted, which implies that the 14-bit static shift register is filled.

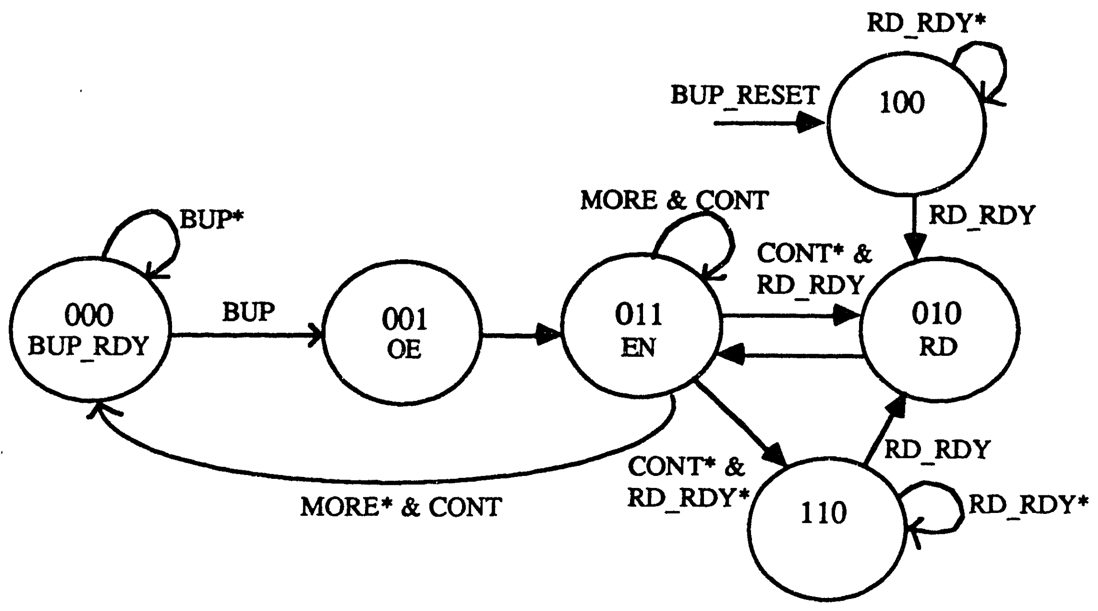


Figure 3.28 State Diagram for the Bit Unpacker Controller

Finally, in the reset state (1 0 0), the Count-to-8 Counter is cleared and the Countdown Counter is preset to 14. Then the Bit Unpacker prefetches the data to fill up the 14-bit static shift register before returning to the default state to assert BUP_RDY.

3.3.3 Design Evaluation

3.3.3.1 Area Estimate

Since the basic components and the size of FSM are similar to those of Bit Packer, Bit Unpacker is also estimated to take 600 mil^2, assuming the 1 μm, two-layer metal CMOS technology.

3.3.3.2 Throughput Estimate

This design of Bit Unpacker allows bit unpacking to take place before the request for bit unpacking is received; therefore, the external master can obtain the 14-bit codeword instantly as long as BUP_RDY is true. In addition, the Bit Unpacker handshakes directly with the FIFO so that the external master does not have to worry about buffer management. On the other hand, the disadvantage of this approach is that only fixed-length codewords can be extracted, since the width of the static shift register is fixed. Microprogrammed approach seems to be the solution to unpack variable-length codeword, but at the expense of more clock cycles. Therefore, there is a tradeoff in flexibility and throughput between these two approaches.

Normally, a 14-bit codeword can be produced about every 16 cycles. However, like the Bit Packer, the throughput depends on the data I/O devices. If the Bit Unpacker asks for an 8-bit quantity from FIFO memory but it is empty, then the Bit Unpacker has to wait. This is reflected by state (1 1 0), which is entered when RD_RDY is not asserted.

CHAPTER 4

CHIP ARCHITECTURE

This chapter covers the Data Compression Engine chip level architecture. Section 4.1 sketches a hypothetical system environment for the Data Compression Engine. This then leads to a proposed chip interface mechanism described in Section 4.2. Finally, Section 4.3 outlines the functional blocks of the Engine based on a finite state machine approach.

4.1 SYSTEM ENVIRONMENT

The Data Compression Engine (DC Engine) can be used under two different environments. For example, it can be attached directly to the communication channel and process the data in real time. For encoding, the host processor provides the user data; the DC Engine will compress and transmit the data onto the channel. For decoding, the DC Engine receives the compressed data from the communication channel and decompresses them; the host processor then obtain the decoded data from the DC Engine. This arrangement appears to have low memory overhead since the data enters and exits the DC Engine in real time. In addition, the throughput of the VLSM can be fully utilized since the input data tends to follow a constant rate.

However, if the communication channel is time-multiplexed or frequency-multiplexed, which is often the case, then data reception and transmission will require additional processing. Moreover, in a packet switched network, only the data field of a packet is compressed or decompressed; the header and flag fields are normally left intact. This suggests that the DC Engine must identify which portion of the input data should be processed. In order to avoid these processing overheads, a generic system environment is assumed for the thesis as sketched in Figure 4.1. For a more concrete description, the host processor is taken to be a Motorola 68020, with 32-bit address and data buses.

In this arrangement, the data to be processed by the DC Engine is stored in the shared memory, which can be accessed by both the CPU and the DC Engine. The DC Engine fetches the source data from the memory, encodes or decodes them, then stores the result back to the shared memory.

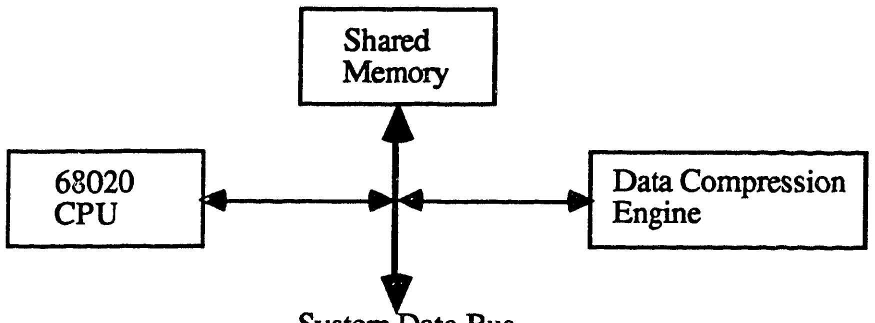


Figure 4.1 Data Compression Engine System Interconnection

One clear disadvantage of this scheme is the large memory transfer overhead, since both the data input and output of the DC Engine require the use of the system data bus, which is shared with the CPU. If very high system throughput is desired, the system i/o could become the bottleneck. One solution is to use a dual port device for the shared memory, so that the CPU and the DC Engine can access the memory independently. The penalty for this approach, however, is the higher cost, particularly if the shared memory is huge. However, the system performance issue is beyond the scope of the thesis; the hypothetical system environment is presented in this section to shed some light on the interface mechanism in which the DC Engine should support.

4.2 INTERFACE MECHANISM

This section attempts to formulate a communication scheme between the external processor and the DC Engine based on the system environment presented in the last section. The major criterion in devising the scheme is to minimize the overhead required by the external processor. In other words, the DC Engine should be as self-contained as possible so that the interface with the external processor will be minimum.

A shared memory approach is proposed and the data structures that support it are fully described. Next, the internal register set that is required as a result of the data structures in the shared memory is proposed. Section 4.2.3 explains the interface mechanism as well as the sequences of events that are necessary to set up the interface. Finally, Section 4.2.4 suggests a non-exhaustive set of commands that the external processor may issue to the DC Engine.

4.2.1 Shared Memory Data Structures

Shared memory is the communication vehicle between the DC Engine and the external processor (EP). It not only contains the packets, but also the information about where the packets are stored in memory. Figure 4.2 shows the proposed functional partition of the shared memory:

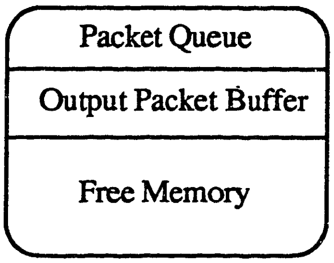


Figure 4.2 Proposed Shared Memory Data Structures

The Packet Queue consists of several Packet Description Tables (PDT), each contains essential information such as the buffer size as well as the location and length of the packet to be processed by the DC Engine. The Packet Queue is organized in a circular buffer fashion, as shown in Figure 4.3. Both the EP and the DC Engine access the Queue in sequential order, starting from PDT #1. After PDT #N is accessed, PDT #1 will be accessed next.

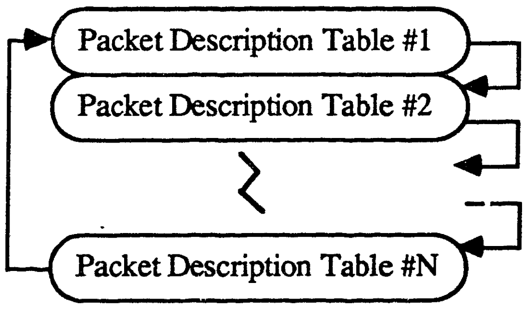
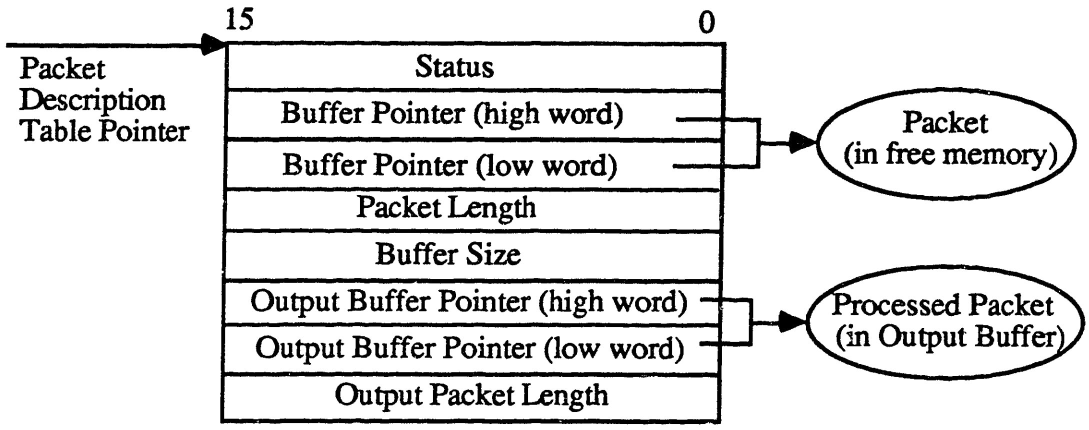

Packet Description Table #1
Packet Description Table #2
Packet Description Table #N

Figure 4.3 Packet Queue Organization

Two data structures are proposed for the PDT. In Option A, it is assumed that the EP supplies not only the location of the source packet but also the storage location of the processed packets. Figure 4.4 illustrates the data structure that supports this scheme. The Buffer Pointer contains the starting address of the packet to be processed. The Packet Length indicates the length of the packet in bytes. The Buffer Size reveals the size of the current buffer in bytes. The Output Buffer Pointer is assigned by the EP. The DC Engine will store the processed packet starting at this location. The Output Packet Length is supplied by the DC Engine so that the EP knows the length of the processed packet. Obviously some cautions must be taken to ensure that the DC Engine won't overwrite any data. This arrangement alleviates the DC Engine's task to locate storage area, but this also violates the principle of minimizing the EP's overhead.



15
0
Packet Description Table Pointer
Status
Buffer Pointer (high word)
Buffer Pointer (low word)
Packet Length
Buffer Size
Output Buffer Pointer (high word)
Output Buffer Pointer (low word)
Output Packet Length
Packet (in free memory)
Processed Packet (in Output Buffer)

Figure 4.4 Packet Description Table Data Structure For Option A

In Option B, the DC Engine determines where to store the processed packets. This could be accomplished by allocating a restricted area in the shared memory called the Output Packet Buffer, as shown in Figure 4.2. Within that memory area, there are several smaller buffers of equal sizes, and each has its status information concerning whether it is free or not. The DC Engine will have to check the buffer status before writing into the buffer. If the processed packet is too large to fit in a buffer, more than one buffer can be used. At the same time, the EP has to update the status portion to mark it free after it reads the buffer. It is expected that the buffers will be filled or emptied in order, much like the way the Packet Queue is accessed.

This scheme effectively alleviates the EP's loading on data compression related output buffer management. The second advantage of the scheme is that the memory space for the the PDT is reduced almost in half, as shown in Figure 4.5. The EP first supplies the packet source address and length in the Buffer Pointer and Packet Length fields, respectively. Later, the DC Engine will supply the encoded packet address and length in the same Buffer Pointer and Packet Length fields by overwriting the source information. Obviously, it is assumed that the EP does not need to know the location of the source packets anymore.

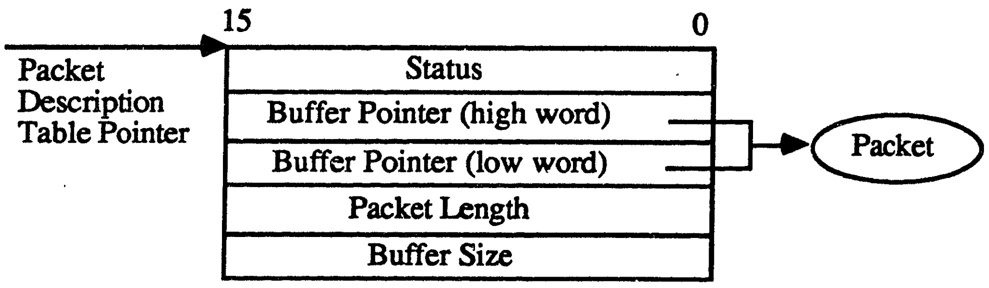

15
0
Packet Description Table Pointer
Status
Buffer Pointer (high word)
Buffer Pointer (low word)
Packet Length
Buffer Size
Packet

Figure 4.5 Packet Description Table Data Structure For Option B

The first word in the PDT is the Status, which carries information about the packet. The status bit assignment is suggested in Figure 4.6. An explanation about each status bit then follows.

15 – 9	8 7 6	5 4	3	2	1	0
Reserved	# of Valid Bits	Type	E/D*	End	Start	New

Figure 4.6 Status Bit Assignment

- NEW indicates whether the packet is current or not. It is set by the EP and reset by the DC Engine after this packet is processed.
- START tells if the packet starts at the current buffer. It is used for buffer chaining purpose, which is required if the packet is too large to fit in a single buffer.
- END tells if the current buffer contains the end of a packet. If both START and END bits are set in a given Status Word, then a whole packet is able to fit in a buffer.
- E/D* indicates whether the packet is to be encoded or decoded. A 0 indicates that it is to be decoded. A 1 indicates that it is to be encoded.
- TYPE is presently set to be a 2-bit quantity which allows the specification of up to four data compression algorithms that the packet is to be processed. In the current implementation, TYPE has a default value of zero since only the ZL77 algorithm will be supported by the DC Engine.
- # of VALID BITS shows the number of valid bits the last byte of the packet contains. When the packets are encoded (compressed), it is likely that the last byte has been padded with dummy bits. On the other hand, when the packets are decoded, the decoder needs to know the number of useful bits the last byte contains. Therefore, the remote Encoder writes into this field and the local Decoder reads from it.

It is fundamental to require that the EP fills the PDTs in the Packet Queue in sequential order and wraps around when the end of the queue is reached. In addition, before the EP writes the information for a packet, it must check the NEW bit field of the Status of the current PDT. If NEW is still set, it means that the DC Engine has not finished processing that packet. As a result, the EP can not overwrite the current PDT and must wait or try again later. This scheme ensures the proper flow control of the Packet Queue.

Based on the requirement proposed above, the DC Engine can process the packets in order by going down the Packet Queue. When the DC Engine detects that the packet is not current, i.e., NEW is not set, it can be sure that no more packet is to be encoded for the moment. Similarly, if later the EP has more packets, it will fill in the PDT where the DC Engine most recently finds it not current. This is an important restriction that must be followed. Otherwise, the EP and the DC Engine will be out of synchronization.

4.2.2 Internal Registers

Based on the Shared Memory data structures, the following set of internal registers shown in Figure 4.7 is necessary for efficient communication.

Data
Command
Interrupt Vector
Semaphore
Base Packet Queue Pointer
Packet Queue Size
Status
Packet Length
Buffer Size
Base Output Buffer Pointer
Output Buffer Size
Number of Output Buffers
Status Register Pointer
Output Buffer Pointer
Packet Queue Pointer

Figure 4.7 Proposed Internal Register Set

Of all these registers, only the first four are directly accessable to the EP. They are selected by the A1 and A0, the lowest two address inputs of the DC Engine. The Data Register is used to load in values for the internal registers. The Command Register allows the EP to issue different commands to the DC Engine, such as initialization, load Buffer Pointer, Stop, Start, etc. The Semaphore Register is read by the EP and written by the DC Engine. Before the EP issues a command, it will first check the Semaphore Register. If it contains a 0, then the DC Engine is ready to accept a command. Otherwise, the EP has to wait. Each time the DC Engine reads the command, it will write a 1 to the Semaphore Register. When the command is finished, the DC Engine writes a 0 to it.[1]

As each data byte is read in, such as in a direct memory access (DMA) cycle, the Buffer Pointer is auto-incremented while the Buffer Length is decremented. These activities are coordinated by the DMA controller which will be explored in Section 4.3.4. The Base Packet Queue Pointer and the Base Output Buffer Pointer contain the starting addresses of those two data structures. The Packet Queue Size tells the number of PDTs in the Packet Queue.

The Status Register Pointer is reserved for future use. For example, if more information needs to be communicated between the EP and the DC Engine, the pointer can be used to point to a memory space where the information resides. The EP can issue a READ STATUS command to tell the DC Engine where to dump its internal registers contents before the READ INTERNAL REGISTER command is issued.

The Interrupt Vector Register can be read or written by both the EP and the DC Engine. Its use will be described in the next Section. Note that this internal register set is not complete. A more detailed look might reveal that more registers are necessary.

4.2.3 Interface Scenario

When the system is powered up, the EP is responsible for initializing the DC Engine. The major tasks include loading the Engine with the Packet Queue Pointer, Packet Queue Size (the number of PDTs in the Packet Queue), Output Packet Buffer Pointer (if Encode Table Option B is chosen), and Output Packet Queue Size. The first pointer contains the starting addresses of the Packet Queue. Each of the PDT in the Queue will be referenced by a constant offset from this starting address.

In the normal operating mode, the DC Engine checks for the Status portion of the current PDT. If the NEW bit is set, the DC Engine will fetch the Buffer Pointer, Buffer Size, and Packet Length into appropriate internal registers and then DMA the data into its internal buffer. Since the PDTs in the Queues are written by the EP sequentially, the DC Engine, when

[1] This concept is similar to Motorola X.25 Protocol Controller Chip's interface with external processor.

finished with the current packet, resets the NEW bit. It will then fetch the next PDT Status word and check to find out if there are more packets to be processed. On the other hand, the EP also keeps track of whether the processed packets have been fetched. When the EP checks the Status and detects that NEW is reset, it will fetch the processed packet and goes to the next PDT.

When the DC Engine exhausts the packets, it will interrupt the EP. There are two possibilities as to how the EP signals to the DC Engine for the arrival of new packets. The first solution is to have the DC Engine poll the same PDT until its Status Word's NEW bit is set by the EP. This requires the least amount of overhead for the EP, but the polling by the DC Engine will consume the system bus bandwidth since the DC Engine needs to load the Status word from the shared memory into its internal register for bit field checking. The other choice is to have the EP interrupt the DC Engine when new packets become available for processing. This can be accomplished by writing appropriate information into the Interrupt Vector Register. The DC Engine will check the Interrupt Vector Register to decode the message. In this scheme, external polling is unnecessary, but internal polling of the Interrupt Vector Register is required.

4.2.4 Command Set

This section only gives a flavor of what commands might be necessary to establish the interface mechanism or to control the activities of the DC Engine. At present, there is no way to come up with a more complete set of commands because we don't have the exact data structures nor system functional specification, which really depends on the environment the DC Engine is situated. Some useful and generic commands are listed below:

Reset
Initialize
Dump Internal Registers
Read DC Engine Status
Stop
Read Status

Note : The EP issues command only when the Semaphore Register is 0. The 8-bit command is loaded into the Command Register via the data lines D7-D0.

4.3 FINITE STATE MACHINE ARCHITECTURE

In order to discover the maximum throughput of the DC Engine utilizing the VLSM structure, a totally dedicated hardware approach is taken to encode and decode data. This approach requires exploiting the regularities and peculiarities of the data compression algorithm used, which, in this case, is a ZL77 scheme with fixed-length codeword encoding and decoding and without the innovation character as part of the codeword.

By clearly defining the operations required at the expense of generality, the finite state machine controlled data compression encoder and decoder becomes the optimal solution. Since the use of the VLSM for string matching tremendously speeds up the encoding process and yet requires very little control, treating it as a peripheral controlled by a microprogrammed environment seems inefficient. Furthermore, decoding needs to be at least as fast as encoding. If encoding is facilitated by the VLSM, decoding should be made fast with appropriate hardware support as well, such as the bit unpacker.

Although the resulting architecture will be inflexible, it has its own great value, because it allows us to get a sense for the maximum possible throughput of the DC Engine. In addition, it shows one of the data paths most suitable for full duplex data compression processor, and it would not be too difficult to modify the architecture to provide more flexibility in the future. In short, this Section examines the extreme case, and the result can be used to compare against the performance of a more general DC Engine in the future.

4.3.1 The DC Engine Architecture

Figure 4.8 shows the functional blocks of the DC Engine.

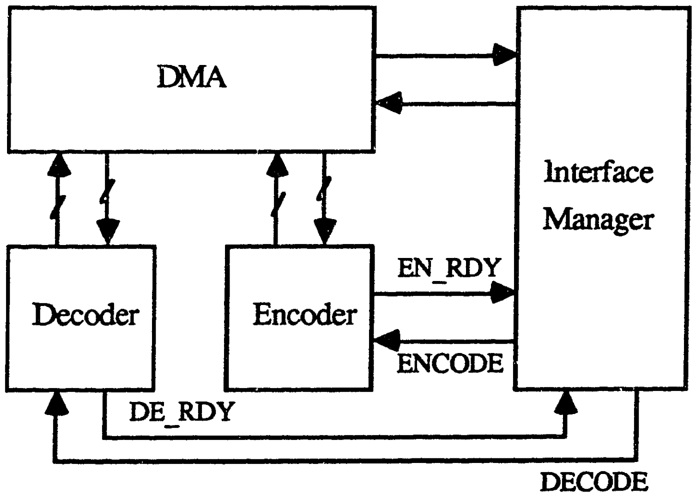


Figure 4.8 DC Engine Functional Blocks

There are four major functional blocks: the Encoder, the Decoder, the DMA Controller, and the Interface Manager. The Encoder performs the ZL77 encoding operation as explained in Section 3.1.1. The Decoder decodes the ZL77 codewords. Both the Encoder and the Decoder receive source data from and output processed data to the DMA Controller block. The DMA Controller is in charge of chip level data input/output operation, including shared memory address generation. Finally, the Interface Manager is responsible for interfacing with the external processor, maintaining the shared memory data structure, providing the source or destination addresses to the DMA Controller, and coordinating the Encoder, Decoder, and DMA Controller blocks. It also contains the internal register set. The following sections will describe the Encoder and Decoder architectures in detail. The DMA Controller and the Interface Manager are presented only in terms of their general functionalities, not their implementations.

4.3.2 The Encoder

The Encoder in the ZL77 context is an entity that receives the original data and compresses them into fixed-length codewords. The codeword length is 14 bits as a result of the implementation parameters described in Section 2.2.4. On the functional level, the Encoder obtains a maximum of 256 data bytes at a time from the DMA Controller. If the data portion of a packet is longer than 256 bytes, then more than one block transfer is necessary. The Interface Manager is responsible for coordinating this activity. As for the output, a maximum of 128 bytes can be transferred to the DMA Controller from the Encoder. Again, if the resulting codewords for a packet take up more than 128 bytes, more than one block transfer will take place. The Encoder is responsible for informing the Interface Manager when the encoding for a packet is finished, so that the DMA Controller can perform necessary chip level input/output operation.

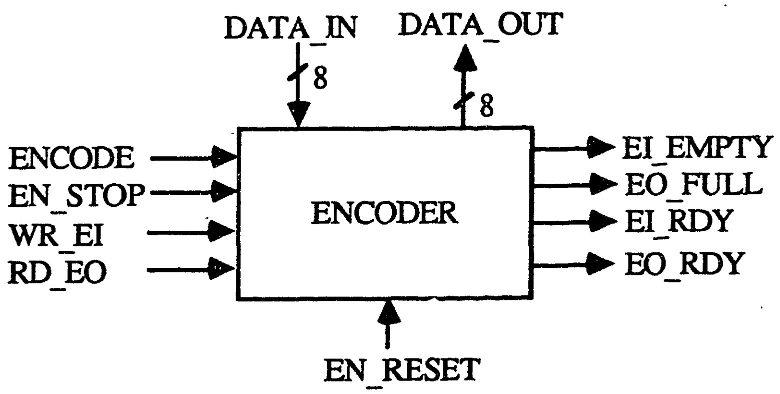


Figure 4.9 The Encoder Functional Block Diagram

Figure 4.9 shows the Encoder functional block diagram. ENCODE is asserted by the Interface Manager to activate the Encoder, while EN_STOP is used to stop the Encoder, such as when the last byte of the packet is being encoded. WR_EI and RD_EO come from the DMA Controller to indicate whether data are to be written into or read from the Encoder. EI_EMPTY, EO_FULL, EI_RDY, and EO_RDY are status signals to the Interface Manager so that it can coordinate the DMA Controller to perform appropriate i/o functions. EN_RESET is used to reset the Encoder. Finally, DATA_IN is the input data bus, and DATA_OUT is the output data bus, both are 8-bit wide.

4.3.2.1 The Encoder Implementation

The data path and the major components of the Encoder are illustrated in Figure 4.10. Data to be encoded originates from the DMA Controller, enters the Encode Input FIFO (First In First Out), and travels through the VLSM, the Bit Packer, and finally the Encode Output FIFO. The Encoder is controlled by a finite state machine, called the Encoder FSM.

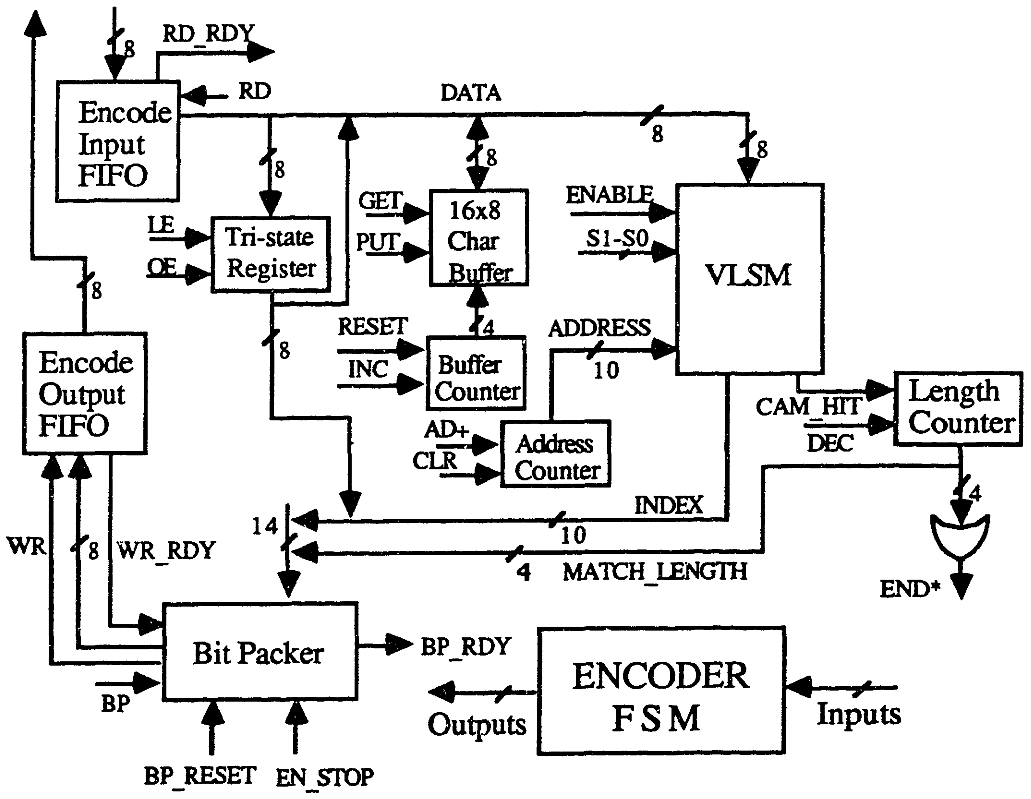


Figure 4.10 The Encoder Architecture

The Encode Input FIFO is 256-word deep and 8-bit wide. It is written by the DMA Controller and is read by the Encoder. The FIFO is designed to minimize the memory management overhead required by the external devices [37]. Therefore, it contains internal logic for updating the FIFO address pointer every time a FIFO read or a write occurs. In addition, the internal logic responds to asynchronous handshake signals as well as the condition of the FIFO. For example, a byte can be read by asserting the RD signal provided the RD_RDY signal is asserted by the FIFO. Similarly, WR is asserted by the external device to write a byte into the FIFO as long as WR_RDY is positive. When the FIFO is empty or almost empty, a signal EI_EMPTY will be generated to warn the external devices which use the FIFO. In short, this FIFO design is well-suited for FSM control.

The use and control of the VLSM have been discussed extensively in Section 3.1.4, so they will not be repeated here. The VLSM device is naturally the largest and the most important component of the Encoder. The Encoder FSM generates a logical sequence of control signals to the VLSM to implicitly find the longest match and produce the codeword.

The Bit Packer takes in 14-bit quantities and packs them into bytes. Its outputs are connected directly to the Encode Output FIFO. The Bit Packer therefore is designed to handshake with the FIFO as well. As can be seen, the 4-bit MATCH_LENGTH and the 10-bit Index are hard-wired as inputs to the Bit Packer. Note that the output of the Tri-state Register could also be input to the Bit Packer. The Tri-state Register always latches the input character presented to the VLSM. If there is no match at all, i.e., MATCH_LENGTH equals to zero, then the character itself is stored in the Index portion of the codeword, as explained in Section 2.2.2. In this case, the OUTPUT command of the VLSM is never issued by the Encoder FSM; instead, the content of Tri-state Register is fed into the Bit Packer in addition to MATCH_LENGTH, which is zero.

The Encode Output FIFO has the similar structure as the Encode Input FIFO, except that it has only 128 bytes and is written by the Encoder (the Bit Packer, to be more specific) and read by the DMA Controller. The Bit Packer generates WR signal to write into the FIFO. An EO_FULL signal is generated when the FIFO is full or almost full.

The Length Counter is capable of counting up and down. CAM_HIT is the count-up clock for the counter while DEC acts as the count-down clock. The four-bit output of the Length Counter are inputs to the Bit Packer as well as an OR gate. When the outputs are all zeroes, END* is asserted low. It is used as an input to the Encoder FSM and means different things at different FSM states.

The Address Counter is essentially the history buffer pointer which always points to the next position where a new input character is to be inserted into the VLSM. This 10-bit binary counter is incremented by one whenever a UPDATE command for the VLSM is executed. When it counts to 1023, it will return to zero and start again. The Address Counter is reset when the CLR signal is asserted.

The Character Buffer can buffer up to 16 input characters. This is required because after the codeword is generated, each encoded input character needs to be written into the VLSM. The Buffer Counter is cleared before a new encoding cycle begins. As each input character is read from the Encode Input FIFO, it is being presented to the VLSM as well as written into the Character Buffer, and the Buffer Counter is incremented to point to the next position. When the characters are about to be inserted into the VLSM, the Buffer Counter is reset to point to the first character. During the UPDATE cycle, the Length Counter is decremented as each input character is read from the Character Buffer and written into the VLSM location specified by the Address Counter. The Encoder FSM will use the END* signal to determine when to conclude the UPDATE activity.

In addition to controlling every component shown in Figure 4.10, the Encoder FSM needs to worry about some special situations. For example, the COMPARE cycle stops when a miss occurs, i.e., CAM_HIT is not asserted. At that point, the last character that is not part of the longest match is still in the Tri-state Register. The FSM has to make sure that in the next encoding cycle, the first character must come from the Tri-state Register, not from the FIFO.

Another important provision is that when the last character of the current packet has been compared, a codeword must be generated even though the longest match has not been found. In other words, the Encoder FSM must recognize the end of packet and break the normal VLSM operating cycle by forcing a OUTPUT command to get the index. Together with the content of the Length Counter, the codeword is fed to the Bit Packer. If one recalls from Section 3.2.2, the Bit Packer FSM is able to respond to such condition, i.e., if the signal EN_STOP, an input to the Bit Packer, is asserted, the Bit Packer will output the Byte Template even though it is not filled with 8 bits yet.

Finally, the Encoder must be resetable, in which case all the counters will be cleared and the FSM will be in standby mode. Figure 4.11 is a state diagram for the Encoder FSM. This diagram illustrates the major activities of the Encoder.

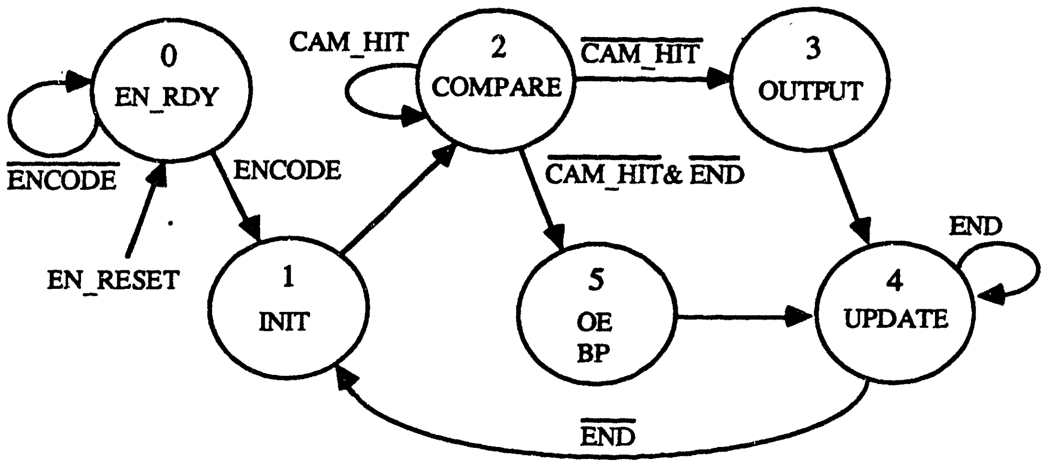

0
EN_RDY
$\overline{\text{ENCODE}}$
ENCODE
EN_RESET
1
INIT
CAM_HIT
2
COMPARE
$\overline{\text{CAM\_HIT}}$
3
OUTPUT
$\overline{\text{CAM\_HIT}}$& $\overline{\text{END}}$
5
OE
BP
END
4
UPDATE
$\overline{\text{END}}$

Figure 4.11 Simplified Encoder FSM

4.3.2.2 The Alternative Encoder Architecture

If we closely examine the current Encoder design, we find it awkward having to buffer the input characters in the Character Buffer and then read from it later to update the VLSM. It would be cleaner and more elegant if the input character could be inserted into the VLSM right after the COMPARE cycle, so that it does not have to be saved somewhere. Some silicon area could be saved by eliminating the Character Counter and Character Buffer which is a static RAM structure with address decoding and sense amplifier circuits.

The first question comes to mind before any modification: is it okay to modify the history buffer before the current encoding cycle is finished? The example in Figures 4.12 will help answer this question. In Figure 4.12 (a), the history buffer contains "abcdefg", and the input string is "abcf". The history buffer pointer points to position 1. Suppose we insert the input character immediately after the COMPARE cycle. Figure 4.12 (b) shows the result of a COMPARE and a UPDATE cycles. Note that 'b' under position 1 was overwritten with 'a', the input character! Therefore, when we presented the character 'b' in the second COMPARE cycle, we failed to get a match! Have we destroyed the chance to find the longest match by modifying the history buffer before the longest match is found?

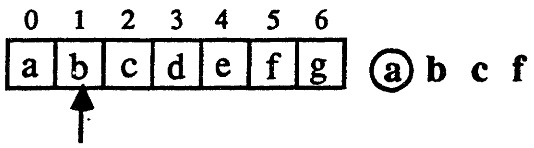


Figure 4.12 (a) History buffer before the COMPARE and UPDATE cycles

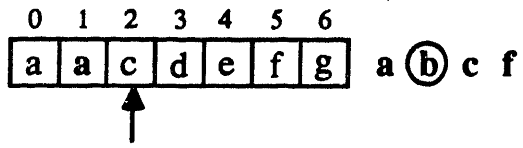


Figure 4.12 (b) History buffer after the COMPARE and UPDATE cycles

Superficially, we see that the longest match is "abc" in the Figure 4.12 (a). However, if we unfold the history buffer in time, we learn that the order in which the characters had arrived is 'b', 'c', 'd', 'e', 'f', 'g', and 'a', with 'a' being the most recent character. Therefore, there is no such string as "abc" in the history buffer. The fact that the history buffer wraps around creates the illusion that the string "abc" is present. Furthermore, the history buffer implemented in the thesis contains 1024 characters, so that the probability of such occurrence shown in this example is very small. In short, we have concluded that it is fine to update the history buffer immediately after a COMPARE cycle.

The next question is: how do we do it? Clearly, the sequence of commands issued to the VLSM has to change. In the past, the commands required for a match length of two go as follows:

INIT COMPARE COMPARE COMPARE OUTPUT UPDATE UPDATE.

With the modification, the new sequence becomes:

INIT COMPARE UPDATE COMPARE UPDATE COMPARE OUTPUT.

Since the next COMPARE cycle comes one cycle later instead of the immediate next cycle, we need to have one more flip flop in each VLSM Word Cell to delay the match result for one more clock cycle. Thus, overall, there will be 1024 more flip flops in the VLSM device, or 8192 more transistors as each dynamic flip flop consists of 8 transistors. Preliminary estimate reveals that the Character Buffer and the Buffer Counter will take up 630 mil^2, compared to 2450 mil^2 consumed by 1024 flip flops. However, it is difficult to evaluate the impact of this modification upon the total VLSM area cost, because it depends on the layout of each VLSM Word Cell. It is possible that for the original Word Cell, compact layout can not be achieved, so adding one more flip flop to each Cell does not significantly increase the Cell area, thus the total area. As a result, the actual additional area is likely to be much smaller than the estimate suggests. Furthermore, the Encoder FSM does not have to control those two components anymore, so the PLA (programmable logic array) that implements the FSM will become smaller. Therefore, the modification can still be justified, even though the original idea of area saving does not seem to hold up.

The alternative Encoder architecture made possible by the modified VLSM design is cleaner, since the Character Buffer and the Buffer Counter are eliminated. Moreover, it becomes easier to see that the OUTPUT cycle of the current encoding process and the INIT cycle of the next encoding process can actually be combined! Therefore, the effective VLSM cycle requirement per codeword produced can be reduced to (2n + 2) from (2n + 3), a saving of one clock cycle! Appendix C shows the alternative Encoder architecture and its detailed FSM implementation.

4.3.2.3 Area Estimate

The total area of the Encoder is estimated by counting the number of transistors from each component other than the VLSM and the Bit Packer, whose areas have been presented in Chapter 3. The regular structures such as the PLA, Character Buffer, and the FIFO's are treated separately, using the RAM or ROM cells approximation. The total area is about 32,500 mil^2. Chapter 7 will show how this area is compared to a typical die size.

4.3.2.4 Throughput Estimate

In estimating the peak throughput of the Encoder, we assume that the source data are available all the time. For example, whenever RD is asserted, the Encode Input FIFO will supply a character. Furthermore, we assume that the encoded data can always be output. For example, whenever the Bit Packer asserts WR, the Encode Output FIFO can always accept the byte. Finally, the Bit Packer is assumed to be running a clock four times as fast as the Encoder clock, so that the outputs of the VLSM can be packed in time. All these restrictions can be relaxed in real implementation; the state diagram for the Encoder FSM shown in Appendix B takes care of possible congestions in the Encode Input FIFO, Encode Output FIFO, and the Bit Packer by inserting wait states.

Under these "ideal" condition, the throughput of the Encoder approaches that of the VLSM, which can be found in Section 3.1.8.2. It is more difficult to predict the effective throughput of the Encoder when the i/o traffic is congested or when the Bit Packer can not keep up with the VLSM.[1] However, the peak throughput should give us a good approximation.

[1] This can happen when there are consecutive "no-matches"; the VLSM produces a codeword every four cycles for a no-match, while the Bit Packer requires at least 14 cycles to pack the codeword.

4.3.3 The Decoder

The Decoder is more straightforward because it mainly takes apart the codewords into Index and Length, reads the characters off from its history buffer, updates the history buffer, and outputs them to the DMA Controller. On the functional block level, the Decoder resembles the Encoder, as shown in Figure 4.13. The interface signals work the same way as those of the Encoder, so they will not be explained here.

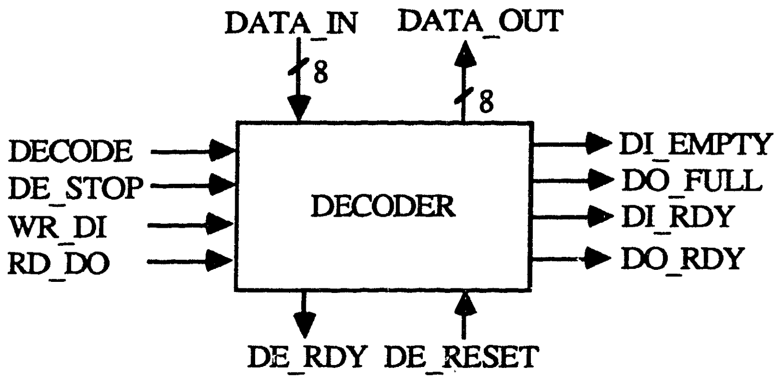


Figure 4.13 The Decoder Functional Block Diagram

4.3.3.1 The Decoder Implementation

The Decoder basically is concerned about three things. First, it has to unpack the byte quantities into 14-bit codewords. Second, it has to subtract the Index from the Length plus one to get the correct starting Index, because the Index produced by the VLSM is the index of the last character, not the first character, of the longest match. Third, when the Length field is zero, the character the codeword encodes is in the Index field, so there is no need to read from the history buffer. The major blocks of the Decoder reflect these tasks, as shown in Figure 4.14. The Decoder is controlled by a Decoder FSM.

The DATA_IN of Figure 4.13 enters the Decode Input FIFO, which has the same structure as the Encode Input FIFO except that the Decode Input FIFO is only 128-byte deep. It is written by the DMA block and read by the Bit Unpacker. The proposed implementation requires the Bit Unpacker to take in 8-bit quantity and output 14 bits. Obviously at least two 8-bit words are needed to provide the 14-bit codeword.

The 14-bit outputs of the Bit Unpacker are broken into a 10-bit Index field and a 4-bit Length field. The Index and Length are operands to the Subtractor. The subtract operation can be done in either the Encoder or the Decoder, but is chosen to be implemented in the Decoder to balance the operating overheads between the two. The difference plus one is the correct index which is latched into the Address Counter.

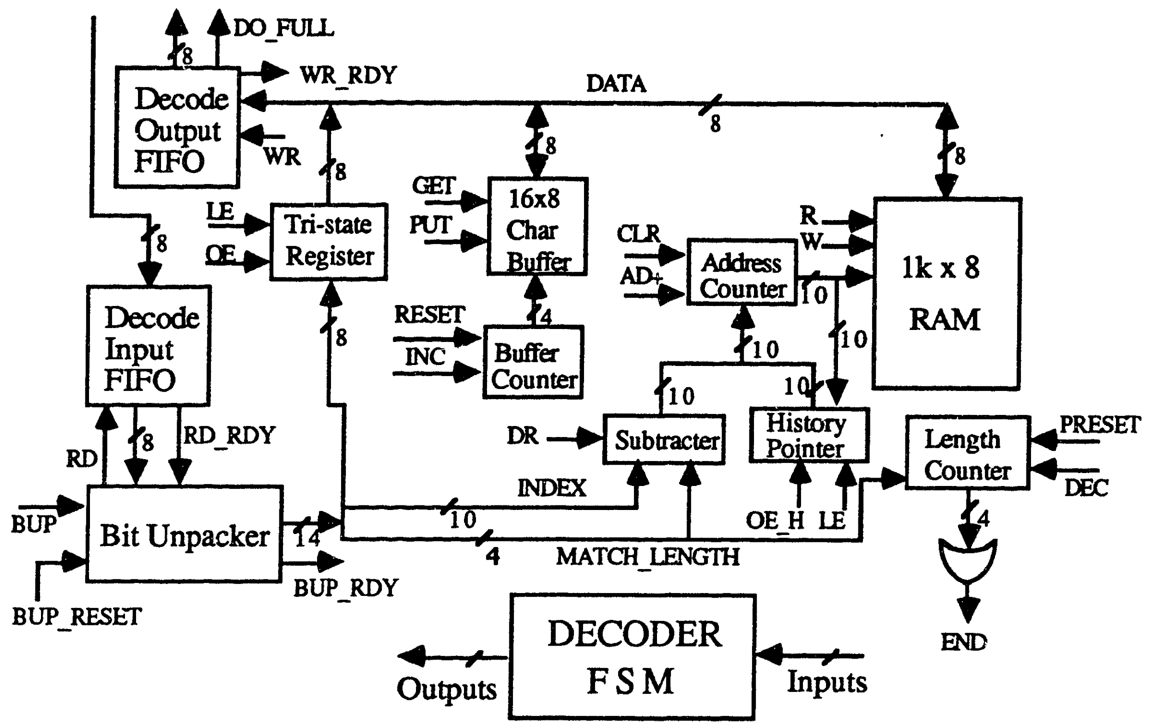


Figure 4.14 The Decoder Architecture

The 4-bit Length presets the Length Counter. The decoding procedure is to extract characters from the history buffer (stored in RAM) based on the index and length specified in the codeword, and output them to the Decode Output FIFO. Every time a character is read, the Address Counter is incremented, the Length Counter is decremented, and the same character is temporarily stored in the Character Buffer. When the Length Counter reaches zero, the history update cycles begin. First, the Address Counter is loaded with the value in the History Pointer. Next, the characters are recalled from the Character Buffer and written into the history buffer RAM one by one. At the end of the update operations, the value in the Address Counter is stored into History Pointer for future update use.

The Tri-state Register always latches the lower 8-bit of the Index field. In the special case in which the Length field of the codeword is zero, the lower 8 bits of the Index field will be the character itself. Therefore, the Decoder will just output-enable the Tri-state Register and bypass the procedure of getting the raw character from the history buffer. Certainly, the character has to be written into the history buffer.

The Decode Output FIFO is 256-byte deep, twice the size of the Input FIFO. This is in anticipation of data expansion, assuming a 2 to 1 compression ratio.

4.3.3.2 The Alternative Decoder Architecture

There appears to be at least two modifications that can be made to optimize the Decoder in area and performance. The first one is similar to the proposal discussed in Section 4.3.2.2, namely the newly decoded character can be inserted into the history buffer before all characters represented by the codeword are read off. This change of procedure will eliminate the Character Buffer and the Buffer Counter and simplify the Decoder FSM. However, the tradeoff is that now the History Pointer, originally a 10-bit latch, has to become a much larger 10-bit counter. Nonetheless, this 10-bit counter only takes 20% of the area occupied by the Character Buffer, Buffer Counter, and the 10-bit Latch, so a net area saving is achieved. The address input to the history buffer RAM now must be multiplexed bewteen the Address Counter and the History Pointer, since both memory read and write require addressing.

The second modification concerns the (Index - Length + 1) operation. It would be nice to avoid the "plus one" step since it takes up a cycle. One possibility is for the VLSM to produce an Index that is one more than the actual index. This way, just (Index - Length) will give the correct starting address of the longest match. In order for this scheme to work, the VLSM address encoding has to be modified. Fortunately, it only involves the physical location change for the logic circuits that pull down the H_MATCH and the V_MATCH lines of the VLSM Word Cells. In other words, the logic circuits for a Word Cell N now is located inside its righthand neighbor, or Word Cell (N+1), so that when a match occurs in Word Cell N, the Index of (N+1) will be produced, assuming it is the lowest index. Figures 4.15 give a graphical comparison between the original Word Cell and the rearrangement of the Word Cell.

4.3.3.3 Area Estimate

The total area of the Decoder without the Character Buffer and the Buffer Counter is 12,000 mil^2, about 37% of the Encoder.

4.3.3.4 Throughput Estimate

The peak throughput of the decoder again assumes the ideal conditions described in Section 4.3.2.4, namely infinite data supply and output, and the Bit Unpacker running a clock four times as fast as the Decoder clock. Furthermore, we assume that only (Index - Length) operation is required, not (Index - Length + 1).

Then we can proceed to observe that for each character decoded, a memory read and a memory write (history buffer update) are required. One or two cycles are required to fetch the codeword and subtract the Length from the Index. Therefore, the number of cycles to process a codeword is about (2N + 2), where N is the number of characters the codeword encodes.

Therefore, the throughput is similar to that of the Encoder or the VLSM. Again, please refer to Section 3.1.8.2 for the throughput plot as a function of the clock period.

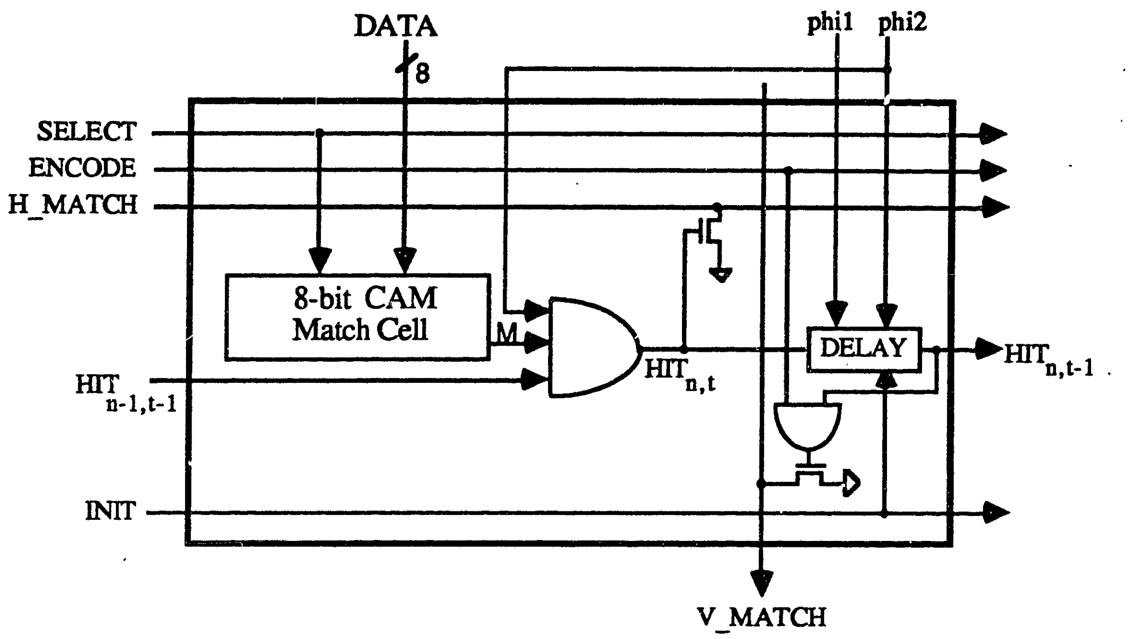


Figure 4.15 (a) The Original CAM Word Cell

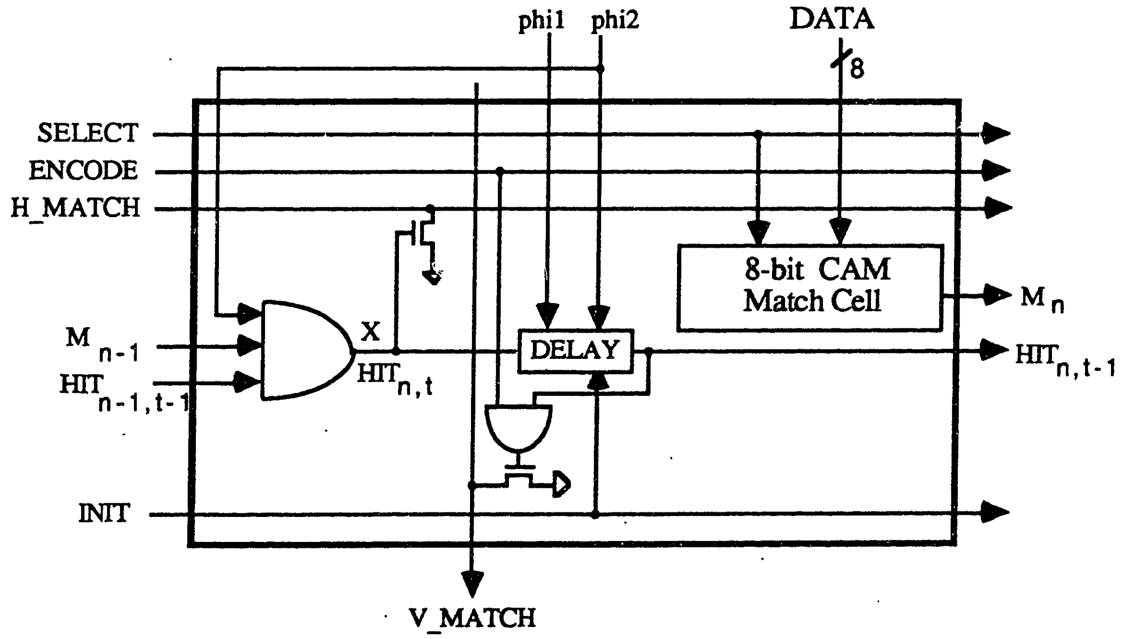


Figure 4.15 (b) The Modified CAM Word Cell

4.3.4 The DMA Controller

The DMA Controller handles the DC Engine's data input/output operations. It essentially bridges the external shared memory and the internal FIFOs. The DMA Controller must perform the following tasks:

- Get control of the system data bus, so that the DC Engine becomes the bus master
- Generate external addresses
- Handshake with the shared memory to perform memory read/write cycles
- Transfer data in blocks whose sizes are specified by the Interface Manager
- Read from Encode and Decode Output FIFO's and write into external memory
- Read from external memory and write into Encode and Decode Input FIFO's
- Interface the DC Engine data bus with the FIFOs which are only 8-bit wide
- Handshake asynchronously with the FIFOs

In the thesis, the DMA Controller is treated on the functional block level only, due to time constraints and the fact that not too much novel design is possible. Figure 4.16 shows the conceptual blocks that are necessary to implement the requirements specified above.

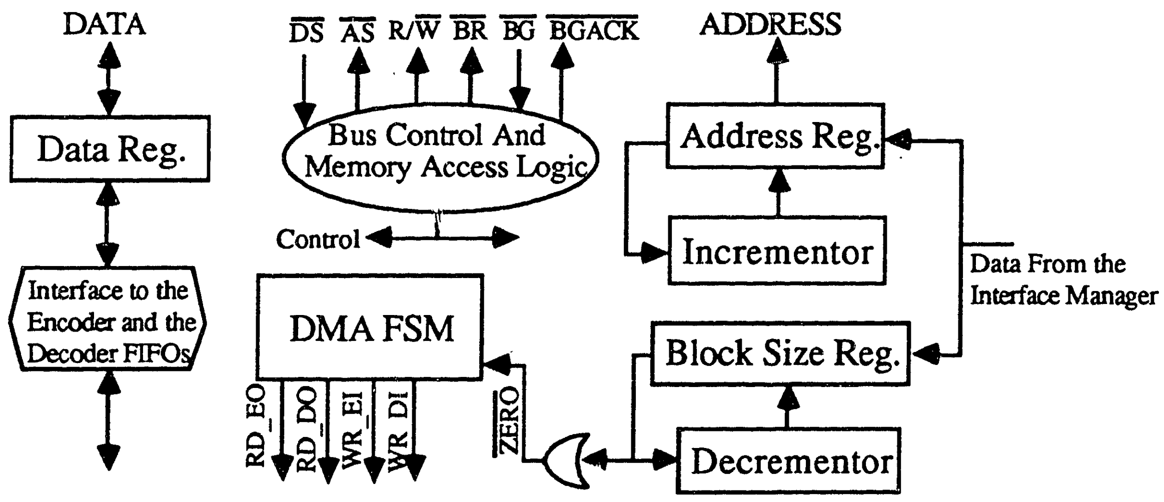


Figure 4.16 The DMA Controller Architecture

The Interface Manager provides the Base External Address which is stored in the Address Register. In addition, the Interface Manager specifies the data transfer block size. Then the DMA Controller will activate the Bus Control and Memory Access Logic to read data from or write data into the shared memory. The content of the Block Size Register is decremented every time a byte is accessed, while the content of Address Register is incremented. If the system data bus is 32-bit (= 4 bytes) wide, then the amount to be decremnented or incremented will be four at a time. When the Block Size Register reaches zero, the memory transfer activity will terminate.

The interface to the Encoder or Decoder FIFOs ensures proper byte alignment between the Data Register and the FIFOs. The latter are only 8-bit wide while the former is either 16 or 32-bit wide. Finally, the DMA FSM basically coordinates each functional blocks. It also performs asynchronous handshakes with the FIFOs. For example, to read from the Encoder Output FIFO, RD_EO will be asserted to get a byte from the FIFO. Again, Figure 4.16 only presents a conceptual model of the DMA Controller; therefore, many signals and interconnections are omitted.

4.3.5 The Interface Manager

The Interface Manager is the highest level controller of the DC Engine. Its responsibilities include carrying out the interface mechanism with the external processor as described in Section 4.1; providing the DMA Controller with information regarding the base address of external memory, data transfer block size, and data transfer source port and destination port; and finally, responding to the status signals from the Encoder and the Decoder.

The DC Engine communicates with the outside world mainly through the Data Register, which can be accessed by both the DMA Controller and the Interface Manager. The latter is responsible for routing the data structure parameters from the Data Register to the appropriate internal registers described in Section 4.2.2, such as the Packet Queue Pointer, Packet Size, Buffer Length, etc.

The Interface Manager is governed by a complicated FSM which might consist several smaller FSMs. In addition, ramdom logic must be designed to perform Status Word bit field testing to determine if the current Packet Description Table contains a new packet, if the packet should be decoded or encoded, and so forth. Furthermore, the Status Word must be updated when a packet is processed, and the processed packet length must be provided in the appropriate field of the current Packet Description Table. Finally, the Interface Manager has to respond to the command or interrupt issued by the external processor when the Command or the Interrupt Register is written.

Possible data transfer source-destination pairs for the DC Engine are summarized in Table 4.1. SM denotes shared memory, EI denotes Encoder Input, EO denotes Encoder Output, and DI denotes Decoder Input, etc.

For example, when EO_FULL, signalling that the Encoder Output FIFO is full, is asserted by the Encoder, the FSM will inform the DMA Controller to transfer the data from the EO FIFO to the shared memory. The Encoder is in wait state until at least some data of the

FIFO is emptied. The Interface Manager can handle data transfer requests from each source-destination pair on a round-robin, first-come-first-serve, or priority basis.

SM -> EI FIFO	SM -> DI FIFO	EO FIFO -> SM	DO FIFO -> SM
SM -> Data Reg.	Data Reg. -> SM	SM -> Command	SM -> Interrupt
SM -> Semaphore			

Table 4.1 Data transfer source-destination pairs

CHAPTER 5

SIMULATION

5.1 VERILOG BEHAVIORAL MODELLING

The Encoder block architectural design of the DC Engine was simulated with VERILOG, a digital design language and interactive simulation system that encompasses the capabilities for behavioral, register-transfer, gate, and switch levels modeling. A C-like programming environment of VERILOG enables functional description of individual blocks as procedures and interconnections as arguments, thus the control logic and interface among blocks can be verified without the presence of actual transistor circuits.

The VLSM is modelled only as a 4x4 array, or 16-byte history buffer, in order to simplify the address decoding and priority encoding logic descriptions. The verification involves the internal logic of all VLSM Words as well as their interconnections, and is independent of the array sizes. A two-phase (phi1 & phi2), non-overlapping clock at 10 MHz is used. In phi1, many signals are precharged high, such as H_MATCH, M, etc. In phi2, everything is evaluated.

Each VLSM Word is characterized by its inputs and outputs together with its internal states. These I/O signals and internal states are in turn manipulated as arrays whose sizes are the same as the history buffer size. For example, in the simulation, the HIT signals are declared as an array, HIT[15:0]; each element of the array is a bit.

Figure 5.1 shows the partial logic of a VLSM Word, whose index is n. In VERILOG, the 8-bit CAM Match Cell for Word #n is then modelled as CAM_MATCH_CELL[n], where CAM_MATCH_CELL is defined as an 8-bit quanity. DATA is represented as DATA[7:0] to indicate that it's a byte quantity. Figure 5.2 demonstrates how the logic in Figure 5.1 is described in VERILOG. The block is executed only when phi2 is positive, as indicated by the *@ posedge phi2* statement. *#AND_GATE_DELAY* means that the second IF statement is not executed until *AND_GATE_DELAY* clock cycles later. This is how delays can be modelled in VERILOG.

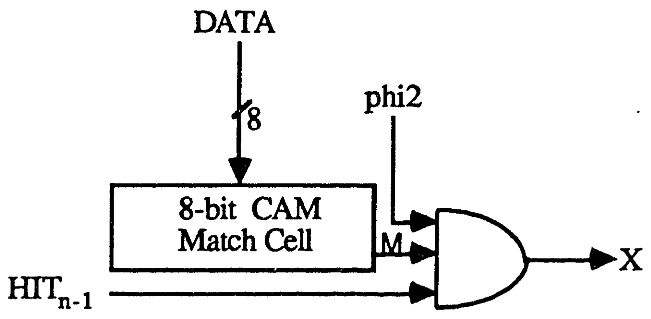


Figure 5.1 A Subsection of a VLSM Word #n

```
(@ posedge phi2)
IF (CAM_MATCH_CELL[n] == DATA[7:0])
        M[n] = TRUE;
#AND_GATE_DELAY
IF (M[n] && HITn-1[n])
        X[n] = TRUE
```

Figure 5.2 An Example of VERILOG Description

The four possible VLSM cycles, INIT, COMPARE, OUTPUT, and UPDATE, are described as different procedures, or states. In each state, the relevant signals are evaluated or modified. For example, one of the evaluations that occurs in a COMPARE cycle is shown in Figure 5.2. The Encoder FSM is implemented in a WHILE loop consisting of several IF-ELSE statements. Based on some key status signals such as CAM_HIT, different VLSM states are entered. A Bit Packer is also constructed to convert the codewords into byte-oriented outputs. Some rough delay estimates are included between each discrete events, such as the generation of H_MATCH signals, to make it more realistic. The complete listing of the VERILOG description is shown in APPENDIX C.

The simulation result proved to be positive, as the codewords were correctly generated given a filled history buffer and an arbitrary input string. The number of cycles required to produce the whole input string is as expected. The major issue raised in constructing the Encoder behavioral model is how to best detect the end of packet. Since the Interface Manager keeps the Packet Length information in its internal register, it naturally is a candidate to generate the EN_STOP signal to stop the Encoder when every character in a packet has been encoded. However, the problem is that the Interface Manager can only count the number of bytes that have been transferred into the DC Engine, not necessarily the number of bytes that has been processed by the Encoder.

One solution is to keep a separate counter in the Encoder. The counter is incremented every time a character is read from the Encoder Input FIFO; the counter value will be compared with the Packet Length register in hardware. When they match, the EN_STOP signal will be asserted to terminate the encoding process and force a codeword to be produced. Notice that a packet can be terminated under two different encoding scenarios: the last character of the packet has a match or does not have a match. In either case, a codeword has to be output.

Another solution is when the FIFO is empty, the Interface Manager will be notified by the EI_EMPTY signal. If there is no more data to be transferred into the DC Engine, i.e., end of packet, then EN_STOP can be asserted by the Interface Manager. In the period between the FIFO runs out of data and the time EN_STOP is asserted, the Encoder has to be put in a wait state.

The second solution is better because the Encoder could not encode the next packet right away without the FIFO being written with new data, which is coordinated by the Interface Manager. Therefore, the first solution does not really improve the throughput, since it has to be idle after the last codeword is output; yet, an extra counter and a comparator are needed.

5.2 FUTURE EXTENSION

The VERILOG program created for this thesis can be further extended to perform a more complete simulation. For example, the history buffer can be expanded to 1024 characters; the input strings can be read from a file that contains real data; compression ratio can be computed; and finally, the number of cycles required to encode an input file can be logged. Ideally, the program can also be written to encode with an initially empty history buffer, which is gradually built up as more input strings are encoded. The extended simulation program will be a useful tool to study problems such as the effect of history buffer size on compression ratio, the effective DC Engine throughput under different data source characteristics, etc.

CHAPTER 6

FUTURE WORK

6.1 DYNAMIC RAM VLSM IMPLEMENTATION

Throughout this thesis, high throughput has been a consistent priority when making a design tradeoff, particularly in the design of the VLSM. Therefore, the fast but low density static RAM (SRAM) structure is most suited to form the basis of a CAM bit cell. However, some applications might not need such high throughput provided by the VLSM, which accounts for nearly 40% of the chip area. For a better cost/performance ratio, reducing the silicon area becomes the major concern. In this situation, the slow but high density dynamic RAM (DRAM) structure is a good alternative to implement the CAM cell. Undoubtedly, DRAM requires more control, since it is a charge array. Periodically, the memory cells must be recharged, or refreshed, to retain their integrity. Moreover, every memory read is destructive, so it is necessary to "write back" the data that has been read. Nonetheless, its high density, as briefly mentioned in Section 3.1.5.1, makes DRAM implementation worth investigating.

6.2 CONTEXT SWITCHING

In a communications network, where there are multiple switching nodes, the history buffer used by the Encoder and the Decoder must be changed according to the context of the data being processed; otherwise, data can not be correctly decoded. This argument is best supported by an example. As shown in Figure 6.1, there is a wide area network that interconnects New York City (NY), Los Angeles (LA), Chicago, and Boston. The circles denote the switching nodes in each city. Suppose the data is compressed by the source node before being transmitted over the communication channel (represented by the straight lines), and is decompressed by the destination node. The following scenario will illustrate why context switching is necessary.

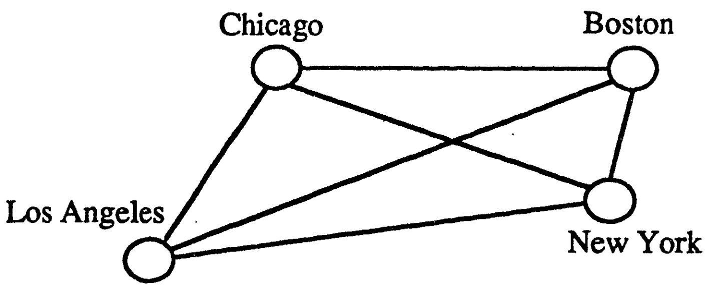


Figure 6.1 A Context Switching Example

Assuming no context switching takes place, i.e., the Encoder and Decoder in each node do not change their history buffers, regardless of the source of the data. In the beginning, when the network is first set up, everyone has empty history buffers. Let LA be the first node send a file, in the form of ZL77 codewords, to NY. NY builds up the history buffer as the codewords are decoded. After successful decompression, the history buffer in the NY Decoder now contains the data, whose "context" is LA. Next, Boston sends a file to NY. However, the file will never be correctly decoded, because NY uses the codewords from Boston to pull out the data from the LA file! It would be fine if NY recognizes that the file is from Boston, and before decoding, switches to the Decoder history buffer which has the context of Boston. In this particular example, the history buffer associated with Boston context is still empty, so NY should build it up from scratch, just like the LA file. It is clear that both the Encoder and Decoder need context switching to maintain data integrity.

Even in single link (point-to-point) topologies, multiple virtual circuits (VCs) must be treated separately to optimally compress their diverse source characteristics.

The architecture of the DC Engine has to be modified to accommodate the new requirement. Figure 6.2 shows the new general system configuration, in which an interface to a local memory, which stores the history buffers from different VCs, must be incorporated into the DC Engine. It is important to study how context switching will affect the overall Engine throughput, as each context switch will demand saving the changes in the current history buffer and then loading an entire new history buffer into the VLSM. A varieties of implementation are possible with different cost/performance tradeoffs. For example, the choice between SRAM or DRAM as local memory depends on the cost, memory size and speed requirements of the system. SRAM is faster but more expensive, while DRAM slower but cheaper and denser. If context switching overhead adversely affects the throughput, maybe multiple VLSMs should be employed, each dedicated to a particular VC. Or maybe the history buffer size should be reduced to minimize the context switching overhead but at the expense of degraded compression ratio. In short, context switching is an essential and interesting topic and should be further investigated.

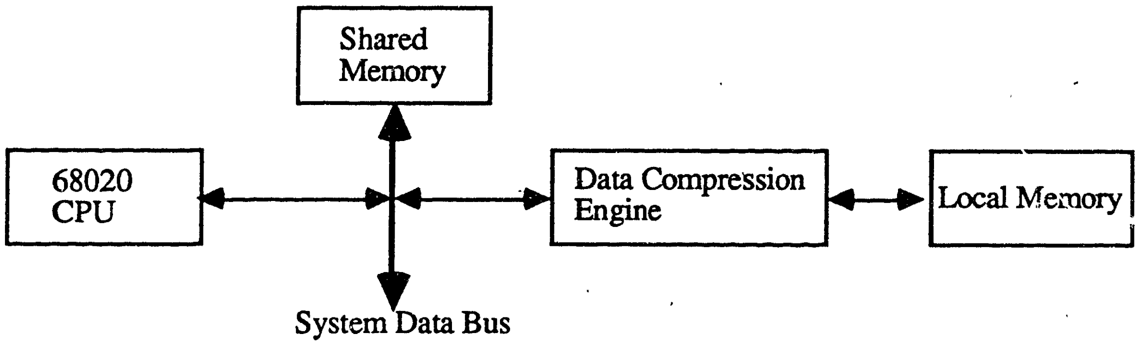


Figure 6.2 General System Configuration With Context Switching

6.3 Testing

Due to the architectural nature of this thesis, a testing circuit has not been included in the area estimates for the VLSM and other components. However, chip testing is vital in practice. As the chip complexity is increasing more rapidly than the number of I/O pins available to access the internal nodes, incorporating testing circuits on chip to enhance testability is not an option anymore. Today, VLSI circuit desigers are willing to sacrifice precious silicon area or performance for testing purpose in exchange for easier chip debugging after it is fabricated. Therefore, the actual transistor level design of the DC Engine must consider design for testability.

CHAPTER 7

CONCLUSION

A VLSI architecture suitable for ZL77 data compression algorithm encoding implementation, known as the VLSM, has been developed in this thesis. This structure utilizes the parallel searching capabilities of content addressable memory to tremendously facilitate the string matching process. It thus provides the potential for data compression in high bandwidth data communications networks.

Table 7.1 summarizes the area and throughput estimates of the special VLSI structures. The area is based on the 1 μm, two-layer metal CMOS technology and is expressed as a percentage of a 300 mil x 300 mil silicon area. The throughput is defined as the *input rate which each structure can sustain.* The estimates assume a 10 MHz system clock and are in units of Mbps. In the Encoder and Decoder, the Bit Packer and Unpacker are assumed to be clocked at 40 MHz.

Structure	Area (mil2)	Percentage	Throughput
VLSM	26,950	30	28
Bit Packer	600	.67	10
Bit Unpacker	600	.67	10
Encoder	32,500	36	28
Decoder	12,000	13	28

Table 7.1 Area and Throughput Summary

As can be seen, the Encoder (which includes the VLSM and Bit Packer) and the Decoder (which includes the Bit Unpacker) takes up about half of the given silicon area. There is plenty of room to implement the DMA Controller and Interface Manager.

APPENDIX A: Hashing Instruction Cycles Estimate

I. Instruction Format

A. ALU operations

```
opcode  dest <- op1  op2
opcode  dest <- op1  literal
```

B. Memory Access

```
ld      dest <- < op1 + op2 >
st      < reg > -> op1 + op2
```

II. Register Names

88000 instruction set is chosen to implement the hash table data structures because it has excellent instruction set for bit manipulations. Furthermore, there are 32 32-bit general registers in 88000. To enhance readability of the code, registers used in the program are given names as listed below:

A. The following registers contain the starting addresses for the hashing table structures:

HPT	Hash Pointer Table
HLT	Hash Link Table
HB	History Buffer
IB	Input Buffer

B. The following registers contain the offset into the tables:

HBEP	History Buffer End Pointer
IBP	Input Buffer Pointer

C. The following registers hold variables:

LMC	Longest Match Count
TMC	Temporary Match Count
LMA	Longest Match Address
TMA	Temporary Match Address
HV	Hash Value

D. The following registers are temporary variables:

TMP
RA
RB

E. This register ia a hardwired zero:

ZERO

There are 15 registers used in this program.

III. 88000 Code

```
/* THIS LOOP SEARCHES FOR THE LONGEST MATCH */

        /* calculate hash value for a given input substring */
begin   add   TMP   IB    IBP        ; TMP has the effective address of IBP
        jsr   hash

        /* initialize TMP, LMA & LMC */
        ld    TMP   HPT   HV         ; TMP contains the First entry (from the hash
                                       pointer table), which is both a offset pointer
                                       (ranging from 0 to 1023) to into history buffer
                                       and hash link table

        add   LMA   TMP   ZERO       ; initialize the longest match start address
        sub   LMC   LMC   LMC        ; initialize the longest match count to 0

        /* search loop for the next substring in the history buffer with the same hash value as
        the input */
loop0   add   TIBP  IBP   ZERO       ; initialize TIBP with IBP
        cmp   RA    TMP   ZERO       ; check for the end of linked list
        bb1   2     RA    update     ; if TMP = 0, then the end of the linked list
        sub   TMC   TMC   TMC        ; clear temp. match count
        add   TMA   TMP   ZERO       ; initialize the temp. longest match start address

        /* search loop within one substring with the same hash value as the input substring */
loop1   ld.b  RA    HB    TMP        ; RA contains the char from hist. buffer
        ld.b  RB    TIBP  ZERO       ; RB contains the char from input buffer
        cmp   RA    RA    RB
        bb1   3     RA    check      ; branch if RA != RB (i.e., a miss occurs)
        add   TIBP  TIBP  1          ; increment TIBP (temp input buffer pointer)
        add   TMP   TMP   1          ; increment TMP (offset to hist. buffer)
        add   TMC   TMC   1          ; increment TMC (temp. match count)
        jmp   loop1

        /* check whether LMC & LMA should be updated */
check   cmp   RA    TMC   LMC        ; compare TMC & LMC, result in RA
        bb1   7     RA    pass       ; pass update if TMC < LMC

        /* update LMC & LMA */
        add   LMC   TMC   ZERO       ; update LMC
        add   LMA   TMA   ZERO       ; update LMA

pass    add   TMP   HLT   TMP        ; obtain the offset pointer fo: the next substring from
                                       the hash link table
        jmp   loop0                  ; go to search the next substring (if any)
```

```
/* THE FOLLOWING BIG LOOP UPDATES THE HASH TABLES */

update add    RB    LMC   ZERO       ; assign RB with the longest match count
       cmp    RA    RB    ZERO       ; check if RB is zero
       bb1    2     RA    done       ; if yes, then done

       /* delete the oldest char from the hash tables */

       add    TMP   HB    HBEP       ; get effective address for end buffer pointer
       jsr    hash
       ld     TMP   HPT   HV         ; TMP contains the offset into history buffer,
                                       from First
       ld     TMP   HLP   TMP        ; TMP contains the 2nd link entry
       st     TMP   HPT   HV         ; update the First entry

       /* insert the new char into history buffer */

       ld     TMP   IB    IBP        ; TMP has the input char
       st     TMP   HB    HBEP       ; insert the input char into hist. buffer

       /* insert the link info for the new char hash tables */

       sub    HBEP  HBEP  1          ; subtract HBEP by 1
       add    TMP   HB    TMP        ; obtain effective address for the char before
                                       oldest char
       jsr    hash
       add    HV    HV    HV         ; calculate effective address
       add    RA    HV    1
       ld     TMP   HPT   RA         ; TMP has the Last entry
       add    HBEP  HBEP  1
       st     HBEP  HLT   TMP        ; extend the linked list
       st     HBEP  RA    ZERO       ; update the Last entry of hash pointer table

       /* update counter and pointers */

       add    HBEP  HBEP  1          ; increment hist. buffer end pointer
       add    IBP   IBP   1          ; increment input buffer pointer
       sub    RB    RB    1          ; decrement the count for # of char to be updated
       jmp    update

/* THE FOLLOWING SUBROUTINE CALCULATES HASH VALUE */

hash   ld.b   RA    TMP   ZERO       ; load the first char into RA
       add    TMP   TMP   1          ; increment the pointer
       ld.b   RB    TMP   ZERO       ; load the second char into RB

       shl    RA    2     RA         ; multiply the 1st char by 4
       add    HV    RA    RB
       and    HV    HV    $3f        : extract the lowest 6 bits
       add    HV    HV    HV         ; get the effective address (mul. HV by 2)
       rte                           ; return from subroutine
```

IV. Throughput

In calculating the number of instruction cycles required to find the longest match and update the history buffer, the following assumption is made:

- Average Longest Match Length = 3.5 (X)
- The search through each substring with the same hash value as the source is 2 (Y)
- Average linked list length is 16 (i.e., 16 substrings are searched per hash value) (Z)
- Load and Store intructions take two instruction cycles
- The 88000 can operate at 17 MIPs

Then, number of instructions per character =

$$\{ 8 + Z*[5+ (X*10) + 6] + Y*49 \} / X$$

With the above parameter values, it comes out to be **225 cycles / character**

APPENDIX B: C Code For Binary Search Tree

```
main()
{
    tree_init();

    while( NOT_END_OF_INPUT_STRING )
    {
      /* initialize the global variables */
      match_length[LEFT] = 0;
      match_length[RIGHT] = 0;
      current_node = root;

      read_char();  /* read in a new character, update window[N], look[F], lookahead */

      length = search();  /* return the longest matching string length & update the window */
                          /* index where the match starts */

      delete();  /* delete the oldest character in the window and update the tree */
    }
}

search()        /* return the length of the longest string match */
{
        int i, j;

        current_length = 0;
        for ( i = current_node, j=0; i < (current_node+F), j < F; i++, j++ )
        {
                if ( look[lookahead+j] == window[i] )
                {
                        current_length++;
                }
                else if ( look[lookahead+j] < window[i] )
                {
                        if ( tree[current_node].left_son == NIL )
                        {
                                tree[current_node].left_son = lookahead;
                                if ( current_length > match_length[RIGHT] )
                                {
                                        match_index = current_node;
                                        return(current_length);
                                }
                                else
                                {
                                        match_index = index[RIGHT];
                                        return(match_length[RIGHT]);
                                }
                        }
                        else
                        {
                                current_node = tree[current_node].left_son;
                                match_length[LEFT] = current_length;
```

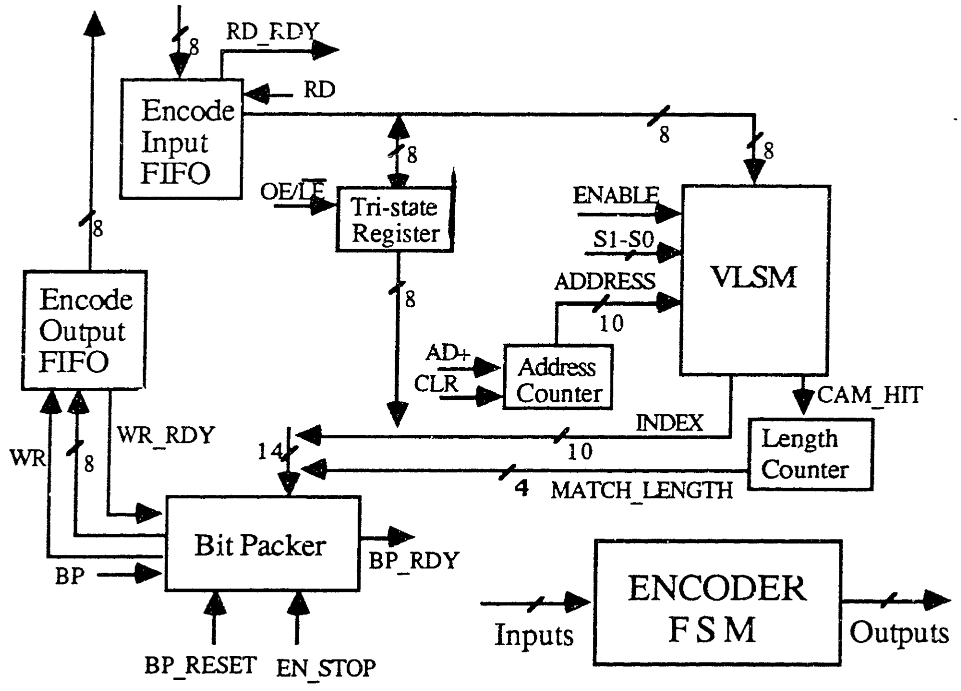
```
                    match_index[LEFT] = current_node;
                }
            }
            else
            {
                if ( tree[current_node].right_son == NIL )
                {
                    tree[current_node].right_son = lookahead;
                    if ( current_length > match_length[LEFT] )
                    {
                        match_index = current_node;
                        return(current_length);
                    }
                    else
                    {
                        match_index = index[LEFT];
                        return(match_length[LEFT]);
                    }
                }
                else
                {
                    current_node = tree[current_node].right_son;
                    match_length[RIGHT] = current_length;
                    match_index[RIGHT] = current_node;
                }
            }
        }
        search();
}

delete()
{
    int node_index, son_node;

    if (oldest_char == root)
    {
        if ( tree[oldest_char].left_son == NIL )
        /* the node to be deleted has no left child --> the right child becomes the new
            root */
        {
            root = tree[oldest_char].right_son;
            tree[root].parent = NIL;
            /* delete the oldest character */
            tree[oldest_char].parent = oldest_char;
            oldest_char++;
        }
        else if (tree[oldest_char].right_son == NIL )
        /* the node to be deleted has no right child --> the left child becomes the new
            root */
        {
            root = tree[oldest_char].left_son;
            tree[root].parent = NIL;
            /* delete the oldest character */
            tree[oldest_char].parent = oldest_char;
            oldest_char++;
        }
        else
        /* the node to be deleted has both left and right children --> the smallest element
            in the right subtree becomes the new root */
        {
            root = find_min(tree[oldest_char].right_son);
            tree[root].parent = NIL;
            tree[root].left_son = tree[oldest_char].left_son;
            tree[root].right_son = tree[oldest_char].rightt_son;
            /* delete the oldest character */
            tree[oldest_char].parent = oldest_char;
            oldest_char++;
        }
    }
    else
    /* the node to be deleted is not the root */
    {
        /* the node to be deleted is a terminal node */
        if ((tree[oldest_char].left_son == NIL) && (tree[oldest_char].right_son == NIL))
        {
            node_index = tree[oldest_char].parent;
            if (tree[node_index].left_son == oldest_char)
                tree[node_index].left_son == NIL;
            else
                tree[node_index].right_son == NIL;
            /* delete the node */
            tree[oldest_char].parent = oldest_char;
            oldest_char++;
        }
        else if ( tree[oldest_char].left_son == NIL )
        /* the node to be deleted has no left child --> the right child  replaces the node */
        {
            son_node = tree[oldest_char].right_son;
            node_index = tree[oldest_char].parent;
            if (tree[node_index].left_son == oldest_char)
                tree[node_index].left_son = son_node ;
            else
                tree[node_index].right_son == son_node;
            tree[son_node].parent = node_index;
            /* delete the oldest character */
            tree[oldest_char].parent = oldest_char;
            oldest_char++;
        }
        else if (tree[oldest_char].right_son == NIL )
        /* the node to be deleted has no right child --> the left child replaces the node */
        {
            son_node = tree[oldest_char].left_son;
            node_index = tree[oldest_char].parent;
            if (tree[node_index].right_son == oldest_char)
                tree[node_index].right_son = son_node ;
            else
                tree[node_index].left_son == son_node;
            tree[son_node].parent = node_index;
            /* delete the oldest character */
            tree[oldest_char].parent = oldest_char;
            oldest_char++;
        }
        else
        /* the node to be deleted has both left and right children --> the smaliest element
            in the right subtree replaces this node */
        {
            /* find the node to replace the node to be deleted */
            son_node = find_min(tree[oldest_char].right_son);

            node_index = tree[oldest_char].parent;
            if (tree[node_index].right_son == oldest_char)
                tree[node_index].right_son = son_node ;
            else
                tree[node_index].left_son == son_node;
            tree[son_node].parent = node_index;
            tree[son_node].left_son = tree[oldest_char].left_son;
            tree[son_node].right_son = tree[oldest_char].right_son;
            /* delete the oldest character */
            tree[oldest_char].parent = oldest_char;
            oldest_char++;
        }
    }
}


/* this function returns the index of the smallest element in a given binary
    search tree */
find_min(node_index)
int node_index;
{
    if (tree[node_index].left_son == NIL)
        return(node_index);
    else
        find_min(tree[node_index].left_son);
}
```

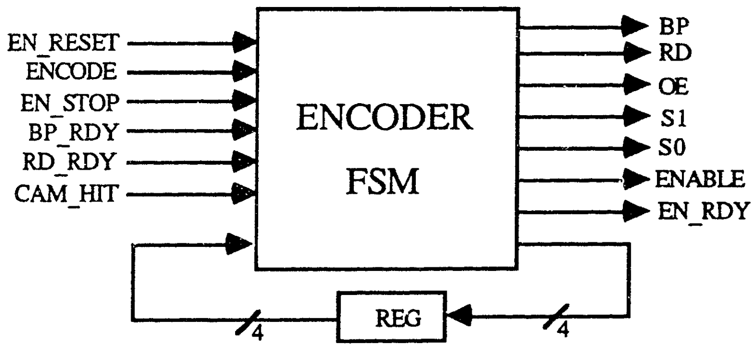
APPENDIX C: Alternative Encoder Implementation

I. Introduction

In Section 4.3.2.2, a different VLSM control sequence was discussed. By adding an extra flip flop in each VLSM Word, the Character Buffer and Buffer Counter are no longer necessary. The consequence is a smaller Encoder FSM due to fewer number of inputs and outputs required. The next section shows the resulting Encoder architecture.

II. Encoder Architecture

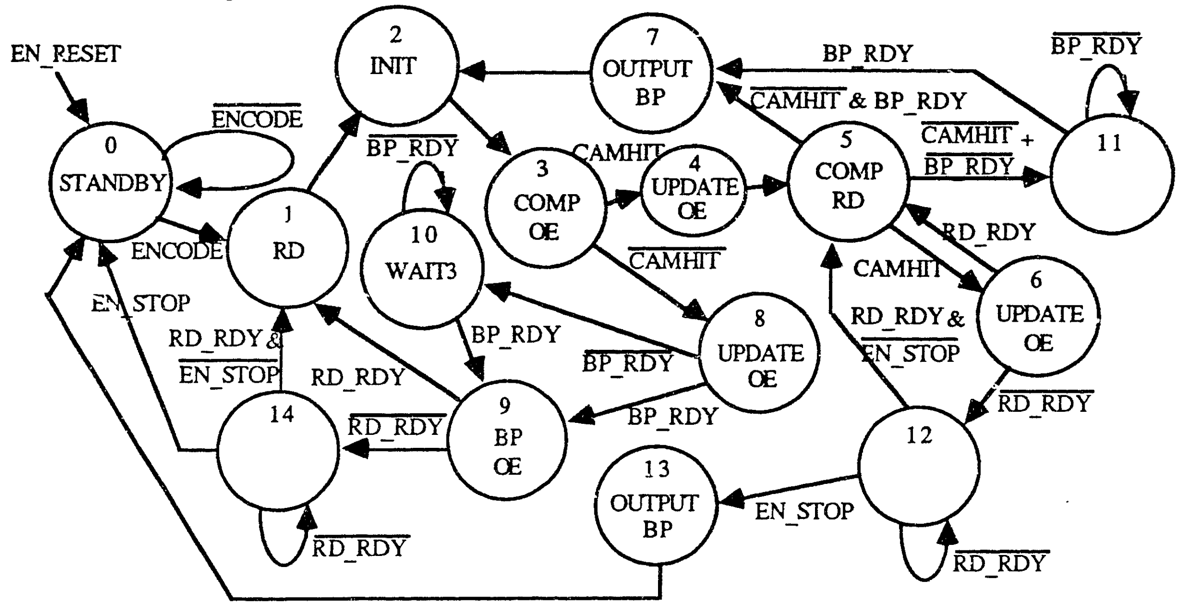




Note: AD+ is always asserted in the UPDATE cycle; CLR is the same as EN_RESET

ENABLE	S1	S0	INSTRUCTION
0	x	x	NOP
1	0	0	INIT
1	0	1	COMPARE
1	1	0	OUTPUT
1	1	1	UPDATE

III. State Diagram



IV. Activities in Each Encoder FSM State

State 0: Encoder idle/reset state; EN_RDY is asserted

State 1: Read a char from Encode Input FIFO; Latch the char into Tri-state Register

State 2: Initialize the VLSM to set up for comparison

State 3: Output-enable Tri-state Register; COMPARE cycle

State 4: Output-enable Tri-state Register; UPDATE cycle

State 5: Read a char from Encode Input FIFO; COMPARE cycle

State 6: Output-enable Tri-state Register; UPDATE cycle

State 7: OUTPUT cycle; bit pack the codeword

State 8: Output-enable Tri-state Register; UPDATE cycle

State 9: Output-enable Tri-state Register; bit pack the codeword

State 10: Waiting for Bit Packer to be ready

State 11: Waiting for Bit Packer to be ready

State 12: Waiting for Encode Input Buffer to be ready

State 13: OUTPUT cycle; bit pack the codeword

State 14: Waiting for Encode Input Buffer to be ready

When a no match occurs, states 8, 9, and 1 are traversed. The next string search starts at 1 because a new character needs to be fetched from the FIFO.

When it's the end of the packet, as indicated by the assertion of EN_STOP, the codeword will be generated under all circumstances. The Bit Packer will also be forced to output the last byte which might contain useless bits. States 12 and 14 are entered when the FIFO is not ready. EN_STOP is checked in these two states.

APPENDIX D: VERILOG Simulation Code

```
/* This module models the Variable-Length String Matcher (VLSM), which is based on

   Content Addressable Memory (CAM). VLSM has four modes of operations: INIT, COMPARE,

   UPDATE, and OUTPUT.  An external finite state machine is required to control the VLSM

   to find the longest match.  For now, the FSM is also modelled in this module */


/* Created by Brian T. Hou          October 19, 1988     */

/* Codex Corporation     */

module VLSM (enable, s1, s0, address, target, index, camhit);
    input enable, s1, s0, address, target;
    output index, camhit;

    /* VLSM Array Parameters */
    parameter
        WORD_SIZE = 8,        /* Word Bit Length --> 8 for an ASCII char */
        ARRAY_SIZE = 16,      /* History Buffer Size */
        ADDR_SIZE = 4,        /* Address Bit Length = log2(ARRAY_SIZE) */
        ROW_SIZE = 4,         /* Row size for the array */
        COLUMN_SIZE = 4,      /* Column size for the array */
        MAX_LENGTH = 15;      /* Maximum match length allowed */

    /* Encoder FIFO size parameters */
    parameter
        EI_FIFO_SIZE = 16,  /* Size of the Encoder Input FIFO */
        EO_FIFO_SIZE = 8;   /* Size of the Encoder Output FIFO */

    /* Bit Packer shift register size parameters */
    parameter
        CODEWORD_SIZE = 7,  /* Bit Length of the codeword */
        TEMPLATE_SIZE = 8,  /* Bit Length of the packed data */
        LEN_SIZE = 3;       /* Bit Length of the Length field */

    /* VLSM Delay Parameters (units in ns) */
    parameter
        m_delay = 2,
        h_match_delay = 5,
        v_match_delay = 5,
    addr_encode_delay = 5,
    addr_decode_delay = 5;

/* VLSM block level i/o signals */
reg [ADDR_SIZE-1:0] index;
reg [ADDR_SIZE-1:0] address;
reg enable, s1, s0, cam_hit;

/* Control signals and interconnects among VLSM Word Cells */
reg [ROW_SIZE-1:0]  encode;
reg [ROW_SIZE-1:0]  h_match;
reg [1:0]  row;
reg [COLUMN_SIZE-1:0]  v_match;
reg [ARRAY_SIZE-1:0] y;
reg [ARRAY_SIZE-1:0] z;
reg [ARRAY_SIZE-1:0] m;
reg [ARRAY_SIZE-1:0] hit_in;
reg [ARRAY_SIZE-1:0] hit_out;
reg [ARRAY_SIZE-1:0] hit;
reg [ARRAY_SIZE-1:0] vhit;
reg test_cam_hit;

reg [WORD_SIZE-1:0] show [0:MAX_LENGTH];

/* History buffer of size ARRAY_SIZE, 8-bit each */
reg [WORD_SIZE-1:0] data [0:ARRAY_SIZE-1];

/* VLSM input character register */
reg [WORD_SIZE-1:0] target;

/* Encoder tri-state register */
reg [WORD_SIZE-1:0] tri_state_reg;

/* Encoder Input FIFO memory */
reg [WORD_SIZE-1:0] EI_data [0:EI_FIFO_SIZE-1];

/* Encoder Input FIFO Pointer */
integer EI_ptr;

/* VLSM two-phase non-overlapping clocks */
reg phi1, phi2;
```

```
/* VLSM State Variables */
reg INIT_DONE, DO_COMPARE, DO_WRITE;
reg EI_EMPTY, NEW_STRING, FINISH, ESCAPE;

integer i, j, k, len, addr, cnt;

initial
    begin
        /* Initialize VLSM I/O */
        test_cam_hit = 1;
        len = 0;
        addr = 0;
        cnt = 0;
        row[1:0] = 2'bxx;
        index[ADDR_SIZE-1:0] = 4'bxxxx;

        /* Initialize the VLSM states */
        INIT_DONE = 0;
        DO_COMPARE = 0;
        DO_WRITE = 0;
        EI_EMPTY = 0;
        NEW_STRING = 0;
        FINISH = 0;
        ESCAPE = 0;

        /* Initialize EI_FIFO pointer */
        EI_ptr = 0;

        /* Fill the history buffer from the file History.buffer.
           We assume that at the start of simulation, the history buffer is already
           full.  The history buffer pointer is set to 0.  This means that VLSM Word #15
           contains the most recently encoded character    */

        $readmemh("History.buffer", data);

        /* Initialize the match status */
        for (i = 0; i < ARRAY_SIZE; i = i + 1)
            begin
                hit_in[i] = 0;
                hit_out[i] = 0;
                hit[i] = 0;
            end
    end   /* End  initial */

always @(posedge phi1)
    begin

        /* The following lines are precharged high during phi1 */

        for (i = 0; i < ROW_SIZE; i = i + 1)
            h_match[i] = 1;
        for (i = 0; i < COLUMN_SIZE; i = i + 1)
            v_match[i] = 1;
        for (i = 0; i < ARRAY_SIZE; i = i + 1)
            m[i] = 1;
        cam_hit = 1;

        /* Determine whether a history update should take place */
        if (INIT_DONE)
        begin
            if (test_cam_hit)
                DO_WRITE = 1;
            else if (!test_cam_hit && (len == 0))
                begin
                    DO_WRITE = 1;
                    ESCAPE = 1;   /* Indicate special condition */
                end

            else
                DO_WRITE = 0;
        end

        /* VLSM function decoding */
 /*        if (enable)
            begin
                if (!s1 & !s0)  init;
                else if (!s1 & s0)  compare;
                else if (s1 & !s0)  write;
                else get_index;
            end
 */
    end

 always @(posedge phi2)
```

```
begin
    fork
        /* The following simulates the two cascated flip-flop action
           inside each CAM Word Cell.  y and z are the input and output
           of the first flip-flop, and z and hit_out are the input and
           output of the second flip-flop                              */
        begin
            #1 for (i = 0; i < ARRAY_SIZE; i = i + 1)
                hit_out[i] = z[i];
            #1 for (i = 0; i < ARRAY_SIZE; i = i + 1)
                z[i] = y[i];
            #5 for (i = 0; i < ARRAY_SIZE; i = i + 1)
                y[i] = hit[i];
        end

        /* Connect the hit_out[n] to hit_in[n+1] between CAM Word Cells */
        #1 for (i = 1; i < ARRAY_SIZE; i = i + 1)
            hit_in[i] = hit_out[i-1];
        #1 hit_in[0] = hit_out[ARRAY_SIZE-1];

        /* VLSM Finite State Machine */
        begin
            if (INIT_DONE)
            begin
                if (DO_COMPARE)
                    begin
                        if (NEW_STRING)
                            begin
                              /* If processing a new string, then target data is the last char
                              fetched from the EI_FIFO, which was stored in tri_state register */
                                target = tri_state_reg;
                                NEW_STRING = 0;
                            end
                        else
                            /* Fetch a new character from Encode Input FIFO */
                            begin
                                EI_FIFO(EI_ptr,target,EI_EMPTY);
                                tri_state_reg = target;
                            end

                        if(EI_EMPTY) /* Either the packet is finished
                                        or a DMA transfer is necessary */
```

```
                                    begin
                                        FINISH = 1;
                                        target = 'hxx;
                                    end
                                /* $display($time,,"target = %c",target);  */
                                compare;
                                DO_COMPARE = 0;
                                /* $display($time,,"COMPARE task is finished");  */
                            end
                        else if (DO_WRITE)
                            begin
                                write;

                                /*  For display purpose  */
                                show[cnt] = target;
                                $display("show[%d] = %c",cnt,show[cnt]);
                                cnt = cnt + 1;
                            end
                        else get_index(cnt);
                    end
                    else
                        /* If not in COMPARE, WRITE, or GET_INDEX states, go to INIT */
                        #3 init;
                end
            join
        end

    task EI_FIFO;
        inout EI_ptr;
        output [WORD_SIZE-1:0] target;
        output EI_EMPTY;
        integer EI_ptr;
        begin
            /* Assume the EI_FIFO is full to start with */

            if (EI_ptr == EI_FIFO_SIZE)
                EI_EMPTY = 1;
            else
                begin
                    target = EI_data[EI_ptr];
                    /* $display($time,,"EI_FIFO EI_ptr = %d EI_data = %c",EI_ptr,EI_data[EI_ptr]);  */
                EI_ptr = EI_ptr + 1;
            end
    end
endtask


task init;
    begin
        /* $display($time,,"INIT time");  */

        /* Force hit_out to be true */
        for (i = 0; i < ARRAY_SIZE; i = i + 1)
            z[i] = 1;
        DO_COMPARE = 1;
        INIT_DONE = 1;
    end
endtask

task compare;
    begin

        /* $display($time,,"Inside Compare task"); */

        /* Check the content of each CAM Word Cell with target data.      */
        /* If a match occurs, the respective m's remains high. Otherwise */
        /* m's will be low.                                               */
        #m_delay
        for (i = 0; i < ARRAY_SIZE; i = i + 1)
            begin
                if (data[i] == target) m[i] = 1;
                else m[i] = 0;
            end

        /* If both m[n-1] and hit_in[n-1] are true, then hit[n] is true */
        for (i = 0; i < ARRAY_SIZE; i = i + 1)
            begin
                if (m[i] & hit_in[i])  hit[i] = 1;
                else hit[i] = 0;
            end

        /* If both encode and hit_out are true, then vhit is true */
        j = 0;
```

```
for (i = 0; i < ROW_SIZE; i = i + 1)
    begin
        if (encode[i]) /* If Row i is enabled */
            begin
                for (k = 0; k < COLUMN_SIZE; k = k + 1)
                    begin
                        if (hit_out[j+k]) vhit[j+k] = 1;
                        else vhit[j+k] = 0;
                    end
            end
        else  /* Else, vhit[] for that column is zero */
            for (k = 0; k < COLUMN_SIZE; k = k + 1)
                vhit[j+k] = 0;
        j = j + 4;
    end
    #1
for (i = 0; i < ROW_SIZE; i = i + 1)
    encode[i] = 0;

fork
    j = 0;

    /* If at least one CAM Word Cell has a hit, h_match for that row */
    /* will be pulled low after h_match_delay.                        */
    #h_match_delay
        for (i = 0; i < ROW_SIZE; i = i + 1)
            begin
                if (hit[j])  h_match[i] = 0;
                else if (hit[j+1])  h_match[i] = 0;
                else if (hit[j+2])  h_match[i] = 0;
                else if (hit[j+3])  h_match[i] = 0;
                j = j + 4;
            end

    /* If at least one CAM Word Cell has a vhit, v_match for that  */
    /* column will be pulled low after v_match_delay.              */
    #v_match_delay
        for (i = 0; i < COLUMN_SIZE; i = i + 1)
            begin
                if (vhit[i])  v_match[i] = 0;
                else if (vhit[i+4])  v_match[i] = 0;
                else if (vhit[i+8])  v_match[i] = 0;
                else if (vhit[i+12])  v_match[i] = 0;
            end
join

/* Row Address flip flop */
index[3:2] = row[1:0];
/* $display($time,,"index[3:2] = %b%b", index[3],index[2]);    */

#addr_encode_delay
fork
    begin

        /* Row address priority encoding */
        if (h_match[0] == 0)
            begin
                row[1:0] = 2'b00;
                encode[0] = 1;      /* Enable Row 1 */
                /* $display($time,,"00 row[1:0] = %b%b", row[1],row[0]);  */

            end
        else if (h_match[1] == 0)
            begin
                row[1:0] = 2'b01;
                encode[1] = 1;      /* Enable Row 2 */
                /* $display($time,,"01 row[1:0] = %b%b", row[1],row[0]);  */
            end

        else if (h_match[2] == 0)
            begin
                row[1:0] = 2'b10;
                encode[2] = 1;      /* Enable Row 3 */
                /* $display($time,,"10 row[1:0] = %b%b", row[1],row[0]);  */
            end

        else if (h_match[3] == 0)
            begin
                row[1:0] = 2'b11;
                encode[3] = 1;      /* Enable Row 4 */
                /* $display($time,,"11 row[1:0] = %b%b", row[1],row[0]);  */
            end

        /*  else cam_hit = 0;  */
```

```
                        else
                            begin
                                test_cam_hit = 0;
                                row[1:0] = 2'bxx;
                            end
                    end

                    /* Column address priority encoder */
                    begin
                        if (v_match[0] == 0) index[1:0] = 2'b00;
                        else if (v_match[1] == 0) index[1:0] = 2'b01;
                        else if (v_match[2] == 0) index[1:0] = 2'b10;
                        else if (v_match[3] == 0) index[1:0] = 2'b11;
                        /* $display($time,,"index[1:0] = %b%b", index[1],index[0]); */

                    end
                join

        end
    endtask

    /* Update the history buffer */
    task write;
        begin
            data[addr] = target;

            /* If the end of history buffer, then wrap around */
            if (addr == ARRAY_SIZE)  addr = 0;
            else addr = addr + 1;   /* Increment addr */

            /* If escape sequence (i.e. len = 0) then you don't want to do COMPARE next */
            /* Instead, you want to get the codeword and go on to fetch a new char from FIFO */
            if (ESCAPE)
                begin
                    ESCAPE = 0;     /* Reset ESCAPE */
                    get_index(cnt);
                end
            else
                begin
                    DO_COMPARE = 1;
                    if (len == MAX_LENGTH) get_index(cnt);
                    else len = len + 1;     /* Increment len */
                end

            /* $display($time,,"Write is executed at this time");  */
        end
    endtask

    /* Print the index and match length */
    task get_index;
        inout cnt;

        begin
            #addr_encode_delay

            if (len == 0)
                /* If there is no match, the raw character is put in the Index field */
                begin
                    for (i = 0; i < ADDR_SIZE; i = i + 1)
                        index[i] = 1'bx;
                end
            else
                begin
                    /* Calculate the starting address */
                    index[3:0] = index[3:0] - len + 1;

                    /* Check for boundary condition */
                    if (index[3:0] < 0)
                        index[3:0] = index[3:0] + ARRAY_SIZE;
                    /* Get ready for the next input string */
                    NEW_STRING = 1;
                end

            /* Display the simulation result */
            $display("");
            for(i = 0; i < cnt; cnt = cnt + 1)
                $display("%c", show[i]);
            $display("is encoded as < %d, %d >", index[3:0],len);

            /* Initialize len */
            cnt = 0;
            len = 0;
            test_cam_hit = 1;
            if (FINISH)
                begin
                    $display("");
                    $display($time,,"Encoder Input FIFO is empty!!");
                    $finish;
                end
            else
                init;

        end
    endtask

    /* Generate two-phase non-overlapping clocks */
    always
        begin
            #5  phi1 = 1;
            #45 phi1 = 0;
            #5  phi2 = 1;
            #45 phi2 = 0;
        end

endmodule
```

```
/* This module takes in non-byte oriented data and bundle them into byte oriented quantities */
module bit_packer;
    /* input [6:0] codeword;  */

    /* Bit Packer size parameters */
    parameter
        CODEWORD_SIZE = 14,
        TEMPLATE_SIZE = 8,
        COUNTER_SIZE = 4,
        COUNTDOWN_SIZE = 4;

    /* Bit Packer Controller state assignments */
    parameter
        DEFAULT_STATE = 3'b100,
        SHIFT_STATE = 3'b001,
        WRITE_STATE = 3'b011,
        WAIT_STATE = 3'b010,
        RESET_STATE = 3'b000;

    /* Bit Packer counter values */
    reg [COUNTER_SIZE-1:0] counter;
    reg [COUNTDOWN_SIZE-1:0] countdown;

    /* Bit Packer block level control status */
    reg BP,
        BP_RDY,
        WR_RDY,
        BP_RESET,
        EN_STOP;

    /* Encode Output FIFO parameters */
    parameter
        EO_FIFO_SIZE = 10;

    /* EO_FIFO variables */
    integer EO_ptr;
    reg EO_FULL;
    reg [TEMPLATE_SIZE-1:0] EO_data [EO_FIFO_SIZE-1:0];

    /* Bit Packer storage elements */
    reg [2:0] STATE;
    reg [TEMPLATE_SIZE-1:0] template;
reg [CODEWORD_SIZE-1:0] code;
reg [CODEWORD_SIZE-1:0] codeword;

reg MORE;
reg CONT;

integer i;

reg bp_phi1, bp_phi2;

initial
    begin
        STATE[2:0] = DEFAULT_STATE;
        BP = 0;
        WR_RDY = 1;
        BP_RESET = 0;
        counter[COUNTER_SIZE-1:0] = 0;
        CONT = 1;
        codeword[CODEWORD_SIZE-1:0] = 14'b10000000000001;
        BP = 1;
        EO_ptr = 0;
        EO_FULL = 0;
        #513 codeword[CODEWORD_SIZE-1:0] = 14'b10110011110010;
        #1 BP = 1;
        #300 $finish;
    end

always @(posedge bp_phi2)
    begin
        if(STATE[2:0] == DEFAULT_STATE)
            begin
                if (BP)  /* If BP is asserted, then get ready for bit packing operation */
                    begin
                        BP_RDY = 0;  /* Reset BP_RDY to indicate that Bip Packer is busy */
                        BP = 0;      /* Usually an external controller
                                        will negate BP after one clock cycle */
                        MORE = 1;

                        /* Preset the countdown counter to CODEWORD_SIZE */
                        countdown[COUNTDOWN_SIZE-1:0] = CODEWORD_SIZE;

                        /* Load code shift register with the codeword */
                    code[CODEWORD_SIZE-1:0] = codeword[CODEWORD_SIZE-1:0];
                    $display($time,,"code[] = %h",code[CODEWORD_SIZE-1:0]);

                    /* Prepare for state transition */
                    STATE[2:0] = SHIFT_STATE;
                end

            else
                /* In DEFAULT_STATE, BP_RDY is asserted to indicate that Bit Packer is ready */
                BP_RDY = 1;
            $display($time,,"I am in DEFAULT_STATE");
        end

    else if (STATE[2:0] == SHIFT_STATE)
        begin
            /* Shift the template shift register */
            for(i = 0; i < (TEMPLATE_SIZE-1); i = i + 1)
                template[i] = template[i+1];
            template[TEMPLATE_SIZE-1] = code[0];

            /* Shift the code shift register */
            for(i = 0; i < (CODEWORD_SIZE-1); i = i + 1)
                code[i] = code[i+1];

            /* Decrement the countdown counter */
            countdown[COUNTDOWN_SIZE-1:0] = countdown[COUNTDOWN_SIZE-1:0] - 1;

            /* Increment the count-to-TEMPLATE_SIZE counter */
            counter[COUNTER_SIZE-1:0] = counter[COUNTER_SIZE-1:0] + 1;

            if (counter[COUNTER_SIZE-1:0] == TEMPLATE_SIZE)
                CONT = 0;
            $display($time,,"counter = %d  CONT = %b",counter[COUNTER_SIZE-1:0], CONT);

            if (countdown[COUNTDOWN_SIZE-1:0] == 0)
                MORE = 0;
            $display($time,,"countdown = %d  MORE = %b",countdown[COUNTDOWN_SIZE-1:0],MORE);

            /* If the codeword has been completely shifted into the template,
               then return to DEFAULT_STATE */
            if(!MORE && CONT)
                STATE[2:0] = DEFAULT_STATE;
        /* If template is full and EO_FIFO is ready to accept a character,
             then go to the WRITE_STATE */
        else if(!CONT && WR_RDY)
            STATE[2:0] = WRITE_STATE;

        /* If EO_FIFO is not ready, then go to WAIT_STATE */
        else if(!CONT && !WR_RDY)
            STATE[2:0] = WAIT_STATE;
    end

else if(STATE[2:0] == WRITE_STATE)
    begin
        EO_FIFO(template[TEMPLATE_SIZE-1:0], EO_FULL);  /* Write the template into EO_FIFO */
        $display($time,,"template = %h", template[TEMPLATE_SIZE-1:0]);
        if(MORE)   /* if codeword is not completely shifted into the template,
                      return to SHIFT_STATE */
            STATE[2:0] = SHIFT_STATE;

        else if(!MORE || EN_STOP)
            STATE[2:0] = DEFAULT_STATE;

        CONT = 1;
        counter[COUNTER_SIZE-1:0] = 0;  /* Clear count to TEMPLATE_SIZE counter */
        $display($time,,"I am in WRITE_STATE");

    end

else if(STATE[2:0] == WAIT_STATE)
    begin
        if(WR_RDY)
            STATE[2:0] = WRITE_STATE;
        $display($time,,"I am in WAIT_STATE");

    end

else if(STATE[2:0] == RESET_STATE)
    begin
        counter[COUNTER_SIZE-1:0] = 0;
        CONT = 1;
        STATE[2:0] = DEFAULT_STATE;
        $display($time,,"I am in RESET_STATE");
    end
```

```
        end

    always @(posedge BP_RESET)
        STATE[2:0] = RESET_STATE;

    task EO_FIFO;
        input [TEMPLATE_SIZE-1:0] packed_data;
        output EO_FULL;

        begin
            /* Assume the EI_FIFO is empty to start with */

            if (EO_ptr == EO_FIFO_SIZE)
                EO_FULL = 1;
            else
                begin
                    EO_data[EO_ptr] = packed_data;
                    $display($time,,"EO_FIFO EO_ptr = %d EO_data = %h",EO_ptr,EO_data[EO_ptr]);

                    EO_ptr = EO_ptr + 1;
                end
        end
    endtask


    /* Generate two-phase non-overlapping clocks */
    always
        begin
            #2  bp_phi1 = 1;
            #11 bp_phi1 = 0;
            #1  bp_phi2 = 1;
            #11 bp_phi2 = 0;
        end

endmodule
```

References

[1]. R. Gallager, *Information Theory and Reliable Communication*, John Wiley and Sons, Inc., New York , 1968.

[2]. D. A. Lelewer and D. S. Hirschberg, "Data Compression," *ACM Computing Surveys*, Vol. 19, No. 3, September 1987, pp. 261-295.

[3]. J. A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Inc., 1988.

[4]. D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, 40, 1962, pp. 1098-1101.

[5]. G. V. Cormak and R. N. Horspool, "Algorithm for Adaptive Huffman Codes," *Information Processing Letter*, Vol. 18, No. 3, 1984, pp. 159-165.

[6]. R. Gallager, "Variations On a Theme by Huffman," *IEEE Trans. Information Theory*, Vol. IT-24, No. 6, 1978, pp. 668-674.

[7]. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Information Theory*, Vol. IT-23, No.3, May 1977, pp. 337-343.

[8]. T. C. Bell, "Better OPM/L Text Compression," *IEEE Trans. on Communications*, Vol. COM-34, No. 12, December 1986, pp. 1176-1182.

[9]. J. A. Storer and T. G. Szymanki, "Data Compression via Textual Substitution," *Journal of the Asso. for Computing Machinery*, Vol. 29, No. 4, October 1982, pp. 928-951.

[10]. T. A. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, June 1984, pp. 8-19.

[11]. M. Snyderman and B. Hunt, "The Myriad Virtues of Text Compaction," *Datamation*, December 1, 1970, pp. 36-40.

[12]. M. E. Gonzalez Smith and J. A. Storer, "Parallel Algorithms For Data Compression," *Journal of the Asso. for Computing Machinery*, Vol. 32, No. 2, April 1985, pp. 344-373.

[13]. M. Rodeh, V. R. Pratt, and S. Even, "Linear Algorithm for Data Compression via String Matching," *Journal of the Asso. for Computing Machinery*, Vol. 28, No. 1, January 1981, pp. 16-24.

[14]. D. G. Serverance, " A Practioner's Guide to Database Compression," *Information Systems*, Vol. 8, No. 1, 1983, pp. 51-62.

[15]. F. Rubin, "Experiment in Text File Compression," *Comm. ACM*, Vol. 19, No. 11, November 1976, pp. 617-623.

[16]. D. Gottlieb, S. A. Hagerth, P. G. H. Lehot, and H. S. Rabinowitz, "A Classification of Compression Methods and Their Usefulness for a Large Data Processing Center," *Proc. Nat. Comput. Conf.*, Vol. 44, 1975, pp. 453-458.

[17]. G. Held, "Data Compression Devices," Auerback Publishers Inc., 1986.

[18]. S. J. Adams, M.J. Irwin, R. M. Orwens, "A Parallel General Purpose CAM Architecture," Leiserson, editor, *Advanced Research in VLSI, Proceedings of the Fourth MIT Conference*, 1986, pp. 51-72.

[19]. T. Kohonen, *Content Addressable Memories*, Springer-Verlag, 1980.

[20]. P. J. Osler, "A Prototype Content Addressable Memory System," Master Thesis, Dept. of Electrical Engineering and Computer Science, MIT, 1987.

[21]. R. M. Karp et al., "Rapid Identification of Repeated Patterns In Strings, Trees, and Arrays," *Proc. of the Fourth ACM Symposium on Theory of Computing*, 1972, pp. 125-136.

[22]. N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[23]. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1980.

[24]. C. G. Bell, J. C. Mudge, and J. E. McNamara, *Computer Engineering*, Digital Press, 1978.

[25]. D. A. Patterson, "Reduced Instruction Set Computers," *Comm. ACM*, Vol. 28, No. 1, January 1985, pp. 8-21.

[26]. M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures For VLSI*, The MIT Press, Cambridge, MA, 1985.

[27]. J. Mick and J. Brick, *Bit-Slice Microprocessor Design*, McGraw-Hill Book Company, New York, 1980.

[28]. J. F. Korsh, *Data Structures, Algorithms, and Program Styles*, PWS Computer Science, 1986.

[29]. C. Sodini et al, "The MIT Database Accelerator: A Novel Content Addressable Memory," Dept. of Electrical Engineering and Computer Science, MIT, Cambridge.

[30]. J. L. Mundy, "High Density Four-Transistor MOS Content Addressable Memory," *U.S. Patent No. 3,701,980*, October 31, 1982.

[31]. J. L. Mundy, J. F. Burgess, R. E. Joynston, C. Neugebauer, "Low Cost Associative Memory," *IEEE Journal of Solid State Circuits*, Vol. SC-7, October 1972, pp. 364-369.

[32]. J. P. Wade, "An Integrated Content Addressable Memory System," Ph. D. Dissertation, Dept. of Electrical Engineering and Computer Science, MIT, 1988.

[33]. C. C. Foster, *Content Addressable Parallel Processor*, Van Nostrand Reinhold, 1976.

[34]. C. Weems, S. Levitan, C. Foster, "Titanic: A VLSI Based Content Addressable Array Processor," *Proc. IEEE Int. Conf. on Circuits and Computers ICCC '82*, p. 236.

[35]. S. Berkovich, J. M. Pullen, *Proc. of the IEEE Int. Conf. on Computer Design: VLSI in Computers ICCD'84, p. 382.*

[36]. D. Parkinson, H. M. Liddell, "The Measurement of Performance On a Highly Parallel System," *IEEE Trans. On Computers*, Vol. C-32, No. 1, January, 1983, pp. 32-37.

[37]. B. Adair, "A FIFO Building Block", *Internal Memo*, Codex Corporation, 1988.

[38]. M. J. Foster, H. T. Kung, "The Design of Special-Purpose VLSI Chips", *IEEE Computer*, January, 1980, pp. 26-40.