Data Compression With MUSIC CAMs

Conventional data compression schemes can be implemented with minimal hardware when using a MUSIC Semiconductors CAM (content-addressable memory). CAMs are fully associative memory devices that can replace the binary trees or hash tables normally found in data compression algorithms. Since a good portion of a compression algorithm's time is spent searching and maintaining these data structures, replacing them with a hardware search engine can greatly increase the throughput of the algorithm.

Associative memories (CAMs) operate in a converse way to their counterparts, RAMs (random access memories). In RAM, an address is presented to the device and that particular location is accessed. Data may be written to or read from this location. In CAM, data is presented to the device, and the address that contains the requested data appears as output. For this reason, the CAM is a very efficient search engine. Locating specific data in a table can be performed much more quickly than by conventional software techniques using RAM. A CAM will generate a result in a single transaction, regardless of table size or length of search list. This makes the CAM an ideal candidate for data compression schemes that use sparsely populated tables as part of their algorithm.

DATA COMPRESSION

The theory behind data compression is simply to remove any redundancy that resides in a given piece of information, producing an equivalent but shorter message. If the original data stream can be recovered from the compressed stream with complete integrity, the method is known as lossless. Because no data corruption can be tolerated in lossless systems, compression ratios generally only range from about 1.5:1 to 5:1 (depending on the algorithm and the source material). Lossless compression should be distinguished from the so-called lossy (or compaction) techniques. Data compaction basically reduces a volume of information by omitting some of the data. The key element to efficient utilization of compaction techniques is determining which information to omit. These techniques seem to work well for audio and video material, where superfluous information is easier to identify and where perfect reproduction is not a requirement of the human sensory system. Compression ratios as high as 100:1 can be achieved, depending on the source material and the subjective level of accuracy required.

A number of lossless compression techniques with varying capabilities (compression efficiency and throughput) have been introduced throughout the years. The trade-off seems to remain invariable - the simple algorithms execute very quickly, yet they generally have poor compression efficiency. The more complex algorithms will generally compress better, but at the expense of slower execution speed. The more complicated algorithms use more complex data structures, and the reduction in speed is generally due to searches and maintenance of these structures. Supplying a piece of hardware to simplify the search and maintenance functions allows a total hardware solution of relatively low complexity. A hardware solution will, of course, dramatically increase the system's throughput. A few of the more popular lossless compression techniques will be briefly described.

HUFFMAN CODING

Huffman coding is probably the best-known data compression technique, see Reference [1]. The idea behind Huffman coding is simply that in a given data set, certain symbols are used more frequently than others. Huffman coding exploits this fact by assigning variable length codes to each symbol in the data set. The more frequently encountered symbols are given the shortest codes. Static Huffman coding requires that a table of probabilities exists before compression begins. When the data in question follows a known statistical pattern (such as English text), an existing table may be employed. For other types of information, a histogram of the data set may need to be generated in order to create an encoding table that will produce the highest compression ratio. The symbols of interest for Huffman coding are generally stored in a binary-tree structure. The branches taken in traversing the tree from leaf to root will give the Huffman code for that symbol (in reverse order). Using the Huffman code to traverse the tree from root to leaf will produce the encoded symbol. Once this structure has been constructed, the data (symbols and associated codes) can be transferred to a linear table.

In situations where a data pre-scan is not possible, a dynamic version of Huffman coding can be employed, see Reference [2]. In the dynamic version, the symbol probabilities are continuously adapted as information is processed. The algorithm basically creates a binary-tree structure in which the symbols that occur most frequently reside closest to the root of the tree (the distance from the root to the symbol indicates the length of the Huffman code). As each symbol is processed, its location in the tree is modified based on its frequency of occurrence. The symbol may actually move closer to the root of the tree if its probability becomes higher than a symbol that has a shorter code. This constant tree maintenance slows the dynamic compression process relative to its static counterpart. In addition, the dynamic method does not compress as tightly as the static method. This is due to the fact that an extra code must be sent whenever a symbol appears that is not yet in the table. This extra code is necessary in order to inform the expander of an incoming untabulated symbol (it also removes the necessity of sending an encoding table with the data). However, because most of the compression time is spent with symbols that reside in the table, the extra codes are of very little consequence.

BSTW CODING

BSTW coding, see Reference [3], is similar to Huffman coding in that it also is a word adaptive scheme. The BSTW coding table, unlike the Huffman binary-tree structure, is a simple self-organizing list. The table is basically a list of codes that uses an LRU (Least Recently Used) replacement algorithm. Simply put, this means that the most recently used symbol is at the top of the list, and the least recently used symbol is at the bottom. Variations of this replacement policy, such as LFU (Least Frequently Used - most often used symbol at the top, least often used at the bottom) are also possible. When new symbols are encountered that are not in the list, the bottom symbol is removed, the list is reorganized, and the new symbol is inserted in the appropriate location. This process requires that the table be updated each time a new symbol is added so that the list remains in proper order. The number of symbols per code and the length of the code are both variable. Finding the optimal combination is usually a matter of experimentation with the data sets of interest.

LZW COMPRESSION

LZW compression is probably the most common non-Huffman form of data compression. LZW compression is the Lempel-Ziv algorithm modified by Terry Welch, see Reference [4]. LZW is a compression technique that is based on repeating patterns in a data file. Since it relies on patterns, it is fairly weak on data that is random in nature. The algorithm basically parses strings of symbols into substrings of variable length. These substrings are then mapped to unique codes in a table. As the procedure progresses, the longest input string that matches a substring in the table is parsed off, with the next symbol appended. Code words of larger and larger strings are built up in this way. When the code table has been filled, most implementations of the algorithm clear the table, then begin building a new one. More complex implementations selectively delete table entries (those that are older and are not used by other entries). As with Dynamic Huffman and BSTW, the LZW algorithm does not require the string table to be transmitted with the compressed data. The nature of the algorithm allows the expander to build its own string table based on the compressed data.

ALGORITHM PERFORMANCE

The compression efficiency of each of these algorithms is presented in Table 1 (the size entries represent the compressed size as a percentage of the original size). The table entries represent an average taken over several different files. Certain files (with a high degree of local redundancy) will compress more dramatically, while others may actually expand. The information in parentheses after

Algorithm	Hardware		Compressed Size		
	Throughput	Complexity	Text	Image	Binary
BSTW (1-4)	5.1 MB/s	Low	80%	72%	102%
BSTW (2-9)	11.0 MB/s	Low	64%	80%	104%
BSTW (3-10)	12.4 MB/s	Low	63%	81%	106%
Huffman (Static)	1.3 MB/s	Low	60%	75%	84%
Huffman (Dynamic)	0.8 MB/s	High	61%	76%	84%
LZW (10-bit)	9.5 MB/s	Medium	39%	60%	80%
LZW (12-bit)	9.5 MB/s	Medium	33%	55%	78%

Table 1: Compression Algorithm Comparison (Compressed Size Equals Percentage of Original)

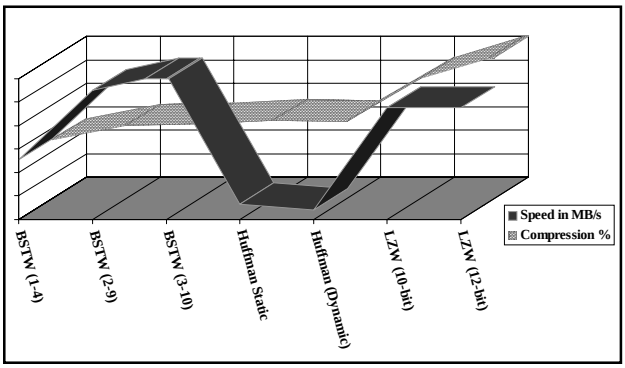

Figure 1: Compression Algorithm Comparison

the algorithm name represents variations in the algorithm's implementation. These variations will be explained in the sections detailing each algorithm.

In addition to the estimated amount of compression, Table 1 illustrates the estimated throughput for a hardware implementation. This data is derived from CAM-based hardware implementations as outlined in this document. Results from other implementations may vary from those presented here. A relative comparison of the algorithms is depicted graphically in Figure 1.

COMPRESSION AND THE CAM

The brief description of the operation of various compression algorithms helps to demonstrate the utility of a CAM in data compression. The algorithms all produce a sparsely-populated table that is then indexed using either binary-tree techniques or hashing algorithms. The CAM indexes sparsely-populated tables at very high speed and requires very little overhead to maintain. This makes the CAM a logical replacement for RAM in data compression applications. In order to illustrate how the CAM can help speed the compression/expansion process, it is necessary to explore specific compression techniques in greater detail.

HUFFMAN CODING USING THE CAM

As was discussed earlier, Huffman coding can be implemented in either a static or dynamic fashion. The static method has the disadvantage of requiring the code table to exist before compression. If a suitable table is not available for the current data set, then the data must be read twice: once to establish the probability table then again for the actual compression. The static method must also provide a copy of the code table to the expander. The dynamic method circumvents the two-pass problem by creating the probability table on-the-fly. The cost is slower execution (due to table maintenance) along with slightly less compression efficiency (longer codes must be sent for never-before-seen symbols).

While implementations using more than a single byte are possible, coding tables can grow to an unmanageable size (64 KB or more) or require complex replacement policies and code maintenance. For this reason, this study will focus on implementations that operate on 8-bit symbols.

Static Huffman (Code Conversion)

Implementation of the encoding portion of static Huffman coding is basically a simple table lookup. The code for each entry resides in an array that is simply indexed by the incoming symbol. The decoding process, on the other hand, offers a little more challenge to the designer. Since Huffman codes are of variable length, simple table lookups do not work very well. Huffman decoders are usually built as binary trees in which each incoming bit is used to select the appropriate branch. When a terminating leaf is encountered (no more branches), the proper symbol has been accessed.

Software implementations of this process do not execute very quickly. In addition, hardware implementations of binary tree searches can be very complex. The perfect replacement for the binary search engine is the CAM. The CAM has the ability to search its entire list of available codes in a single transaction, so a binary tree architecture becomes unnecessary. To convert Huffman codes, the input stream is built up one bit at a time by shifting each bit into an input register (refer to Figure 2). A CAM lookup is performed after each bit is presented. If the code is not found in the CAM, then another bit is shifted in. If the code is found, then the appropriate symbol is output and the input register is flushed. Using this approach, one standard MUSIC MU9C1480A CAM can handle codes as long as 48-bits.

The throughput of such a system is dictated by the speed of decoding (remember that encoding is very fast - just a table lookup). The main-loop cycle time for such a system will be the time required to shift in a bit, perform a CAM lookup, and perform a CAM match flag test. Using a 66 MHz controller and a 90 ns CAM, the cycle time is 165 ns (11 CLKs). This provides a throughput of 6.1 Mbits/s (Huffman code bits). To convert this number to bytes/s requires an estimate of the amount of compression

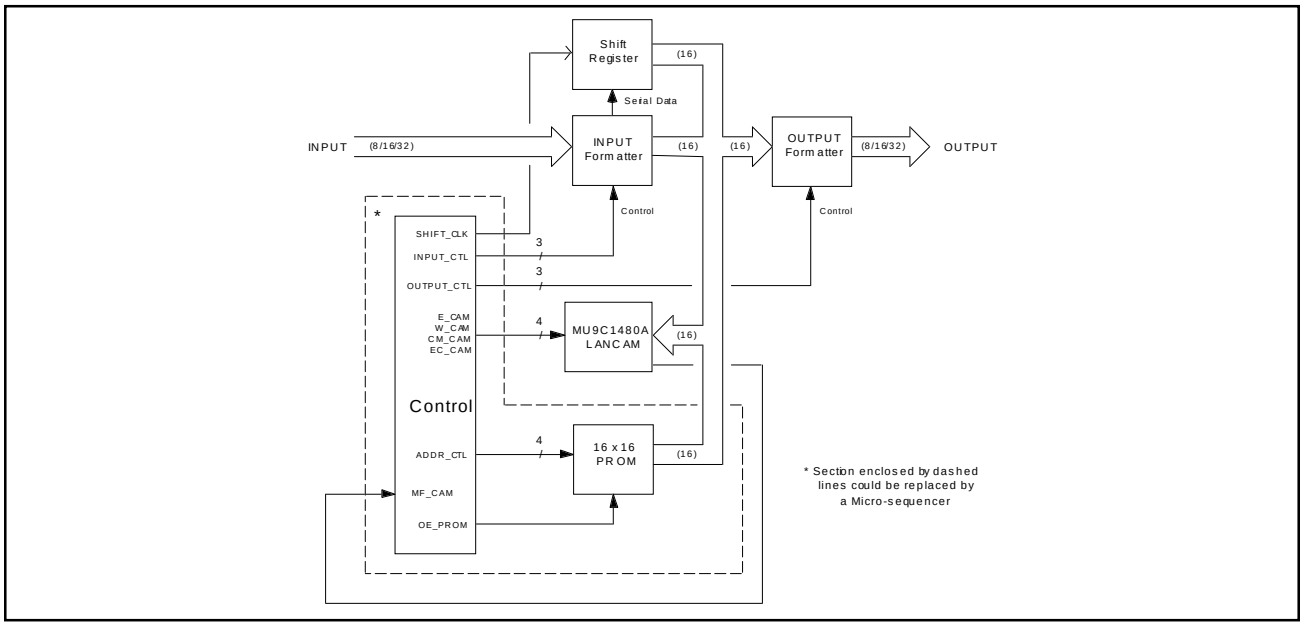

Figure 2: Static Huffman Code Converter

achieved. If a compression rate of 60% is assumed (about right for most Huffman-coded text), then the estimated throughput is calculated as follows:

((6.1 Mbits/s)/0.60)/(8 bits/byte) = 1.3 MBytes/s

(The actual throughput will generally range from 1–2 MBytes/s, depending on the source material.) Higher rates could be achieved using a faster speed or a CAM with a wider I/O, such as the WidePort CAM family.

Dynamic Huffman Coding

Dynamic Huffman coding presents a more complex problem than for the static case. In order to dynamically assign codes based on symbol frequency, the Dynamic Huffman Coding algorithm must continually perform some form of maintenance on its tables. This maintenance involves various search and replace functions as well as table data swaps. The complexity of the algorithm along with the necessity of major table maintenance slows execution speed considerably. This, coupled with the fact that the level of compression is only moderate, indicates that this algorithm is a relatively poor candidate for hardware implementation. The implementation details will not be discussed here. For further information regarding the algorithm itself, see Reference [2].

MODIFIED BSTW CODING USING THE CAM

For greater simplicity, the coding strategy chosen for BSTW is fixed length. This means that, for single byte symbols, BSTW coding becomes a simple table lookup (just as with static Huffman). For multiple byte symbols, the table becomes sparsely populated and requires some form of hashing algorithm to access in a reasonably sized table. The CAM provides the hardware equivalent of the required hashing algorithm. Unfortunately, the CAM is not well suited to reasonable implementation of either the LRU or the LFU replacement policies. To greatly simplify the hardware (and increase the throughput), a simple round-robin replacement policy is implemented. A comparison of the policies is displayed in Table 2.

Replacement Policy	Compressed Size		
	Text	Image	Binary
Round-Robin	64%	80%	104%
LRU	62%	77%	101%
LFU	62%	77%	102%

Table 2: Replacement Policy Comparison for BSTW (2-9)
(Compressed Size Equals Percentage of Original)

As can be seen, the benefit gained by using a non-trivial replacement policy is relatively small. Considering the amount of additional hardware required and the time penalty incurred by the more complex policies, the simple approach is superior.

A control flow diagram for a hardware implementation of the described version of the BSTW compression algorithm can be found in Figure 3. The basic compression flow is as follows: After initialization, a symbol is read in. An EOF symbol terminates the process by outputting the NIT (Not In Table) code followed by the EOF character. For any other symbol, a CAM lookup is performed with the symbol as the comparand. If the symbol is found, the code for that symbol (the CAM match address) is sent to the output section. If the symbol is not found, then the symbol is entered into the CAM at the address pointed to by the internal Address register. With proper CAM initialization, this register will now increment to the next address. The value of this register must now be tested for table limits. The current address must be read and checked against the limit. If the limit has been reached, then the Address register must be reset to a value of 1 (1 is the first available code; 0 has been reserved as the NIT code). Now that the table has been updated, the symbol must also be sent to the output. The NIT code is sent first, followed by the actual symbol. This sequence allows the expander to identify symbols that do not yet appear in its table. It also allows the expander to build a table that mirrors the table built by the compressor. Symbols are processed in this fashion until EOF is encountered.

The expansion flow (Figure 4) is very similar to the compression flow. After performing initialization, a code is read. If the code is NIT (Not In Table), then another symbol is read. If the symbol is not EOF, then it is written to the CAM at the address pointed to by the Address register. This address will increment after the write. The address register must then be checked against the table bounds. The current value of the address pointer may either be read from the CAM, or it may be kept locally in a small counter within the controller. If the boundary has been exceeded (or if the counter has wrapped back to 0), then the register must be reset to a value of 1 (remember that 0 is reserved for NIT). Now that this table has been updated the same as the compression table, the symbol should be sent to the output.

If the code is not NIT, then a CAM lookup is performed with the code as the comparand. This is a simple table lookup, so the code must be found or else a fatal error has occurred. When the code is found, the associated symbol is sent to the output. Codes are processed in this fashion until the EOF symbol is encountered.

Now that a working algorithm has been outlined in detail, it is necessary to assess suitable encoding parameters for the design. A number of parameters will be of importance in determining the algorithm's efficiency:

Coding method
It has already been determined that fixed length coding is the simplest to implement. Using a 1K CAM, the maximum code size is 10-bits ($Log_2(1024)$). A deeper CAM will handle larger codes.

Table length
Obviously, the greater the table length, the greater the chances of finding an encoding for any given symbol. However, a longer table necessitates longer codes, thereby reducing compression.

Symbol size (bytes)
The more bytes combined to form a symbol, the better compression achieved when matched. However, fewer matches will occur for longer symbols.

In order to determine the optimum combination of symbol size versus code size, a number of data sets were compressed using various combinations of these parameters (Figure 5). The data sets included text, image, and binary information. The results are expressed as a compression ratio (taller bars indicate greater compression).

Notice that some of the combinations actually expand the file rather than compress it (those bars below the 1:1 level). This can occur for situations where the table is too large (long codes), the table is too small (low hit ratio), or the data shows very little local redundancy (as with binary data). The experimental results indicate that the optimum configuration for text data is 2 bytes encoded as 9 bits (2–9) or 3 bytes encode as 10 bits (3–10). Image data appears to compress better in a smaller table, such as 1 byte encoded as 4 bits (1–4). The binary files either compress or expand very minimally (the average is marginal expansion). It seems that the binary data is random to the point that none of the various scenarios does a very good job. Based on the collected data, parameter set ups for (2–9) and for (3–10) both seem to perform adequately. System throughput will be greater, however, for the system

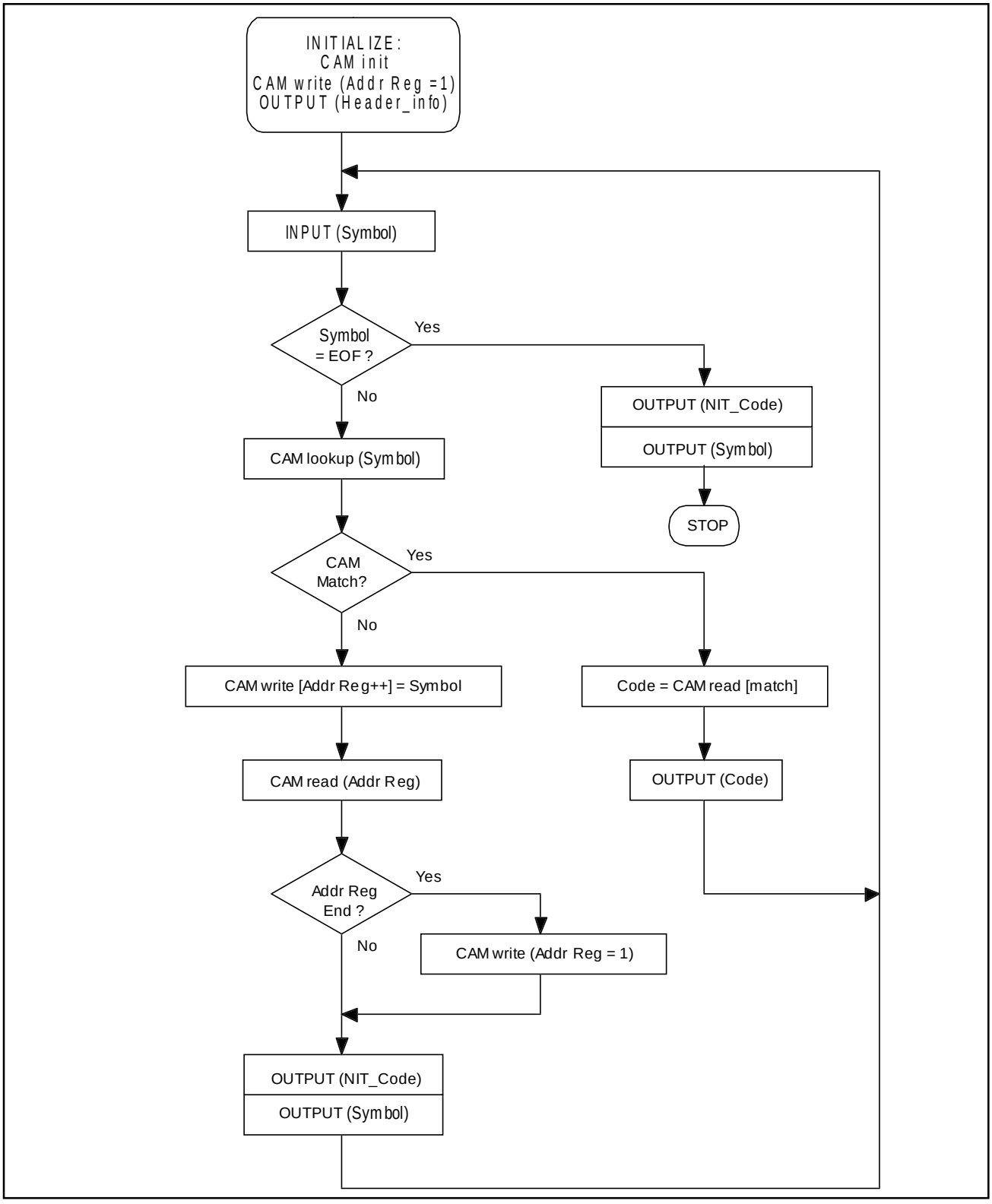

Figure 3: Modified BSTW Compression Control Flow

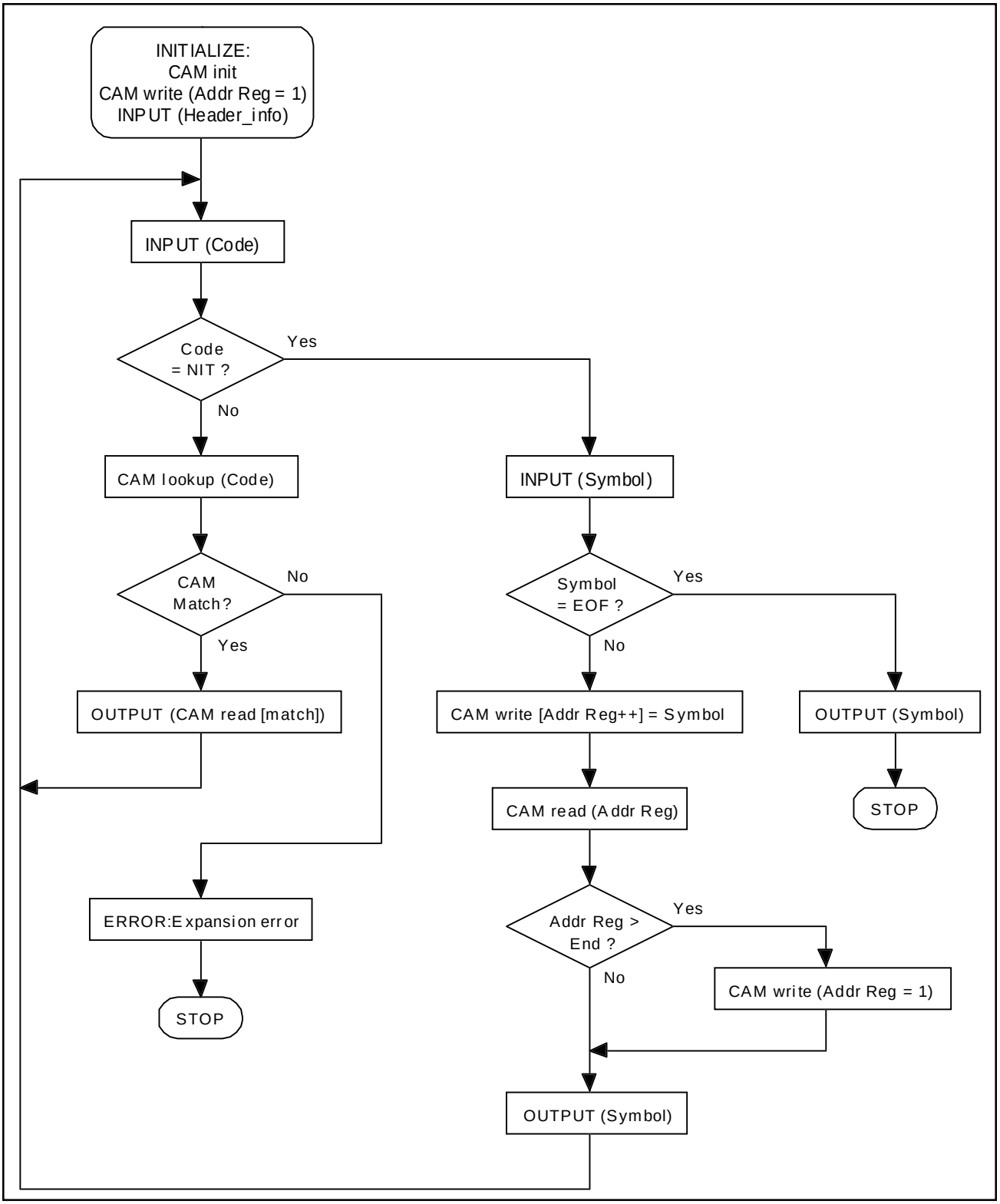

Figure 4: Modified BSTW Expansion Control Flow

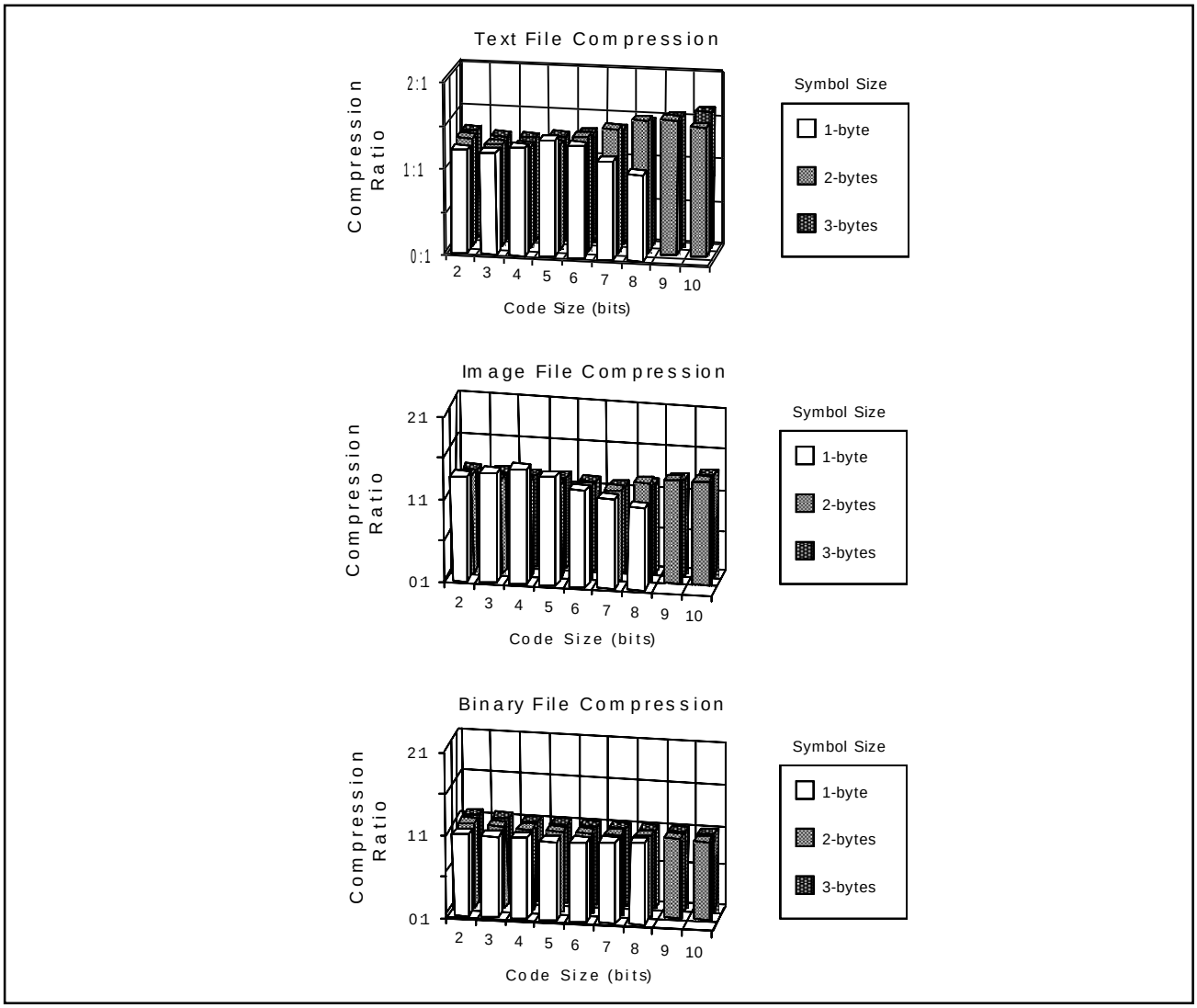

Figure 5: BSTW Parameter Effects on Compression

that encodes more bytes per symbol. Keep in mind that the described hardware solution can be made to support any parameter combination up to 6-byte symbols and 10-bit codes. Using a larger CAM allows code lengths up to 13 bits. If the hardware is designed properly, these parameters could be set differently for each data set of interest. (Note: lowest possible compressed size using 2-byte symbols with 9-bit codes is 56%; 3-byte symbols with 10-bit codes is 42%).

The hardware required to implement this flow is really quite simple (Figure 6). Beyond the CAM itself, all that is required is a simple controller, a PROM (alternatively a micro-sequencer), and input and output formatters. The input and output formatters adjust the data format to the system interface. For example, 9-bit codes would require some type of packing in the output formatter in order to accommodate an 8-bit, 16-bit, or 32-bit bus. Likewise, the input formatter should be able to disassemble those same 9-bit codes for expansion. Proper control of these formatters would also be required if variable parameter setups is desired. Control of the entire system could be achieved through a relatively simple state machine with Op-Codes for CAM control being retrieved from a small PROM.

The system's throughput can be estimated based on the control flow diagrams (Figure 3 and Figure 4) and statistical information regarding the branches taken. The flow has two

main branches: one for symbol found, and one for symbol not found. Using a 66 MHz controller (7 ns PLD), a -90 MU9C1480A CAM, and pipelining as many functions as possible, the following cycle estimates can be made:

Not Found	# Clocks	Found	# Clocks
CAM look-up	6,6,9	CAM lookup	6,6,9
EOF check	5,5,5	Match?	5,5,5
CAM write	6,6,6		
Reg bounds	1,1,1		
	18,18,21		11,11,14

(Given clock values are for (1–4), (2–9), and (3–10), respectively.)

Based on statistical information, the time spent in the Found branch for (1–4) codes is about 65-75% of the total, for (2–9) codes about 80-90% of the time, and for (3–10) codes about 60-80% of the time. An estimated throughput based on this information would be:

(1–4): (.70 * 11) + (.30 * 18) = 13.1 clocks = 197 ns (@ 66 MHz)
(2–9): (.85 * 11) + (.15 * 18) = 12.1 clocks = 181 ns (@ 66 MHz)
(3–10): (.70 * 14) + (.30 * 21) = 16.1 clocks= 242 ns (@ 66 MHz)

Estimated compression rate:

(1–4) -> 1-byte / 197-ns = 5.1 MBytes/sec
(2–9) -> 2bytes/181-ns = 11.0 MBytes/sec
(3–10) -> 3bytes/242-ns = 12.4 MBytes/sec

Using the same analysis as for compression throughput, expansion throughput may be estimated:

Not-In Table	# Clocks	In-Table	# Clocks
Symbol read	1,1,1	CAM lookup	6,6,9
EOF check	1,1,1	Match?	5,5,5
CAM write	6,9,12		
CAM read	6,6,6		
	14,17,20		11,11,14

(Given clock values are for (1–4), (2–9), and (3–10), respectively.)

Based on statistical information, the time spent in the In-Table branch for (1–4) codes is about 65-75% of the total, for (2–9) codes about 80-90% of the time, and for (3–10) codes about 60-80% of the time. An estimated throughput based on this information would be:

(1–4): (.70 * 11) + (.30 * 14) = 11.9 clocks = 179 ns (@ 66 MHz)
(2–9): (.85 * 11) + (.15 * 17) = 11.9 clocks = 179 ns (@ 66 MHz)
(3–10): (.70 * 14) + (.30 * 20) = 15.8 clocks= 237 ns (@ 66 MHz)

Estimated expansion rate:

(1–4) -> 1-byte / 179-ns = 5.6 MBytes/sec
(2–9) -> 2bytes/179-ns = 11.2 MBytes/sec
(3–10) -> 3bytes/237-ns = 12.7 MBytes/sec

Since the slower rate governs the system throughput, a rate of 5.1 MB/s is appropriate for the (1–4) case, 11.0 MB/s for the (2–9) case, and 12.4 MB/s for the (3–10) case.

LZW COMPRESSION USING THE CAM

Before examining the block diagram and hardware control flow of the CAM-based LZW routine, the basic algorithm will be described in greater detail.

The assumption will be made that the source material consists of 8 bits ASCII codes. The algorithm starts with a table containing the 256 ASCII literal codes. Additional entries in the table will store combinations of other codes (known as strings). Beginning with 9-bit codes, the table length is 512 entries (providing an additional 256 entries). String comparisons may now take place. The base string is created by reading and appending the first two symbols. If the base string is in the table, create a new string by reading in the next symbol and appending it to the base string's code from the table. Now perform a lookup with the new string. This process is repeated until a string has been constructed that is not in the table. This unknown string is added to the table and the code for the previous known string is sent to the output. The last symbol now forms a new base string. This process is illustrated in Table 3. As compression proceeds, each new table entry consists of either a pair of literals or a code-literal combination. This is how very long strings are built up with only a dual-entry index. This entire process is repeated until EOF is reached or the table fills. When the table is filled, two options are

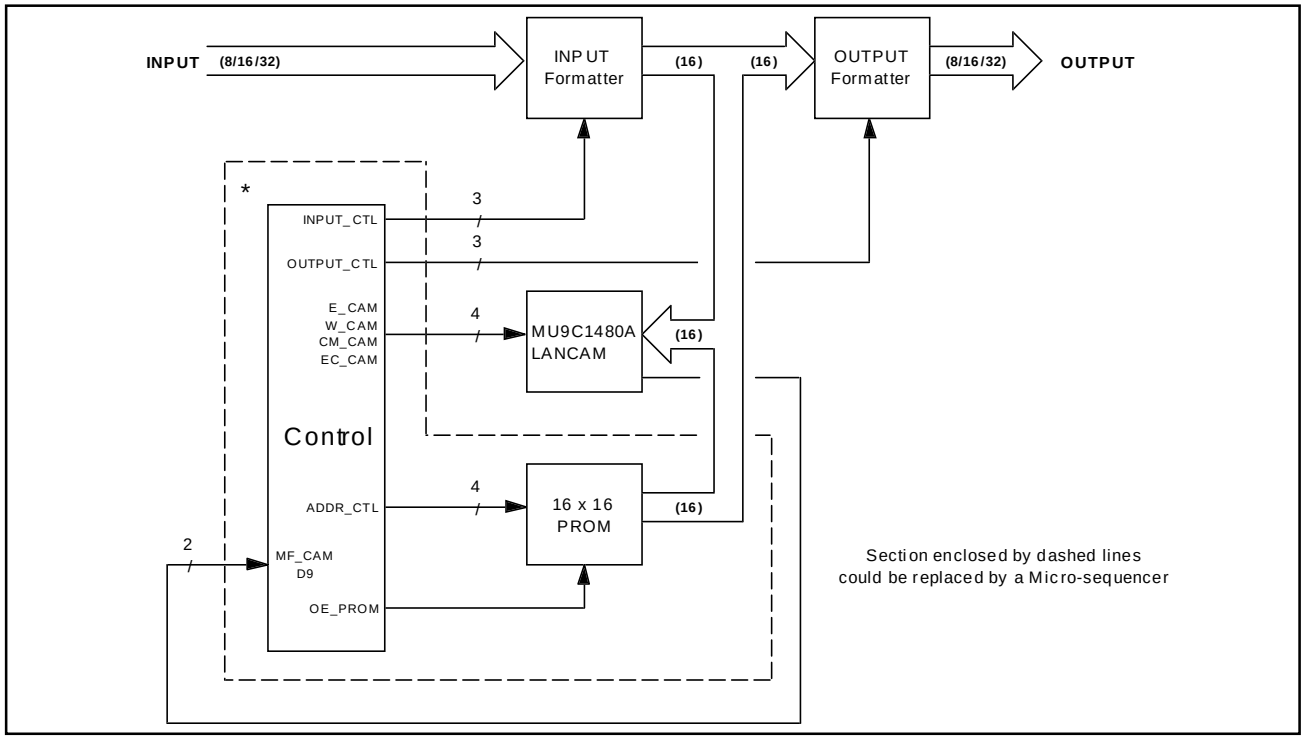

Figure 6: Modified BSTW Compressor/Expander Block Diagram

available: clear the table and begin a new one, or extend the table length. Extending the table length is an automatic implementation of LZW variable length coding. When the table is extended, the number of encoding bits grows accordingly. For example, when the 9-bit table is filled, 10-bit codes may be employed thereby extending the table length by 512 entries (total of 1024). The reason that longer codes are not utilized from the start is that greater compression can be achieved if the shortest possible code is used for as long as possible. When no more table space is available, the table must be modified in some fashion. The simplest modification is to clear the table and to send a Clear code with the data so that the expander is aware of the table change. More complex methods in which entries are selectively deleted have been explored, but the complexity of implementation seems to outweigh any benefit of additional compression.

The LZW expansion process begins with a table of literal ASCII codes, just as with the compression process (LZW expanders do not require any previous information regarding string tables; a local string table will be built from the compressed data). The first code from the data stream is read and sent to the output. This code is the base string for future reference. The next code is now read and checked for special codes (EOF or Clear code). If not a special code, the code is checked to see if it is a literal. A literal code would have a magnitude less than 256. If the code is literal it is sent to the output section. The code is then appended to the base string and it is written to the table at the next free code location. This current code now becomes the new base string. If the code had not been literal, then a table lookup using the code as index would have produced a new string. The literal portion of this new string would be placed on an output stack and the base string portion examined. This process would be repeated until the base string portion is a literal. In this manner, an output stack containing all of the symbols that built up the code can be produced. Each symbol would now be popped from the stack and sent to the output section. The final literal would be appended to the original base string and written to the table. An example of the expansion process can be seen in Table 4.

As long as the table limits have not been exceeded, this process continues until a special symbol is encountered. A Clear code will essentially restart the algorithm. An EOF will terminate the process. If the next free code location is beyond the current code size, then the table must be lengthened and the code size incremented. A Clear code should be received before the table limit is reached.

As presented, the expansion routine is not completely bullet proof. There is a certain type of string that can cause the compressor to send a code that has not yet been constructed in the expander table. The situation can be illustrated with the example string ABABA. If the string AB already exists in the table, then the compressor will send the code for AB, then add ABA to its table. The compressor will then process the next known string, ABA (beginning with the middle A), and send its code. If the expansion flow is interpreted, it is found that this code is not yet in the expander table.

Fortunately, this case is handled by placing a simple test in the expander routine. If the expander receives an unknown code (if the code value is greater than the next free code), then it has encountered this special case. The expander should at this point output the last translated code (AB), and then repeat the first letter from that code (A). This code string (ABA) is also added to the next free location in the expander's string table. The compressor and expander string tables are now back in sync and the symbols sent to the output are correct.

A control flow diagram for the hardware implementation of an LZW compressor is given in Figure 7. The flow is essentially the same as has been described in the preceding paragraphs with a few more details. After initialization, the Clear code and the EOF code are calculated. These codes should be the first two available (for example, 256 and 257). Header information may include initial code size and maximum table size. The first code output in this implementation is the Clear code. This serves to insure that both tables are cleared and that the expander is synchronized. At this point the flow is as described previously. The code table is, of course, a CAM and any table lookups will be completed in a single transaction.

INPUT	TABLE ENTRY	OUTPUT
M	----------	-
i	[258] = M +i	M
s	[259] = i + s	i
s	[260] = s + s	s
i	[261] = s + i	s
s	----------	-
s	[262] = 259 + s	259
i	----------	-
p	[263] = 261 + p	261
p	[264] = p + p	p
i	[265] = p + i	p
	[266] = i + _	i

Table 3: LZW Compression Example

INPUT	TABLE ENTRY	OUTPUT
M	----------	M
i	[258] = M +i	i
s	[259] = i + s	s
s	[260] = s + s	s
259	[261] = s + i	is
261	[262] = 259 + s	si
p	[263] = 261 + p	p
p	[264] = p + p	p
i	[265] = p + i	i

Table 4: LZW Expansion Example

Similarly, a control flow diagram for the hardware implemented LZW expander is provided in Figure 8 on page 14. The expander will first read any header information that the specific implementation provides. A code will then immediately be read. According to the rules for this system, this first code must be a Clear code. This will insure a clean table and proper synchronization with the compressed data. The flow proceeds as outlined in the above description. The expansion process creates a contiguous table that may be efficiently stored and accessed in a RAM, so a CAM is not required for this process.

Following are descriptions of variable names referenced to in Figure 7:

Variable	Description
Clear_code	Clear table command (CAM initialization)
EOF_code	Signifies end of data stream (literally End Of File)
match	CAM lookup match location
Code	Current symbol
Next_code	Next available code (CAM - next free address)
Lastcode	Base string for Compressor
Oldcode	Base string for Expander
Origcode	Code holder during Expander stacking procedure
Lastout	Last literal output holder for LZW expansion anomaly
Outstack[ ]	Output stack array for Expander
Code_size	Current code size in bits
Max_code	Current maximum code ($_2$(Code_size))
Overflow	Maximum table length
Max_size	Maximum code size for table (Log_2(Overflow))

A block diagram for the LZW compression hardware can be found in Figure 9. As shown, the actual hardware implementation for such a complex algorithm is fairly simple. This is due to the fact that the CAM provides a complex lookup function that would normally require a good deal more hardware. The bulk of the hardware is basically found in the control section, which consists of a moderately complex state machine controller. This section provides all of the flow control functions as well as CAM Op-Codes.

The only hardware required apart from the control section and CAM is a latch and a multiplexor. The input and output formatters are required to pack (or unpack) the data into widths that match the external I/O bus. All of the functions specified in the control flow diagrams (Figure 7) can be implemented using this handful of components.

Use of a single MUSIC MU9C4320L ATMCAM will support a table length of 4096 (12-bit codes). Longer tables are possible simply by cascading CAMs to increase the table length (2 CAMs = 13 bits, 4 CAMs = 14 bits). MUSIC ATMCAMs have been designed to cascade with no additional external logic and with no additional lookup penalty.

A single ATMCAM could also be used to support a number of smaller tables. This feature could be used to support multiple data streams going over a single network link. For example, the table could be divided into four 1K segments to support four 1K tables with only a small reduction in throughput.

An estimate of the throughput of the described system can be made based on the control flow diagram (Figure 7), the block diagram (Figure 9), and statistical information regarding the branches taken. The flow has two main branches: one for symbol found, and one for symbol not found. All non-CAM related functions (input and output formats, etc.) would be pipelined. Using a 66 MHz controller (7 ns PLD) and a -70 MU9C4320L ATMCAM, the following cycle estimates can be made:

Not Found	# Clocks	Found	# Clocks
CAM look-up	6	CAM look-up	6
CAM write	4		
	10		6

Based on statistical information, the time spent in the Found branch is about 65-85% of the total. An estimated throughput based on this information would be:

(.75 * 6) + (.25 * 10) = 70 clocks =
105 ns (@ 66 MHz)

Estimated compression rate:
1-byte/105-ns = 9.5 MBytes/sec

Implementation of common data compression/expansion algorithms in hardware has been shown to be simplified through the use of MUSIC Semiconductors CAM technology. Beyond those examples presented here, any compression scheme that utilizes sparsely populated tables can take advantage of the extremely high-speed search capabilities of the LANCAM or ATMCAM.

References

[1] Gallager: “Variations on a theme by Huffman.” IEEE Transactions, 1978, IT-24, (6), pp. 668-674.

[2] Knuth: “Dynamic Huffman Coding.” Journal of Algorithms,1985, 6, (2), pp.163-180.

[3] Bentley, Sleator, Tarjan, Wei: “A Locally Adaptive Compression Scheme.” CACM, 1986, 29, (4), pp.320-330.

[4] Welch: “A Technique for High-Performance Data Compression.” IEEE Computer, 1984, 17, pp. 8-19.

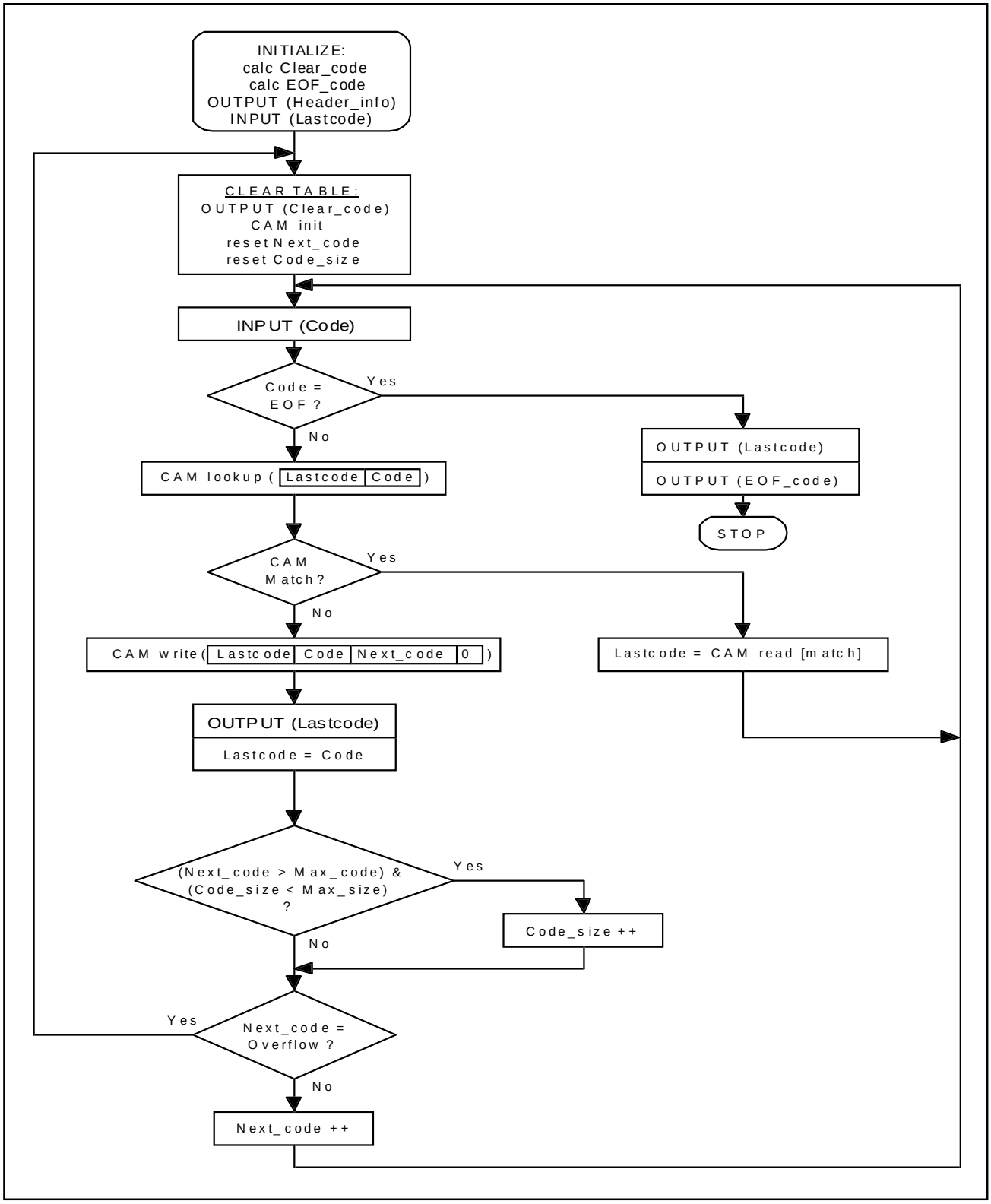

Figure 7: LZW Compression Flow Control

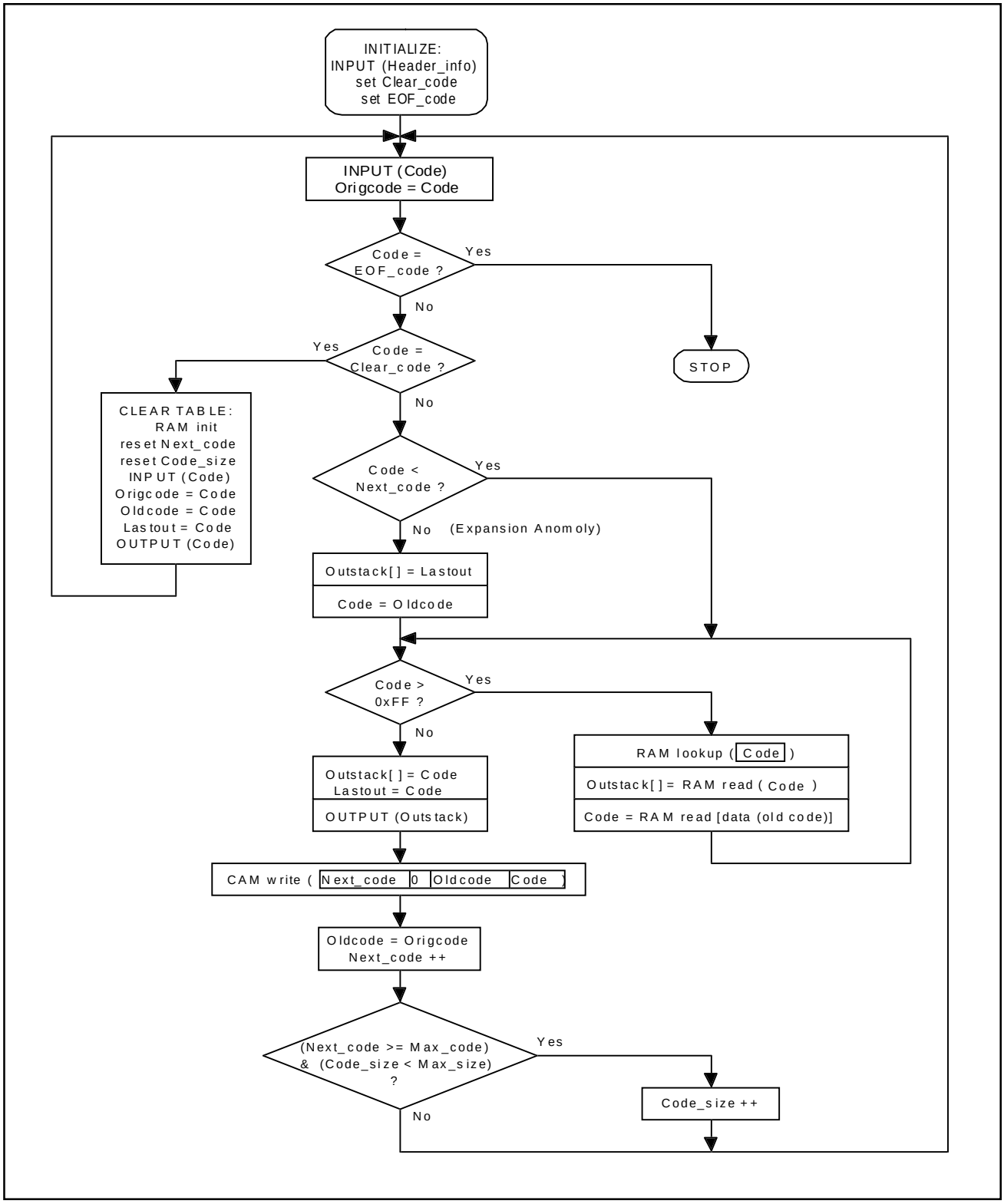

Figure 8: LZW Expansion Control Flow

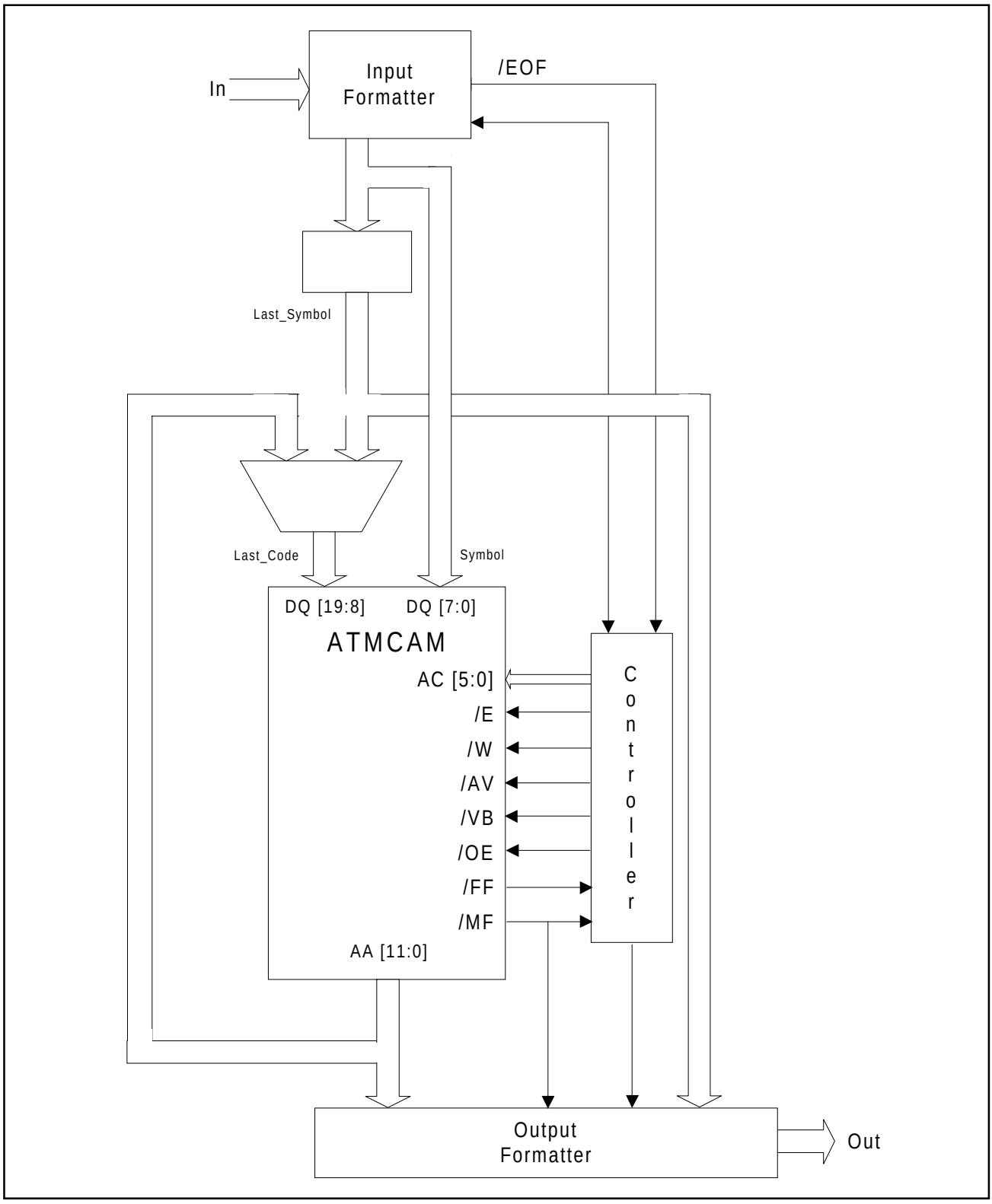

Figure 9: LZW Compression/ Expansion Block Diagram

NOTES

MUSIC Semiconductors Agent or Distributor:

MUSIC SEMICONDUCTORS

http://www.music-ic.com
email: info@music-ic.com

USA Headquarters
MUSIC Semiconductors
254 B Mountain Avenue
Hackettstown, New Jersey 07840
USA
Tel: 908/979-1010
Fax: 908/979-1035
USA Only: 800/933-1550 Tech. Support
888/226-6874 Product Info.

Asian Headquarters
MUSIC Semiconductors
Special Export Processing Zone 1
Carmelray Industrial Park
Canlubang, Calamba, Laguna
Philippines
Tel: +63 49 549 1480
Fax: +63 49 549 1023
Sales Tel/Fax: +632 723 62 15

European Headquarters
MUSIC Semiconductors
Torenstraat 28
6471 JX Eygelshoven
Netherlands
Tel: +31 45 5462177
Fax: +31 45 5463663