論文 95-7-7-23

렘펠-지브 압축 알고리즘을 구현한 효율적인 VLSI 구조 연구

正會員 南 承 鉉*, 李 泰 永*, 李 文 基*, 李 溶 錫*

An Efficient VLSI Architecture Implementing the Lempel-Ziv Compression Algorithm

Seung Hyun Nam*, Tae Young Lee*, Moon Key Lee*, Yong Surk Lee* Regular Members

ABSTRACT

This paper proposes a novel VLSI architecture capable of processing the Lempel-Ziv-based data compression algorithm very fast. The architecture is composed of five main blocks, i.e., PE-based Match Block, Consecutive Hit Checker, Pointer Generator, Length Generator, and Code Packer. The first part, *Match Block*, generates match hit signals which inform the Consecutive *Hit Checker*(CHC) whether there are matches between the symbols of dictionary buffer and one symbol to be compressed. The dictionary buffer is included in the Match Block. The second part, *Consecutive Hit Checker*, checks whether there are any consecutive hits to detect the longest match substring. The Pointer Generator is a priority encoder which generates the pointer(address) of the longest match substring and the *Length Generator* is a counter which generates its length. Finally the *Code Packer* makes a bit stream using the combination of the pointer and the length of the match substring or a uncoded symbol. To reduce hardware costs and to improve compression ratio, the dictionary buffer of 1024 bytes and the maximum allowable match length 16 bytes are used in this architecture.

The proposed architecture has the following features. (1) The modularity of PE-based structure makes it possible to adapt to various buffer sizes without any loss of speed and control overhead. (2) Designed in systolic architecture, the architecture has a simple control logic. (3) It processes exactly one character per clock cycle, so it does not need any accumulated shift operations for preparing the dictionary buffer, which are common problems found in most other architectures. (4) Implemented in 0.6 μm CMOS technology, it can operate up to 100 MHz according to the netlist simulation of the critical path, the pointer generator. (5) It can be used in real-time data compression for video coding or text compression over communication channels to reduce communication costs and time with a throughput rate of 100 Mega samples(characters) per second.

*연세대학교 전자공학과 VLSI & CAD 연구소
VLSI & CAD Lab., Dep. of Electronic Engineering, Yonsei University
論文番號 : 95164-0428
接受日字 : 1995年 4月 28日

要 約

본 논문은 렘펠-지브 데이터 압축 알고리즘을 빠르게 수행할 수 있는 새로운 VLSI 구조를 제안한다. 전체 구조는 처리소자를 기반으로 한 정합 블록(PE-based Match Block), 연속 정합 검사기(Consecutive Hit Checker), 주소 생성기(Pointer Generator), 정합 길이 생성기(Length Generator), 그리고 코드 결합기(Code Packer)의 다섯 가지 주요 블록들로 구성된다. 첫 번째 부분인 정합 블록은 현재 사전 버퍼의 심볼과 압축될 심볼간에 정합의 여부를 결정하여 정합 적중(match hit) 신호를 다음 블록인 연속 정합 검사기에 전달한다. 두 번째 부분인 연속 정합 검사기는 최장의 정합된 문자열을 얻기 위해서 연속적인 정합이 있는 지의 여부를 가린다. 주소 생성기와 정합 길이 생성기는 최장의 정합된 문자열의 위치와 그 길이를 생성한다. 마지막으로 코드 결합기는 전 단계에서 얻어진 정합된 위치와 길이를 조합하거나, 압축되지 않은 심볼로 최종 인코딩된 비트 스트림(bit stream)을 만든다. 하드웨어 비용과 압축 비율의 관점에서, 본 논문에서는 1024 바이트의 사전 버퍼를 사용하고 최장의 정합길이를 16 으로 제한한다.

제안된 구조는 다음과 같은 특징을 가진다. (1)처리소자(Processing Element)를 기반으로 한 구조이기 때문에 다양한 버퍼 크기에 속도의 감소와 컨트롤의 추가 부담이 없이 적용될 수 있다. (2)시스토릭 구조로 설계되어 간단한 콘트롤으로 구현된다. (3)본 구조는 한 사이클에 정확히 한 심볼을 처리하며 여타 다른 구조에서 문제점으로 지적되는 누적된 쉬프트 연산이 전혀 필요 없다. (4)0.6μm CMOS 공정의 임계경로의 네트리스트 시뮬레이션에 의하면 100 MHz까지 작동될 것으로 예상된다. (5)초당 1 억 개의 샘플을 처리함으로써 통신 선로상에서의 통신 비용과 시간을 줄이기 위한 영상 및 텍스트 압축을 실시간으로 수행할 수 있다.

I. INTRODUCTION

Data compression technology is a reduction of the redundancy in data so that data storage requirements and data transfer costs can be reduced. It is a process of encoding the body of data into a shorter form from which the original or some approximation of the original can be restored in a decoder at a later time. The lossy compression can recover the approximation of the original data. In this paper, we focus on lossless data compression, from which the compressed data must be recovered identically to the original data. With the demand for efficient data compression methods, a number of lossless data compression algorithms have been proposed such as Huffman coding[1], run-length coding[2], arithmetic coding[3], LZ algorithm[4,5], LZW algorithm[6], etc. Most of these algorithms have been applied to software implementations. Therefore, they do not satisfy the performance requirements of the future systems. Multimedia systems need real time data compression over the communication channels to reduce communication cost and time. For example, the international standard V42.bis is adopted for data compressing modems using modified LZ-based data compression algorithm implemented in 8-bit microprocessor with a 40-Kbyte RAM and 64-Kbyte ROM[7]. In the future, one single chip must do this task much more faster than 14,400 bps which is the current common transfer rate. In recent years, several special purpose hardware architectures have been proposed[8,9,10,11]. A few designs using CAM(Content Addressable Memory), microcode approach, and microprocessor-based system have been reported. A set of parallel algorithms for compression

using textual substitution are proposed, and a hardware system consisting of several chips implementing their algorithm has been built. However, some idle cycles must be allocated in processing elements during operations, causing the control overhead and the limit of the number of processing elements. For example, 'shift and update' process is needed every 33 cycles in Wei's study[11]. Yang and Lee[9] exploited the CAM approach. However, the compression ratio is very low due to the limit of CAM size and the architecture proposed needs one extra cycle to load new symbols from buffer into the CAM. In Burleson's paper[10] and Henriques's paper[8], many delay elements, long encoding latency, and unnecessary 'shift and update' are incurred. Our approach is a VLSI architecture which implements the modified LZ algorithm. The hardware algorithm is systolic and can be efficiently implemented as a single chip system. They can process up to 100 Mega samples/sec with a clock frequency of 100 MHz simulated through the post-layout circuit simulation of the critical part of the design. The chip can process data compression and decompression in real-time systems. An efficient hardware algorithm implementing the LZ algorithm is described. The rest of this paper is organized as follows. The next section describes the LZ algorithm and a modification for hardware performance improvements in detail. Section Ⅲ and Ⅳ describe the proposed systolic algorithm and its architecture. In section Ⅴ, decompression algorithm and architecture are described. Conclusions are drawn in Section Ⅵ.

Ⅱ. Lempel-Ziv Coding Algorithm

LZ algorithm is a compression method which encodes source symbols into fixed length codewords which represent match pointer C_p, match length C_l, and last symbol C_s[8]. The match pointer C_p indicates where the longest match starts in the dictionary buffer. C_l is the maximum match length, and C_s is the first symbol in the uncompressed source symbols. The LZ algorithm has two main steps. First, we find the pointer and the length of the longest substring in the dictionary buffer that matches the string to be coded. Hereafter, that substring is defined as champion substring. Second, we shift out the old symbols in the dictionary buffer by the amount of the maximum matching length plus one(last symbol length) and shift the symbols coded in the previous step, into the dictionary buffer from the uncoded symbol buffer. This second step is the accumulated shift operations which cause hardware and time overhead in other architectures[,8,9,10,11] and are deleted in this proposed architecture.

The above steps can be realized just by comparison and shift operations but they require one overhead cycle for every encoded codeword. Suppose that the dictionary buffer stores N recently compressed symbols and the uncoded symbol buffer stores M uncompressed symbols. The compression speed and ratio are heavily influenced by size N. If N is infinitely large, the compressed data can be reduced near information entropy with the infinite size of M. But, as N increases, processing time and chip area also increase by the requirements of the comparison operations and the storage elements for buffering the dictionary symbols. The compression ratio is also affected by the maximum allowable matching length M. If M is large, the compression efficiency can be increased when the length of the *champion substring* is suffi-

ciently large. But if the length is less than M, the achieved compression ratio will not be improved. Moreover, the increased maximum matching length requires more processing time and hardware. In practical applications, sizes from 4 to 32 are suitable for M, and from 128 to 4000 bytes for N. With these buffer sizes, compression ratio of 2 to 3 can be achieved[9]. Fig 1 illustrates the LZ compression process.

The compression steps in Fig 1 is summarized as followings.

(1) Initialization : initialize the dictionary buffer with '0' and fill the uncoded symbol buffer with M uncoded symbols.

(2) Parsing : find the longest substring in the dictionary buffer that matches consecutively with the substring starting from the leftmost of the uncoded symbol buffer.

(3) Encoding : encode the match pointer(C_p), the match length(C_l), and the last symbol(C_s) into a fixed length codeword.

(4) Shifting : shift the leftmost (C_l+1) old symbols out of the dictionary buffer and shift (C_l+1)recently compressed symbols into the dictionary buffer from the uncoded symbol

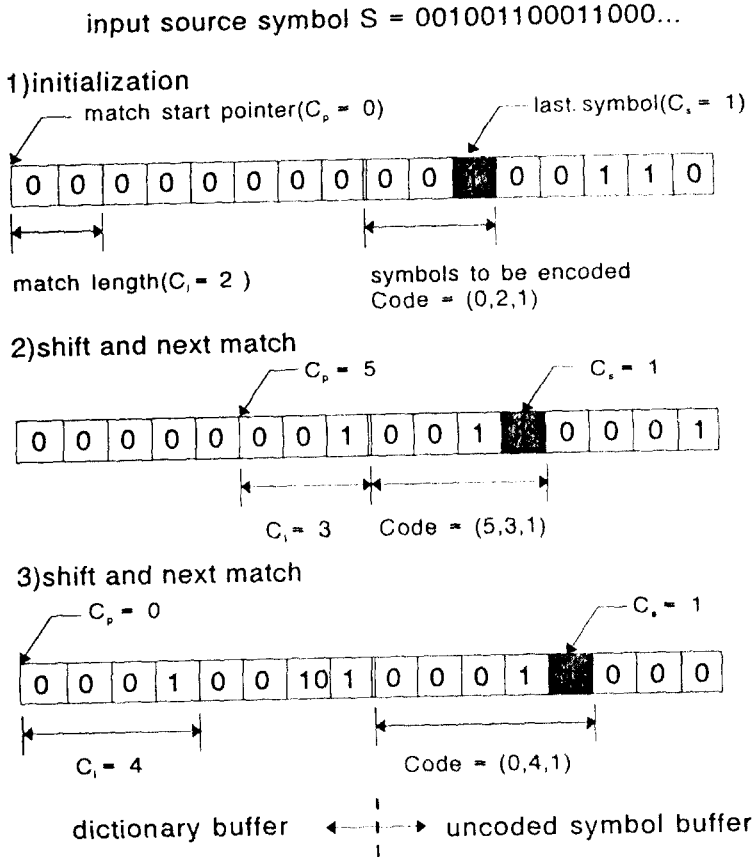

Fig. 1. LZ compression process
그림 1 LZ 압축 과정

buffer.

(5) If the current symbol is EOF, encode the EOF and finish the encoding, or else go to step (2).

If there is no match or the length of the champion substring is less than two or three, then data expansion problem can arise rather than compression. Namely, when there are few duplicated data patterns, the compression efficiency will be very low.

The following method is adopted to avoid the expansion problem in the case of text data[8]. The first byte of the codeword can be the last symbol which is represented in ASCII code. Then the unused bit(most significant bit) can be used as a 'tag bit' which is set to '1' if the next two bytes indicate the pointer and the length. When the match length is less than 3, the tag bit is set to '0' to represent that the following byte is the uncompressed text data, i.e., the ASCII code itself(hereafter defined as a 'literal symbol'). During decompressing codewords, the decompressor checks this tag bit to find out how it should decode the codewords. When the tag bit is '1', the decompressor uses the pointer and the length in the codeword to decode the codeword. Otherwise the decompressor outputs the next byte in the codeword as a decoded bit stream. Thus by using this reserved tag bit, the expansion problem can be avoided and compression ratio also can be further improved. In the above method described, every codeword must include one byte of uncoded literal symbol with one bit tag. Thus, symbols that may be compressed with other symbols together must be coded as a literal symbol in every codeword and the compression efficiency is reduced. In our proposed architecture, we utilize only one bit tag instead of the entire last symbol of one byte. We choose 1024 bytes(1024 symbols) for the dictionary buffer and 16(4 bits) for the maximum matching length. Therefore, 10 bits and 4 bits are required to encode the match pointer and the match length each. With our dictionary buffer size and maximum match length, the threshold length of the data expansion is 2. When the maximum matching length is more than or equal to 2, then the compressor transmits the pointer and the length with tag bit '1'. And when the length is less than 2, it transmits one uncompressed symbol with tag bit '0' and does not transmit neither the pointer nor the length. Therefore, the codeword representation is 1, match pointer and match length, i.e., (1, C_p, C_l), or 0 and a literal symbol, i.e. (1, symbol). Thus the size of one codeword is 15 or 8 bits.

Ⅲ. Hardware Algorithm for LZ compression

1. Derivation of systolic algorithm

Here, we illustrate the LZ process with a simple example. Let us define that ai is the ith symbol in the dictionary buffer and bj is the jth symbol in the uncoded symbol buffer. For the sequential algorithm in case of N=5, M=3, the detection of the maximum match substring involves five sets of comparisons as shown in Table 1. Time1 is the horizontal time index and time2 is the vertical one and both progress sequentially.

In the Table 1, the symbol, *, means the equality operator between two symbols. Consecutive match length is calculated for five sets. The index of the set with the maximum matching length is the match pointer and its length is encoded. This sequential algorithm needs $O(NM)$ comparisons.

To derive a systolic architecture, let us

Table 1. Sequential operation flow for LZ algorithm(N=5, M=3)
표 1. LZ 알고리즘을 위한 순차적인 동작흐름(N=5, M=3)

	Set	$time_1$		
$time_2$	(1)	(a1 * b1)	(a2 * b2)	(a3 * b3)
	(2)	(a2 * b1)	(a3 * b2)	(a4 * b3)
	(3)	(a3 * b1)	(a4 * b2)	(a5 * b3)
	(4)	(a4 * b1)	(a5 * b2)	(b1 * b3)
	(5)	(a5 * b1)	(b1 * b2)	(b2 * b3)

Table 2. Systolic operation flow for LZ algorithm
표 2. LZ 알고리즘을 위한 시스톨릭 동작 흐름

PE_1	PE_2	PE_3	PE_4	PE_5	PE_6
time	$(a_1 * b_1)$	$(a_2 * b_1)$	$(a_3 * b_1)$	$(a_4 * b_1)$	$(a_5 * b_1)$
	$(a_2 * b_2)$	$(a_3 * b_2)$	$(a_4 * b_2)$	$(a_5 * b_2)$	$(b_1 * b_2)$
	$(a_3 * b_3)$	$(a_4 * b_3)$	$(a_5 * b_3)$	$(a_1 * b_3)$	$(b_2 * b_3)$
set	(1)	(2)	(3)	(4)	(5)

transform the comparison sets into the following sets shown in Table 2.

Every PE_i performs one of three operations in the vertical column at every cycle. Because five PEs performs the operation simultaneously, the above matching process can be done in 3(=M) cycles. So this transformed structure consumes O(M) time and has a better performance in processing time compared with the sequential algorithm which consumes O(MN) time.

Our architecture needs only M or less time for encoding one codeword and processes one symbol per cycle, and no accumulated shift operations are needed. This is the reason why our architecture has better performance than other ones[8,9,10,11]. This improvement is mostly due to adopting label update method(LUM) which will be explained later.

2. Hardware Algorithm for Compression

As mentioned before, the functional block diagram is composed of four main blocks as shown in Fig 2. The buffer Q located to the right of the Match Block stores the input symbol to be compressed and distributes it to all PEs concurrently. Note that the M-sized uncoded symbol buffer is not needed in our architecture because the symbols in PEs are shifted simultaneously at every cycle. Each PE compares the distributed current symbol with the dictionary symbol stored in each PE and outputs match hit signal '1' when two symbols are identical, and otherwise match hit signal '0'. The Consecutive Hit Checker gets match hit signals from all PEs in the Match Block and monitors the consecutive hits.

As it proceeds, the Consecutive Hit Checker filters out disqualified candidates and finally finds the champion substring. The Pointer Generator and the Length Generator extract the match pointer and the length of the champion substring by the result of the CHC. And the Code Packer combines the pointer and the length into a codeword if the match length is more than one. Otherwise one uncoded symbol is output.

In the above process, every symbol in PE_i is shifted left in a systolic manner. To make an easy understanding of the proposed architecture, we show an example whose dictionary buffer size is 16 bytes (see Table 3).

The register in each PE is reset at initialization and later carries one symbol(*dictionary symbol*) to the left neighboring PE at each cycle. Symbols to be compressed are input into PE_N one at time. We do not need

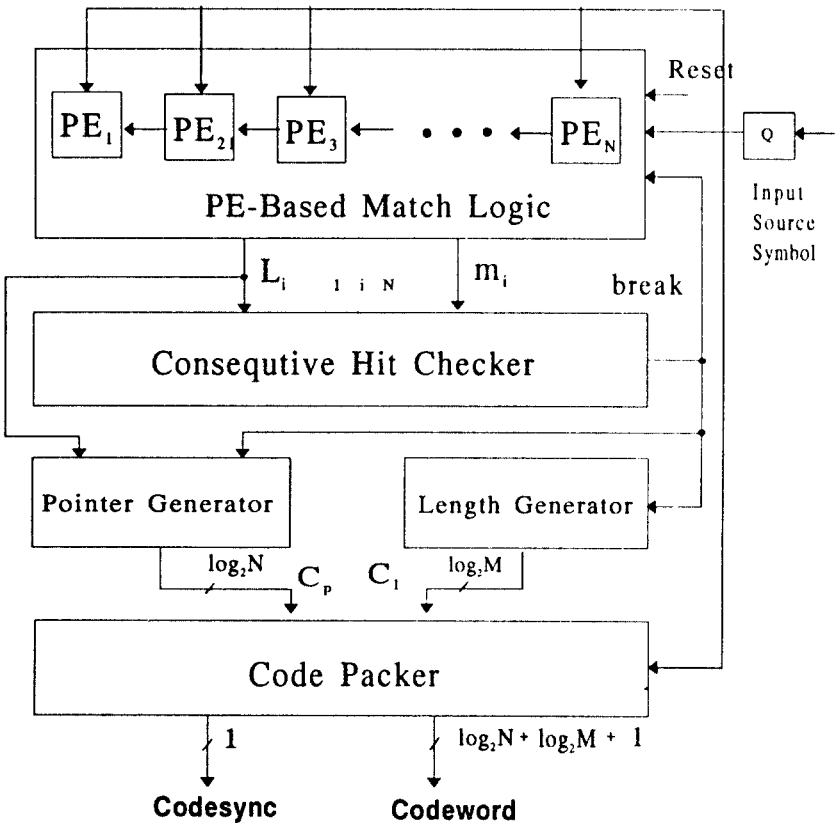

Fig. 2. System diagram of the proposed architecture
그림 2. 제안된 구조의 시스템 다이어그램

the uncoded symbol buffer, but only one byte Q register to latch and distribute the symbol ready to be compressed. Now, as shown in Table 3, we explain the compression process in detail. First, let us assume that some compression process was done and the string "THESE①ARE①YOURS①" is stored in the Match Block in such a way that ith symbol resides in PEi and the string "HERE①YOU① ARE..." is ready for the next compression.

In Table 3, the vertical column represents the time and the symbols to be compressed, and the horizontal upper row line represents the symbols stored in each PE. The combination of the match result(A) and the updated label(B) is denoted as (A/B) for each PE in the lower row line. The PE with label value '1' means that it can be one of the champion candidates and the one with '0' is the disqualified candidates.

To make the explanation simple, all labels are set to '1' at time t_0. This means that all PEs can be the champion candidates at the first cycle. At time t_1, each PE compares the current symbol 'H' with its own symbol and PE2 generates a match hit signal '1'. Its label is maintained at '1' and those of the others are reset to '0'. At time t_2, the sym-

Table 3. An Example for the proposed hardware algorithm for LZ compression. (dictionary size is 16(=N). X means unknown state.)
표 3. LZ 압축을 위한 제안된 하드웨어 알고리즘의 예. (사전의 크기는 16(=N)이고 X는 정해지지 않은 상태를 의미함)

	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7	PE_8	PE_9	PE_{10}	PE_{11}	$PE1_2$	PE_{13}	PE_{14}	PE_{15}	PE_{16}
t_0	T	H	E	S	E	ⓣ	A	R	E	ⓣ	Y	O	U	R	S	ⓣ
	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1	X/1
H/t_1	T	H	E	S	E	ⓣ	A	R	E	ⓣ	Y	O	U	R	S	ⓣ
	0/0	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
E/t_2	H	E	S	E	ⓣ	A	R	E	ⓣ	Y	O	U	R	S	ⓣ	H
	0/0	1/1	0/0	1/0	0/0	0/0	0/0	1/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
R/t_3	E	S	E	ⓣ	A	R	E	ⓣ	Y	O	U	R	S	ⓣ	H	E
	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	0/0
E/t_4	S	E	ⓣ	A	R	E	ⓣ	Y	O	U	R	S	ⓣ	H	E	R
	0/0	1/0	0/0	0/0	0/0	1/1	0/0	1/0	0/0	0/0	0/0	0/0	0/0	0/0	1/0	0/0
$ⓣ/t_5$	E	ⓣ	A	R	E	ⓣ	Y	O	U	R	S	ⓣ	H	E	R	E
	0/0	1/0	0/0	0/0	0/0	1/1	0/0	1/0	0/0	0/0	0/0	1/0	0/0	0/0	1/0	0/0
Y/t_6	ⓣ	A	R	E	ⓣ	Y	O	U	R	S	ⓣ	H	E	R	E	ⓣ
	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	0/0	0/0	1/0	0/0	0/0	1/0	0/0
O/t_7	A	R	E	ⓣ	Y	O	U	R	S	ⓣ	H	E	R	E	ⓣ	Y
	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/0	0/0
U/t_8	R	E	ⓣ	Y	O	U	R	S	ⓣ	H	E	R	E	ⓣ	Y	O
	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0
$ⓣ/t_9$	E	ⓣ	Y	O	U	R	S	ⓣ	H	E	R	E	ⓣ	Y	O	U
	0/0	1/1	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0
A/t_{10}	ⓣ	Y	O	U	R	S	ⓣ	H	E	R	E	ⓣ	Y	O	U	ⓣ
	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
R/t_{11}	Y	O	U	R	S	ⓣ	H	E	R	E	ⓣ	Y	O	U	ⓣ	A
	0/0	0/0	0/0	1/1	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0
E/t_{12}	O	U	R	S	ⓣ	H	E	R	E	ⓣ	Y	O	U	ⓣ	A	R
	0/0	0/0	0/0	0/0	0/0	0/0	1/0	0/0	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0
$ⓣ/t_{13}$	U	R	S	ⓣ	H	E	R	E	ⓣ	Y	O	U	ⓣ	A	R	E
	0/0	0/0	0/0	1/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	1/0	0/0	0/0	0/0

bol in PE_{i+1} is shifted left into PE_i, and new symbol 'E' is distributed to each PE whose match result is shown in Table 1. The label of PE_2 remains at '1' due to the consecutive match. Of course, the symbols in both PE_4 and PE_8 match the current symbol. However, since they were excluded from the candidates at the previous time t_1, their labels are maintained at '0' until the champion substring is chosen. At time t_3, there is a mismatch in

PE_2, so the pointer of PE_2 becomes the match pointer and the number of matches until now becomes the match length, and thus the first codeword, (tag bit, match pointer, match length), is (1,2,2). During time t_3, PE_6, and PE_{12} are entitled to new champion candidates for the next substring due to the match hit in its location. The other PEs are disqualified for the champion candidate. As the process goes on, the consecutive hit match is continued in PE_6 until time t_8. Therefore the next codeword is (1,6,6). At time t9, PE2, PE8 and PE_{13} which contain the symbol '①' become the new champion candidates and have their labels set to '1'. But at time t_{10}, there is no match of symbol 'A', that all PE labels are reset to '0', and the symbols '①' and 'A' are encoded as literal symbols, i.e., (0,①), (0,A), respectively. During time t11 through t_{13}, codeword (1,9,3) is obtained by the similar process.

The above example described can be summarized as the following compression procedures.

(1) By reset, the label in each PE is reset to '1' simultaneously.

(2) PEi generates the match hit signal '1' if there is a match between the dictionary symbol in PE_i and the symbol to be compressed, and otherwise PE_i generates the match hit signal '0'.

(3)By the combined condition of the match hit signal in step (2) and the current label, the update label is obtained as follows.

A: Case 1 : match hit signal == '0'.
Current label == '0' :
- Maintain the label value at '0'.
- Go to step (2).

Current label == '1' :
- Reset the label to '0'.
- If the other labels are all '0', then generate the codeword with the pointer and the length of the champion substring. If the value is less than 2, the codeword is assigned a literal symbol.
- The current label of PE_i is updated with the current match hit value.
- Reset the counter which counts the match length.
- Go to step (2).

B: Case 2 : Match hit signal == '1'
Current label == '1' :
- Maintain the label value at '1'.
- Match length is incremented by '1'.
- Go to step (1) if the match length is the maximum allowable length, otherwise go to step (2).

Current label == '0' :
- Maintain the label value at '0'
- Go to step (2).

When the current label is '1' and the match hit signal is also '1', this means that there is a consecutive hit, so we keep the current label remained, and check out if there is any match in the following symbol at the next cycle. When the current label is '0', we maintain the current label at '0' until the *'break'* of the champion candidate happens(this means that until the champion substring is determined). The label is renewed according to the current value of the match hit signal. If the champion substring is determined, the match pointer can be extracted from the previous label of PE array using a priority encoder.

The maximum length is obtained through a

incremental counter which counts the match hits of the champion substring and stores it when the 'break' happens. After that, the counter is initialized for the matching process.

Ⅳ. System Design of High Throughput LZ Compressor

In this section, we describe the system which realizes the LZ algorithm discussed above. The symbol ø in the next figures and paragraphs denotes a precharge signal.

A. Processing Element

Fig 3 shows the internal block diagram of the Processing Element(PE) which generates match hit signal mi and the current label signal L_i. The label Li in PE_i is reset to '1' at system initialization so that all PEs are entitled to the champion candidates. Signal *'break'* is used to update labels whenever the champion substring is determined. The Break signal originates from the Consecutive Hit Checker whose value is '1' for the permission and '0' for the prohibition of update. When disqualified for champion candidate(m_i==0), L_i is reset to '0' by the m_i signal. Once L_i is reset to '0', then L_i remains at '0' regardless of m_i throughout the current substring matching process.

B. Consecutive Hit Checker

Fig 4 illustrates the internal block diagram

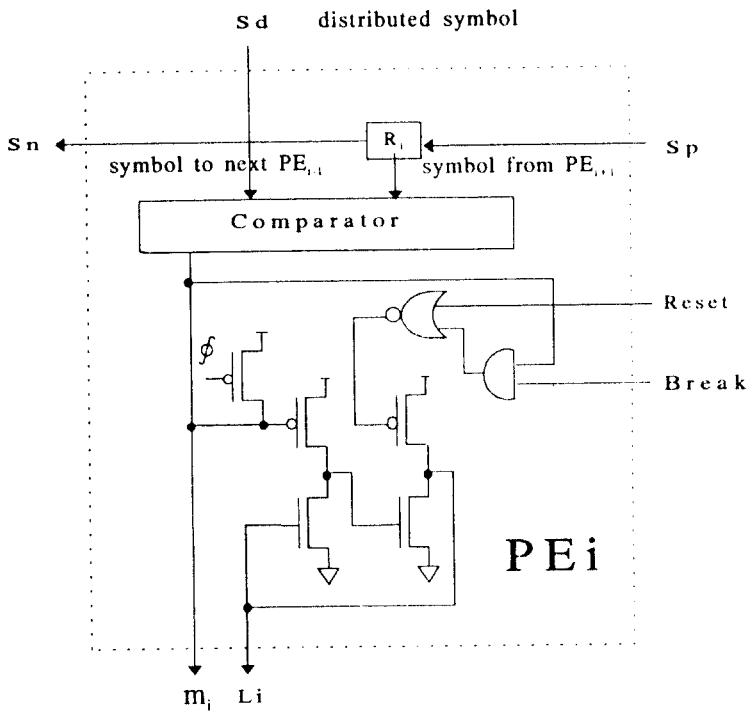

Fig. 3. Internal block diagram of the Processing Element
그림 3. 처리소자(PE)의 내부 블럭다이어 그램

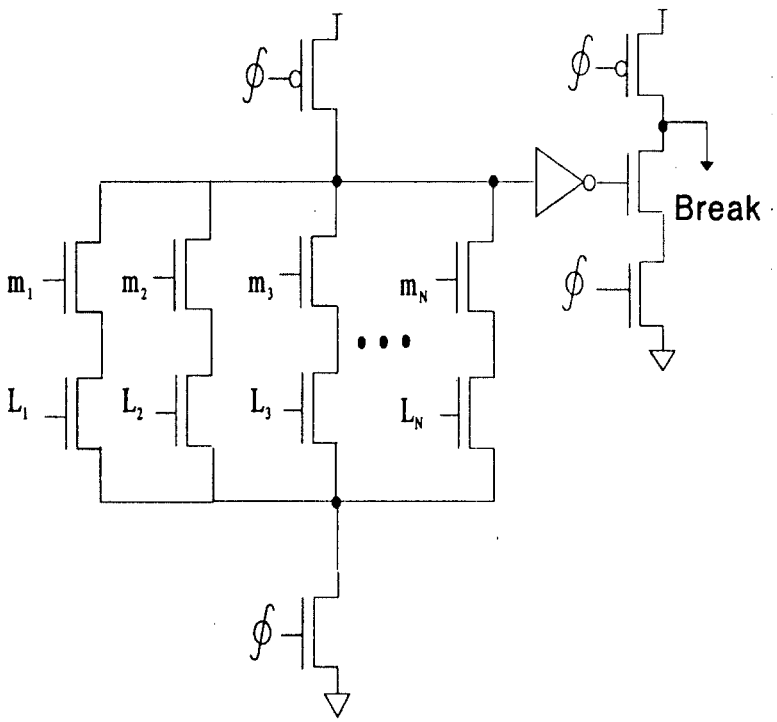

Fig. 4. Internal block diagram of the Consecutive Hit Checker
그림 4. 연속정합검사기의 내부 블럭 다이어그램

of the Consecutive Hit Checker which generates the '*break*' signal to indicate whether the champion substring is determined or not. After a precharge cycle, the '*break*' signal is evaluated as '0' if there exist any champion candidates, otherwise evaluated as '1' to inform all PEs of the beginning of a new matching process. A novel and high throughput dynamic architecture was proposed for this implementation by V. G. Oklobdzija[12].

C. Pointer Generator

The Pointer Generator(PG) converts the current labels to the matching address C_p and transfers it to the Code Packer. At every cycle, PG performs the priority encoding with the current labels and stores its result into a pointer register. And when the '*break*' occurs, the pointer latched at the previous cycle becomes the matching pointer of the current codeword and this pointer is output. Two stage pipelined priority encoding structure was published previously[9].

D. Length Generator

The Length Generator(LG) is the counter which indicates the number of match hit successes. The counter increments by one whenever the consecutive match hit is encountered after the system initialization and its count value is stored in a length register. The counter must be reset whenever the '*break*'

happens. In this case, the match length is the value stored in the length register just before the the '*break*'. To prevent overflow in the match length, '*break*' is reset to '0' when the accumulated match length surpasses the maximum count. Then a new matching procedure is initiated.

E. Code Packer

The Code Packer(CP) makes a codeword using a match pointer from the PG and a match length from the LG, or one symbol for a literal symbol when the '*break*' occurs. If the match length is less than 2, the codeword is encoded into a literal symbol which contains one tag bit(0), and one symbol. If the match length is equal to or more than 2, then the codeword is encoded into a combination of one tag bit(1), match pointer, and match length. The proposed architecture has constant input rate but variable output rate due to the characteristics of the algorithm itself. Therefore, there has to be a synchronization signal for output validity. '*Codesync*' is used for this purpose and is produced by the '*break*'. The '*Codesync*' signal '1' is for valid output codeword and '0' for the invalid one.

V. ARCHITECTURE FOR LZ DECOMPRESSION

This section explains the LZ decompression algorithm and our proposed architecture. The decompressor processes one decompressed symbol per cycle.

A. Decompression Process

To decode the codeword obtained by the modified LZ algorithm, the decompressor must have a dictionary buffer of size N for storing the recently decompressed symbols. Decompression process is summarized below.

(1) Initialize the dictionary buffer with '0'.

(2) Fetch the codeword(tag, match pointer(Cp), match length(Cl)).

(3) If th tag bit is '1', choosethe dictionary buffer part which corresponds to the match pointer C_p and go to step (4).
If the tag bit is '0', input the symbol following the tag bit into the dictionary buffer and go to step (2).

(4) Perform the following operations until the C_l is counted down to '0'.
- Shift left all the symbols in the dictionary buffer at every cycle.
- Mux all the dictionary registers to the right-most register of the dictionary buffer selected by the C_p.

(5) If EOF is encountered, finish the decompression process after N remaining symbols in the dictionary buffer are output. Or else go to step (2).

While the decompression continues, decompressed symbols are output from the left-most dictionary register after N+1 cycles from the initialization.

B. Architecture for decompression

The proposed decompressor is composed of mainly four blocks as shown in Fig 5 : Dictionary buffer, Pointer Selector, Codeword Loader, and Symbol Selector. The Codeword Loader generates the '*loadsync*' signal to accept new codewords. The '*loadsync*' signal is synchronized by the '*finish*' signal from the Symbol Selector. If the tag of the current codeword is '0', which means a literal symbol, the Symbol Selector controls the MUX to multiplex the direct symbol to the dictionary buffer. Then it generates the '*finish*' signal to accept the next new codeword. If the tag

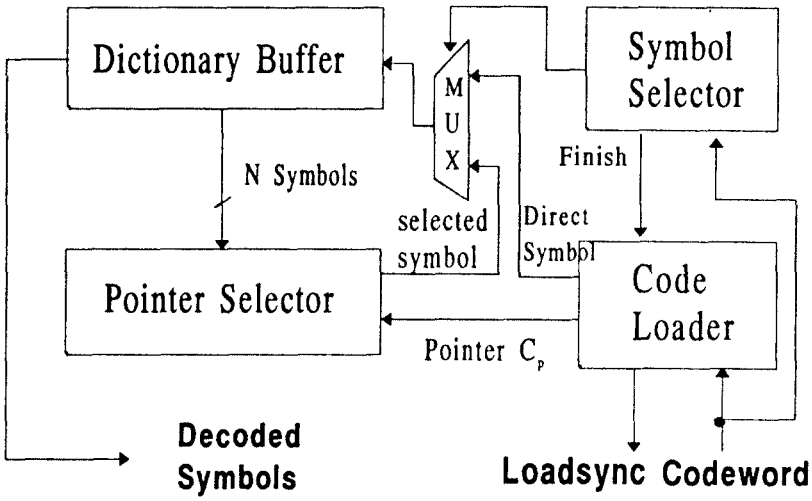

Fig. 5. Block diagram of LZ decompressor
그림 5. LZ 역압축기의 블록 다이어그램

is '1', the Pointer Selector selects the dictionary register corresponding to the C_p and inputs this symbol to the right-most input buffer. In this case, the Symbol Selector loads the C_l count into a down counter and decreases it by one per one clock cycle. When the count becomes '1', the Symbol Selector transmits '*finish*' signal to the Code Loader which generates the '*loadsync*' signal to receive the new codewords.

Ⅵ. Conclusions

In this paper, a novel systolic architecture is proposed for the Lempel-Ziv data compression which yields high throughput rate. No idle cycles for accumulated shift are needed. Input bytes are processed one per every clock cycle without waiting the 'shift and update'. Thus the data input rate of 100 mega symbols per second is obtained when the clock frequency is 100 MHz. Based on our systolic hardware for LZ algorithm, the VLSI architectures for encoding and decoding LZ-based modified algorithm are developed.

In encoding process, a novel label updating method is proposed for consecutive hit check to remove accumulated shift cycles which are the frequent drawbacks of the other architectures[8,9,10,11]. To speed up the clock frequency, a dynamic structure is also proposed for the label updating unit which was one of the critical paths of our system.

Our architecture is very modular and thus can be expanded very easily to various dictionary sizes. For example, the dictionary size can be doubled by cascading two existing 1024 byte architectures with slight modifications to the counters and the '*break*' signals, and address generation logic. The derived systolic architecture demands simple control circuitry. For the encoder, no control signals are provided via external pins except a reset signal. Comparison, shift and update are automatically done at every cycle. Complex signal such as barrel shifter control signals are needless because the accumulated shifts are deleted in the proposed architecture.

Moreover the '*break*' signal and label update can be implemented with minimal hardwares. Static versions can be considered instead of the dynamic circuits shown in Fig 3 and Fig 4. In this case, OR gates can be used for the Consecutive Hit Checker, and several gates including clocked gates can be used for label update. The architecture with the static circuits require less design time than the one with the dynamic circuits with minimal speed decrease.

The critical path lies in the PE and the CHC. The static version of the proposed Lempel-Ziv compressor was verified through verilog HDL(Hardware Description Language) and the source code was synthesized using 0.6 μm CMOS COMPASS ASIC Synthesizer. The critical path is composed of a 7-bit comparator, some clocked gates, and one 1024-input OR gate. The netlist simulation result shows that the critical path delay is 8.9 ns. According to our netlist simulation implemented in 0.6 μm CMOS technology, the chip is expected to operate up to 100 MHz(100 mega samples) which is fast enough for real time data compression applications like text and image compression over the communication channels.

References

1. R. Gallager, "Variations on a theme by Huffman," *IEEE Trans. Inform. Theory*, vol. IT-24, pp.668-674, 1978.
2. S. Golomb, "Run-length encoding," *IEEE Trans. Inform. Theory*, vol. IT-12, pp.399-401, July. 1966.
3. G. Langdon, "An introduction to arithmetic coding," *IBM J. Res. Develop.*, vol. 28, no. 2, pp.135-149, Mar. 1984.
4. J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. IT-23, pp.337-343, 1977.
5. J. Ziv and A. Lempel, "Compression of individual sequences via variable rate coding," *IEEE Trans. Inform. Theory*, vol. IT-24, pp.530-536, 1978.
6. T. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp.8-19, 1984.
7. C. Thomborson, "The V.42bis Standard for Data-Compressing Modems", *IEEE Micro*, pp.41-53, Oct. 1992.
8. N. Ranganathan and S Henriques, "High-speed VLSI designs for Lempel-Ziv-based data compression", *IEEE VLSI Algorithms and Architectures : Advanced Concepts*, pp.255-265, 1993.
9. R. Y. Yang and C. Y. Lee, "High-throughput data compressor designs using content addressable memory," *IEEE ISCAS '94*, vol. 4, pp.147-150.
10. B. Jung and W. Burleson, "A VLSI systolic array architecture for Lempel-Ziv-based data compression," *IEEE ISCAS '94*, vol. 3, pp.65-68.
11. B. W. Y. Wei, R. Tarver, J. S. Kim, and K. Ng, "A single chip Lempel-Ziv data compressor," *IEEE ISCAS '93*, pp.1953-1955.
12. V. G. Oklobdzija, "An algorithm and novel design of a Leading Zero Detector circuit : comparison with logic synthesis," *IEEE Trans. on VLSI systems*, vol. 2, no. 1, March. 1994.

南 承 鉉(Seung Hyun Nam) 정회원

Received the B.S. and M.S. degrees in Electronic Engineering from Yonsei University in 1986 and 1988, respectively, where he is currently working toward the Ph.D. degree. He has been with Daewoo Co. since 1994.

His research interests include the VLSI design of digital signal processing, image processing, and image compression algorithms.

李 文 基(Moon Key Lee) 정회원

Received the Ph.D. degree in Electronic Engineering from Yonsei University in 1973.

He also received the Ph.D. degree in Electronic Engineering from the University of Oklahoma in 1980 and was a senior researcher in ETRI.

Since 1982, he has been a Professor in the department of Electronic Engineering at Yonsei University. He has been the chief of the Research Institute of ASIC Design located in Yonsei University since 1992. He was the vice president of the Korea Institute of Telematics and Electronics in 1994 and is now the president of the Korea Institute of Telematics and Electronics. His research interests include VLSI & CAD, the design of ASIC chips, etc.

李 泰 永(Tae Young Lee) 정회원

Received the B.S. degree in Electronic Engineering from Yonsei University in 1994, where he is currently working toward the M.S. degree. His research interests include the VLSI design of digital signal processing, image processing, and image compression algorithms.

李 鎔 錫(Yong Surk Lee) 정회원

Received the B.S. degree in Electrical Engineering from Yonsei University in 1973. He received the M.S. and Ph.D. degrees in Electrical Engineering from the University of Michigan, Ann Arbor, in 1977 and 1981, respectively.

He has been with various Silicon Valley semiconductor companies such as Sperry-Univac Computer, Hyndai Electronics USA, National Semiconductor, Performance Semiconductor, and Intel Corporation as a design engineer. He is now an Associate Professor in the department of Electronic Engineering at Yonsei University. His research interests include the design of microprocessors, SRAMs, caches, image compression system, etc.