VLSI design for high-speed LZ-based data compression

J.-M.Chen and C.-H.Wei

Abstract: A simple real-time parallel architecture for CMOS VLSI implementation of a Ziv–Lempel data compression system is presented. This encoding system employs a linear systolic array to find concurrently the matches between each input data character and its corresponding dictionary, and can easily achieve ideal compression ratio by cascading the chips of the encoding cell. A new encoding architecture is proposed to improve the encoding speed and reduce hardware complexity for the encoding cells. In addition, the number of memory accesses is reduced to save power consumption for high-speed applications. The encoder codes one character (more than eight bits) per encoding cycle. The clock rate by Verilog simulator can be constrained below 15ns using the Compass standard cell library for the 0.6μm CMOS process.

1 Introduction

In recent years the need to develop efficient data compression methods has increased considerably owing to the increasing applications of data compression in various areas. The most widely used classes of lossless data compression algorithms are those developed by Ziv and Lempel in 1977 and 1978, labelled as LZ77 and LZ78, respectively [1, 2]. In the LZ77 algorithm, pointers are used to denote phrases in a fixed-size window that precedes the coding position. There is a maximum length for substrings that may be replaced by a pointer, given by the parameter F (typically 2^4~2^5). These restrictions allow LZ77 to be implemented using a 'sliding window' of N characters. In this scheme the first $N - F$ characters have already been encoded and the last F characters constitute a lookahead buffer. The window is illustrated in example 1 (see Fig. 1).

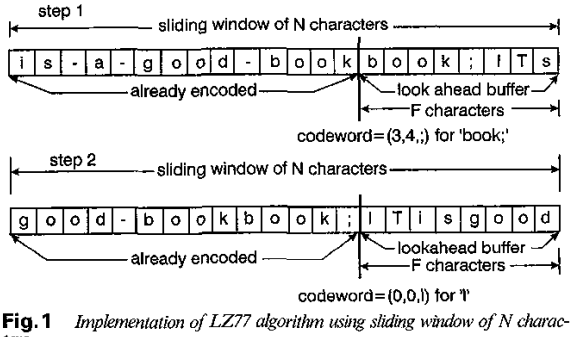

Fig. 1 *Implementation of LZ77 algorithm using sliding window of N characters*

To encode a character, the first $N - F$ characters of the window are searched to find the longest match with lookahead buffer. The match may overlap with the buffer but obviously cannot be the buffer itself. The longest match is then encoded into a triple codeword (o, l, a) where o is the offset of the longest match from the lookahead buffer, l is the longest length of match, and a is the first character that did not match the substring in the window. The window is then shifted right $l + 1$ characters, ready for the next encoding step. Attaching the explicit character to each pointer ensures that coding can proceed even if no match is found for the first character of the lookahead buffer.

In the LZ77 algorithm, whenever there is no match or if there is a match of length one, each symbol that respectively constitutes the matched substrings would be substituted by a codeword of three bytes long. This may result in expansion rather than compression in the encoding process. If many such unfavourable codewords occurred in a file, the compression achieved would be low. The LZSS algorithm [3], one of the variants of the LZ77 algorithm, adopts a free mixture of pointer and characters to replace the triple codeword in LZ77 [1] to overcome the expansion problem. A character itself will be used only when a pointer takes more space than the characters it encodes. An extra bit, known as *flag*, is added to each pointer or character to distinguish between them. The output is packed so that there are no unused bits. Many designers have incorporated this technique in the software implementation which significantly improves the compression ratio. However, the speed of data compression by software is usually not satisfactory for a real-time system.

Several LZ hardware architectures have been presented in the literature [4–8]. The most computationally expensive step in the hardware is to search for the maximum matching strings in the sliding dictionary. However, the LZ-based algorithm must be executed at a very high throughput rate for real-time transmission and storage. The content addressable memory (CAM) approach [7] can provide constant time to search the matching strings for each symbol, but it must consume static DC current for many of the CAM cells during the match action cycle. If the size of the sliding window is increased the power consumption problem will dominate the chip performance.

We present the design of a systolic array processor for efficient implementation of the LZSS compression technique by using wrap architecture [9]. This new approach can find a maximum match length for each clock cycle. The new hardware structure can execute the data compression task on-the-fly in the real-time communication system.

IEE Proceedings online no. 19990535

DOI: 10.1049/ip-cds:19990535

Paper first received 16th January 1998 and in revised form 26th March 1999

The authors are with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, Republic of China

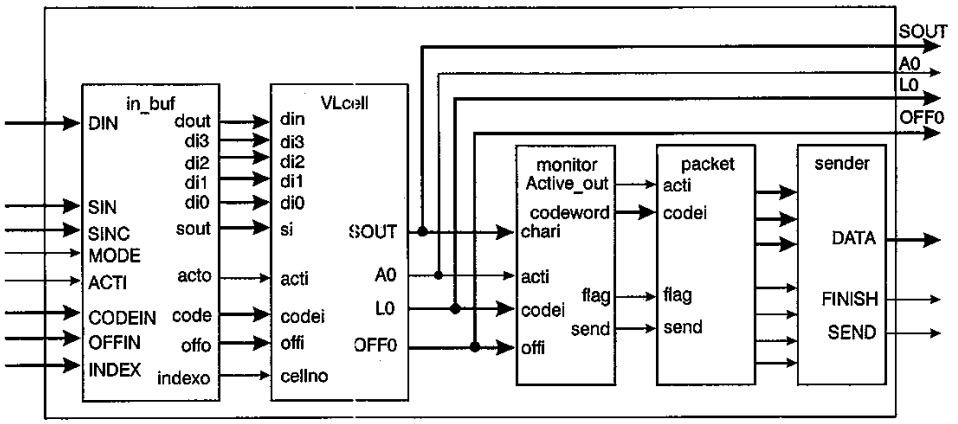

Fig. 2 *Block diagram of encoder*
CODEIN: initial *length* value for codeword (default: 0)
OFFIN: initial *offset* value for codeword (default: 0)
INDEX: cell number for last stage of encoding cell (default: 0)
SOUT: output character of cell 0 (the last stage of systolic array)
A0: active signal output of cell 0
L0: *length* output of cell 0
OFF0: *offset* output of cell 0

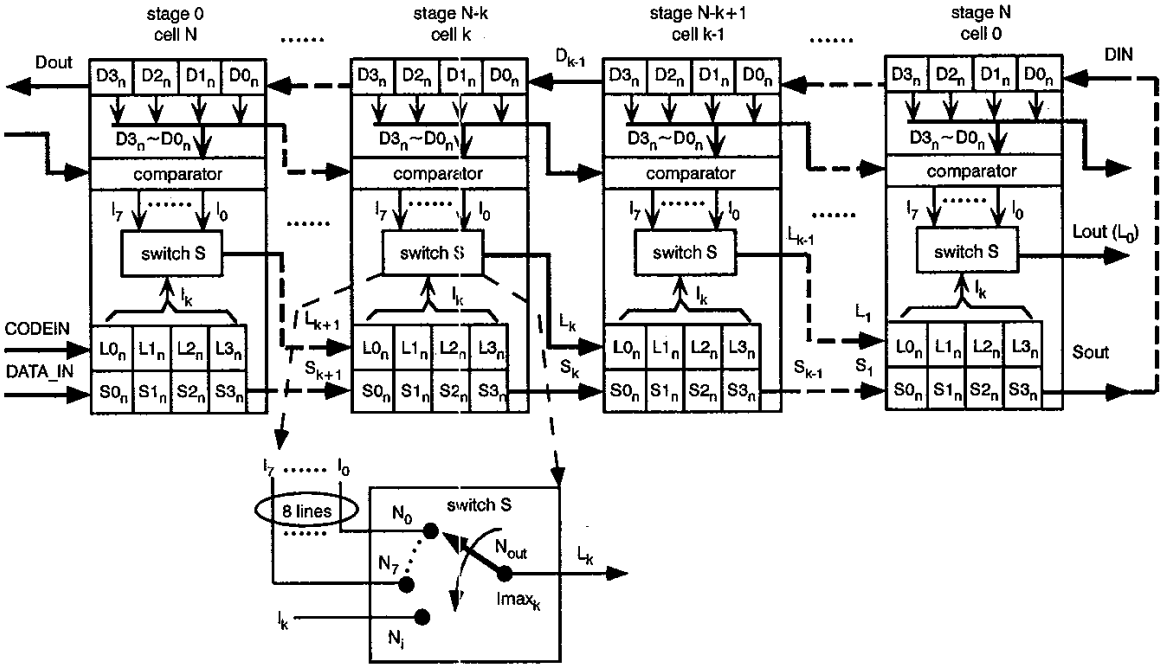

Fig. 3 *Array data flow of wrap architecture for encoding system*

To implement the encoder with VLSI ASIC design technology, the system is divided into several modules and facilitated by the systolic architecture. As shown in Fig. 2, the encoding system contains several blocks: VLcell, monitor, packet, sender, and in_buf. The VLcell block is composed of a segment of the systolic array illustrated in Fig. 3 to find a series of optimal matches. The in_buf block includes input pads, registers, and buffers to receive the input signals and convert them to internal CMOS signals. DIN is the input of encoded character string for the dictionary registers; SIN and SINC are the input uncoded character string under normal mode and cascade mode, respectively. When the ACTI signal goes to high the encoder begins to compress the uncoded input character string. The size of the dictionary is configurable up to a suitable window size by adjusting the MODE value of encoder. If the MODE value is set to 0, the encoder can cascade several encoders in a pipeline and the buffer size of a sliding window embedded in VLcell can be expanded to a suitable width. Typical width of the window is 2~8 K of dictionary, which has 0.25~1 K cells. The bit number n for the offset is 11~13. The primary advantages of our system are high speed and simple hardware structure with expandable dictionary size to increase the compression ratio easily. We also adopt an efficient output buffer to control the output rate and convert the output packets to and from words of fixed width, e.g. 16 bits for compatibility with typical computer hardware. Since power consumption is an important consideration for high throughput-rate application, we use a parallel process technique to reduce the number of register access times and reduce power dissipation.

2 Wrap architecture for encoding and decoding

In the systolic array described in [10], data pass from one processor to the neighbouring ones in a regular, rhythmic pattern. As described in [11], the array data flow of the

wrap architecture for encoding is given in Fig. 3. Each encoding cell, consisting of four dictionary registers $D0_n$ ~ $D3_n$ and four storage registers $S0_n$ ~ $S3_n$, is capable of recording four characters. Data flows from left to right along the bottom through the storage registers $S0_n$ ~ $S3_n$ and then wraps back, flowing from right to left along the top through the dictionary registers $D0_n$ ~ $D3_n$. Notice that the order direction of stage number is opposite to that of the cell number. Each cell selects one of the four register's contents ($S0_n$ ~ $S3_n$) to compare with the contents of its dictionary registers $D0_n$ ~ $D3_n$ and those of the four dictionary registers obtained from its preceding stage, respectively. Thus, the symbols l_0 ~ l_7 represent the corresponding match length values for encoding cell k, and the symbol l_k represents the optimal length selected from one of four registers $L0_n$, $L1_n$, $L2_n$ and $L3_n$. The symbol L_{k+1} represents the length received from the preceding stage (encoding cell $k + 1$). The decoder system has the same register structure and data flow but a different decoding unit cell.

The encoding cell k selects the maximal length among nine length values (l_0 ~ l_7 and l_k), and send the largest length L_k with its appending *offset* O_k value to the encoding cell of next stage, cell $k - 1$. The O_k value is determined by the path of largest length L_k. The offset Ok of cell k is equal to that of $8k + i$ for L_k being equivalent to l_i (where l_i is one valus of l_0 ~ l_7). Otherwise, O_k is copied from the *offset* O_{k+1} value of the preceding stage (cell $k + 1$) when the largest length L_k is equal to l_k. Each stage utilises switch S to select the largest length L_k and its appending *offset* value O_k. Nine length values (l_0 ~ l_7 and l_k) in each encoding cell k are initially set to zero when the encoding cell k begins to process the input string. Assuming that l_0 first becomes unity in the first stage (cell N), the position of switch S is switched to node N_0. Thus, only l_0 and its appending *offset* value O_k ($8N$) can be sent to the second stage. In the next cycle l_0 will continue to add one for the consecutive match between the next input character and the content of same storage register. The value of l_0 is still larger than the values of the others (l_1 ~ l_7 and l_k), even though l_1 ~ l_7 and l_k are ready to increment by one. Hence, the switch position is still located at node N_0, and this encoding cell still sends l_0 and its *offset* value O_k ($8N$) to the next stage. From now on, if l_0 continues to increase during the following encoding cycles, switch S will stay at node N_0 until l_0 stops increasing. Thus, we ignore the values of l_1 ~ l_7 and l_k during these encoding cycles until l_0 is reset to zero. From this description, it can be seen that the length value is not unique in determining the position of switch S.

In our design we utilise only the group code and comparison results instead of the real length value to control the position of switch S. Hence, each encoding cell just sends (*code*, *offset*) pairs instead of (*length*, *offset*) pair to the next stage. Finally, these pairs are sent to 'monitor' to extract a sequence of matches that exactly include the corresponding input stream. Thus our system doesn't use any counter or magnitude comparator in the series of encoding cells to achieve optimal compression ratio. Hence, these modifications reduce the complexity in hardware implementation.

3 VLSI implementation of encoding scheme

As described in Fig. 3, the encoding cell selects the optimal length based on the match length which indicates the relationship between adjacent input characters. If the match length is monotonically increased among adjacent input characters, these characters will be compressed into the same codeword. Hence we classify the characters as the same group, where the first character is named as group leader.

As an example, cell k and cell $k - 1$ are two consecutive cells, where cell k is the preceding stage of cell $k - 1$. The uncoded character string is $\{S_0S_1...S_{12}...\}$, being equal to {*groupGroupNOT*...}, and the encoded string is $\{D_0D_1...D_{23}...\}$, such as {*ThegrouproupFGroupEND-YZA*...}, stored in the dictionary registers. As shown in Fig. 4, during clock cycle i cell k compares the new

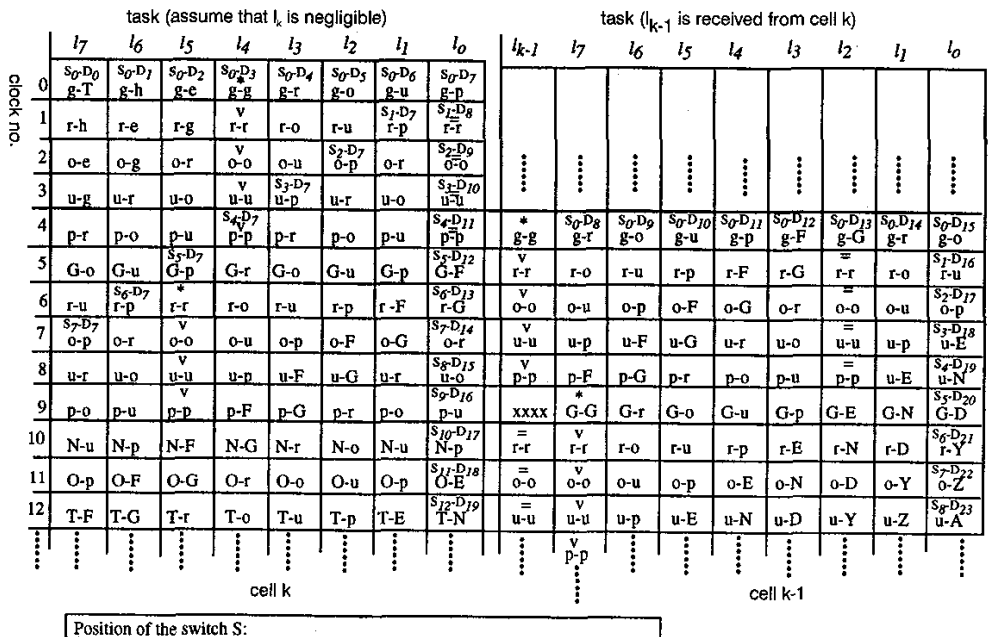

Fig. 4 *Example of encoding cell*

uncoded character S_i with the eight encoded characters $D_iD_{i+1}...D_{i+7}$ and obtains nine length values (l_0, l_1, ..., l_7, l_k), where l_k is obtained from the preceding stage. For the sake of simplification, we assume that the l_k value from cell k + 1 is always zero and negligible in Fig. 4. The symbol '–' represents the comparison operator between the uncoded character and the encoded character, and '=' represents the input character matching the encoded character. From cycles 0 to 4 in cell k the first five uncoded characters {*group*} find a series of matched string $\{D_3D_4D_5D_6D_7\}$ in dictionary. Thus the l_4 value will be sent to cell $k-1$, i.e. it is the l_{k-1} *value of cell* $k-1$, and the switch position should be at node N_4 from cycles 1 to 4. During these five clock cycles the five input characters {*group*} are assembled into the same group by cell k, where the first uncoded character S_0, *g*, marked by symbol '*' is the group leader and the following four characters {*roup*} marked by symbol 'v' are its group members. Since the l_0 length value of cell k is always less than the l_4 value of cell k from cycles 0 to 4, the l_0 value of cell k is negligible. During cycle 5 there is no match and it is not necessary to determine the switch position of cell k. From cycles 6 to 9 the encoding cell k can find a series of matches between the input characters string $\{S_6S_7S_8S_9\}$, {*roup*} and the coded characters string $\{D_8D_9D_{10}D_{11}\}$. Thus, these four character {*roup*} should be assembled into the same group, and the switch position of cell k should be switched to node N_5 and send l_5 value of cell k to cell $k-1$.

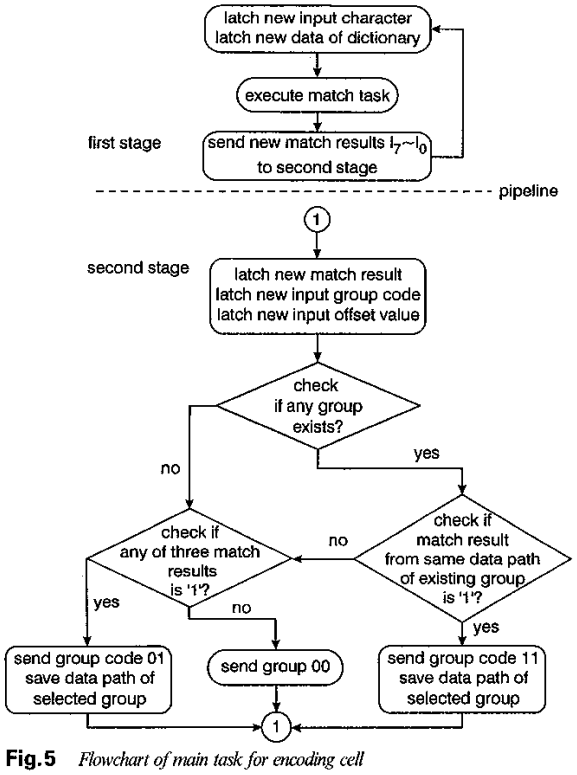

Fig.5 *Flowchart of main task for encoding cell*

The selected group in cell k will be transmitted to cell $k-1$, and it (l_{k-1} of cell $k-1$) becomes a candidate to compete with the eight comparison results of cell $k-1$ after four clock cycles. If the group leader of l_{k-1} first appears in cell $k-1$, the (*length*, *offset*) data of l_{k-1} will be transmitted to cell $k-2$. As shown in Fig. 4 the group composed of string {*group*} becomes the unique group in cell $k-1$ from clock cycles 4 to 8, so the switch position of cell $k-1$ is allocated N_i node. Therefore the (*length*, *offset*) data of comparison results between the first five input characters {*group*} and the encoded string $\{D_3D_4D_5D_6D_7\}$ in cell k will be transferred transparently from cell k to cell $k-1$, cell $k-2$, ... and the final stage (cell 0) according to the arrival time of group leader in the future consecutive cells.

As described in Fig. 4, the main task of the encoding cell is to control the switch position. If the arrival time of any group leader is earlier than the others, the switch position of that encoding cell will be switched to the node that can construct a data path for the optimal group leader to pass, such as node N_4 of cell k during clock 0 or node N_i of cell $k-1$ during clock 4 in Fig. 4. The flowchart of the encoding cell's main task can be shown as Fig. 5, where codes 01 and 11 indicate that the character is a group leader and a group member, respectively. Code 00 indicates that the character does not belong to any group. Switch S of the encoding cell shown in Fig. 3 can be treated as a finite state machine which can save the data path of the existing group and control the switch position. If one data path is selected in this encoding cycle, it has the highest priority to be selected during the next encoding cycle, e.g. the first group string {*group*} in cell k from cycles 0 to 4 and in cell $k-1$ from cycles 4 to 8, respectively. On the contrary, the finite state machine will select the optimal data path again when the original data path gets an unmatched result during the next encoding cycle. For example, during clock cycle 9, in cell $k-1$ the switch position is switched to N_7 position.

Table 1: Transition table of finite state machine for each state register

Current state $Q(t_1+T_{dr})$	Current input $I^*(t_1+T_{dr})$	$NL[1]$ $N(t_1+T_{dr}+T_{dx})$	Next state $Q(t_1+T+T_{dg})$
1	0	X	0
1	1	X	1
0	0	X	0
0	1	1	1
0	1	0	0

T: clock period
t_1: positive edge time for the current clock clcye
$NL[1]$: inverse value of L_n [1], MSB of L_n
T_{dr}: delay time of D type register for latching new data
T_{dx}: delay time of combinational logic circuit for NL [1]
T_{dg}: delay time of one AND gate
$I^*(t_1+T_{dr})$: I_0 (t_1+T_{dr}) for q_{0n}, I_1 (t_1+T_{dr}) for q_{1n}, ..., I_k (t_1+T_{dr}) for q_{kn}

The block diagram of the modified encoding cell is illustrated in Fig. 6, where the diagram of the finite state machine is marked by the dash line, and its transition table is shown in Table 1. In Fig. 6, nine state registers are used to generate a new group code L_n and its appending *offset* values O_n. The individual value of l_0~l_7 is equal to 1 when the respective comparison result is matched, otherwise it is 0 for unmatched status. These filtered group codes and their appending *offset* values arrive at the MONITOR to generate the optimal code pair (*length*, *offset*). From now on the main task of the encoding cell can be adjusted to encode the group codes based on the current match results, the current state, and the current incoming group code as follows:

$$\text{MSB} \quad L_n[1] = (l_0 \cdot q_{0n}) + (l_1 \cdot q_{1n}) + (l_2 \cdot q_{2n}) + \ldots + (l_7 \cdot q_{7n}) + (l_k \cdot q_{nk}) \tag{1}$$

$$\text{LSB} \quad L_n[0] = l_0 + l_1 + l_2 + \ldots + l_7 + l_k \tag{2}$$

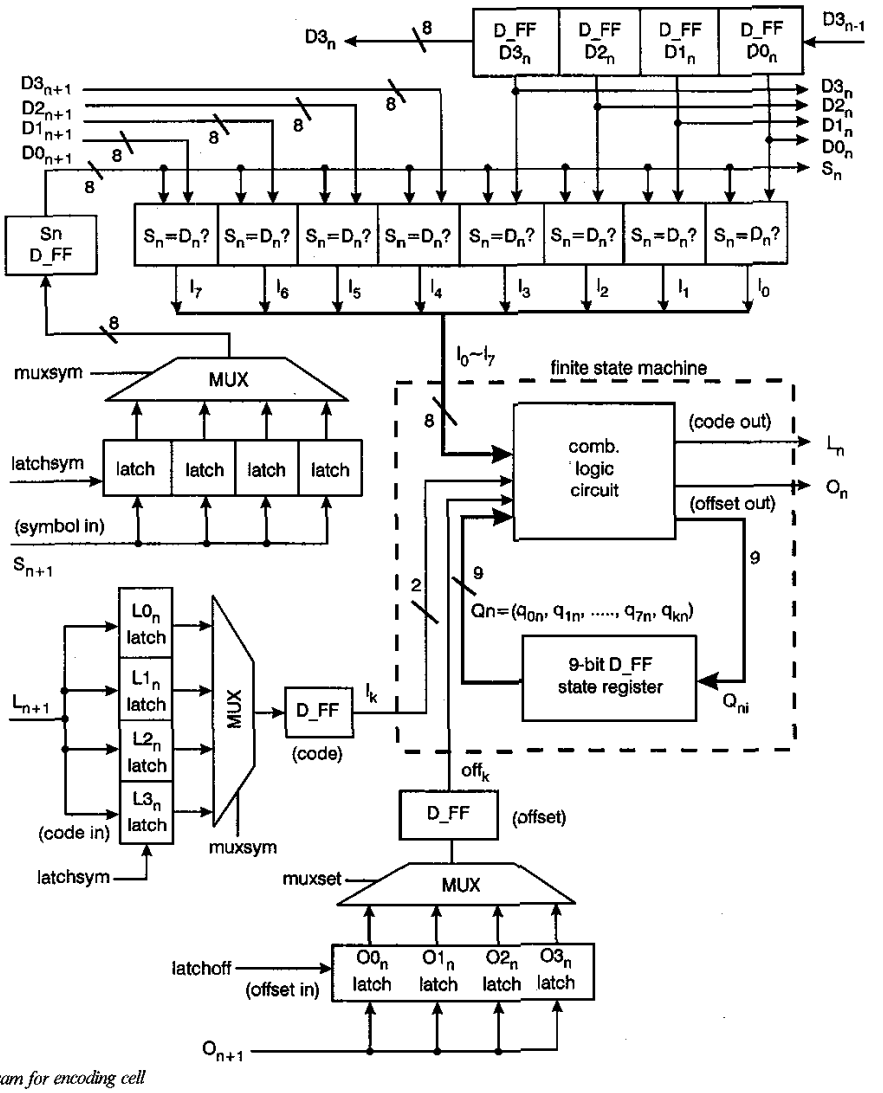

Fig. 6 *Circuit diagram for encoding cell*

where symbol '+' and symbol '·' represent logical AND and OR operators, respectively. We use a simple finite state machine circuit to generate the optimal group code according to $l_0 - l_7$, and l_k so that neither counter nor magnitude comparator are needed. This system has been implemented by using a 0.6μm standard cell library, e.g. provided by Compass Company, USA.

The timing sequence of the encoding cell given in Fig. 6 is shown in Fig. 7. At time T_0 for encoding cell n, all of the D-type flip-flops latch the individual new values, then compare the input character S_n with the eight encoded characters $D3_n$~$D0_n$ of cell n and $D3_{n+1}$ ~ $D0_{n+1}$ from the dictionary registers of cell $n + 1$. The current comparison results l_0~l_7 and l_k are transmitted to the finite state machine to obtain a new code L_n and the new next state Q_{ni}. At time $T_0 + T$, cell n latches the new state of group leaders Q_n, where Q_n is composed of q_{0n}, q_{1n}, ..., q_{7n}, q_{kn}. Finally, encoding cell n utilises the new state values to obtain the new offset output value O_n. Thus the encoding cell is separated into two pipeline stages to execute the encoding process for one input character. As simulated by the Synopsys synthesis tool, the critical path of cell n is identified from the S_n, $D3_n$~$D0_n$, and $D3_{n+1}$~$D0_{n+1}$ to the output of the new code L_n. The critical path delay is about 7ns, calculated from positive edge ck to $L0$ as shown in Fig. 8 by Verilog simulation. Hence the clock period T should be larger than 7ns to get the correct encoding results for the encoding cell.

3.1 Monitor module

As depicted in Fig. 9, the monitor module receives the character, offset and group codes from cell 0 and decodes the longest length based on group code, where the longest length is equivalent to *pending* values in Fig. 9. The main function of the monitor, as shown in Fig. 9, is to indicate whether one codeword is found or not by the send signal.

Suppose a pointer uses the space of p uncoded characters, the encoding rule is based on the length value of the codeword. If the length value is greater than that of p, the monitor will send the codeword. Otherwise, the monitor sends the uncoded character.

3.2 Packet module

The packet module receives codewords from the monitor, concatenates these codewords together and segments them into 16-bit words for output. In [12], encoding and decoding buffers have been employed for the Huffman coding algorithm. Since the Huffman code has variable-length codewords it needs a complicated programmable logic array (PLA) to record the lengths of the individual code-

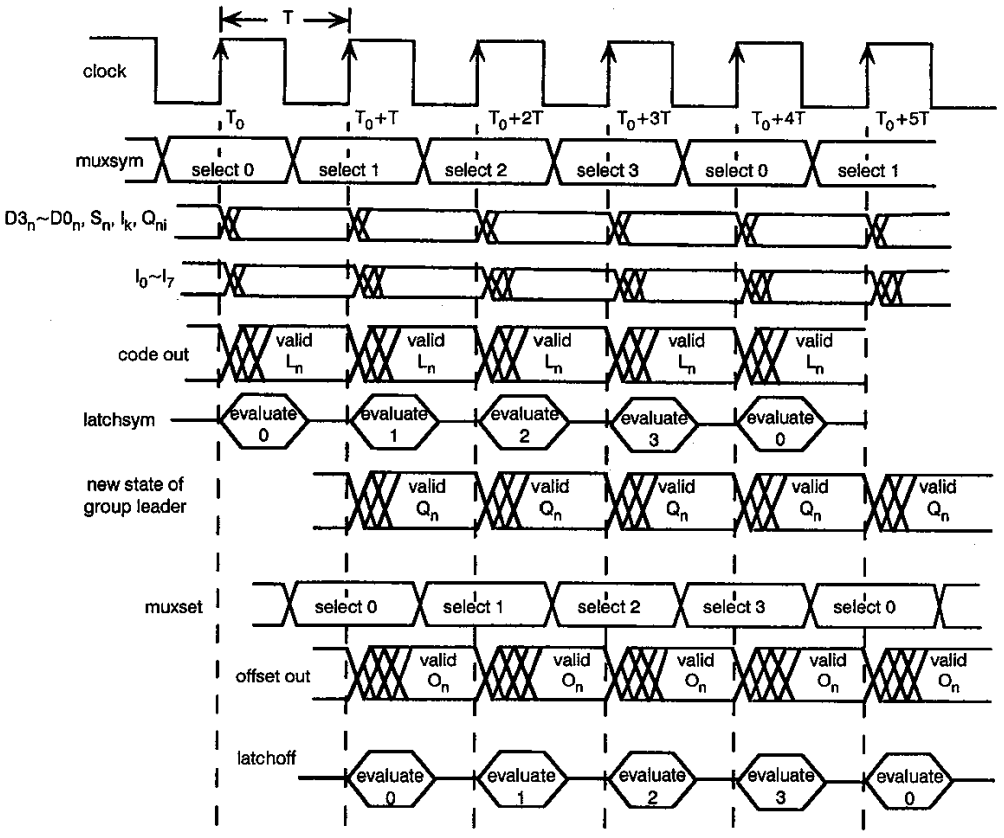

Fig. 7 *Timing sequence for encoding cell*

words for executing the packing task. The LZSS codeword has two codeword formats, i.e. uncompressed and compressed forms. Thus, it is not necessary for the LZSS codeword to utilise the PLA table. In our design a simple combinational logic circuit instead of the PLA table is used to control the barrel shifter.

The circuit diagram of the packet module is shown in Fig. 10. The 16-bit register W2 is the output latch. If the sum of Residue and Length is greater than 15, the Full output of the accumulator will be set to High and the packet module will put the contents of the W2 register at the positive trigger of Out_clock signal. Since the maximum bit-number of the concatenation of Flag and codeword is greater than 16, the sum of Length and Residue may be equal to 32, sum of 17 and 15. In this case there are two 16-bit packets being ready for output, the system will send W2 contents first and then W3 contents at the positive trigger of Out_clock and that of Out_again, respectively. Here Out_again is generated by Next_carry signal of the accumulator.

The parallel concatenation of codewords is done by MUX1 and Barrel_Shifter, which provide 16-bit windows on their 33 input bits. MUX1 is controlled by the Flag signal and shifts the codeword from W0 into W1 so that the rightmost bit of W1 is the last bit of the codeword. Consequently the data stored in W1 is ready to concatenate with the next codeword. The Barrel_Shifter is controlled by the residue value which represents the number of the residual bits in W1. The residual bits in W1 are determined by MUX1.

3.3 Sender module

When the Over signal of the packet module falls from high to low the system is to finish the encoding process. When the value of the residue register is not equal to zero all the final residual bits in W2 or Barrel_Shifter must be sent to the sender module for output. If the packet module has just sent an output in the previous ck1 cycle, the 16-bit output of Barrel_Shifter is sent to sender. Otherwise, packet sends the 16-bit output of register W2.

4 VLSI design of decoding scheme

A block diagram of the decoding processor is shown in Fig. 11. The functions of the major modules are described as follows. The input to the Frontend is a bit stream without explicit word boundaries. The Frontend has to decode a codeword, determine its length and Flag, and shift the input data stream by the number of bits corresponding to the decoded code length before decoding the next codeword. The pre-processor separates the codeword into the corresponding offset and character parts, and generates Active=1 for the successive *length* cycles. As the system receives a Ready signal the counter generates three enable signals to start the decoding process. As enable_0 rises to high, the 16-bit register W1 stores the first 16-bit register. During enable_1 cycle, W1 stores the second 16-bit input stream and the first 16-bit input stream must be shifted into W0. The flag and length of the first codeword are determined at the same cycle. When the enable_2 signal goes high the Frontend produces the first codeword and its flag, and the system begins to enter the normal decoding process in the consecutive cycle.

The circuit diagram of the pre-processor for decoding is shown in Fig. 12. The pre-processor has to separate the codeword into character and offset, generate the Active signal to the rightmost cell, and send a Request signal to Frontend to process the next best match. In our decoding

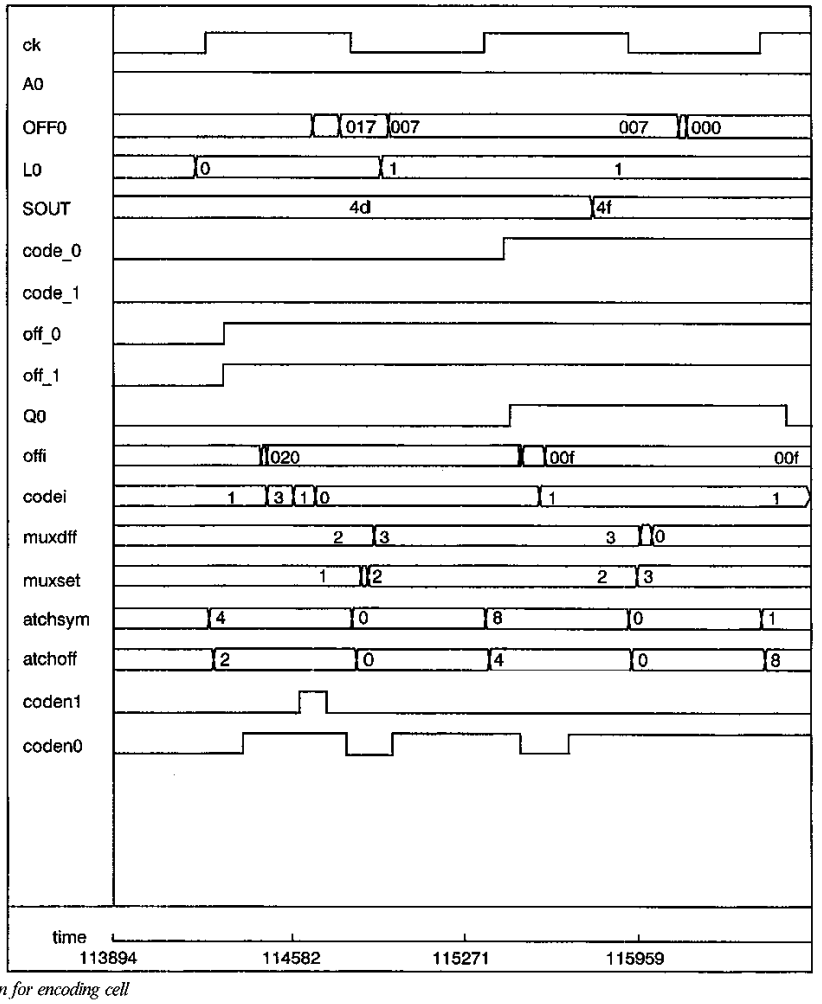

Fig. 8 *Verilog simulation for encoding cell*

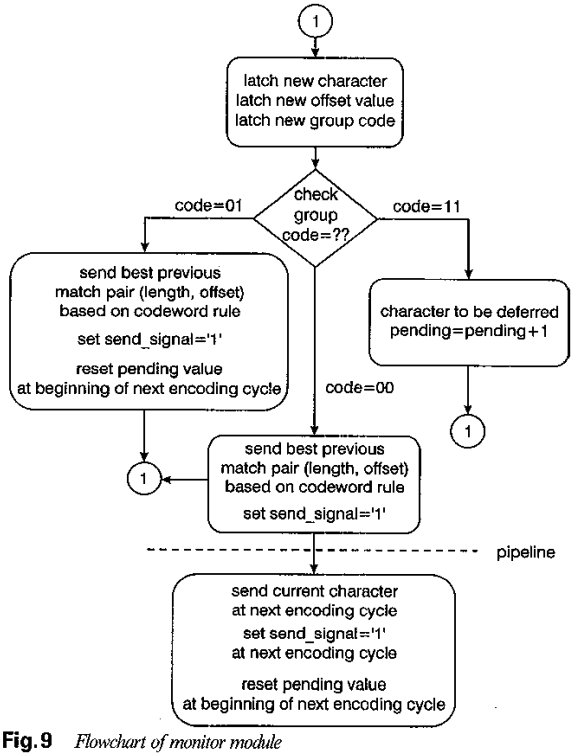

Fig. 9 *Flowchart of monitor module*

process the output rate is fixed, i.e. the output decoded characters have the same length and will be generated to the same rate (one character per decoding cycle), but the input rate is variable. If the pre-processor receives the match of a pair (*offset*, *length*) this match will be decoded into a series of successive *length* characters.

It is not necessary for the pre-processor to receive the codeword from Frontend in every decoding cycle. When the Request signal goes high it indicates that the previous codeword has decoded completely and the buffer register should receive the next codeword being ready for decoding. The down counter takes the responsibility of informing the pre-processor itself to take the new data from the Frontend. The down counter latches the *length* value from DEMUX, sends the high level of Active signal for the *length* consecutive cycles when the Flag is equal to 1, and will decrease one at each positive trigger of ck timing clock. As soon as the down counter decreases to two it changes the state of the Request signal to high. The Buffer, Flag and End_signal registers will then receive the new data in the next decoding clock cycle. If the new Flag is equal to 0, the current codeword in the Buffer is just a character. In this case the Request signal must also be changed to high and all the registers will latch the new data in the next cycle, too.

Dataflow in the systolic array of the series decoding cell has been described in previous literature [11]. The main task of the decoding cell n is to compare the input *offset*

value with the stored *offset* values in cell n, where the stored *offset* values are $2n$ and $2n + 1$ when the cell number is n. If the input *offset* value is equivalent to one of the stored *offset* values $2n$ or $2n + 1$, the output decoded character will be copied from S_n register or S_{n+1} register, repectively, where S_{n+1} register comes from the preceding stage (cell $n + 1$). Otherwise, the input character is retransmitted to the next stage (cell $n - 1$). The original characters will be recovered without distortion by the systolic array of the series decoding cell. The decoding cell [11] is shown in Fig. 12.

5 Hardware synthesis and simulation and power analysis

The new encoding architecture is implemented by the 0.6μm standard cell library supported by the Compass Company in the USA. We utilise the synthesis tool from Synopsys to synthesise the Verilog RTL code to gate-level schematic. The hardware cost including the estimated equivalent wire load for each module and encoding cell were listed in Table 2. In this encoding system the buffer size of the encoding window is 512, and the total gate

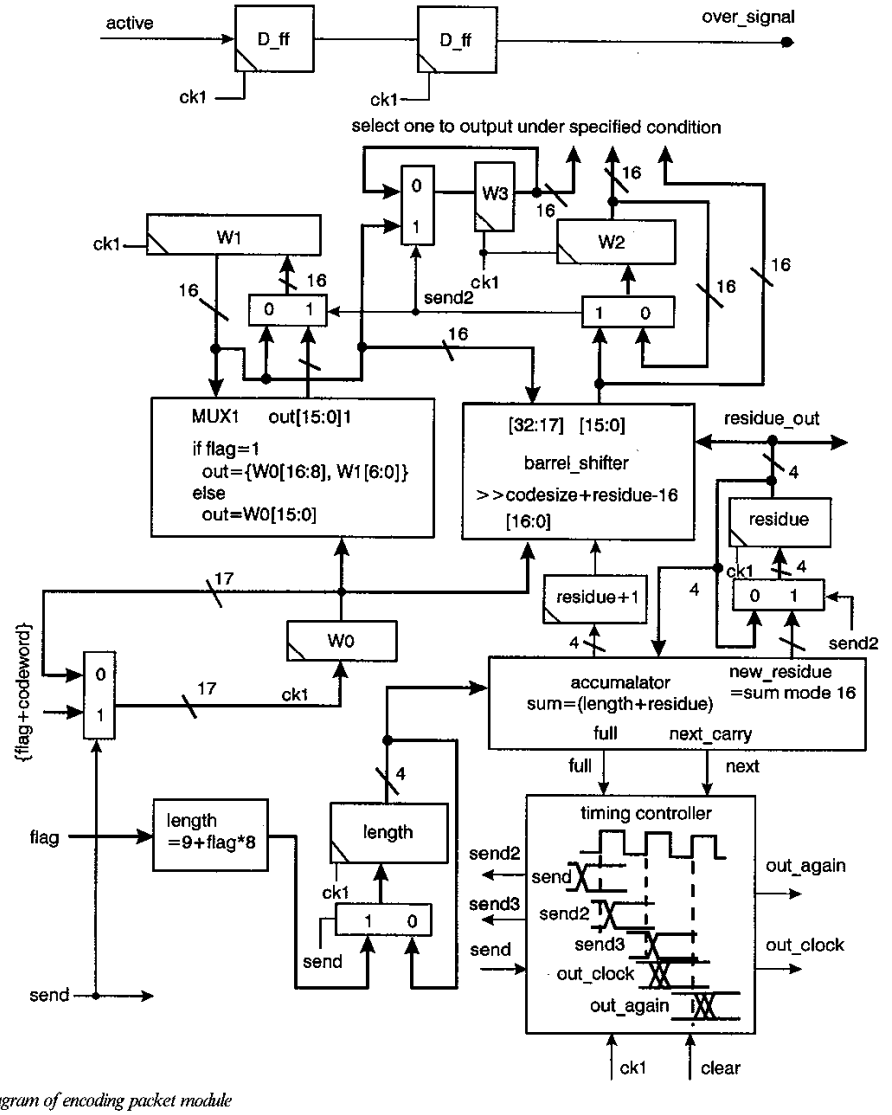

Fig. 10 *Circuit diagram of encoding packet module*

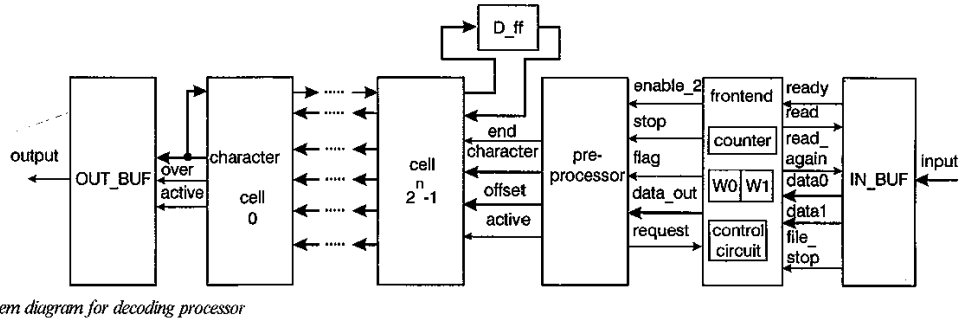

Fig. 11 *System diagram for decoding processor*

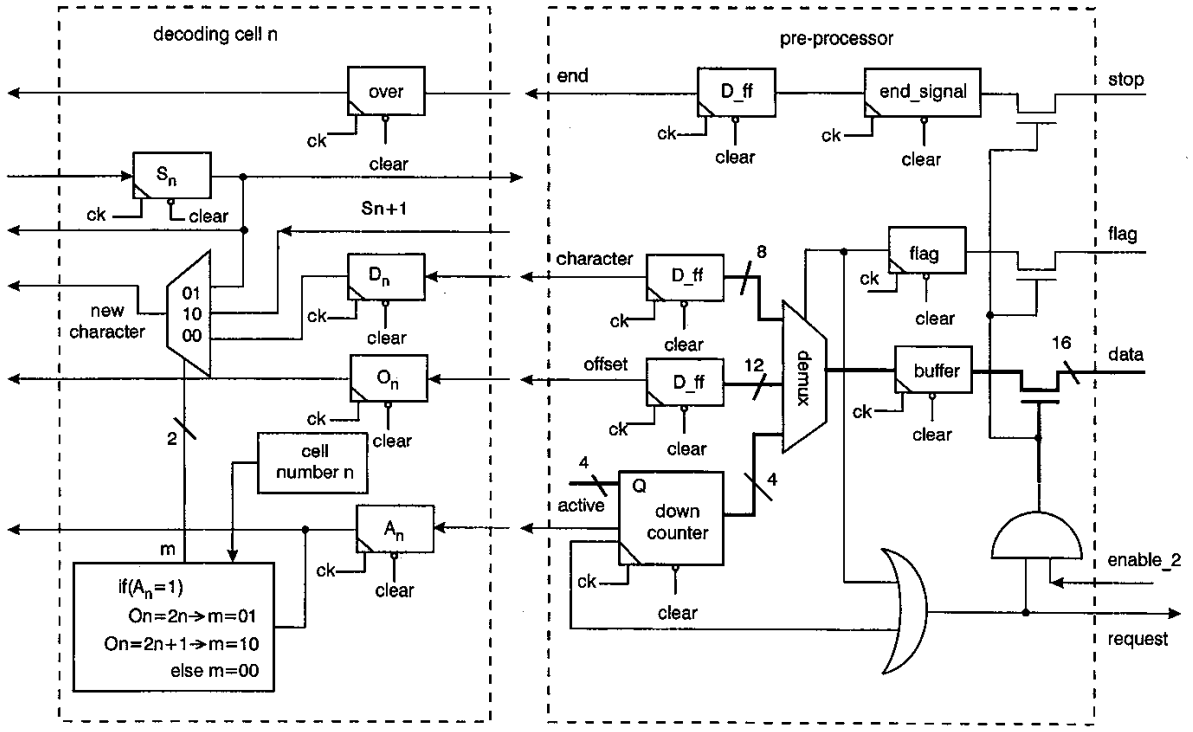

Fig. 12 *Circuit diagram of pre-processor and decoding cell for decoding*

Table 2: Equivalent gate count for each module

Module name	Cell	Vcell (64 Cells)	VLcell (256 Cells)	Receiver	Monitor	Packet	Sender
Gate Count (including wire load calculated by equivalent gate count)	1345	21771	88472	364.5	346	1376.7	275.12
Total	90472.8						

count, not including I/O pads, is about 90473 gates. The gate count is less than that in other architectures [8] with the same buffer size.

To verify our encoding system we compare the input strings of encoding processor with the output strings of the decoding processor whose block diagram is shown in Fig. 11. If there is no difference between them the modified architecture should be correct. We use some input files for verification by Verilog simulator [13]. From these simulation results we verify that our encoding and decoding circuits are correct. The hardware simulation results of the whole chip including I/O and power pads are shown in Fig. 13. The DATA of Fig. 13 is the data output consisting of flag and encoded codeword. The SEND signal is the handshaking signal to inform the receiver whether the encoding system transmits output data or not. The low level of the FINISH signal indicates that the encoding system has finished the entire encoding task. As shown in Fig. 13, the function of our encoding system can work correctly even when the clock cycle is about 11ns. The operating frequency is the reciprocal of the clock period so the maximum operating frequency can reach 91MHz and the compression bit rate is about 728Mbit/s. To reduce ground bounce noise for high-speed application we use ten pairs of power pads for the drivers of I/O pads, and six pairs of power pads for the core of digital circuit and predrivers of I/O pads. The total I/O pin count is 128.

For high speed-application power consumption is an important issue. The gate count of the registers contributes the major part of the hardware for most LZ-type compressors. In the Zito–Wolf system each register must access and latch new data in every clock cycle, and thus the dynamic power consumption is very large. As described in [14], we estimate the percentage activity of the circuit P_a and the total capacitance C_t driven by gate outputs in circuit. Thus, the estimated power consumption of the complex circuit for CMOS design can be estimated as follows:

$$P_d = P_a \times C_t \times V_{DD}^2 \times f_c \qquad (3)$$

where f_c is the clock frequency. From eqn. 3, minimising the percentage of the switching capacitance P_a can save power dissipation. In our design only one quarter of registers execute the data access operation for input characters, codes, and offsets as shown in Fig. 7 during one clock cycle. Therefore minimising the number of memory accesses will reduce the power dissipation for the sake of reducing P_a. The number of memory accesses for our encoding cell is about 7/19 of the Zito–Wolf architecture [11], so that the dynamic power dissipation can be lowered with the same throughput rate. Since our implementation is based on CMOS design, there is no DC path during the operating cycle. For the CAM implementation [7] the DC path exists in the circuit of the CAM cell such that the power dissipation of CAM architecture is larger than the CMOS design under the same throughput rate, clock fre-

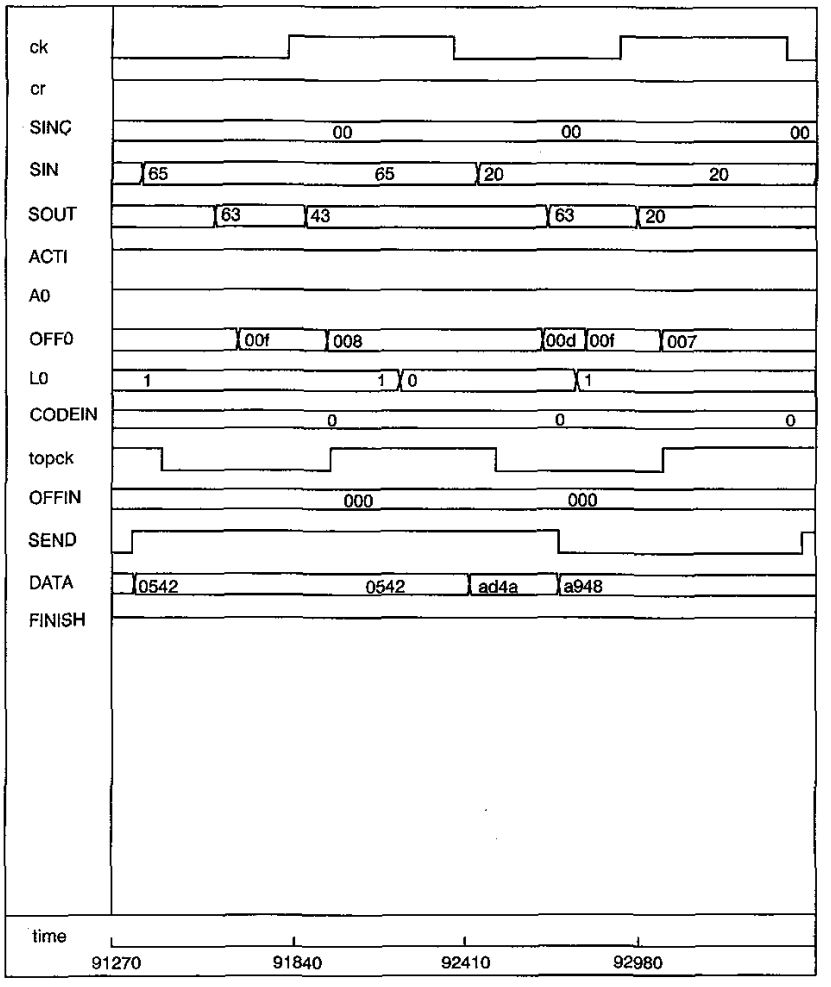

Fig. 13 *Whole chip simulation by Verilog simulator*

quency, and the same buffer size conditions. For high-speed applications the power dissipation of our architecture implemented by CMOS process is lower than those of the other implementations for the same compression ratio.

6 Conclusion

A parallel structure for a high-speed LZSS coder has been introduced. This parallel LZSS coder encodes each character in one clock cycle, and its operating frequency can reach 91MHz. This system has several advantages. First, the compression time is linearly proportional to the input length. Only a single clock cycle is required for processing one character, and the clock cycle is bounded by the critical delay of the encoder and independent of window size. Secondly, the architecture is simple and modularly expandable. In our design only 256 encoding cells are integrated in one chip at most because of the problems of power, clock distribution, and hardware complexity. We can cascade several chips to increase the dictionary buffer size and achieve the ideal desired compression ratio.

Although the hardware complexity of our modified architecture is higher than those of other architectures in [4, 6], the speed is much higher. Suppose the trigger cycle C_{trig} represents the number of time unit (clock cycle) between two initiations of a pipeline. A trigger cycle C_{trig} of k means that two initiations are separated by k clock cycles. Assume the clock period is T_c. The average compression speed bit rate of our architecture is $(1 * 8)/(C_{trig} * T_c)$, where C_{trig} is equal to one, and $T_c = Td_{total} + Td_{load\_effect}$. Assume M and N represent the maximum length of the matching string and buffer size, respectively. The *maximum* compression speed bit rate in [4] is $(M * 8)/(T_c * C_{trig})$, where C_{trig} is equal to $(N/2) + M$. If we want to achieve the ideal compression ratio, the encoding buffer in [4–6] is 512 at least. Our C_{trig} is much lower than that of [4, 6], so our speed is apparently higher for the same process technology. Thus, the new compression system is more suitable for real-time application to increase the bandwidth of a communication system and can also be used to effectively increase the amount of mass storage available to computer systems. By utilising VLSI technology to implement the system chip, the data compression hardware can be integrated into real-time systems so that data can be compressed and decompressed on-the-fly.

7 Acknowledgment

This work was supported by the National Science Council of Republic of China under grant NSC82-0404-E009-338.

8 References

1 ZIV, J., and LEMPEL, A.: 'A universal algorithm for sequential data compression', *IEEE Trans. Inf. Theory*, 1977, **IT–23**, pp. 337–343
2 ZIV, J., and LEMPEL, A.: 'Compression of individual sequence via variable-rate coding', *IEEE Trans. Inf. Theory*, 1978, **IT–24**, pp. 530–536

3 BELL, T.C.: 'Better OPM/L text compression', *IEEE Trans. Commun.*, 1986, **COM-34**, pp. 1176–1182
4 JUNG, B., and BURLESON, W.: 'A VLSI systolic array architecture for Lempel–Ziv-based data compression'. Proceedings of IEEE international symposium on *Circuit and systems*, June 1994, Vol. 3, pp. 65–68
5 STORER, J.A.: 'Data compression: methods and theory' (Computer Science Press, Rockville, Maryland, 1988)
6 RANGANATHAN, N., and HENRIQUES, S.: 'High-speed VLSI designs for Lemple–Ziv-based data compression', *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, 1993, **40**, (2), pp. 96–106
7 LEE, C.Y., and YANG, R.Y.: 'High-throughput data compassor design using content addressable memory', *IEE Proc. Circuits Devices Syst.*, 1995, **142**, (1), pp. 69–73
8 CHEN, C.T., and CHEN, L.G.: 'A novel architecture for Lempel-Ziv-based data compression'. Proceedings of ICCE international conference on *Consumer electronics*, 1996, pp. 210–211
9 STORER, J.A.: 'Image and text compression' (Kluwer Academic Publishers, Norwell, Massachusetts, 1992)
10 KUNG, S.Y.: 'VLSI array processors' (Prentice-Hall, Englewood Cliffs, NJ, 1988)
11 ZITO-WOLF, R.J.: 'A systolic architecture for sliding-window data compression'. Proceedings of IEEE Workshop on *VLSI signal processing*, Part 4, November 1990, pp. 339–351
12 LEI, S.M., and SUN, M.T.: 'An entropy coding system for digital HDTV application', *IEEE Trans. Circuit Syst. Video Technol.*, 1991, **1**, (1), pp. 147–155
13 Cadence Design System Inc., Verilog-XL manual, version 1.7, Sept. 1993
14 WESTE, N.H.E., and ESHRAGHIAN, K.: 'Principles of CMOS VLSI design: a perspective system' (Addison-Wesley Publishing Company, 1993, 2nd edn.), pp.236–238