On the Design and Implementation of a Lossless Data Compression and Decompression Chip

D. Mark Royals, Tassos Markas, *Member, IEEE*, Nick Kanopoulos, *Senior Member, IEEE*, John H. Reif, *Fellow, IEEE*, and James A. Storer, *Member, IEEE*

***Abstract*— A lossless data compression and decompression (LDCD) algorithm based on the notion of textual substitution has been implemented in silicon using a linear systolic array architecture. This algorithm employs a model where the encoder and decoder each have a finite amount of memory which is referred to as the dictionary. Compression is achieved by finding matches between the dictionary and the input data stream whereby a substitution is made in the data stream by an index referencing the corresponding dictionary entry. The LDCD system is built using 30 application-specific integrated circuits (ASIC's) each containing 126 identical processing elements (PE's) which perform both the encoding and decoding function at clock rates up to 20 MHz.**

I. Introduction

LOSSLESS data compression is the process of encoding a body of data into a smaller body that can be uniquely decoded at a later time back to the original data. Data that arise in practical applications typically contain redundancy of which compression algorithms can take advantage. Data compression can be applied in communication systems to increase the effective bandwidth of the channel, and in data storage systems to increase the effective capacity of the storage device. The processing speed of modern data processing systems such as disk controllers and data transmitters requires that data compression should be executed at very high speeds. To meet these stringent requirements, it is necessary to minimize the execution time of the compression algorithms. This can be only accomplished by defining parallel architectures capable of executing the defined data compression algorithms and implementing them in hardware systems. The most widely used class of data compression algorithms is the Lempel–Ziv method [1], [2]. The most computationally expensive step of this class of algorithms is the searching of the reconstruction vocabulary. During this step the compression processor has to search the entire vocabulary to identify any entries that match the input string. One of the primary methods that is used to reduce this high computational requirement is hashing [4]. A data compression chip set that utilizes hashing has been proposed in [5]. The disadvantage of such systems is that they require a two-level external memory to implement the hash tables. This implies that two memory accesses are required to find each key. In addition, the hash tables generally require large memories, which often experience slower access times. Because of these reasons the execution performance of such systems is rather limited. The reported speed in [5] is 7.5 MHz.

The textual substitution method that has been implemented in this architecture can be viewed as a practical realization of the universal data compression algorithms developed by Lempel and Ziv [1], [2]. These algorithms are used extensively in UNIX[1] systems (*compress* command) [3]. Textual substitution is a class of data compression algorithms capable of compressing text by identifying repeated substrings and replacing them by references to other copies defined in a dictionary D. The encoder repeatedly finds a match between the incoming characters of the input stream and the dictionary, deletes these characters from the input stream, transmits the index of the corresponding dictionary entry, and updates the dictionary with some method that depends on current contents of the dictionary and the input stream. A detailed description of the general class of textual substitution algorithms can be found in [6].

The basic idea of this method is to build a dictionary D to store a set of variable-length strings. The encoder (a finite state machine that compresses the data) repeatedly searches the incoming characters to identify a match with the dictionary entries based on a match heuristic (MH). This heuristic attempts to identify the longest possible match between the input stream and the dictionary entries to maximize the compression rate.

The encoder and the decoder maintain identical copies of the dictionary D which is augmented progressively by concatenating the previous match with new data. We will refer to this process as the update heuristic (UH). Finally, when there is no longer enough room to update the dictionary, an entry is deleted based on a delete heuristic (DH). Although, the least recently used (LRU) replacement policy is a good candidate for this purpose, it cannot be efficiently implemented in hardware. The SWAP heuristic [6], a simpler method with lower compression performance, can be used as an alternative solution.

II. Parallel Implementation

Parallelization of the textual substitution algorithms can be accomplished by simultaneously comparing the input data with multiple dictionary entries. This implies an associative

D. M. Royals, T. Markas, and N. Kanopoulos are with the Center for Systems Engineering, Research Triangle Institute, Research Triangle Park, NC 27709.

J. H. Reif is with the Department of Computer Science, Duke University, Durham, NC 27706.

J. A. Storer is with the Department of Computer Science, Brandeis University, Waltham, MA 02254.

[1] UNIX is a trademark of AT&T.

Preprint of paper appearing in IEEE Journal of Solid State Circuits (JSSC), Vol. 28, No. 9, pp. 948-953, Sep. 1993.

search between all vocabulary entries and the input data. This search can be efficiently implemented using content addressable memories. Although this scheme is very attractive in terms of implementation efficiency because of the high integration levels that we can achieve, it requires large data buffers to store data during the decompression phase. This is necessary since multiple decompressions of the same data token are possible. To eliminate the large buffering requirements, the searching process can be parallelized by distributing the reconstruction vocabulary in a number of processing elements, and having each element responsible for identifying entries within its local vocabulary. This fine-granularity parallelism can be efficiently exploited using systolic-type parallel architectures. The developed data compression system is based on this systolic approach, consisting of a number of identical processing elements capable of performing basic data compression operations, namely, comparison, multiplexing, and storing. The cells are connected in a linear pipe so that each cell communicates data and instructions with its immediate neighboring cells. The dictionary D is distributed among the processing elements so that each vocabulary entry is processed only by the corresponding element. The current implementation utilizes two vocabulary entries for each processing element.

During encoding, each cell compares two data tokens of the input string with the local vocabulary entries. If a match is found, the cell substitutes the two data tokens with a single token (the cell's index). The one token is substituted by the index and the other token is marked as invalid (loaded with a nilpointer). When both data tokens are valid and there is no match, the data are released to the next processing element one at a time. During decoding, each cell compares the incoming compressed data with its index. If a match is found, the cell replaces the index with the data tokens that are stored in the local entries while one of the data tokens is released to the next cell. A "stop" signal is also sent to the preceding cell in the pipe indicating that the current processing element is not ready to receive any new data. Tokens are either compressed/decompressed, or passed unmodified to the next cell. Compression is achieved in a progressive fashion, where a large number of processing elements can contribute to the compression of an input string. One interesting feature of this architecture is that no distinction is made between valid data tokens and valid index pointers. This feature enables pointers and other tokens to be further compressed by succeeding cells. This progressive compression of data builds larger matches from smaller matches and allows one to replace large character strings by only a few pointers.

This architecture offers high system throughput since comparisons for identifying matches between incoming data and dictionary strings can be executed in parallel for all the vocabulary entries. The implemented algorithm employs an on-line model, where compression is achieved using only current and previous data. This implies that neither the transmitter nor the receiver needs *a priori* the entire data volume to perform data compression or decompression operations.

The dictionary consists of variable-length strings and is built using combinations of ASCII characters and indices that point to previously defined strings. The implemented textual substitution algorithm utilizes the *identity* heuristic [7]: each cell uses the first two valid tokens as the new vocabulary entries that are stored in the local registers. Once a cell has learned a pair of tokens, it switches automatically to encoding mode. This training algorithm guarantees that encoder and decoder operate in lock-step fashion, which is necessary to maintain identical dictionaries in both ends of the communication channel or the storage media. Each pair of vocabulary entries is represented by a 12-b token which is the index of the corresponding cell. The size of the token limits the vocabulary to 4096 entries. The first 256 entries of the vocabulary are implied and they represent the ASCII character set. It has been found through simulation that a pipe of 3780 cells is adequate for compressing various types of text data

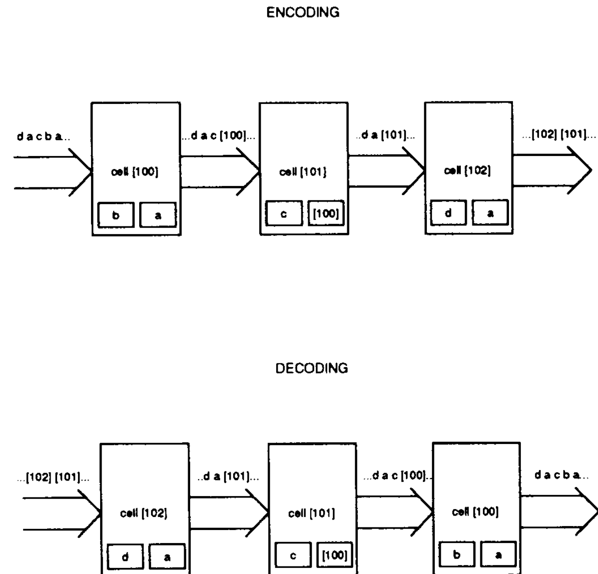


Fig. 1. LDCD compression.

The on-line model requires that the encoder and the decoder should learn their data in reverse order: strings compressed by the rightmost cell of the encoder pipe should be decompressed by the leftmost cell of the decoder pipe (Fig. 1). This implies that the dictionary entries and the index of the leftmost cell of the encoder should be identical with those of the rightmost cell of the decoder. Similarly, the rightmost cell of the encoder should be identical with the leftmost cell of the decoder. This symmetry can be very easily exploited in the described systolic architecture.

III. Additional Modes of Operation

Besides the encoding and decoding modes that were previously described, the lossless data compression and decompression (LDCD) system can also operate in "flush" mode, where the data can be pushed out of the pipe without destroying the existing vocabulary. This mode is very useful in cases where input data may become temporarily unavailable, in which case the already processed data can be pushed out of the pipe while

the existing vocabulary remains intact. To compress multiple files in a successive manner, a sequence of CLEAR instructions can be used to clear the vocabulary entries in each cell. In this case, the already processed data are pushed out of the pipe, while data following the CLEAR instruction are used to construct the vocabulary of the new file. One final mode of operation consists of a special built-in test function for the data-path logic. The TEST instruction is used to perform off-line testing of the data-path logic by shifting identical data to each data register and comparing them. Any mismatches among these data registers indicate a system failure.

IV. LDCD Chip Architecture

To implement the LDCD algorithm in silicon, a linear systolic array architecture consisting of identical processing elements is used to achieve high throughput. Each processing cell performs both the encode and decode operations on 12-b data operands and communicates control and data information only with its two neighboring cells. Due to the complexity of a single processing cell, only 126 cells can be placed on a chip given the chosen technology for implementation. System specifications required that the chip contains $2^n - 2$ cells such that the symmetry between the encoding and decoding indices was maintained and that sufficient addresses were available for nilpointers (which are nonvalid indices). Based on the developed architecture, the LDCD system will be composed of 30 VLSI ASIC chips implementing a total pipe length of 3780 processing elements together with the necessary controller logic, data buffers, and host interface logic.

The architecture of each LDCD processing element (PE) consists of a bit-slice-oriented data-path block and a control logic block. The data-path module, shown in Fig. 2, consists of four 12-b static registers which store the dictionary entries (*storeA*, *storeB*) as well as the input data tokens (*bufA*, *bufB*), two 12-b equality comparators, and several 12-b multiplexers.

During the encoding operation, data tokens are passes into the buffer registers *bufA* and *bufB* where they are compared to the previously stored dictionary entries found in *storeA* and *storeB*. If the comparators determine that no match occurs between the two register pairs, the data are passed unchanged through the PE to its successor cell. If a match does occur between these two register sets, the PE replaces the two stored data tokens (representing some character string) by a single pointer representing the index of the PE.

When compressed data are decoded, the incoming data pointers are placed in the *bufA* where they are compared to the PE's index. If a match occurs, the pointer is replaced by the two vocabulary entries that were previously "learned" by the PE. If no match is found, the data are passed unchanged through the PE.

Control of each PE is governed by its own local 4-b finite state machine, a 2-b register containing status information called *markA* and *markB*, four control signals from the two neighboring cells, and the two match signals from the data-path comparators as shown in Fig. 3. In addition to these, there are two nonsystolic global signals that affect the PE's operation. These two global signals are the *set* signal which

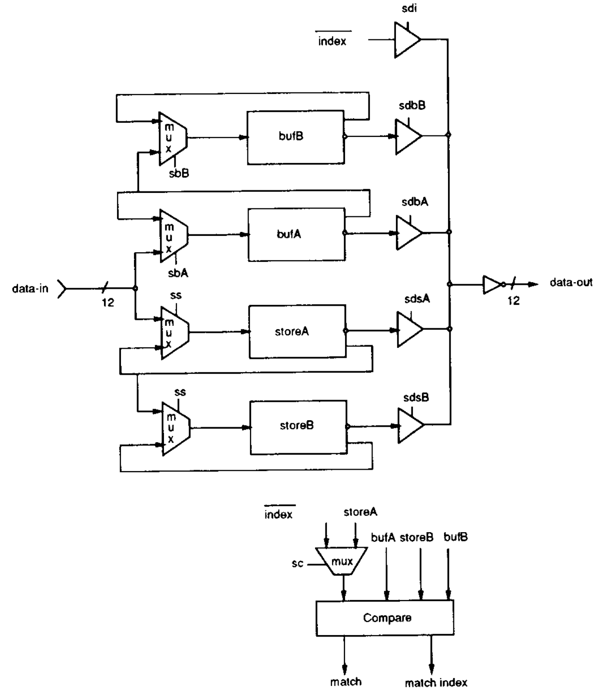


Fig. 2. LDCD data-path logic.

synchronously initializes all data and control registers on the chip, and the *hold* signal which simultaneously freezes all cells on the chip.

As shown in Fig. 3, a 4-b instruction called *state_in* is input to each PE from the preceding cell. Depending upon the current state of the cell and the control inputs, the state multiplexer is enabled to either accept the instruction directly into the state registers on the next clock edge or to accept a 4-b state instruction computed by one of the internal combinational logic blocks. The processor of each PE is a 16-state finite state machine. Of those states, six are reserved for encoding data, seven are required for decoding data, and one state is needed for the resetting of the processing element.

Two states in the finite state machine were used to implement a built-in self-test function on the data-path logic. In this self-test scheme, the user sends the TEST instruction along with a stream of input data to the LDCD array. This data is first stored into the *bufA* and *storeA* registers and is then passed to the *bufB* and *storeB* registers. By utilizing the comparators, it is possible to check if the data in these registers are an exact match. If a problem occurs in the data-path portion of a chip such that the comparator flags a mismatch in one of the processing cells, then that cell is instructed to output its index as the output data token along with the ERROR instruction. To propagate this error indication to the output, each subsequent processing cell is instructed to adopt the ERROR state and to propagate the index of the failed cell to its output at the next clock edge. In this manner, a fault in the data path of the array cannot only be detected by the system but can be isolated to the failed chip and even to the processing element

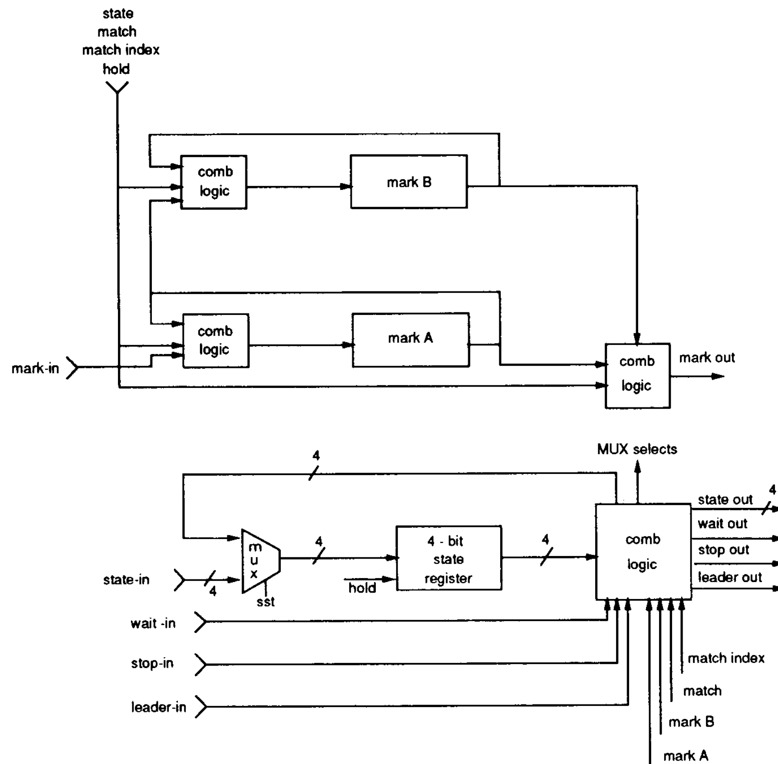


Fig. 3. LDCD control logic.

where the failure occurred. This method provides a convenient mechanism for detecting and isolating data-path failures in the LDCD system to the component responsible for the failure, thereby minimizing system down time and thus increasing overall system availability.

The combinational logic blocks are also responsible for generating an output instruction to the next cell (*state_out*), generating the four output control signals (*mark_out, wait_out, stop_out,* and *lead_out*), and creating the select signals for the various multiplexers which are present in each PE. Each cell also has a 12-b index which is divided into a 7-b cell address (which is hardwired internally) and a 5-b chip address which is provided externally through the *Index* pins. Although the 5-b chip index assignments are programmable, it is necessary to impose certain constraints in the chip addressing scheme when constructing the LDCD array in order to maintain the symmetry between the encode and decode modes of operation. A block diagram of the LDCD processing cell along with its I/O is shown in Fig. 4.

V. Chip Design and Implementation

In parallel to the design procedure for the control logic, the data-path logic was implemented using a bit-slice approach. Using custom layout for maximum area utilization, a 1-b slice was developed that contains an instance of each of the four data registers along with two comparator cells and the assorted multiplexer cells. Portions of this bit-slice cell were also used to create the state and mark registers. The 12-b data comparator was implemented as two 6-b ripple comparators operating in parallel in order to achieve the necessary circuit performance. Upon completing the layout of the basic processing element, a generator program was developed to place and route the individual cells to form the core logic. Due to the layout floorplan of the PE, most I/O connections between cells were made through simple abutment. At the same time, the 7-b index of each PE was hardwired to its appropriate address through the application of the generator code. In addition to the array of processing cells in the chip logic, high-drive buffers were developed and manually integrated to distribute the clock and other global signals to the internal array.

A prototype of the LDCD chip, containing 126 processing cells, was designed and fabricated. The chip was designed in the 1.0-μm n-well Hewlett-Packard CMOS process with double metal interconnects and was packaged in a 68-pin ceramic leadless chip carrier. Some important characteristics of the design are shown in Table I. Upon receiving the packaged prototypes from the foundry, the chips were tested using a set of 1820 fault-graded test vectors. Fault coverage for these vectors was found to be in excess of 94%. The built-in test function utilizing the TEST instruction provides 100% fault coverage for data-path faults which occupy 63.3% of the total device count. Since this instruction also tests a small portion of the control logic, this off-line test strategy can detect and isolate approximately 70% of the single stuck-at faults in the entire LDCD array.

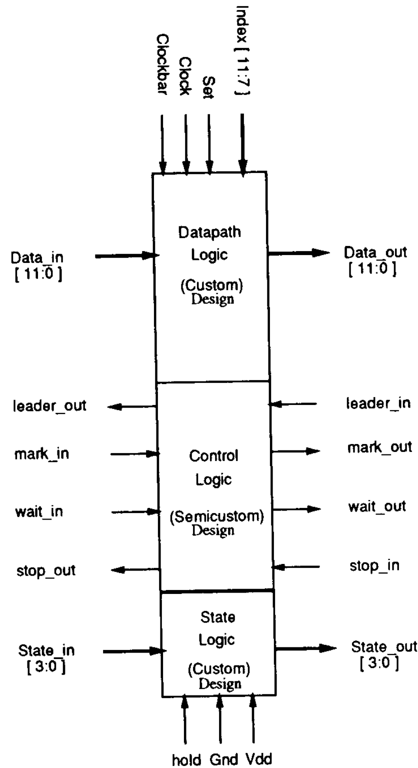


Fig. 4. Layout view of cell.

TABLE I
LDCD CHIP CHARACTERISTICS

	Full-scale Chip (126 cells)
Size	10.2mm x 9.65 mm
Transistors	> 367,000
Inputs	28
Outputs	20
Clock rate	20MHz*

*For a capacitive load of 20pF

After testing the prototypes on the tester, it was concluded that the functional chips were capable of operating at a maximum speed of 20 MHz. A microphotograph of the 126-cell chip is shown in Fig. 5. The total design effort from final specifications to the completed chip layout for the design was approximately eight months.

VI. CONCLUSIONS

A lossless data compression and decompression algorithm has been implemented in a system built around a complex VLSI chip containing 126 processing elements. The chip is technology aggressive in that it contains over 367 000 transistors and is capable of processing data at an input rate of 160 Mb/s. A board containing the ASIC chips, the control logic, and a VME bus interface has been developed at the Microelectronics Center of North Carolina. The LDCD system can be used in real-time applications to increase the bandwidth of communication systems and can also be used to effectively increase the amount of mass storage available to computer systems. During the design of the LDCD chip we have facilitated a design methodology that includes both custom and semicustom design styles. Custom design was used to maximize circuit density in highly repetitive logic blocks thus reducing the overall die size and maximizing system performance. Semicustom design, assisted by the available silicon compilation tools, was used during logic minimization and implementation of the control circuitry to maintain a quick design

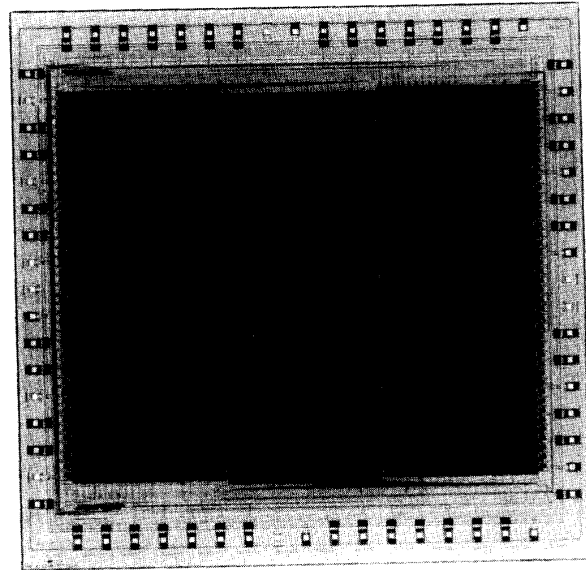


Fig. 5. Microphotograph of the 126-cell chip.

ACKNOWLEDGMENT

The authors would like to acknowledge the efforts of Dr. N. Vasanthavada and J. Dodrill in defining the LDCD architecture. In addition, we would like to thank D. Pantzartzis for his support in testing the LDCD prototype chips.

REFERENCES

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[2] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inform. Theory*, vol. 24, no. 5, pp. 530–536, 1978.

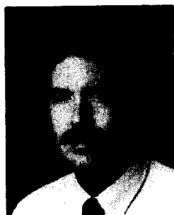
[3] T. A. Welch, "A technique for high-performance data compression," *Commun. Ass. Comput. Mach.*, pp. 8–19, June 1984.

[4] D. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching*. Reading, MA: Addison-Wesley, 1973, pp. 507–549.

[5] I. A. Shah, O. Akiwumi-Assani, and B. Johnson, "A chip set for lossless image compression," *IEEE J. Solid-State Circuits*, vol. 26, pp. 237–244, Mar. 1991.

[6] J. Storer, *Data Compression, Methods and Theory*. Rockville, MD: Computer Science, 1988.

[7] M. Gonzalez and J. A. Storer, "Parallel algorithms for data compression," *J. ACM*, vol. 32, no. 2, pp. 344–373.



D. Mark Royals received the B.S. and M.S. degrees in electrical engineering from North Carolina State University in 1985 and 1987, respectively.

In 1987 he joined the Center for Digital Systems Engineering at the Research Triangle Institute, Research Triangle Park, NC, where he is now the project leader for all VLSI-design related activities. Since joining RTI, he has participated in the development of several VLSI circuits including SRAM's, special-purpose data processors, and technology portable module generators. His research interests include VLSI technology and design automation, fault-tolerant circuit design methods, and design for testability and built-in self-test techniques.

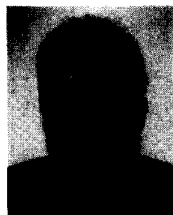
Tassos Markas (S'86–M'92) received the B.S. degree in physics from the University of Athens, Greece, in 1985, and the M.S. and Ph.D. degrees in electrical engineering from Duke University, Durham, NC, in 1987 and 1993, respectively.

He is a research engineer in the Center for Digital Systems Engineering at Research Triangle Institute, Research Triangle Park, NC, where he develops application-specific integrated circuits (ASIC's) and CAD tools for ASIC design and simulation. He is also involved in the area of data compression where he develops image coding algorithms, as well as hardware architectures for real-time applications. His main research interests include data compression, ASIC development, CAD tool development for ASIC design, fault-tolerant systems, and distributed applications.

Nick Kanopoulos (S'82–M'84–SM'92) received the Electrical Engineering Diploma degree from the University of Patras, Patras, Greece, in 1979 and the M.S. and Ph.D. degrees in electrical engineering from Duke University, Durham, NC, in 1980 and 1984, respectively.

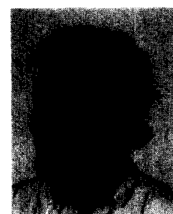
From 1980 to 1982 he was a Design Engineer with Bendix Avionics at the VLSI Design Center in Ft. Lauderdale, FL, where he designed full-custom integrated circuits for avionics applications. In 1984 he joined the Research Triangle Institute in Research Triangle Park, NC, where he is currently the Manager of the VLSI Design and Test Department which performs R&D work in the areas of ASIC design, design for testability, built-in self-test, on-line error detection, application-specific module design, automatic generation of technology-portable circuit macrofunctions, and data compression. He is also an Adjunct Associate Professor at Duke University. His current areas of research are high-speed circuit design, on-line fault detection, efficient implementation of data and image compression algorithms, and implementation of testable circuits using scan-based and BIST techniques.

Dr. Kanopoulos is a member of the organizing committee of the IEEE BIST Workshop, Co-Founder and Chairman of the Program Committee of the IEEE Rapid System Prototyping Workshop, and was a member of the Program Committees of the Government Microcircuit Applications Conference (1988 and 1989). He became a member of the Editorial Board of the IEEE TRANSACTIONS ON VLSI SYSTEMS in 1992 and IEEE TRANSACTIONS ON COMPUTERS in 1993. His publications include numerous papers, two book chapters, and the book *GaAs Digital Integrated Circuits: A Systems Perspective* (Prentice Hall, 1989). He is a member of Tau Beta Pi, Eta Kappa Nu, and the Technical Chamber of Greece.



John H. Reif (S'77–M'78–SM'92–F'93) received the B.S. degree in applied mathematics and computer science from Tufts University, Medford, MA, in 1973 and the M.S. and Ph.D. degrees in applied mathematics from Harvard University, Cambridge, MA, in 1975 and 1977, respectively.

He is currently a Professor in the Computer Science Department of Duke University, Durham, NC, where he works on the development and analysis of algorithms (particularly parallel algorithms) for various fundamental problems such as solution of sorting, graph problems, algebraic and numerical problems, and the like. He has implemented sophisticated algorithms on existing massively parallel machines for sorting, data compression, solution of sparse linear systems, and molecular dynamics. He is co-architect and co-inventor of the BLITZEN 128 processor chip, President of RSIC, Inc., which recently developed special purpose massively parallel hardware for very high rate lossless data compression, and has published more than 130 papers. He has served as a consultant to IBM, GTE Laboratories, Thinking Machines, Microsoft, NASA Goddard, MRJ of Park and Elmer, and the Microelectronic Center for North Carolina.



James A. Storer (S'76–M'79) received the B.S. degree in mathematics and computer science from Cornell University, Ithaca, NY, in 1975, and the M.S. and Ph.D. degrees in computer science from Princeton University, Princeton, NJ, in 1977 and 1979, respectively.

From 1979 to 1981 he was a researcher at Bell Laboratories, Murray Hill, NJ. In 1981 he accepted an appointment at Brandeis University, Waltham, MA, where he is currently Professor of Computer Science. His research interests include data compression, text and image processing, parallel computation, computational geometry, VLSI design and layout, machine learning, and algorithm design.

Dr. Storer is a member of the ACM and the IEEE Computer Society.