STRING STORAGE AND SEARCHING FOR DATA BASE APPLICATIONS: IMPLEMENTATION ON THE INDY BACKEND KERNEL

George P. Copeland
Tektronix, Inc.
Beaverton, Oregon 97077

ABSTRACT

User and hardware cost trends dictate that data base systems should provide more complete functionality, simplicity of use, and reliability by increasing the amount of hardware present in the system. These goals are accomplished with a simple hardware arrangement within a one-dimensional cellular storage system called INDY. The INDY backend kernel is intended as a powerful tool for implementing all data models. The INDY cellular storage array is intended to provide functionality that is difficult to implement efficiently using a conventional hardware arrangement. It allows a simple implementation of improved data independence at high speeds. INDY simultaneously satisfies the time windows of future hardware technologies and user requirements.

The importance of strings as a mechanism for defining abstract data types for data base languages is discussed in more detail in another paper. In that paper, a language called STRING is introduced which allows names of data objects to be semantically defined as variable-length strings and compared based on string pattern membership. This paper is concerned with the implementation of string storage and searching required by the STRING language. Implementation of higher level structures and searching requirements (such as sets, rows, tables and hierarchies) on the INDY kernel is treated elsewhere.

1 Introduction

One purpose of this article is to describe a simple hardware arrangement that implements data base requirements that are difficult to implement using a conventional hardware arrangement. An equally important goal is to explain the reasoning behind the design; therefore, these reasons, rather than simply the final system, are described. This method of presentation should not be interpreted as an attempt to prove that the hardware arrangement described here is optimal. A strong proof is not possible because the basic premises may change, and because all conceivable alternatives cannot be compared - especially those that do not yet exist. Instead, the goal is to improve communication and encourage more widespread criticism by displaying the design process itself in as much detail as possible. Otherwise, the issues remain private and generally acceptable solutions cannot be found.

Computer hardware costs at the component level are drastically decreasing with time. No other industry has experienced such a sustained and incredible reduction in cost per function. This improvement in productivity can make a significant contribution to overall international economic productivity only if the base of applications is broadened (Shephard 1977). Expanding applications requires several improvements in future systems. System functionality and user simplicity must be simultaneously increased. This will allow more application areas to be reached, because the costs and delays involved in putting systems to practical use are reduced. Also, system reliability must be increased to allow more complex systems to be applied to more critical functions. Reliability is an increasingly critical issue, since we are expecting the system to do more for us than ever before and, at the same time, we must depend on it more than ever before.

Unfortunately, providing more complete and easy-to-use functionality with a strictly software approach results in increased system complexity, which increases the system cost and development time and decreases reliability (Brooks 1975, many others too numerous to list). But implementing existing software techniques directly in hardware does not address the basic problem of total system complexity. Hardware has similar complexity limitations. The consideration of hardware flexibility in addition to software provides much more freedom to simplify the total system. If the basic hardware functions are closely matched to the basic operations of the user language, then software development is greatly simplified. Furthermore, if simple hardware arrangements can be found that directly implement these language operations, the overall system is simplified and made more efficient.

Data base systems can be improved by increasing data independence for the user. Data independence means that the way data is stored is independent of (or orthogonal to) the way that it can be used. A flexible way of storing and searching the data is required to do this. The following three capabilities increase data independence by increasing this flexibility for strings (Copeland 1978). A variable-length capability eliminates any decision or restriction by the user regarding length for existing or future data. Secondly, a single and flexible data type eliminates premature data type declaration by the user. Instead, data type can

be determined dynamically for each query. Thirdly, the capability to dynamically search any portion of the string removes the burden of deciding and restricting the degree of resolution required to partition data into separate strings. Improvements in data independence at the string level only are described in this paper. Storage and searching of higher level structures (such as rows, sets, tables, and hierarchies) are considered elsewhere (Copeland et al. 1973, Copeland 1974).

In Section 2, a language called STRING is described that provides the degree of data independence described previously. Section 3 argues that the functionality of STRING is best implemented with an exhaustive search. A simple yet efficient LSI-oriented hardware arrangement for realizing the exhaustive search is described in Sections 4, 5 and 6. Section 4 concerns STRING storage, while Sections 5 and 6 describe implementation of searching for STRING pattern membership.

2 The STRING language

A language called STRING, described by Copeland (1978), provides improved data independence for strings. Names of objects can be described as variable-length strings of characters, which are chosen to describe the objects in the most natural way. Detailed characteristics regarding string length, data type, and how data is partitioned into strings need not be determined prior to storing the data. The language then allows any string or portion of a string to be searched based on a powerful set of criteria. Multiple variable-length equality and inequality (based on alpha or numeric comparisons) patterns within a string can be specified as the search criteria. Also, logical combinations of string patterns (using the operators AND, OR and NOT) can be specified to any degree of complexity using parentheses to indicate precedence of the operations. STRING is intended as a simple extension to data languages (such as Query-By-Example, SEQUEL, etc.) whose purpose is to relate strings, treating strings as the most basic unit. The functionality of STRING is thus limited to the specification of the semantic integrity constraints of strings and complex pattern specification within strings.

Alpha and numeric inequality comparisons ($<, \leq, >, \geq$) have different definitions. (For example, $2<14$ is true when compared numerically, but $2>14$ is true when compared as alpha strings.) Both comparisons have their most significant character to the left in the string. However, alpha inequality requires left justification and numeric inequality requires right justification of the two strings being compared. Thus, the user must specify which portion of the stored string is to be searched as well as whether the search is based on an alpha or a numeric comparison.

The key advantage of the suggested approach is that the user is not forced to specify any of these data characteristics before the data is stored. Instead, he is provided with a way to specify whatever search criteria are desired at the time that the comparison is required. This approach increases the independence of the stored data from the way in which the data might be used.

STRING semantic definition allows the use of natural and familiar delimiters (such as commas, parentheses, hyphens, blank spaces, etc.) that partition the strings into variable-length substrings. For example, a person's name can be defined in a familiar format, such as [John Quincy Adams]. Blank spaces are used in this string definition to allow the variable-length first, middle and last names to be individually compared. STRING uses square brackets, when necessary or desirable, to indicate the beginning and end of a string or substring.

Two operators (¢ and $) are used to specify how the stored strings are partitioned for the search by allowing any portion of the strings to be ignored. To ignore a fixed number of characters, ¢ is used. To ignore an arbitrary number (zero or more) of characters, $ is used. To illustrate, suppose that the format for a person's name given previously is used. If only first and last names are known, then a search for pattern membership can be specified using the pattern [John $ Adams]. The symbol $ is used to ignore the variable-length middle name. The pattern specifying all names with the first letter of the last name equal to [A] is [$ $ A$]. The first two $ symbols are used here to ignore the first and middle names, and the last $ ignores the remainder of the variable-length last name. The pattern specifying all names with the third letter of the last name equal to [a] is [$ $ ¢¢a$]. The two ¢ symbols are used to ignore the first two characters of the last name, and the last $ ignores the remainder of the variable-length last name.

Suppose that the user desires to find all persons whose last name comes after [Smith] in an alpha comparison. The pattern [$ $ >A [Smith]] specifies that the portion of the stored string to be searched begins after the second blank space and ends at the end of the stored string. It also specifies with >A[Smith] that the portion of the stored string to be searched must contain a string that comes after [Smith] in an alpha comparison. Thus, equality patterns are used to delimit variable-length inequality comparisons. A numeric inequality comparison would be specified using N instead of A.

3. Comparison of data structures and access methods

This section compares the methods for storing and searching a set of strings. For simplicity, the various methods are categorized as direct access methods, the binary search, and the exhaustive search. These three basic methods are compared on the basis of the degree of generality, storage requirements, variable-length storage restrictions, and the computing energy needed to search and to edit. Table A summarizes the comparison.

Direct access methods compute the location of each string using a carefully chosen compaction function (such as hashing). The generality of this technique is limited to exact equality searches of the entire string. The strings must be stored at the computed location, and pointers are needed between each string and its position in a higher level structure, such as a row of a table. Usually strings are stored again in the structure itself.

Variable-length strings are very difficult to handle. Only one access is usually needed, although several are needed when the compaction function maps more than one string onto the same location. Editing requires maintenance of the ordering required by the compaction function, maintenance of the pointers, and the allocation and collection of storage.

Table A. Comparing Storage Structures and Access Methods.

	EXHAUSTIVE	BINARY	DIRECT ACCESS
DEGREE OF GENERALITY	EQUALITY; INEQUALITY; SEARCHES WITHIN STRING	EQUALITY; INEQUALITY	EQUALITY
STORAGE REQUIRED	DATA	DATA; REDUNDANT DATA; POINTERS	DATA; REDUNDANT DATA; POINTERS
VARIABLE-LENGTH STORAGE	SIMPLE	DIFFICULT	VERY DIFFICULT
COMPUTING ENERGY FOR SEARCHING	NUMBER OF STRINGS	LOG_2 OF NUMBER OF STRINGS	ONE OR MORE
COMPUTING ENERGY FOR EDITING	STORAGE MANAGEMENT	STORAGE MANAGEMENT; MAINTENANCE OF SORTED ORDERING AND POINTERS	STORAGE MANAGEMENT; MAINTENANCE OF ORDERING AND POINTERS

The binary search method stores the set of strings in either an alpha or numeric order. The search consists of an iterative procedure that converges on the desired string. The search is begun by going to the middle of the ordered list; an inequality comparison then eliminates half of the list. This is repeated on the remaining half list, reducing it again by half. This is continued until the search converges on the desired string. The generality of this technique is limited to either equality or inequality searches of the entire string, and the choice between alpha or numeric inequality must be made beforehand. The strings must be stored in the ordered list, and pointers are needed between each string and its corresponding higher level structure. Usually, the strings are stored again in the structure itself. Variable-length strings are difficult to handle. The number of accesses is approximately log base 2 of the number of strings. Editing requires sorting to maintain the ordering, maintenance of the pointers, and the allocation and collection of storage.

The exhaustive search method examines every string in the list. This technique provides maximum generality. Either alpha equality and inequality or numeric equality and inequality searches are possible not only on the string as a whole, but also on any portion within the string. Only the strings themselves need be stored. Redundant data and pointers are not required. Variable-length strings are relatively easy to handle. The penalty that must be paid for the exhaustive search is that all strings in the list must be searched, requiring a great deal of computing energy for searching. However, this does not imply that the exhaustive search is slower, since parallelism can be used to reduce search time. Also, the computing energy required for editing is greatly reduced, since this technique requires no special ordering of the strings, does not store redundant data, and pointers are not necessary.

From the above comparison, it is obvious that the exhaustive search offers a much higher degree of data independence for the user and reduces the complexity of storing, searching, and editing of variable-length data. It provides simplicity and generality. It increases data independence, since the criteria for searching is not restricted by predetermined assumptions about the data. Its only disadvantage is the large amount of hardware (computing power) required for fast searching. This combination of features best fits the user and hardware trends discussed earlier in this paper. For these reasons, implementation of the exhaustive search approach will be examined in some detail.

4 A one-dimensional cellular hardware structure for storage of variable-length strings.

Although strings and higher level structures for data base applications are highly variable in length, many technical and economic considerations dictate that hardware building blocks must be of fixed size and arranged in regular structures. A way to map variable-length data onto fixed-length hardware is needed. Lee (1962, Lee and Paull 1963) proposed a one-dimensional array of cells (Figure 1) as the basic hardware structure that allows strings to be stored and exhaustively searched in a simple way with no restriction on length. Lee's approach was to use very small cells, storing only one character per cell. The characters of a string were stored in contiguous cells, and a special character was used to indicate the beginning of the string. This approach provided extremely fast parallel searching but required a great deal of hardware logic per character.

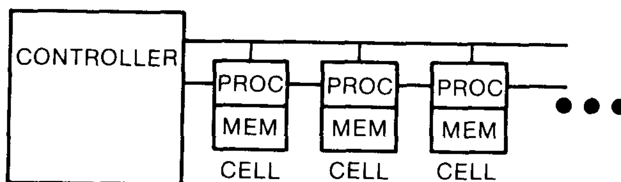

Figure 1. A one-dimensional cellular hardware structure for a backend data base system.

A more cost effective approach is to place a much larger memory in each cell, scanning the data exhaustively within the serial memory of each cell. This allows a tradeoff between cost and response time by varying the amount of memory per cell. That is, for a fixed total memory size, the number of processors can be varied to yield a large spectrum of response times.

The decision to use a regular arrangement of identical cells is based on three considerations. LSI and circuit board development and manufacturing costs are minimized if only a single generic chip is reproduced many times, and interconnected in a regular circuit board arrangement. Secondly, a cellular system opens up many techniques for dynamically recovering from a hardware failure. Thirdly, a cellular system allows modularity of expansion.

The decision to use a one-dimensional array is based on a number of considerations. A one-dimensional array requires fewer LSI pins per package, since the cell communicates with only two neighbors. Secondly, a one-dimensional array allows more cells per package without increasing the number of pins per package. Thus, improvements in lithography can be directly exploited without limit to reduce cost. Higher dimensional arrays require more pins per package as the number of cells per package increases. For a two-dimensional array, even though the total number of pins in the system varies as the inverse of the square root of the number of cells per package ($1/\sqrt{N}$), the number of pins per package increases as the square root of the number of cells per package ($\sqrt{N}$). Thirdly, a one-dimensional array can be much more highly utilized than higher dimensional structures. This was found to be true on the ILLIAC IV (Kuck 1968), where the one-dimensional hardware arrangement is usually chosen over a two-dimensional one. Only one restraint instead of two is required to store the data. Fourthly, a one-dimensional structure allows a much simpler scheme for the allocation and collection of storage than either a tree or a higher-dimensional structure. Finally, highly variable-length data structures are not easily mapped into a fixed-size hardware tree or higher dimensional structure. However, trees, sets, and tables are easily linearized in a number of ways. (Copeland et al. 1973.)

The decision that a cell should consist of a memory segment with a dedicated processor is based on two reasons. Access of multi-memories by multi processors reduces utilization of both processors and memories due to interference unless a one-to-one correspondence always exists. Cellularization of an associative memory allows this one-to-one correspondence between processors and memories. Secondly, this one-to-one correspondence can be fixed in hardware, so that the communication network between memories and processors is reduced to a fixed routing with a minimum of communication links.

The decision to use memories with block serial access (such as charge coupled devices, magnetic bubbles and disc technologies) is based on three reasons. Low cost per bit is important to avoid excessive moving of data between this memory system and a cheaper one. Secondly, an exhaustive search can be efficiently implemented on those memories. Such memories have gained a reputation of having long access times because they have been used to emulate a random access memory. However, when searching exhaustively, block serial access is as fast as random access. Only the bit rate and the number of points of simultaneous access are important. Thirdly, storage management is more efficient in these block-organized serially-circulating memories than in random access memories (Copeland et al 1973, Copeland 1974, Otis and Copeland 1978), because of their dynamic nature.

Storage of variable-length strings and searching for patterns within the strings using the storage organization previously described were first proposed by Healey et al (1972). Following Lee's approach, strings of characters were stored with no restrictions on length, using a special symbol to indicate the beginning of each string. All strings were packed tightly together, forming one long character string. This long string was stored on the one-dimensional array of cells by dividing the string into equal-length segments which correspond to the length of the serial memory within each cell. For example, a list of names would be stored as:

Cell 1: ØJohn Q. S
Cell 2: mithØLydia
Cell 3: M. Gentry
Cell 4: ØDonald H.
Cell 5: MadisonØV
etc.,

where Ø is used to indicate the beginning of each variable-length string and each cell can store ten characters. Allowing strings to overlap between cells at any arbitrary point within the string removes all restrictions on length and simplifies the storage management required by editing.

5 Marking variable-length strings

A small random access memory is shown to provide a method of remembering whether the search within each string is successful. Then a rail is shown to provide the communication within a string that spans several cells. It is shown elsewhere (Su et al. 1973, Copeland et al. 1973, Copeland and Su 1974, Copeland 1974) how this hardware arrangement can be used to mark between items within higher level structures (such as rows, sets, tables, and hierarchies). This method of marking or tagging data in-place is much more efficient than moving large amounts of data into intermediate tables.

5.1 Small RAM for marking strings

Strings of arbitrary length cannot be searched and output during the same circulation of a serial memory. It is not known whether the string qualifies for output until after the beginning of the string has passed. Also, the string may span several cells. Thus, either their position must be remembered or they must be marked for further use. Copeland et al.(1973) proposed using the small, one bit wide random access memory (RAM) and the COUNTER of Figure 2 to mark strings and higher level structures. For each string, this RAM provides one mark bit that can be accessed independently of the position of the circulating memory. The COUNTER, initially set to zero at the beginning of each circulation, is incremented each time the beginning of a string (Ø) is scanned. It acts as a hardware pointer to the RAM, providing the necessary mapping between the variable-length string and the fixed-length RAM. When the search within the string is completed, the COUNTER will contain the RAM address of the mark bit for that string, allowing the marking to take place.

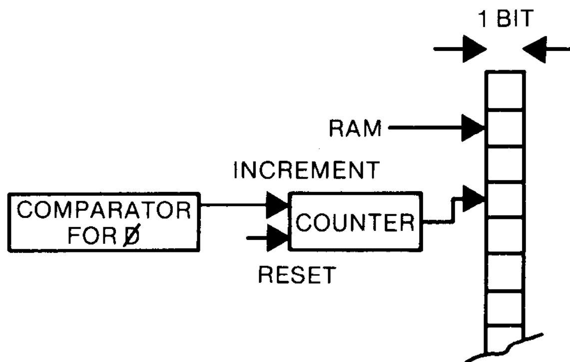

Figure 2. Hardware for marking variable-length items.

More than one mark bit per string is needed for implementing logical combinations of patterns and for other reasons not explicitly considered here. These additional mark bits can be stored within their corresponding string in the cheaper circulating memory, rather than providing several RAM bits per string. It is most convenient to locate these storage mark bits at the beginning of the string (for example, within the ∅ character of the item), since strings are accessed sequentially and since a string's RAM bit is always located in the same cell as the beginning of the string. The storage mark bits can then be marked by first marking the RAM bit as described above, and then transferring the RAM bit into the specified storage mark bit during the next circulation. This two circulation method of marking storage bits can, in effect, be done in one circulation (Copeland 1974) using pipelining within the cell processor. Using this technique, the cell processor is partitioned into several pipeline modules as shown in Figure 3. The serial data stream passes through the first module, then the second, etc. The transfer of RAM bits to storage bits is done in the second circulation in the pipeline module labeled MARK that encounters the serial data stream before any other pipeline module. This pipeline technique allows many operations to be done in one circulation that would otherwise require two circulations.

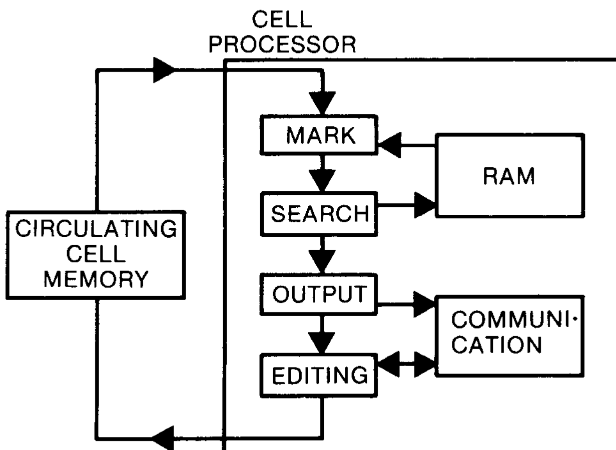

Figure 3. A Cell.

Searching and marking for logical combinations of patterns (as well as other functions) can be most conveniently implemented by arranging these storage mark bits as a stack of bits (Copeland 1974), rather than using a set of mark bits that are individually addressed. This issue is analogous to the stack versus general register issue in von Neumann processors. Marking for logical combinations of patterns can be accomplished by searching for each pattern in separate circulations. The first pattern is searched and a storage bit is marked as described above. Then the next pattern is searched and ANDed or ORed onto the top of the stack. Arbitrarily complex logical combinations of patterns can be searched and marked using this technique, using AND, OR, and NOT as the logical operators.

5.2 Rail communication within a string

The RAM bit used to mark a string lies in the same cell as the beginning symbol ∅ of the string. For strings that span cell boundaries, a search within a portion of the string may take place in a different cell. A communication is needed between these two cells. A single bit, indicating the outcome of the search, must be communicated upward to the cell with the string's RAM bit. Also, an indefinite number of cells may lie in between. An asynchronous rail communication, similar to that proposed by Lee and Paull (1963), is ideal for this. Figure 4 shows the simple hardware arrangement. The DESTINATION COMPARATOR looks for any occurrence of ∅ in the data stream. The SOURCE COMPARATOR examines the data for the pattern specified by the user. The final status of the SOURCE and DESTINATION FLAGS at the end of the serial scan is then used to initiate or terminate the communication. At the end of the communication, the ENABLE FLAG indicates the binary value that was communicated. The communication within each of the overlapped strings takes place independently but during the same short time interval. The final value of the COUNTER in Figure 2 contains the RAM address of the mark bit for the string. Thus, when the communication reaches its destination, the COUNTER can be used to mark the string.

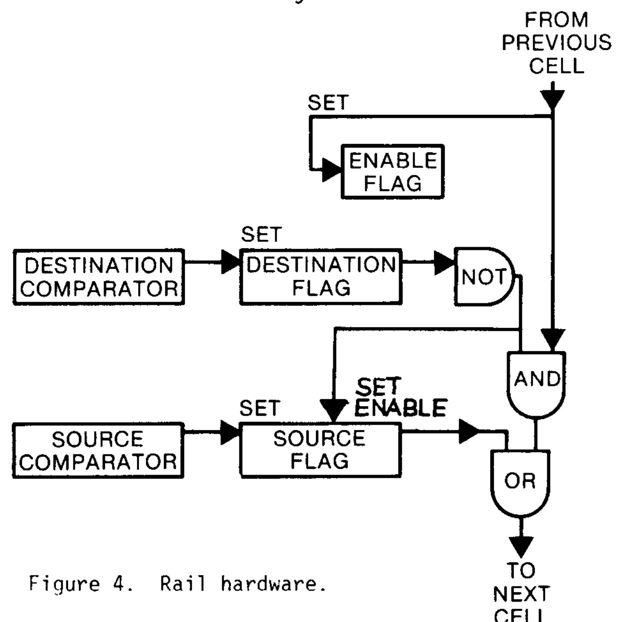

Figure 4. Rail hardware.

6 Searching the variable-length patterns within variable-length strings

A method for storing and marking variable-length strings has already been described. This section shows how a simple hardware arrangement provides a generalized algorithm for finding whether each of the stored strings contains any specified pattern of equalities and inequalities. The search can be based on alpha or numeric comparison. Furthermore, these patterns can be of arbitrary length and logical complexity. Thus, the user need concern himself only with the semantic constraints of his data, and is freed of implementation constraints.

This section begins with a description of how simple patterns are searched. Then the hardware and algorithms are generalized to implement more involved patterns. A very simple reduction algorithm is described which allows implementation of arbitrarily complex patterns expressed in STRING with multiple use of a single instruction.

6.1 Single variable-length equality patterns

Sequentially searching for an equality pattern within a string requires that a search for the pattern must begin with each character of the string, since left alignment of the two strings is not known. Healey et al. (1972) first proposed a method of doing this on the variable-length storage structure previously shown. A method of marking individual characters is required to remember intermediate results. To accomplish this, an additional bit per character could be stored with each character.

Healey's method of searching for patterns within variable-length strings is limited to a search for only one pattern character per circulation. A simple hardware arrangement (Copeland 1974) can search a larger number of pattern characters in one circulation. The maximum number of pattern characters searched per circulation can be chosen to be any number less than the number of characters in the cell's serial memory. For a system with many thousands of characters per cell, a realistic number would correspond more closely to the typical size of a pattern (for example, sixteen). It is shown later how this basic fixed-length hardware comparator arrangement can search for arbitrarily long patterns by using several circulations.

6.1.1 Searching strings contained within a cell

Figure 5 shows the arrangement for four search characters per circulation. The width or parallelism within each of the comparators is determined by the relative data rate of the comparator technology and the data stream. For a data stream rate greater than the comparator rate, parallelism within each comparator can be used to match the data stream rate. For a data stream rate less than the comparator rate, several memory streams can be multiplexed to match the comparator speed capability. Each EQUALITY COMPARATOR has a one character comparand register, which contains a pattern character. The X array of bits is used to propagate a successful search to the end of the array. Figure 6 shows the changing bit patterns for the X array during a search for the comparand pattern [$ABAB$] within the string [ABABABAD]. The $ symbols indicate here that neither left nor right justification is required. The end of the X array (X4) indicates a successful search and allows marking of either the individual data characters or the entire string (Ð). Two occurrences of the pattern ABAB are found in this example. Left and/or right justification of the pattern to the stored string can optionally be implemented by searching for the special delimiter symbol Ð to indicate the beginning or end of the stored string. The output of this comparator is convenient for several control functions (the COUNTER of Figure 2, the FLAGS of Figure 4, the ENABLE input of Figure 5, and others).

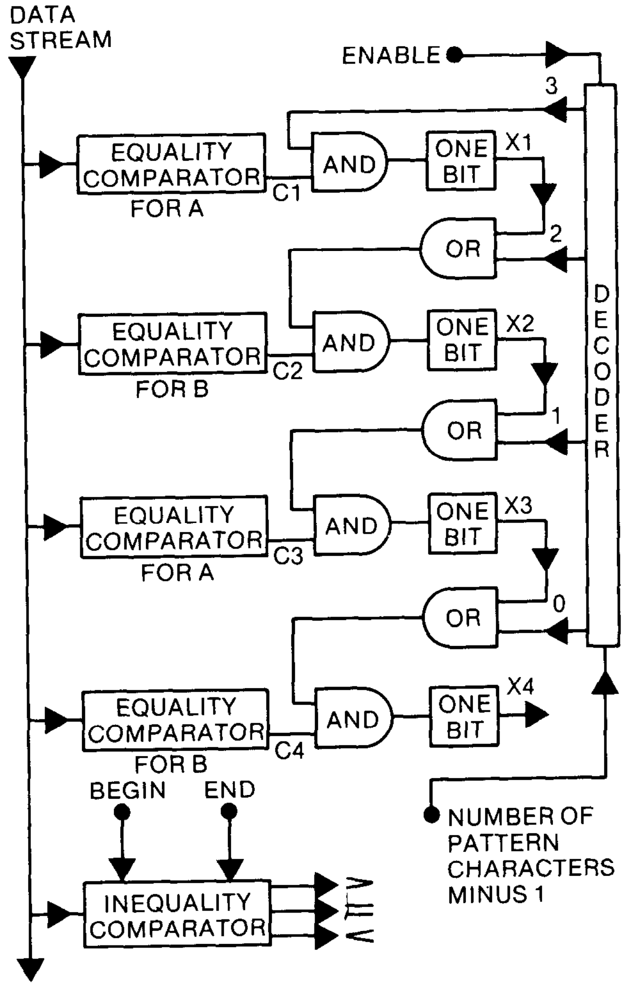

Figure 5. Hardware array for searching variable-length patterns within variable-length items.

SCAN OF DATA STREAM →

		A	B	A	B	A	B	A	D
A	X1	1	0	1	0	1	0	1	0
B	X2	0	1	0	1	0	1	0	0
A	X3	0	0	1	0	1	0	1	0
B	X4	0	0	0	1	0	1	0	0

Figure 6. Bit sequence for items contained within a cell.

A mask register with one bit for each EQUALITY COMPARATOR of Figure 5 (that is, one bit for each comparand character) allows the ¢ symbol to be implemented directly. Setting a mask bit indicates that the corresponding EQUALITY COMPARATOR successfully compares to any stored data character. A bit level mask register, which has one bit for each bit in the comparand register, allows individual bits to be ignored. This capability can provide more complete functionality in two ways. First, the ability to ignore case differences when searching becomes simple to implement, since most character sets (ASCII, EBCDIC, and others) differ in upper-case and lower-case by only one bit. Secondly, the complete functionality of the STRING language can be applied to single bit characters (binary data) as well as for larger character sets. Thus, the resolution for searching within a string can be brought down to the bit level.

For an equality pattern that is shorter than the comparator array, the pattern string is stored justified to the bottom of the array. The DECODER of Figure 4 is then used to apply the ENABLE control line at the point where the pattern begins. This effectively trims off the excess hardware from the top of the X array. For an equality pattern that is longer than the comparator array, two or more circulations are required and a method for marking characters is needed. To illustrate, assume that the length of the comparator array of Figure 4 is three characters. The pattern

[$ABCDEFG$]

could be implemented in three circulations by marking in the forward direction using the special bit F̸ to indicate that the character has been marked:

IF [ABC] THEN mark F̸ bit of C
IF [$F̸DEF$] THEN mark F̸ bit of F
IF [$F̸G$] THEN mark D̸ of string.

The X4 bit of Figure 4 initiates marking the last character of the pattern in the first two steps and the marking of the entire string in the last step. The previous character marking is removed before marking again. The equality pattern can be implemented backwards in three circulations using the special bit B̸ to indicate that the character has been marked:

IF [EFG] THEN mark B̸ bit of E
IF [$BCDB̸$] THEN mark B̸ bit of B
IF [$AB̸$] THEN mark D̸ of string.

Backward marking requires a delay or buffering in the processor of the number of characters searched per circulation in this direction. This delay is needed to mark the beginning of pattern occurrences within the string.

6.1.2 Searching strings that span several cells

The additional hardware of Figure 7 is required to search and mark pattern occurrences that overlap across a cell boundary in one circulation. The Y array (2N-3 bits long, where N is the length of the X array) is used to propagate and store a successful search for all possible overlapped substrings. For example, an occurrence of the pattern [ABAB] may overlap in three (N-1) ways. Either [BAB], [AB] or [B] could overlap into the next cell. Figure 8 shows the bit sequence in the X array of cell i and the Y array of cell i+1 during the circulation. Between circulations a communication is made between adjacent cells. For marking forward (F̸), the first N-1 bits of the X array of cell i are shifted into the X array of cell i+1 for all cells simultaneously. Then the X and Y arrays are logically combined to determine whether each of the three possible positions of the occurrence of the pattern is found. This is remembered in the M array (N-1 bits long) so that the characters at the end of the pattern occurrence can be marked (with F̸) in the next circulation. The time required for this communication is independent of the number of cells in the system, since all adjacent cells communicate concurrently.

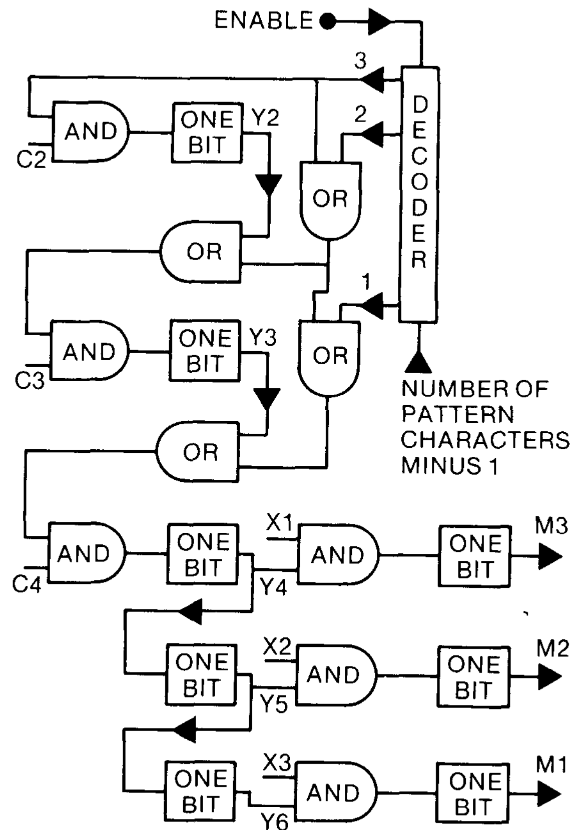

Figure 7. Additional hardware for pattern occurrences that overlap between cells.

For marking backward, the last N-1 bits of the Y array of cell i+1 are shifted into the Y array of cell i for all cells. The M array is determined in the same way but is used to mark (with B̸) the characters at the beginning of the pattern occurrence during the next circulation. A simple pipeline technique (see MARK unit in Figure 3) can be used to make this marking overlap with the operations required by the next instruction, so that effectively only one circulation is required.

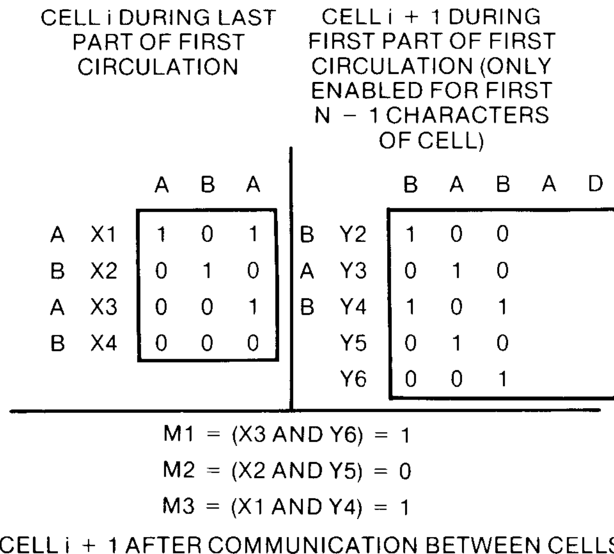

Figure 8. Bit sequence with overlap between cells i and i+1.

6.2 Multiple variable-length equalities

A pattern having more than one equality string separated by an arbitrary number of characters is specified using the $ symbol imbedded between the strings. For example,

[$ABCD$XYZ$LMNO$]

specifies any string containing the three substrings in sequence but having an arbitrary number of characters (zero or more) between them. This can be implemented with a minimum of additional hardware if the search is broken down into three searches for single variable-length equalities:

IF [$ABCD$] THEN mark F̸ bit of D
IF [$F̸$XYZ$] THEN mark F̸ bit of Z
IF [$F̸$LMNO$] THEN mark Ø of string.

Between these steps, a rail communication is required to enforce the specified sequencing of the substrings. The hardware in Figure 4 is used to enable all cells between the cell containing the end of the previous substring and the cell containing the end of the entire string. The DESTINATION COMPARATOR of Figure 4 would look for F̸ (indicating the end of the previous substring) and the SOURCE COMPARATOR would look for Ø (indicating the end of the entire string). This communication would take place for all items simultaneously. The ENABLE FLAG of Figure 4 would then be used to tell each cell whether to begin the next circulation looking for the next substring.

For searching backwards the three steps would be:

IF [$LMNO$] THEN mark B̸ bit of L
IF [XYZB̸$] THEN mark B̸ bit of X
IF [$ABCD$B̸$] THEN mark Ø of string.

The procedure for enabling cells is identical, except that the DESTINATION COMPARATOR would look for the beginning of the entire string (Ø) and the SOURCE COMPARATOR would look for the beginning of the last substring (B̸).

6.3 Single bounded-length inequality within multiple variable-length equalities

When searching for an alpha or numeric inequality pattern, the beginning and end of the portion of the stored string to be searched must be specified. The pattern

[$ABCD$XYZ<N[123]LMNOPQR]

specifies a numeric inequality search for <N[123] which is partitioned on the left by the equality pattern [$ABCD$XYZ] and on the right by the equality pattern [LMNOPQR]. This can be implemented with a minimum of additional hardware by breaking the search down into three simpler searches:

IF [$ABCD$XYZ$] THEN mark F̸ bit of Z
IF [$F̸LMNO$PQR$] THEN mark B̸ bit of L
IF [$F̸<N[123]B̸$] THEN mark Ø of string

First the left and right boundaries are marked in each string using the same algorithm described previously, except that the character markings (F̸ and B̸ bits) are left in place until the inequality search is completed. Comparators for these character markings are used to control the INEQUALITY COMPARATOR of Figure 5.

The same comparand register can be used for both the EQUALITY and INEQUALITY COMPARATORS.

Alpha and numeric inequality comparisons have different definitions. Table B describes these differences. It assumes that the INEQUALITY COMPARATOR is started when the beginning of the data is reached, and stopped when the end of either the data or the comparand is reached.

For strings that overlap across cell boundaries, a communication is required between each pair of adjacent cells to provide the alignment needed to control the INEQUALITY COMPARATOR. Otherwise, the comparator would not know which comparand character to start with at the beginning of the circulation. A small counter in each cell (reset whenever either Ø or F̸ if found) can be used to count the number of characters starting from the beginning of the partitioned data field. Between circulations, the counter of each cell i is shifted into the counter of cell i+1. The counter then determines which comparand character is used to start the search at the beginning of the next circulation. If this procedure is always done for each circulation, then an extra circulation is not required for inequality comparisons. During the circulation, the complete comparison is made for all strings contained within a cell, and the partial comparisons are made for strings that overlap between cells. Between circulations, a bit of information concerning the success of the partial comparison in each cell i is communicated to cell i+1. The complete comparison can then be determined from the partial comparisons:

GT=GTi OR (EQi AND GTi+1)
LT=LTi OR (EQiAND LTi+1).

6.4 Single variable-length inequality within multiple variable-length equalities

Inequalities that are longer than the comparand register can be implemented with logical combinations of patterns. For example, for a comparand register of length three, the pattern

[$ABCD>N[12345678]XYZ$]

can be implemented by

[$ABCD>[123]¢¢¢¢¢XYZ$]
OR [$ABCD123>N[456]¢¢XYZ$]
OR [$ABCD123456>N[78]XYZ$].

The ¢ symbol is used here as an alignment tool, allowing the rightmost characters to be ignored. If < was specified, the implementation would be identical, except that < would be substituted for >. If equality is also specified (for example, ≤ or ≥), then the pattern

[$ABCD12345678XYZ$]

is logically ORed with the above patterns.

6.5 Multiple variable-length inequalities within multiple variable-length equalities

Multiple inequalities can be implemented using logical combinations of single inequalities. For example, the pattern

[$ABC>A[HI]XYZ$LMNO>N[1234]RST$]

can be implemented by

[$ABC>A[HI]XYZ$LMNORST]
AND [ABCXYZ$LMNO>N[1234]RST$].

The $ symbol is used here to eliminate all but one inequality, so that each can be searched separately and the results logically ANDed together.

7 Conclusion

The implementation described here simultaneously satisfies two time windows in the near future. The timing for what the user will need and for what technology is expected to provide are both satisfied for the same time interval.

User requirements call for improved data independence so that applications can be addressed with a minimum of user cost and delay. They call for simplicity of implementation so that important applications can be addressed with adequate reliability. They call for a way to satisfy different cost and response times for various applications. The exhaustive search provides the increased functionality and generality needed to improve data independence. The one-dimensional array of identical cells provides a way of implementing the exhaustive search with simplicity and allows cost and response time to be traded to match various user environments. The close match between the data language and the basic hardware operations allows compilers or interpreters to be implemented with simplicity and reliability.

Memory technologies having serial access within blocks of storage (CCD's, magnetic bubbles and discs) are exploited. These low cost technologies have traditionally been classified as having slow access because of their serial nature. However, this is true only when such devices are used to emulate a random access memory. Their serial nature is exploited when an exhaustive search is required. In addition, the exhaustive search greatly improves storage utilization of these devices, since duplicated data and pointers are not required to implement the search.

Table B. Differences in alpha and numeric inequality comparisons.

	APPLIES TO ALL NUMERIC COMPARISONS AND APPLIES TO ALPHA COMPARISONS WHEN RESULT OF INEQUALITY COMPARATOR IS =	APPLIES TO ALPHA COMPARISONS WHEN RESULT OF INEQUALITY COMPARATOR IS ≠
DATA SHORTER THAN COMPARAND	DATA < COMPARAND	RESULTS
DATA LONGER THAN COMPARAND	DATA > COMPARAND	OF INEQUALITY
EQUAL IN LENGTH	COMPARATOR APPLIES DIRECTLY	

Acknowledgments: Thanks to Allen Otis for his helpful discussions.

REFERENCES

F. P. Brooks, The Mythical Man-Month, Addison-Wesley, Reading, Mass. (1975).

G. P. Copeland, G. J. Lipovski and S. Y. W. Su, "The Architecture of CASSM: A Cellular System for Non-numeric Processing," Proceedings of the First Annual Symposium on Computer Architecture (December 1973).

G. P. Copeland and S. Y. W. Su, "A High Level Data Sublanguage for a Context-addressed Segment-sequential Memory," Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control (May 1974).

G. P. Copeland, "A Cellular System for Non-numeric Processing," Ph.d. Dissertation, University of Florida (1974).

G. P. Copeland, "The Importance of Strings for Data Base Abstractions," in preparation (1978).

L. D. Healey, K. L. Doty and G. J. Lipovski, "The Architecture of a Context Addressed Segment Sequential Storage," Proceedings of Fall Joint Computer Conference, Vol. 41, Part I (1972).

D. J. Kuck, "ILLIAC IV Software and Application Programming," IEEE Transactions on Computers, Vol. C-17, No. 8 (August 1968).

C. Y. Lee, "Intercommunicating Cells, Basis for a Distributed Logic Computer," Proceedings of Fall Joint Computer Conference (1962).

C. Y. Lee and M. C. Paull, "A Content Addressable Distributed Logic Memory with Applications to Information Retrieval," Proceedings of IEEE, Vol. 51 (1963).

A. J. Otis and G. P. Copeland, "Editing Requirements for Data Base Applications and Their Implementation on the INDY Backend Kernel," Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing (August 1978).

M. Shephard, Jr., "Distributed Computing Power: A Key to Productivity," Computer (November 1977).

S. Y. W. Su, G. P. Copeland and G. J. Lipovski, "Retrieval Operations and Data Representations in a Context-addressed Disc System," Proceedings of the ACM SIGPLAN/SIGIR Interface Meeting on Programming Languages and Information Retrieval, Gaithersburg, Maryland (November 1973).