A Hardware Pattern Matching Algorithm On A Dataflow

SAKTI PRAMANIK AND CHUNG-TA KING
Computer Science Department, Michigan State University, East Lansing, Michigan 48824, *USA*

A hardware pattern matcher is presented, which searches for patterns on a data flow, such as characters read from a disk. The backing up on the data flow, for a general pattern matching, is avoided by means of a set of cells running in parallel. Each cell can search for a pattern independently, but requires only one one-character comparator. The interesting feature of this search hardware is the use of a simple priority line which can dynamically allocate these cells. Further, the number of cells required can be arbitrarily reduced by means of a marking technique which is also accomplished by the priority line. Finally, an information retrieval system, based on this basic pattern matching hardware is presented. Here the content as well as the context search are done by usign the same marking technique.

1. INTRODUCTION

Conventional pattern matching in large databases, where the data is stored on direct access secondary storages, is done by transferring blocks of data into main memory buffer and then searching through it. This process requires a lot of data transfer from the secondary storages while only a small fraction of it may be useful. One way to avoid this unnecessary data transfer is to use indexes. But this can be done only at the cost of creating and maintaining a large index database. And the problem of accessing a lot of unnecessary data still exists. Another approach is to search for data directly on the secondary storages by means of hardware, while data are on the fly. This approach requires additional search hardware on the disk memory system. But it will filter out much of the unnecessary data, and the host, in turn, needs to process only a very small fraction of the database. A significant amount of research work (1 – 4,6 – 12) has been done in designing such backend processors.

Here, we will first describe one such backend processor for text processing applications. We will then show that the critical part of such a system is the pattern matching hardware. One pattern matching hardware design will be illustrated. And finally the advantages of this design over other existing systems will be given.

2. BACKEND SEARCH SYSTEM

An architecture of a backend search system (reference 15) is given in figure 1. The basic structure of this system consists of a disk memory system, a controller, a term matcher, a query resolver, and some overall controller for the backend system. The main issue of this system is the design for the term matcher, because it must be able to find terms (strings of tokens representing characters) at a speed less than the character delivery rate of the disk memory system. The query resolver, which processes the results of the term matcher, determines whether the terms occur in the correct proximity or match the required boolean expression. Its processing requirements are substantially less than the term matcher's and can be implemented using a high-speed microprocessor perhaps with hardware augmentation. The function of the search controller can be performed by either the query resolver or another microprocessor, while the disk controller will only drive the positioner for the disk heads.

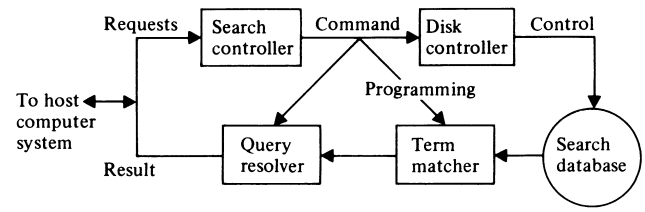

Fig 1. A backend search system

This search processing system can be attached to the disk system at several points. However, there are a number of advantages if we attach it directly to the output of the disk drive. Thus, it requires no buffer, and data are transferred only a short distance from the drive. Therefore, no elaborate cabling is required. Furthermore, if it is connected to each disk drive, a high degree of parallelism results.

The direct connection of the search system to the disk drive without the use of a buffer memory imposes some severe requirements on the term matcher. The small time window between the consecutive characters read may not allow the text characters to be processed completely. This is because a rotating disk will not allow backing up in text to try another alternative when a character mismatch occurs. For example, to find 'ISSIP' in 'MISSISSIPPI', we will detect a failure when trying to match the 'P' of the pattern against the third 'S' in text, and we can not start with the second 'I' of text again because this character is already passed in the data stream.

In the following section, we will present a term matcher design, which avoids this backing-up problem by using parallel comparison cells. We will then evaluate this design and show its advantages over the existing methods. Finally, we will propose a detailed design of an information processing system using this hardware design.

3. TERM MATCHER BY CASCADING CELLS

Figure 2 shows the basic construction of the term matcher. This term matcher uses a small number of one-character comparators. Each one-character comparator will be called a cell. These cells are selected by a priority-line. Each cell finds a pattern match independently without having to back up in the text stream. Whenever a possible matching string appears in

the text stream, one idle cell is activated to test for complete match. (By possible matching strings, we mean those substrings in the text that begin with the same character as that of the pattern). This activation is done by the priority-line. When there is no more idle cells, the priority-line will mark the database. And this part of the text stream which is not processed will require another revolution to search for matching, starting from the place where there is a mark. For example, the pattern ABAB and text ABABA will result in the following scenario: Cell 1 is activated first, and it will try to find a matching string starting with the first text character. When the third text character arrives, we already have a partial match with the substring AB found in Cell 1. But this third text character could be a start for another possible matching string. Consquently, a second cell will be activated at this point. Now we have two active cells and they search for matching independently.

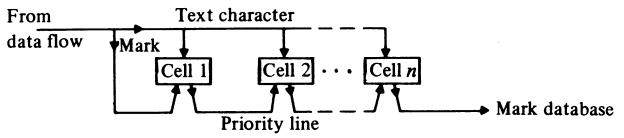

Fig 2. Cells for pattern matching in parallel

The algorithm presented here has some similarity to the cellular approach of Copeland[2] and Mukhopadhyay[8]. In their approach, each pattern character is stored in a single-character comparator, called a cell. Propagation of a status bit in the cellular array eliminates the need for backing up in the text stream. In our approach, the backing up in the text stream is avoided by using one cell to check for one possible matching string. The priority-line is used to activate idle cells when there is a possible matching string, it is also used to mark the database when all the cells are busy. Furthermore, these mark bits can be used to search for complex patterns, allowing context search. The work by Healy[4] also uses mark bits to avoid backing up in the text stream. A mark is used to set up the context for the next character in the pattern there. In our approach, we mark the database only when there are overlapping matching strings.

4. HARDWARE DESCRIPTION

Figure 3 provides a term matcher design where each cell of the circuit contains a one-character comparator. A cell compares a text character with the pattern character, and if it is a match, it compares the next character with the next pattern character, and so on. A complete match is found by a cell when this has proceeded up to the last character of the pattern. The selector in selection circuit is used to load the next pattern character in the pattern register for the associated cell. A status bit S is set in a cell when the cell is active and is progressing with a check. When there is a mismatch, this bit is reset and the cell becomes available again. A priority-line (L-line) is used here to dynamically activate one of those idle cells to check for matching. It will always carry a one in the first revolution to allocate the cells to check the whole text stream. This signal is transmitted through the L-line to the right. While travelling to the right, if it finds a cell available, the search is initiated on the cell, and the signal

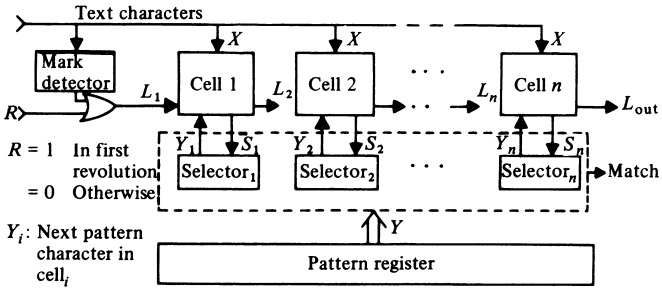

Fig 3. Details of a matching hardware

on the L-line becomes a zero. On the other hand, when all cells are busy, this signal travels all the way to the right, and is used to mark the text character on the database. In subsequent revolutions, L-line will carry a one only when the mark detector detects a mark. This signal will activate one idle cell to check that portion of text that was not checked in previous revolutions, and it will delete this mark, too.

The state of each cell is given in Table 1, and the corresponding logic diagram is given in figure 4.

Table 1. State table for ith cell

Si^{t-1}	Mi^t	Li^t	$L_{i+1}{}^t$	Si^t
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1

Where

$L_{i+1}{}^t = Si^{t-1} \wedge Li^t$

$Si^t = (Si^{t-1} \vee Li^t) \wedge Mi^t$

$Mi = \begin{cases} 1 \text{ when match occur} \\ 0 \text{ otherwise} \end{cases}$

t: instant at which the present character is read

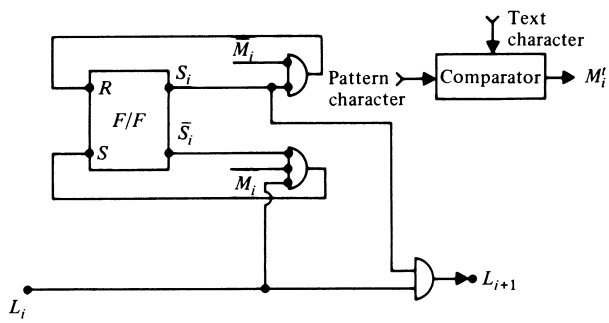

Fig 4. Details of logic diagram of ith cell

One possible design of the selection circuit is given in Figure 5, where each row of the flip-flops is called one selector and corresponds to one cell. If the ith cell is idle, S_i of this cell will be kept at zero, and the output of FF1 in the selector of that cell is always one (we call it select signal), and outputs of other flip-flops are reset to 0. Thus, the selection circuit always loads the first character of pattern string to cell i. When the ith cell is activated

and finds a possible matching string, S_i will become one. Thus, a zero will be fed into the selector and the select signal at the output of FF1 will propagate to the next stage. Then, the second character of the pattern string is loaded into cell i to check for matching, and so on.

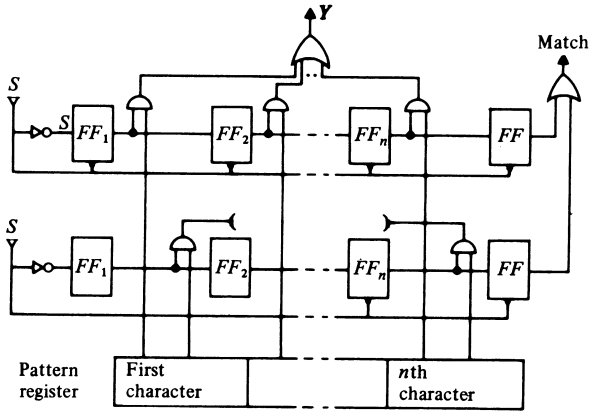

Fig 5. Details of selection circuit

To synchronize the operations of cells and selection circuit, we can use one clock in the matcher. The operations that are performed during each cycle are shown in Figure 6. The clock is connected to selectors as well as each cell. In this design, the loading of a pattern character can be done in parallel with the propagation of the priority signal. On the other hand, comparison of characters and setting the S bit have to be done after the loading. Taking a conservative estimation we will need 10-gate delays for loading a pattern character during time T_1. In the same time, we can propagate the priority signal as many as 10 cells. Note that with the facility of independent checking, one cell will be enough to search for the whole pattern. Referring to Figure 4, the propagation delay in each cell (ie delays to compare and set S bit) is less than 10-gate delays. Thus, one clock with cycle time equals to 20-gate delays would be sufficient to process a complete one-character comparison.

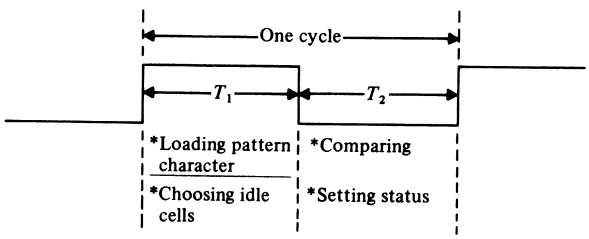

Fig 6. Clock cycle in the matcher

In the rest of this section we shall give one typical example to show how the cells perform searching in parallel, how they are allocated dynamically, and how the marks are used to defer the searching until the cells become available in subsequent revolutions. Here, we assume that the term matcher consists of only two cells. A tabular form to show the operations in each cycle is also given in Figure 7.

Pattern = ABAB Text = ABABABC

First revolution: ABABABC (1 1, 2)

Second revolution ABABABC (1 1)

Revolution . . .	1							2						
Text . . .	A	B	A	B	A	B	C	A	B	A	B	A	B	C
Cell 1														
Pattern	A	B	A	B	A	B	A	A	A	A	A	A	A	A
L_1	1	1	1	1	1	1	1	0	0	0	1	0	1	0
S_1	1	1	1	1†	1	1	0	0	0	0	0	0	0	0
Cell 2														
Pattern	A	A	A	B	A	B	A	A	A	A	A	A	A	A
L_2	0	1	1	1	0	1	1	0	0	0	0	0	0	0
S_2	0	0	1	1	1	1†	0	0	0	0	0	0	0	0
Mark-bit (L_3)	0	0	0	1	0	1	0	0	0	0	0	0	0	0
Match-bit	0	0	0	1	0	1	0	0	0	0	0	0	0	0

$\underbrace{\quad}_{i}$: Characters that are checked in cell i

† : Resetting the status bit when complete match occurs

* : Marking the text characters and indicating it is not examined

Fig 7. An example of finding matches by a two-cell matcher

5. PATTERN MATCHING IN MULTIPLE TEXT STREAMS

From the discussions above, we can see that the main features of this design are the ability to dynamically allocate the cells by using a priority-line and to independently check for matching in each cell. With these features in mind, some simple extensions could lead to far more complex functions, like searching a pattern in multiple text streams in parallel or searching several patterns simultaneously.

One extension is to add one-character comparators to the matcher of Figure 3. This is shown in Figure 8, and we shall call them start-up cells. Several start-up cells are used to search for multiple patterns. A start-up cell will always compare the text stream with the first pattern character. When the start-up cell finds a match, it will activate one of the idle cells by using the same priority line, to go on to check that possible matching string. The synchronizer swithes the right pattern to the right cell. This will be explained further later.

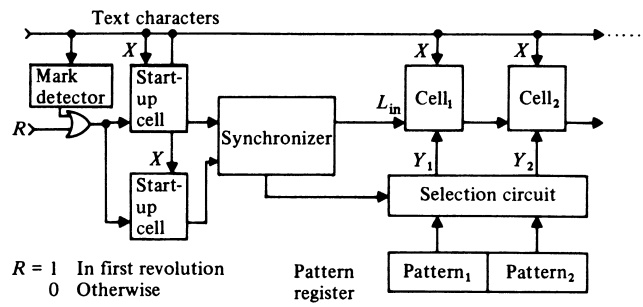

Fig 8. Matching hardware with start-up cells

Another extension is to modify the selectors by making them programmable. With a little more hardware, we can make the select signal in the selector starts propagating from ith flip-flop and ends in jth flip-flop. Thus, in addition to dynamically allocating cells, we can dynamically load desired pattern characters to each cell.

Assume that we try to match m patterns with one text stream. To implement it in a parallel fashion, we can use one start-up cell to search for each pattern, and keep a small set of cells in a central area. Whenever a start-up cell finds a possible matching string for that pattern, it will signal the synchronizer. The synchronizer will, in turn, try to activate one idle cell and switch the specific pattern characters into that cell by properly programming the selector associated with that cell. This will require some extra hardware to implement the switching circuit (ie the synchronizer).

We can also dynamically allocate a set of cells for multiple text streams. For each stream, we can use one start-up cell to search for possible matching strings in that stream. Still, we need only keep a small number of cells in some central area to serve for all the text streams. If there is one match found in any stream, the start-up cell corresponding to that stream will switch this data stream to one of the idle cells. Again, the switching circuit will require some extra hardware.

6. COMPARISON

As mentioned before, the matcher design here has some similarity to the cellular logic. In cellular logic, pattern matching is done through a cellular array. In our approach the text stream is first searched by a start-up cell, and then it is checked for matching by other cells. Though the cells in our approach function much intelligently from those in cellular logic, they have similar construction – both are one-character comparators. We can use cells in cellular logic array to construct cells in our approach, as shown in Figure 9. Comparing with our design in Figure 4, we do not have to store pattern characters in each cell, because this function is taken care of by pattern register.

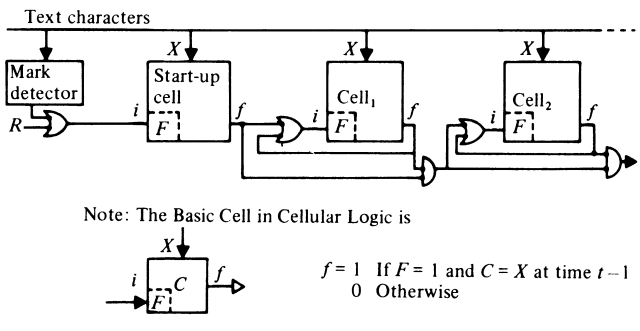

Fig 9. Construction of matcher using cells of cellular logic

From the analysis before, we see that it takes 20-gate delays to compare one character in our design. Comparing with the cellular lagic (8) we use twice the time. The longer dalay in our design is because we have to load pattern characters in each cycle. With a 5 ns/logic stage for monolithic circuits, pattern matching can be performed at the data flow rate of 10 million characters/second, which still exceeds the data transfer rates of high-speed secondary storages.

The features which make our design different from that of cellular logic are the dynamic allocation of cells by a simple priority circuit and the use of mark bits. They are further explained below:

With the ability of dynamic allocation, we can use only a small set of cells to search for several patterns simultaneously. We can also extend our design to search on multiple text streams using the same set of cells. In cellular logic, because the cells are statically allocated, we have to use as many arrays of cells as there are patterns. Dynamic allocation is possible in cellular logic but it is done at the cost of more expensive interconnection network. Thus, the hardware complexity is reduced considerably in our design and yet a high degree of parallelism remains.

Marking technique is used here to mark a possible matching string in the text when all the cells are busy. Subsequent revolutions are required to check for matching within that marked string. In practical applications, those overlapping subpatterns are seldom, so the set of cells can be reduced arbitrarily without degrading the performance of the matcher too much. In this way, we can minimize the hardware complexity considerably.

7. APPLICATION OF MATCHING CIRCUIT FOR DATABASE RETRIEVAL

A more general pattern matching operation and information retrieval primitives can be implemented by controlling L-line of figure 3 by a more general control circuit. The mark detector of figure 3 now becomes a part of this control circuit. Besides L, we use another input line 'RESET' to reset the status bits of all the cells. This is used to terminate all partial searches that may be in progress in a cell. Figure 10 shows the hardware organization of a stage consisting of a control circuit and a matching circuit. The matching circuit is simply the cascaded cells of figure 3, while the control circuit provides the hardware to control L_{in}, mark bits, RESET-lines, all under program control. In fact, the database queries can be implemented by properly masking the pattern characters, and executing a sequence of operation codes. The operations use two types of marks, namely, type A and type B. With these marks a more general context search can be performed. Communication between adjacent stages is needed to allow a record to reside across track boundaries. A line is used for this purpose to transfer the values of the counters of the ith stage to those of the $(i+1)$th stage at the beginning of each cycle. A 'DONE' bit in each stage indicates that the stage has completed the operation. The controller activates the next operation code when all the stages are done with the current operation.

The control circuit consists of several flip-flops. Values of these flip-flops control the Anchor (ie L_{in}), RESET, Mark/Unmark A, and Mark/Unmark B lines. Which flip-flop(s) will control a particular line can be selected by means of the different bits of the operation code. The flip-flops are reset by the clocklines at the begnning of each cycle. The other control lines, namely L_{out} and MATCH are coming from the matching circuit, and the 'A' and 'B' lines are coming from the mark detectors.

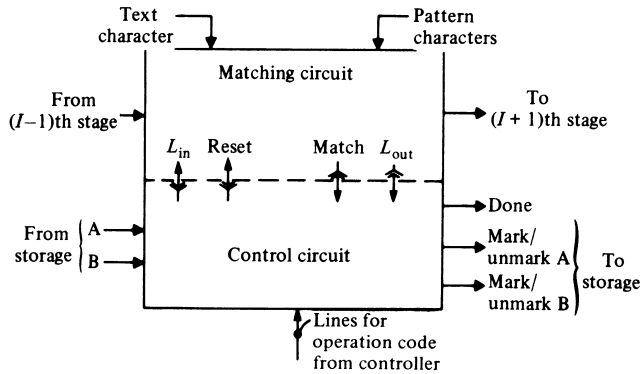

Fig 10. Hardware organization of the ith stage

The context for every search operation is established by the mark bits. Three types of contexts, namely, open, semi-open and closed are used in the application discussed below. In open context, the whole text is searched and no mark is necessary; in semi-open context, on the other hand, every search is initiated by a mark representing the left context. In closed context all the consecutively marked characters represent the context. Any particular context can be selected by the appropriate bits of the operation code. The bits of the operation code are described below.:

OC: open context
SC: semi-open context
CC: closed context
ANCA: anchor A-mark (ie an A-mark will activate the anchor line)
ANCB: anchor B-mark
MRKA: mark character with A-mark
MRKB: mark character with B-mark

Conditions under which actual marking will take place depend on the values of the MARK A (MA), MARK B (MB), CURRENT MATCH (CM) and PREVIOUS MATCH (PM) flip-flops. Any subset of these groups can be chosen by the operation code, not shown here, and their values can be ANDed or ORed. These flip-flops are described in Table 2 below. A more detailed description of the control circuit is given in Reference 11.

Table 2. Description of the flip-flops

FLIP-FLOP	*OPERATION*
MA	Set by an 'A' mark Reset by $(SC \wedge MATCH \wedge ANCA) \vee (CC \wedge \overline{A} \wedge ANCA)$
MB	Set by 'B' mark Reset by $(SC \wedge MATCH \wedge ANCB) \vee (CC \wedge \overline{B} \wedge ANCB)$
CM	It is set by MATCH Reset by $\overline{MATCH} \wedge (OC \vee SC)$ or $CC \wedge ((\overline{MA} \wedge ANCA) \vee (\overline{MA} \wedge ANCB))$
PM	Set by CM Reset by $\overline{CM}$

The example given below explains the use of the operation codes and the flip-flops.

Example:

Find all occurrences of the supplier record with supplier

name = ? SMITH ? CO
? stands for any string

We assume that the records are separated by the special symbol δ and the fields within a record are separated by the symbol a. The field name at the beginning of each field identifies the field and is separated from the field value by the delemeter β. The field name for the supplier name is assumed to be SUPNAM. The following steps accomplish the above query. Each step below represents the execution of an operation code.

	Comments
Step 1 OC = 1 Mark with A-mark when PM = 1 Contents of Pattern register: aSUPNAMβ	Mark first character of all the supplier name value field with A-mark
Step 2 SC = 1, ANCA = 1 Mark with B-mark when MA ∨ CM = 1 Contents of Pattern register: a	Mark all characters of supplier name value mark with B-mark
Step 3 CC = 1, ANCB = 1 Mark with A-mark when CM = 1 Contents of Pattern register: SMITH	Search for SMITH in the string whose characters are all marked B. If a match is found, mark all characters in the name field following SMITH with A-mark
Step 4 CC = 1, ANCA = 1 Mark with B-mark when CM = 1 Contents of Pattern register: CO	Search for CO following SMITH
Step 5 SC = 1, ANCA = 1 Mark with A-mark when CM = 1 Contents of Pattern register: δ	Mark all records with A-mark which has a B-mark present on any character within the record

The use of mark bits for both data structuring as well as query processing was first suggested by Parhami[9]. The search time there was prohibitively long because of a complete revolution needed for each character of the pattern. The mark bits have also been used in RAPS[13], CASSM[14] and others, providing a hardware cost versus reponse time trade-off for processing queries. CASSM also uses a one bit wide random access memory to mark backwards as well as related tuples in another relation in the subsequent revolutions. These schemes use marks at

the tuple level and are not sufficient for information processing systems where manipulation at the character level is required. Character level marking, for example, makes implementation of embedded variable length don't cares simpler.

8. CONCLUSIONS

A hardware pattern matcher for data while it is on the fly is presented. The advantages of this approach over the others are given. In general, pattern matching hardware can be very simple if the patterns do not require any back-up. When they do, algorithms become more complex. We either have to have parallel searching embedded in these algorithms, Copeland[2], Mukhopadhyay[8], or we need to preprocess the patterns to generate the next (J) table, Knuth, Morris, Pratt[5]. In the latter case, the input text has to be held statically in a buffer during the searching, or for each mismatch the text needs to be dynamically skipped, depending on the value of next (J). We solve the problem here by using mark bits. This simplifies the pattern matching hardware considerably. The extra revolution required, due to the marks, are also minimal because we accomplish some parallel searching in the cells by a simple priority circuit.

REFERENCES

1. E. Babb, Implementing A Relational Database by means of Specialized Hardware, *ACM TODS*, **Vol 1,** (1979).
2. G. Copeland, String Storage And Searching for Database Applications: Implementation of The INDY Backend Kernel, *Proc. 4th Workshop on Comp. Architecture for Nonnumeric Processing*, pp 8 – 17, (1978).
3. F. Burkowski, A Microprogrammed Search Controller for Text Scanning Processors, *Proc. 5th Workshop on Comp. Architecture for Nonnumeric Processing*, (1980).
4. L. Healy, A Character-Oriented, Context-Addressing Segment-Sequential Storage, *Proc. 3rd Annual Symp. on Comp. Architecture*, pp 172 – 177, (1976).
5. D. Knuth, J. Morris, V. Pratt, Fast Pattern Matching in String, *SIAM J. Comp.*, **Vol 6,** No 2, pp 323 – 350, (1977).
6. G. Lipovsky, L. Healy, K. Doty, The Architecture of A Context-Addressed Segment-Sequential Storage, *Proc. of AFIPS Conf.*, **Vol 41,** (1972).
7. A. Masri, J. Rohmer, D. Rusera, A Machine for Information Retrieval, *Proc. 4th Workshop on Comp. Architecture for Nonnumeric Processing*, pp 117 – 120, (1978).
8. A. Mukhopadhyay, Hardware Algorithms for Nonnumeric Computation, *IEEE Trans. Comp.*, **Vol C – 28,** (1979).
9. B. Parhami, A Highly Parallel Computer System for Information Retrieval, *Proc. Fall Joint Comp. Conf.*, (1972).
10. S. Pramanik, Highly Parallel Associative Search And Its application to Cellular Database Machine Design, *Proc. AFIPS Conf.*, (1981).
11. S. Pramanik, Hardware Organization for Nonnumeric Processing, *Proc. on VLDB*, (1981).
12. D. Roberts, A Specialized Computer Architecture for Text Retrieval, *Proc. 4th Workshop on Comp. Architecture for Nonnumeric Processing*, pp 51 – 59,(1978).
13. S. Schuster, et al, RAP2 – An Associative Processor for Database And Its Applications, *IEEE Trans. Comp.*, **Vol C – 28,** (1979).
14. S. Su, et al, The Architectural Features and Implementation Techniques of The Multicell CASSM, *IEEE Trans. Comp.*, **Vol C – 28,** (1979).
15. D. Hsiao, Advanced Database Machine Architecture, Prentice-Hall, (1983).