MICROPROCESSORS FOR NON-NUMERIC PROCESSING

S. G. Zaky
Department of Electrical Engineering
University of Toronto
Toronto, Ontario, Canada

Abstract

The problem of processing of non-numeric data has received considerable attention in the last few years. This is primarily motivated by the pressing needs in the area of data base management. It has long been recognized that the parallel processing capabilities of an associative processor are fundamentally well suited to this environment. However, the complexity and cost of truely associative memories make this approach impractical. In this paper, the demands that non-numeric processing place on memory and processor hardware are discussed. Some emerging trends are presented, and a suggestion is given regarding the development of a "general purpose" microprocessor that is suited to this environment.

Bascially, the microprocessor discussed here is capable of performing simple search and update functions on a high speed, serial data stream. Therefore, it is suited to any application where such a situation is encountered, such as in digital communications.

1. Introduction

The study reported in this paper is an attempt to relate processor architecture to the environment in which it operates. It is relatively easy to design a special purpose processor to handle a given problem. This is not the intent of the discussion below. The main purpose of the study is to define, in general terms, the salient characteristics of the non-numeric processing environment. These characteristics are then utilized to derive a number of architectural features that are desirable in a processor that is to be used in this environment. The range of applications of the resulting processor is quite wide. Hence, it may be regarded as a general purpose processor for the non-numeric environment.

This study was motivated by the current research in the area of information retrieval. The following section gives a brief look at some hardware related aspects of data base design. This is followed by an analysis of the requirements of non-numeric processing. The remainder of the paper is devoted to the development of an architecture for a general purpose processor that satisfies these requirements.

2. The Data Base Problem

The main element in a data base system is the storage device. A survey of current developments in storage technology indicates that almost all memory systems that are, or are likely to become, economically feasible share a block oriented organization. That is, they provide random access at the block level and serial access to data within a block. This organization can be found in mass storage devices that are currently in common use, such as magnetic disks and drums, as well as in devices that are just becoming commercially available, such as CCD [1] and bubble memories [2]. The same organization can be found in very high capacity storage devices that are currently in the development stage, such as holographic [3] and electron beam accessed [4] memories. Some research is aimed at the development of mass storage devices with true associative capabilities [5]. However, the practicality of this approach is still in doubt. Therefore, it is reasonable to conclude that a block oriented organization will be in use for a long period to come.

The information retrieval problem in a block oriented memory organization can be approached in one of two ways. The first of these is to move one data block at a time to either a general purpose or a special purpose processor which implements the required search algorithm [6]. This approach is limited by the speed of the processor and by the bandwidth of the I/O transfer. While the processing speed may be increased through the use of an associative processor, the speed of transferring data from mass storage to this processor remains as a fundamental limitation. The second approach is to provide a separate processor for each block of data. This is feasible in many types of mass storage devices. For example, in a fixed head magnetic disk system, where there is a dedicated head for each track, simultaneous access to all data blocks is possible. The same situation obtains in bubble and CCD memories. The utilization of the possibility of parallel access to data blocks seems to be the closest practical alternative to true associative processing on a large amount of data. Since the idea was first introduced [7], many designs have been proposed [8-10], and some of them are currently in the implementation stage. The use of the term "associative processors" to describe such multiprocessor systems seems to have been accepted

in the literature, despite the fact that they are only partially associative.

3. Processing Requirements

Almost all current designs for the associative processors mentioned above are based on the use of special purpose hardware to perform the function of the processor that is dedicated to each data block. Because of considerations of size, cost, etc., a single chip microprocessor would be ideally suited for this purpose. Unfortunately, the general purpose, von Neumann type machines that are currently available are not capable of performing the required tasks. In what follows, the problem of designing a processor architecture that is suited to this application will be considered. The intention is to develop a design that is not restricted to a particular data structure or to a particular set of search and update instructions. In that sense, the processor may be regarded as a "general purpose" processor for non-numeric applications.

Let us start by defining, in as abstract a form as possible, the environment in which a block processor may operate and the functions it may be expected to perform. These may be described as follows:

1. Data is fed to the computer serially.
2. The timing and order of data transfers is controlled by extraneous factors that are not subject to alteration by the processor (in the general case).
3. Access to the stored data takes place via a "window" against which data is continuously moving. The size of this window and the amount of time that the processor is allowed to inspect or update its contents are design parameters that may vary from one implementation to another. They are primarily dependent upon the amount of buffering provided in the processor.
4. The stored information consists of units that are variable in length.. It is not unreasonable, however, to expect a fundamental unit in the form of a fixed length character to exist in the system.
5. The main function of the processor is the detection of an arbitrary character string that is defined by the instructions the processor receives. It should also be capable of performing simple update or counting operations on the data within the window when such strings are detected.

The most striking feature of the environment of the block processor is item 2 above. This is fundamentally incompatible with the operation of a von Neumann computer. In the latter, the processor fetches the selected data words from memory only when it decides to do so, as a result of the execution of an instruction. The block processor does not have this privilege. We should note that a similar situation is encountered in many applications. Usually, it is handled by storing the incoming information into the main memory via DMA. Then, the processor can access this information in any order, at any time. In other words, the problem is transformed from its natural domain to that in which the von Neumann processor can operate. This is a rather time consuming approach. It is feasible only where the processing speed is much higher than the rate at which data is received.

The above discussion leads to an interesting concept. The block processor should be "data driven", rather than "instruction driven". That is, the internal activity of the processor should be synchronized and controlled by the continuous stream of input data. Instructions to the block processor change relatively infrequently, and they simply condition its response to the various types of data items encountered in the data stream.

The concept of a data driven computer suggests an organization such as that shown in Fig. 1. The instruction sequencing unit inspects the input data. Upon the occurrence of a given event, such as the arrival of a given character or character string, the unit fetches an appropriate instruction from the program memory and sends it to the execution unit. The instruction specifies further action to be taken. This may include fetching a another instruction from the program memory, operating on data in the data memory, or searching for a new character in the input data stream. Therefore, the most fundamental operation in the instruction sequencing unit is a search operation on the input data stream. In what follows, we will consider means by which this may be accomplished.

A number of ideas will be put forth regarding the design of a processor that is suited to the non-numeric environment. While the block processor discussed above has motivated the design, many of the design concepts have a much wider range of applications.

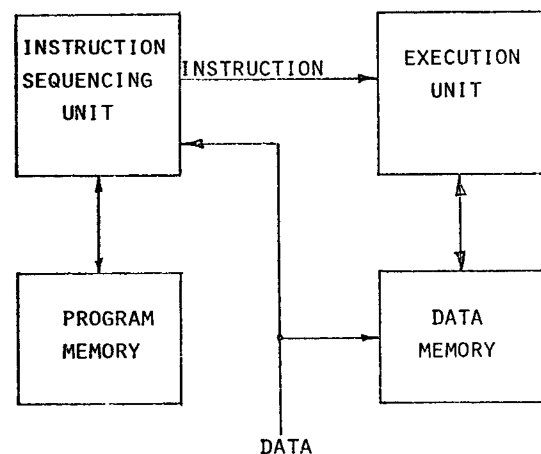


Fig. 1 Suggested organization for a general purpose non-numeric processor.

4. The Search Operation

Consider a simple linear search in which it is required to detect a given character in an input data stream. In a conventional general purpose architecture, this search may be implemented by a program loop such as that shown in Figure 2. While the details of actual instructions inside this loop may vary from one machine to another, the structure remains essentially the same. Despite the fact that there are only few instructions, and that they have to be executed repeatedly, they have to be fetched from memory for every pass through the loop. In general, the number of memory fetches increases with the complexity of the search, as for example, in the case of searching for more than one character. In order to decrease the overhead resulting from instruction fetches, it is possible to define "high level" machine instructions. Such instructions combine commonly used sequences of primitive operations, such as mask, test and branch, into a single machine instruction [11]. Another possibility is to use instructions that operate on a sequence of data words, rather than on a single word, as in the case of the string instructions in the IBM machines. This results in replacing the whole loop by a single instruction, for some simple operations. In some sense, the processor presented in this paper may be regarded as a processor with a number of such high level instructions. However, it differs from a conventional processor in the way in which the timing and sequencing of instruction fetches are controlled.

```
LOOP:   INPUT WORD
        APPLY MASK
        COMPARE WORD, REGISTER
        IF <CONDITION> THEN, BRANCH TO END
                       ELSE, CONTINUE
        BRANCH TO LOOP
END:    .....
```

Fig. 2 Program loop for a simple search operation.

Let us consider first the means by which a simple search operation may be performed in hardware. The main component required for this operation is a comparator. The search for a string of characters $C_1C_2...C_n$ in the input stream may be accomplished by a hardware organization such as that shown in Fig. 3. The string $C_1C_2...C_n$ is stored in a random access memory which is addressed via the counter POINTER. The address counter is initially set to point at the memory location containing C_1. Whenever the character C_1 appears in the input stream, a match is indicated by comparator 1. The pointer is then incremented to point at C_2. If the next comparison operation results in a match, the pointer is incremented to point at C_3, otherwise it is reset to its initial value to restart the search at C_1. In order to detect the end of the comparand string $C_1C_2...C_n$, a delimiter character DLR may be stored following C_n, and detected by the second comparator shown in the figure. Alternatively, comparator 2 may be used to detect the address of C_n when it appears on the memory address bus.

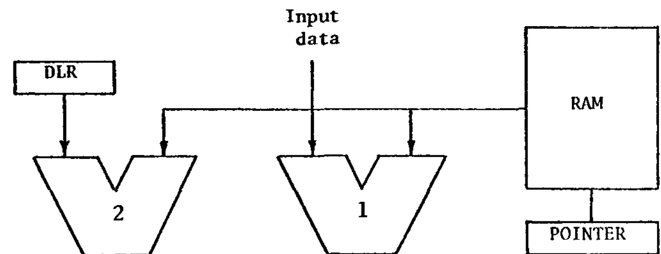
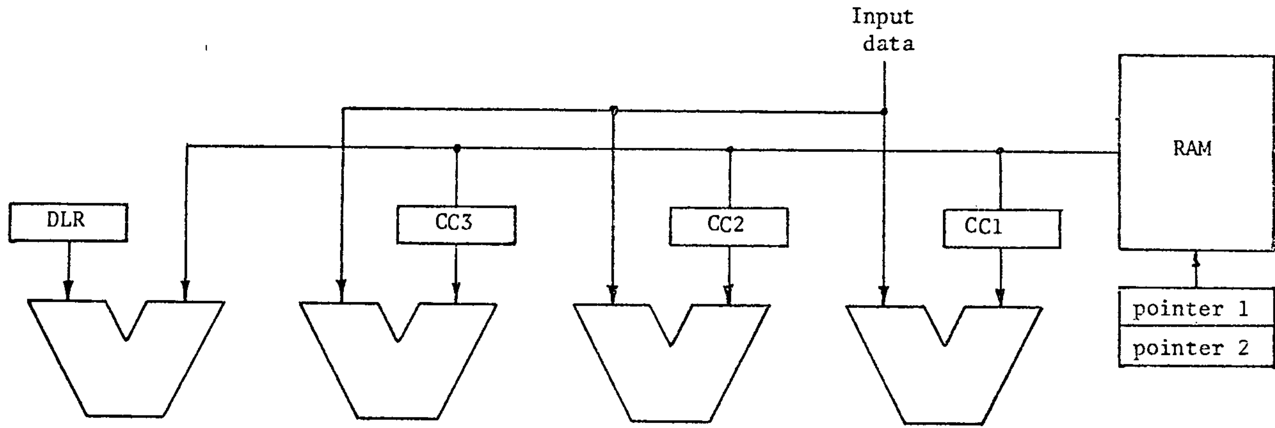


Fig. 3 Simple hardware organization for search operations.

Unfortunately, the simple scheme of Fig. 2 cannot handle the general case where the character C_1 may be repeated inside the string. This is easily seen by considering the comparand string ABAC and the sequence ABABAC in the input stream. The memory address pointer will be advanced until it reaches the letter C in the memory, at which point a mismatch will occur and the pointer will be reset to point at the first character of the comparand. Thus, the starting point of the string ABAC in the input stream is not detected. In order to overcome this problem, we should continue to check the first character C_1 against every new character in the input stream, even after the pointer is incremented to point at other characters of the comparand. Whenever C_1 is detected, a new sequence of comparisons should be started. This means that multiple comparisons are required for each input character, which may be accomplished by using a number of comparators in parallel. The sequence of comparisons for the above example becomes as shown in Fig. 4. In this case, 4 comparators are needed. Two of these are dedicated to detecting the first character A and the delimiter DLR, while the other two are used in conjunction with the two memory address pointers for the two comparison sequences that develop during the search operation. It can be easily seen from this example that one comparator should be added for each new occurrence of the first character C_1 within the comparand string. The need for multiple comparators is not surprising in due of the associative nature of the problem.

5. Use of a Content Addressable Memory

Let us now consider the possibility of using a content addressable memory to perform the multiple comparisons that are required for the search operation discussed in the previous section. The configuration of Fig. 5 is suggested for this purpose. The system consists of a content addressable memory, a "mark" register, a priority register and an instruction memory. The Mark register contains one bit for every word in the CAM. Only those words whose corresponding bits in the Mark register are set are considered at any stage in the comparison. Other words in the CAM are ignored. In cases where it is required to consider the results of the comparison at a number of words in the CAM, the



	input	A	B	A	B	A	C
comparand string ABAC	CC1	A	A	A	A	A	A
	CC2	d	B	A	C	d	B
	CC3	d	d	d	B	A	C → search successful

d indicates "don't care".

Fig. 4 Use of multiple comparators to handle strings with repeated first character.

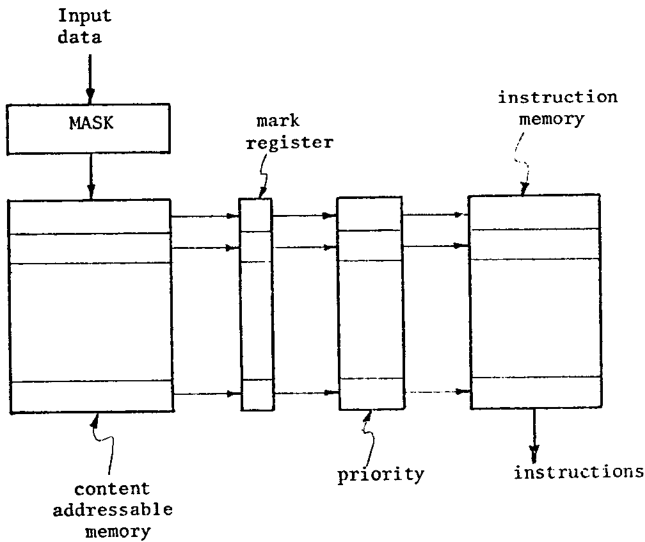


Fig. 5 Organization of the Associative Search Element (ASE).

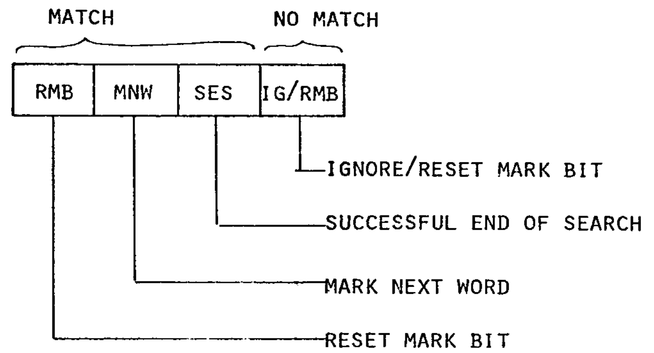


Fig. 6 4-bit instruction format for the associative search element.

words are considered according to the order indicated in the priority register.

When a character is applied to the input port, the control logic uses the contents of the Mark and priority registers to select an instruction from the instruction memory. The instruction memory contains one entry for every word in the CAM. The entry specifies the action to be taken in the case of a match and in the case of no match, at a word whose mark bit is set. These actions may be any of the following:

In the case of a match:

- reset mark bit (RMB)
- mark the next word (MNW)
- signal a successful end of search (SES).

In the case of no match:

- ignore (IG)
- reset mark bit (RMB)

The above instructions may be encoded in 4-bit instruction words, as shown in Fig. 6.

The associative search element of Fig. 5 may be used to perform a variety of search operations. For example, in order to perform the string search operation discussed in Section 4, the comparand string ABAC is loaded into the CAM, and the search program is loaded into the instruction memory, as shown in Fig. 7. At the beginning, only the first bit in the mark register, which corresponds to the first occurrence of the letter A in the CAM, is set. As the input characters are received, marking of other words in the CAM proceeds as shown in the figure, until the instruction Successful End of Search in reached. It can be easily seen that the same program is capable of detecting multiple, overlapping occurrences of the comparand string in the input character stream. This is demonstrated in Fig. 8.

The search capabilities of the associative search element are not restricted to the string search of Fig. 7. For example, the character string of Fig. 7 with "don't care" characters inserted between any two letters may be handled through the use of the Ignore command when no match is found. Also, the occurrence of a set of characters in any order may be detected by loading these characters in the CAM and setting all the corresponding Mark bits. Then, the Mark bits are reset as the characters are encountered in the input stream. The search is completed when all the mark bits are equal to zero. Hence, the Associative Search Element of Fig. 5 may be regarded as a general purpose processing element that can be used in a variety of applications in the non-numeric processing environment.

6. Processor Organization

Let us now focus our attention on the development of a general purpose processor that makes use of the Associative Search Element described in the previous section. A possible organization for such a processor is shown in Fig. 9. It follows the general organization of Fig. 1, where the ASE serves as the instruction sequencing unit. The instruction memory of the ASE is assumed to be increased in width to accommodate some of the instructions of the processor, in addition to the 4-bit ASE instructions of Fig. 6.

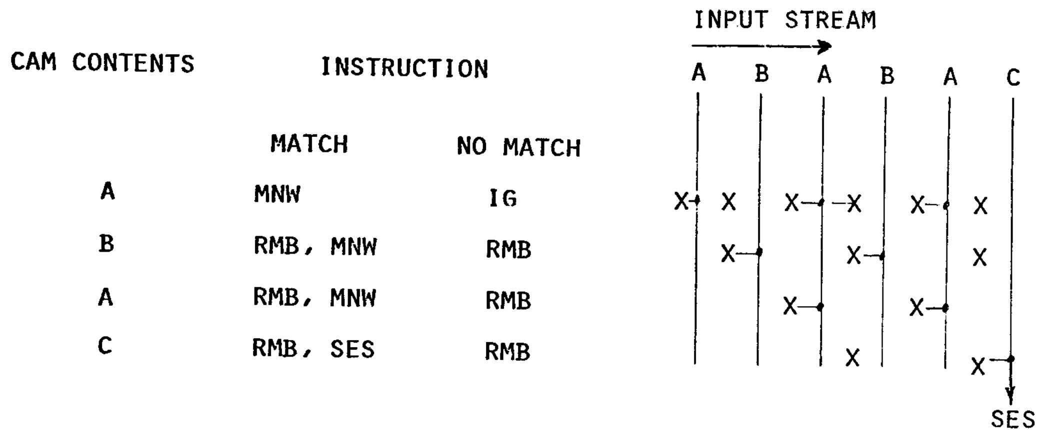
Operation of the processor of Fig. 9 may be described as follows. Each data item received at the serial input port is converted into a parallel format in the S/P register. Then, it is transferred to the ASE for a search operation. If the search is successful, an instruction is transferred from the instruction memory of the ASE to the instruction bus, and executed by the control unit. In most cases the output of the ASE is a call instruction to a service routine stored in the program memory. This routine may update the contents of the CAM, the mark register, or the instruction memory in the ASE. After execution of the service routine, the processor wâits for another instruction from the ASE, which will result in a call to either the same or a different routine in the program memory. Sequencing of instructions within a service routine is controlled by the program counter PC.

Data Structure

The function of the two blocks "format Control" and "RAM Access Control" in Fig. 9 is to facilitate handling of the data structures at the I/O port and in the RAM, respectively. Let us consider first the format of the input data. Any data structure when mapped on a linear medium results in a linear array of records separated by "record separators". Each record may be divided internally into smaller data units or fields. Again, field separators may be used to identify individual fields within a record. Fields and records are, in general, of variable length. During processing of such data it is necessary to identify the beginning and end of each field and each record. We will assume that, in general, these points are detected by special hardware that is not a part of the processor. This information is passed on to the processor in the form of two synchronizing signals which we will refer to as Record Start (RS) and Field Start (FS). This assumption is necessary, since in some cases, the detection of these points is device dependent, as for example in the case of the record gap on a magnetic tape, or the starting point of a track which is optically detected on a floppy disk. The interpretation of the RS and FS signals is program dependent and takes place in the Format Control block. The RAM Access Control block facilitates access to such data structures as FIFO buffers and LIFO stacks.

Programming Example

In order to illustrate the operation of the processor of Fig. 9, we will consider the following simple example. Assume that it is required to count the number of occurrences of a given string STRING, until another delimiter string DLR is detected in the input stream. The program for this operation is as follows:



X INDICATES THAT THE MARK BIT IS SET.

Fig. 7 Search program for the string ABAC.

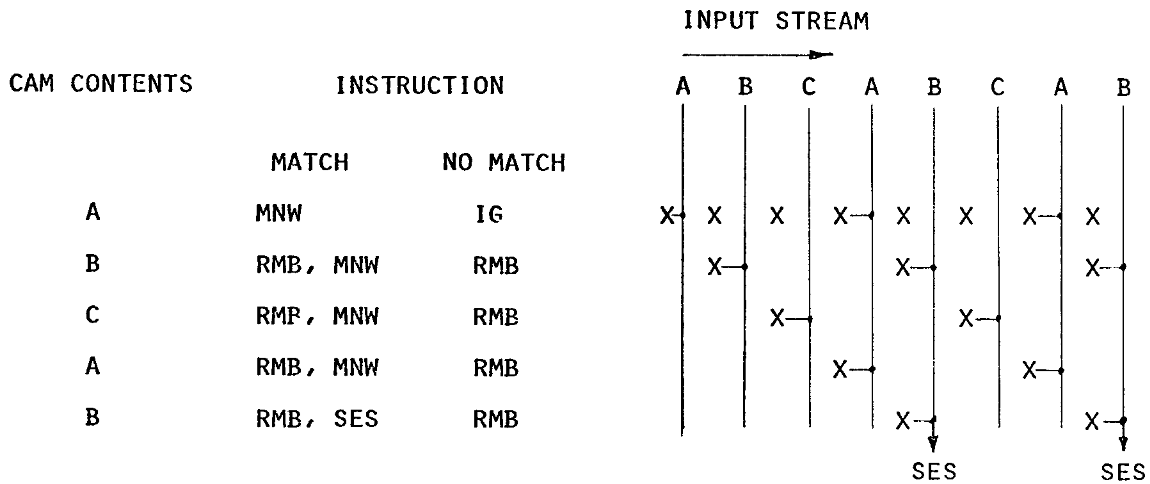


Fig. 8 Detection of overlapping strings.

```
        Load "Search for STRING, call COUNT"

        Load "Search for DLR, call END"

        Clear CNTR

        Start Search when RS is received

COUNT:  Increment CNTR, Return

END:    - - -
```

The first and second instructions load two programs similar to that in Fig. 7, which are assumed to be stored somewhere in the program memory, into the ASE. The second parameter in each instruction specifies the instruction that is to be generated by the ASE when the search is successful. We should note that the two Load instructions do not start the search operation. They only load the ASE programs. The search operation is started by the Start Search instruction, which specifies that the processor should wait for the external synchronizing signal RS before starting the search.

The counter CNTR is one of the general purpose registers. It is cleared initially, then incremented every time the routine COUNT is called. This is a 1-instruction routine that increments CNTR, then returns control to the ASE.

The following observations regarding the above program should help to clarify the nature of the new processor.

1. The program involves very few accesses to the program memory during the search operation. Once the search is started, the program memory is accessed only when the counter CNTR is to be incremented. Furthermore, each access involves only one instruction fetch.

2. The program instructions are closer to being high level instructions rather than rudimentary machine operations. This is a direct result of the fact that the processor operates in the input data domain, without the need to map the data onto another arbitrary structure such as a random access memory.

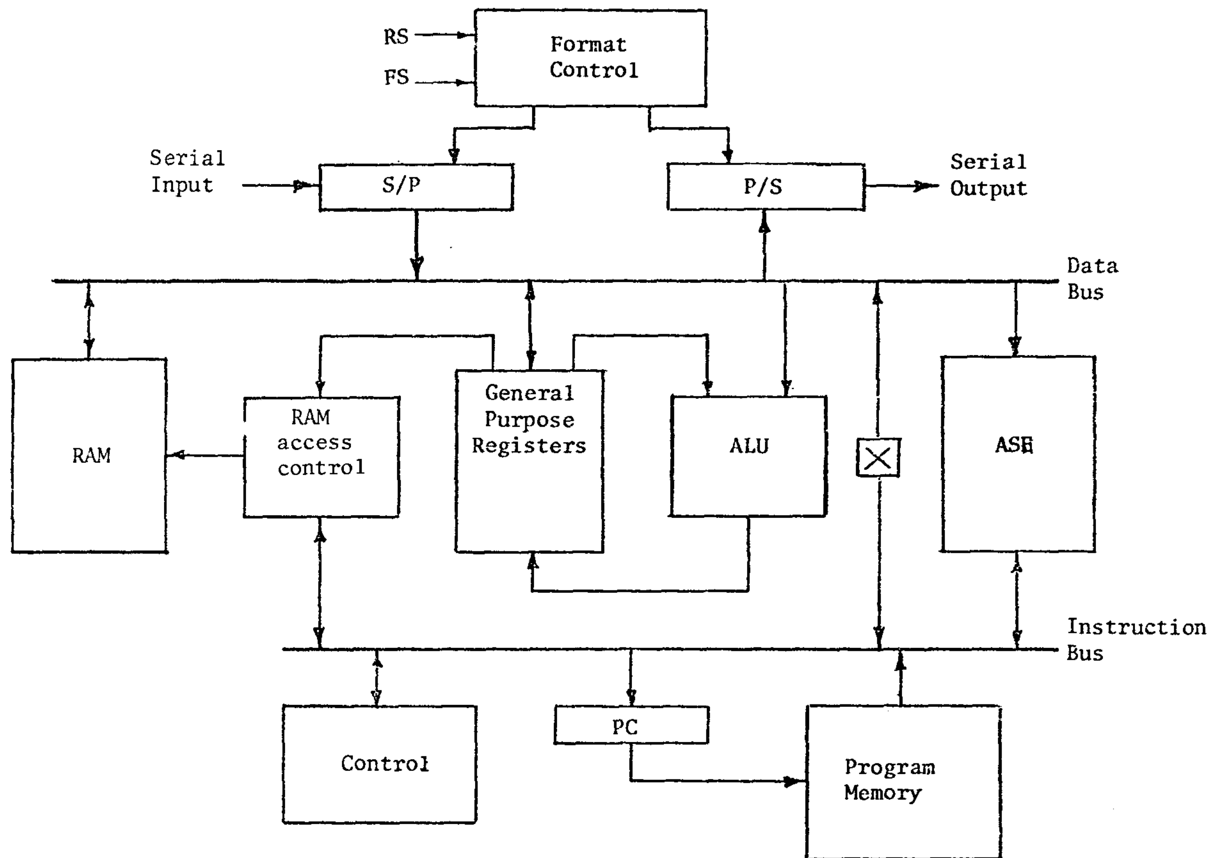


Fig. 9 A processor organization using an Associative Search Element and an intelligent RAM access controller.

3. Two searches proceed in parallel, one for the string STRING and one for DLR.

The third observation points to an interesting possibility. Since all the parameters related to a search operation are stored within the ASE, the control unit is free to initiate other operations at other parts of the processor. These are operations on the same data stream, and may proceed in parallel with the search in the ASE. The two most important operations that can be handled in this way are the RAM control and the I/O format control. As mentioned earlier, the RAM control provides automatic access to a number of commonly used data structures such as queues and stacks. The I/O format control enables such operations as identification of record and field boundaries. (Both control blocks may request execution of service routines in the program memory in the same way as in the case of the ASE.) Details of these operations are still under investigation.

Handling of Multiple Requests for Service

There is one resource in the processor of Fig. 9 that is shared between all the operations that may be in progress at any given time. This is the program memory. Therefore, a scheme is required to resolve multiple requests for instructions. This is accomplished by assigning a programmable priority to each element that is capable of independent operation, such as the ASE and the RAM control. This priority is used only for determining the order of servicing multiple requests. However, a high priority request cannot interrupt servicing of a low priority request which is already in progress. Such interruption would introduce unnecessary complexity in the system. Since the timing constraints for all processes originate in the input data stream, there is no gain in interrupting one process to service another. Furthermore, the service routines are usually very short. Hence, the time overhead that would result from allowing interruption may easily exceed the time required to complete execution of the service routine in progress.

7. Conclusions

When von Neumann introduced an architecture for a stored program computer, where instructions and data reside in a random access memory to be accessed in an arbitrary sequence defined by the instructions, it represented an ingenious concept for the organization of hardware to handle a general, undefined program. The flexibility and power of this concept are demonstrated by the fact that it has remained essentially unchanged for over 30 years. However, over this long period of time, our understanding of problem solving on computers has increased considerably. Data items that reside in a

random access memory of a computer are, in fact, stored in well defined structures. However, these structures are defined exclusively by the software.

In this paper, an attempt has been made to define a general purpose hardware organization for the non-numeric processing environment. In order to retain the flexibility of a conventional machine, instructions and data are stored in random access memories. However, instruction sequencing is simplified by introducing the concept of a data driven machine. Also, extra intelligence is incorporated in the memory access and I/O control circuitry to take advantage of the program and data structures.

There are two main advantages to this approach, in comparison with a more conventional architecture. First, the processing capability is increased, because of the reduction in the number of instruction fetches during execution, and the increased parallelism. Secondly, it results in machine instructions that are much more problem oriented. Evolution of such "structured hardware" may eventually lead to a general purpose machine that is directly programmable in a high level language.

8. References

1. Amello, G.F., "Charge-coupled devices for memory applications", AFIPS Conf. Proc., vol. 44, pp. 515-522, May 1975.

2. Ypma, J.E., "Bubble domain memory systems", AFIPS Conf. Proc., vol. 44, pp. 523-528, May 1975.

3. Gillis, A.K. et al., "Holographic memories - Fantasy or reality?" AFIPS Conf. Proc., vol. 44, pp. 535-539, May 1975.

4. Hughes, W.C. et al., "BEAMOS, A new electronic digital memory", AFIPS Conf. Proc., vol. 44, pp. 541-548, May 1975.

5. Pitchard, J.P. and Wald, L.D., "Design of a fully associative cryogenic data processor", IEEE Trans. Magn., vol. MAG-1, pp. 68-71, March 1965.

6. Berra, P.B., "Some problems in associative processor applications to data base management", AFIPS Conf. Proc., vol. 43, pp. 1-4, May 1974.

7. Slotnick, D.L. "Logic per track devices", in Advances in Computers, vol. 10, New York: Academic Press, 1970, pp. 291-296.

8. Parhami, B., "Associative Memories and Processors - An Overview and Selected Bibliography", Proc. IEEE, vol. 61, No. 6, pp. 722-730, June 1973.

9. De Martinis, M., Lipovski, G.J. et al., "A Self-managing Secondary Storage Memory System", The 3rd Annual Symp. on Comp. Arch., Conf. Proc., pp. 186-194, January 1976.

10. Pzkarahan, E.A., Schuster, S.A. and Smith, K.C., "RAP - An associative processor for data base management", AFIPS Conf. Proc., vol. 44, pp. 379-387, May 1975.

11. Vranesic, Z.G. and Zaky, S.G. "Non-numeric Appl Applications of Microprocessors", Proc. IEEE, vol. 64, No. 6, pp. 954-959, June 1976.