IMPLEMENTATION OF
CHARACTER STRING PATTERN MATCHING
ON A MULTIPROCESSOR*

Maniel Vineberg
Naval Ocean Systems Center
San Diego, California 92152

ABSTRACT

An algorithm to do pattern matching, a basic character string operation, is presented. The Programmable Algorithm Machine (PAM), a proposed special-purpose computer which will feature multiple processing elements and operate efficiently over a wide class of applications, is described. It is shown that the multiple processing elements of the PAM allow concurrent execution of independent operations both in a special case of the pattern matching algorithm, where the string sizes (lengths) are known at compile time, and in the general case, where the sizes are not known.

1. INTRODUCTION

Character string processing is currently of wide interest, finding application in such areas as language translation, information storage and retrieval, text editing, and message processing. Many high-level languages now offer limited string processing features [1, 2]. A language which offers extensive character string processing features, SNOBOL [3], has been implemented on existing machines [4, 5]. The inefficiency of these implementations, due to the mismatch between the SNOBOL language features and the facilities available on conventional machines, motivated research to design a SNOBOL Machine [6].

The efforts to implement SNOBOL are interesting with respect to SNOBOL itself. However, a more modest character string processing capability, implemented efficiently, would be of value in those environments where SNOBOL is not the programming language. The approach taken here is 1) to define a fundamental character string operation, pattern matching, independent of any programming language, 2) to propose an algorithm to perform that operation, independent of any existing machine, and 3) to show that the Programmable Algorithm Machine (PAM) [7], a multiprocessor proposed for use over a wide class of applications, could perform the algorithm efficiently.

*This research was supported by the Naval Electronics Systems Command under Project No. 720.

2. CHARACTER STRING OPERATIONS

Several operations commonly performed on character strings are concatenation, substring extraction, conversion, pattern matching, and replacement. Concatenation is the joining of 2 strings to form a third; for example, concatenating the strings (enclosed in single quotes) 'A HEAVY ' and 'LOAD' gives 'A HEAVY LOAD'. Substring extraction is the forming of a new string from a continuous part of an existing string; for example, 'AVY LO' is a substring of 'A HEAVY LOAD'. Conversion is an operation upon a string to produce a numerical value. An example of conversion is string size (length); the size of 'A HEAVY LOAD' is 12 (characters). Pattern matching is the detection of the occurrence of one string (or pattern) within another; for example, the pattern 'HEAV' occurs within 'A HEAVY LOAD'. Pattern matching may be followed by replacement, the substitution of one pattern in a string for another; for example, 'HEAV' may be replaced by 'FUNN' in 'A HEAVY LOAD' to yield 'A FUNNY LOAD'

Of the operations above, only conversion and pattern matching require extensive computation; the others rely on storage and retrieval conventions. Even conversion (for example the computation of character string size) may depend more on string header format than on any computation. By contrast, pattern matching generally requires extensive processing, characterized by a large number of comparisons. For this reason, pattern matching, which places a heavy burden on a sequential processor, will profit from implementation on a multiprocessor.

3. A PATTERN MATCHING ALGORITHM

A pattern matching algorithm is

developed here, independently of any implementation constraints. The independent operations within the algorithm are identified to make a case for a highly concurrent implementation. Iverson's notation [8] is used; comments are included where the notation is not self explanatory.

3.1 DEFINITION

The operation of pattern matching determines if and where a given character string, P (the pattern), occurs as a continuous substring within a second given character string, S (the sample). It will be assumed that $0 < \rho P < \rho S$ (ρP is the size of P), since otherwise no match can be made. For example, given P = 'ISSI' and S = 'MISSISSIPPI', the pattern matching operation locates P in S twice, as indicated by the underscored characters in the following: MISSISSIPPI and MISSISSIPPI.

3.2 THE ALGORITHM

An algorithm to match pattern P to sample S, where $0 < \rho P \leq \rho S$, is now given, followed by a detailed explanation and an example.

```
∇ R ← P MATCH S

[1] C ← 0
[2] →3 x (1 + (C = ρP))
[3] C ← C + 1
[4] T[C;] ← (C - 1)↓(P[C] = S),
            (C - 1) ρ 0
[5] →2
[6] R ← ∧/T
[7] M ← ∨/R
[8] N ← +/R
∇
```

The algorithm, MATCH, accepts inputs P and S and produces a result vector R. Steps are sequential unless a branch operator (→) is specified. Step [1] initializes counter C to 0. Step [2] branches to step [3] if $C < \rho P$ (ie, $(C = \rho P) = 0$) or to step [6] if $C = \rho P$ (ie, $(C = \rho P) = 1$). Step [3] increments C. Step [4] compares the Cth character in P to each character in S to obtain a vector (P[C] = S) the size of S, consisting of '1's where matches were made and '0's where matches were not made; this vector is shifted left (↓) C - 1 positions, with '0's being entered into the rightmost C - 1 positions (ρ is a repetition operator), and is recorded in row C of table T. Step [5] returns to step 2. Step [6] computes the intersection over the rows (vectors) of T and records the result in R; a '1' in any position R[k] indicates that pattern P occurs in S beginning at S[k]. Steps [7] and [8] assign an indicator of the presence (1) or absence (0) of a match to M, and the number of matches to N.

Algorithm MATCH is now used to perform the example of Section 3.1.

```
∇ R ← 'ISSI'  MATCH  'MISSISSIPPI'

MATCH[1]  C ← 0
MATCH[2]  →3 x (1 + (C = 4))
          →3 x (1 + 0)
          →3
MATCH[3]  C ← 1
MATCH[4]  T[1;] ← 0↓('I' =
                  'MISSISSIPPI'), 0 ρ 0
                ← 01001001001
MATCH[5]  →2
MATCH[2]  →3
MATCH[3]  C ← 2
MATCH[4]  T[2;] ← 1↓('S' =
                  'MISSISSIPPI'), 1 ρ 0
                ← 1↓00110110000,0
                ← 01101100000
MATCH[5]  →2
MATCH[2]  →3
MATCH[3]  C ← 3
MATCH[4]  T[3;] ← 11011000000
MATCH[5]  →3
MATCH[3]  C ← 4
MATCH[4]  T[4;] ← 01001001000
MATCH[5]  →2
MATCH[2]  →3 x (1 + (C = 4))
          →3 x (1 + 1)
          6
MATCH[6]  R ← ∧/T[1;] T[2;] T[3;] T[4;]
            ← ∧/01001001001  01101100000
              11011000000  01001001000
            ← 01001000000
MATCH[7]  M ← ∨/01001000000
            ← 1
MATCH[8]  N ← +/01001000000
            ← 2
∇
```

MATCH halts with R corresponding to the result indicated in Section 3.1. M = 1 confirms that at least one match was made. N = 2 shows that a total of two matches were made.

3.3 ADVANTAGES

The algorithm, although presented sequentially, has an important advantage: all comparisons (step [4]) are independent and can therefore be made concurrently. As expected, the algorithm determines all matches (even those which overlap as in the above example).

4. THE PROGRAMMABLE ALGORITHM MACHINE (PAM)

The PAM, introduced in [7], has been proposed as a dedicated special-purpose computer. The PAM will include a processor composed of multiple processing elements, separate instruction and operand memories, and instruction pipelining. It is designed to execute efficiently over a class of algorithms that exhibit (1) a high frequency of independent operations and (2) a low frequency of branching. The first property offers operations whose order of execution is arbitrary and which therefore can be allocated efficiently to the PAM processor and maybe executed concurrently. The second property

offers program statements which are intended to follow one another in sequence and which may therefore be allocated to the PAM processor in large blocks, to take best advantage of processing elements. Arithmetic algorithms, signal processing, array processing, and character string manipulation are examples of applications which exhibit these properties.

The PAM represents a family of machines. Each PAM version (family member) is characterized by a version processor, including a number of processing elements defined in terms of the operators (e.g., "+", "-", etc.) that each accepts and the times required for the execution of the respective operations.

The PAM will normally be programmed in an algorithmic language (e.g., a subset of ALGOL). The PAM will also be programmable in the PAM Assembly Language (PAL). PAL combines postfix assignments with sequencing information in a single statement format. This representation is convenient for exploiting the PAM architecture and as an intermediate representation for the algorithmic language.

The PAM will be supported by a unified software system. A compiler will translate algorithmic language programs into PAL. A parameterized assembler, capable of accepting a PAM version specification and a PAL program, will produce code executable by the specified PAM version. A parameterized simulation will allow translated programs to be executed and verified on a specified version of the PAM.

The PAM is discussed in four parts: 1) PAM executable programs; 2) PAM operation; 3) the PAM processor; 4) PAM languages and support software.

4.1 PAM EXECUTABLE PROGRAMS

The PAM is designed to execute a program composed of nets. A net (Figure 1) is a collection of operations. An operation (Figure 1) is a transformation on two input operands (i_1 and i_2) by an operator (o) to yield an output operand (i_1 o i_2). An input operand may be known prior to the processing of a net ($i_{1,b}$) or it may be the output of another operation within the net ($i_{2,c}$). Similarly, an output operand may be required as the input to one or more operations within the net (output of operation b) and it may also be required outside the net (output of operation a).

An output operand may also be used to modify the successor condition. The successor condition is "true" before a net is processed and may be either "true" or "false" when the net processing is complete. Each net may be accompanied by the specification of two possible successor nets, a true and a false successor. Once a net has been processed, the successor condition will indicate whether the true successor or the false successor is to be processed next. If the successor condition indicates an unspecified successor (e.g., the successor condition is "false" but no false successor is specified) then the program will terminate.

Nets are indivisible program entities. A net must be completely processed before a successor is initiated.

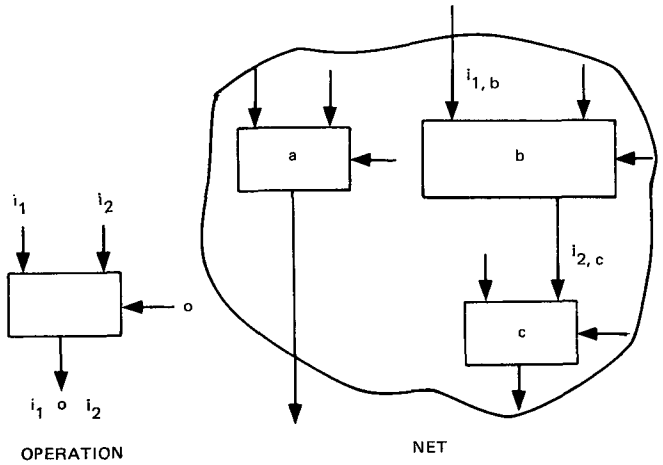

Figure 1. Operation and Net

4.2 PAM OPERATION

PAM operation will be discussed with reference to the block diagram shown in Figure 2.

Programs will be stored in two parts. Nets, consisting of operation specifications - operators, operand and subscript addresses, etc. - and successor net identities, will reside in the instruction memory. Operand values will reside in the operand memory.

A controller will manage PAM operation in load/execute cycles. During a cycle, a net will be executed from processor registers by the processor. The processor may return outputs, via processor (output) registers, to the operand memory. The processor may also update the successor condition bit (initially "true"). During the same cycle, the true and false successor nets will be loaded from the instruction and operand memories into true and false load registers.

A cycle will be complete when both loading and execution are complete. Then, depending on the value of the successor condition bit, the processor registers will be loaded from true or false load registers and a new cycle may begin. During the first cycle of a program, only loading will take place.

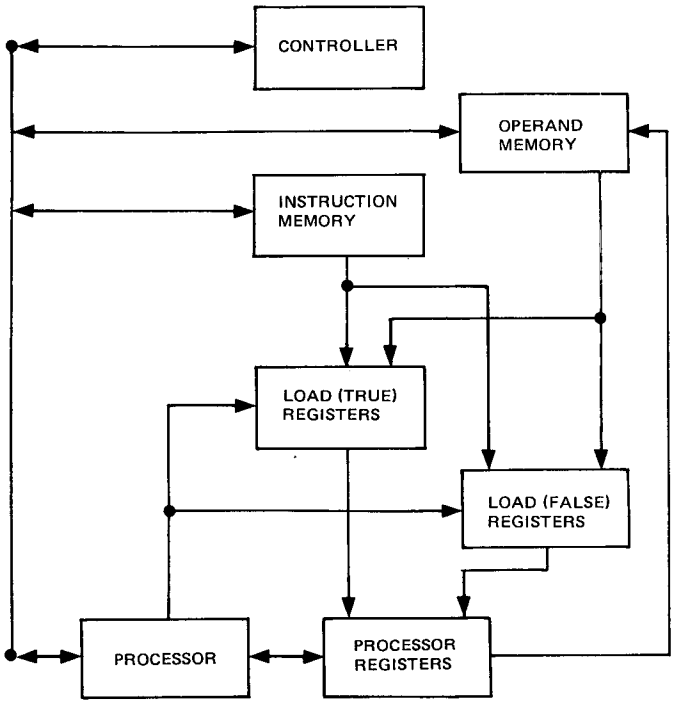

Figure 2. PAM block diagram.

4.3 THE PROCESSOR

The heart of the PAM is the processor. The processor comprises a number of processing elements which are controlled by information in the processor registers. During a cycle, the processor (1) executes the net specified in the processor registers, (2) returns net outputs to processor output registers, and (3) transmits outputs to load registers via a data bus.

Figure 3 shows a processing element, the accompanying processor registers, the data bus, a counter, and an ID register. Processor registers include two input operand registers, an operator register, an output operand register, and an output address register. The operator register includes an operation code field, used to determine the processing element algorithm (i.e., +, -, etc.), and a time field, used to initialize the processing element counter. The input operand registers include 1) a data field, 2) a status field, to indicate the presence or absence of a valid operand in the data field, and 3) a tag field, to gate information from the data bus into the data field.

Processing element operation begins when the required input operands are present (indicated by S_1 and S_2). The operation is timed by the counter. When the operation is complete, the counter gates the ID register (a unique processing element identifier) onto the tag part of the data bus and gates the processing element output onto the data part of the data bus. The output will be received by any processor or load registers which hold a tag matching the tag on the data bus. The output is also loaded into the output operand register, which, along with the output address register, may be used to update the operand memory.

The support software will compute the processing element operation times, introducing delay as required, to prevent data bus transmission conflicts. It is also the support software which creates nets such that operations are allocated only to processing elements capable of performing them.

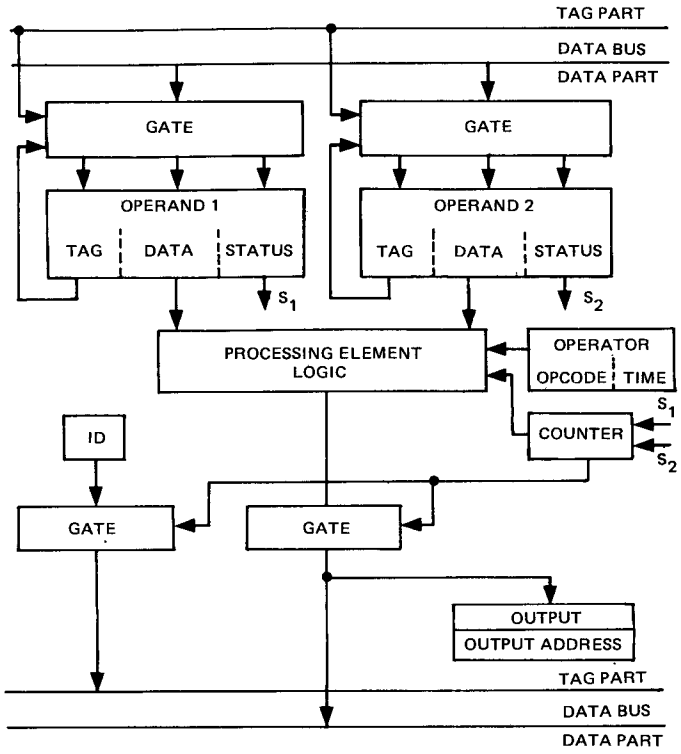

Figure 3. Processing element and processor registers.

4.4 PAM LANGUAGES AND SUPPORT SOFTWARE

The PAM will normally be programmed in an algorithmic language. Algorithmic language programs will be translated by a compiler into PAL. The PAM will also be programmable in PAL, a language based on postfix assignment statements.

The PAM software support system including the compiler, the assembler, and the simulation, is diagrammed in Figure 4.

The compiler will accept an algorithmic language program and will produce a PAL program. The assembler will accept

a PAL program and a PAM version specification and will produce code executable by that version of the PAM. The simulation will accept a version specification and code assembled for that version and will produce program results and performance measurements. The support software may be used both to verify programs and to compare the performance of various PAM versions before assembling actual PAM hardware.

For this discussion, it will be helpful to understand the operation of the assembler. The assembler (part I) accepts a PAL program and maps individual PAL instructions into parse trees. Next, the assembler (part II) accepts a version specification and forms nets, partitioning parse trees where required and combining parse trees where possible. Nets do not in general carry a one-to-one correspondence to parse trees. The assembler will allocate processing elements to parse tree operations as efficiently as possible, under the constraint that no transfer of program control may be made into or out from the middle of a net.

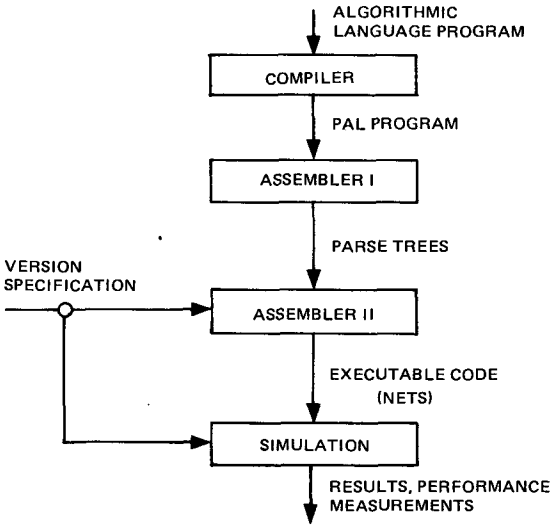

Figure 4. PAM software support system.

5. IMPLEMENTING PATTERN MATCHING ON PAM

The algorithm of Section 3.2 can be implemented efficiently on the PAM. To demonstrate this, a special case of that algorithm - namely the case where the sizes of both the pattern and the sample are known at compile time - will be described. A compiler generated pattern matching routine for the special case will be proposed; typical PAL code and resulting parse trees for the example of Section 3.1 will be displayed. The extension of the routine to the general case will also be discussed.

5.1 COMPILER GENERATED ROUTINE

Without entering into the details of the algorithmic language, it will be assumed that the PAM compiler can recognize a requirement to encode (in PAL) a pattern matching routine such as the following: given pattern P of size ρP and sample S of size ρS, find all occurrences of P in S as described in Section 3.2. The fact that the string sizes, ρP and ρS, are known to the compiler will allow it to create an optimized routine.

The compiler first checks ρP and ρS. If $\rho P = 0$ or if $\rho P > \rho S$, no match can be made and therefore no routine is generated.

If $0 < \rho P \leq \rho S$, then the compiler generates statements to compute the result vector R as follows:

$$R[X] \leftarrow (P[1] = S[X]) \wedge (P[2] = S[X+1]) \wedge \ldots \wedge (P[\rho P] = S[X+\rho P-1])$$

where $X = 1, 2, \ldots, \rho S + 1 - \rho P$

These equations are a reordering of steps [4] and [6] from Section 3.2, omitting the temporary results T. Note that since the sizes are known in advance, certain unnecessary comparisons ($P[Y] = S[X]$, where $X < Y$ or $X > \rho S + Y - \rho P$) can be eliminated; this is possible since a substring in S which matches P will never originate before S[1] or terminate after $S[\rho S]$. This optimization also yields the above condition that R is a vector having $\rho S + 1 - \rho P$ elements (note that for $\rho S = \rho P$, X may only be 1, ie, a matching substring may only begin in S[1]; for $\rho S = \rho P + 1$, X may be 1 or 2, ie, a matching substring may begin in S[1] or in S[2]; etc.).

5.2 A SPECIFIC EXAMPLE

The following is a PAL encoding which would be generated by the compiler routine described in the previous section. The example, from Section 3.1, would include the following declarations (control words *DEC and *CED delimit the PAL program declaration section):

```
*DEC ;
   VAR' S(11) CHAR 'MISSISSIPPI'
   VAR' P(4) CHAR 'ISSI'
*CED ;
```

The two character strings, S and P, have sizes 11 and 4 respectively, which satisfy the condition $0 < \rho P \leq \rho S$. Note that since $\rho S + 1 - \rho P = 8$, R will be a vector having 8 elements (no matching substring can begin beyond S[8]). The postfix instructions follow (*INS and *SNI delimit the PAL program instruction section):

```
*INS ;
   R(1)←P(1) S(1) = P(2) S(2) = ∧
        P(3) S(3) = P(4) S(4) = ∧ ∧
   R(2)←P(1) S(2) = P(2) S(3) = ∧
        P(3) S(4) = P(4) S(5) = ∧ ∧
.
.
.
   R(8)←P(1) S(8) = P(2) S(9) = ∧
        P(3) S(10) = P(4) S(11) = ∧ ∧
*SNI ;
```

The parse tree for R(1) is shown in Figure 5 (subscripts are used for simplicity) along with individual results.

Two observations can be made regarding Figure 5: 1) since this parse tree includes several independent operations, and since all 8 parse trees are identical in form and independent of each other, the operations within the resulting nets will include many operations to be executed concurrently; 2) a version of the PAM configured to be used extensively in pattern matching should include a large number of processing elements which implement the operations "=" and "∧".

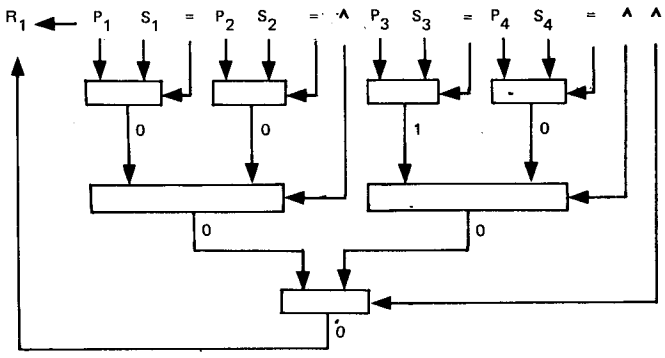

Figure 5. Matching P ('ISSI') to S(1:4) ('MISS').

5.3 THE GENERAL CASE

In general, when occurrences of pattern P are to be found in sample S, ρP and/or ρS are not known at compile time. In order to implement this general case of algorithm MATCH (Section 3.2), it must be possible 1) to access ρP and ρS and 2) to efficiently access P and S, or substrings of P and S, for repeated operations as delimited by ρP and ρS and by the number of available processing elements.

ρP and ρS can be stored with each character string in an accompanying descriptor (described by Shapiro [6]), thereby making string sizes available in the PAM operand memory. In addition to the string size, a descriptor includes a pointer to the first character in the string and an (optional) offset indicating where a specific substring begins. By modifying the offset, a separate descriptor can be created for each matching substring of a given sample, thereby providing a natural vehicle for retaining the results of a pattern matching operation.

Implicit in the above representation is the idea that each string is stored in a contiguous block of memory. For the PAM, this will necessitate the imposition of upper bounds for string sizes since, at present, there are no plans to incorporate a dynamic allocation capability. This restriction will manifest itself in a tradeoff between maximum string sizes (which can be specified at compile time for each string) and the number of strings which can be represented for a given operand memory size.

The general implementation of the algorithm, MATCH, unlike the special case of the previous sections, requires iteration over the pattern size, ρP, because in general ρP is not known at compile time. Note that each case will require iteration due to any mismatch between the sample size, ρS, and the number of available processing elements. In addition, due to the iterative nature of the computation, implementation of the general case will normally be unable to exploit the available processing elements as efficiently as the special case; this is true since computations which are not strictly sequential (eg, a comparison and an iteration, separated by a conditional branch) cannot be combined in a single net.

Despite these limitations, the general implementation of MATCH on the PAM will be efficient for two reasons: 1) the comparisons (step [4]) and subsequent reductions (steps [6], [7], and [8] will benefit from multiple processing elements, just as in the special case, and 2) the PAM allows pipelining through load registers thereby obviating the need for any "special" registers without compromising the efficiency of the algorithm.

6. SUMMARY AND CONCLUSIONS

An algorithm to do character string pattern matching has been presented; the algorithm is characterized by a large number of independent operations. It has been shown that the Programmable Algorithm Machine (PAM) will provide an excellent vehicle for the implementation of this algorithm. The multiple processing elements, which comprise the PAM processor, allow concurrent execution of independent oprations both in a special case of the algorithm, where the string sizes are known at compile time, and in the general case, where the sizes are not known.

The PAM is expected to add another dimension to programmable machine processing capability, allowing the rapid configuration and check-out of special-purpose computers to perform a variety of time-critical tasks, thereby offering the possibility for decreased development times and improved performance in a broad range of systems.

The PAM software support system, currently being specified for implementation on a large host computer, will be the focus of attention for the immediate future. Once that software is operational, such applications as character string pattern matching can be implemented, refined, and compared to other implementations. The PAM architecture can then be evaluated and steps leading to the actual construction of PAM hardware can be taken.

ACKNOWLEGEMENTS

The author is pleased to acknowledge Russ Eyres of NOSC, who continues to support and believe in the PAM project, and Jim Barksdale, Neal Hampton and Tricia Santoni of NOSC for their helpful comments.

REFERENCES

1. Sperry Univac 1100 Series, PL/1 Programmer Reference, UP-8277, Sperry Univac Computer Systems, 1975.

2. Sperry Univac 1100 Series, FORTRAN (ASCII), Programmer Reference, UP-8244, Sperry Univac Computer Systems, 1976.

3. Griswold, R. E., Poage, J. F., and Polonsky, I. P., The SNOBOL4 Programming Language, Second Edition, Prentice-Hall, 1971.

4. Griswold, R. E., The MACRO Implementation of SNOBOL4, Freeman, 1972.

5. Santos, P. J., FASBOL, A SNOBOL4 Compiler, Ph.D. Thesis, University of California, Berkeley, 1971.

6. Shapiro, M. D., A SNOBOL Machine: Functional Architectural Concepts of a String Processor, Ph.D. Thesis, Purdue University, 1972.

7. Vineberg, M., The Programmable Algorithm Machine (PAM): Part 1 - Theory, Part 2 - Applications, NELC/TD 478, Naval Electronics Laboratory Center, San Diego, 4 June 1976.

8. Iverson, K. G., Algebra: An Algorithmic Treatment, Addison-Wesley, Menlo Park, California, 1972.