INTRODUCTION

In recent years, a number of hardware associative memories had been designed and experimentally verified.\(^1,2\) These memories allow simultaneous comparison of all stored data to external data. Data may be read from, or written into, comparing words. These memories, acting as peripheral devices to conventional computers, have been studied for application to various tasks described in references 1 and 2. The concept of "associative processing," i.e., simultaneous transformation of many stored data by associative means, has been described previously.\(^3,4,5\) This processing mode showed promise in a variety of tasks, but was not efficient when peripherally controlled by a conventional machine. Novel machine organizations were required to fully exploit the potential of these techniques for solving poorly structured nonnumeric problems, at which present-day machines are not efficient.

This paper describes a novel Associative Parallel Processor (APP), having an associative memory as an integral part of the machine. Arithmetic algorithms are described which allow it to perform adaptive pattern recognition by evaluating threshold logic functions. Novel algorithms allow simultaneous processing of many operands in bit-parallel fashion. The processor is a stored program device with a powerful command set, and thus has general utility in problems which allow a single set of commands to be executed independently, and thus simultaneously over many data sets. These conditions frequently arise in nonnumeric data processing tasks such as pattern recognition.

Parallel processing is accomplished within the associative array of APP by the powerful technique of "sequential-state-transformation," previously described by one of the authors.\(^3\) The parallel search function of associative memories requires that comparison logic be provided at each memory word cell. The APP, by moderate additions to this logic, allows the contents of many cells, selected on the basis of their initial content, to be modified simultaneously through a "multiwrite" operation. Content search and multiwrite are the primitive operations necessary to parallel processing by sequential-state-transformation.

\(^*\)The work reported here was supported by AF Rome Air Development Center, Griffiss Air Force Base, N.Y., under Contract AF 33 (602)-3371.
To illustrate the concept of sequential-state-transformation, consider an associative memory which stores two operands, \( A_i \) and \( B_i \), in each word of memory. We desire to add operand \( A_i \) to operand \( B_i \) simultaneously in some subset of these words. Processing is serial by bit, and parallel by word, starting at the least significant bit of each field. Each word has an auxiliary storage bit, \( C_i \) stored within the memory array. Bits within operand field \( A_i \) are designated \( A_{ij} \) \( (j=1,2,\ldots,N) \), where \( N \) is the field length. Bits in field \( B_i \) are similarly designated. The truth table defining the addition is as follows:

<table>
<thead>
<tr>
<th>State</th>
<th>Number</th>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that variables \( B_{ij} \) and \( C_i \) differ in their present and next states only in states numbered 2, 4, 5, 7.

The search and multiwrite operations may be used to perform the addition of all number pairs, starting at the least significant bit, as follows:

1. Search words having \( A_{ij} = 1, B_{ij} = 1 \) and \( C_i = 0 \).
   For these words multiwrite \( B_{ij} = 0, C_i = 1 \).
2. Search words having \( A_{ij} = 0, B_{ij} = 0 \) and \( C_i = 1 \).
   For these words multiwrite \( B_{ij} = 1, C_i = 0 \).
3. Search words having \( A_{ij} = 0, B_{ij} = 1 \) and \( C_i = 1 \).
4. Search words have \( A_{ij} = 1, B_{ij} = 0 \) and \( C_i = 0 \).
   For these words multiwrite \( B_{ij} = 1 \).

Steps (1) through (4) are repeated at each bit of the operands. Within each bit time, processing is sequential by state over present states which differ from the next state in one or more variables. All words in a given present state are transformed simultaneously to the desired next state.

Sequential-state-transformations used to perform the above word-parallel, bit-serial addition, is evidently a very general mode of associative processing. It allows transformation of memory contents according to any Boolean function of stored and external variables. It makes full use of comparison logic, implemented at the bit level within an associative array, and thereby simplifies logic required at the word level. It compares favorably with other associative processing methods in both speed and processor complexity.

In the next section we describe the organization and command set for a processor using the sequential-state-transformation mode of associative processing. In the following section command routines are given for word-parallel, bit-serial processing described above; and also for a novel mode of word-parallel, bit-parallel processing which yields significant speed improvement over bit-serial modes. The pattern-processing applications is discussed last.

**ORGANIZATION AND COMMAND SET**

The addition operation presented in the preceding section is a typical example of the associative processing technique. From it, several conclusions concerning the desired structure for an associative processor can be formulated.

1. The single primitive step in associative processing is identification of all words storing some configuration of binary state variables, followed by binary complementation of some state variables within identified words. This primitive makes the basis of an associative "micro instruction" which, repeatedly executed with varying parameters, can transform memory contents according to any Boolean function of stored and external binary variables.

2. Since processing is simultaneous over all stored data, no explicit word address is provided within an associative instruction. Many words may be transformed in response to a given instruction. These words are identified by search criteria contained within the instruction.

3. Data is processed column-serially to minimize the number of state variables and thus memory states which must be identified and transformed sequentially. Associative micro instructions thus address a small number of bit columns in memory. Since many consecutive bit columns are sequen-
Initially transformed, efficient means for column indexing are required.

4. Several temporary storage or “tag” bits within each word are useful to identify words as members of various sets currently undergoing transformation. The carry storage bit, defined for the addition task of the previous subsection, is a tag bit. The location of tag columns are unchanged as successive data columns are processed.

5. Each word cell must have electronics, external to the memory array, which temporarily store the match status of the word, relative to the most recent search criteria, and allow writing of selected bits in matching words to either the one or zero state. Writing is simultaneous over all matching words.

6. For generality, stored program control of the associative processor is desired. Instructions are accessed sequentially from control memory, with possible branching as in conventional machines. Since no benefit derives from storing these instructions in associative memory, the control memory is a less costly location-addressed random-access memory. Having separate instruction and data memories, the access times for each may be overlapped.

Elements of the Librascope processor are shown in Fig. 1. This realization contains an associative array, partitioned into data and tag columns (fields), together with requisite word and digit electronics. Instructions are read from a random-access control memory, and are interpreted by control logic to sequence operations within the processor. The associative array may be loaded word-serially through the data register and unloaded column-serially through word electronics. The data register also stores search keys. The random-access memory is written or read through the data register. Contents of word flip-flops (i.e., match status of words) may be transferred to the data register for subsequent use as a search key or to be output. Column indexing, over two distinct fields (A and B), is provided by the A and B counters. A and B limit registers store either the upper or the lower limit column for their respective fields.

The electronics repeated at each word cell (Fig. 2) contains a match storage flip-flop and a word driver capable of writing ones or zeros into selected bits of matching words. The state of each match flip-flop is fed to a “match status gate” whose output denotes the presence or absence of matching words. The group of match flip-flops, termed a detector plane (DP), may be set collectively to one or zero by signals $E_{1}$ or $E_{0}$ common to all match flip-flops. Prior to writing into lower neighbors, contents of all match flip-flops are shifted down one word position.

Operation of the processor required the following steps:

1. Load instructions to control memory.
2. Load data to associative array.
3. Process data according to state transforming instructions read from the control memory.
4. Output results.

Instructions are fed to the data register then written into the control memory by action of the central control. Data are likewise fed to the data register then written into the associative array by data and word drivers. The associative array is content-addressed rather than location-addressed. Prior to writing word zero, the match flip-flop for this word is set to one. The first data element is written into this “matching” word. Subsequent words are written after shifting this one into consecutive match flip-flops.

Data within the associative array are processed by use of the associative command described below. Following associative processing the output data may be a single quantity (e.g., a picture “name”) or may be the entire transformed contents of the associative array (e.g., field potentials at all nodes in a discretized solution space. Either class of data may be efficiently outputed through the match flip-flops. The contents of a column within the associative array are read to match flip-flops by a search on that column then transferred to the data register for output. The alternative of reading words sequentially from the associative array requires the availability of bit sense amplifiers and word-read drivers together with means for addressing word-read drivers. The latter readout scheme would be more costly in hardware with no gain in efficiency.

The format for associative commands is shown in Fig. 3. Each associative command effects a primitive transformation of state variables as discussed in the preceding section. The left-most bit identifies
the command as associative. The two adjacent bits define the initial state of match flip-flops in word logic units (i.e., the detector plane). Other bits define search and rewrite criteria for the \( A \) field, the \( B \) field, and for each of four tag bits. The right-most bit controls rewrite into matching words or their next lower neighbors. Functions of these bits are described in Fig. 3.

To illustrate the utility of this command, consider the task of searching the associative memory for words matching the data register over a field having its upper limit stored in the \( A \) limit register and its lower limit stored in the \( A \) counter. Matching words are to be tagged in tag bit 1.

The following command accomplishes the desired tasks:

\[
E_{\text{En}} S \quad L \quad D \quad W \quad I \quad S \quad - \quad W \quad - \quad S \quad W
\]

A Control B Control Tag 1

The following routine loads data into each word in the associative array. The word field to be writ-
AN ASSOCIATIVE PARALLEL PROCESSOR

Figure 2. Word electronics.

ten is again defined by contents of the $A$ counter and the $A$ limit register:

1. Set the match flip flop for word 0 to "1."
2. I N S L D W S -- -- S - W N
3. I N S L D W S -- -- S - W L

A Control B Control Tag 1
4. If not match, exit: otherwise go to (3).

Instruction (2) writes into word 0; instruction (3) writes sequentially into each remaining associative word.

Nonassociative commands are provided to load the $A$ and $B$ counters and limit registers, to branch from linear instruction sequencing either unconditionally or when specified conditions are met, and to input or output data. Nonassociative commands are specifically defined in the illustrative programs presented in the next section.

ARITHMETIC ALGORITHMS

In previous parallel processors arithmetic algorithms were typically executed over many operands simultaneously, but in bit-serial fashion. All bits of an operand are stored within a single word cell as shown in Fig. 4. For operations requiring two operands, operands may be paired by storing them in the same cell or by restricted communication between word cells (e.g., communication between "nearest neighbors" only). This word-per-cell ($W/C$) organization is efficient when all or most operands in memory are processed by each associative command.

An alternate data organization stores bits of an operand in separate contiguous word cells as shown in Fig. 5. A similar organization, independently derived, was recently described. The bit-per-cell ($B/C$) organization allows many operands to be processed simultaneously in bit-parallel fashion. Command execution is thus appreciably faster than for the $W/C$ organization, but each command is typically executed over fewer operand pairs. Any operands stored in the same set of contiguous word cells may be simply paired. The $B/C$ organization is thus efficient for problems which do not allow simultaneous processing of all operands by a single command, and for problems in which each
operand must be paired with many others at various computational steps.

The processor organization and command set described in the preceding section is equally applicable to $W/C$ and $B/C$ data organizations. Arithmetic algorithms, appropriate to each data organization, are presented below.

Consider first an algorithm for the $W/C$ data organization (Fig. 4), which adds contents of all $A$ fields to contents of respective $B$ fields, leaving the resulting sum in the $B$ fields. Tag 1 is used for carry storage and is assumed initially cleared. The routine is as follows:

---

### Figure 3. Format for associative command.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>Clear DP to zero</td>
</tr>
<tr>
<td>E1</td>
<td>Clear DP to one</td>
</tr>
<tr>
<td>N</td>
<td>No change in DP</td>
</tr>
<tr>
<td>W</td>
<td>Write</td>
</tr>
<tr>
<td>W'</td>
<td>Write lower neighbor</td>
</tr>
<tr>
<td>S</td>
<td>Single counter increment</td>
</tr>
<tr>
<td>S'</td>
<td>Don't search</td>
</tr>
</tbody>
</table>

---

### Figure 4. Word per cell data organization.

Consider first an algorithm for the $W/C$ data organization (Fig. 4), which adds contents of all $A$ fields to contents of respective $B$ fields, leaving the resulting sum in the $B$ fields. Tag 1 is used for carry storage and is assumed initially cleared. The routine is as follows:

---

### Figure 3. Format for associative command.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>Clear DP to zero</td>
</tr>
<tr>
<td>E1</td>
<td>Clear DP to one</td>
</tr>
<tr>
<td>N</td>
<td>No change in DP</td>
</tr>
<tr>
<td>W</td>
<td>Write</td>
</tr>
<tr>
<td>W'</td>
<td>Write lower neighbor</td>
</tr>
<tr>
<td>S</td>
<td>Single counter increment</td>
</tr>
<tr>
<td>S'</td>
<td>Don't search</td>
</tr>
</tbody>
</table>

---

### Figure 4. Word per cell data organization.

Consider first an algorithm for the $W/C$ data organization (Fig. 4), which adds contents of all $A$ fields to contents of respective $B$ fields, leaving the resulting sum in the $B$ fields. Tag 1 is used for carry storage and is assumed initially cleared. The routine is as follows:
The routine first addresses least significant bits of \( A \) and \( B \) fields and tag 1. Of eight possible states for these variables, four must be transformed to new states (see Introduction). Commands 0-3 accomplish the four required state transformations. Command 3 increments \( A \) and \( B \) column addresses. The routine is repeated at each column in the \( A \) (and thus \( B \)) field.

A routine which performs the equivalent operation for operands stored in the \( B/C \) organization (Fig. 5) is shown as follows:

<table>
<thead>
<tr>
<th>Instruction No.</th>
<th>( A ) Field</th>
<th>( B ) Field</th>
<th>Tag 1</th>
<th>Lowest Neighbor Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 ( E_{sl} )</td>
<td>( S - 1 W N S - 0 W N - - - )</td>
<td>( N )</td>
<td>( N )</td>
<td></td>
</tr>
<tr>
<td>1 1 ( E_{sl} )</td>
<td>( S - 1 W N S - 1 W N - - - )</td>
<td>( N )</td>
<td>( N )</td>
<td></td>
</tr>
<tr>
<td>2 1 ( N )</td>
<td>( S - - W - S - - W - S 0 W )</td>
<td>( L )</td>
<td>( L )</td>
<td></td>
</tr>
<tr>
<td>3 1 ( E_{sl} )</td>
<td>( S - - W - S - 0 W N S 1 W )</td>
<td>( N )</td>
<td>( N )</td>
<td></td>
</tr>
<tr>
<td>4 1 ( E_{sl} )</td>
<td>( S - - W - S - 1 W N S 1 W )</td>
<td>( N )</td>
<td>( N )</td>
<td></td>
</tr>
<tr>
<td>5 1 ( N )</td>
<td>( S - - W - S - - W - S 0 W )</td>
<td>( L )</td>
<td>( L )</td>
<td></td>
</tr>
<tr>
<td>6 0 ( N )</td>
<td>If ( DP \neq 0 ) jump to (3).</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instructions 0, 1, and 2 form the partial sum. Instructions 3, 4, and 5 ripple all carries to completion, as is detected by instruction 6. For a worst-case carry, \( N \) -1 iterations of steps 3, 4, and 5 are required.

However, even when the number of parallel data words is very large, the longest expected carry string is significantly less than \( N \) -1 bits. Typically, the foregoing algorithm is two to three times faster than for the algorithm presented for the \( W/C \) configuration.

ASSOCIATIVE PATTERN PROCESSING

A number of linear threshold-pattern-recognition devices have been built using analog techniques. Such devices are relatively fast and inexpensive when applied to simple pattern recognition tasks. They are limited in the allowed number of input variables and in the dynamic range of weights assigned to these variables. Wiring complexity increases rapidly with the number of threshold units used. Modification of the weights assigned input variables or of thresholds is expensive and time-consuming. Higher-order restructuring is even more difficult. Analog units are thus not suited for classification of complex problems for which many properties are measured, and where suitable properties may not be known a priori.

The parallel processing capability of an associative processor is well suited to the tasks of abstracting pattern properties and of pattern classification by linear threshold techniques. Threshold pattern recognition devices execute a given operation independently over many data sets, and thus allow the parallelism necessary for efficient associative processing. Associative processing affords the accuracy of digital number representation, and is thus unlimited in fan-in and dynamic range of weights. Weights are simply altered by changing memory contents. Wiring and components are regular and are thus amenable to low-cost, batch-fabrication techniques. The set of measured pattern properties is changeable by changing memory contents, rather than by rewiring as for analog units. Adaptation is thus possible in measured properties as well as in classification.

The Pattern-Processing Model

In this subsection, the pattern-processing model will be briefly described. Figure 6 represents the model of the pattern recognition system. \( N \) binary valued sensor units are summed, with weights \( \pm 1 \), into some or all of \( K \) thresholding logic units. A threshold level, \( t_k \), is established for each logic unit. If the sum of weighted inputs exceeds the threshold, the unit becomes active and the output, \( b_k \), is one; otherwise the output is zero. Each logic unit has a weighted connection to some or all of \( N_r \) response units. Weights of active logic units are summed and thresholded at each response unit. A pattern is classified according to the set of activated response units.

![Figure 6. Analog model of pattern recognition system.](From the collection of the Computer History Museum (www.computerhistory.org))
Associative Realizations

The associative memory is organized into three sections containing, respectively, the connectivity vectors $C_k$, $1 \leq k \leq K$; the system of weights $w_{kn}$, $1 \leq k \leq K$, $1 \leq n \leq N_r$; and the target vectors $\tau^m$, $1 \leq m \leq M$. The general organization of the associative memory is shown in Fig. 7, which is interpreted as follows: In Phase (1), the set of logic units activated by the $i$th pattern is determined, using the input vector $A^i$ and the stored connectivity vectors $C_k$. Logic unit outputs which yield the property vector $B^i$ are formed in the detector plane. In Phase (2), the inputs to the response units are calculated, using the vector $B^i$ and the weights stored in the appropriate portion of the associative memory. This yields the response vector $R^i$ in the detector plane. In Phase (3), the response vector $R^i$ is compared with the target vectors $\tau^m$ stored in the associative memory, and the classification of the pattern associated with $A^i$ is determined. The three processing phases are further described as follows:

**Phase (1).** The set of sensor-logic unit connections with weights 1 and -1 may be represented by a matrix $[C]$ which is stored in the connectivity sector of an associative memory in the format of Fig. 8. The matrix $[C]$ may be written

$$[C] = [C^+] + [C^-]$$

where $[C^+(-)]$ is a matrix with binary values whose entries represent those connections which are positive (negative). That is, the element $c^{+}_{jk}$ is +1 if there is a positive connection between sensor $j$ and logic unit $k$, and is 0 otherwise. Similarly $c^{-}_{jk}$ is 1 if there is a negative connection between sensor $j$ and logic unit $k$. Thus, all connections from sensors to the $k$th logic unit are represented by the row vector $C_k = C^+_{k} + C^-_{k}$. Matrix rows $C^+_{k}$ and $C^-_{k}$ are stored in adjacent fields of an associative memory word.

---

**Figure 7.** Associative parallel processor realization of pattern recognition system.

**Figure 8.** Connectivity sector of associative parallel processor.
The input vector \( \mathbf{A}^i \) is used as a key to interrogate matrix \( [C] \) in the associative memory in order to determine the set of logic units activated by the input vector \( \mathbf{A}^i \). Bit positions having \( a^i_j = 1 \) are interrogated for \( c^+_k \) or \( c^-_k = 1 \). A binary count of the number of bits in \( C^+_k \) which satisfy this condition is made in the accumulation fields \( S_c \) respectively of each word in Fig. 8. The procedure is again repeated for \( C^-_k \). Thresholds \( t_k \) are prestored in fields as indicated. After counting, the contents of the \( T \) fields are subtracted from those of fields \( S_c \). Elements of \( S_c \) remaining positive under these operations correspond to activated logic units.

Following these Phase (1) operations, a single search for positive counts \( S_c \) sets the detector plane to the match state at each word corresponding to an activated logic unit. Contents of this segment of the detector plane are transferred to the data register for use as search criteria in Phase (2).

**Phase (2).** This phase generates the output vector \( r^1_i, \ldots, r^N_i \) corresponding to the \( i \)th (input) pattern. The components

\[
r_n = \text{Sgn} \left[ \sum_{k=0}^{K} w_{kn} b^i_k \right]
\]

are to be generated within the associative processor, using \( b^i_k \) as an input from Phase (1) and the weights \( w_{kn} \) stored in the weight section of the associative memory, Fig. 7. Arithmetic operations, similar to those discussed in the second section of this paper, are then used to yield the response vector \( (r^1_i, \ldots, r^N_i) \).

**Phase (3).** The target vectors stored in the associative memory (Fig. 7) are vectors with binary valued components \( (r^1_i, \ldots, r^N_i) = \tau^m \), where \( \tau^m \) represents the known classification of the \( i \)th pattern. It is assumed that the \( K \times N_r \) matrix of weight \( [W] \) stored in the associative memory have been predetermined such that each of the \( M \) patterns of interest is correctly classified, i.e., the output of Phase (2) matches the appropriate target vector, component for component.

Components \( (r^1_i, \ldots, r^N_i) \) of target vectors \( \tau^m \) are stored in a portion of the associative array at locations derived from "names" \( d_m(i \leq m \leq M) \) of patterns associated with the target vectors (Fig. 7). A response vector generated during Phase (2) is used as a search key to interrogate the target vector portion of the associative array. The location of any responding word denotes the name of the pattern, as in the following table, showing the target vector portion of associative memory.

<table>
<thead>
<tr>
<th>Target</th>
<th>Vector</th>
<th>Name</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r^1_i )</td>
<td>( r^N_i )</td>
<td>( d_i )</td>
<td>( f(d_i) )</td>
</tr>
<tr>
<td>( r^2_i )</td>
<td>( r^N_i )</td>
<td>( d_2 )</td>
<td>( f(d_2) )</td>
</tr>
<tr>
<td>( r^M_i )</td>
<td>( r^N_i )</td>
<td>( d_M )</td>
<td>( f(d_M) )</td>
</tr>
</tbody>
</table>

**Processing Times**

A program was written for the described pattern-recognition model using the instruction set presented in the second section of this paper.

The pattern recognition program has an adaptive or learning mode, requiring 120 instructions, in which weights are adjusted to properly classify a set of input patterns. The program for each mode is invariant to pattern parameters used for classification. Since 82 instructions are common to the two modes, only 131 instructions need be stored in program memory.

![Figure 9. Time for associative classification of a single pattern as a function of the number of patterns and sensors for the \( W/C \) data organization.](image-url)

*This cycle time is based on a magnetic film realization of the associative array described in reference 12.*
Based on the aforementioned recognition program, a word-per-cell data organization, and an associative command cycle of 0.8 microseconds, the graph of timing efficiency shown in Fig. 9 was constructed. Note that "N" represents the number of sensor units at the input and "M" the number of patterns distinguishable by the processor. The "MARK I Perceptron" used 400 sensors in a 20 × 20 array and 512 logic units.

It can be seen that the APP could solve this problem in approximately 3 milliseconds using some 2000 words of associative storage. The APP realization offers significantly greater ease of alteration and somewhat lower cost at a moderate increase in processing speed relative to the Mark I.

CONCLUSION

The associative parallel processor, described in this paper, achieves considerable generality with simple word and bit logic through the use of the sequential-state-transformation mode of associative processing. Its range of applicability is increased by novel arithmetic algorithms allowing simultaneous processing of many operands in bit parallel fashion. These algorithms allow efficient use of the processor for problems in which only a fraction of the stored operands are processed by a given command. Earlier processors4,5,6 were efficient only when nearly nearly all operands were processed by each command.

An important feature of the parallel processor, when used as a pattern recognition device, is the ability to modify its functional structure, through alteration of memory contents, without change in its periodic physical structure. This adaptive feature has importance in applications where patterns change with time, or where the processor is used as a prototype of subsequent machines having fixed recognition capabilities. Further research is required to fully exploit this adaptive capability.

Linear threshold pattern classifiers of the type here presented are beginning to find many applications. To date, these types of pattern classifiers have been studied and/or implemented for character recognition, photointerpretation, weather forecasting by cloud pattern recognition, speech recognition, adaptive control systems and more recently, for medical diagnosis from cardiographic data. Other possible applications include terminal guidance for missiles and space vehicles and bomb damage assessment.

Currently the processor is being studied for application to the tasks of job shop scheduling, optimum commodity routing and processing electromagnetic intelligence (ELINT) data. In each instance significant speed gains have been shown possible over conventional sequential digital computers. It is interesting to note that the processor described in the second section of this paper may be applied to this variety of tasks without significant changes in organization or command structure.

ACKNOWLEDGMENTS

The authors take pleasure in acknowledging contributions to this effort by Mr. J. Medick and Dr. J. C. Tu. We are most appreciative of support given this work by Messrs M. Knapp and A. Barnum of the Rowe Air Development Center.

REFERENCES

9. J. C. Hay, F. C. Martin and C. W. Wight-

