NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-81-126

Recognize Regular Languages With Programmable Building-Blocks

M. J. Foster and H. T. Kung

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213, USA

June 19, 1981

This paper is to appear in the proceedings of the VLSI-81 Conference, Edinburgh, August 1981.

Copyright (C) 1981 Michael J. Foster and H. T. Kung

This research was supported in part by the Office of Naval Research under Contract N00014-80-C-0236, NR 048-659, and in part by the Defense Advanced Research Projects Agency under Contract F33615-78-C-1551 (monitored by the Air Force Office of Scientific Research). M. J. Foster was supported in part by a National Science Foundation Graduate Fellowship.

Abstract

This paper introduces a new programmable building-block for recognition of regular languages. By combining three types of basic cells a circuit for recognizing any regular language can be constructed or "programmed" automatically from the regular expression describing that language. Recognizers built in this way are efficient pipeline circuits that have constant response time and avoid broadcast. In addition, the paper proposes the use of a single, regular layout, called the PRA (programmable recognizer array), that can be "personalized" to recognize the language specified by any regular expression. PRA's provide compact reconfigurable layouts for recognizer circuits, requiring only $O(n \log n)$ area for regular expressions of length n.

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

1. Introduction

Construction of future VLSI systems will rely on the use of *programmable building-blocks*. A building-block consists of a set of cell designs together with rules for combining the cells into larger circuits, and for using these circuits in large systems. The PLA (programmable logic array) for example, is a programmable building-block frequently used for implementing random logic. Because the structure of a building-block is fixed and prespecified, layout generators, simulators and other high-level design tools can be used effectively. Thus using building-blocks helps manage the complexity of VLSI design. A *programmable* building-block can be "personalized" to realize various functions. If building-blocks are programmable, designers can proceed to high-level designs before all the low-level functions are specified. This often speeds up the design process and increases the flexibility of the final system. These advantages of using programmable building-blocks have already been demonstrated in several recent projects [4, 8]. This paper proposes a new programmable building-block for constructing efficient circuits that recognize regular languages.

Regular languages are precisely those languages that can be recognized by finite-state machines (see, *e.g.*, [5]). They are well suited for describing identifiers in a programming language or patterns to be matched by a text editor, and for modeling processes associated with electronic circuits and nervous systems. Language recognizers are often used as components in larger systems, such as controllers and sequencers. For example, Haskin [3] has recently suggested using language recognizers as term matchers in special-purpose database machines. We show that by combining some basic building-blocks a recognizer circuit for a language can be constructed automatically from the regular expression describing that language, and a circuit so constructed can itself be a building-block for constructing larger systems.

2. Basic Ideas

To motivate the construction of recognizer circuits using this building block we present several examples of increasing complexity. Our first example is a linear pipeline that can recognize concatenations. Figure 2-1 shows a pipeline that recognizes any three character pattern. Before the computation starts, the pipeline is loaded with the pattern ABC, one pattern character at each recognizer cell. During the computation the cells are synchronized to operate together on discrete clock ticks, or beats. On each beat, the text to be matched moves through the pipeline from right to left, and the results of the match move from left to right. Data in both streams are separated by one cell to permit each character to meet every result. On each beat, every cell that is active compares its prestored pattern character with the text character received from its right, then sends the text character on to its left. The cell AND's this comparison result with the result received from the left,

and sends the new result to the right. The result of comparing a text string with the pattern is available from the pipeline on the beat after the last text character is input, thus a *constant response time* is achieved. Figure 2-1 traces the action of the pipeline for several beats. Cell contents on each beat are shown underneath the corresponding cells.

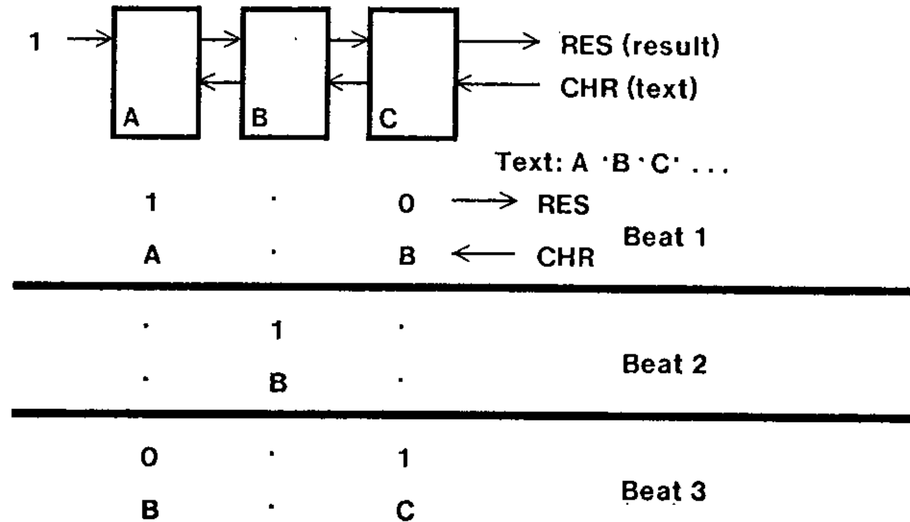


Figure 2-1: *Circuit Programmed for the Expression* ABC

Our second example is a tree-structured pipeline that can recognize any regular expression consisting of a union of several concatenations followed by a single concatenation. Figure 2-2 shows a tree-structured pipeline programmed to recognize the language generated by expression (AB + C)DE. This is an obvious extension of the linear pipeline in Figure 2-1: characters fan out to both branches of the tree when they reach the " + " node, and results from the two branches go through an OR gate at that node. Neither characters nor results are stored in the " + " node; they just pass right through. Once again, the result of matching a text string against the pattern is available one beat after the last text character goes into the pipe.

This pipeline scheme cannot be extended in the obvious way to expressions such as A(BC + D)E, which contain a union preceded by a concatenation. If we try to use a pipeline like that in Figure 2-3, where characters and results flow around both branches of the loop, it is impossible to maintain synchronization when the branches differ in length. Instead we must add a third data stream called the *enable* stream, as shown in Figure 2-4. In addition to coordinating the CHR and RES data streams, this new stream will serve the function of the "anchor" or "λ" in previous recognizers [2, 7], permitting matching at selected positions within the text stream. On each beat the enable stream

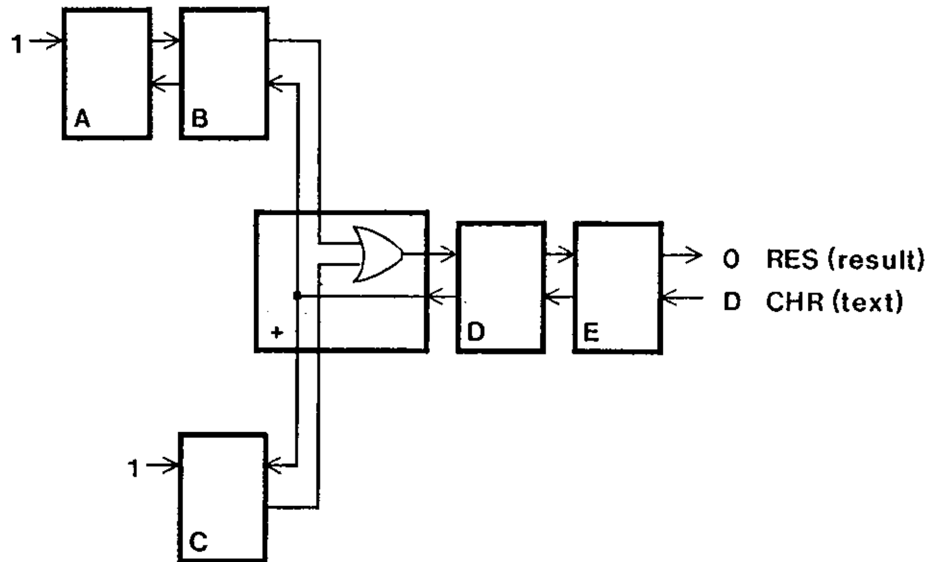


Figure 2-2: *Circuit Programmed for the Expression* (AB + C)DE

moves from right to left, and is fed into the result stream at the end of each branch. As in the other two streams, data in the enable stream are separated by one cell.

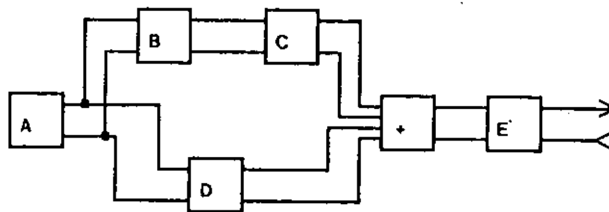
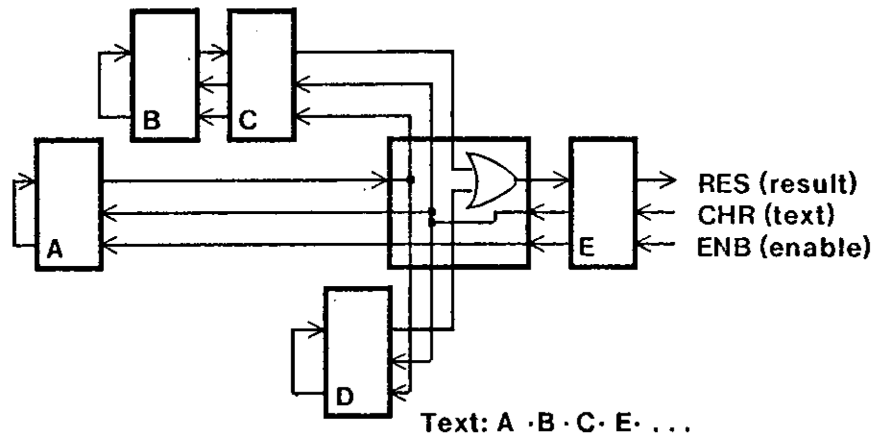


Figure 2-3: *Obvious (Wrong) Extension of Tree for* A(BC + D)E

A correct pipeline for the expression A(BC + D)E is shown in Figure 2-4, together with a record of the contents of each cell for several consecutive beats. The 1's in boxes track a successful match through the pipeline from cell A to cell E. Notice that the text characters are sent through the " + " node to the A node, as well as to the C and D nodes. Thus on each beat, the same text character is in all three of those nodes, and on the next beat it is in the B node. The result from the A character cell fans out (through the " + " node) to the enable stream of both the BC and D branches of the tree. The match results from the A cell thus reach the result stream input to both the B and D cells in synchrony with the character string.



[1]	·	0	0	·	→ RES
A	·	A	A	·	← CHR
·	0	·	·	1	← ENB (= 1)
·	0	·	·	0	
·	A	·	·	B	Beat 2
1	·	[1]	1	·	
0	·	0	0	·	
B	·	B	B	·	Beat 3
·	[1]	·	·	1	
·	[1]	·	·	0	
·	B	·	·	C	Beat 4
1	·	0	0	·	
0	·	[1]	0	·	
C	·	C	C	·	Beat 5
·	0	·	·	1	
·	0	·	·	[1]	Result = 1
·	C	·	·	E	Beat 6
1	·	0	0	·	

Figure 2-4: *Correct Extension of Tree for* A(BC + D)E

As is shown in the next section, similar tree-structured recognizers can be built for any regular expression. Notice that, unlike the recognizers of Floyd and Ullman [1] and Mukhopadhyay [7], our circuits achieve a constant response time, and do not require broadcast of the text characters. Our recognizers are thus well suited to VLSI implementation, in which broadcast is slow, but local communication is fast.

3. Recognizers for Arbitrary Regular Expressions

How can we construct a recognizer for an arbitrary regular expression? In this section we describe three types of basic cells and give a procedure for hooking them up to form any recognizer.

Three kinds of cells are used in constructing recognizers: comparators for single characters, and combinational cells corresponding to the union (" + ") and Kleene closure ("*") operators. Each of the basic cells of a recognizer has one or more data paths passing through it, with three data streams on each data path. The CHR and ENB streams, which flow from right to left, carry the text characters and enable bits. The RES stream, which flows from left to right, carries the result bit. We can hook these cells together to form recognizers by connecting the data path at the right side of one cell to a data path at the left side of another cell.

The character comparator is shown in Figure 3-1, together with a symbol used in designing large recognizers. This cell is similar to the character comparator described by Foster and Kung [2]. On each beat, the cell performs these steps:

1. Compare the text character with the stored pattern character X in the PAT register, AND that result with the RES register, and pass the one bit result to the right.
2. Pass the CHR and ENB registers leftward and receive new contents from the right.
3. Receive new contents for the RES register from the left.

The cells for the " + " and "*" operators contain only combinational logic, and are shown in Figures 3-2 and 3-3.

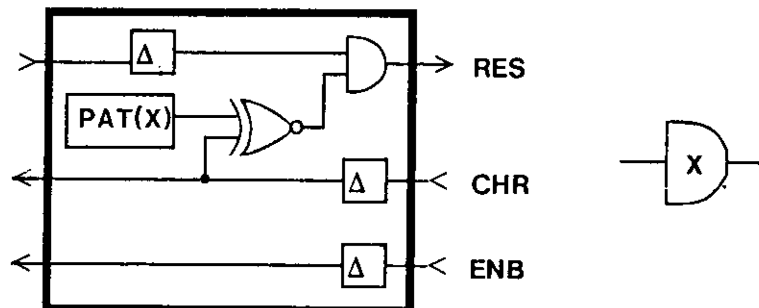


Figure 3-1: *Character Comparator Cell*

To construct recognizers for arbitrary regular expressions, these cells are combined into larger circuits. We use a new technique for combining the cells in which a context-free grammar describes

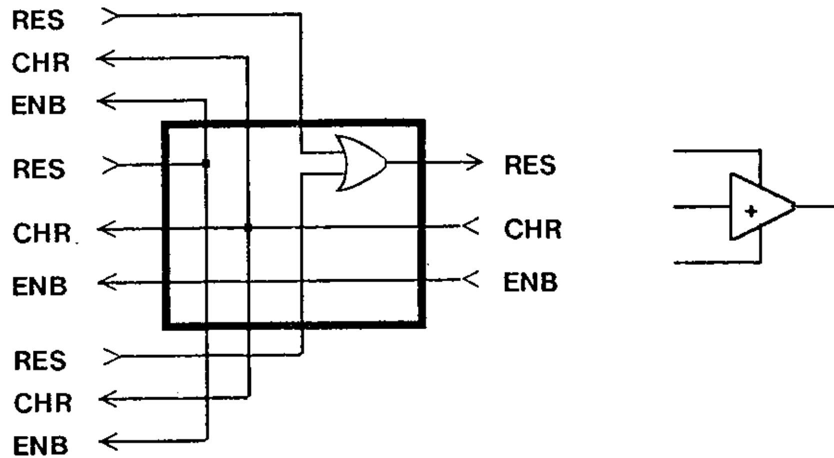


Figure 3-2: " + " *Operator Cell*

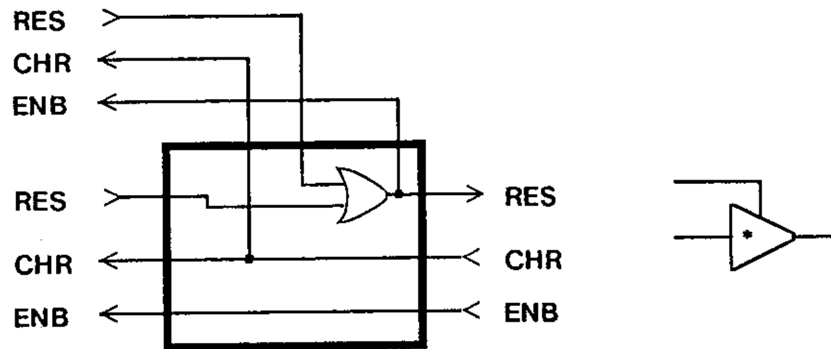


Figure 3-3: "*" *Operator Cell*

both the structure and function of the final circuit. Terminal symbols in the grammar correspond to basic cells, and semantic actions attached to the productions of the grammar tell how to hook them together.

To construct a recognizer for a regular expression, we parse the expression using the grammar:

R ::= P | RP
P ::= <letter> | (R)* | (R + R)

Each symbol of this grammar represents a kind of circuit. The <letters> represent the character comparator cells, for example, and the non-terminal symbol R represents a recognizer for a regular expression. Each production has an associated semantic rule that tells how to connect the circuits on the right side of the production to form the circuit on the left side. Every time a production is used in parsing the regular expression, its semantic rule is used to add to the circuit.

This syntax-directed construction technique eases verification of functional correctness, and other properties of the resulting circuits. Proof of a single theorem for each production in the grammar will verify the correctness of *any* recognizer. We attach to each symbol in the grammar a predicate describing its circuit. For each production in the grammar, we then prove that if circuits satisfying the predicates on its right hand side are connected according to its semantic rule, then the resulting circuit satisfies the predicate on the left hand side. This verification technique promotes confidence that large recognizers will work as expected.

4. PRA: Programmable Recognizer Array

The pipeline circuits constructed above form ternary trees, so each of them can be laid out in an area efficient manner [1, 6]. This section describes an alternative to the approach of individually laying out the tree corresponding to each recognizer circuit. We propose the use of a single, compact layout, called the PRA (programmable recognizer array), that can be personalized to recognize the language specified by any regular expression. Our methods are similar to those of Leiserson [6].

For a recognizer of size n, we lay out n basic cells on the bottom line of the array, and provide O(log n) channels in the top portion of the array for data paths parallel to the line, as shown in Figure 4-1(a). To configure the layout for a particular tree, we route the edges of the tree through the channels. Ternary trees have a constant separator theorem, so that by removing a single edge of the tree we can split it into two subtrees of roughly equal size. We split the line of cells into two lines, one for each of the subtrees, and use one channel to route the data path corresponding to the removed edge between the two subtrees. We then apply the same procedure recursively to lay out the subtrees on their lines of cells. Figures 4-1 (b) and (c) show the same PRA "programmed" for two different regular expressions.

A PRA of dimensions n by O(log n) can be programmed to recognize the language generated by *any* regular expression of length n. For the same problem a PLA implementation, as proposed by Floyd and Ullman [1], would require n by n area in the worst case. For recognizing languages described by large regular expressions, a number of small PRA's can be combined using the syntactic method described earlier.

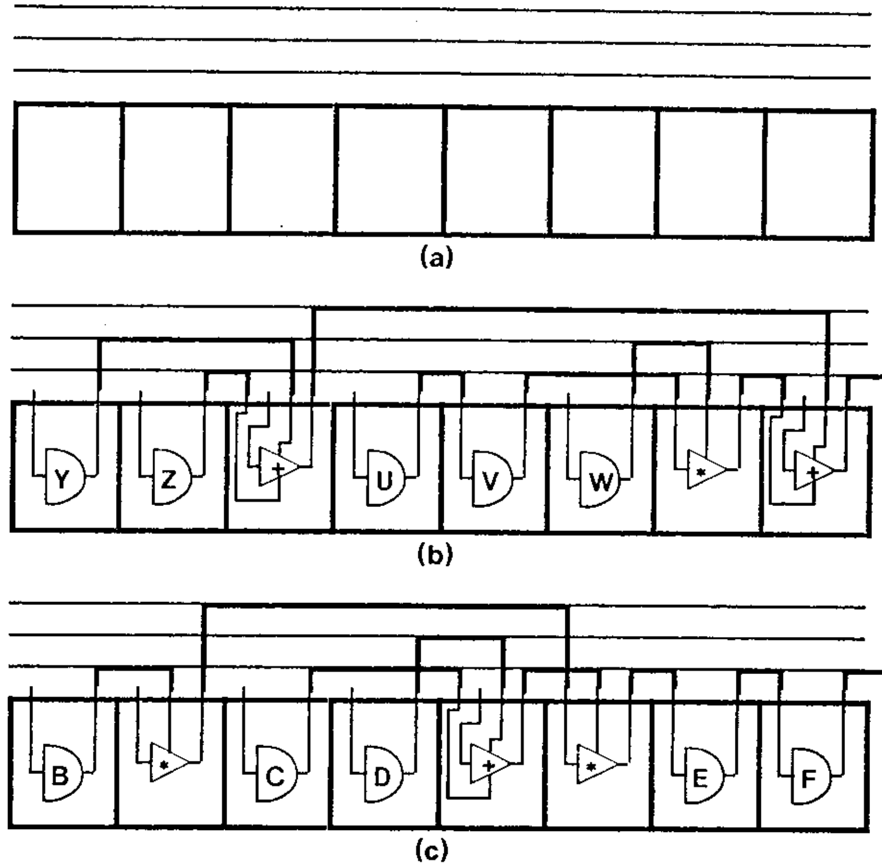


Figure 4-1: (a) *PRA before "personalization"*
(b) *PRA for* $UVW^* + (Y + Z)$
(c) *PRA for* $B^*(C + D)^*EF$

5. Conclusions

This paper introduces a new programmable building-block for recognition of regular languages. The building-block can be formed (or "programmed") for any regular expression using a syntax-directed construction method, which also allows easy and mechanical verification of circuit properties. Recognizers built using these building-blocks are efficient pipeline circuits that have constant response time and avoid broadcast. In addition, PRA's provide compact reconfigurable layouts, requiring only $O(n \log n)$ area for regular expressions of length n. Programmable recognizers should be included as one of the building-blocks in the I.C. designer's toolbox.

References

[1] Floyd, R.W. and Ullman, J.D.
The Compilation of Regular Expressions into Integrated Circuits.
In *Proceedings of 21st Annual Symposium on Foundations of Computer Science.* IEEE Computer Society, oct, 1980.

[2] Foster, M. J. and Kung, H.T.
The Design of Special-Purpose VLSI Chips.
Computer Magazine 13(1):26-40, January, 1980.
A preliminary version of the paper, entitled "Design of Special-Purpose VLSI Chips: Example and Opinions", appears in *Proceedings of the 7th International Symposium on Computer Architecture*, pp. 300-307, La Baule, France, May 1980.

[3] Haskin, R. L.
Hardware for Searching Very Large Text Databases.
PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[4] Holloway, J., G. L. Steele, G. J. Sussman and A. Bell.
The SCHEME-79 chip.
Technical Report AI Memo No. 559, EE&CS Integrated Circuit Memo No. 80-6, Massachusetts Institute of Technology Artificial Intelligence Laboratory, January, 1980.

[5] Hopcroft, J. E. and Ullman, J. D.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley Publishing Co., 1979.

[6] Leiserson, C.E.
Area-Efficient VLSI Computation.
PhD thesis, Carnegie-Mellon University, 1981.

[7] Mukhopadhyay, A.
Hardware Algorithms for Nonnumeric Computation.
IEEE Transactions on Computers C-28(6):384-394, June, 1979.

[8] Stritter, S. and Tredennick, N.
Microprogrammed Implementation of a Single Chip Microprocessor.
In *Proceedings of the IEEE Eleventh Annual Microprogramming Workshop.* IEEE, November, 1978.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-81-126	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) RECOGNIZE REGULAR LANGUAGES WITH PRGRAMMABLE BUILDING-BLOCKS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) M.J. FOSTER & H.T. KUNG		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0236
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA. 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE June 19, 1981
		13. NUMBER OF PAGES 11
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)