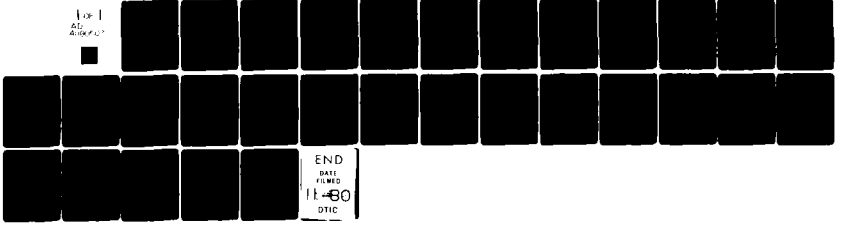
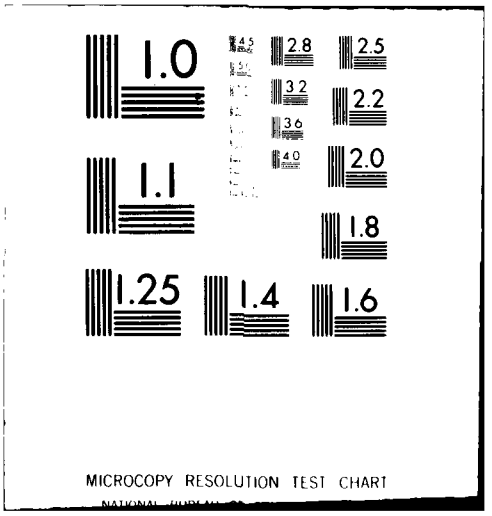
AD-A090 507 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/5
THE COMPILATION OF REGULAR EXPRESSIONS INTO INTEGRATED CIRCUITS--ETC(U)
APR 80 R W FLOYD, J D ULLMAN MDA903-80-C-0102
UNCLASSIFIED STAN-CS-80-798 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
MICROCOPY RESOLUTION TEST CHART

Stanford Computer Science Department
Report No. STAN-CS-80-798

April 1980

LEVEL II

THE COMPILATION OF REGULAR EXPRESSIONS INTO INTEGRATED CIRCUITS

by

Robert W. Floyd and Jeffrey D. Ullman

Research sponsored by

Advanced Research Projects Agency
and
National Science Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: STAN-CS-80-798

2. GOVT ACCESSION NO.: AD-A090 507

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): The Compilation of Regular Expressions into Integrated Circuits,

5. TYPE OF REPORT & PERIOD COVERED: technical, April 1980

6. PERFORMING ORG. REPORT NUMBER: STAN-CS-80-798

7. AUTHOR(s): Robert W. Floyd and Jeffrey D. Ullman

8. CONTRACT OR GRANT NUMBER(s): ARPA MDA 903-80-C-0102, NSF-MCS-79-04528

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Department of Computer Science
Stanford University
Stanford, California 94305 USA

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Avenue, Arlington, Virginia 22209

12. REPORT DATE: April 1980

13. NO. OF PAGES: 28

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office):
Mr. Philip Surra, Resident Representative
Office of Naval Research, Durand 165
Stanford University

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited.

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

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)

(see other side)

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

We consider the design of integrated circuits to implement arbitrary regular expressions. In general, we may use the McNaughton-Yamada algorithm to convert a regular expression of length n into a nondeterministic finite automaton with at most $2n$ states and $4n$ transitions. Instead of converting the nondeterministic device to a deterministic one, we propose two ways of implementing the nondeterministic device directly. First, we could produce a PLA (programmable logic array) of approximate dimensions $4n \times 4n$ by representing the states directly by columns, rather than coding the states in binary. This approach, while theoretically suboptimal, makes use of carefully developed technology and, because of the care with which PLA implementation has been done, may be the preferred technique in many real situations. Another approach is to use the hierarchical structure of the automaton produced from the regular expression to guide a hierarchical layout of the circuit. This method produces a circuit $0(\sqrt{n})$ on a side and is, to within a constant factor, the best that can be done in general.

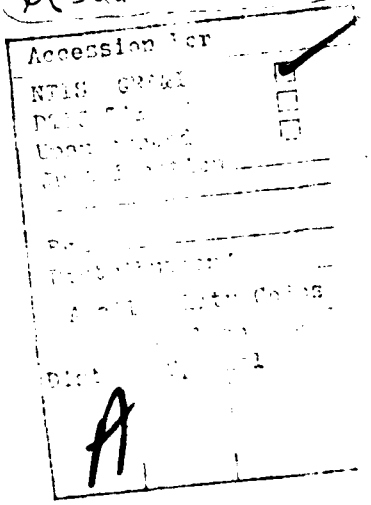
THE COMPILATION OF REGULAR EXPRESSIONS INTO INTEGRATED CIRCUITS

Robert W. Floyd
Jeffrey D. Ullman†
Stanford University
April, 1980

ABSTRACT

We consider the design of integrated circuits to implement arbitrary regular expressions. In general, we may use the McNaughton-Yamada algorithm to convert a regular expression of length n into a nondeterministic finite automaton with at most $2n$ states and $4n$ transitions. Instead of converting the nondeterministic device to a deterministic one, we propose two ways of implementing the nondeterministic device directly. First, we could produce a PLA (programmable logic array) of approximate dimensions $4n \times 4n$ by representing the states directly by columns, rather than coding the states in binary. This approach, while theoretically suboptimal, makes use of carefully developed technology and, because of the care with which PLA implementation has been done, may be the preferred technique in many real situations. Another approach is to use the hierarchical structure of the automaton produced from the regular expression to guide a hierarchical layout of the circuit. This method produces a circuit $\theta(\sqrt{n})$ on a side and is, to within a constant factor, the best that can be done in general.

θ(SQUARE ROOT OF N)



† Work partially supported by DARPA contract MDA903-80-C-0102 and by NSF grant MCS-79-04528.

I. Introduction

There are a number of projects, such as [S, G, J] whose goal is "silicon compilation," that is, the automatic layout of circuits from their behavioral description. These projects tend to be oriented around the design of computer-like circuits, certainly an important goal, but one that is analogous, in the software domain, to implementing languages suitable for writing operating systems, but little else. It appears that the "Fortran" of circuit implementation must be quite general-purpose, allowing us to specify a great variety of different kinds of circuits and to implement anything we can specify, with a fair degree of efficiency.

It is the purpose of this paper to discuss only one possible component of such a general-purpose language, a regular expression facility. Regular expressions are capable of specifying any finite-state process, although they are not always as succinct as other representations [EZ]. Fortunately, there is a common class of finite-state processes for which regular expressions appear very well suited indeed. In the software world, lexical analyzers, which recognize the tokens (e.g., identifiers, keywords) of a programming language, have been generated automatically from regular expressions defining the tokens. Regular expressions also make a good language for describing patterns to be matched by a text editor. [AU] describes these and other software applications of regular expressions.

In the hardware world, regular expressions are well suited to describing processes like controllers, where it is desired that we signal "events," where each "event" consists of a sequence of significant input signals, perhaps interspersed with arbitrary numbers of irrelevant signals. We shall later give a design example for a simple device of this sort. On the other hand, regular expressions are not very good for describing counting processes. For example, the event "876 zeros" is most naturally described by the regular expression $00\cdots0$ (876 times). Obvious techniques for producing a circuit from this expression will only succeed in producing a unary counter with 876 distinct memory elements, rather than a binary counter with ten memory elements. Extensions to the regular expression language can alleviate this problem somewhat, but the fact remains that regular expressions cannot be billed as a panacea, even if we restrict our domain of interest to sequential processes. However, they do represent a promising approach to the automatic design of some components, and they probably have a place in any general-purpose compiled circuit design language.

II. The Circuit Model

To be specific, let us assume that circuits are implemented in the nMOS technology, using the Mead-Conway [MC] design rules. However, what we say applies to any technology in which

1. 2-input logical operations can be implemented in constant space.
2. Wires have a fixed constant width, and signals can be driven through the wire in an acceptably short time by a driver no larger than the wire itself.
3. No more than a constant number of wires or logic elements may occupy the same area; the constant 3 applies to the nMOS technology.

This model of integrated circuits is discussed in [T, BK], for example.

III. Regular Expressions and Nondeterministic Automata

We assume the reader is familiar with finite automata theory as discussed in [HU], for example, and we only sketch the essential details here. Regular expressions are built from an alphabet Σ (in practice, Σ might be the set of ASCII characters, for example) using the following rules.

1. For each a in Σ, a is a regular expression denoting $\{a\}$, that is, the set consisting of one string; that string is of one symbol, a.
2. $\emptyset$ and ϵ are regular expressions denoting, respectively, the null set and $\{\epsilon\}$, that is, the set consisting of the *empty string* (zero-length string) only.
3. If R_1 and R_2 are regular expressions denoting sets of strings S_1 and S_2, respectively, then $(R_1)+(R_2)$, $(R_1)(R_2)$, and $(R_1)^*$ denote $S_1 \bigcup S_2$, $S_1 S_2$, and $S_1{}^*$, respectively. Here $S_1 S_2$ is the *concatenation* of sets S_1 and S_2, that is,

$$\{ xy \mid x \in S_1 \wedge y \in S_2 \}$$

Also, $S_1{}^*$, the *closure* of S_1, is

$$\{\epsilon\} \bigcup S_1 \bigcup S_1 S_1 \bigcup S_1 S_1 S_1 \bigcup \cdots$$

That is, $(R)^*$ means "zero or more occurrences of R."

4. Parentheses may be dropped when they are implied by the following precedence order: closure highest, then concatenation, then union. For example, $a + bc^*$ is grouped $a + (b(c^*))$ and stands for the set of strings

$$\{ a, b, bc, bcc, bccc, \ldots \}$$

Sometimes it is useful to extend the regular expression language in several ways that do not affect the collection of sets of strings we can define. For example, LEX [Les], the UNIX lexical analyzer generator, uses . to stand for "any character," that is, the expression $a_1 + a_2 + \cdots + a_n$, where the a_i's are all the symbols in Σ. Also, $(R)^+$ stands for the *positive closure* of R, that is, $R + RR + RRR + \cdots$, or "one or more occurrences of R. The expression $(R)?$ means "zero or one occurrence of R, that is, $\epsilon + R$.

A *nondeterministic finite automaton* (NFA) is conventionally represented by a directed graph, whose nodes are *states*, and an arc from state p to state q can be labeled by any symbol from Σ or by ϵ, the empty string. We allow multiple arcs between two states, but we usually represent these arcs by a single arc with more than one label. One state is designated the *start state*, and one or more states are designated *accepting* or *final* states. The NFA *accepts* a string $a_1a_2\cdots a_n$ if there is a path from the start state to some accepting state, and the labels of the arcs along that path read $a_1a_2\cdots a_n$. Note that ϵ may be a label of one or more of those arcs, but ϵ is "invisible," that is, it can appear any number of times along the path without appearing in the string accepted.

Example 1: Let us now take an example of how a sequential process can be represented by regular expressions and by an NFA. Consider a control unit that receives a sequence of two bits, which it interprets as a command according to the code

$$\begin{aligned} 00 &= \text{add} \\ 01 &= \text{subtract} \\ 10 &= \text{load} \\ 11 &= \text{load complement} \end{aligned}$$

For simplicity, we assume that the source of commands is "well behaved"; we never receive anything but two bits at consecutive times, nor can a second command be received while the previous command is being processed.

The output consists of three lines, A, C, and L, which respectively cause (A) add the memory buffer to some particular register, (C) complement the memory buffer, and (L) load the memory buffer into the register. When the C signal is sent, the controller waits for a completion input signal (X) before sending the A or L signal. As the machine is synchronous, we actually have a fourth input symbol besides 0, 1, and X in our alphabet Σ. We use N to indicate that no command bit or completion signal is present on the input.

As an aside, we note that the input alphabet $\Sigma = \{0, 1, X, N\}$ should be regarded as consisting of logical, rather than physical inputs. For example, in practice there might be three binary input lines: "command bit," "command present indicator," and "completion." The 0 input is represented by a command bit of 0, with the command-present bit set to 1. The completion bit can be ignored, as a 1 on that line while the command is present violates our assumption that commands do not overlap. The interpretation of the three bits as input symbols from Σ is shown in Fig. 1.

The regular expression for the "add" output signal is given by

$$A = .^*0(0 + 1N^*X)$$

command bit	0	0	0	0	1	1	1	1
command present	0	0	1	1	0	0	1	1
completion	0	1	0	1	0	1	0	1
logical input	N	X	0	0	N	X	1	1

Fig. 1. Actual-to-logical input interpretation.

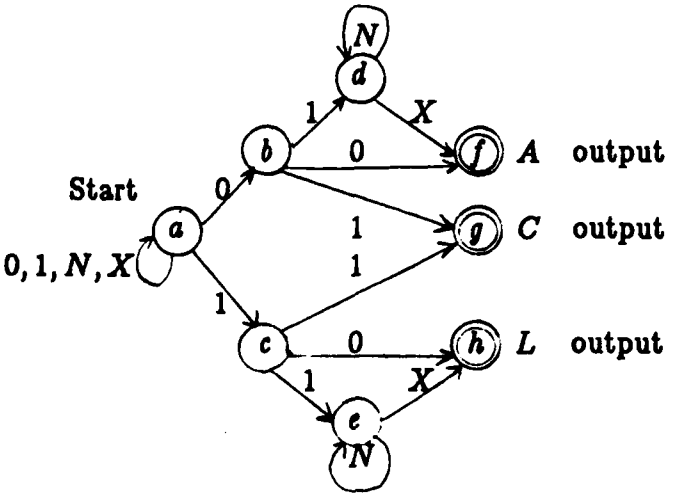


Fig. 2. An NFA for the controller example.

where . stands for "any input symbol." That is, we wish to signal an addition if after any sequence of inputs we see a 0 followed immediately by either

1. another 0, completing the command 00=add, or
2. a 1, completing the command 01=subtract, followed by any number of N's and an X. In this case, we assume the "complement buffer" signal C is sent after receiving 01. The N's represent "clock ticks" while we wait for the completion signal. When the X is received, we know the buffer has been complemented and immediately issue the "add" signal.

Similarly, we can specify the conditions under which we should emit the C and L signals by

$$C = .^*(0+1)1$$

$$L = .^*1(0+1N^*X)$$

We shall subsequently discuss an algorithm to convert any regular expression to an NFA with some arcs labeled ϵ. However, we first illustrate the NFA concept with one NFA for the controller; this NFA, shown in Fig. 2, uses no ϵ-arcs, but it does have nondeterminism, in the sense that it can be in more than one state at

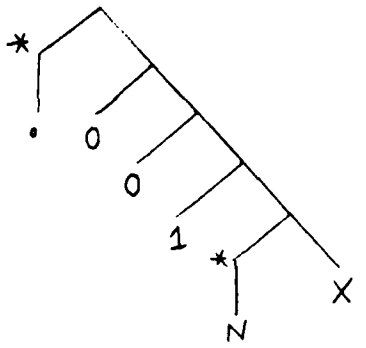


Fig. 3. Parse tree for $.^*0(0+1N^*X)$.

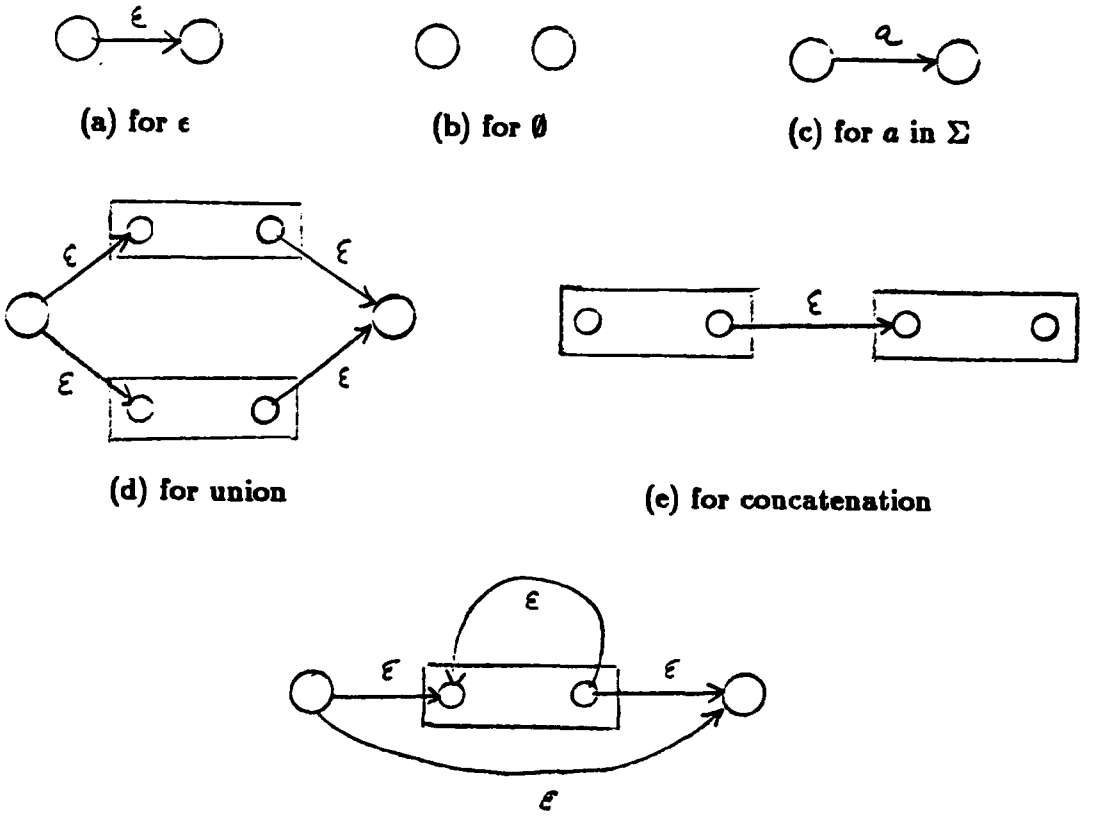
the same time.† For example, suppose we have input $N01$. We begin in state a, the start state. The only arc with label N leaving state a leads back to a. Thus, after the first input symbol we are only in state a. The next input, 0, labels arcs from a to a and b, so after the second input we are in those two states. Then we look for arcs out of a or b labeled 1, and we find them from a to a and c, and from b to d and g. Thus, after the third input we are in a, c, d, and g. Since g is a final state, we "accept" $N01$. In practice, state g represents the C signal, which is appropriate, since our input is one instance of the "subtract" command. []

IV. The McNaughton-Yamada Construction

We shall now discuss a recursive algorithm for converting regular expressions to NFA's with some ϵ-arcs. The algorithm produces NFA's for the regular expressions $(R_1)+(R_2)$, $(R_1)(R_2)$, and $(R_1)^*$, given NFA's for R_1 and R_2. To begin, we must "parse" a regular expression. That is, we view the regular expression as a *parse tree*, where leaves represent symbols in Σ (or ϵ or $\emptyset$ if needed), and interior nodes represent the application of union, concatenation, and closure operators to subexpressions. For example, the parse tree for expression A of Example 1 is shown in Fig. 3. See [AU] for a description of how parse trees for regular expressions can be constructed.

The McNaughton-Yamada algorithm [MY, HU] constructs for any regular expression an NFA with one start state and one final state. It is conventional to draw NFA's with the start state on the left and the final state on the right. Figure 4(a)–(c) shows the basis of the construction, the two-state NFA's that recognize

† Nondeterminism should not disturb us here. The NFA is a mathematical abstraction, and its implementation in hardware is quite different from that of its deterministic counterpart (a *DFA*), which is guaranteed to be in only one state at a time.



(a) for ϵ (b) for $\emptyset$ (c) for a in Σ

(d) for union (e) for concatenation

(f) for closure

Fig. 4. The McNaughton-Yamada constructions.

$\epsilon, \emptyset$, and any particular a in Σ, respectively. Figure 4(d)–(f) shows how NFA's M_1 and M_2 for regular expressions R_1 and R_2 are combined to get NFA's for $(R_1)+(R_2)$, $(R_1)(R_2)$, and $(R_1)^*$. Simple modifications of construction (f) give us the positive closure ($^+$) and zero-or-one (?) operators. In the first case, eliminate the ϵ-arc from the new start state to the new final state, and in the second case, eliminate the backward arc.

Example 2: The NFA constructed from the expression $A = .^*0(0 + 1N^*X)$ is shown in Fig. 5. There, and henceforth, we adopt the convention that final states are indicated by double circles. Note the great superfluity of ϵ-arcs. Many of these can be eliminated by considering special cases in the recursive construction

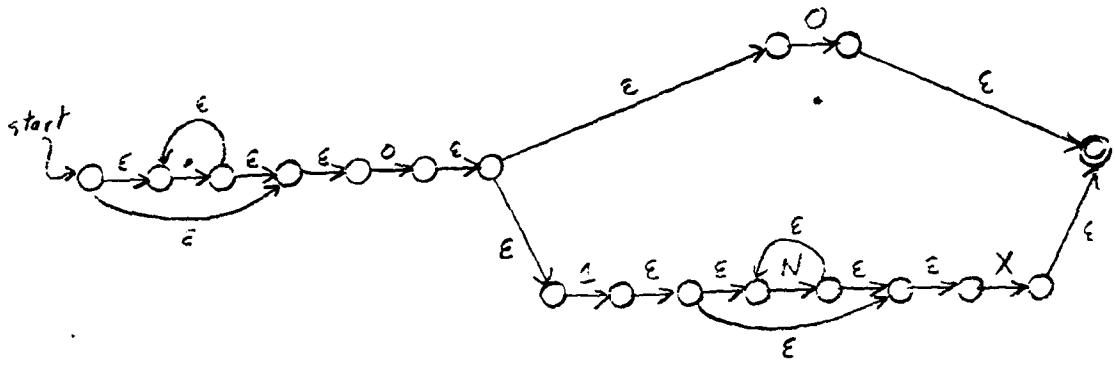


Fig. 5. NFA for expression *A*.

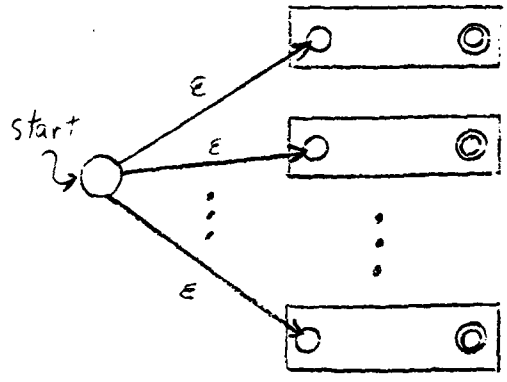


Fig. 6. Combining several NFA's into one.

rules. They can all be eliminated by replacing an ϵ-arc from state p to state q by arcs from p to whatever states q's arcs go to. Also, if q is final, make p final. Then we may eliminate q if it is not the start state and it no longer has any entering arcs. See [HU] for details. []

There is one more step to the construction of an NFA from a collection of regular expressions. We introduce a new start state with ϵ-arcs to the start states of the NFA's for all the regular expressions in the collection. This construction is illustrated in Fig. 6. Note, however, that the various final states of the combined NFA are not indistinguishable. Each represents one of the output signals for the device. In a sense, the NFA of Fig. 6 represents an extension to the usual concept of an NFA, since there are differing output signals associated with the different final states.

Unlike deterministic finite automata, for a given NFA there is not necessarily

a unique minimum-state NFA equivalent to it. Moreover, finding even one of the minimum-state NFA's is a hard combinatorial problem. Nevertheless, often we can simplify the NFA suggested by Fig. 6 considerably, if we first eliminate ϵ-arcs, as suggested above. Then merge any two states that have identically labeled arcs to the same states, unless one, but not the other, of these states is accepting.

However, it is not always clear that these simplifications are beneficial. For example, the NFA's constructed by the McNaughton-Yamada algorithm never have more than two arcs into or out of any state. This limited connectivity makes certain implementations especially simple. If we eliminated ϵ-arcs and merged states, we might get states with large fan-out. When implementing such a state in logic, we might require a tree of limited fan-out gates to represent this state, thus undoing all the benefit of merging states.

V. A PLA Implementation of NFA's

The programmable logic array (PLA) has been used as a systematic implementation of deterministic finite automata (see [MC], e.g.). In these implementations, the states are binary-coded, and the bits representing the new state are computed from the bits of the old state and the current inputs.

While we shall not attempt to describe the mechanics of PLA's in detail here, a rough idea of how they work can be obtained by looking ahead to Fig. 7. There we see a typical PLA, which is a two-dimensional array of wires, divided vertically into an *and-plane* and an *or-plane*. Certain signals (labeled state b,...,state e in Fig. 7) are fed back from the or-plane to the and-plane, with an implied delay of one time unit. New values of the feedback signals and output signals ($\dot{L}, C$, and A in Fig. 7) are computed in the following manner. Imagine signals with value 1 originating at the left end of each horizontal wire. In order for that signal to cross the and-plane, all the vertical wires that it intersects at a dot must have value 1. If the signal reaches the right end of the and-plane, it enters the or-plane and makes 1 every vertical line it intersects at a dot. For example, in Fig. 7, if the fourth and fifth vertical wires (X input and state e) are 1, the top wire will have its signal reach the or-plane and turn on the L output.

An alternative use of PLA's is to use one bit for each state of an NFA. We begin by assuming that the NFA has no ϵ-arcs, but that restriction will be relaxed later. For each arc of the NFA, labeled a and entering state q from state p, we create a term in the formula that tells whether q is one of the states in which the NFA is currently found.† This term is ap; that is, the term has the value true if and only if the input is a and state p was previously on. State q will be on at the next clock tick if and only if one of its terms has the value true; that is, there is

† We shall say a state is *on* when the NFA is in that state.

some arc labeled a to q from a previously on state.

We may conclude from the above remarks that the number of rows of the PLA, each of which corresponds to a term in the formula for one or more states, is no greater than the number of arcs in the NFA.† The number of columns in the PLA is twice the number of states (for the next and previous versions of each state) plus the number of input bits and their complements, if needed.

Example 3: Let us implement the NFA of Fig. 2 as a PLA. We begin by noticing that there are eight states, so in principle, we need sixeen columns for the next and previous states. The inputs are coded by three bits, so we might assume we need six more columns. However, let us assume that the inputs are decoded into the four logical signals $0, 1, N$, and X, by the table in Fig. 1. We thus need a total of 20 columns. Furthermore, if we sum the numbers of labels on each of the arcs in Fig. 2, we see that we apparently need 16 rows. However, we can do considerably better than this if we observe the following.

1. States f, g, and h have no arcs out, and therefore their values need not be fed back, as those values are not used in the terms for any states. However, we must compute values for these states because they are final states. This arrangement saves three columns.
2. State a is always on. Therefore, it need not be computed, and terms involving state a can use "true" in its place. This saves four rows and two columns.
3. The transitions from b to d and g on input 1 require only one row, since the conditions are the same. Similarly, the two transitions from c on input 1 require only one row. Thus two additional rows can be saved.

The resulting PLA has 15 rows and 10 columns. It is shown in Fig. 7, where circles represent connections. []

It is interesting to compare Fig. 7 with the conventional PLA implementation of machines. If we convert the NFA of Fig. 2 to a minimum-state DFA, we find the latter has 11 states. By way of comparison, we chose a particular encoding for states of this DFA. The encoding included the A, C, and L output bits and three other bits (the minimum necessary, since five of the states have $A = C = L = 0$). The state transition table and the encoding is shown in Fig. 8. Blanks in the state code entries indicate that either 0 or 1 may be used, i.e., states with blank S_2 and S_3 entries have four alternative encodings, and we can use the most convenient one when one of these states is the next state.

Obviously, we could use only four bits to encode states, but then we would have to compute the output bits anyway, giving back some of the columns we saved by using shorter codes for the states, and also requiring additional terms to be computed, possibly increasing the number of rows required.

† Recall that technically, an arc with several labels is shorthand for a set of arcs, each with one label and the same source and destination.

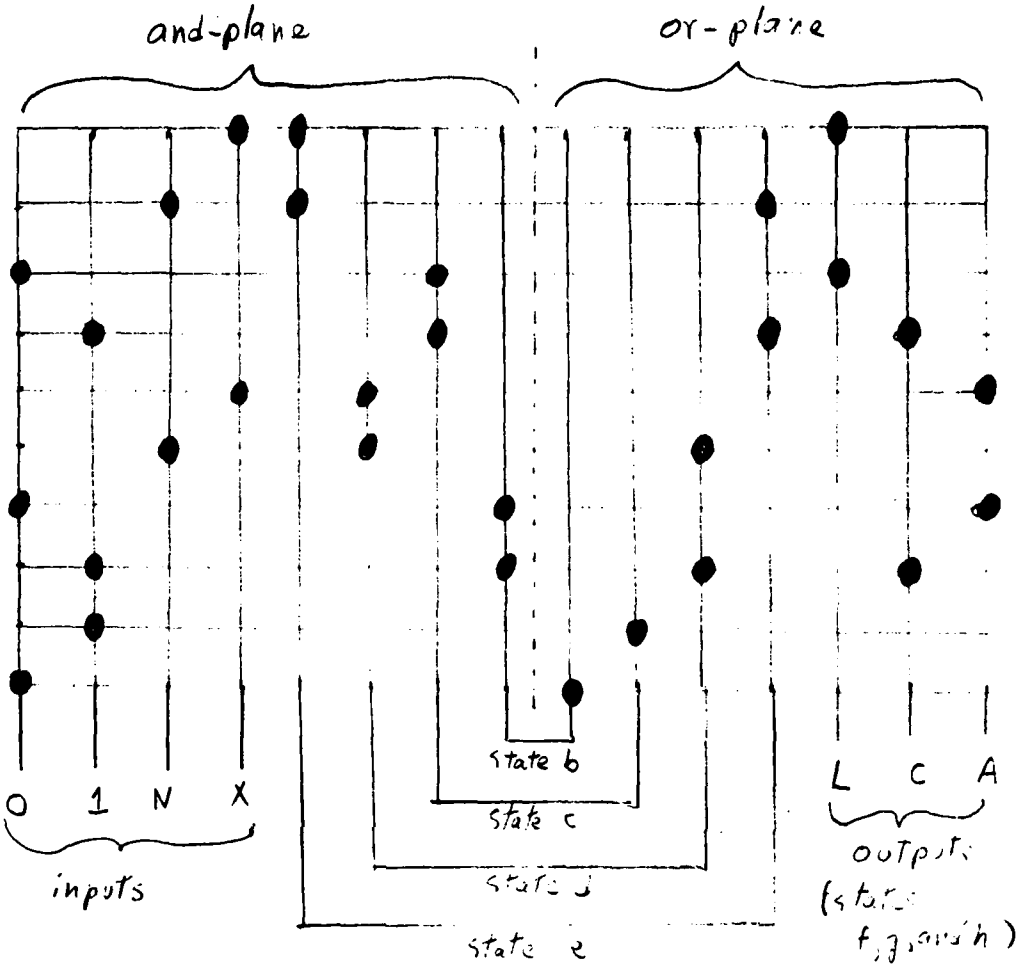


Fig. 7. PLA for machine of Fig. 2.

While we cannot be sure we have a minimum-row PLA, even after restricting ourselves to the state encoding of Fig. 8, a careful selection of terms sufficient to compute the six state bits resulted in a 22 × 26 PLA. That is, there were 26 terms required, and the 22 columns consist of six for the next state, twelve for the previous state bits and their complements,† and four for the input bits.

The product of the dimensions for the conventional PLA implementation is about four times what it is for the NFA-based implementation. The inclusion

† Note that the PLA implementation of nondeterministic finite automata never requires the complements of state bits.

	inputs				state code					
	0	1	X	N	A	C	L	S_1	S_2	S_3
a	b	c	a	a	0	0	0	1	0	0
b	d	e	a	a	0	0	0	0	0	0
c	d	e	a	a	0	0	0	0	0	1
d	d	e	a	a	1	0	0	0		
e	f	g	h	i	0	1	0	0		
f	d	e	a	a	0	0	1	0		
g	f	g	j	k	0	1	0	1		
h	b	c	a	a	1	0	0	1		
i	b	c	h	i	0	0	0	0	1	0
j	b	c	a	a	0	0	1	1		
k	b	c	j	k	0	0	0	0	1	1

Fig. 8. DFA from Fig. 2 and one possible state encoding.

of space around the peripheries of the two PLA's for drivers and clocking gates will reduce the 4:1 ratio somehat, but there is still a clear advantage for the NFA approach to this design problem.

We do not wish to generalize the results of one example to all sequential machine designs. Our method will be advantageous only when the problem at hand lends itself to a succinct description by regular expressions. For example, our methods do not work well on the traffic light example in [MC], because that controller embodies a modulo four counter, and regular expressions are not convenient for expressing counts.

Let us summarize this section by formalizing the relationship between the size of regular expressions and the size of PLA's needed to implement them.

Theorem 1: For every collection of regular expressions of total length n, over an alphabet of at most 2^{i_0} symbols (i.e., i_0 bits are used to code inputs), there is a PLA signaling the recognition of each of these expressions; this PLA has at most $4n$ rows and $4n + 2i_0$ columns.

Proof: First observe that each of the NFA constructions of Fig. 4 introduces at most two new states, and the concatenation construction introduces none. The parse tree for a regular expression with n symbols and m concatenations (which are not represented explicitly by symbols in the regular expression notation) has at most $n + m$ nodes, m of which represent concatenations. Thus, in applying the McNaughton-Yamada construction to each node, in a bottom-up order, we create at most $2n$ states.

We claim that at most $4n$ arcs are created. The union and closure nodes

introduce four each, and the leaves ($\epsilon, \emptyset$, and symbols from Σ) introduce one each. It is easy to show that the number of concatenation operators cannot exceed the number of leaves, so the total number of arcs due to non-concatenation nodes is at most $4n - 3m$. The concatenation nodes contribute another m arcs, for a total that does not exceed $4n$.

The arcs are each labeled by one symbol, so $4n$ is an upper bound on the number of terms needed to express next states in the manner of Fig. 7. Also generalizing Fig. 7, we need $2n$ columns for previous states, $2n$ for next states, and $2i_0$ for the inputs and their complements, for a total of $4n + i_0$ columns.

There is, however, one nuance that is not apparent from Fig. 7, which implemented an NFA with no ϵ-arcs. We have assumed that feedback wires have delay built into them, in the form of *clocking gates* that allow the signal to pass only at certain times, the *clock ticks*. If we have an arc labeled ϵ from state p to state q, then p by itself is one of the terms for q. Most importantly, the use of that term must not be delayed by clocking; q must be turned on at the same clock tick in which p is turned on. That can only be achieved if there are no clock gates in the feedback path for state p; i.e., if next state p is turned on then the previous state p wire must also be turned on.

Fortunately, the above rule causes no inconsistency, because a check of Fig. 4 confirms that if state p has an ϵ-arc out, then all its arcs out are labeled ϵ. Thus we should put clocking gates in the feedback paths of all and only the states whose arcs out are not ϵ-arcs. []

As as consequence of Theorem 1, for fixed input alphabet Σ, we can implement regular expressions of length n in $O(n^2)$ area. It is hard to compare this figure, in general, with the area needed to implement the same expression or expressions by first converting to a DFA. In the worst case, an n state NFA requires 2^n states when converted to a DFA. This DFA requires n bits to represent its states, and in the worst case there could be as many as 2^{n+i_0} terms needed to compute all the next state functions. Thus a PLA as large as $O(n) \times O(2^n)$ cannot be ruled out if we use the conventional approach.

However, in practical examples, it is more common for an n-state NFA to be converted to a DFA with roughly n states. For example, the 8-state NFA of Example 1 becomes an 11-state DFA. If that is the case, then an n-state NFA might be implemented by an $O(\log n) \times O(n)$ PLA. In that case, the DFA-based implementation of machines would be superior. Surprisingly, we shall see in the next section that there is a totally different approach to the implementation of regular expressions that yields a circuit of dimensions $O(\sqrt{n}) \times O(\sqrt{n})$.

Before completing this section, we should comment on the lengths of paths created by the proposed PLA design in Theorem 1. As the PLA has no clock gates in the feedback wires, a signal may have to propagate k times around the PLA

in one clock phase if there is a path in the NFA with k consecutive ϵ-arcs. There is no complete cure for this problem, but we can avoid many of the ϵ-arcs if we modify the construction of Fig. 4 in several ways. For example, we could merge states instead of introducing ϵ-arcs in the union and concatenation constructions. If we do so, we cannot then modify the closure construction to simply identify the start and final states; the construction fails to produce a correct NFA in the general case if all three modifications are made.

VI. A Hierarchical Implementation of Regular Expressions

An inspection of Fig. 4, which shows how to construct an NFA from a regular expression, suggests that we could lay out a circuit directly on a chip, if we represent states of the NFA by appropriate logical elements, represent ϵ-arcs by wires, and represent arcs labeled by input symbols by wires with gates checking for that symbol. The states used in Fig. 4 can be divided into two classes.

1. Those that have ϵ-arcs out, and
2. Those that do not, i.e., they are final states or have arcs leaving that are labeled by an input symbol.

States in the second group are implemented by *latches*, that is, pairs of inverters connected in a loop, with one clock phase to control the output of each. Those in the first group are really nothing more than junction points in the circuit, allowing two signals to merge (through an or-gate) or one signal to fan out into two identical signals (no logic at all is needed here).

When building large circuits from smaller ones, it helps if we view each circuit as a rectangle, as suggested in Fig. 9. Needing a specific convention from among several options, we have chosen to assume that power, ground and two-phase clock signals are passed into the circuit from above and, if needed, are passed through the circuit to another circuit below. Similarly, the bits needed to represent an input symbol from Σ are passed in from the left and can be passed out to the right, unchanged, if needed by another circuit to the right.

There is a signal called *state-in* that, if it is 1, turns the start state of the circuit on at phase one of the clock. An output signal, called state-out, is turned on at clock phase two if the circuit enters its accepting state. In general, phase one of the clock is used to decide which states will be on after processing the current input symbol, and to propagate this information through states with ϵ-arcs leaving. Phase two is used to transfer the decisions made at phase one to the output of the latches that do not have ϵ-arcs out.

Let us suppose we have circuits for regular expressions R_1 and R_2, and we wish to construct a circuit for $(R_1)(R_2)$. We can connect the circuits in cascade as suggested by Fig. 4(e); this connection is shown in Fig. 10(a). Note that the

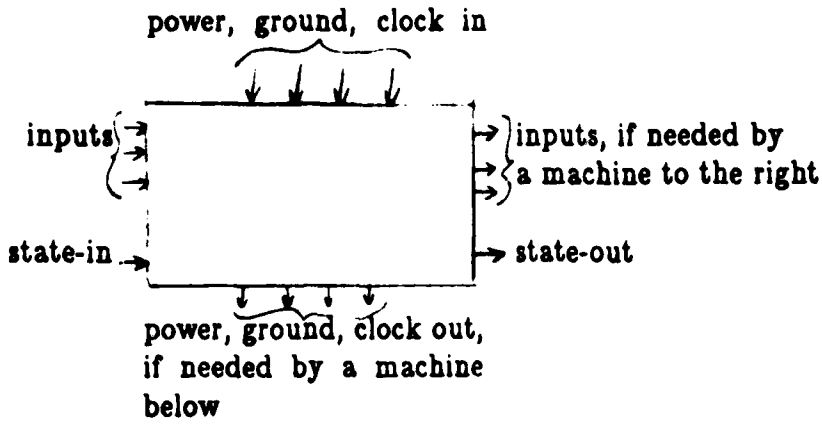
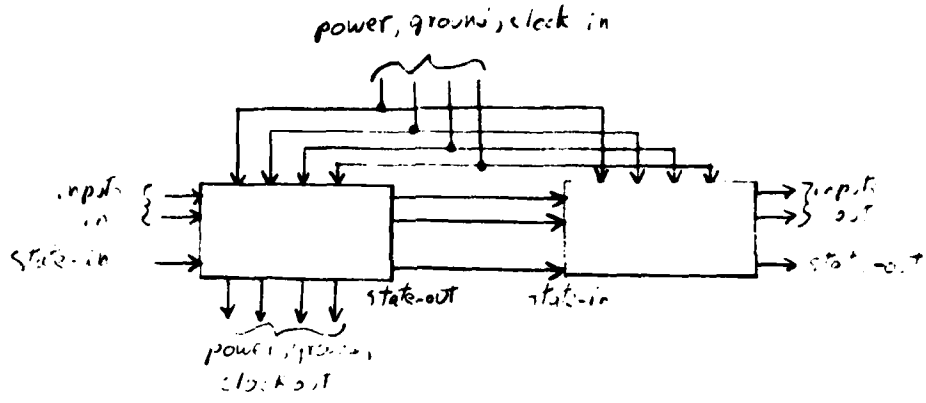


Fig. 9. Format of a circuit implementing a regular expression.

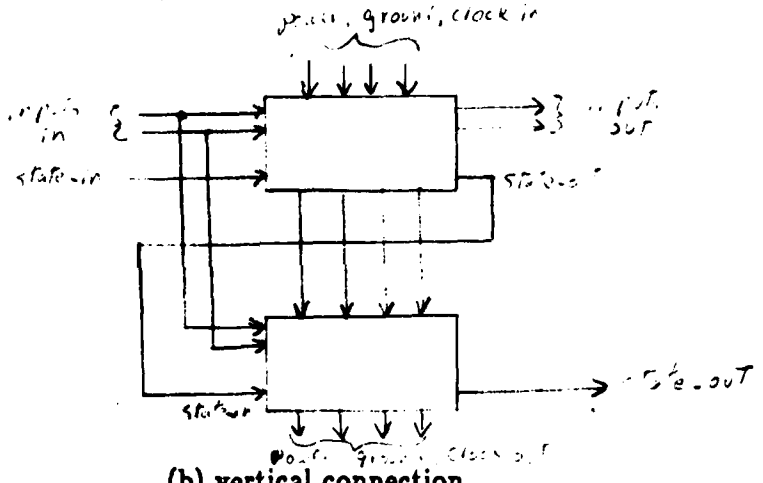
final state of the first machine is given an ϵ-arc out. Thus the latch representing it is no longer needed or appropriate. We must replace it by a junction point or, if there are several input arcs for that state, by an or-gate. As latches can be expected to require more area than a single gate or junction point, we can make this replacement without worrying about the geometry of the circuit, and we shall henceforth assume such changes are made when necessary, not only in the concatenation construction, but in the union and closure constructions as well.

Figure 10(b) shows an alternative organization for the circuit, in which the first machine is placed above the second. Similarly, when we implement the union construction of Fig. 4(d), we can choose to place either constituent circuit above the other or place either to the left of the other. The closure construction, since it does not combine two circuits, gives us little choice; we must simply augment the circuit with surrounding feedback and feed-forward wires as suggested by Fig. 4(f).

The reason we care about the relative positioning of circuits is that we desire each circuit to have an *aspect ratio* (ratio of height and width) near one. For example, if we must combine two circuits that are longer than they are high, we would prefer the vertical connection of Fig. 10(b) to the horizontal connection of Fig. 10(a), since the former has a squarer shape than the latter. The reason, in turn, for desiring an aspect ratio near one is that on the average, we can couple squarish circuits with less waste space than we can couple elongated circuits. For example, neither Fig. 10(a) or (b) is very good if one of the constituent circuits is very tall and thin, while the other is short and wide. Another motivation for keeping aspect ratios low is that the basic circuits, such as latches cannot be designed in a fixed area with a fixed aspect ratio if the area allotted is small and the aspect ratio is high. Thus the rectangles representing the basis constructions



(a) horizontal connection



(b) vertical connection

Fig. 10. Circuit connections.

of Fig. 4(a)-(c) must be allocated space of limited aspect ratio.

Unfortunately, just keeping the aspect ratio within bounds is not sufficient to guarantee efficient use of space, for one of two constituent circuits could be significantly larger than the other. For example, an expression like

$$(\cdots((a_1 + a_2)a_3 + a_4)a_5 + \cdots)a_n$$

forces us to create either a long, thin circuit with many long wires or an L-shaped circuit, if we restrict ourselves to the constructions of Fig. 10. As another example,

$$(((a_1 * a_2) * a_3) * \cdots a_n) *$$

requires n nested feedback loops, so it appears to require $O(n^2)$ space no matter

what we do.† As we shall see, all these problems can be solved, and circuits for these expressions, taking area that is proportional to the length of the expressions, can be generated automatically. Before proceeding to the techniques involved, let us illustrate the basic McNaughton-Yamada construction and also show how the combination of unequally sized circuits tends to waste space.

Example 4: Let us build a circuit for the regular expression $.*0(0 + 1N^*X)$, whose parse tree was given in Fig. 3. Using judicious choices between horizontal and vertical connections when union and concatenation constructions are used, we might obtain the layout‡ suggested by Fig. 11. There, only state-in and state-out wires are shown; input, power, ground and clock wires are omitted. []

VII. A Compact Hierarchical Implementation of Regular Expressions

There are three insights necessary to our implementation of regular expressions. First, we must observe that given any regular expression whose parse tree has $n \geq 2$ leaves, we can find a subtree that has more than $n/3$ but no more than $2n/3$ leaves. For example, the tree of Fig. 3 has six leaves, and its subtree for expression $1N^*X$ has three leaves, which is greater than two and no greater than four. The subtree for $0 + 1N^*X$ would also qualify. This application of "divide and conquer" to binary trees was first used by [LSH].

Once we have found a subtree of about half the leaves, we can build a circuit C_1 for it, and we can build a circuit C_2 for the remaining tree, with a dummy leaf in place of the deleted subtree. This leaf is an imaginary input symbol, and when applying the McNaughton-Yamada algorithm to it, we generate a start state s and a final state f, using the construction of Fig. 4(c), but without the arc. A wire connects state s of C_2 to the start state of circuit C_1, and another wire runs from the final state of C_1 to f. In effect, we have simply removed C_1 from its rightful place between s and f. Note that both states s and f are unnecessary and can always be replaced by junction points, even if latches are created for them initially. The arrangement is sketched in Fig. 12.

Notice how, if C_1 and C_2 are about the same size and shape, they are likely to fit together, either side-by-side, as shown, or one above the other. In comparison, if we had to distort C_2 by "squeezing" C_1 between s and f, we might or might not achieve a compact layout.

As our circuit desgin rules introduced in Section II do not permit us to cross more than three wires at a point, simply laying down the wires shown in Fig. 12

† Note, however, that there is an equivalent regular expression with an $O(n)$ area circuit.

‡ We shall use the term "layout" in what follows to refer to the relative positioning of various subcircuits. The term does not have its more usual connotation of a more more detailed design. However, the positionings we use are intended to be such that a layout, in the usual sense, could be done without repositioning.

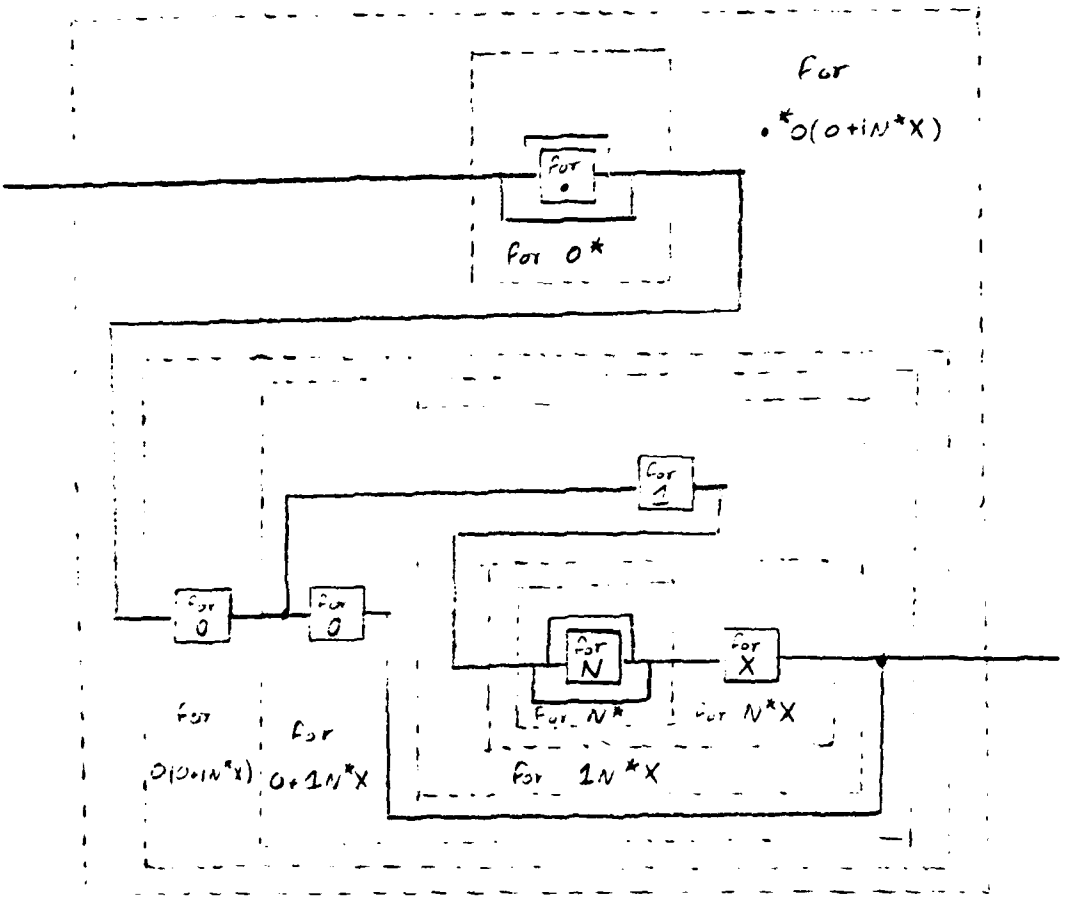


Fig. 11. Straightforward implementation of the McNaughton-Yamada algorithm.

could lead to an illegal circuit. We must therefore "pull apart" C_1 and C_2 at four *channels*, in which the wires can run. The idea is shown in Fig. 13. To create a channel, we select a line across the circuit. Circuit elements and wires running parallel to the line are held at one side of the line, while wires perpendicular to the line are stretched. After stretching some constant amount, there will be room to fit another wire parallel to the line. Circuit elements to which the wire must be connected are, we presume, crossed by the line, and can be moved into the channel to connect with the wire.

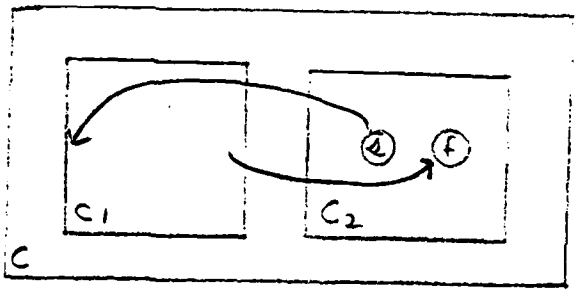


Fig. 12. Divide-and-conquer implementation of regular expressions.

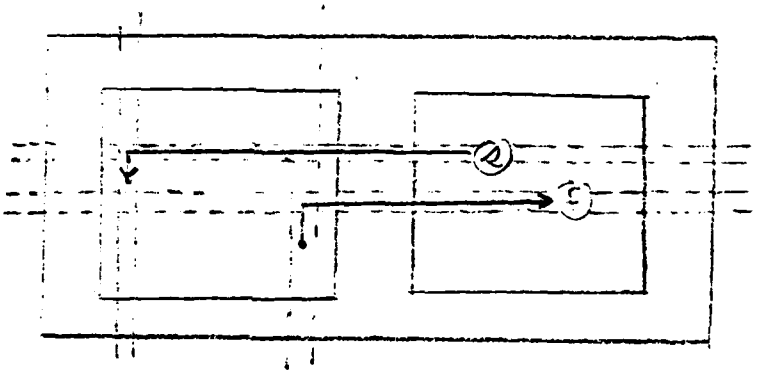


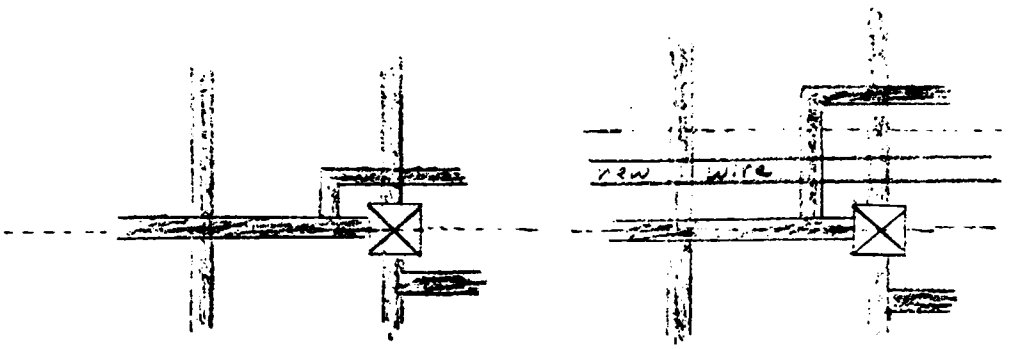
Fig. 13. Channels to carry wires of Fig. 12.

The above method for creating channels will be successful if the original circuit

1. has all wires running horizontally or vertically,
2. never has more than two wires crossing at a point, and
3. uses "circuit elements" from some fixed set, so there is an a *priori* bound on the size of a circuit element.

Condition (3) guarantees that a channel of some fixed width will be sufficient to run a new wire without crossing any circuit elements, and (1) and (2) assure that the new wire will only cross one other wire at a time. Figure 14 gives an example of the channel creation process.

The second insight needed is that even if C_1 and C_2 are about the same size, their aspect ratios and relative sizes might be such that they do not have a common dimension, either the same width (for a vertical arrangement as in Fig. 12),



(a) before creating channel (b) after creating channel

Fig. 14. The channel creation process.

or the same height (for a horizontal arrangement). Unless our recursive circuit layout algorithm works in such a way that when applied to C_1 and C_2 we can expect a dimension in common, we may be forced to connect C_1 and C_2 in a manner that wastes about a quarter of the space. Since the waste can go on at every level of the recursion, we shall have an algorithm that uses area $n^{\log_2 8/3} = n^{1.41}$ to implement a regular expression of size n. This result is superior to obvious methods, but not as good as we can do.

The solution to the above problem is to design our recursive layout algorithm to take as parameters

1. the parse tree of the expression for which we want to design a circuit,
2. the number of nodes of that tree, and
3. the desired aspect ratio, a real number in the range 1/4 to 4.

We assert that there is a constant d such that for each parse tree of $n \geq 2$ leaves, there is a circuit of aspect ratio r and area dn, for any r in the range $1/4 \leq r \leq 4$. It will be shown that for $n \geq 3$ and aspect ratio r between 1/4 and 4, we can always arrange C_1 and C_2, either horizontally or vertically, with a border and channels adequate for connections between C_1 and C_2 and to the "outside world," and with this arrangement, recursive calls to design C_1 and C_2 can be given appropriate aspect ratios between 1/4 and 4, so that C_1 and C_2 will have a side in common.

Example 5: Let us consider how the parse tree of Fig. 3 would be processed recursively by the circuit layout algorithm. First, we must find a node from which between 1/3 and 2/3 of the leaves descend. The preferred candidate is the root of the subtree for $1N^*X$, which divides the leaves into two equal parts. As the initial call to the circuit routine would normally ask for a square circuit (aspect

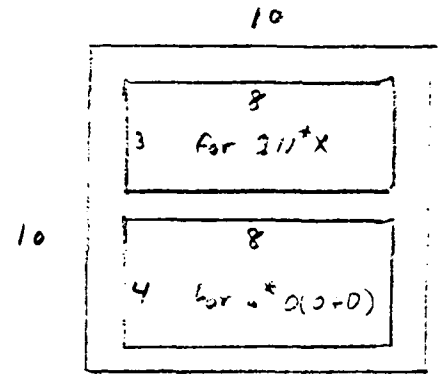


Fig. 15. Initial layout.

ratio 1:1), we may position the subcircuits for $.*0(0+D)$† and $1N^*X$ either horizontally or vertically; let us choose the latter. As the first expression has four leaves and the second has three,‡ the heights of the two subcircuits should be in the ratio 4 to 3. They are given the same width. A sample arrangement, in which the entire circuit is allocated a 10×10 area (in some units), and borders are one unit wide, is shown in Fig. 15.

We now lay out the circuits for $1N^*X$ and $.*0(0+D)$ in the rectangles of aspect ratios 3:8 and 1:2, respectively. We should, in principle, divide each of these expressions into two parts and recursively synthesize their circuits from circuits for the parts. However, we omit the details of those recursive calls. One circuit that could result is shown in Fig. 16. []

The third necessary insight is that two or more consecutive applications of the closure operator are equivalent to one. That is, for any regular expression R we have $(R)^* = ((R)^*)^*$. As a consequence, we may eliminate superfluous *'s and view regular expressions as if all the operators were binary operators chosen from the list: union, concatenation, union-then-closure, and concatenation-then-closure. We use the constructions of Fig. 4(d) and (e), followed by (f), when closure is desired, to build circuits, just as in the McNaughton-Yamada algorithm. Operands are either single symbols, or symbols to which closure is applied, and circuits for operands can be constructed by Fig. 4(a)–(c) optionally followed by 4(f). Note that this algebraic simplification is necessary to avoid awkward situations like the expression $a^{**\cdots*}$ which, if the McNaughton-Yamada algorithm were applied blindly, would result in a circuit of area $0(n^2)$.

The heart of the circuit layout algorithm is the recursive procedure LAYOUT

† D stands for the particular dummy symbol used as a placeholder for the expression $1N^*X$.

‡ However, if we are careful, we can avoid allocating circuit area for the dummy symbol, which we know will be represented in the circuit by junction points only.

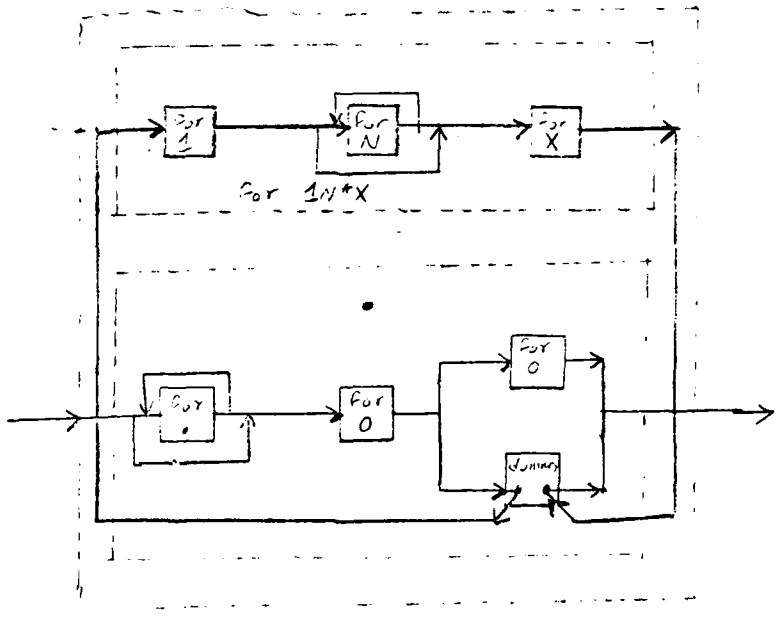


Fig. 16. Complete layout for the expression of Fig. 3.

sketched in Fig. 17. The algorithm itself is a call to LAYOUT($T, n, 1$), where T is the parse tree for a regular expression of length n, i.e., T is assumed to have n leaves. *In LAYOUT, Σ is assumed to be a fixed input alphabet defined globally,* so its size may be regarded as constant. Also b is a constant chosen large enough that the total width of the channels and border area, either in the horizontal or vertical direction, is bounded above by b. Note that channels need to carry one wire each, while border areas may need to carry $4 + \log_2 \|\Sigma\|$ wires, one for each of the input bits, and one each for power, ground, and the two clock phases. Finally, $A(n)$, the area alotted to a circuit for a regular expression of length n, is a function of the form $dn - e\sqrt{n} - f$, whose adequacy we shall show in the next section.

```
function LAYOUT(T, n, r); {T is a parse tree with n ≥ 2 leaves. LAYOUT
    returns a circuit of area A(n) and aspect ratio r; we assume without loss of
    generality that r ≥ 1; otherwise rotate the layout 90°. The circuit returned
    has only horizontal and vertical wires, and at no point do more than two wires
    overlap.}
begin
        if n = 2 then
                use McNaughton-Yamada algorithm to produce a circuit C
        else {n ≥ 3}
        begin
                select a node N of T such that N is the root of a subtree with
                    n_1 nodes, where n/3 < n_1 ≤ 2n/3;
                let T_1 be the tree with root N;
                let T_2 be T with the subtree rooted at N replaced by a dummy
                    leaf;
                n_2:=n − n_1 + 1; { T_2 has n_2 leaves}
                {now we perform horizontal decomposition, as in Fig. 14}
                h:=√(A(n)/r); {h is the height of circuit C of Fig. 14}
                h_1:=h − b; {height of C_1 and C_2 in Fig. 14}
                w_1:=(h * r − b) * n_1/(n_1 + n_2); {width of C_1}
                w_2:=h * r − b − w_1; {width of C_2}
                r_1:=w_1/h_1; r_2:=w_2/h_2; {aspect ratios for C_1 and C_2}
                C_1:=LAYOUT(T_1, n_1, r_1);
                C_2:=LAYOUT(T_2, n_2, r_2);
                Separate circuits C_1 and C_2 to make two horizontal and four
                    vertical channels for their interconnections, as shown in Fig.
                    14. Figure 15 showed how this operation could be done in
                    such a way that wires could be laid along the channels without
                    violating the circuit design rules we have assumed;
                Add border around C_1 and C_2, and run wires for inputs, etc., to
                    feed both circuits and to produce wires out of the bottom and
                    right edge, as indicated in Fig. 9;
                Call the resulting circuit C;
        end;
        return C;
end
```

Fig. 17. The recursive procedure LAYOUT.

VIII. Analysis of the Algorithm

We now show that LAYOUT can be made to use $O(n)$ area, by showing that a linear function $A(n)$ can be chosen. We must pick $A(n)$ to satisfy the following constraints.

1. The area $A(n)$ available for C in Fig. 14 must not exceed area $A(n_1) + A(n_2)$ used for C_1 and C_2 plus the area needed for borders and channels.
2. The aspect ratios of C_1 and C_2 must be in the range 1/4 to 4 if that of C is.
3. $A(2)$ must be large enough that we can build a circuit for any regular expression of length 2 in that area, with any aspect ratio up to 4.

Lemma 1: If $A(n) \geq 25b^2$, and C of Fig. 14 has aspect ratio 4 or less, then C_1 and C_2 have aspect ratios in the range 1/4 to 4.

Proof: The extreme cases we must consider are when $n_1 = 2n_2$† and either

a) C has aspect ratio 1, in which case C_2 could be too tall and narrow, or
b) C has aspect ratio 4, in which case C_1 could be too short and wide.

Let the height of C be h. Then in case (a) the height of C_2 is $h - b$ and its width is $(h - b)/3$, so its aspect ratio is 3, satisfying the lemma. In case (b), the height of C_1 is again $h - b$, and its width is $\frac{2}{3}(4h - b)$. Thus its aspect ratio will be 4 or less provided

$$h - b \geq (\frac{1}{4})(\frac{2}{3})(4h - b)$$

that is, $h \geq \frac{5}{2}b$. Since the area of C in this case is $4h^2$, the lemma follows. []

Theorem 2: There exist positive constants d, e, and f such that for all $n \geq 2$, the function LAYOUT will succeed in producing a circuit if the allotted area $A(n)$ is $dn - e\sqrt{n} - f$.

Proof: Let us, for the moment, assume that d, e, and f satisfy the lemma for $n = 2$. Notice that when we divide a tree T of $n \geq 3$ leaves into T_1 and T_2 in function LAYOUT, neither n_1 nor n_2 can be 1. Thus we can attempt to prove by induction, with a basis of $n = 2$, that area $A(n) = dn - e\sqrt{n} - f$ suffices for LAYOUT to produce a circuit. To develop the induction, let $n \geq 3$, and $n_1 = an$ for some constant a, $1/3 < a \leq 2/3$. Then the areas of C_1 and C_2 are $A(an)$ and $A((1-a)n+1)$, respectively, since $n_1 + n_2 = n + 1$. Observe that C_1 and C_2 are, by Lemma 1, of limited aspect ratio, and their areas are also chosen by LAYOUT to be of limited aspect ratio. Hence the borders and channels in Fig. 14 have area that is proportional to any side of C_1 or C_2, the constant of proportionality naturally depending on which side is chosen. Specifically, there is some constant c such that the extra area of circuit C, beyond that of C_1 and C_2, is at most

† Since $\frac{2}{3}n \geq n_1 > \frac{1}{3}n$ and $n_1 + n_2 = n + 1$, it is easy to show that n_1/n_2 must be in the range 1/2 to 2.

$c\sqrt{A(an)}$. Thus

$$A(n) = \max_{1/3<a\leq 2/3}\left[A(an) + A((1-a)n+1) + c\sqrt{A(an)}\right] \tag{1}$$

We assume $A(m) \leq dm - e\sqrt{m} - f$ for $2 \leq m < n$, and show that the same holds when $m = n$. By (1) it suffices to show that

$$dn - e\sqrt{n} - f \geq$$

$$\max_{1/3<a\leq 2/3}\left[adn - e\sqrt{an} + (1-a)dn + d - e\sqrt{(1-a)n+1} - 2f + c\sqrt{adn}\right] \tag{2}$$

In the last term of (2), $\sqrt{A(an)}$ has been conservatively replaced by $\sqrt{adn}$. Simplifying (2) we obtain

$$0 \geq \max_{1/3<a\leq 2/3}\left[e\sqrt{n} - e\sqrt{an} - e\sqrt{(1-a)n+1} + d - f + c\sqrt{adn}\right] \tag{3}$$

Dividing (3) by $-e\sqrt{n}$ yields

$$0 \leq \max_{1/3<a\leq 2/3}\left[\sqrt{a} + \sqrt{1-a+\frac{1}{n}} - 1 + \frac{f-d}{e\sqrt{n}} - \frac{c}{e}\sqrt{ad}\right] \tag{4}$$

The first three terms on the right of (4) sum to at least 0.39. The next term can be made zero if we pick $f = d$. The last term is no more than 0.28 if we choose $e = 3c\sqrt{d}$. Thus, for these choices of e and f in terms of d, (4) is satisfied; hence so is (2).

Now we must satisfiy the condition that $A(2)$ is adequate to hold all circuits for regular expressions of length 2. We simply observe that we can pick d so that $A(2) = d - 3c\sqrt{d}$ exceeds any quantity we choose, so an adequate value of d can be found. []

One may wonder if the linear bound on area for a general regular expression is the best that could be achieved. We believe it is, because of another assumption that is generally made ([BK], e.g.) about integrated circuits, that there is a finite (as opposed to infinitesimal) amount of area needed to store one bit of information. If that is the case, then we cannot improve on the linear growth rate in Theorem 2, because there are regular expressions of length proportional to n that require n bits of information to be remembered if we are to recognize them.

A simple example is the family

$$(0+1)^*1(0+1)(0+1)\cdots(0+1)$$

where n terms $(0+1)$ follow the 1. For each n, this regular expression denotes the set of strings of 0's and 1's that have a 1 n positions from the end. Clearly, we must remember the last n inputs if we are to recognize all strings in the language.

IX. Implementation Considerations

Theorem 2 gives an upper bound on the required area of a circuit. However, to design a circuit we do not necessarily wish to use the function LAYOUT starting with the maximum theoretically needed area, which we have called $A(n)$. Rather we should try it with half this area, initially. If we are successful in producing a circuit, try the algorithm again with half as much area. After the i^{th} try, if successful, reduce the area by $2^{-(i+1)}$ of the maximum possible area, and if we fail to produce a circuit, increase the allotment by this amount. By this binary search technique we can obtain the minimum possible area for a square circuit to within a maximum error of one part in 2^i in i tries after we have our first failure. Surely $i = 10$ is adequate in practice. As the algorithm itself requires only $O(n^2)$ time at most, we do not expect that repeated tries with different areas would be odious.

Another variable that might be adjusted is the initial aspect ratio. It is possible that a ratio other than one leads to a circuit of smaller area than if the initial aspect ratio were restricted to one. However, "tuning" this parameter is not done as systematically as it was for the area parameter. In particular, there is no reason to believe that area varies monotonically with aspect ratio, so binary search cannot be used.

Another potentially promising modification of the algorithm is to allow aspect ratios greater than 4 in certain circumstances. There must be some limit on the aspect ratio, as circuits implementing single states cannot be designed, in a rectangle of fixed area, if that rectangle is too long and thin. However, there is nothing sacred about the limit 4.

There are a number of other ideas that could improve the quality of the circuit. Among them are:

1. Use a catalog of circuits for small n. Indeed, the proof of Theorem 2 implies that circuits for regular expressions of with only one operand will be selected by table lookup. Circuits for some larger expressions could be stored.
2. As we have mentioned, the second state in the constructions for $n = 1$ (Fig. 4(a)-(c)), unless it is an accepting state, will eventually get an ϵ-arc out. Thus a latch for that state is not needed, and it can be replaced by a junction

point. We can predict, by examining the entire tree, which states will be accepting states of the entire circuit, and we can then save space by representing the others by points or or-gates, rather than latches, when they are first introduced into the circuit.

3. Do not pass a wire (power, input, etc.) through a subcircuit if it is not needed anywhere to the right or below.
4. When LAYOUT calls itself recursively, optimize the area of one subcircuit before selecting the area and aspect ratio of the second. This technique may cause the aspect ratios within the second circuit to exceed tolerable limits, which in turn may require modification of the first circuit. Thus the time spent by LAYOUT mat be exponential in n, but even this amount of time may be worthwhile if it leads to a superior circuit design.

X. Related Work

The ideas of divide-and-conquer layout and of channel creation were also used independently by C. Leiserson and by L. Valiant [V]. In terms of [Lei], we could show Theorem 2 by proving a "2-separator theorem" for the graphs of nondeterministic finite automata that we obtain by the McNaughton-Yamada construction. Strictly speaking, the connections needed for supplying, input, power, and so on, must by ignored in that theorem and handled outside the framework of [Lei].

References

[AU] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, Mass., 1977.

[BK] R. P. Brent and H. T. Kung, "The area-time complexity of binary multiplication," CMU-CS-79-136, Dept. of C. S., CMU, July, 1979.

[EZ] A. Ehrenfeucht and P. Zeiger, "Complexity measures for regular expressions," *J. Computer and System Sciences* 12:2 (April, 1976), pp. 134-146.

[G] J. P. Gray, "Introduction to silicon compilation," 16th *Design Automation Conf. Proceedings*, pp. 305-306, June, 1979.

[HU] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation," Addison Wesley, Reading, Mass., 1979.

[J] D. Johannsen, "Bristle blocks, a silicon compiler," 16th *Design Automation Conf. Proceedings*, pp. 310-313, June, 1979.

[Lei] C. E. Leiserson, "Area-efficient graph layouts (for VLSI)," unpublished memorandum, Dept. of C. S., Carnegie-Mellon Univ., Aug., 1979.

[Les] M. E. Lesk, "LEX—a lexical analyzer generator," CSTR-39, Bell Laboratories, Murray Hill, N. J.

[LSH] P. M. Lewis II, R. E. Stearns, and J. Hartmanis, "Memory bounds for the recognition of context-free and context-sensitive languages," *Proc. IEEE Sixth Annual Symp. on Switching Circuit Theory and Logical Design*, pp. 191-202, Oct., 1965.

[MC] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison Wesley, Reading, Mass., 1980.

[MY] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IEEE Trans. on Computers* 9:1 (March, 1960), pp. 39-47.

[S] D. P. Siewiorek, "A survey of research on synthesis, evaluation, and automation of digital systems at CMU," 1979.

[T] C. D. Thompson, "Area-time complexity for VLSI," *Proc. Eleventh Annual ACM Symposium on the Theory of Computing*, pp. 81-88, May, 1979.

[V] L. Valiant, "Universality considerations in VLSI circuits," unpublished memorandum, Dept. of C. S., Edinburgh Univ., Dec., 1979.