June 1983 Report No. STAN-CS-83-972

# Experience with a Regular Expression Compiler

by

Anna R. Karlin, Howard W. Trickey, and Jeffrey D. Ullman

**Department of Computer Science**

Stanford University
Stanford, CA 94305

# EXPERIENCE WITH A REGULAR EXPRESSION COMPILER†

Anna R. Karlin
Howard W. Trickey
Jeffrey D. Ullman
Stanford Univ., Stanford CA

The language of regular expressions is a useful one for specifying certain sequential processes at a very high level. They allow easy modification of designs for circuits, like controllers, that are described by patterns of events they must recognize and the responses they must make to those patterns. This paper discusses the compilation of such expressions into reasonably compact layouts. The translation of regular expressions into nondeterministic automata by two different methods is discussed, along with the advantages of each method. A major part of the compilation problem is selection of good state codes for the nondeterministic automata; one successful strategy is explained in the paper.

## I. The Regular Expression Language

We shall give a brief introduction to the language of regular expressions; for more information on this language and on nondeterministic finite automata, the reader is referred to Hopcroft and Ullman [1979]. Regular expressions consists of operators and operands. The operands are abstract symbols that represent events in terms of combinations of wires. Events are assumed to occur at discrete times, so regular expressions define synchronous systems.

### Operators

The operators of our language are:

1. Juxtaposition (no operator) standing for sequencing of events. That is, if regular expressions $R$ and $S$ each represent a set of events, then RS represents the set of events consisting of an event of $R$ followed by an event of S.
2. The + operator standing for union. Thus, $R + S$ stands for the set of events that are either events of R or events of S.
3. The unary postfix operator * standing for the closure, or "any number of" operator. Thus, $R^*$ stands for any sequence (including the null sequence, denoted $\epsilon$) of events in $R$.

In our language we also use the shorthand operators:

4. $R^{++}$ stands for one or more occurrences of events in $R$, that is, $RR^*$.
5. $R?$ stands for zero or one occurrence of an event in $R$, that is, $R + \epsilon$.

**Example 1:** Let $a$ and $b$ be events, i.e., abstract symbols of our language. Then $ab$ stands for an a followed

† Research supported by DARPA contract MDA- 80-C-0107 and NSF grant MCS-82-03405.

immediately by a 6; $ab^*$ stands for an $a$ followed by any number of b's, i.e., {a, $ab$, $abb$, . . . }; $ab^{++}$ stands for $\{ab, abb, \ldots\}$. Also, $a$ + 6 stands for either an $a$ or a 6, and (a + $b)^*$ stands for any sequence of u's and b's, in any order. □

### Special Operands

In addition to abstract symbols as operands, we allow two other operands.

1. A dot (.) is an operand that matches any combination of input wires, i.e., it is always **seen.**
2. The symbol # is never seen, no matter what input wires are on. Although seemingly without purpose, this symbol is essential when we use state names in our expressions, as described below.

### Line Declarations

Our expression language **has** declarations of five types: lines, symbols, outputs, states, and subexpressions. The lines are wire names, from which the symbols are constructed. For example,

line x, y[8]

declares x to be the name of a wire, and y to be the name of a group of eight wires, which may be referred to individually in symbol definitions as $y[1], \ldots, y[8]$.

### Symbol Declarations

Symbols are the operands of regular expressions mentioned above. Each is defined by a set of wires that must be on and a set that must be off. Thus,

symbol zap(x, y [1], -Y [3])

declares that abstract symbol ***zap*** is seen whenever wires x and $y[1]$ are on, and wire $y[3]$ is off. Any other wires are ignored when deciding whether event *zap* is seen. Note **that, unlike the usual conventions in** automata theory, we allow more than one symbol to be seen at **the** same time. for example, we could define

symbol zip(y[1], y[2])

and see both zip and ***zap*** at the same time, if x, $y[1]$, and $y[2]$ arc on, while $y[3]$ is off.

### Output Declarations

Output symbols are embedded in the regular expression and represent output wires. The exact rules determining when an output wire is raised arc complicated, and the details appear in Ullman [1983]. However, the general idea is that if $R$ is a regular expression, $U$ is an output symbol and $RU$ a subexpression of the complete regular expression, then wc raise output $U$ immediately after seeing a sequence of inputs that forms an event of R. An example will help make the ideas known. If we declare

output u, v

and write the regular expression

aip (zip U + zap U V)*

then after the first event, which must be *zip* or no output is ever made, we look for any sequence of events *zip* and *tap.* Each time we see *zip,* we raise output $U$, and each time we see *zap* we raise both $U$ and *V.* If at any time we see neither *zip* nor *zap,* we not only raise no output, but recognition of the regular expression has "derailed," and we never make any more output.

### State Declarations

State names arc used like "goto's" in the regular expression. While the regular expression language is most appropriately used in situations where there is little need for explicit state transitions, we have found that the occasional use of such transitions is almost cssential. With this feature, our language has all the power of deterministic finite automaton languages, like SLIM (Henncssy [1981]), while also offering the expressive power of regular expressions where that is more appropriate. WC can declare $s$ to be a state by the declaration

**state s**

Then, in the regular expression, there will bc one occurrence of $s$ followed by a colon; thus $s$ : marks the "label" position of a, where control transfers whenever it is determined that state $s$ is entered. Often, we find a : preceded by #, the symbol that is never matched, so that control does not accidentally reach state $s$ without an explicit transfer to that state.

In the expression there can be any number of occurrences of $s$ not followed by a colon; these are the "goto's" to state $s$. As with output symbols, a state symbol is activated when a match for the preceding regular expression is recognized. The reader should be reminded that because of the inherent nondetcrminism in the input recognition process defined by regular expressions, the use of states can bc more general than in a deterministic finite automaton language. For cxamplc, two or more goto's to different states could be activated at the same time, causing us to be in both states at once.

### Program Structure

The fifth kind of declaration is a subcxprcssion, where an identifier is declared to stand for a regular cxprcssion. Thus

```
line x
symbol
    zero(-x)
    one(x)
output OUT
;
.* one one (one zero? zero?)++ OUT
```

**Fig. 1.** Bounce filter regular expression.

```
subexp string = (zip + zap)*
```

declares ***string*** to stand for any sequence of ***zip***'s and ***zap***'s, so

```
string zip zip string
```

stands for any sequence of ***zip***'s and ***zap***'s with at least two ***zip***'s in a row.

After all declarations for a program, there is a single semicolon, followed by a single regular expression. The limitation to one expression is not significant, since there can be any number of output symbol occurrences in the expression. We can even use the # operand to simulate a multistate automaton by using an expression of the form

```
statel: expression1
+ # state2: expression2
        ...
+ # Staten: exprcssionn
```

Presumably, each of the expressions has within it one or more state symbols, which cause transitions to other states.

Example 2: A ***bounce filter*** is a device with a single input and single output; the output will generally agree with the input, but we wish to ignore small "bounces," where for a small number of cycles the input changes and then returns to its original value. For our example, we shall ignore one or two consecutive 0 or 1 inputs that do not match their surroundings. The regular expression that defines this output as a function of the input is shown in Fig. 1.

The first line of text declares $x$ to be a wire; this wire is the only input wire in this example The next two lines declare *zero* and ***one*** to be abstract symbols, seen, respectively, when the input wire is off and on. The fourth line declares OUT to bc an output signal, the only output in this example. 'Then comes the obligatory semicolon, and finally the expression itself. The expression says that the output OUT is to be raised whenever we see an input pattern that requires the output to be 1. That is, we may see anything at all (indicated by the .*), then two *one*'s and one or more groups represented by the expression

```
line cars, tol, tos
output RESET, IIWYGREEN, IIWYYEL, FARMGRN, FARMYEL
state highway, farm
symbol
     carstol(cars, tol)
     carsntol( cars, -tol)
     nocars(-cars)
     timeup(tol)
     notol(-tol)
     switch( tos)
     wai t(-tos)

highway: (nocars+notol)* IIWYGREEN
     carstol RESET
     wait* IIWYYEL switch farm RESET
+ # farm: carsntol* FARMCRN
     (nocars+timeup) RESET
     wait* FARMYEL switch highway RESET
```

Fig. 2. Traffic light controller.

one zero? zero?

**that** is to say, each group consists of a 1 followed by up to two optional 0's. The 1 from the first group forms the third consecutive 1, along with the l's that match the two carlier symbols *one* in the expression. After these three l's, there cannot be three O's in a row, no matter how many groups are present, so the output in response to any input that matches the pattern of the expression given in Fig. 1 should be 1. □

Example 3: Now, let us see how to write as a regular expression the famous traffic light controller from Mead and Conway (19801. This problem involves a light at the crossing of a highway and farm road. A sensor detects cars waiting on the farm road to cross the highway; its output is the line *cars* in Fig. 2. Two timing signals are also used to control the light. The short time out signal, *tos* in Fig. 2, indicates that enough time has elapsed from the last time the $RESET$ output signal was raised that a yellow light may be turned to red. The long time out, *tol* in Fig. 2, is used to measure the minimum time, from the last RESET, that we shall allow the highway to be green, even when cars are waiting on the farm road, and also to measure the maximum time that we shall allow the farm road to be green, even if there is a steady stream of cars on that road.

Let us examine the parts of the expression in detail. Starting from the highway state, the subcxpression

(nocars + notol)*

matches any sequence of events in which either thcrc are no cars waiting, or the long timeout interval has not clapscd. As long as that is the case, the highway stays green, as rcllcctcd by the fact that the IIWYGREEN output follows this subcxprcssion. Then if both the cars and ***tol*** signal are on, the input no longer matches

(**nocars** + **notol**)*

but it matches the longer expression

(**nocars** + notol)* carstol

because the ***carstol*** symbol is seen **whenever** both the wires ***cars*** and ***tol*** are on. Note that the output and state symbols, such as **HWYGREEN**, are ignored **when** considering subexpressions of the complete expression that might match the input.

In response to a match of the above expression we emit signal RESET, which starts the counter for the purpose of measuring the time of the yellow light. Any input that matches the above expression also matches

(**nocars** +notol)* carstol wait*

since ***wait*** can occur zero times in a match of ***wait*.*** Thus, at the same time RESET is signaled, HWYYEL is also signaled, and the highway light turns yellow, while the farm road remains red.

As long as the short timeout **period** has not elapsed, ***wait*** will continue to be seen, so the above expression will be matched, and IIWYYEL, but not RESET, will be emitted continuously. Then, when ***switch,*** the abstract symbol that represents **the** ***tos*** wire going on, is seen, we can no longer emit HWYYEL, because the input seen since we entered the ***highway*** state no longer matches the above expression. However,

(**nocars** + notol)* carstol wait* switch

is matched. Thus, we emit **thc** two following signals, *farm* and **RESET.** The first takes us to the farm **statc,** and since that state is **followcd** by subcxpression ***carsntol*,*** which **is** matched by the empty string, we immediately signal FARMGRN as well. That causes the farm **road** to become green and the highway red. The events following the farm state are similar to those just discussed for the highway state, and we shall omit a detailed description.

**We** might note that although the **traffic** light is inherently a four-state device, we used only two states, and in fact, we did so only for convenience; we could do without **statcs** altogether. **There** is really no need for **thc** farm **statc, bccausc** whcncvcr WC **cntcr** it, we would "fall through" to it anyway. WC can do without **thc** ***highway*** if we put the closure operator around the whole expression, thus causing the cycle to **rcpeat** indefinitely. The state-free expression for the **traffic** light is

((**nocars** + notol)* **IIWYGREEN** carstol RESET
wait* **IIWYYEL** switch **RESET**
**carsntol* FARMGRN** (**nocars** + **timcup**) RESET
wait* **FARMYEL switch RESET**)*

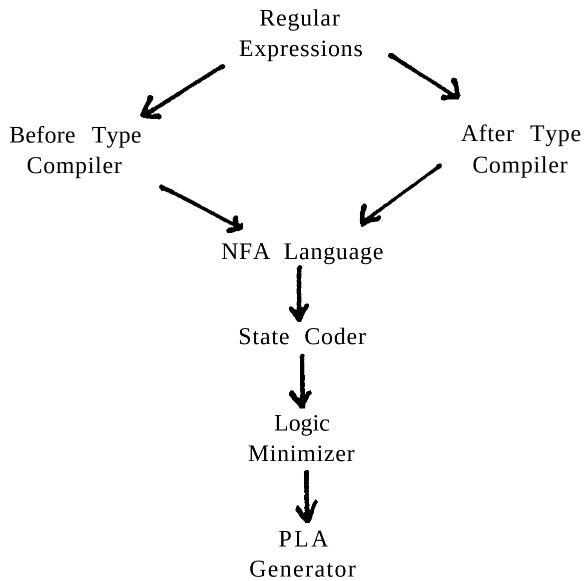



Fig. 3. Outline of RE Compiler.

## II. The Compilation Strategy

Figure 3 outlines the way regular expressions are compiled into PLA's. The language of nondeterministic finite automata (NFA's) is used as an intermediate language We shall not detail the language here, as it is fairly conventional. The important thing to remember is that the nondeterministic states each correspond to a single operand of the expression. There are two reasons we prefer to work from the NFA, rather than converting regular expressions into deterministic automata and using standard state-coding heuristics.

1. Sometimes the regular expression is short, yet the number of states of the deterministic automaton is enormous. We worked with one example of an expression, that described pattern matching with don't care's, where the regular expression has 72 operands, yet the deterministic automaton has over eight million states. By coding the NFA directly, we were able to get a PLA with 24 feedback wires, which is only one more than the minimum possible for the implementation of an 8,000,000 state machine.

2. The regular expression gives us important clues to a good state coding. In particular, we shall see below that we can always find a PLA implementation with one term per operand, i.e., one term per NFA state. If we converted to a deterministic automaton, we might lose some of the useful information and wind up with a PLA with more terms, unless we spent a great deal of effort optimizing the coding.

The "before" and "after" type compilers are really implemented by a switch on a single compiler; we shall discuss the difference below. The details of the compiling algorithms involved in translating regular expressions to NFA's by either method are found in Trickey [1982] and Ullman [1983].

We have experimented with several strategies for the state coder. They all depend on knowing the conflict matrix of the NFA, i.e, which pairs of states can be on at the same time. We shall have more to say about these strategies later.

The output of the state coder is a PLA personality. This personality has the number of its terms reduced by a program called GRY, written by Hemachandra [1982] and basedon algorithms described in Hachtel et al. [1982]. The output of the minimizer is fed to a PLA generator written by Kevin Karplus.

### III. The Partition of Regular Expressions

The first thing the regular expression compiler does is break up the given expression into manageable pieces; we try to have each piece represent about fifty operands. One of the important features of the regular expression approach to design is that expressions can be broken up into subexpressions that have very little interaction; in essence the outer expression "calls" the subexpression at exactly one place, and the call can be represented by a pair of wires carrying a startup signal to the subexpression recognizer and a completion signal back to the caller. For example, the bounce filter of Example 1 could have its expression broken into

$$main = .^{*}\ one\ one\ sub^{++}\ OUT$$
$$sub = one\ zero?\ zero?$$

It is important to realize that the circuit recognizing the subcxpression can receive start signals more than once, and may even be working on more than one "call" at a time, but this activity is a correct implcmcntation of regular expressions. We should also be aware that if there are state "goto's" connecting a subexpression to its environment, then more than one pair of wires will be necessary for the interconnection.

Before translating the subcxprcssions into NFA's, the compiler does a certain amount of algebraic manipulation of the subexpression to reduce the number of NFA states needed, if possible. For example, we left- and right-factor expressions, so

$$abc + adc$$

becomes

$$a(b + d)c$$

The motivation for splitting the expression into small pieces is that the PLA implcmcntation degrades in both speed and in area used per regular expression operand, as the number of operands grows. That is, the area of a PLA for an n-operand expression could bc proportional to $n^2$. The reason WC do not therefore break the expression into PLA's of size 1 is that the PLA cost also has an overhead term. The result is that to get the lowest ratio of operands to PLA area we should USC subexpressions of about 15-25 operands for each PLA. However, because of the wasted area involved in putting many PLA's togcthcr, WC prefer somewhat more than the optimal number of operands per PLA to reduce the overbcad due to PLA's that don't quite lit together.

A previous incarnation of the compiler attempted to translate the expression directly to a layout using an algorithm of Floyd and Ullman [1982] that requires area that grows only proportionally to the number of operands. However, the network-of-I'LA's implementation was found superior in practice. The reader should also be aware of another approach to laying out regular expression recognizers in linear area due to Kung and Foster [1982], which we have not tried.

Another style of implementation is described in Ullman [1982], where regular expressions were translated into the lgen logic language (Johnson [1983]) and thus implemented as Weinbcrger arrays. The area of such implementations was found to be comparable to the PLA implementation. Theoretically we might expect the Weinberger array approach to use less area than PLA's, but to form circuits of very large aspect ratio as the size of the expressions compiled grows.

The reader is also referred to Trickey [1981] for a description of some experiments with the systematic exploration of the different ways a regular expression could bc partitioned into subexpressions, and the subexpressions converted to PLA's that would fit together with little wasted area: It was found that significantly improved layouts could be obtained, but the computation time grew exponentially with expression size. That makes it doubtful the method could be applied to expressions with more than a few hundred operands, unless some way of focussing the search for partitions'were found.

## IV. Before and After NFA Constructions

Now, let us return to the two methods whereby NFA's arc constructed from regular expressions. We begin . either process by identifying each operand with a state.

**Example 4: Consider the bounce filter** of Example 2. We may number the operands of the expression from left to right as follows.

$$\cdot_1{}^{*}\ one_2\ one_3\ (one_4\ zero_5?\ zero_6?)^{++}\ OUT_7$$

WC may then associate with operand $i$ the state $N_i$.

In Fig. 4 we see what looks like a transition diagram for a finite automaton. It actually represents the successor relationship among the states or operands, i.e., which operands can follow which in the regular expression. For example, there is an arc from $N_5$ to $N_4$ because after seeing a 0 corresponding to $zero_5$, we could begin another group consisting of a 1 and up to two O's, and such a group must begin with a 1 that matches $one_4$. WC also have an arc from $N_5$ to $N_6$, because after matching $zero_5$ WC could see another 0

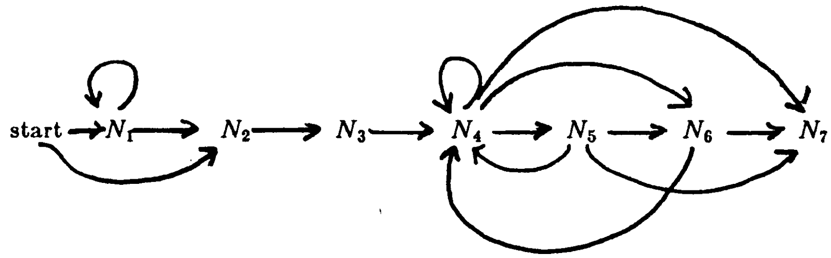


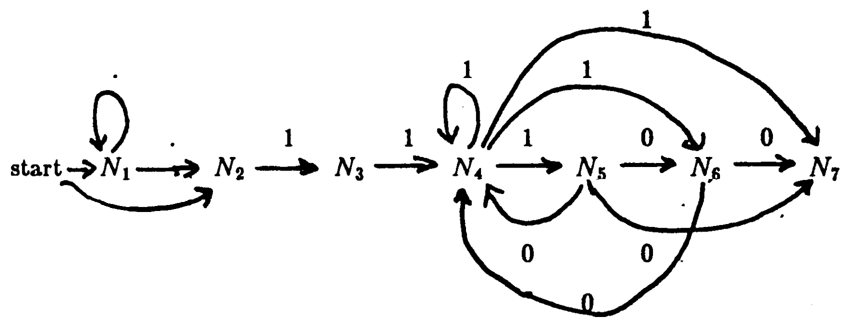

Fig. 4. Successor relation for bounce filter.

that matched $zero_6$. Finally, there is an arc from $N_5$ to $N_7$ because after seeing a match for $zero_5$ we have seen an input that matches the subexpression prior to the OUT output and therefore must make the OUT signal. See Ullman [1983] for the details on the algorithm used to compute the successor function. □

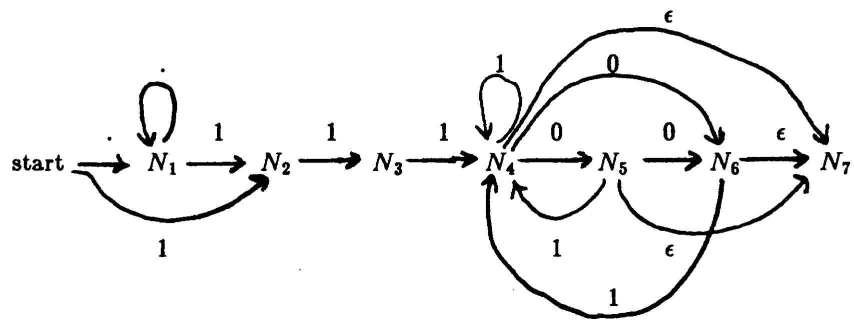
Whether we use the before or the after interpretation of states, we can see transition diagrams like Fig. 4 as representing places that can be "active," which we might represent by putting a marker on a subset of the nodes. When we use the before interpretation, a marker at state $N_i$ tells us we are ready to recognize the operand corresponding to that state. Thus, if state $N_5$ of Fig. 4 is active at a given time unit, it will activate for the next time unit the states $N_4$, $N_6$, and $N_7$, provided an input 0 is seen. If the input is not 0, those states will not be activated by $N_5$. $N_5$ will not be active at the next time unit, unless it is activated by a transition from $N_4$, its only predecessor.

Figure 5(a) shows the before interpretation of the NFA for the bounce filter. Each transition in Fig. 4 is made on the input that corresponds to the state at the tail of the transition. Those states that correspond to the operands that could match the first input seen, namely $N_1$ and $N_2$, are designated initial. A transition into $N_7$ causes the associated output, OUT, to be raised.

In the after interpretation of NFA's, each state represents a situation where we have already seen the corresponding operand of the regular expression and are ready to recognize the symbol corresponding to any of its successor states. Figure 5(b) shows the after interpretation of the bounce filter. In general, each transition is labeled by the symbol corresponding to the state entered by the transition, while in the before interpretation the same transition is executed when the input matches the operand of the state that the transition leaves. In the after interpretation, the start is a state itself, with transitions to its successors on the appropriate inputs, as shown in Fig. 5(b). Finally, states associated with output symbols are no longer states in any useful sense. Rather, transitions into such states, shown in Fig. 5(b) as associated with $\epsilon$, mean that the states from which such a transition is made are to raise that output signal as soon as they themselves are entered.




(a) Before method interpretation




(b) After method interpretation

Fig. 5. Interpretations of bounce filter **NFA.**

Comparison **of Methods**

Neither method is uniformly superior to the other. The advantage of the before method is that when we convert the NFA to a PLA, we need only one term per state (plus extra terms corresponding to transitions from **thc** initial states when the start signal is raised). To see why, we have **tc** understand that each **NFA** state is coded by turning on a subset of the feedback wires of the PLA; we shall discuss the method of selecting the representation of **each** state shortly. In **the** before interpretation, we need for each state N a term that checks

1. The code bits representing N were turned on at the previous time unit, and
2. The input corresponding to N is seen.

This term must turn on all **thc** wires in the or-plane that are **ncedcd** to rcprcscnt any of the successors of state N. It may be unclear at the moment how one represents **NFA** states (which may be on simultaneously in various combinations), unambiguously **by** turning on sets of bits; we shall cover the method **in the ncxt** section.

**Examplc 5:** Figure G(a) shows the PLA constructed from Fig. 5(a) if we code states $N_1$ through $N_7$ with a **singlc wirc cach.** (This turns out to bc as good as we can do for the **bouncc filter** NFA.) The **lcft** and right

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 2 | 2 | 1 | 2 | 2 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 2 | 1 | 2 | 2 | 2 | 1 | 2 | 2 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 2 | 2 | 2 | 2 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 2 | 0 | 2 | 2 | 2 | 2 | ·2 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 2 | 2 | 2 | ' 2 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| . | $X$ | $N_1$ | $N_2$ | $N_3$ | $N_4$ | $N_5$ | $N_6$ . | $N_1$ | $N_2$ | $N_3$ | $N_4$ | $N_5$ | $N_6$ | $N_7$ |

(a) Before method PLA.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 2 | 2 | 1 | 2 | 2 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 2 | 2 | 2 | 1 | 2 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 2 | 0 | 2 | 2 | 2 | 1 | 2 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 | 2 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 2 | 0 | 2 | 2 | 2 | 2 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 | 2 | 2 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| S | $X$ | $N_1$ | $N_2$ | $N_3$ | $N_4$ | $N_5$ | $N_6$ | $N_1$ | $N_2$ | $N_3$ | $N_4$ | $N_5$ | $N_6$ | $N_7$ |

(b) After method PLA

Fig. 6. Before and after PLA's for bounce filter.

groups are the and- and or-planes. A 0 or 1 in the and-plane means that the wire represented by the column must be off or on, respectively, for the term corresponding to the row to be seen. A 2 in the and-plane means "don't care." In the or-plane, 1's represent taps, so each column is the logical "or" of the terms with l's in that column. Note that $N_7$, the output, need not be fed back.

The first six rows are the terms for states $N_1$ through $N_6$. For example, row one says that if we are in $N_1$, and the input "dot" is seen (i.e., the input may be 0 or 1, represented by the 2, or "don't care, in the input column, X), we turn on $N_1$ and $N_2$ for the next time unit. Row two says that if state $N_2$ is on, and the input is 1, turn on $N_3$ for the next time unit.

The last two rows duplicate rows one and two, but with the start signal S replacing the wires for $N_1$ and $N_2$, respectively, in the term's conditions. Thus, these last two wires express the fact that $N_1$ and $N_2$ are on initially. Cl

If we use the after interpretation of NFA's then we must create for each state N, and each symbol $a$ labeling a transition out of that state, a term to check that the stntc is on and that the input is seen; if so, the successors of N on input a arc turned on in the or-plane for the next time unit. If successor $M$ has an c-transition to an output signal, then those terms that turn on $M$ in the or-plane also turn on that output wire.

**Example** 6: Figure 6(b) shows the after method PLA for the bounce filter. For example, rows three and four represent the transitions from $N_1$ on "dot" and 1 to states $N_1$ and $N_2$, respectively. Note that since $N_4$, $N_5$, and $N_6$ have c-transitions to $N_7$ in Fig. 5(b), the last five rows of Fig. G(b), which represent transitions into those states, also turn on the output wire, which is $N_7$. $\square$

If we compare Examples 5 and 6 we might get the impression that the before method is superior to the after; each uses the same number of columns, and the after method uses more rows. While it is typical that the before method saves rows, it is often true that the after method saves columns because it allows better NFA state codes. It just happens that for the bounce filter, no better state code is possible with the after method.

## V. Selecting NFA State Codes

We shall now take up the matter of how the compiler selects codes for states of an NFA. We first discuss the notion of conflicting states, that is, pairs of states that can be on at the same time. We show how the conflict information determines the permissible state codes and we discuss a particular method for finding legal codes.

### Conflicting Symbols and States

Before discussing conflicting states, we need to define conflicting input symbols. Symbols $a$ and $b$ conflict if both can be on at the same time, i.e., there is no wire $x$ that is on in the definition of a and off in the definition of $b$, or vice versa.

If we are using the before interpretation of an NFA, then we USC the following two rules to compute pairs of states that conflict. Rule (1) initializes the set of conflicts; we then add conflicting pairs by rule (2) until no more can be added.

1. Each state conflicts with itself. All initial states conflict with one another.
2. Suppose N and A4 are states that conflict, and they are associated with conflicting symbols a and $b$. (Note N = A4 is allowed.) Then for each successor $P$ of N and each successor $Q$ of $M$, $P$ and $Q$ conflict.

   There are similar rules that can be applied if we use the after interpretation of NFA's; they are:

1. Each state conflicts with itself. If N and $M$ are initial states that are 'associated with conflicting symbols, then N and $M$ conflict.

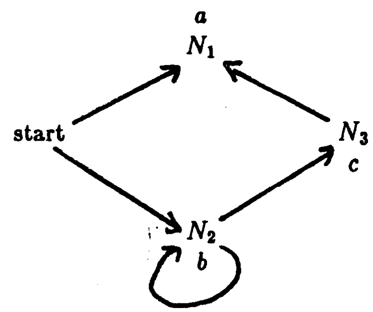



**Fig. 7.** Example NFA.

2. Suppose N and M conflict, $P$ is a successor of N and Q is a successor of M. Also suppose that $P$ is associated with symbol $a$ and Q with $b$, and a and $b$ conflict. Then $P$ and **Q conflict.**

Note that the set of conflicts under the after interpretation is always a subset of those found under the before interpretation. It is this effect that explains why we often get better state codes with the after method.

**Example** 7: Suppose we declare symbols by

line x, y
symbol a(-x, -y), b(x), c(y)

Then $b$ and c conflict. However, $a$ and b do not conflict, because of wire $x$, and a and c do not conflict because of wire $y$.

Consider the NFA shown in Fig. 7. In the before interpretation, $(N_1, N_2)$ is a conflicting pair because both are initial. Next, by rule (2) we find that $(N_2, N_3)$is conflicting because they are **both successors of** the conflicting "pair" $(N_2, N_2)$. Then, we find $(N_1, N_3)$ is a conflicting pair because they are, respectively, successors of the states $N_3$ and $N_2$, which are conflicting and associated with conflicting symbols.

In the after interpretation, rule (1) yields no nontrivial conflicting p-airs, because the start states $N_1$ **and** $N_2$ are associated with nonconflicting symbols. However, the successors $N_2$ and $N_3$ of the trivial conflict between $N_2$ and itself are associated with conflicting symbols, so $(N_2, N_3)$is a nontrivial conflicting pair. There are no other nontrivial conflicts in the after interpretation. □

### Legal Codes for NFA States

It is useful to think of the conflict information as a conflict graph, with states for nodes and edges between pairs of states that conflict. The compiler makes the simplifying assumption that any clique† in the conflict graph represents a set of states that can all bc on at the same time. Surely if some set of states can all be

† A *clique* is a set of nodes with an edge between any two nodes in the set. The clique is *maximal* if no node outside the clique has an edge to each member of the clique.

on at once, then each pair of states in the set **conflicts,** but the converse is not true; there could be three different input sequences that lead, respectively, to states $M$ and N, to M and $P$, and to N and $P$, yet no one input sequence turns on M, N, and $P$ together.

Our decision to consider only conflicts between pairs, rather than all subsets, was so that the amount of information handled by the compiler would grow only quadratically with the **regular** expression size, not exponentially as it would if we considered conflicts among arbitrary sets of states. The assumption that all cliques represent conflicting sets is conservative, in the sense that it may prevent us from taking advantage of some good codes for states but will not lead us into an error where we design a malfunctioning PLA.

**When** choosing codes for states, we make the following hypothesis, which is oriented toward the **PLA implementation** of **NFA's.** We suppose that associated with each state is a vector of $k$ 0's, 1's, and 2's, with 2 standing for "don't care." Let $C(N, i)$ be the $i^{th}$ position in the vector for state N. If state N is to be on, then we turn on the $i^{th}$ feedback bit whenever $C(N, i) = 1$. If $C(N, i)$ is 0 or 2, we do not turn on the $i^{th}$ bit because of N, although it could be turned on because of some other state.

In the and-plane, when we must recognize that we are in state N, perhaps among others, we examine the feedback bits. If $C(N, i) = 1$, we check that the $i^{th}$ feedback bit is **1;** if $C(N, i) = 0$, we check that it is 0, and if $C(N, i) = 2$, we do not check the $i^{th}$ feedback bit. We must consider under what conditions the code $C$ allows us to interpret all possible combinations of feedback bits correctly. There are two conditions that together ensure that we shall make the proper inferences from the feedback bits.

1. ***When*** state N ***is*** on, we ***detect N.*** If $C(N, i) = 2$, we do not check bit $i$, so there are no constraints on $i$ as far as N is concerned. If $C(N, i) = 1$, and N is on, we know bit $i$ will be turned on, so the test for N will be met at bit $i$. Finally, if $C(N, i) = 0$, then we must be assured that no other state $M$ that conflicts with N, and could therefore be on at the same time as N, has $C(M, i) = 1$. For if there were such an M, then we could find bit $i$ equal to 1, and fail to detect N even though it is on.

2. ***If*** N **is** detected, ***then*** N ***is on.*** Here, we must check that there is no (not necessarily maximal) clique $\{M_1, \ldots, M_r\}$ that does not contain N but can forge the code for N. That is, for no such clique is it the case that for all i, $1 \leq i \leq k$:

   a) If $C(N, i) = 0$, then for all j, $C(M_j, i) \neq 1$.

   b) If $C(N, i) = 1$, then there is some j for which $C(M_j, i) = 1$.

   If no clique satisfies (a) and (b), then the code $C$ satisfies condition (2).

**Example** 8: Figure 8 shows a conflict graph. A possible 3-bit code for this set of states is:

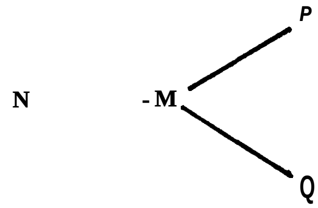



**Fig.** 8. Example conflict graph.

| | 1 | 2 | 3 |
|---|---|---|---|
| $M$ | 2 | 2 | 1 |
| N | 1 | 0 | 2 |
| $P$ | 0 | 1 | 2 |
| $Q$ | 1 | 1 | 2 |

To check condition (1) we have only to examine **the** 0's. For example, $C(N, 2) = 0$, but there is **no** other state conflicting with N that has 1 in its second bit. That is, only $M$ conflicts with N, but $C(M, 2) = 2$.

We must also check condition (2). For example, it looks like N and $P$ together could forge Q, but $\{$ N, $P$ $\}$ is not a clique, because N and $P$ do not conflict. □

### Simple Coding Methods

The first coding method implemented, which we call **the** greedy method, is to look for maximal independent sets in the conflict graph. An independent set is a set of nodes no two of which are connected by an edge. We may partition the nodes into maximal independent sets by starting with any node and adding nodes that do not conflict with any of the nodes previously added, until no more can be added. The result is one maximal independent set. We then remove the nodes of this set from the graph and start with another node to grow another independent set, and so on. This method has been used for similar purposes in several other works, such as Haskin [1980] and Nagle, Cloutier, and Parker [1982].

Having obtained a partition into independent sets, we may binary code the states in each set, omitting the all-zero code, so a set of $m$ states can be coded with $\lceil \log_2(m + 1) \rceil$ bits. Each of the independent sets uses bits of its own in the state code, and the code for each state has don't care's in the bits belonging to the independent sets other than its own. This coding method works because the only possible combinations of states have at most one from each independent set. The bits for each set tell us which, if any, state from that set is on. By not using the all-zero code for any state, we can detect the case where no member of an independent set is on.

**Example 9:** Consider the conflict graph of Fig. 9. We might start growing an independent set with state

N. We may add $P$ and Q, because neither conflicts with N or with each other. However, we cannot add $M$ because it conflicts with N. Thus, we start a second independent set with $M$, and the partition of Fig. 8 is $\{\{N,P,Q\},\{M\}\}$.

The first of these sets requires two bits and the second one bit. The resulting state code is that given in Example 8. □

### The Clique Compatability Class Method

A second coding method tried is described in Ullman [1982]. It gave better codes than the greedy method in some cases. We shall omit a description of that method and instead describe the most recent and most successful coding method. This approach partitions the states into maximal clique compatibility classes (MCCC's). An MCCC consists of a collection of cliques, such that no node of one clique is connected to a node of another clique by an edge of the conflict graph.

We grow MCCC's by starting with maximal independent sets and growing them by adjoining nodes whenever possible. We can adjoin node N to clique Q if N is adjacent to every node in Q but N is adjacent to no node in any of the other cliques in the MCCC. After partitioning the conflict graph into MCCC's, we code each MCCC in a manner to bc described. We then find the overall state code by using a separate set of bits for each MCCC, just as we did for independent sets in the greedy algorithm. Each state has its code in the bits of its own MCCC and don't care's elsewhere.

### Coding MCCC's

The basic idea is that we try to use the same code bits for as many cliques in the MCCC as we can. We start off coding each clique individually, and then try to combine the codes for different cliques. As the states of one clique can be on or off in any combination, there is nothing better than to use a one-hot code, i.e., use as many bits as there are states in the clique, with each state given a code consisting of a 1 in a unique position and 2's (don't care) elsewhere.

Then, we combine cliques, in pairs, until we have combined all pairs. The priorities for which pair to combine are as follows.

1. If there are two cliques or sets of cliques that have codes with the same number of bits, combine them. However, that number of bits must be at least two, i.e., we do not apply step (1) to a pair of cliques consisting of one state each. If there are two or more pairs that may be combined, pick a pair that have the shortest codes.
2. If no sets of cliques may be combined by rule (1), find the two cliques or sets of cliques with the shortest

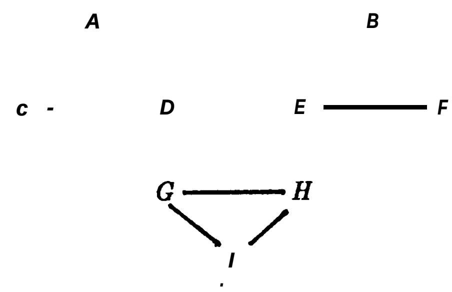



**Fig. 9. An** MCCC.

codes (including codes of one bit), and combine them. However, do not choose a singleton clique unless the number of singleton cliques is at least as great as the number of other sets of cliques remaining.

The combination of two sets of cliques may have the side effect of combining with them a singleton clique, as we shall see. That is the reason we do not wish to pick singleton cliques for combination unless there are so many that they cannot all be consumed in the combination of other cliques. In general, the rules for combining the codes for two sets of cliques are the following.

1. If one code is shorter than the other, pad the shorter out with leading O's until they are of equal length.
2. Place a 0 in front of all the codes in one set and a 1 in front of all the codes of the other set.
3. If there are remaining cliques consisting of a single state, choose one and give it the code 10· · ·0. It is easy to prove by induction on the number of cliques combined that we never produce an all-zero's code. Thus, 10· · ·0 will not be the same as the code of any other state in the set.

**Example 10:** Consider the MCCC shown in Fig. 9, where there are two singleton cliques, two doubletons, and one clique of size three. The initial codes for the cliques are the following.

| | | | |
|---|---|---|---|
| *A* | 1 | *E* | 21 |
| | | *F* | 12 |
| *B* | 1’ | | |
| | | *G* | 221 |
| c | 21 | II | 212 |
| *D* | 12 | I | 122 |

Note that the bits for different cliques do not yet bear any relation to each other, so there is nothing wrong with assigning the code **1** to both *A* and *B*, for example.

According to rule (1), our first task is to combine cliques *CD* and *EF*, since they are not singletons, but have the same code length. Let us say we put 0 in front of the codes for $C$ and *D* and 1 in front of the codes for *E* and $F$. We then add a singleton, say $B$, giving it the code 100. The result is a set of five states with the following code:

| | |
|---|---|
| *B* | 100 |
| C | 021 |
| *D* | 012 |
| *E* | 121 |
| *F* | 112 |

Now, the set *BCDEF* and the clique *GHI* have the same length code, so we combine them, consuming the singleton *A* in the process. The resulting code is:

| | | | |
|---|---|---|---|
| *A* | 1000 | *F* | 0112 |
| *B* | 0100 | *G* | 1221 |
| C | 0021 | *H* | 1212 |
| *D* | 0012 | *I* | 1122 |
| *E* | 0121 | | |

## VI. Evaluation of the Compiler

We believe that the principal reason to express designs in the regular expression language is its ability to accept descriptions of the patterns it must recognize and the responses it must make, in a flexible manner. For example, additional patterns may be added to the description of a controller, and the compiler will produce the necessary modifications without the user having to worry about the possibility of interactions between the new patterns and the old ones. Design systems based on deterministic automata do not have this robustness.

However, it is also important that the design produced by the compiler be of good quality. We have run several test cases, and these indicate that the compiler performs well, in some cases better than obvious hand designs of PLA's. We shall mention some of these trials here.

### The Bounce Filter

Here we do not do well; a PLA with about half the area of that of Fig. 6(a) can be designed.

### The Traffic Light Controller

Using either the before or after method, the compiler comes up with essentially the same PLA as appears in Mead and Conway [1980]. The only difference is that the compiler introduces an initialization signal, which is not really needed for the perpetually running traffic light.

### The Pattern Matcher

We alluded above to a regular expression with 72 operands and an 8,000,000 state deterministic automaton.

```
line input[2]
symbol
     zero(-input[1])
     symbol one(-input[2])
output MISMATCH
,
.*(
(. . .  (
. . (. (zero .* one + one .* zero) +
     (zero .* one + one .* zero) .)
+
(. (zero .* one + one .* zero) +
     (zero .* one + one .* zero) .) . .
)
+
(. . (. (zero .* one + one .* zero) +
     (zero .* one + one .* zero) .)
+
(. (zero .* one + one .* zero) +
     (zero .* one + one .* zero) .) . .
) . . . .
)
) MISMATCH
```

**Fig. 10.** Pattern matching regular expression.

This expression is shown in Fig. 10. The problem is to signal mismatches between the first eight symbols read and the last eight read. Input 1 is represented by turning input [1] on and *input*[2] off; input 0 is represented by the opposite, and don't care, which matches anything, is represented by turning both input wires on. The expression appears complicated, but the idea is that a mismatch between the first and last eight symbols can be expressed recursively as either a mismatch between the first four and the next-to-last four; or a mismatch between the second four and the last four; these mismatches can be expressed as mismatches of two pairs, and so on.

An obvious hand implementation of a PLA was attempted, using the straightforward idea that each of the symbols to be remembered, the first eight and the last seven, would be coded by two bits, for 0, 1, and don't care. This approach requires 16 feedback wires for the first eight inputs, 14 more to remember the most recent seven inputs, and four feedback wires to represent a counter that counts up to eight, to tell the PLA whether to remember its current input as one of the first eight symbols. As for terms of the PLA, we need 16 to feed back the first eight inputs, 14 more to feed back and shift the seven most recent inputs, 16 to load the first eight symbols originally, 32 to detect mismatches, and eight to implement the counter. Thus, the hand design uses 34 feedback wires and 86 terms.

In comparison, using the before method and the greedy state coder, we require 28 feedback wires and 62

```
line x[3]
symbol
    in0(x[1] -x[2])
    in1(x[2] -x[1])
    badin(x[1] x[2])
    ack(x[3])
    noin(-x[1] -x[2])
    noack(-x[3])
output OUTA, OUTB, OUTC, ERROR
state statea, stateb, statec
subexp somein = in0 + in1 + badin
subexp waitin = noin + badin
subexp allbut01 = ack + badin
;
waitin* (
    allbut01 ERROR +
    in0 statea +
    in1 stateb
    )
+ # statea: noack* OUTA (
        somein ERROR +
        ack waitin* (
            allbut01 ERROR +
            in0 stateb +
            in1 statec
            )
        )
+ # stateb: noack* OUTB (
        somein ERROR +
        ack waitin* (
            allbut01 ERROR +
            in0 statec +
            in1 statea
            )
        )
+ # statec: noack* OUTC (
        somein ERROR +
        ack waitin* (
            allbut01 ERROR +
            in0 statea +
            in1 stateb
            )
        )
```

Fig. 11. Regular expression for transmitter.

terms. When we implemented the MCCC method of state coding, this number was reduced to 24 feedback wires, which is only one more than the theoretical minimum. This problem is an example where the before method yields significantly better results than the after method, as well as significantly better results than the obvious hand design.

| | And Cols. | Or Cols. | Terms | Area |
|---|---|---|---|---|
| Hand-Unoptimized | 13 | 7 | 29 | 580 |
| Hand-Optimized | 13 | 7 | 23 | 460 |
| Before Compiler | 17 | 11 | 23 | 644 |
| After Compiler | 13 | 7 | 25 | 500 |

**Fig. 12.** PLA's for communication protocol.

## A Communication Protocol

We designed the transmitter portion of the protocol for handling lost bits discussed in Aho, Ullman, and **Yannakakis [1979].** Briefly, that transmitter has two inputs, $x[1]$ and $x[2]$ telling it to send a 0 or 1 down the channel, while $x[3]$ is another input wire, used as an acknowledgement signal. The protocol works because the transmitter sends one of three signals, *a, b,* and c, which we may view as arranged in a circle. To send a 0, the transmitter steps one around the circle and to send **1,** it steps twice. States *statea, stateb,* and *statec* are the states in which the transmitter is trying to send *a, b,* and c, respectively. We assume that **signals are not mutated,** so any signal acknowledged must be the correct one. The regular expression program is shown in Fig. 11.

In Fig. 12 we see the results of hand and mechanical generation of PLA's for the transmitter. A straightforward hand design was optimized by GRY, to reduce the number of its terms. The results of the before and after methods are shown after optimization by GRY. We should be aware that when we count columns in the and-plane, we count one column if a signal is needed either true or complemented, but not both; if needed both ways we count two columns. This method of counting is realistic, provided the PLA generator used does not force all wires to become true and complemented pairs in the and-plane.

The "area" of the PLA in Fig. 12 is the product of rows and total columns, which is not precisely accurate, but serves to measure approximately the area actually used by the PLA.

The reader should note that the results shown in Fig. 12 for the compiler are the result of the greedy algorithm, not the MCCC algorithm. In both the before and after interpretations, the greedy method achieves the smallest possible number of feedback wires. The MCCC algorithm uses the same number of bits to code the states as the greedy algorithm does, i.e., the number of feedback wires is the same, but because of differences in the coding used, the number of terms after optimization was slightly larger when the MCCC coder was used.

## References


Aho, A. V., J. D. Ullman, and M. Yannakakis [1979]. "Modeling communications protocols by automata,,, *Proc. Twentieth* Annual ACM Symposium on *the* Theory of Computing, pp. 267-273.

Floyd, R. W. and J. D. Ullman [1982]. "The compilation of regular expressions into integrated circuits," J. ACM **29:2,** pp. 603-622.

Brayton, R. K., G. D. Hachtel, L. A. Hemachandra, A. R. Newton, and A. L. M. Sangiovanni-Vincentelli [1982]. "A comparison of logic minimization strategies using EXPRESSO: an APL program package for partitioned logic minimizalization," *Proc.* IEEE *Intl. Conf. on Circuits and* Computers.

Haskin, R. L. [1980]. "Hardware for searching very large text databases," Ph. D. Thesis, Dept. of Computer Science, Univ. of Illinois, Urbana, Ill.

Hemachandra, L. A. [1982]. "GRY: a PLA minimizer,,, unpublished memorandum, Dept. of Computer Science, Stanford Univ., Stanford, CA.

Hennessy, J. L. [1981]. "SLIM: a simulation and implementation language for VLSI microcode," LAMBDA, April, 1981, pp. 20–28.

Hopcroft, J. E. and J. D. Ullman [1979]. Introduction *to* automata *theory,* languages, and computation, Addison-Wesley, Reading Mass.

Johnson, S. C. [1983]. 'Code generation for silicon," *Proc.* Tenth ACM Symposium *on* Principles *of Pro*gramming Languages.

Foster, M. J. and H.-T. Kung [1981]. "Recognize regular languages with programmable building blocks," in VLSI-81 (J. P. Gray, ed.), Academic Press, New York, pp. 75-84.

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

Nagle, A. W., R. Cloutier, and A. C. Parker [1982]. "Synthesis of hardware for the control of digital systems,,, IEEE Trans. *on* Computer-Aided Design of Integrated *Circuits* and *Systems* CAD-1:4, pp. 201-212.

Trickey, H. W. [1981]. "Good layouts for pattern recognizers," IEEE Trans. on Computers C-31:6, pp. 514-520.

Trickey, H. W. [1982]. "Using NFA's for hardware design," unpublished memorandum, Stanford Univ., Dept. of C. S.

Ullman, J. D. [1982]. "Combining state machines with regular expressions for automatic synthesis of VLSI circuits," STAN-CS-82-927, Computer Science Dept., Stanford Univ., Stanford, CA.

Ullman, J. D. [1983]. Algorithmic Aspects of VLSI, Computer Science Press, Rockville, Md.