September 1982

Report No. STAN-CS-82-927

Corn bining State Machines and Regular Expressions for Automatic Synthesis of VLSI Circuits

by

Jeffrey D. Ullman

Department of Computer Science

Stanford University
Stanford, CA 94305

COMBINING STATE MACHINES AND REGULAR EXPRESSIONS FOR AUTOMATIC SYNTHESIS OF VLSI CIRCUITS†

Jeffrey D. Ullman

Stanford **Univ.**

ABSTRACT

We discuss a system for translating regular expressions into logic equations or PLA's, with particular attention to how we can obtain both the benefits of regular expressions and state machines as input languages. An extended example of the method is given, and the results of our approach is compared with hand design; in this cxamplc we use less than twice the area of a hand-designed, machine optimized PLA.

I. The Regular Expression Compiler

A collection of routines have been written by II. Trickey and J. Ullman to translate regular expressions into circuits. At present, we first compile regular expressions into a language that describes nondeterministic finite automata (NFA's). These NFA's arc then compiled into either PLA's or S. C. Johnson's *lgen* logic language.

A description of the regular expression language appears in [TU]. The language is quite standard, with perhaps the following exceptions.

1. Input symbols are not "disjoint," in the sense that at any time only one can be seen on the input. Rather, input symbols are defined in terms of some set of wires being on or off. Since not all wires must be specified for each symbol, there is the possibility that two or more symbols are on at a time. This **has** the consequence that apparently deterministic processes can in fact have nondeterminism in them.
2. Output signals are represented by ordinary-looking symbols in regular expressions. When the input is such that an output symbol is reached in the expression, we emit that signal, and proceed to recognize any continuation of the expression that the input allows us to recognize.

Example 1: In Fig. 1 we see an input to the regular expression compiler that forms a running example for this report. Without dealing now with the issue of what this program does, let us observe a few salient features. The first line says that there are seven input wires, $x[1], \ldots, x[7]$. Next come the definitions of the input symbols. For example, we see signal *in*0 on the input whenever the first wire is on and the second off. Note, for example, that *wc* could see symbol *in*0 and also *acka*, if the first three wires were 1, 0, and 1.

† Work supported by DARPA contract MDA 90% 80-C-0107.

Following this comethe declaration of output signals, and then three subcxpressions, *somein*, which is recognized when either *in*0 or *in*1 is seen, *waitin* which is recognized when neither input is seen, or both wires $x[1]$ and $x[2]$ are on simultaneously (which represents a "bad input," the symbol *badin*), and *allbut*01, which stands for the union of all signals but *in0* and inl. After the declaration portion is a semicolon and the expression itself.

As an example of how the expression is to be interpreted, consider the seventh line of the expression, beginning *stateia*·... It says that if we get a signal telling us wc are in state a, and then receive any number of symbol *noacka* (*noacka** means "any number of *noacka*'s"), we emit the signal $OUTA$. We regard $x[3]$ as a wire that "acknowledges" the fact that signal $OUTA$ was received, so symbol *noacka*, defined by $x[3]$ being 'off, is seen until the *acka* symbol, $x[3] = 1$, is seen. In effect, we emit the output signal $OUTA$ until it is acknowledged.

When $OUTA$ is acknowledged by *acka* appearing on the input, the process of recognizing the expression proceeds to *waitin**, which is recognized for as long as the first two wires remain at 0, or both become 1 simultaneously (a bad input). Then, when in0 or *in*1 is seen, a signal to change to state *b* or c is made. If any of the symbols represented by *allbut*01 is received after the *acka*, an error is declared. □

II. Coxnbining States and Expressions

The motivation for using regular expressions as a source language is twofold. First, they are a succinct and nonprocedural description of a large class of sequential processes. Thus they can provide some simplification in the design process for the right problem. Second, being structured descriptions of patterns, they are appropriate for proofs of correctness, and even if a formal proof is not attempted, they provide useful intuition that helps the reader convince himself of the correctness of the expression. In comparison, transition functions for automata are analogous to programs with goto's; they are inherently hard to understand and verify, either formally or intuitively.

On the other side of the coin, there are distinct advantages to process descriptions involving states. Often, it is natural to view a process as being in one of several states; for example, counting is especially easy when you have states available and very hard to do with regular expressions. It is the purpose of this report to describe a simple modification to the regular expression compiler that allows us, in effect, to declare states and then define transitions among states in regular expression terms. As a result, wc get the best of both worlds; states are available when they are more succinct than regular expressions or when they helpus organize our design, and regular expressions are available when patterns of symbols are useful as a description of events.

```
line      x[7]
symbol        in0(x[1] -x[2])
          in1(x[2] -x[1])
          badin(x[l] x[2])
          acka(x[3])
          ackb(x[4])
          ackc(x[5])
          stateia(x[6] x[7])
          statcib(x[6] -x[7])
          stateic(-x[6] x[7])
          start(-x[6] -x[7])
          noin(-x[1] -x[2])
          noacka(-x[3])
          noackb(-x[4])
          noackc(-x(5])
output    OUTA
          OUTB
          OUTC
          stateoa
          stateob
          stateoc
          ERROR
su bexp   somein=in0 + in1 + badin
subcxp    waitin=noin + badin
subexp    allbut01=acka + ackb + ackc + badin
;
start waitin* (
          allbut01 ERROR +
          in0 stateoa +
          in 1 stateob

+
stateia noacka* OUTA (
                  (ackb+ackc+somein) ERROR +
                  acka waitin* (
                            allbut01 ERROR +
                            in0 stateob +
                            in1 stateoc
                            )
                  )
+
stateib noackb* OUTB (
                  (acka+ackc+somcin) ERROR +
                  ackb waitin* (
                            allbut01 ERROR +
                            in0 stateoc +
                            in 1 stateoa


+
stateic noackc* OUTC (
                  (acka+ackb+somcin) ERROR +
                  ackc waitin* (
                            allbut01 ERROR +
                            in0 stateoa +
                            in1 statcob
                            )
                  )
```

Fig. 1. Input to regular expression compiler.

To introduce states into the regular expression language, we make the following modifications.

1. The names *statei*X for any X are input symbols that represent the fact that we have just entered state X. Symbol *start* serves as the initial state. To indicate that these states are disjoint, i.e., we can be in only one of them at a time, we can use imaginary wires, such as $x[6]$ and $x[7]$ in Fig. 1, to make it appear to the compiler that at most one of these input symbols can be present on the input at any time. Of course, if the states were not disjoint in this sense, we could express the legal subsets by another combination of dummy wires.
2. Output symbols *stateo*X for any X are used as goto's. If we emit the symbol *stateo*X, we shall in effect turn on the input symbol *statei*X and enter state X.

The complete regular expression consists of the sum of expressions that begin *start* and *statei*X for the various X's. The portion of the regular expression associated with each state is recognized, if possible, each time we enter that state, and we make whatever outputs the regular expression tells us to make in response to what inputs we see.

After the regular expression compiler converts the expression into a nondeterministic finite automaton, an edit script is used to identify the input symbol *statei*X with the output symbol *stateo*X and make certain other changes so things work properly.

Example 2: A case in point is the problem to which the regular expression program of Fig. 1 is a solution. This program implements the transmitter from [AUY] that sends bits reliably over a channel that has a high probability of losing bits, but does not change O's into l's or vice-versa. This view of a channel is plausible if we assume that any noise or other error causes the system to fail to detect a bit. This strategy, of assuming no signal whenever something goes wrong, is modeled after the Datakit protocol [F].

The general idea is that when the transmitter is given a bit to send, it sends one of three signals *OUTA*, *OUTB*, or *OUTC*, chosen by a method to be described. It keeps sending the signal until it receives an acknowledgement of the signal sent. Then, it stops sending the signal until the next input, 0 or 1, is received, whereupon it sends the next signal (in the sense that c follows *b*, which follows a, which follows c) if 0 is input, and it sends the previous signal in this cyclic order if 1 is input.

The purpose of this arrangement is so that whenever the transmitter sends a new input, it changes the signal sent; that change serves to acknowledge the acknowledgement. If we did not always make a signal change, the receiver could not tell, say upon receiving two O's, whether these were two different inputs, or the acknowledgement of the first had been lost, and the second 0 was a retransmission of the first.

Another way to look at the signal selection algorithm, is that we count one for an input 0 and two for an input 1, and transmit *OUTA*, *OUTB*, or *OUTC* depending on whether the sum of inputs received so

far is congruent to 1, 2, or 0, modulo 3. Counting, even counting modulo 3, is very difficult to express in the regular expression language. Thus it is natural to introduce three states, a, *b*, and c, that are entered whenever we receive an input that makes this running modular sum 1, 2, or 0, respectively.

We already discussed briefly in Example 1 what happens in one of these states, say a. After receiving the *stateia* signal to say we have entered state a, we emit *OUTA* for as long as the input matches *noacka**, that is, the *acka* acknowledgement is not received. A sequence of *noacka's* can be followed by either of two events that cause special outputs. First, the *acka* signal can be received, and then, after any sequence of *waitin's*, i.e., no input, an in0 or *in*1 triggers a signal that causes a jump to another state, *b* or c, respectively. After receiving *acka*, any input but in0 or *in*1 causes an error signal. Note that *allbut*01 and *waitin* can be seen at the same time, so we can continue waiting for a good input even while signaling when errors occur.

Now let us return to the point in the cxprcssion where we are recognizing *noacka** and waiting for *acka*. At the same time we are waiting for *acka*, if *we* receive *ackb*, *ackc*, or *sornein*, we have an error condition; in the first two cases, the wrong acknowledgement was received, in the last, we received either a bad input, or a good input before we are ready to transmit it. In this case, we emit the output signal ERROR. Note that all of these error conditions are seen on the input at the same time *noacka* is seen, so emitting *ERROR* does not prevent us from continuing to *see noacka** and eventually to see *acka* and another input. However, inputs received erroneously do not cause a change of state, because we cannot reach a term like *in0 stateob* in the regular expression until after the *acka* has been received.

The portions of the expression following *stateib* and *stateic* are analogous to what we have described. The portion following *start* differs only in that we are not waiting for an acknowledgement, and if any is received it is an error.

The result of compiling Fig. 1 is shown in Fig. 2. This figure illustrates the NFA language used. Each type of statement begins with a unique letter. For example, D is a declaration of an input symbol, much like in the regular expression compiler. However, note that the input symbol *stateiX* and the output symbol *stateoX* have both become *stateX*, and this input symbol is dcclarcd (in lines 6-8, e.g.) to be present when the wire of the same name is on; that wire is the corresponding output signal.

The letter N indicates the name of the NFA, and F indicates the name of the final signal, if any (there is none in Fig. 2), and the states that cause the final signal to bc ernittcd. Letter I introduces the name of the initial signal, *init* in this case, and a list of the initial states, *st*2, *st*3, and so on.

A state is dcclarcd by the letter S, followed by the state name, and the input symbol that it recognizes. The NFA langauge is restricted in that each state recognizes only one input symbol. However, this restriction is not bothersome for NFA's that arc output by the regular cxprcssion compiler, and in general, wc can create

```
D in0 ( in0 -in1)
D in1 ( -in0 inl)
D badin ( in0 inl)
D acka ( acka)
D ackb ( ackb)
D ackc ( ackc)
D statca (statca)
D stateb (stateb)
D statec (statec) .
D noin ( -in0 -inl)
D noacka ( -acka)
D noackb ( -ackb)
D noackc (-ackc)
N nfal
F ;
I init; st2 st3 st4 st5 st6 st9 st11
S st2 noin
T st2 st3 st4 st5 st6 st9 stll
S st3 badin
T st2 st3 st4 st5 st6 st8 st9 stll
S st4 acka
T st8
S st5 ackb
T st8
S st6 ackc
T st8
S st7 0 stateb
S st8 0 error
S st9 in0
T st10
S st10 0 state8
S stll in1
T st7
S st12 0 statec
S st13 statca X
T st5 st6 st14 st15 st17 st18 st19 st22
S st14 noacka
T st5 st6 st14 st15 st17 st18 st19 st22
S st15 0 outa
S st16 badin
T st4 st5 st6 st8 st9 st11 st16 st29
S st17 badin.
T st8
S st18 in0
T st8
S st19 in1
T st8
S st20 badin
T st4 st5 st6 st17 st20 st27 st31 st33
S st21 statec X
T st4 st5 st17 sL18 st19 st25 st26 st37
S st22 acka
T st4 st5 st6 st17 sL23 st24 st30 st32
S st23 noin
T st4 st5 st6 st17 st23 st24 st30 st32
S st24 badin
T st4 st5 st6 st17 st23 st24 st30 st32
```

Fig. 2(a). Beginning of NFA description.

```
S st25 ackc
T st4 st5 st6 st9 stll st16 st29
S st26 noackc
T st4 st5 st17 st18 st19 st25 st26 st37
S st27 noin
T st4 st5 st6 st17 st20 st27 st31 st33
S st28 ackb
T st4 st5 st6 st17 st20 st27 st31 st33
S st29 noin
T st4 st5 st6 st9 stll st16 st29
S st30 in0
T st7
S st31 in0
T st12
S st32 in1
T st12
S st33 in1
T st10
S st34 stateb X
T st4 st6 st17 st18 st19 st28 st35 st36
S st35 noackb
T st4 st6 st17 st18 st19 st28 st35 st36
S st36 0 outb
s st37 0 outc
C st2; st3 st4 st5 st6 st8 st9 st11
C st3; st4 st5 st6 st8 st9 stll
C st4; st5 st6 st8 st9 stll st16 st17 st18 st19 st20 st23 st24 st25 st26 st27 st28 st29 st30 st31 st32 st33
      st35 st36 st37
C st5; st6 st8 st9 stll st14 st15 st16 st17 st18 st19 st20 st22 st23 st24 st25 st26 st27 st29 st30 st31
      st32 st33 st37
C st6; st8 st9 stll st14 st15 st16 st17 st18 st19 st20 st22 st23 st24 st27 st28 st29 st30 st31 st32 st33 st35 st36
C st7; st8
C st8; st9 st10 st11 st12 st14 st15 st16 st17 st18 st19 st20 st22 st23 st24 st25 st26 st27 st28 st29
      st30 st31 st32 st33 st35 st36 st37
C st9; stll st16 st29
C stll; st16 st29
C st13; st21 st34
C st14; st15 st17 st18 st19 st22
C st15; st17 st18 st19 st22
C st16; st29
C st17; st18 st19 st20 st22 st23 st24 st25 st26 st27 st28 st30 st31 st32 st33 st35 st36 st37
C st18; st19 st.22 st25 st26 st28 st35 at36 st37
C st19; st22 st25 st26 st28 st35 st36 st37
c st20; st27 st31 st33
c st21; st34
c st23; st24 st30 st32
c st24; st30 st32
C st25; st26 st37
C st26; st37
c st27; st31 st33
C st28; st35 st36
c st30; st32
c st31; st33
C st35; st36
E
```

Fig. 2(b). End of NFA description.

several states with the same predecessors to simulate one state with transitions on several inputs.†

An 0 preceding the symbol associated with a state means that the symbol is an output symbol, rather than an input symbol. The letter X following the symbol, as in *st*13, means that the state is *external*; it is always on and waiting to see its input symbol.‡ It is exactly the states of the NFA that represent the states used in the regular expression specification that become external states of the NFA.

All states are followed by the letter T and a list of their transitions, that is, their successors. Finally there are conflict statements introduced by the letter C. The state following the C is declared to conflict with all the states after the semicolon. Conflicting states are those that can be on at the same time, a result not only of the nondeterminism but of the fact that several input symbols may be recognizable at once. Conflicts among states are taken into account when we find a coding of the NFA's states for an implementation. □

III. Logic Generation

The NFA is converted to the logic language lgen by an algorithm described in [U]. Briefly, the nondeterministic states must receive representations that will enable us to identify that each state is on, regardless of what other states are also on at the same time. Here is where knowing the state conflicts may help, because when state *i* is on, that fact can only be obscured by states that conflict with *i* also being on. For example, if the NFA were really deterministic, there would be no conflicts, and we could use a binary coding of the states.

One way to code states is to give each a private signal. Then we can tell the state is on independent of any other states. The actual approach taken by the logic generator used is slightly more sophisticated. It attempts to identify groups, which are sets of mutually nonconflicting states.†† Within a group, binary codes are selected so that any conflicting states from other groups will receive the same code. To do so, a minimal number of states that would make this coding impossible are expelled from groups and given private signals. Groups of a single state are similarly given private signals.

The result is that in addition to private signals, there are code bits and group *bits*. A state without a private signal is recognized by the bit of its group being on, and the code bits being on or off as appropriate

†[T] describes a more general NFA language that allows, multiple transitions, c-transitions, and a variety of options not available in the NFA language described here.

‡There is another kind of state like external states, that does not appear in Fig. 2. These states, called advance states and designated by A, are like external states, but when they see their input, they enable their successors to recognize their own inputs at the same time unit. Advance states are needed to implement correctly networks of NFA's that together recognize one large regular expression. Large expressions need to be broken into pieces implemented by separate NFA's for two reasons. First, processing large expressions is too time consuming, especially minimizing the states of the NFA and computing conflicts. Second, the circuits implementing the NFA's such as PLA's or Weinberger arrays, will be too large and badly shaped if the NFA has too many states.

††However, before looking at conflicts, states that have exactly the same predecessors (and therefore are really just different transitions from the "same" state) are combined into one.

ost2 = ¬in1 * ¬in0 * est2
ost3 = in1 * in0 * est.2
ost4 = acka * est4
ost5 = ackb * est5
ost6 = ackc * est6
ost9 = ¬in1 * in0 * est9
ost11 = in1 * ¬in0 * est9
ost13 = statea
ost14 = ¬acka * ce1 * ¬ce2 * ¬ce3
ost17 = in1 * in0 * est17
ost18 = ¬in1 * in 0 * est18
ost19 = in1 * ¬in0 * est18
ost22 = acka * ce1 * ¬ce2 * ¬ce3
ost16 = in1 * in0 * ¬ce1 * ce2 * ¬ce3
ost29 = ¬in1 * ¬in0 * ¬ce1 * ce2 * ¬ce3
ost20 = in1 * in0 * ce1 * ce2 * ¬ce3
ost27 = ¬in1 * ¬in0 * cet * ce2 * ¬ce3
ost31 = ¬in1 * in0 * ce1 * ce2 * ¬ce3
ost33 = in1 * ¬in0 * ce1 * ce2 * ¬ce3
ost21 = statec
ost25 = ackc * ¬ce1 * ¬ce2 * ce3
ost26 = ¬ackc * ¬ce1 * ¬ce2 * ce3
ost23 = ¬in1 * ¬in0 * ce1 * ¬ce2 * ce3
ost24 = in1 * in0 * ce1 * ¬ce2 * ce3
ost30 = ¬in1 * in0 * ccl * ¬ce2 * ce3
ost32 = in1 * ¬in0 * ce1 * ¬ce2 * ce3
ost28 = ackb * ¬ce1 * ce2 * ce3
osL34 = stateb
ost35 = ¬ackb * ¬ce1 * ce2 * ce3
est2 = LAST fst2 CLEAR globalinit + init
fst2 = osL2 + ost3
cst4 = LAST fst4 CLEAR globalinit + init
fst4 = ost2 + ost3 + ost22 + ost16 + ost29 + ost20 + ost27 + ost21 + ost25 + ost26 + ost23 + ost24 + ost28 + ost34 + ost35
est5 = LAST fst5 CLEAR globalinit + init
fst5 = ost2 + ost3 + ost13 + ost14 + ost22 + ostl6 + ost29 + ost20 + ost27 + ost21 + ost25 + ost26 + ost23 + ost24 + ost28
est6 = LAST fst6 CLEAR globalinit + init
fst6 = ost2 + ost3 + ost13 + ost14 + ost22 + ost16 + ost29 + ost20 + ost27 + ost25 + ost23 + ost24 + ost28 + ost34 + ost35
est9 = LAST fst9 CLEAR globalinit + init
fst9 = ost2 + ost3 + ost16 + ost29 + ost25
est17 = LAST fst17 CLEAR globalinit
fst17 = ost13 + ost14 + ost22 + ost20 + ost27 + ost21 + ost26 + ost23 + ost24 + ost28 + ost34 + ost35
est18 = LAST fst18 CLEAR globalinit
fst18 = ost13 + ost14 + ost21 + ost26 + ost34 + ost35
error = ost3 + ost4 + ost5 + ost6 + ost17 + ost18 + ost19 + ost16
stateb = ostll + ost30
statea = ost9 + ost33
statec = ost31 + ost32
outa = ost13 + ost14
outc = ost21 + ost26
outb = ost34 + ost35
ccl = LAST cfl CLEAR globalinit
cf1 = ost13 + ost14 + ost22 + ost20 + ost27 + ost23 + ost24 + ost28
ce2 = LAST cf2 CLEAR globalinit
cf2 = ost16 + osL29 + ost20 + ost27 + ost25 + ost28 + ost34 + ost35
cc3 = LAST cf3 CLEAR globalinit
cf3 = ost22 + ost21 + ost26 + ost23 + ost24 + ost34 + ost35

Fig. 3. Logic implementing communication protocol.

to its code.

Example 3: In Fig. 3 we see the output of the NFA-to-logic compiler, with certain header information, indicating clocking and the borders on which signals appear, omitted. Because it turns out that there is only one nontrivial group, and that group does not have exactly a power of two states, we were able to eliminate the group bit, and, by not using the all-zeros code for any state in the group, detect the presence of a state in the group by one or more of the code bits being on.

The overall organization of the logic in Fig. 3 is not unlike that of a PLA. The variables are in three groups, designated by the letters e, f, and o. The first group, e, corresponds to columns in the and-plane of a PLA and represents the fact that a certain state is "enabled"; if its input symbol is now seen it can enable its successors for the next input cycle. Some states have private enablers, like *est2* for state 2. Other states are coded, and in Fig. 3 there are three coded enabler variables *ce1*, ce2, and ce3, combinations of which represent the enablers for various states. Note that not every state has an enabler, either private or coded. States without enablers have the same entering transitions as some other state that does have an enabler, and the same enabler serves for both.

The f group of variables are "feedback"; they correspond to columns in the or-plane. State *fX* at one time unit becomes *eX* at the next time unit by means of *lgen* statements such as

est2 = LAST *fst2* CLEAR glob&nit + *init*

which says that state 2 is enabled either by the initial signal *init*, or by *fst2* being on at the previous time unit. The output signals, such as *statea* or *OUTA*, also correspond to columns of the or-plane.

The o group of variables correspond to the terms of the PLA. For each state there is an o variable that is turned on when the state is enabled, and the proper input is seen. For example, line 1 of Fig. 3 says oat2 is on whenever state 2 is enabled (*est2* is on) and *noin* is seen on the input (detected by both wires in0 and *in*1 being off). Line 9 says that *ost*14 is turned on when input *noacka* is seen and state 14 is enabled (represented by the coded enabler bits being 100).

The only difference between a PLA structure and the organization of the variables in Fig. 3 is that the *statea*, *stateb* and *statec* variables do not fit into the scheme. Rather, we can view them as fed back from the or-plane, where they arc generated, to the and-plane, where they arc used, with no delay due to clocking. Thus, Fig. 3 can be used almost directly as input to a PLA generator that permits unclocked signals as an option.

IV. Evaluation of Results

It is difficult to compare the logic of Fig. 3 with the "best possible" logical description of an equivalent circuit. It appears that, when the ability of the *lgen* compiler to eliminate common subexpressions and perform other optimizations is taken into account, the resulting logic will be very close to that found in the hand designed PLA described below. Thus, we are optimistic that our automatic synthesis method behaves very well when amount of logic generated is the criterion used.

We can obtain a more concrete estimate of the quality of the circuit designed if we view it as a PLA specification and compare it with a PLA designed carefully by hand. In our hand design, we used three feedback wires. Two were used to binary code the "state," i.e., whether we were in the start condition, or in what we have called states a, *b,* and c. The third feedback wire indicated whether we were waiting for an acknowlcgcment or had received the acknowledgement and were waiting for the next input. Terms based on this encoding were written down and optimized using the *gry* PLA optimizer [II). The resulting PLA had:

1. 32 terms.
2. 17 columns in the and-plane, representing an initializing signal, the three feedback wires, and the five input wires (*in0*, inl, *acka*, *ackb*, and *uckc),* each of which except the initializer requires inversion.
3. 7 columns in the or-plane, representing the three feedback wires and four outputs (*ERROR*, *OUTA*, *OUTB*, and *OUTC*).

The resulting area is 32 * (17 + 7) = 768.

In comparison, the **PLA** constructed directly from Fig. 3 requires the following:

1. 30 terms (the o variables plus one term to carry the initial signal to the or-plane).†
2. 27 columns in the and-plane, consisting of
   a) 10 columns for the inputs and their complements.
   b) 7 columns for the private state enablers; these do not have to bc inverted.
   c) 3 columns for *statea*, *stuteb*, and *statec*; these also do not require inversion.
   d) 6 columns for the three coded enablers, which do require inversion.
   e) 1 column for the initial signal.
3. 17 columns in the or-plane, consisting of four output signals and 13 feedback wires.

The resulting size is 30 * (27 + 17) = 1320. This figure is 72% greater than the hand-designed one. The overhead of the I'LA borders will tend to reduce this figure somewhat, as will the fact that clocking is not nccdcd on six of the columns of the machine-generated PLA. But the fact that space is required for 13

† It is not unusual for PLA's generated from regular expressions to have fewer terms than hand-designed ones, because the former PLA's tend to USC one-hot codes (private enablers) for states, and that sort of code costs columns, but may save terms.

```
line      x151
symbol    in0(x[1]-x[2])
          in1(x[2] -x[1])
          badin(x[1] x[2])
          ack(x[3])
          stateia(x[4] x[5])
          stateib(x[4] -x[5])
          stateic(-x[4] x[5])
          start(-x[4] -x[5])
          noin(-x[1] -x[2])
          noack(-x[3])
output    OUTA
          OUTB
          OUTC
          stateoa
          stateob
          stateoc
          ERROR
su bcxp   somein=in0 + in1 + badin
subcxp    waitin=noin + badin
su bexp   allbut01=ack + badin
;
start waitin* (
          allbut01 ERROR +
          in0 stateoa +
          in1 stateob        -

+
stateia noack* OUTA (
                    somcin ERROR +
                    ack waitin* (
                              allbut01 ERROR +
                              in0 stateob +
                              in 1 stateoc
                              )
                    )
+
stateib noack* OUTB (
                    somcin ERROR +
                    ack waitin* (
                              allbut01 ERROR +
                              in0 statcoc +
                              in 1 state08


+
stateic noack* OUTC (
                    somein ERROR +
                    ack waitin* (
                              allbut01 ERROR +
                              in0 statcoa +
                              in 1 statcob
```

Fig. 4. Revised input to regular expression compiler.

feedback wires for the machine-gcncratcd PLA will serve to increase the ratio.

V. Correction of Errors

One important advantage of the regular expression approach to design, as with high-level descriptions in general, is that modifications arc easier to make, and make reliably, than with ad-hoc designs.

Example 4: It turns out that our design of Fig. **1** is not the simplest that meets the specifications of [AUY]. Rather, since the channel is assumed never to make a mutation error, only to lose signals, there is no need to distinguish between the three acknowledgements; whenever we receive an acknowledgement, wc know it was actually sent by the receiver, and we know that if the receiver is not broken, then it was sent in response to the receipt of the correct signal.

Thus, we should modify Fig. 1 in the following two ways.

1. *acka, uckb,* and *uckc* should be identified as the signal *ack;* similarly, their complements, *noacka,* etc., are identified as *noack.*
2. While waiting for an acknowledgement, there are no "wrong acknowledgements" to receive, so terms like

   (ackb + ackc + somein) ERROR

should be replaced by

somein ERROR

The resulting revision is shown in Fig. 4.

VI. Conclusions

Regular expressions, in conjunction with conventional state machine definitions of processes, is a promising way to design circuits at a very high level. The use of optimizing logic compilers to do the actual implementation may be superior to PLA implementation of regular expressions, but the evidence of the case described in this paper, and other cases we have analyzed by hand, is that even PLA implementation of regular expressions comes within a factor of two of the area used by hand-designed PLA's. E'urthcr, the problem of coding nondeterministic states is not yet fully resolved, and there is hope that better I'LA and/or logic implementations of regular expressions will be developed in the future.

References

[AUY] Aho, A. V., J. D. Ullman, and M. Yannakakis, "Modcling communications protocols by automata,"

Proc. Twentieth Annual IEEE Symposium on Foundations of Computer Science, pp. 267–273, 1979.

[F] Fraser, A. G., "Datakit-a modular network for synchronous and asynchronous traffic," *Proc. IEEE Intl. Conf. on Communications, 1979.*

[H] Hemachandra, L. A., "GRY, a PLA minimizer,", unpublished memorandum, Dept. of C. S., Stanford Univ., 1982.

[T] Trickey, H., "Regular expressions and NFAs," unpublished memorandum, Dept. of C. S., Stanford Univ., 1982.

[TU] Trickey, H. and J. D. Ullman, "A regular expression compiler," *Proc. IEEE* COMPCON, 1982.

[U] Ullman, J. D., "Description of NFA-to-logic compiler," unpublished memorandum, Dept. of C. S., Stanford Univ., 1982.