August 1981 Report. No. STAN-CS-81-871

Good Layouts for Pattern Recognizers

by

Howard W. 'I'rickey

Rcscarch sponsored by

National Scicncc Foundation
and
Defense Advanced Research Projects Agency

Department of Computer Science

Stanford University
Stanford, CA 94305

Good Layouts for Pattern Recognizers

by

Howard W. Trickey†

Computer Science Department

Stanford University

Abstract

A system to lay out custom circuits that recognize regular languages can be a useful VLSI design autornation tool. This paper describes the algorithms used in an implementation of a *regular expression compiler.*' Layouts that use a network of prograrnmable logic arrays (PLA's) have smaller areas than those of some other methods, but there are the problems of partitioning the circuit and then placing the individual PLA's. Regular expressions have a structure which allows a novel solution to these problems: dynamic programming can bc used to find layouts which arc in some sense optimal. Various search pruning heuristics have been used to increase the speed of the compiler, and the experience with these is reported in the conclusions.

Index Terms: VLSI layout, silicon compilers, string pattern recognition, control logic design, regular cxprcssions, dynamic programming, programmable logic arrays, partitioning.

---

† Work supported by an NSERC scholarship, NSF grant MCS-80-12907, and DARPA grant MDA 903-80-c-0107.

§1 Introduction

The design of VLSI circuits is currently a very time-consuming operation. Some of the recent work to help alleviate this problem has taken its lead from programming language compiler technology, where great strides have been made by using programs to convert high level descriptions into lower level programs. The idea of a *silicon compiler* to convert high level descriptions of circuits into layouts has arisen [1,4,5,10,11,12].

A problem with silicon compilers is the definition of a suitable circuit description language. Some languages are basically descriptions of the upper levels of a hierarchical design. These become "high level" descriptions when the lower levels of the hierarchy can be derived from libraries and/or a familiarity with the class of circuits being described, The "Bristle Blocks" [5] system is an example of this type of system: it can be used to describe a *data path chip* (registers, shifters, ALU's, etc., built around a data bus).

A second approach is to use a notation which gives the external behavior required. One method of doing this is to give a sort of program which runs on a machine specified at the register transfer level (10,12]. This technique is meant to be used for designing computer-like chips. Another notation, which can be used for specifying the controlling logic portion of any chip, is that of *regular expressions.* A regular expression can be used to describe a pattern: a sequence of states in which certain inputs must be seen. One can require that various outputs be given whenever certain patterns have been seen. Some of the many uses of pattern detectors can be found in [7]. This paper discusses a silicon compiler whose input is a regular expression and whose output is a layout for the pattern recognition circuit defined by that expression.

In particular, a way of laying out a circuit for a pattern recognizer in a small area will be described. It is fairly easy to give a programmable logic array (PLA) to implement a pattern recognizer, but a single PLA can be rather large. At the other extreme, one can have logic to recognize each basic symbol of the pattern, joining them up with other logic. Such a method can be proved to yield a layout with an area which is linear in the length of the expression [2], but in practice the resulting layouts have been found to be large. The regular expression compiler uses a network of PLA's, and it gives layouts better than either of the extremes.

The next section will explain how regular expressions represent patterns. Then the implementation of recognizers using networks of PLA's will be described. Numerous networks are possible, so a big part of finding a good layout involves searching for a the best (or at least, near-best) division of the expression. The fourth section will discuss how dynamic programming and some judicious heuristics can be used to effect this search. Finally, the last section will give some conclusions, based on experience, about what the various search heuristics can accomplish and how much they cost.

§2 Regular Expressions as Patterns

A regular expression is a notation for representing a set of strings of symbols. It is defined recursively as follows:

- The symbol is the most basic kind of regular expression. In the application to circuits, the occurrence of a symbol means that the input wires must be zero or one, according to the symbol definition, within the "current state".

0 If E and F are regular expressions, then the *union* $E + F$ is a regular expression which means: either E or F.

o If E and F are regular expressions, then the *concatenation* $E \cdot F$ (or simply EF) is a regular expression which means: E followed by F.

0 If E is a regular expression, then the closure E^* is a regular expression which means: zero or more occurrences of E.

- If E is a regular expression, then the positive *closure* E-t-t is a regular expression which means: one or more occurrences of E.

- If E is a regular expression, then the optional *occurrence* $E?$ is a regular expression which means:' zero or one occurrence of E.

- If E is a regular expression, then (E) is a regular expression (used for grouping). Unless parentheses are used, the unary operators have precedence over the binary operators, and concatenation has precederice over union.

The use of regular expressions to describe pattern recognizers is perhaps best seen by means of an example. The following is the complete input file required by the regular expression compiler for a small example:

```
line data[2]
symbol zero(data[1],-data[2]),one(-data[1],data[2]),any()
;
any (one any* zero + zero any* one) +
(one any* zero + zero any* one) any
```

The line declaration gives the wires that are input to the circuit. A line name can be subscripted (with [..]), as data is, to represent more than one wire. One can declare any number of lines. The *symbol* declaration gives the names of the symbols that will occur in the regular expression, with the values of the input wires which identify a symbol given in parentheses after its name. Here there are three symbols: zero, recognized when `data[1]` is a logical "1" and `data[2]` is a logical "0" (indicated by the "-" in front of `data[2]`); one, recognizetl when the data wires are reversed; and any, which doesn't specify either "1" or "0" for the data wires, so it is a "don't care." Note that any will be recognized at the same time as zero or one: there is no requirement that Lhe wire combinations for different symbols be disjoint..

The regular expression itself follows the declaration. This one gives all strings of symbols where either (a) the first symbol differs from the second last symbol, or (b) the second symbol differs from the last symbol. This expression will be referred to as PR2.

The pattern recognizer is a synchronous machine. The successive symbols of a string must appear in successive clock cycles (states) for the pattern Lo be recognized. Whenever the symbols seen in the preceding states form one of the complete strings specified by an

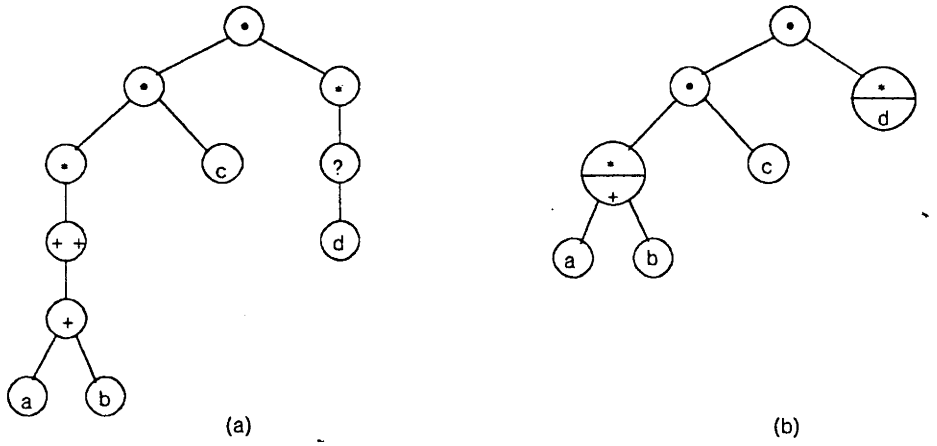


Figure I. (a) Expression Tree (b) Compressed Expression Tree

expression, an output signal is given.

The notion of an expression tree for a regular expression will be useful later on. The expression tree has symbols as leaves and regular expression operators as internal nodes. It is formed in the same recursive manner that expressions are: the tree for $E + F$ is a node containing "+" with the expression trees for E and F as children; similarly for the other operators. Figure 1(a) gives the expression tree for $\left((a+b)^{++}\right)^{*} \cdot c \cdot (d?)^{*}$.

A unary operator can be combined with the symbol or operator node beneath it. A cascade of unary operators can be reduced to a single one using obvious rules. This yields a *compressed expression tree,* such as the one shown in Figure 1(b) for $\left((a+b)^{++}\right)^{*} \cdot c \cdot (d?)^{*}$.

An NFA (nondeterministic finite automaton) can easily be given to implement a regular expression recognizer. In Figure 2, an NFA to recognize PR2 is shown. Initially the start state is made active. At any time there may be a number of active states. In each successive clock cycle, any active states with transitions marked by a symbol seen in that cycle will make the successors of those transitions active. States only remain active for one cycle unless explicitly reactivated. Whenever the final state is active, an output signal is given. If desired, the machine can keep operating so that it can detect overlapping occurrences of patterns.

The derivation of an NFA to recognize a pattern is straightforward. For details, see [2].

§3 Layout of Regular Expression Recognizers

An easy way to implement a regular expression recognizer is to use a PLA to simulate the NFA corresponding to it. Each state can be represented by a dynamic register whose value is calculated by the PLA using the inputs and the current state values (which are fed

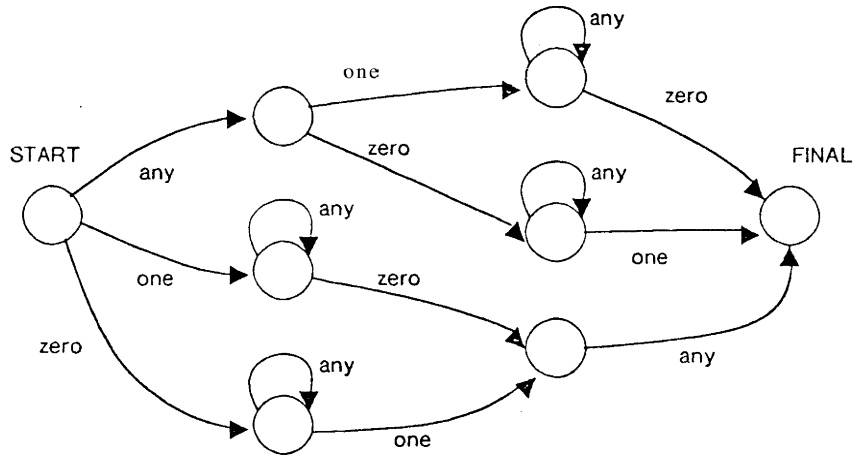


Figure 2. NFA to recognize PR2

back from the registers). Details of this method are given in [2].

The prohlern'is that the area used by such a layout will tend to grow quadratically with expression size. A method that leads to a linear growth of the required area is to implement each symbol as a dynamic register, together with logic which tests whether or not the symbol is on the input wires. The "symbol modules" have an enable input and a *recognized* output. By using appropriate connecting logic, it can be arranged that the symbol modules act like the states of the NFA, where a state is activated by asserting its enable input. (Actually, the circuit is not exactly like the NFA, because the state memory is distributed over the transition edges.) It was shown in [2] that as long as the expressions are compressed by combining cascades of unary operators, this method can yield a linear layout. A divide and conquer technique is used to decide where to place the symbol modules and connecting logic. A similar layout would be obtained using the systolic recognizers of [3].

Using individual logic for each symbol gives reasonable layouts, but experience with an implementation of this method has shown that for small expressions, the PLA method is better. This is perhaps to be expected, since the regularity of PLA's allows one to pack small numbers of gates more closely than is possible with an *ad* hoc circuit. Thus, the idea of using a combination of the two methods arose. The current implementation of the regular expression compiler uses PLA's for low level subexpressions, connected together with logic to take care of the operators near the root of the expression tree.

Suppose that one has laid out modules to recognize expressions E and F. It is assumed that these modules are rectangles, and that they have *enable* wires coming in at the left and *recognized* wires leaving at the right. Any input wires required to recognize the symbols in the module's expression must also enter at the left. Then the expressions $E + F$ and $E \cdot F$ can be laid out as shown in Figures 3(a) and 3(b), respectively. Operators which have been combined with unary operators can be implemented similarly, as illustrated in the layout for $(E \cdot F)^{++}$ in Figure 3(c). This type of layout is called an operator split. Note that no matter what operator is involved, the two subparts can be laid out either side by side (a

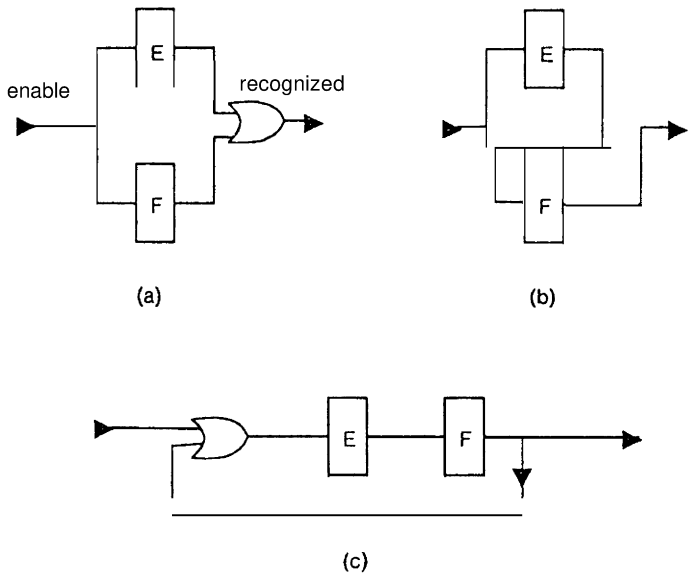


Figure 3. Operator splits: (a) $E + F$ (b) $E \cdot F$ (c) $(E \cdot F)^{++}$

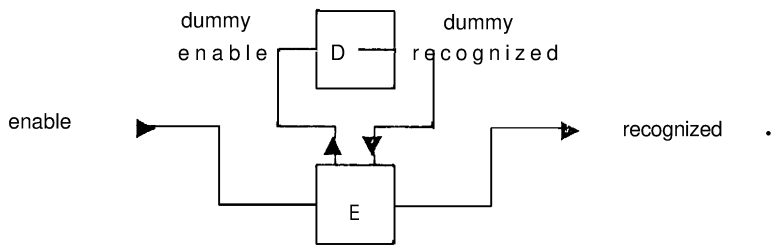


Figure 4. Substitution split

horizontal split) or one on top of the other (a *vertical* split).

The use of operator splits might be enough to accomplish a layout, but there is the problem that the layouts for the two operand expressions might have very different sizes. This would lead to a lot of white space when a rectangle surrounding the whole layout is defined. The solution to this is to do *a substitution split.* In a substitution split for an expression E, some node D deep in the expression tree for E is replaced by a dummy node. Then the expression rooted at D is laid out (the *dummy tree), as* well as the now smaller expression E (the *father tree).* E will have an *enable dummy* output wire and a *dummy recognized* in put wire. The former is attached to the enable input of D, and the latter is fed by the recognized wire of D, as shown in Figure 4.

The method for laying out a regular expression, given a compressed expression tree is to either (i) use a single PLA, or (ii) do an operator split or substitution split at the root and recursively lay out the subparts. This accomplishes the goal of using logic to form a

network of PLA's for recognizing the regular expression. What remains is to specify how to choose among the various layout strategies. At each stage of the recursion, the following choices must be made:

C1. Should a single PLA, an operator split, or a substitution split be used?

C2. If a split is used, should it be a horizontal or a vertical split?

C3. If a substitution split is used, which descendant expression should become the dummy tree?

One option of the regular expression compiler is to make the above choices guided by the principles that PLA's should be neither too small nor too large, and that when splits are used the subparts should be approximately equal in size. In this method, splits are performed by looking for a split, which yields subparts closest in size, and the recursion continues until the expressions are under some prespecified size. The "size" in terms of area is approximated by the *weight* — the number of leaves in the expression tree.

This heuristic method produces fairly good layouts quite quickly (in approximately 7 seconds on a VAX/780 for a 150-leaf expression). However, it usually requires some playing around with the parameters of the method to find the best layout possible with this scheme. Even then, a better layout is usually possible. There are several reasons why the heuristic method can be improved upon:

- The idea that two subparts should have the same area isn't strictly correct. What really is wanted is for the heights or widths to be about the same. Now, the PLA's generated from regular expressions all tend to have similar aspect ratios (height/width), so that if the subparts are simple PLA's then the "equal area" principle should hold. It seems plausible that if the subparts are themselves split, then there are some approximately square layouts for them, and so again the equal area principle should yield a reasonable layout. However, an unequal-area layout could be even better, and in practice there are many cases where one is better.

- The weight of an expression is only a rough indication of the area needed to lay it out. If the layout involves splits then the shape of the expression tree affects the economy of the layout.

- The area of a layout depends somewhat on the number of input wires needed. Thus, even if two subparts have equal weights, the layout for one subpart might be taller if it uses more inputs.

- Finally, some optimizations are performed when laying out a PLA (having an effect similar to factoring the expression). This is another reason why the weight of an expression only roughly predicts the area of the resulting layout.

To overcome some of these problems, the regular expression compiler has another option: search systematically through a specified collection of layout strategies, looking for the best one.

§4 Finding Optimal Layouts

An exhaustive search can find the best layout for an expression, given that one is using the general scheme of operator and substitution splits with PLA's at the lowest level. All possible combinations of choices C1, C2, and C3 can be tried, using all possible layouts for the subparts in the case of splits.

Clearly, such an exhaustive search would be very time consuming, even for for quite small expressions. One way to avoid a lot of the work is to note that the dimensions of a layout for an expression remain about the same when the layout is made part of a layout for a containing expression. There is often some height increase when a module is incorporated as a subpart in a split, because the input wires to the other subpart may have to run through the module. This effect can be calculated, however, so the conclusion is that the strategies for laying out a given subexpression need be calculated only once. The significance of this is that a sort of dynamic programming can be used to effect the search.

Dynamic programming can be used to find optimum strategies for problems that can be broken up as follows: starting out at a first "stage", some choices are made leading to a collection of smaller, similar problems — the second stage; this continues until some final stage is reached where there are no more choices to be made. If the problem is such that a knowledge of all the optimal solutions at stage i is sufficient to find all the optimal strategies for stage $i-1$, then dynamic programming can be used. The layout scheme satisfies this condition (approximately), where the problems of stage i are finding the best layouts for subexpressions whose roots are at depth i in the expression tree.

One problem in applying dynamic programming to layout is that one needs more than just the minimum area layouts for the subexpressions: a slightly larger layout may be better to use as a subpart, in a split if its height (or width) is closer to that of the other subpart. What is really needed is the best area for all possible heights and widths. In practice this would probably mean keeping 311 layouts tried, which would eliminate most of the savings that are entailed by the use of dynamic programming.

The solution to this problem is to use an approximation: divide up the continuum of possible aspect ratios into a small number of intervals, and for each subexpression keep only the smallest-area layout in each aspect ratio interval.

If the only splits allowed were operator splits, then the search for a layout could follow the standard dynamic programming procedure: start at the last stage (the lowest leaves) and find layout strategies there; then move up the expression tree, trying single PLA's and operator splits. Trying an operator split is a relatively quick operation, where the dimensions of the children are added to the logic dimensions to give the resulting layout dimensions. (There is also an adjustment for input wires, as mentioned above.)

It is the substitution split which greatly increases the work required to find an optimal layout. After a descendant expression is replaced by a dummy node, optimal layouts have to be found for the father tree. Only some of the layouts found so far can be used: those for subexpressions not involving the dummy tree. Thus, a somewhat independent layout problem must be solved for each possible father tree, and each of those will involve still more father tree layout problems. The work required increases dramatically as the root is approached because there are many more possible father trees (one for each descendant,

not including the subproblem father trees).

In fact., by the time all the subproblems have been solved for an expression, layouts will have been found for all possible *prefix* trees. A prefix tree is what is left attached to the root after any combination of descendants have been replaced by dummy nodes.

To get some idea of how many prefix trees there can be, consider T_n, the complete binary tree of n levels. Let S_n be the set of prefix trees of T_n, and N_n be the number of trees in S,. Any binary tree with $\leq n$ levels is a prefix tree of T_n. A binary tree of $\leq n$ levels can only be formed by having a root with a member of S_{n-1} or the empty tree as left child, and a member of S_{n-1} or the empty tree as right child. Therefore,

$$N_n = (N_{n-1} + 1)^2 \leq 2^{2^{n-1}}$$

T_n has $m = 2^n - 1$ nodes, so $N_n < 2^{m/2}$. This calculation shows that just enumerating the possible father trees for a balanced expression of 30 leaves (i.e., about 60 nodes) is out of the question.

An obvious partial solution to this is to have some minimum expression size — say 6 leaves — below which an expression will not be considered as a subpart of a split. This has the effect of chopping off some number, I, of the most populous levels from consideration as dummy tree roots. This changes the above calculation so that now $N_{n-l} < 2^{m/2^{l+1}}$. With this improvement, one could perhaps handle expressions of 30-50 leaves, but it might take a long time, considering that at the very least a PLA has to be considered out for each father tree tried.

To be able to handle expressions with up to, say, 300 leaves, the search needs further pruning. The "equal area" principle mentioned above suggests that splits where one subpart is much bigger than the other are likely to waste space. The regular expression compiler has a *split-ratio* parameter, S. Splits will only be considered when the weight ratio of one subpart to the other is in the range [1/S, S]. It has been found that in practice $S \approx 2$ yields layouts as good as $S = \infty$.

When all splits are not considered, there turn out to be a large number of subexpressions whose layouts couldn't possibly be used in the layout for the whole expression. This means that the dynamic programming paradigm of working on the expression tree bottom-up wastes a lot of calculation. It is better to work top-down, looking for subpart layouts whenever required.

To retain the advantages of dynamic programming, a dictionary of layouts is kept so that layouts need never be found twice for the same subexpression. The dictionary can contain layouts for each of the possible prefix trees of each subexpression. This is allowed by having the dictionary indexed by (e, l), where e is an expression node and l is an excision list: nodes that have been replaced by dummies.

Here is the final algorithm for finding layout strategies. There are three tuning parameters, to allow trading off search thoroughness for execution time: S, the split-ratio; L, the lowest weight allowed for a PLA; and H, the highest weight allowed for a PLA.

```
FindStrategies(x:ExpressionTree, l:ExcisionList):
{ Find strategies for layout of the expression x,
```

```
    where the expression nodes on l have been replaced by dummies }
      if LookupStrategies(x,l) ≠ INIT then return
         { already found strategies for (x,l) }
      if x.weight ∈ [L..H] then
           TryPLA(x,l)
      if x.lchild.weight/x.rchild.weight ∈ [1/S ... S] then begin
           FindStrategies(x.lchild,l)
           FindStrategies(x.rchild,l)
           TryOperatorSplit(x,l)
           end
      for all descendants y of x such that
        (x.weight−y.weight+1)/x.weight ∈ [1/S . . . S] do begin
           ExciseDummy(x,y)        { replace y by DUMMY in x }
           FindStrategies(x,Append(l,y))
           FindStrategies(y,l)
           TrySubstitutionSplit(x,l,y)
           end
  end FindStrategies

  TryPLA, TryOperatorSplit, TrySubstitutionSplit:
{ These procedures calculate the dimensions of the layouts
    implied by their arguments.  For the splits, all possible layouts
    resulting from combinations of strategies for the subparts are tried.
    The best strategies in various aspect ratio ranges are entered
    into the dictionary. }

  LookupStrategy(e,l):
{ This function looks up in the dictionary the layout strategies
    for expression e with excisions list l.  Any members of l which are not
    descendants of e, or are descendants of other members of l, are ignored.
    INIT is returned if no strategies have yet beer; sought for (e,l). }
```

§5 Performance of the Regular Expression Compiler

The regular expression compiler has been implemented in C on a VAX/780. It can produce layouts using either the heuristic method or the dynamic programming method. By appropriately setting the parameters for the heuristic method, one can also find the layout as a single PLA or as a network of logic connecting individual symbol recognizers. This section will report how the compiler performs on some sample expressions.

The first series of expressions is the PR series. The PR2 expression was given in Section 2. The others in the series have the same line and symbol declarations, and the following definitions ($\texttt{any}^n$ is used as shorthand for n occurrences of any):

Expression Name	Weight	Depth	Layout Method	L	H	S	Area ($M\lambda^2$)	Time (secs)
PR8	72	14	single PLA				.97	2.8
			all logic				.85	6.7
			heuristic	4	17		.58	2.8
			dyn. prog.	6	60	1.5	.56	14.0
			dyn. prog.	6	60	2.0	.55	24.0
			dyn. prog.	6	30	3.0	.55	55.7
PR16	160	23	single PLA				4.43	11.5
			all logic				2.28	15.3
			heuristic	4	17		1.69	6.9
			dyn. prog.	6	40	1.5	1.47	34.4
			dyn. prog.	6	30	2.0	1.23	159.6
PR32	352	40	single PLA				21.00	130.3
			all logic				8.88	35.9
			heuristic	4	17		3.87	17.3
			dyn. prog.	6	40	1.7	3.55	267.1
			dyn. prog.	7	25	2.0	3.19	1482.5

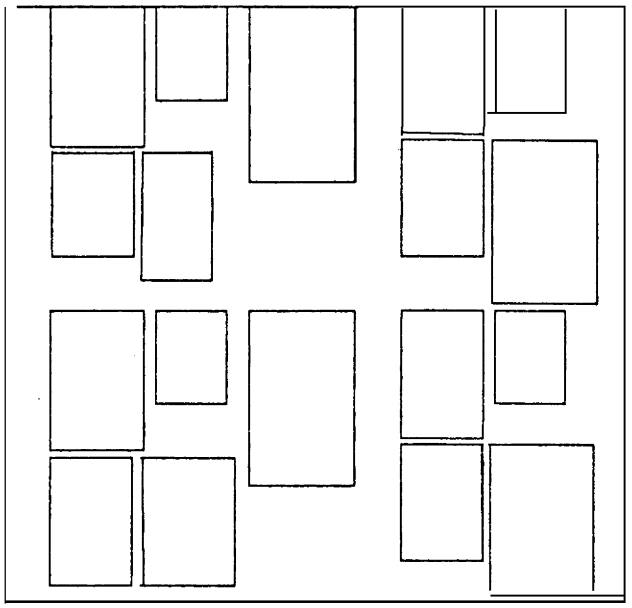
Table 1. Data for PR expressions

$$\begin{aligned} \text{PR4} &= \text{any}^2(\text{PR2}) + \text{PR2}\,\text{any}^2 \\ \text{PR8} &= \text{any}^4(\text{PR4}) + (\text{PR4})\text{any}^4 \\ \text{PR16} &= \text{any}^8(\text{PR8}) + (\text{PR8})\text{any}^8 \\ \text{PR32} &= \text{any}^{16}(\text{PR16}) + (\text{PR16})\text{any}^{16} \end{aligned}$$

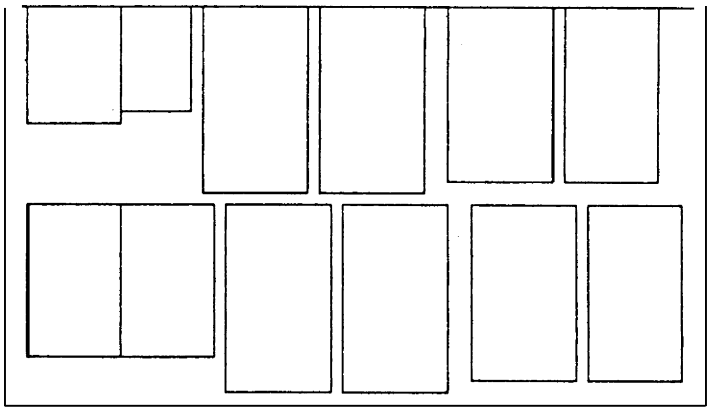
PRn is recognized whenever the last n inputs fail to match the first n. The results of running the regular expression compiler on the PR series is given in Table 1. The times given in the last column are CPU seconds on the VAX. Areas are in $\lambda^2 \times 10^6$, where λ is the minimum feature size. The "heuristic" results were the best that could be found by varying the parameters (there is another parameter, not shown, which indicates the desired shape of the final layout). It can be seen that both the heuristic method and the dynamic programming method are quite a bit better than the single-PLA or all-logic methods. Dynamic programming beats the heuristic method by an amount which increases with the expression size. Several dynamic programming results are shown to give some idea of the tradeoff between search thoroughness and execution time that occurs. Sketches of the layouts found by the compiler for PR16 are shown in Figures 5(a)(heuristic) and 5(b) (dynamic programming). The boxes are the individual PLA's.

The next series of expressions to be tried were the SEQ expressions, where SEQn has the form:

```
line l[n]
symbol a1(l[1]), b1(-l[1]), a2(l[2]), b2(-l[2]), ..., an(l[n]), bn(-l[n])
symbol any()
```



(a)



(b)

Figure 5. Layout sketches for PR16: (a) heuristic (b) dynamic programming

b1 + any* (a1 b2 + a2 b3 + . . . + a*n* any++)

These expressions signal if the input wires are not turned on in sequence. The SEQ expressions are different from the PR ones in that they have a large number of input wires, so that the heuristic strategy (which doesn't pay attention to how many inputs a module needs) might be expected to do poorly. Another fact about these expressions is that the expression trees are tall and sparse. The PR expressions had rather bushy trees. Table 2 gives the results of using the regular expression compiler on the SEQ expressions.

Expression Name	Weight	Depth	Layout Method	L	II	S	Area ($M\lambda^2$)	Time (w c s)
							.30	1.5
			all logic				.51	4.0
							.28	2.1
SEQ16	34	19	single dyn. heuristic prog. PLA	4 6	17 17	1.7	.24	5.0
SEQ32	66	35	single PLA				.97	3.5
			all logic				1.23	9.3
			heuristic	4	28		.64	3.4
			dyn. prog	6	70	1.7	.61	27.5
SEQ64	130	67	single PLA				3.48	9.2
			all logic				3.33	20.7
			heuristic	4	35		1.76	7.9
			dyn. prog.	6	30	1.7	1.62	186.0
BSEQ16	32	5	single PLA				.27	1.4
			all logic				.34	3.2
			heuristic	4	20		.23	1.6
			dyn. prog.	6	40	1.7	.23	2.7
BSEQ32	64	6	single PLA				.92	3.0
			all logic				.74	6.8
			heuristic	4	25		.59	3.6
			dyn. prog.	6	65	1.7	.59	8.9
BSEQ64	128	7	single PLA				3.39	9.8
			all logic				2.28	18.4
			heuristic	4	35		1.91	7.6
			dyn. prog.	6	30	1.7	1.53	15.9

Table 2. Data for' SEQ and BSEQ expressions

The final group of expressions is a slight modification of the SEQ group. To see what effect the depth of the tree has on the execution time, the BSEQ expressions were formed: they are just copies of the SEQ expressions without the `b1+any++` at the beginning, factored so that they form completely balanced binary trees. For example, BSEQ4 is:

```
((a1 b2 + a2 b3) + (a3 b4 + a4 any++))
```

The results of compiling these expressions are also given in Table 2. It can be seen that the compiler works faster on the bushy BSEQ expressions than it did on the corresponding SEQ expressions. This is because there are a smaller number of possible dummy nodes which satisfy the split-ratio requirement in the bushy trees.

§6 Evaluation and Conclusions

It has been shown that regular expressions have a structure which makes them quite amenable to a "divide-and-conquer" partitioning and placement procedure which runs fairly quickly. Clearly, the network-of-PLA's approach is superior to the single PLA or all-logic methods.

The program could certainly run a lot faster if substitution splits weren't tried, but it has been found that these are definitely required. Perhaps the expressions could be parsed in such a way that the children would always be about the same weight: there is some freedom allowed because concatenation and union are associative operators. However, the closure operators form barriers to arbitrary reparsing, so in general one cannot balance the children.

The search over a range of possible dummy tree roots is another aspect which slows the compiler. If one tries only that node which yields the best weight ratio between the father and dummy trees, the resulting areas are somewhere between those found by the heuristic method and dynamic programming. For example, this modification led to the same layout as full dynamic programrning for SEQ16, but for SEQ32 it only did as well as the heuristic method. It was found that. one had to try the five best dummy tree roots before the full dynamic programming layout would be found for SEQ32. The execution times using the best-dummy-only modification were quite close to those of the heuristic method, so perhaps this is the most useful method of all, for small to medium sized expressions.

The dynamic programming method requires keeping a number of "best" layouts for expressions, in each of a number of different aspect ratio ranges. Varying the number of these ranges has some effect on the ability of the compiler to find good layouts. Originally, three ranges were used. This seemed to work, but when the compiler was changed to keep layouts for six ranges, the results were quite a lot better — at least for the larger expressions.

To sum up, each of the capabilities of the regular expression compiler adds incrementally to the quality of the layout, at a cost of extra execution time. However, even the most, expensive dynamic programming searches are still quite fast compared to other aspects of VLSI design — such as check plotting — so it is not unreasonable to use dynamic programming always.

The work described in this paper has some resemblence to previous work on graph theoretic approaches to partitioning [9], but the problem is somewhat more tractable when trees are involved. Also, the idea of doing the placement by recursively splitting the plane into halves has been used before [6]. Not much has been done on automatically choosing a network of PLA's to implement a sequential circuit, though there has been some work done on optimizing single PLA's [8]. A circuit realization using a network of PLA's is given in [I], but the user must specify the splits with a hierarchical circuit definition.

The regular expression compiler is still undergoing improvements. Currently, the ability to have numerous "output signals" embedded in the expression is being incorporated. Also, more PLA optimizations are going to be done. In particular, non-overlapping NFA states will be detected and a group of such states can be assigned binary-encoded state identifiers. This should reduce the current tendency for the PLA's to be fairly sparse.

There are plans to use the compiler to generate much of the control logic for a VLSI chip being designed.

Acknowledgements

The regular expression compiler was originally designed and implemented by Jeff Ullman at Standford University. The author has added the dynamic programming feature and made various other improvements.

References

[1] R. Ayres. "Silicon Compilation — A Hierarchical Use of PLAs." *16th Design Automation Conf. Proceedings,* pp. 314–326, June 1979.

[2] R.W. Floyd, and J.D. Ullman. "The Compilation of Regular Expressions into Inegrated Circuits." Tech. *Rep.* STAN- CS- *80-* 798, Stanford Cornputer Science Dept., April 1980.

[3] M.J. Foster, and M.T. Kung. "PRA: Programmable Building Blocks for Recognizing Regular Languages in VLSI." *Unpublished memorandum,* Dept. of Cornputer Science, Carnegie-Mellon, 1981.

[4] J.P. Gray. "Introduction to Silicon Compilation." *16th Design Automation Conf. Proceedings,* pp. 305-306, June 1979.

[5] D. Johannsen. "Bristle Blocks, A Silicon Compiler." *16th Design Automation Conf. Proceedings,* pp. 310-313, June 1979.

[6] U. Lauther. "A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation." *16th Design Automation Conf. Proceedings,* pp. I-10, June 1979.

[7] A. Mukhopadhyay. '(Hardware Algorithms for Non-numeric Computation." *IEEE Transactions on Computers,* C-28, No. 6, pp. 384-393, June 1979.

[8] J.P. Roth. "Programmed Logic Array Optimization." *IEEE Transactions on Computers,* C-27, No. 2, pp. 174--176, February 1978.

[9] D.G. Schweikert, B.W. Kernighan. "A Proper Model for the Partitioning of Electric Circuits." *8th Design Automation Workshop Proceedings,* pp. 56–62, June 1972.

[10] D.P. Siewiorck, M.R. Barbacci. "The CMU W-CAD System — An Innovative Approach to Computer Aided Design." *AFKPS Fall Joint Computer Conference,* Vol. 45, 1976.

[11] J.D. Williams. "STICKS — A Graphical Compiler for High Level LSI Design." *National Computer Conf. Proceedings,* pp. 289-295, 1978.

[12] G. Zimmerman. "Cost Performance Analysis and Optimization of Highly Parallel Computer Structures: First Results of a Structured Top-Down Design Method." *4th International Symposium on Computer Hardware Description Languages,* October 1979.