Avoiding Latch Formation in Regular Expression Recognizers

M. J. Foster
Computer Science Department
Columbia University
New York City 10027

6 October 1986

CUCS-233-86

Index terms

Custom VLSI, pattern matchers, regular expressions, source-to-source transformations, specialized silicon compilers

Abstract

Specialized silicon compilers, or module generators, are promising tools for automating the design of custom VLSI chips. In particular, generators for regular language recognizers seem to have many applications. This paper identifies a problem called latch formation that causes regular expression recognizers to be more complex than they would first appear. If recognizers are constructed in the most straightforward way from certain regular expressions, they may contain extraneous latches that cause incorrect operation. After identifying the problem, the paper presents a ''source-to-source'' transformation that converts regular expressions that cause latch formation into expressions that do not. This transformation allows regular expression recognizers to be simpler and smaller, thus adding to the advantages of specialized silicon compilers.

1. Introduction

One of the most important tools available to today's designers for speeding up the process of designing custom VLSI is the *specialized silicon compiler.* Sometimes called *module generators*, specialized silicon compilers are tools that combine knowledge of a specific application area with a set of primitive cells and rules for combining the cells to produce a layout. Specialized silicon compilers will only produce layouts for their specialized areas of application, but will do a very good job within those areas. The chips that they produce will be small and fast. Specialized silicon compilers have been built for digital signal processors [6, 14], combinational logic [16], data paths [3, 11, 13], general-purpose processors [17], synchronizers [2, 5], and the topic of this paper, pattern matchers [9, 12]. Anything that improves the efficiency of the chips produced by specialized silicon compilers is a worthwhile contribution to the field of VLSI design automation.

This paper discusses specialized silicon compilers that produce recognizers for regular expressions. A regular expression recognizer is a pattern matching circuit in which the pattern is specified by a regular expression. A specialized silicon compiler for these recognizers will accept a regular expression as input, and produce the layout of a recognizer for that expression. Regular expressions have seen wide application in computer science; among other tasks, they have been used to specify lexical analyzers for programming languages, controllers for sequential machines, filters for on-the-fly database search, patterns in image processing, and communication protocols. Regular expression recognizers therefore have wide application, and are an especially promising application area for specialized silicon compilers.

The contribution of this paper is a ''source-to-source'' transformation of regular expressions that allows more efficient recognizers to be constructed for them. The most straightforward circuits for constructing regular expression recognizers exhibit a problem: recognizers for some expressions contain superfluous latches due to interactions between the cells. Formation of these latches leads to recognizers that function incorrectly. Previous solutions to the problem of latch formation have used more complicated circuits to eliminate or reset the latches. These more complicated circuits increase the size of recognizers, and may make them run more slowly. The transformation introduced in this paper eliminates the expressions that cause latch formation, and so allows the simple, straightforward circuits to be used for all expressions. This allows specialized silicon compilers to produce efficient recognizers for regular languages.

2. Regular Expression Recognizers

This section gives a notation for regular expressions and describes how to compile them into circuits. Straightforward compilation of some regular expressions leads to a problem called *latch formation*, in which extraneous latches are formed within the compiled circuit. These latches lead to incorrect operation. This section indicates how the problem arises and mentions some techniques that have been used to combat it.

A regular expression describes a regular language over some alphabet Σ. A regular expression may represent the empty set ($\varnothing$), the empty string (ε), or any set of strings that can be built up by concatenation, union and repetition from ε and the single characters of Σ. A regular expression over Σ may include some characters that are not in Σ, such as operators and parentheses. Assuming that the characters in the set $\{\phi\ (\ )\ *\ +\}$ are not in Σ, the syntactically correct regular expressions over Σ can be defined inductively as follows.

- ϕ is a regular expression over Σ.
- If $a \in \Sigma$ then a is a regular expression over Σ.
- If α and β are regular expressions over Σ, then so are $\alpha;\beta$, $(\alpha + \beta)$, and $(\alpha)^*$.

The meaning of a regular expression can be defined inductively based on the form of the expression. The set of strings $L(\rho)$ represented by a regular expression ρ is:

- The empty set if ρ is ϕ.
- $\{a\}$ if ρ is a.
- $L(\alpha) \cup L(\beta)$ if ρ is $(\alpha + \beta)$.
- $\{\sigma_1\sigma_2$, where $\sigma_1 \in L(\alpha)$ and $\sigma_2 \in L(\beta)\}$ if ρ is $\alpha;\beta$.
- $\{\varepsilon\} \cup \{\sigma_1\sigma_2 \ldots \sigma_n$, where n is any positive integer and $\sigma_i \in L(\alpha)\}$ if ρ is $(\alpha)^*$. In this article, λ will be used as an abbreviation for ϕ^*. Thus, $L(\lambda) = \{\varepsilon\}$.

This article uses a regular expression to denote the set of strings it represents; for example, $abc \in (a;b + c)^*$.

Regular expressions describe patterns; very often one wants to build pattern recognizers for these patterns. A recognizer circuit for a regular expression accepts a stream of characters from the alphabet of the expression as input, and produces a stream of bits. The operation of a recognizer circuit occurs on discrete clock ticks, or *beats*. On each beat, the recognizer reads a character from the input and produces an output bit. The first bit precedes the string of characters, then the recognizer produces one bit after reading each character. Each bit indicates whether the stream of characters immediately preceding it is in the language of the regular expression. For example, if a recognizer for the expression $(a;b + c)^*$ were given the stream *ababc* it would produce the bit stream 001011.

Several researchers have described tree-based techniques for compiling regular expressions into recognizer circuits [1, 7, 9, 10, 15]. All of these techniques use produce tree-structured circuits in which the nodes of the tree correspond to characters of the regular expression. Edges of the tree carry text and state information between nodes. The nodes of the tree are usually chosen from a library of standard cells, with the edges being the wiring between the cells.

The circuits of Mukhopadhyay are typical examples of tree-based recognizer circuits. These circuits use five types of cells: one for each of the three operators (union, concatenation, and Kleene *), one for the symbol ϕ, and one to recognize individual characters. Figure 2-1 shows the comparator for individual characters. On each beat, a character is input at the same time as the ENB signal. The RES signal is set to true for the following beat if and only if ENB is true and the text character matches the pattern character.

The three operator cells for the expression tree recognizer are shown in Figures 2-2, 2-3, and 2-4. These combine ENB and RES signals from their operands to produce signals for a larger expression. A recognizer built from these cells is a tree with the same structure as the expression tree of the regular expression, with operator cells in place of operators and comparators in place of letters of the alphabet. For example, to build a recognizer for A;B, a comparator for A is connected to the left port of the

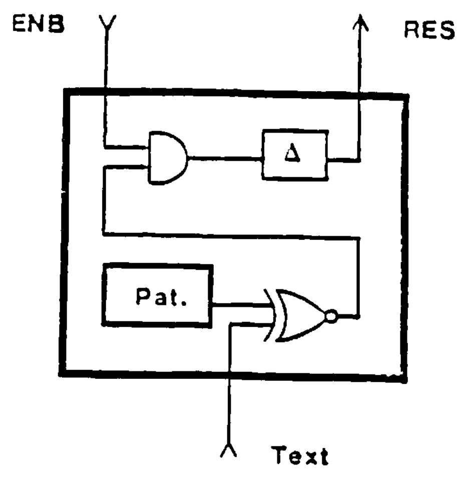

Figure 2-1: Comparator for expression-tree based recognizer

concatenation cell, and a comparator for B is connected to the right port. A recognizer constructed using these cells outputs RES on beat i if and only if some string in the language of the recognizer is input on beats $i-n$ through $i-1$ and ENB_{i-n} is true. (The cell for ϕ is not shown, but is simply a connection to logical 0: it always outputs 0 on RES.)

There is a problem with the expression tree cells in Figures 2-1 through 2-4, which was first pointed out by Backhouse [4]. If a recognizer is constructed for an expression of the form E*, where the empty string is contained in E, the output result will be latched to true. This can be seen most easily in the recognizer for $(a^*)^*$, shown in Figure 2-5. The OR gates in the two Kleene star cells are cross coupled so that they form a latch. If they are ever set to true, they can never be reset to false. Thus if ENB is true on beat 1, and the character b is input on that beat, the recognizer will erroneously output 1 on RES during beat 2. Using these cells then, correct recognizers can be constructed only for expressions in which the star operator is never applied to a subexpression containing the empty string. This problem is common to all of the tree-based recognizers that have been discussed in the literature.

Several solutions to the latching problem have been proposed. Foster [9] uses a *clocked OR* gate in the Kleene closure cell in place of the OR gate. The clocked OR gate behaves like a normal OR gate, except that it briefly sets its output to 0 between beats. This clears any latches formed by a loop of OR gates, and it has been proven that recognizers built using this modified cell operate correctly for all regular expressions [8]. More recently, Ananthamraman [1] has designed a set of cells that can be interconnected to produce delay-insensitive recognizers. Among their other advantages, Anantharaman's cells avoid the latching problem by avoiding direct feedback between OR gates. The RES signals in these cells have more than two values; they take on a special value for matches of the empty string. The cells are designed so that the special value cannot act as an enable signal. In particular, a match of the empty string by a Kleene closure cell cannot re-enable the same cell.

While these techniques correct the problem of latched recognizers, they complicate the cells. The clocked OR gate is somewhat larger than an ordinary OR gate, and it requires connection to a clock that signals the ends of beats. The delay-insensitive circuits used in Anantharaman's cells are much larger and

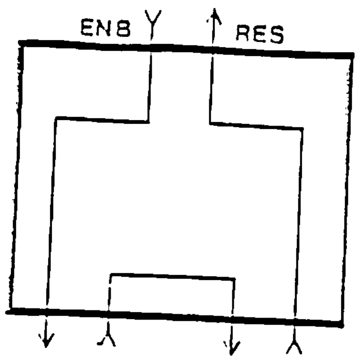

Figure 2-2: Concatenation cell

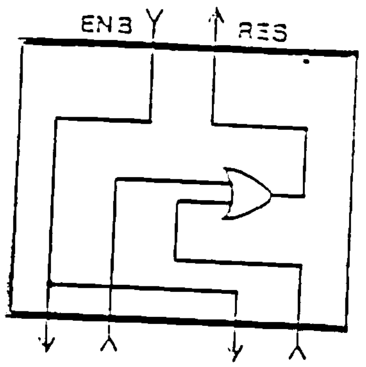

Figure 2-3: Union cell

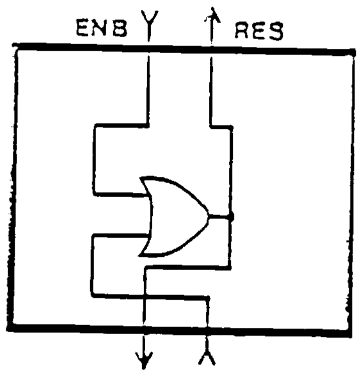

Figure 2-4: Kleene closure cell

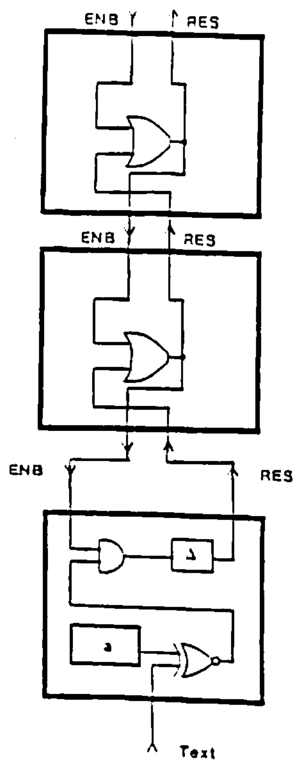

Figure 2-5: Faulty recognizer for (a*)*

more complicated than their synchronous counterparts. In many cases in which a small, simple recognizer is desired, the anti-latching techniques add too much overhead to the prinitive cells.

3. A Conversion Theorem for Regular Expressions

This paper provides a new method for avoiding latch formation in tree-structured recognizers. The idea is to transform the regular expression for which a recognizer is to be built into a new expression that will not lead to latch formation. Latch formation occurs when the star operator is applied to an expression that contains the empty string; that is, any recognizer for an expression E*, where $\varepsilon \in$ E, will have extraneous latches. If we could transform E into a new expression N(E) such that N(E)* = E* and $\varepsilon \notin$ N(E), we could avoid latch formation by building the recognizer for N(E)*. This recognizer would recognize the language E*, and would operate correctly using the simple cells of Figures 2-1 to 2-4.

The obvious way to transform E is to form the set difference E–λ. Although there is no way to build a cell for the set difference operator [8], E–λ is a regular language, and so can be represented by some regular expression D(E). Clearly D(E)* = E* and $\varepsilon \notin$ D(E). Unfortunately, the expression D(E) can be much longer than E. The set difference operator is not used in regular expressions, and converting an expression containing set difference to one that does not can greatly increase the length of the expression. For example, there is a sequence $\{E_n\}$ of regular expressions of increasing length such that the shortest expression for $D(E_n)$ is about $n/2$ times as long as E_n. (The first two expressions in the sequence are a*b*, (a*b* + c*d*)(e*f* + g*h*). In general, $E_{i+1} = (E_i + E'_i);(E''_i + E'''_i)$, where the four copies of E_i

have disjoint alphabets.) Thus there is no constant bound on the ratio of the length of D(E) to the length of E. Since the size of a tree-based recognizer is proportional to the length of its regular expression, transforming E to D(E) may lead to recognizers that are too large. We therefore need a different transformation of E.

The following theorem provides a transformation that is appropriate for constructing recognizers. It shows that any expression of the form E* can be transformed into a new expression N(E)* in which latches are not formed, and which is no longer than E*. This solves the problem of latch formation in tree-based recognizers for regular expressions. We first state without proof some easily verified facts about regular expressions.

Lemma : If P and Q are regular expressions the following statements hold.

a. $P^{**} = P^*$

b. $(P^* + Q^*)^* = (P + Q)^*$

c. If $\varepsilon \in P$ and $\varepsilon \in Q$ then $(P;Q)^* = (P + Q)^*$.

d. $\varepsilon \in (P + Q)$ if and only if either $\varepsilon \in P$ or $\varepsilon \in Q$.

e. $\varepsilon \in (P;Q)$ if and only if $\varepsilon \in P$ and $\varepsilon \in Q$.

These facts are used in the proof of the following theorem.

Theorem : For any regular expression R there is a regular expression N(R) such that

- N(R) does not contain the empty string,
- $R^* = (N(R))^*$
- N(R) is no longer than R.

Proof: We compute N(R) recursively, based on the expression tree of R. If $\varepsilon \notin R$ then N(R) = R. (This grounds the recursion, since N(a) = a for any member of the alphabet.) Otherwise, there are three cases.

1. $R = P^*$
2. $R = P + Q$, where either P or Q contains ε
3. $R = P;Q$, where both P and Q contain ε

In case 1, N(R) = N(P). In cases 2 and 3, N(R) = N(P) + N(Q). Clearly N(R) is no longer than R. We must now show that N(R) does not contain ε and that $N(R)^* = R^*$

We can show both of these facts by induction. First, we show that N(R) does not contain ε. If R has length 1, then it must be either a letter of the alphabet, or one of the expressions ϕ or λ. N(R) is then either ϕ or a letter of the alphabet, neither of which contains ε. Thus the base case of the induction holds. To prove the inductive step, suppose that for every expression P of length less than *i*, N(P) does not contain ε. Suppose R has length *i*. If R does not contain ε, neither does N(R). If R does contain ε, then N(R) is either N(P) or N(P) + N(Q), where P and Q have length less than *i*. In neither case does N(R) contain the empty string, since by the inductive hypothesis neither N(P) nor N(Q) can contain it.

Now we show that $R^* = (N(R))^*$. We need only consider cases in which R contains the empty string, since N(R) and R are identical otherwise. If R has length 1 and contains the empty string, it must be λ. To prove the base case, note that N(R) is then ϕ, and $(N(R))^* = R^* = \lambda$. For the inductive step, suppose that $R^* = (N(R))^*$ as long as the length of R is less than *i*. If R has length *i* and contains the empty string, then R must be one of the three forms noted above.

1. If R = P*, then N(R) = N(P) and (N(R))* = (N(P))*. By the inductive hypothesis, (N(P))* = P*. So (N(R))* = P* = P** = R*.
2. If R = P + Q, then N(R) = N(P) + N(Q). So (N(R))* = (N(P) + N(Q))* = ((N(P))* + (N(Q))*)*. By the inductive hypothesis, (N(P))* = P* and (N(Q))* = Q*, so (N(R))* = (P* + Q*)*. But this is equal to (P + Q)*, which is R*.
3. If R = P;Q, then N(R) = N(P) + N(Q). By the same reasoning as in case 2, (N(R))* = (P + Q)*. Now notice that, since we are assuming that R contains the empty string, both P and Q must also contain the empty string. Thus R* = (P;Q)* = (P + Q)*. Therefore, R* = (N(R))*.

Some examples may clarify this theorem. Given a regular expression R, we can apply the theorem recursively to produce N(R). The reader may verify the following identities:

- N(a*) = a
- N(a*;b*;c*;d*) = a + b + c + d
- N((a* + b*);(c* + d*)) = a + b + c + d

In these examples, a recognizer for N(R)* is smaller than a recognizer for R*. Although this contraction in size may not always occur, Theorem 2 states that the recognizer for N(R)* will never be larger than the one for R*.

A side benefit of the transformation from R into N(R) is that the cycle time of the recognizer for N(R)* may be smaller than that for R*. The cycle time of a recognizer is proportional to the number of combinatorial logic gates that a signal goes through between latches. A general regular expression of length n may have O(n) gates between latches: consider the expression a***· · ·* with n stars. If all subexpressions of an expression of the form R* are replaced by N(R)*, however, the maximum number of gates between two latches is attained in the tree of + cells needed for an expression of the form ($P_1 + P_2 + \ldots + P_k$). Since such a tree can be built with logarithmic depth, the transformation reduces the worst case cycle time of a recognizer of size n from O(n) to O(log n).

4. Conclusion

This paper has described the problem of latch formation in regular expression recognizers, and has suggested a new solution. By transforming the regular expression R* into N(R)*, not only may latches be avoided, but the corresponding recognizer may decrease in size. In addition, this transformation decreases the worst-case cycle time of recognizers from O(n) to O(log n). This source-to-source transformation seems to be a worthwhile addition to silicon compilers that produce regular language recognizers.

References

[1] Anantharaman, T. S.
A Delay Insensitive Regular Expression Recognizer.
1986.
Private Communication.

[2] Anantharaman, T. S., E. M. Clarke, M. J. Foster, and B. Mishra.
Compiling Path Expressions into VLSI Circuits.
In *Proceedings of the 12'th Symposium on Principles of Programming Languages.* ACM, January, 1985.
To appear in *Distributed Computing.*

[3] Anceau, F.
CAPRI: A Design Methodology and Silicon Compiler for VLSI Circuits.
In R. Bryant (editor), *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 15-32. Caltech, Pasadena, Ca., March, 1983.

[4] Backhouse, R. C.
Specification and Proof of a Regular Language Recognizer in Synchronous CCS.
Technical Report CSM-53, University of Essex, January, 1983.

[5] Balraj, T. S. and M. J. Foster.
Miss Manners: A Specialized Silicon Compiler for Synchronizers.
In *Proceedings of the Fourth MIT Conference on Advanced Research in VLSI.* MIT, April, 1986.

[6] Bergmann, N.
A Case Study of the F.I.R.S.T. Silicon Compiler.
In R. Bryant (editor), *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 413-430. Caltech, Pasadena, Ca., March, 1983.

[7] Floyd, R. W. and J. D. Ullman.
The Compilation of Regular Expressions into Integrated Circuits.
JACM 29(3):603-622, July, 1982.

[8] Foster, M. J.
Specialized Silicon Compilers for Language Recognition.
PhD thesis, Carnegie-Mellon University, 1984.

[9] Foster, M. J.
A Specialized Silicon Compiler and Programmable Chip for Language Recognition.
In N. G. Einspruch (editor), *VLSI Electronics: Microstructure Science.* Volume 14: *VLSI Design*, chapter 5, pages 139-196. Academic Press, Orlando, Florida, 1986.

[10] Foster, M. J. and H. T. Kung.
Recognize Regular Languages With Programmable Building Blocks.
Journal of Digital Systems 6(4):323-332, 1982.
A preliminary version of this paper appears in the VLSI-81 Proceedings, edited by John P. Gray.

[11] Hitchcock, C. Y. and D. E. Thomas.
A Method of Automatic Data Path Synthesis.
In *Proceedings of the 20'th Design Automation Conference.* IEEE, June, 1983.

[12] Karlin, A. R., H. W. Trickey, and J. D. Ullman.
Experience with a Regular Expression Compiler.
In *Proceedings of the International Conference on Computer Design*, pages 656-665. IEEE, October, 1983.

[13] Kowalski, T. J. and D. E. Thomas.
The VLSI Design Automation Assistant; Prototype System.
In *Proceedings of the 20'th Design Automation Conference.* IEEE, June, 1983.

[14] Lyon, R. F.
A Bit-Serial VLSI Architectural Methodology for Signal Processing.
In J. P. Gray (editor), *VLSI 81*, pages 131-140. University of Edinburgh, University of Edinburgh, August, 1981.

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

[16] Savage, J. E.
The VLSI Compilation Techniques: PLA's; Weinberger Arrays; and SLAP, a New Silicon Layout Program.
Technical Report CS-82-24, Brown University, October, 1982.

[17] Southard, J. R.
MacPitts: An Approach to Silicon Compilation.
Computer 16(12):74-82, December, 1983.