國 立 交 通 大 學

電 信 工 程 學 系 碩 士 班

碩 士 論 文

自動產生硬體描述語言實現正規表示法比對

Automatic Generation of HDL Code for Regular Expression Matching

研 究 生： 王文彬

指導教授： 李程輝 教授

中 華 民 國 九 十 四 年 六 月

自動產生硬體描述語言實現正規表示法比對

Automatic Generation of HDL Code for Regular Expression Matching

研　究　生：　王文彬	Student: Wen-Bin Wang
指導教授：　李程輝 教授	Advisor: Prof. Tsern-Huei Lee

國 立 交 通 大 學

電 信 工 程 學 系 碩 士 班

碩 士 論 文

A Thesis
Submitted to Institute of Communication Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Communication Engineering
June 2005
Hsinchu, Taiwan, Republic of China.

中 華 民 國 九 十 四 年 六 月

自動產生硬體描述語言實現正規表示法比對

研究生： 王文彬　　　　　　指導教授： 李程輝 教授

國立交通大學

電信工程學系碩士班

中文摘要

正規表示法比對是一個很重要的問題，在科學和資訊處理領域上有很多相關的應用。在這篇論文裡，我們設計一個 C 程式，這個 C 程式讀正規表示法然後輸出一個以 verilog 語言描述的 NFA。將字串輸入此 NFA 電路後，此電路可以檢查字串裡是否有符合正規表示法描述的子字串。使用我們提出的方法，硬體需求的成長將正比於正規表示法的長度。若使用 GNU grep(DFA)的方法，記憶體需求將以 $O(2^n)$ 成長（n 是正規表示法的長度）；使用[1](NFA)所提出來的方法需要 $O(n^2)$ 的硬體面積。除了面積的改善以外，在這篇論文裡介紹的新方法在偵錯、適用硬體、最佳化、電路修改、以及面積的使用效率上皆有不錯的表現。我們提出的方法利用 Pentium 4 的中央處理器和 APEX EP20K600EBC-6521X 型號的 FPGA 來評估。

Automatic Generation of HDL Code for Regular Expression Matching

Student: Wen-Bin Wang　　　　Advisor: Prof. Tsern-Huei Lee

Institute of Communication Engineering

National Chiao-Tung University

Abstract

The regular expression matching is an important problem that occurs in many areas of science and information processing. In this paper, we design a C code which accepts the regular expression and then outputs a NFA described by verilog language. By means of the NFA circuit, the circuit can read the string and examine whether the substring matches the regular expression or not. Using our approach, the area of the hardware used grow in $O(n)$, where n is the length of the regular expression. To match a regular expression, GNU grep (DFA) requires $O(2^n)$ memory and approaches using the NFA in [1] require $O(n^2)$ area. Beside the improvement of the area, this new design has not bad performance on debug, suitable device, optimization, easily modifying the circuit. We evaluate our approach on the machine which has a processor of the Pentium 4 and the target device is the APEX EP20K600EBC-6521X.

誌　　謝

感謝我的指導教授 – 李程輝教授這兩年來給予我的指導，以及在做研究上該有的方法和態度，我覺得很幸運可以跟到老師。

感謝交大資科所林盈達教授、台大資管所孫雅麗教授以及交大電信所闕河鳴助理教授，在口試時所提出的寶貴建議與指教。

很感謝網路技術實驗室的學長們：柏均、柏成、思儒、冠州、程翔和克偉，在我剛踏進實驗室一切都很陌生時，伸出援手，對我照顧和關懷。也感謝實驗室的同學們：偉志、景融、偉倫、冠亨、易霖，怡彣、孟諭、偉臣、名駿、雅婷和謹慧，在交大兩年來給予我許多歡樂、成長以及美好的回憶，謝謝你們。

最後，最感謝我的父親王金來先生以及母親周玉梅女士，從小到大所給予我無微不至的養育及照顧，無論發生任何事總是一直支持著我，給我鼓勵，替我擔憂，你們是最好的父母。

也感謝所有我的家人與朋友們。

這篇論文獻給所有我愛的人與愛我的人。

2005 年六月　于風城交大

Contents

List of Figures

List of Tables

Chapter 1

Introduction

String pattern matching is an important problem that occurs in many areas of science and information processing. In computing, it occurs as part of data processing, text editing, term rewriting, lexical analysis, and information retrieval. In biology, string-matching problems arise in the investigation of DNA sequences. The simplest form of problem is to locate an occurrence of a keyword as a substring in a sequence of characters, which we call the input string. For this problem, several innovative, theoretical, and interesting algorithms have been devised that run significantly faster than the obvious brute-force method. [8]

Pattern-matching problems can be shown in Figure 1.1 where p is the pattern and s is the input string. The pattern is transformed by the pattern-matcher generator into a pattern matcher, which is used to look for an occurrence of the pattern in the input string. The pattern matcher reports “yes” if s contains a substring matched by p, “no” otherwise. In this paper, the pattern p described in the regular expression is transformed into DFA (deterministic finite automaton) or NFA (nondeterministic finite automaton) which used as the pattern matcher to process the input string s by the pattern-matcher generator.

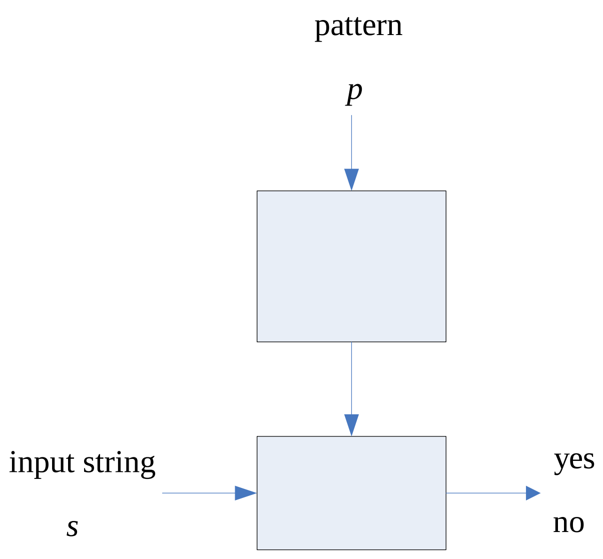


Figure 1.1 Model for pattern-matching problems [8]

About the former, its idea is to transform the regular expression into DFA and then to use DFA for string matching. We take GNU grep for example. In the worst case, the memory and time required would be exponential blowup. About the method using NFA, we take the approache in [1] for instance. The approach proposed in [1] transforms the regular expression into NFA and to implement the NFA by generating the placed and routed netlist on the FPGA. The approach requires $O(n^2)$ area and takes $O(1)$ time for processing per character (n is the length of the regular expression). Compared with the method using DFA, the approache proposed in [1] significantly reduces the space and time requirement.

The approach proposed in this paper also transforms the regular expression into NFA . This approach includes four main parts, “converting the regular expression,” “rearranging the regular expression,” “logic structures,” and “NFA construction algorithm.” Beside the two metacharacters designed in [1], we implemenent metacharacters often used and permit the metacharacter to be matche. The new method provides a general solution for regular expression matching in hardware. Its flexibility includes debug, suitable device, optimization, easily modifying the circuit, and so on. About area, the requirement is reduced from $O(n^2)$ to $O(n)$ and the utility rate becomes higher.

Chapter 2 is the background about the regular expression, finite automata and a pseudo code for converting the infix expression to a postfix expression. Chapter 3 introduces the approach in [1]. Chapter 4&5 describes approach proposed in this paper. Chapter 6 presents the evaluation result and comparison. Chapter 7 is the conclusion.

Chapter 2

Background

2.1 Finite Automata [2]

Before introducing the automaton, we first introduce some important definitions. These concepts include the “alphabet”, “strings” and “language.” An alphabet is a finite, nonempty set of symbols. Common alphabets include the binary alphabet, the set of all lower-case letters and the set of all ASCII characters. Usually, we use the symbol Σ for an alphabet. A string is a finite sequence of symbols chosen from some alphabet. For example, “kitty” is the string that is chosen from the set of all low-case letters. Σ^k is the set of strings of length k and each of whose symbols is in Σ. For example, if $\Sigma = \{0,1\}$, then $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$. A set of strings which are chosen from Σ^*, where Σ^* is the set of all strings over an alphabet Σ, is called a language. For example, Chinese name is a set of strings over the alphabets which are all Chinese words.

The finite automaton is a useful model for many important applications. Before we introduce precise definitions of automata of various types, let us informally introduce the sketch of what a finite automaton is. There are many systems or components in one of a finite number of “states.” The purpose of a state is to remember the relevant portion of the system's history. The advantage of having a finite number of states is to implement the system with finite resources. Let us take a finite automaton (Figure 2.1) for example. A finite automaton has a set of states and its "control" moves from state to state in response to external "inputs." For finite automata, circles represent states and arcs between states are labeled by "inputs," which represent external influences on the system. One of the states is designed the "start state," the state in which the system is placed initially. One or more states are designed "final states" and it is conventional to designate final states by a double circle. In Figure 2.1, state 1 is "start state" and state 5 is "final state." Entering the final state represents that the input sequence is good in some way.

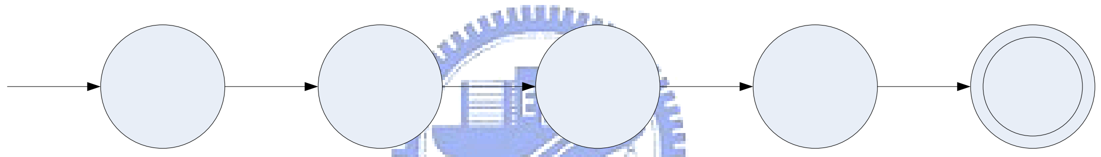


Figure 2.1: finite automaton

One of the crucial distinctions among classes of finite automaton is the number of states once. "Nondeterministic Finite Automaton" means that the automaton may be in several states at any one time but "Deterministic Finite Automaton" means that it cannot be in more than one state at once. Below we introduce the definitions of "Nondeterministic Finite Automaton" and "Deterministic Finite Automaton" for details.

2.1.1 Nondeterministic Finite Automaton (NFA)

The definition of the nondeterministic finite automaton is below.

Start $A = (Q, \Sigma, \delta, q_0, F)$ a b

1 2

where:

1. Q is a finite set of states.
2. $\sum$ is a finite set of input symbols.
3. q_0, a member of Q, is the start state.
4. F, a subset of Q, is the set of final states.
5. δ, the transition function is a function that takes a state in Q and an input symbol in $\sum$ as arguments and returns a subset of Q. If q is a state, and a is an input symbol, then $\delta(q,a)$ is that subset $\{p_1, p_2, \ldots\}$ such that there are arcs labeled a from q to subset $\{p_1, p_2, \ldots\}$.

A "nondeterministic" finite automaton (NFA) has several states at once. Take an example, the automaton (Figure 2.2) is used to recognize occurrences of the words, *web* and *ebay*, in a text. The NFA of Figure 2.2 can be specified formally as $(\{1,2,3,4,5,6,7,8\},\{\text{all ASCII characters}\},\delta,1,\{4,8\})$. The transition table corresponding to the function δ of Figure 2.2 is shown in Table 2.1. The input and state are designated respectively in the first row and column of the Table 2.2. State 2 through 4 have the job of recognizing *web*, while state 5 through 8 recognize *ebay*.

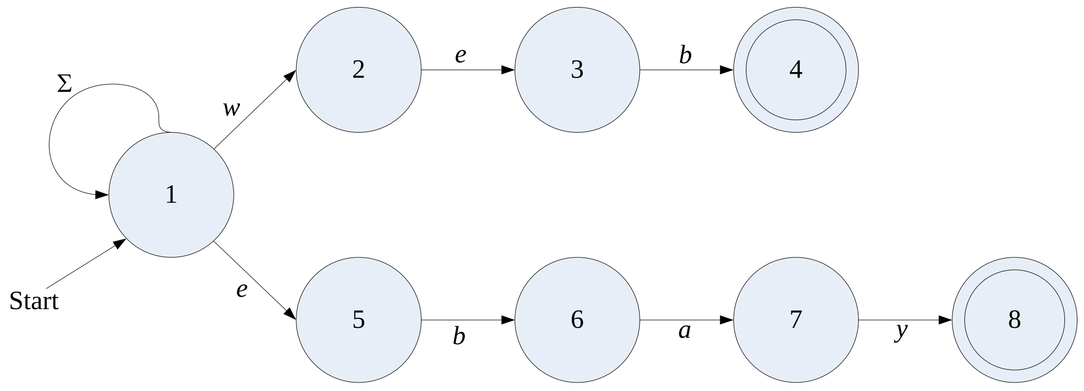


Figure 2.2: the NFA diagram of recognizing *web* and *ebay*

	w	*e*	*b*	*a*	*y*	*other ASCII characters*
1	{1, 2}	{1, 5}	{1}	{1, 5}	{1}	{1}
2	ϕ	{3}	ϕ	ϕ	ϕ	ϕ
3	ϕ	ϕ	{4}	ϕ	ϕ	ϕ
4	{4}	{4}	{4}	{4}	{4}	{4}
5	ϕ	ϕ	{6}	ϕ	ϕ	ϕ
6	ϕ	ϕ	ϕ	{7}	ϕ	ϕ
7	ϕ	ϕ	ϕ	ϕ	{8}	ϕ
8	{8}	{8}	{8}	{8}	{8}	{8}

Table 2.1: the transition table corresponding to the function δ of Figure 2.2

2.1.2 Deterministic Finite Automaton (DFA)

The definition of the deterministic finite automaton is below.

$$A = (Q, \Sigma, \delta, q_0, F)$$

where:

1. Q is a finite set of states.
2. Σ is a finite set of input symbols.
3. q_0, a member of Q, is the start state.
4. F, a subset of Q, is the set of final states.
5. δ, the transition function is a function that takes a state in Q and an input symbol in Σ as arguments and returns a state of Q. If q is a state, and a is an input symbol, then $\delta(q, a)$ is that state p such that there is an arc labeled a from q to p. Notice that the difference between an NFA and

DFA is the type of value that δ returns: a set of states in the case of an NFA and a single state in the case of a DFA.

Figure 2.3 is a simple instance for the DFA which can be specified formally as $(\{1,2,3,4\},\{0,1\},\delta,1,4)$. The transition table corresponding to the function δ of Figure 2.3 is shown in Table 2.2. The input and state are designated respectively in the first row and column of the Table 2.3.

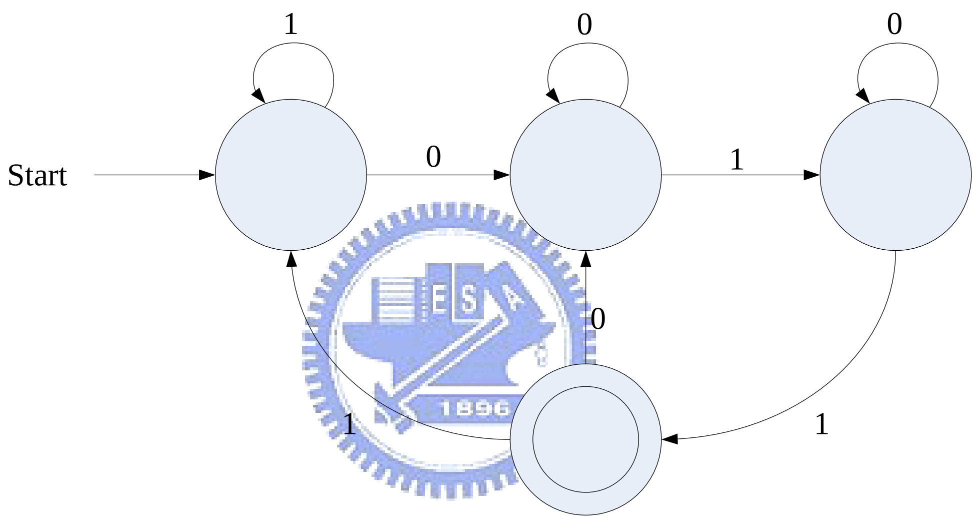


Figure 2.3: the DFA diagram

	0	1
1	2	1
2	2	3
3	3	4
4	2	1

Table 2.2: the transition table corresponding to the function δ of Figure 2.3

1

2.2 Regular Expression [8]

Here we switch our attention from machine-like descriptions of languages, NFA and DFA, to an algebraic description: the “regular expression.” The regular expression can be thought of as a “user-friendly” alternative to the automata notation. Besides, regular expressions offer something that automata don’t: a declarative way to express the strings we want to accept. Therefore, regular expressions serve as the input language for many systems that process strings, e.g. UNIX grep command, UNIX Lex(Lexical analyzer generator) and Flex(Fast Lex) tools.

We define regular expressions and the strings they match as follows:

1. The following characters are metacharacters: | ( ) *
2. A non-metacharacter a is a regular expression that matches the string a.
3. If r_1 and r_2 are regular expressions, then $(r_1 | r_2)$ is a regular expression that matches any string matched by either r_1 or r_2.
4. If r_1 and r_2 are regular expressions, then $(r_1)(r_2)$ is a regular expression that matches any string of the form xy, where r_1 matches x and r_2 matches y.
5. If r is a regular expression, then $(r)*$ is a regular expression that matches any string of the form $x_1x_2...x_n, n \geq 0$, where r matches x_i for $1 \leq i \leq n$. In practice, $(r)*$ matches the empty string, which we denote by ε.
6. If r is a regular expression, then (r) is a regular expression that matches the same strings as r.

Here are two examples for understanding the regular expression. The regular expression $The\ (dog \,|\, cat)\ likes\ (studying \,|\, sleeping)$ matches any of $\{The\ dog\ likes\ studying,\ The\ dog\ likes\ sleeping,\ The\ cat\ likes\ studying,\ The\ cat\ likes\ sleeping\}$

. The regular expression $Thank\ you\ very(,\ very)*\ much$ matches the strings $Thank\ you\ very\ much$; $Thank\ you\ very,\ very\ much$; $Thank\ you\ very,\ very,\ very\ much$; and so on.

2.3 Constructing the NFA from the Regular Expression [8]

A NFA is a directed graph in which the nodes are the states and each edge is labeled by a single character or the symbol ε, which stands for the empty string. One state is designated as a start state, and some states as final states. A NFA accepts a string if there is a path from the start state to a final state whose edge labels spell out the string. Once we have constructed a NFA for a regular expression r, we run it on the input string s. If the NFA enters a final state while processing s, we report that r matched s, otherwise, we report "no". The recursive procedure below can be used to construct an NFA for the regular expression. Using rule (1), we construct a NFA for a non-metacharacter. Rule (2)-(5) show how to combine the NFAs constructed from the constituent subexpressions.

(1) For a non-metacharacter c, construct the NFA in Figure 2.4(a) where i is a new start state and a a new final state. This automaton accepts the string c.

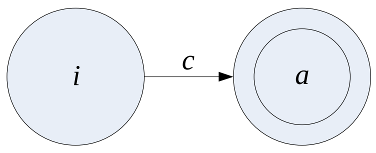
(2) Suppose N_{r_1} and N_{r_2} are NFA for r_1 and r_2. For the regular expression $r = r_1 | r_2$, construct the NFA N in Figure 2.4(b) where i is a new start state and a a new final state. There is an ε-transition from i to the start states of N_{r_1} and N_{r_2}. There is an ε-transition from the final states of N_{r_1} and N_{r_2} to the new final state a. That is, any path from i to a must pass through either N_{r_1} or N_{r_2}. Therefore, N accepts any string accepted by N_{r_1} or N_{r_2}.

(3) Suppose N_{r_1} and N_{r_2} are NFAs for r_1 and r_2. For the regular expression $r = r_1 r_2$, construct the NFA N in Figure 2.4(c) where the start state of N_{r_1} becomes the start state of N and the final state of N_{r_2} becomes the final state of N. The final state of N_{r_1} is merged with the start state of N_{r_2}; that is, all transitions from the start state of N_{r_2} become transitions from the final state of N_{r_1}. The new merged state loses its status as a start or final state in N. A path from i_{r_1} to a_{r_2} must go first through N_{r_1} and then through N_{r_2}, so N accepts any string of the form xy where N_{r_1} accepts x and N_{r_2} accepts y.

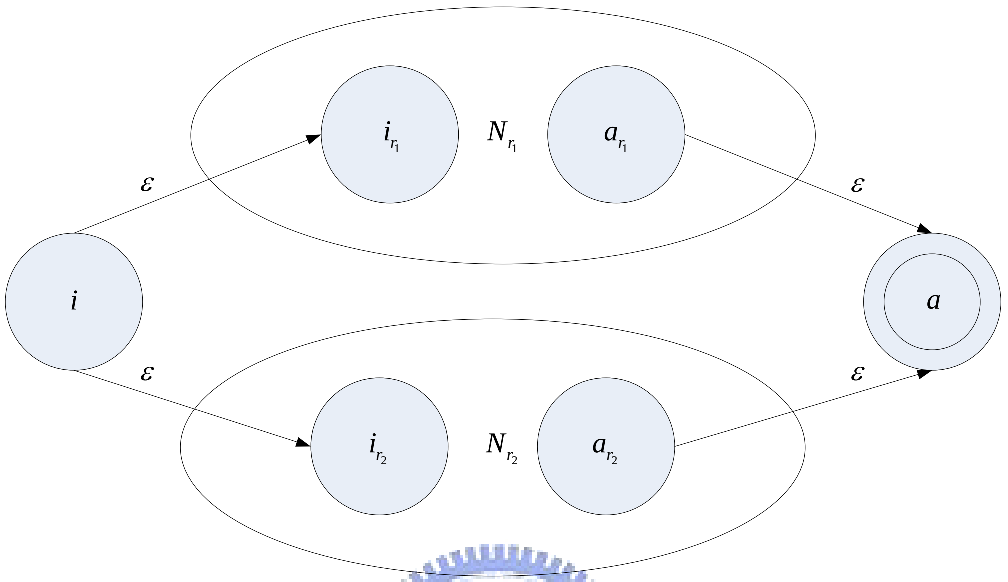
(4) Suppose N_{r_1} is a NFA for r_1. For the regular expression $r = r_1*$, construct the NFA in Figure 2.4(d) where i is a new start state and a a new final state. In N, we can go from i to a directly, along an edge labeled ε, representing the fact that $s*$ matches the empty string, or we can go from i to a passing through N_{r_1} one or more times. Thus, N accepts any string matched by r_1*.

(5) For the regular expression (r) use the NFA for r.

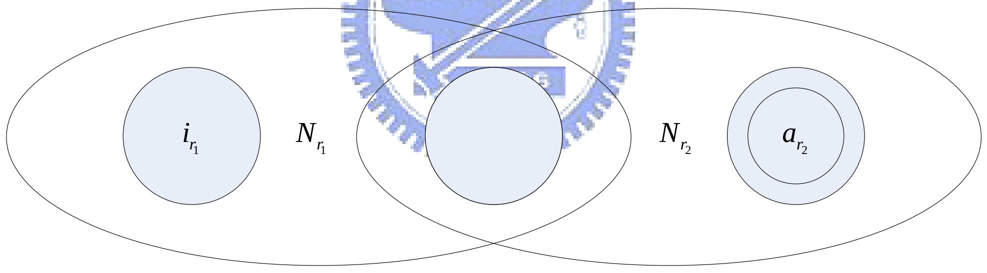
Take a simple instance for understanding the procedure of constructing the NFA from the regular expression. Figure 2.5 shows a NFA that results from this construction for the regular expression $(a|b)*cd$.



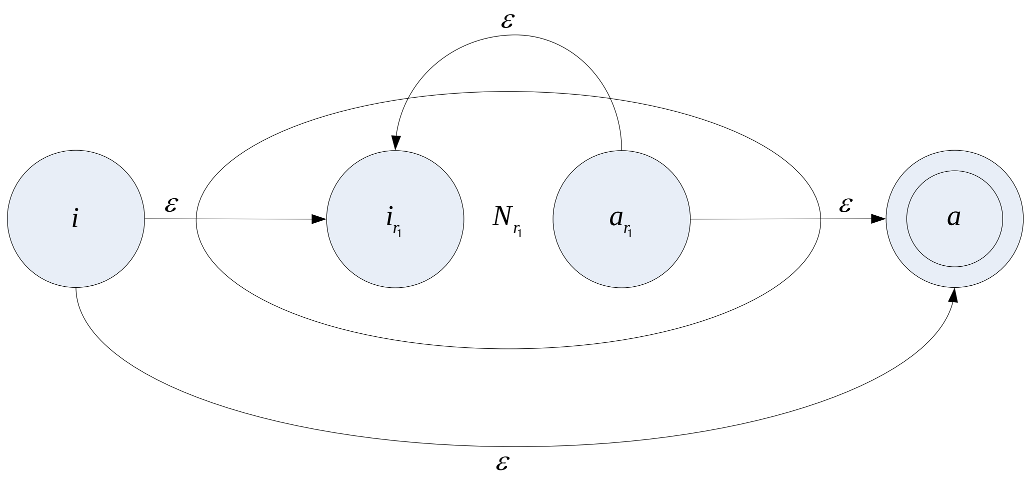
(a)



(b)



(c)



(d)

Figure 2.4: constructing NFA from the regular expression

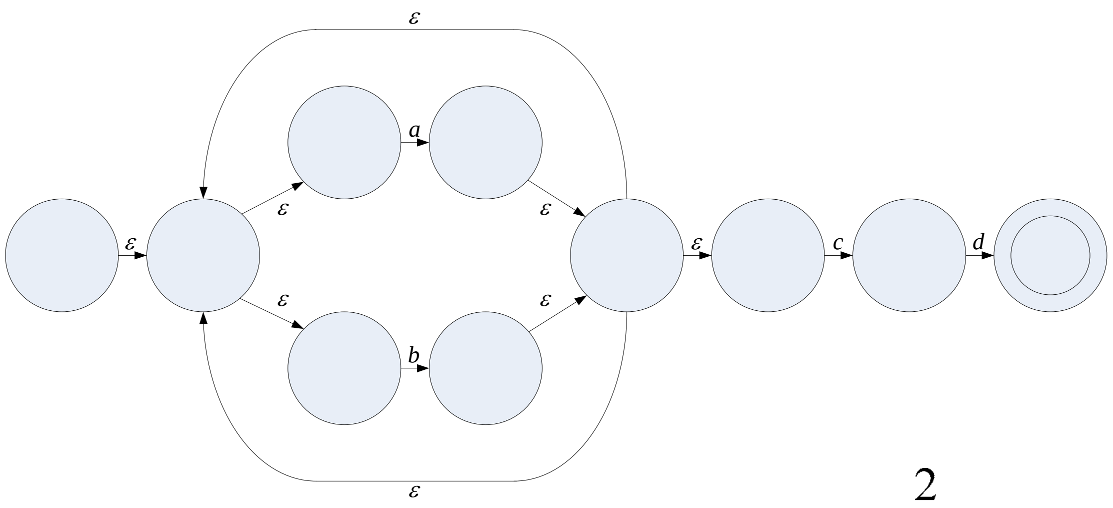


Figure 2.5: NFA for $(a|b)^*cd$

2.4 Extensions to the Regular Expression Notation [8] [9] [10] [11]

Many text-editing and searching programs add abbreviations and new operators to the basic regular expression notation above to make it easier to specify patterns. Here, we introduce some of the regular expressions used by the popular expression matching programs awk, grep, egrep and lex on the UNIX.

(1) The $+$ metacharacter: The character $+$ means "one or more of the preceding characters." For example, the regular expression $de+f$ matches any of the following: $redefine, redeefine, redeeefine...$, and so on.

(2) The [ ] metacharacters: The [ ] metacharacters enable us to define regular expressions that match one of a group of alternatives. For example, the following regular expression matches def or dEf : $d[eE]f$. When the ^ character appears as the first character after the [, it indicates that the regular expression is to match any character except the ones displayed between [and]. For

example, the regular expression $d[\text{^}eE]f$ matches any pattern that satisfies the following criteria: 1. The first character is d. 2. The second character is anything other than e or E. 3. The last character is f.

(3) The ? metacharacter: ? metacharacter matches zero or one occurrences of the preceding character. For example, the regular expression $de?f$ matches either df or def.

(4) Escape sequences for metacharacters: If we want our regular expression to include a character that is normally treated as a metacharacter, precede the character with a backslash $\backslash$. For example, to check for one or more occurrences of * in a string, use the following regular expression: $\backslash*+$. The backslash preceding the * tells us to treat the * as an ordinary character, not as the metacharacter meaning" zero or more occurrences." To include a backslash in a regular expression, specify two backslashes $\backslash\backslash+$. This regular expression tests for one or more occurrences of $\backslash$ in a string.

(5) Matching any letter or number: The regular expression $a[0123456789]c$ matches a, followed by the any digits, followed by c. Another way of writing this is as follows: $a[0-9]c$. Here, the range $[0-9]$ represents any digit between 0 and 9. This regular expression matches $a0c, a1c, a2c,$ and so on up to $a9c$. Similarly, the range $[a-z]$ matches any lowercase letter, and the range $[A-Z]$ matches any uppercase letter. For example, the regular expression $[A-Z][A-Z]$ matches any two uppercase letters.

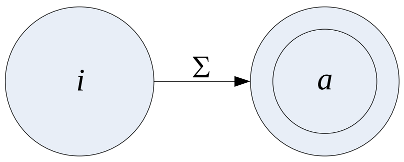
(6) Anchoring patterns: The regular expression ^ and \$ ensure that the regular expression is only matched at the start or the end of the string. For example, the regular expression $\text{^}def$ matches def only if these are the first three characters in the strings. Similarly, the regular expression $def\$$ matches def only if these are the last three characters in the strings. We can combine ^

and $\$$ force matching of the entire string, as follows: $\hat{}def\$$. This matches only if the string is def .

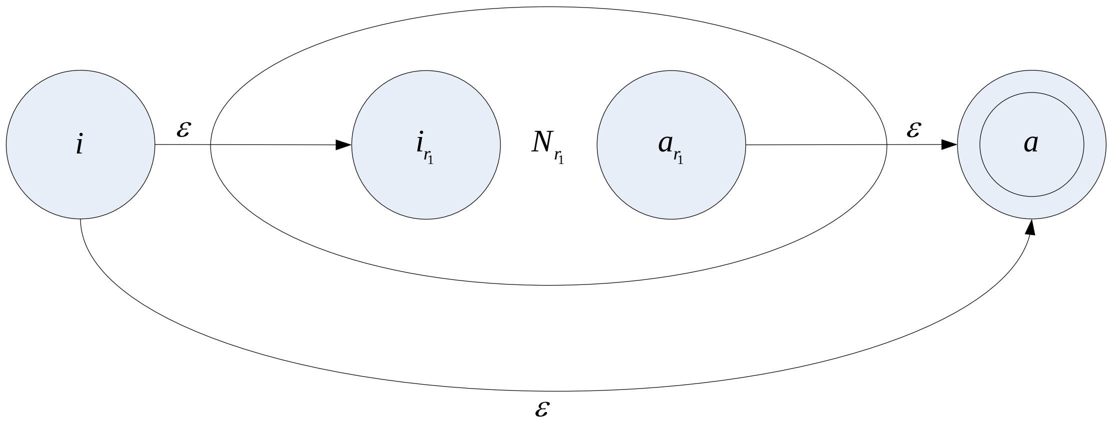
(7) Matching any character: Another metacharacter supported in the regular expression is the period (.) character, which matches any character. For example, the regular expression $d.f$ matches d , followed by any character, followed by f .

(8) Matching a specified number of occurrences: We can define how many occurrences of a character constitute a match. To do this, use metacharacters $\{$ and $\}$. For example, the regular expression $de\{1,3\}f$ matches d , followed by one, two, or three occurrences of e , followed by f . This means that def , $deef$, and $deeef$ match, but df and $deeeef$ do not. To specify an exact number of occurrences, include only one value between the $\{$ and $\}$: $de\{3\}f$. This specifies exactly three occurrences of e , which means this regular expression inly matches $deeef$.

Figure 2.6 shows two NFAs for . and r_1 ?



(a)



(*b*)

Figure 2.6: (a) NFA for . (b) NFA for r_1?

2.5 Converting an Infix Expression to a Postfix Expression [3]

Figure 2.7 is a pseudo code which converts an infix expression to a postfix expression. For the conversion algorithm to be correct, we must check four issues. 1. The postfix expression contains the correct operands in the correct order. 2. The postfix expression evaluates subexpressions in the way indicated by the parentheses in the infix expression. 3. The postfix expression handles operations of differing precedence according to the precedence rules. 4. A string of operations of equal precedence in the infix expression is handled correctly when translated into the postfix expression. Below, we consider each of these four issues.

First we need to know that the operands (the numbers and variables) in the postfix expression are in the same order as they are in the infix expression. Because operands are written out as soon as they are read in, they are clearly in the same order as in the infix expression.

Parentheses are a way of grouping subexpressions. Everything inside a pair of matching parentheses is treated as a single unit by anything outside the parentheses. The parentheses give the following message to the operations outside of the parentheses: We will work things out among ourselves and deliver a single value for you to combine with other operands. This means that all operations between a set of matching parentheses in the infix expression should form a subexpression of the postfix expression. The algorithm keeps track of expressions with matching parentheses by using the stack. When algorithms encounters an opening parenthesis, that is, a '(', it pushes this parenthesis into the stack. The algorithm will never output an operation from the stack that is below the opening parenthesis, '('. It only outputs operations that are within the pair of matching parentheses in the input expression. Moreover, it outputs all of these operations. When it encounters the matching closing parenthesis, it outputs all the remaining operations on the stack all the way down to that matching opening parenthesis. This behavior can be completed by means of the prcd function which executes the comparison of the precedence. The function has two arguments. When the former argument owns the higher precedence than the later, the function returns 1. The function returns 0 when the opening parenthesis is the argument and the closing parenthesis is not. By returning 0, the stack can stop to pop operations outside a pair of matching parentheses. When the closing parenthesis is the argument, the function will return 1 until encountering the opening parenthesis. It will make all operations between a set of matching parentheses in the infix expression form a subexpression of the postfix expression.

When the infix expression contains an operation with low precedence followed by an operation with a higher precedence, then the algorithm should output these operations in reverse order. That is, the higher precedence should be written first. By means of the function prcd, this work also is completed. The former argument of the function is the top of the stack and the latter is the new coming operation. When the precedence of the former is higher than the precedence of the latter, it represents that an operation with higher precedence followed by an operation with a lower precedence in the infix expression. Therefore, the stack is popped. Otherwise, the order of the operations is reversed.

When the infix expression contains a sequence of operations of equal precedence, they represent an evaluation that goes from left to right. The work is solved by returning 1 when the function accepts the same two operations as arguments.

```
opstak = the empty stack;
while ( not end of input )
{
        symb = next input character;

        if ( symb is an operand )
                add symb to the postfix string;
        else
        {
                while ( !empty(opstk) && prcd(stacktop(opstk) , symb))
                {
                        topsymb = pop(opstk);
                        add topsymb to the postfix string;
                } // end while

                // push ( opstk, symb );
                if (empty(opstk) || symb != ')' )
                        push (opstk , symb);
                else
                        topsymb = pop(opstk);
        } // end else
} // end while

while (!empty(opstk))
{
        topsymb = pop(opstk);
        add topsymb to the postfix string;
} // end while
```

Figure 2.7: converting an infix expression to a postfix expression

Chapter 3

Related Work: Generating the Placed and Routed Netlist [1]

This approach was introduced in [1] for finding matches to a given regular expression in a given text. [1] presents an algorithm that constructs the Nondeterministic Finite Automaton (NFA) circuit used for matches. The idea is to map simple NFAs onto logic structures and then to complete the whole circuit by means of these logic structures.

Before introducing the algorithm, we first describe simple logic structures shown in Figure 3.1, a single character, $r_1 | r_2$, $r_1 r_2$ and $r_1 *$. Logic structures showed in Figure 3.1(a), (b), (c) and (d) are the implementations of the NFAs shown in Figure 2.4(a), (b), (c) and (d) respectively. In Figure 3.1(a), the output is 1 only when the flip-flop stores a 1 and the input character matches the character stored in the in the comparator. In Figure 3.1(b), only when N_1 or N_2 has a match, the output of the OR gate would be high. In Figure 3.1(c), only when N_1 and N_2 have matches, the o would be high. In Figure 3.1(d), the o would be high whether N_1 has match or not. The behavior of Figure 3.1 (a), (b), (c) and (d) is the same as Figure 2.4(a), (b), (c) and (d), respectively.

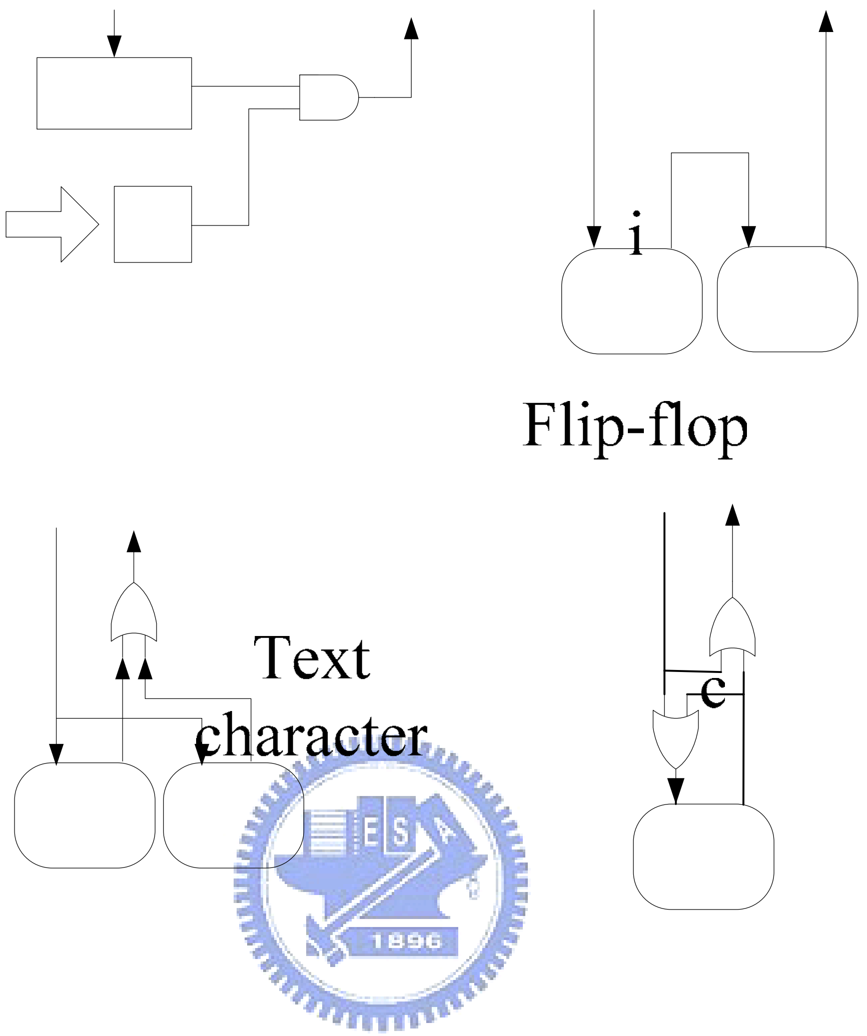


Figure 3.1: logic structures (a) single character (b) $r_1 | r_2$ (c) $r_1 r_2$ (d) $r_1 *$

Here, we introduce the algorithm. The algorithm accepts the regular expression in postfix form obtained by postorder traversal of the syntax tree of the regular expression and directly generates the placed and routed netlist for NFA logic. The order of characters in the postfix form replaces the use of parentheses in the infix form. By eliminating parentheses, the algorithm needs not to deal with parentheses, so it is simplified. The algorithm places the NFA logic as a binary tree. That is, logic structures for characters are leaf nodes and placed in the row 0. Once the logic structure for the character is placed, the column counter is incremented. Logic structures for metacharacters are non-leaf nodes and are placed on separate rows.

Once the logic structure for the metacharacter is placed, the row counter is incremented.

By exploiting this algorithm, the area would depend on placement and routing subroutines. According to [1], the area would be in $O(n^2)$ where n is the length of the regular expression. This algorithm compared with a serial machine is better because the DFA take $O(2^n)$ memory.

Chapter 4

Generating HDL Code

This paper proposes a new method to construct the NFA circuit used for matches. The method proposed in [1] only implements two metacharacters and the metacharacter can't be view as a symbol for matching. Here, this new method implements metacharacters often used and permits the metacharacter to be matched. Figure 4.1 is the flow chart of the new method for constructing NFA circuit. This method includes four main parts, "converting the regular expression," "rearranging the regular expression," "logic structures," and "NFA construction algorithm." In order to generate the HDL code from the C language and conform the syntax of the HDL, these four parts are needed. "Reducing and converting the regular expression" reduces the redundant part of the regular expression and converts some metacharacters into the equivalent regular expression. "Rearranging the regular expression" rearranges the order for simplifying the NFA algorithm. "Logic structures" can be viewed as bricks for complete the circuit. "NFA construction algorithm" accepts the output of "rearranging the regular expression" and completes the circuit. We introduce them below for details.

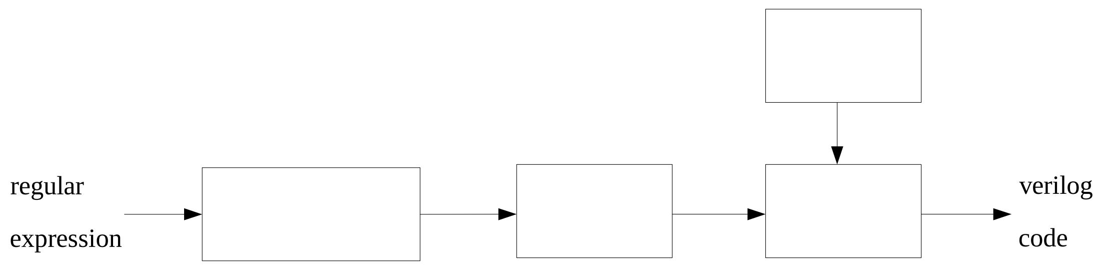


Figure 4.1: the flow chart of the constructing NFA circuit

4.1 Reducing and Converting the Regular Expression

Reducing and converting the regular expression

In the beginning, we first examine whether the regular expression in the infix form can be reduced or not. (regular expression b)*(regular expression a) and (regular expression a)(regular expression b)* can be both reduced to (regular expression a). For the former, since the NFA must match strings beginning at any position in the input text, (regular expression b)* is redundant. For the latter, since the NFA can't indicate which pattern matched, thus (regular expression b)* can be removed. Some metacharacters can be implemented only by converting the regular expression into the other regular expression. We introduce and explain it below.

If r is a regular expression, then $r+$ is a regular expression that matches any string of the form $x_1x_2...x_n, n \geq 1$, where r matches x_i for $1 \leq i \leq n$. Therefore, the regular expression $r+$ could be implemented by converting it into $rr*$. However, when encountering $/+$, we don't convert it into $//*$ since $/+$ means treating $+$ as a character to match.

$r\{n\}$ is a regular expression that matches any string of the form $x_1x_2...x_n$ where r matches x_i for $1 \leq i \leq n$. Therefore, the regular expression $r\{n\}$ could be implemented by converting it into $rrr...r$ where n regular expressions are connected.

The regular expression $[r_1r_2...r_n]$ is just the shorthand of the regular expression $(r_1 | r_2 | ... | r_n)$. Therefore, for the implementation, we convert the shorthand. Similarly, regular expressions $[A-K]$ and $[1-9]$ are also converted. When encountering $[\wedge r_1r_2...r_n]$, we convert it into $(r_1 | r_2 | ... | r_n)+$ where the meaning of $+$ has been changed. After converting the regular expression, there is a $/$ in the front of $+$. By means of this feature, we use $+$ which follows $)$ to solve metacharacters $[\wedge...]$.

4.2 Rearranging the Regular Expression

Here, we eliminate the parentheses and rearrange the order of metacharacters which are $|$ and $\wedge$. The position of the metacharacter $\wedge$ is rearranged since the logic structure which implements $\wedge$ needs to concatenated in the back of the first comparator for checking whether the input character is the beginning of the string or not. The position of the metacharacter $|$ is rearranged in order that the NFA construction algorithm can build $|$ on the top of the two suitable comparators and needn't to deal with parentheses. By eliminating parentheses, the NFA construction algorithm can be simplified.

We use the pseudo code shown in the Figure 2.7. In the function prcd, there are only $|, (, \text{and})$ without $+, -, \times, \text{and} \div$. After rearranging the order of the expression, exchange positions of the metacharacter $\wedge$ and the first character.

4.3 Logic Structures

Metacharacters mentioned in the Chapter 2 can be implemented by building some logic structures which established in the form of modules described by the HDL. We choose and design eight logic structures as basic components of the whole circuit. Figure 4.2 shows these modules. Note that the wire marked the odd number represents the output. On the contrary, the even number indicates the input. The arrangement of numbers is for the generation of the HDL code.

Figure (a) is a comparator which is the implementation of the Figure 2.4(a). The flip-flop can be treated as the state of the NFA and we use the clock to segment the coming symbol of the string. If the flip-flop stores a 1, it represents that the operation of the "regular expression matching" walks to this state which the comparator stands for. If the input symbol matches, the comparator delivers a 1 to the next comparator which stands for the next state.

Figure 4.2(b), (c), (d), (e) and (f) are the implementations of Figure 2.4(b)(c)(d) and Figure 2.5(b)(a), respectively. Figure 4.2(b) implements the regular expression $r_1 | r_2$ where the circuit implementing r_1 connects w3&4 and the circuit implementing r_2 connects w5&6. Any one of circuits implementing r_1 and r_2 output a 1 and then the circuit implementing $r_1 | r_2$ outputs a 1. Figure 4.2(c), (d), (e) and (f) are similar. Figure (c) is used for the regular expression $r_1 r_2$. Figure (d) is used for the regular expression $r_1 *$. Figure (e) is used for the regular expression $r_1 ?$. Figure (f) is used for the regular expression $r_1 . r_2$.

Figure 4.2 (g) is designed for solving metacharacters $\wedge$ and $\$$. Because behaviors of metacharacters $\wedge$ and $\$$ can't be drawn in the NFA, we design a signal "sf" for examining whether the input symbol is the beginning or end of the string or not. When the work for the "regular expression matching" starts or finishes, signals "s" or "f" are designed to pull up for a clock. Take $\wedge ben$ as an example, $\wedge ben$

would be rearranged into $b \wedge en$. The two inputs of Figure 4.2(g) would be the output of the comparator for b and the signal “s.” Only when the first symbol of the input string is b, the regular expression has the possibility of matching. Figure4.2 (h) is designed for solving metacharacters $[\wedge ...]$. After converting the regular expression, the $[\wedge ...]$ becomes $(...)+$. When reading the $+$, we connect the circuit implementing $(...)$ and the inverter shown in the Figure 4.2(h).

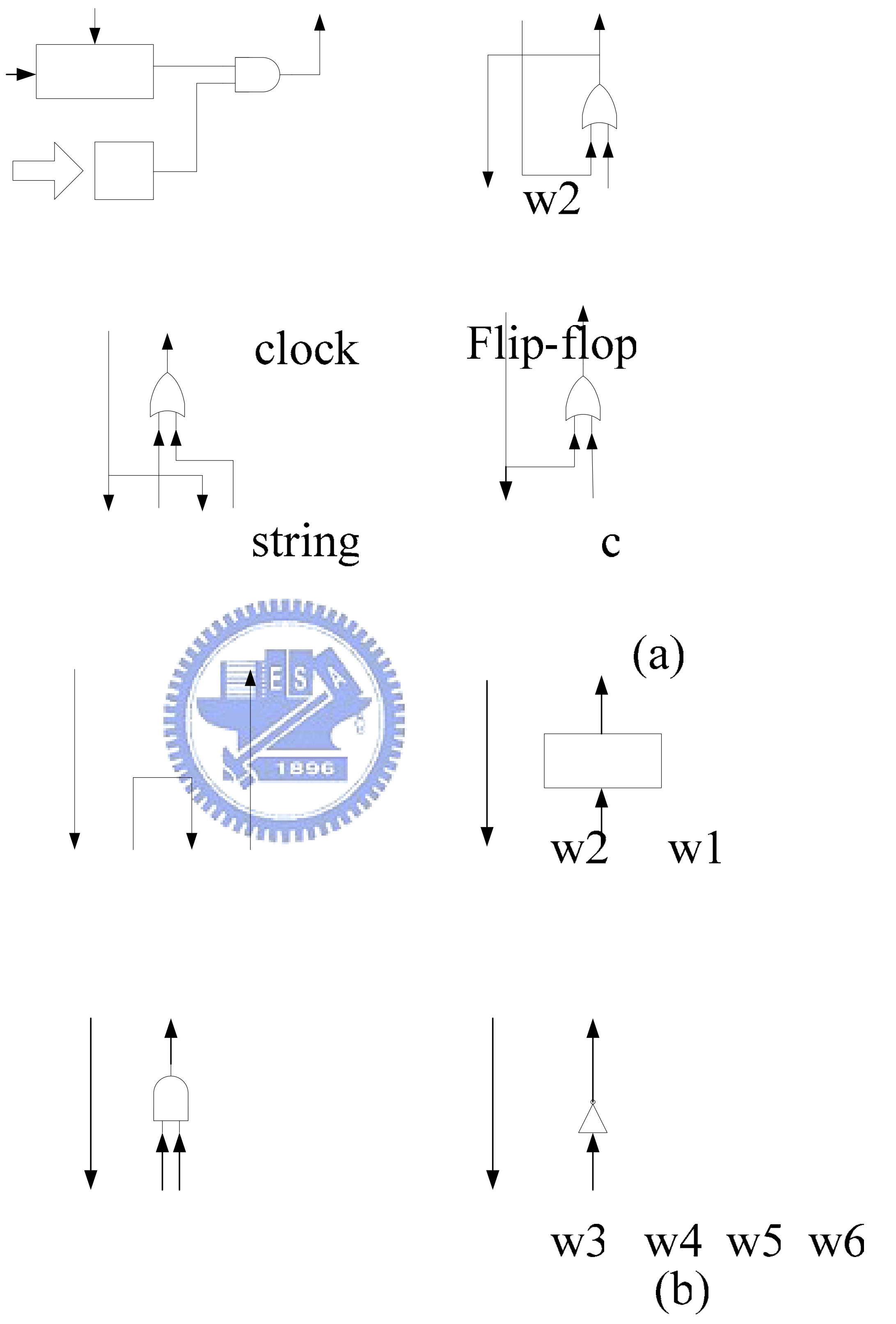


Figure 4.2: logic structures for (a) comparator (b) union | (c) concatenation (d) closure * (e) ? (f) dot . (g) anchor (h) +

w2 w1

4.4 NFA Construction Algorithm

Figure 4.3 shows the NFA construction algorithm and subroutines are shown in the Figure 4.4. To give different names for all modules and wires, it is necessary to compute how many logic structures are being used. Thus the counter, number, is used. After accepting the regular expression in the postfix form, the algorithm would check whether the regular expression finishes or not. If not, it examines the input symbol, and then processes the particular subroutine. Until the regular expression finishes, the algorithm processes the other recursion: examining whether the number which the stack pops is the last one or not. If not, the subroutine, routing for concatenation, works for concatenation, otherwise, the subroutine, routing for i/o port, works.

Figure 4.4(a) is used only when the symbol which has to be matched is accepted and this logic structure dominates how fast the circuit can be. In Figure 4.4(a), after naming wires and the comparator, ports of the comparator are assigned and then push the number of the comparator into the stack. Next, for the next logic structure used the number must be incremented. Figure 4.4(b), (c), (d), (e), (f) and (g) are similar, so we only take Figure 4.4(b) to explain. Since the union of the regular expression in the postfix form applies to the former two, the algorithm pops the stack two times. After names of wires and the logic structure of the union are declared, the algorithm completes the connection between wires by exploiting the “assign”. Figure 4.4(i) is about the i/o port. The i port is stuck to the power supply and the o port is just the wire.

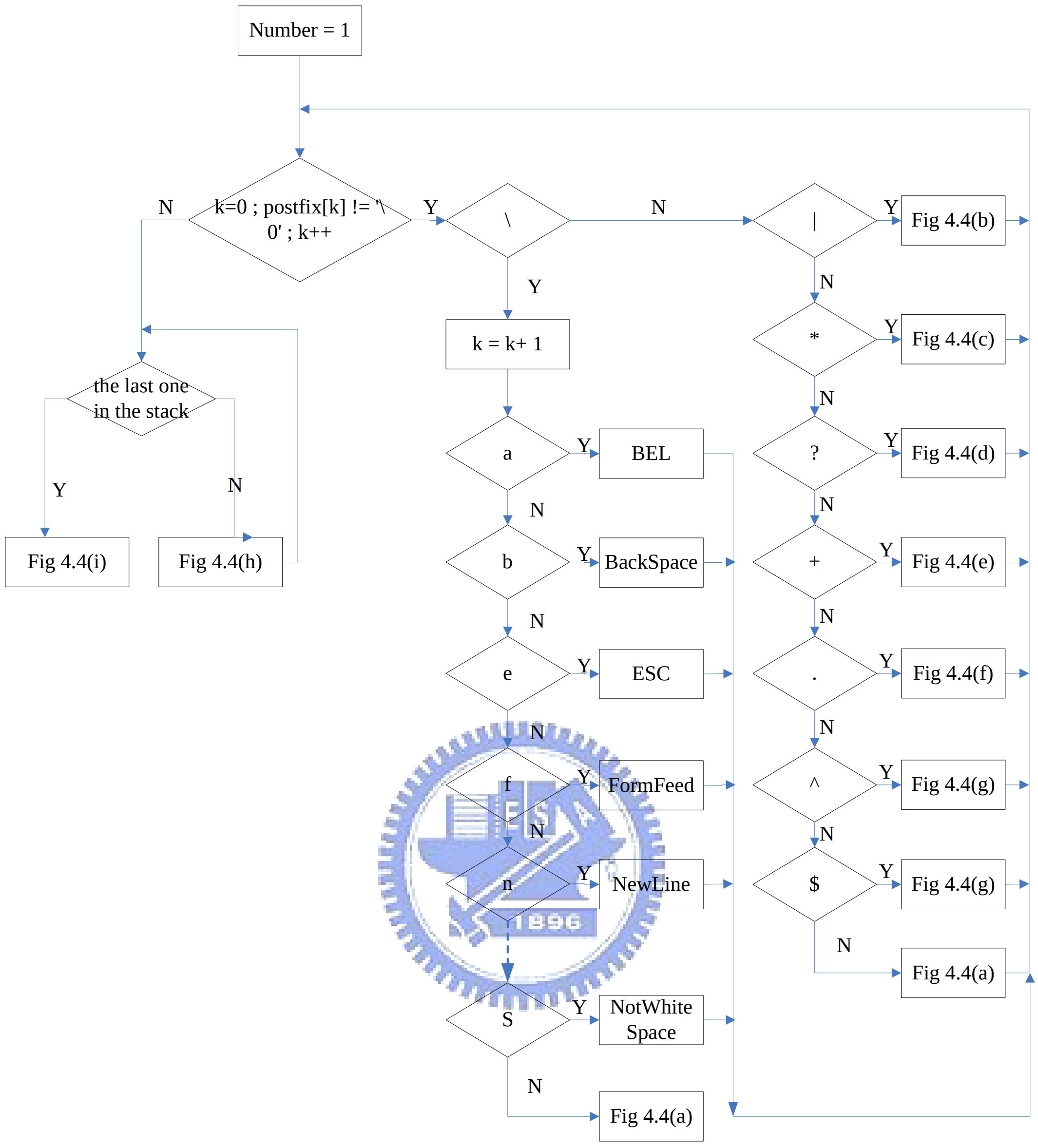


Figure 4.3: the NFA construction algorithm

```
printf("wire w%d_1, w%d_2 ;\n", number, number) ;
printf("comparator_%c comparator%d( w%d_1, w%d_2, clk, data ) ;\n\n",
postfix[k], number, number, number ) ;
push(s2, number) ;
number++ ;
```

(a)

```
number1 = pop(s2) ;
number2 = pop(s2) ;
printf("wire w%d_1, w%d_2, w%d_3, w%d_4, w%d_5, w%d_6 ;\n", number, number, number, number, number,
number) ;
printf("union union%d( w%d_1, w%d_2, w%d_3, w%d_4, w%d_5, w%d_6 ) ;\n", number, number, number, number,
number, number, number ) ;
printf("assign w%d_2 = w%d_3 ;\n", number2, number ) ;
printf("assign w%d_2 = w%d_5 ;\n", number1, number ) ;
printf("assign w%d_4 = w%d_1 ;\n", number, number2 ) ;
printf("assign w%d_6 = w%d_1 ;\n\n", number, number1 ) ;
push(s2, number) ;
number++ ;
```

(b)

```
number1 = pop(s2) ;
printf("wire w%d_1, w%d_2, w%d_3, w%d_4 ;\n", number, number, number, number) ;
printf("closure closure%d( w%d_1, w%d_2, w%d_3, w%d_4 ) ;\n",  number, number, number, number, number ) ;
printf("assign w%d_4 = w%d_1 ;\n", number, number1 ) ;
printf("assign w%d_2 = w%d_3 ;\n\n", number1, number ) ;
push(s2, number) ;
number++ ;
```

(c)

```
number1 = pop(s2) ;
printf("wire w%d_1, w%d_2, w%d_3, w%d_4 ;\n", number, number, number, number) ;
printf("repetition repetition%d( w%d_1, w%d_2, w%d_3, w%d_4 ) ;\n",  number, number, number, number, number ) ;
printf("assign w%d_4 = w%d_1 ;\n", number, number1 ) ;
printf("assign w%d_2 = w%d_3 ;\n\n", number1, number ) ;
push(s2, number) ;
number++ ;
```

(d)

```
number1 = pop(s2) ;
printf("wire w%d_1, w%d_2, w%d_3, w%d_4 ;\n", number, number, number, number) ;
printf("exclusion exclusion%d( w%d_1, w%d_2, w%d_3, w%d_4 ) ;\n",  number, number, number, number, number ) ;
printf("assign w%d_4 = w%d_1 ;\n", number, number1 ) ;
printf("assign w%d_2 = w%d_3 ;\n\n", number1, number ) ;
push(s2, number) ;
number++ ;
```

(e)

```
number1 = pop(s2) ;
printf("wire w%d_1, w%d_2, w%d_3, w%d_4 ;\n", number, number, number, number) ;
printf("dot dot%d( w%d_1, w%d_2, w%d_3, w%d_4 ) ;\n",  number, number, number, number, number ) ;
printf("assign w%d_4 = w%d_1 ;\n", number, number1 ) ;
printf("assign w%d_2 = w%d_3 ;\n\n", number1, number ) ;
push(s2, number) ;
number++ ;
```

(f)

```
number1 = pop(s2) ;
printf("wire w%d_1, w%d_2, w%d_3, w%d_4 ;\n", number, number, number, number) ;
printf("anchor anchor%d( w%d_1, w%d_2, w%d_3, w%d_4, xxx ) ;\n",  number, number, number, number, number ) ;
printf("assign w%d_4 = w%d_1 ;\n", number, number1 ) ;
printf("assign w%d_2 = w%d_3 ;\n\n", number1, number ) ;
push(s2, number) ;
number++ ;
```

(g)

```
number1 = pop(s2) ;
number2 = pop(s2) ;
printf("wire w%d_1, w%d_2, w%d_3, w%d_4, w%d_5, w%d_6 ;\n", number, number, number, number, number,
number) ;
printf("concatenation concatenation%d( w%d_1, w%d_2, w%d_3, w%d_4, w%d_5, w%d_6 ) ;\n", number, number,
number, number, number, number, number ) ;
printf("assign w%d_2 = w%d_3 ;\n", number2, number ) ;
printf("assign w%d_2 = w%d_5 ;\n", number1, number ) ;
printf("assign w%d_4 = w%d_1 ;\n", number, number2 ) ;
printf("assign w%d_6 = w%d_1 ;\n\n", number, number1 ) ;
push(s2, number) ;
number++ ;
```

(h)

```
number1 = pop(s2) ;
printf("wire in ;\n") ;
printf("assign out = w%d_1 ;\n", number1) ;
printf("assign w%d_2 = in ;\n\n", number1) ;
printf("assign in = 1 ;\n\n") ;
```

(i)

Figure 4.4: (a) routing for comparator (b) routing for | (c) routing for * (d) routing for ? (e) routing for + (f) routing for . (d) routing for ? (g) routing for ^ or $ (h) routing for concatenation (d) routing for i/o ports

For more understanding, we demonstrate this method by entering a simple regular expression $(a|b)*c(d|e)f*g$ as the input of the system. Please refer to the Figure 4.1, Figure 4.3 and Figure 4.4. The process of constructing the NFA for the regular expression $(a|b)*c(d|e)f*g$ is below. The verilog code corresponding to the input, $(a|b)*c(d|e)f*g$, is shown in Figure 4.5.

(1) The regular expression is traversed from the infix form, $(a|b)*c(d|e)f*g$, into the postfix form, $ab|*cde|f*g$

(2) In the beginning, the NFA construction algorithm prints 1st - 10th rows in Figure 4.5. The 5th row is shown for the circuit's name and the i/o ports. The 6th, 7th and 8th rows declare that clk, data and sf are input ports, where the size of the data is 8-bits for ASCII characters. The 9th row declares the out as the output.

(3) The NFA construction algorithm begins to read the regular expression. After reading the character a , the algorithm executes the subroutine shown in the Figure 4.4(a) since a belongs to the character. Thus, 11th and 12th rows are printed, where the 12th row declares the comparator and the i/o of the comparator. The stack stores the number of the comparator1.

(4) After reading the character b , the process is similar with (3). The stack stores the number of the comparator2.

(5) After reading the metacharacter |, the algorithm executes the subroutine shown in the Figure 4.4(b). The stack pops 2 and 1. By means of the union3, the union of the a and b is implemented(shown in 18th – 23rd rows). The stack stores the number of the union3.

(6) After reading the metacharacter *, the algorithm executes the subroutine shown in the Figure 4.4(c). The stack pops 3. By means of the closure4, the closure of $(a|b)$ is implemented (shown in 25th – 27th rows). The stack stores the number of closure4.

(7) After reading the character c , the process is similar with (3). The stack stores the number of the comparator5.

(8) After reading the character d , the process is similar with (3). The stack stores the number of the comparator6.

(9) After reading the character e, the process is similar with (3). The stack stores the number of the comparator7

(10) After reading the metacharacter |, the algorithm executes the subroutine shown in the Figure 4.4(b). The stack pops 7 and 6. By means of the union8, the union of d and e is implemented (shown in 39th – 43rd rows). The stack stores the number of the union8.

(11) After reading the character f, the process is similar with (3). The stack stores the number of the comparator9.

(12) After reading the metacharacter *, the algorithm executes the subroutine shown in the Figure 4.4(c). The stack pops 9. By means of the closure10, the closure of f is implemented (shown in 49th – 51st rows). The stack stores the number of closure10.

(13) After reading the character g, the process is similar with (3). The stack stores the number of the comparator11.

(14) Reading the regular expression is finished; the algorithm executes the subroutine shown in the Figure 4.4(h). The stack pops 11 and 10. By means of the concatenation12, the concatenation of $f*$ and g is implemented (shown in 57th – 61st rows). The stack stores the number of the concatention12.

(15) The algorithm executes the subroutine shown in the Figure 4.4(h). The stack pops 12 and 8. By means of the concatenation13, the concatenation of $(d|e)$ and $f*g$ is implemented (shown in 64th – 68th rows). The stack stores the number of the concatention13.

(16) The algorithm executes the subroutine shown in the Figure 4.4(h). The stack pops 13 and 5. By means of the concatenation14, the concatenation of c and $(d|e)f*g$ is implemented (shown in 71st – 75th rows). The stack stores the number

of the concatention14.

(17) The algorithm executes the subroutine shown in the Figure 4.4(h). The stack pops 14 and 4. By means of the concatenation15, the concatenation of $(a|b)*$ and $c(d|e)f*g$ is implemented (shown in 78th – 82nd rows). The stack stores the number of the concatention15.

(18) Finally, the number, 15, which the stack pops is the last one. The algorithm executes the subroutine shown in the Figure 4.4(i), i/o ports of the circuit is connected (shown in 85th – 88th rows).

```
//*****************************************
//*************** main *****************
//*****************************************

module test(clk, data, sf, out) ;
input clk ;
input [7:0] data ;
input cf  ;
output out ;

wire w1_1, w1_2 ;
comparator_a comparator1( w1_1, w1_2, clk, data ) ;

wire w2_1, w2_2 ;
comparator_b comparator2( w2_1, w2_2, clk, data ) ;

wire w3_1, w3_2, w3_3, w3_4, w3_5, w3_6 ;
union union3( w3_1, w3_2, w3_3, w3_4, w3_5, w3_6 ) ;
assign w1_2 = w3_3 ;
assign w2_2 = w3_5 ;
assign w3_4 = w1_1 ;
assign w3_6 = w2_1 ;

wire w4_1, w4_2, w4_3, w4_4 ;
closure closure4( w4_1, w4_2, w4_3, w4_4 ) ;
assign w4_4 = w3_1 ;
assign w3_2 = w4_3 ;

wire w5_1, w5_2 ;
comparator_c comparator5( w5_1, w5_2, clk, data ) ;

wire w6_1, w6_2 ;
comparator_d comparator6( w6_1, w6_2, clk, data ) ;

wire w7_1, w7_2 ;
comparator_e comparator7( w7_1, w7_2, clk, data ) ;

wire w8_1, w8_2, w8_3, w8_4, w8_5, w8_6 ;
union union8( w8_1, w8_2, w8_3, w8_4, w8_5, w8_6 ) ;
assign w6_2 = w8_3 ;
assign w7_2 = w8_5 ;
assign w8_4 = w6_1 ;
assign w8_6 = w7_1 ;

wire w9_1, w9_2 ;
```

```
comparator_f comparator9( w9_1, w9_2, clk, data ) ;

wire w10_1, w10_2, w10_3, w10_4 ;
closure closure10( w10_1, w10_2, w10_3, w10_4 ) ;
assign w10_4 = w9_1 ;
assign w9_2 = w10_3 ;

wire w11_1, w11_2 ;
comparator_g comparator11( w11_1, w11_2, clk, data ) ;

wire w12_1, w12_2, w12_3, w12_4, w12_5, w12_6 ;
concatenation concatenation12( w12_1, w12_2, w12_3, w12_4, w12_5, w12_6 ) ;
assign w10_2 = w12_3 ;
assign w11_2 = w12_5 ;
assign w12_4 = w10_1 ;
assign w12_6 = w11_1 ;

wire w13_1, w13_2, w13_3, w13_4, w13_5, w13_6 ;
concatenation concatenation13( w13_1, w13_2, w13_3, w13_4, w13_5, w13_6 ) ;
assign w8_2 = w13_3 ;
assign w12_2 = w13_5 ;
assign w13_4 = w8_1 ;
assign w13_6 = w12_1 ;

wire w14_1, w14_2, w14_3, w14_4, w14_5, w14_6 ;
concatenation concatenation14( w14_1, w14_2, w14_3, w14_4, w14_5, w14_6 ) ;
assign w5_2 = w14_3 ;
assign w13_2 = w14_5 ;
assign w14_4 = w5_1 ;
assign w14_6 = w13_1 ;

wire w15_1, w15_2, w15_3, w15_4, w15_5, w15_6 ;
concatenation concatenation15( w15_1, w15_2, w15_3, w15_4, w15_5, w15_6 ) ;
assign w4_2 = w15_3 ;
assign w14_2 = w15_5 ;
assign w15_4 = w4_1 ;
assign w15_6 = w14_1 ;

wire in ;
assign out = w15_1 ;
assign w15_2 = in ;

assign in = 1 ;

endmodule
```

Figure 4.5: the verilog code corresponding to the input, $(a \mid b)^* c(d \mid e) f^* g$

Chapter 5

Using the Decoder (Fit for ASIC)

This new approach uses the NFA construction algorithm proposed in Chapter 4 and logic structures which are modulated from Figure 4.2 and Figure 4.4. When the total area which is required for comparing the input with the character stored in the comparator exceeds the area of a single decoder, the work for comparison should be handed over to the decoder. Therefore, when the number of comparators exceeds a particular threshold, this approach can efficiently decreases the area required. Figure 5.1 shows the adjusted comparator and 8-to-256 decoder (because the ASCII has 8 bits). The following are two choices to implement the decoder. One is to individually assign the pin (for example, assign d97 = &(8'b01100001~^a) ;). When optimizing the circuit, pins not used wouldn't be synthesized and thus there is smaller area. The other is to decode total pins once and this method has smaller delay.

This algorithm doesn't always fit the FPGA for reducing the area used since there are various architectures of the FPGA and all kinds of algorithms for placement and routing. Every design team for FPGA has its own method for the compilation. But this approach can surely reduce gate count when it is implemented in the form of ASIC.

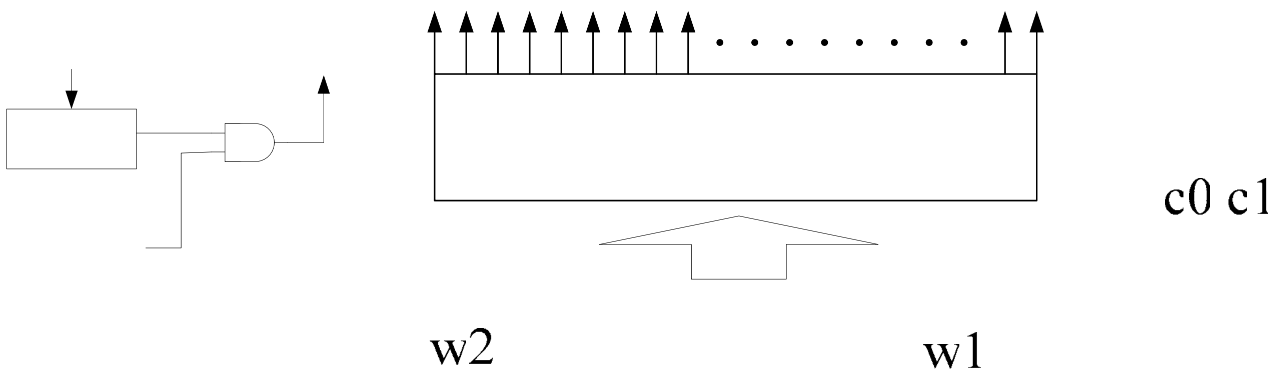


Figure 5.1: logic structures

Chapter 6

Performance and Comparison

The memory needed for the software, GNU grep, and the area of the FPGA required for the implementation of approaches which are mentioned in the chapter 3 and 4are showed in the Table 6.1. The regular expression tested is $(a|b)*a(a|b)^k$ which has k occurrences of $(a|b)$ at the end and k ranges from 8 to 28. This regular expression denotes all sequences of a's and b's in which the $(k+1)^{th}$ symbol from the end is an a. Below we introduce the environment of the simulation, performance and comparison.

The software program, GNU grep version 2.4, runs on a machine with an 800 MHz Pentium III Xeon processor and 2 GB RAM running Linux (Red Hat 6.2). GNU grep is the UNIX command and it uses the DFA for searching. The memory reported (showed in the second and third columns of Table 6.1) is the maximum memory used by the invocation grep. It uses an effective storage-reduction technique, “lazy transition evaluation”. The transition function is only computed when the DFA is run. Computed transitions are kept in a cache. Before a transition is made, the cache is examined. If the required transition is not in the cache, it is computed and stored for a subsequent use. Table 6.1 shows the result of simulation, and the data about GNU grep is quoted from [1].

The performance of the GNU grep depends on the text. The performance of the worst case is obtained by inputting the text which has all kinds of situation - taking the regular expression $a(a|b)^2$ for an instance, the text contains aaa, aab, aba and abb. As k increases, the memory and the time required reach unacceptable level since the complete transition function is constructed. The time mentioned above consists of the DFA construction time and the time searching the text. As k is 19, the time required is 87309.38 seconds (24.2525 hours). For best case, because there is only one transition constructed, the memory required is still a constant when k changes.

The performance of the approach proposed in [1] is obtained using Xilinx Foundation tools running on a 450 MHz Pentium III and the target device is the Virtex XCV100 FPGA. The performance of our approach is carried out by using Quartus II 4.0 running on a Pentium 4 and the target device is the APEX EP20K600EBC-6521X. Table 6.1 shows the result of simulation, and the data about the approach proposed in [1] is quoted from [1].

The performance of the approach in [1] depends on k. As k changes, the area required grows in $O(k^2)$. The construction time includes the time for the NFA construction time, the time for configuration bits generation and the time for configuring the FPGA. The last two terms dominates the construction time and the time for processing per character is in $O(1)$. k also determines the performance of our approach . Compared with the approach in [1] , using our approach only requires $O(k)$ area. However, it leads to longer time required since the construction time consists of the time for compiling the HDL code and the time for FPGA configuration. Because our approach generates the HDL code, the circuit can be implemented on ASIC and all kinds of FPGAs. Furthermore, the design software can easily optimize the circuit.

k	GNU grep	GNU grep	Approach in [1]	Our approach
	(best case)	(worst case)		
8	580 KB	1 MB	10×7 CLBs	11 LEs
9	580 KB	1 MB	11×8 CLBs	12 LEs
10	580 KB	1.9 MB	12×8 CLBs	13 LEs
11	580 KB	2.2 MB	13×9 CLBs	14 LEs
12	580 KB	3.0 MB	14×9 CLBs	15 LEs
13	580 KB	4.4 MB	15×10 CLBs	16 LEs
14	580 KB	7.5 MB	16×10 CLBs	17 LEs
15	580 KB	13 MB	17×11 CLBs	18 LEs
16	580 KB	26 MB	18×11 CLBs	19 LEs
17	580 KB	54 MB	19×12 CLBs	20 LEs
18	580 KB	111 MB	20×12 CLBs	21 LEs
19	580 KB	229 MB	21×13 CLBs	22 LEs
28			30×16 CLBs	31 LEs

Table6.1: the space required

k	GNU grep	GNU grep	Approach in [1]	Our approach
	(best case)	(worst case)		
8	0.01 s	0.00 s	21 ms	36s
9	0.05 s	0.00 s	39 ms	36 s
10	0.15 s	0.00 s	32 ms	36 s
11	0.50 s	0.00 s	34 ms	37 s
12	2.22 s	0.00 s	31 ms	36 s

13	16.11 s	0.005 s	29 ms	36 s
14	82.88 s	0.01 s	33 ms	37 s
15	345.33 s	0.03 s	34 ms	37 s
16	1383.55 s	0.04 s	34 ms	36 s
17	5499.60 s	0.08 s	37 ms	37 s
18	21900.36 s	0.17 s	37 ms	36 s
19	87309.38 s	0.34 s	31 ms	37 s
28			39 ms	37 s

Table6.2: the time required (Notice that definitions of the time in all columns are not the same)

Chapter 7

Conclusion

In this paper, we present the approach that constructs the NFA circuit for regular expression matching by automatically generating HDL code. Approaches in [1] require $O(n^2)$ area and $O(1)$ time for process per character (1 clock). However, the approach proposed in this paper only needs $O(n)$ area and still $O(1)$ time for processing per character. This new approach reduces the area required significantly and utilities the area efficiently. We implement metacharacters often used, and thus applications in reality become practicable. These applications include Snort, Clamav, and so on.

Advantages of generating the HDL code can let the optimization easier and fanout problems can be solved. Besides, the output of our approach are feasible for all kinds of FPGAs and ASIC. If generating the netlist, the optimization and fanout problem would be difficult to handle, and the design in [1] is only for a specific FPGA.

From the discussion above, we know that this new method provides a general solution for regular expression matching in hardware. The flexibility of this new method includes debug, suitable device, optimization, easily modifying the circuit, and so on. About area, the requirement is reduced from $O(n^2)$ to $O(n)$ and the utility rate becomes higher.

References

[1] R. Sidhu and V.K. Prasanna. Fast Regular Expression Matching using FPGAs, *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*,pp. Apr. 2001.

[2] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages, and Computation 2nd edition.* Addison-Wesley, 2000.

[3] M. Main, W. Savitch. *Data structures & other objects using C++ 2nd edition*. Addison-Wesley Longman, 2001.

[4] W. Wolf. *Modern VLSI design 3rd edition.* Prentice Hall, 2002

[5] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon. Afirst generation DPGA implementation. In FPD*'94- Third Canadian Workshop of Field-Programmable Devices*, pages 138-143, May 1995.

[6] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In J. Arnold and K. L. Pocek, editors, *Processings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 22-28, Napa, CA, April 1997.

[7] R. P.S. Sidhu, A. Mei, S. Wadhwa, and V. K. Prasanna. A self-reconfigurable gate

array architecture. In *FPGA’99. Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, Aug. 2000.

[8] A. V. Aho. *Handbook of Theoretical Computer Science, Volume A Algorithms and Complexity*, chapter 5. MIT Press/Elsevier, 1990.

[9] Friedl, Jeffrey E. F. *Mastering regular expressions*, O'Reilly, 2002

[10] Schwartz, Randal L. Christiansen, Tom. *Learning Perl*, O'Reilly & Associates, 1997

[11] Tom Christiansen and Nathan, Torkington. *Perl cookbook*, O'Reilly & Associates, 1998