AN NFA-BASED PROGRAMMABLE REGULAR EXPRESSION MATCHING ENGINE HIGHLY SUITABLE FOR FPGA IMPLEMENTATION

Hiroki Takaguchi, Yoichi Wakaba, Shin'ichi Wakabayashi, Shinobu Nagayama, Masato Inagi

Graduate School of Information Sciences, Hiroshima City University
3-4-1, Ozuka-higashi, Asaminami-ku, Hiroshima 731-3194, Japan
email: {wakaba,s_naga,inagi}@hiroshima-cu.ac.jp

Abstract— Regular expression matching is an operation to find a string that matches a given pattern described as a regular expression in an input text. This paper proposes a new programmable regular expression matching engine based on a string-transition NFA. The proposed engine can perform matching at high speed, and any regular expression can be set as a pattern in a very short time. The proposed hardware engine has a two-dimensional array structure, and thus it is highly suitable for FPGA implementation. In experiments, the proposed engine is implemented on an FPGA. Comparing with an existing hardware matching engine, the effectiveness of the proposed hardware was evaluated.

I. INTRODUCTION

String matching is the problem of finding all occurrences of a character pattern in a text. Since this problem has many applications in computer engineering and information processing, a number of research results have been presented in last 50 years [1, 7]. There are many formulations of string matching. Among them, *regular expression matching* is a string matching problem, in which a regular expression is given as a pattern [7]. Regular expression matching has a broad range of applications. In particular, in recent years, its application to network intrusion detection systems (NIDSs) of high speed networks has attracted much attention [6]. NIDSs monitor network traffic for predefined suspicious activities or data patterns, and notify system administrators when malicious traffic is detected so that appropriate actions may be taken. Regular expressions are used to describe hazardous contents in packet payloads.

As known well, recognizing a character string, which matches a given pattern described as a regular expression, can be realized by implementing either non-deterministic or deterministic finite automata (NFAs or DFAs) [5]. In recent years, there have been many investigations to implement NFAs and DFAs on hardware in order to realize efficient regular expression matching. NIDS is a typical example of such applications [6], since software implementation of regular expression matching for NIDS would be very impractical due to its execution time. For this application, many results have been presented for FPGA implementation of regular expression matching machines [6, 2, 3, 8].

In most previous results on regular expression matching by using FPGAs, a pattern is given before generating hardware configuration data of FPGA chips. This instance-specific approach to the regular expression matching problem has several advantages such as reducing the hardware resources, and improving the execution time. However, this approach has one major disadvantage. If a pattern is updated, then a sequence of FPGA design and implementation processes (i.e., generating a HDL description of hardware, logic synthesis, place and route, and generating a configuration data) should be performed again [4]. For some applications of regular expression matching, this property would be fatal. In particular, for NIDS, when a "suspicious" pattern is newly found, it is strongly recommended that this pattern should be installed in the NIDS as soon as possible.

In this paper, we address design and FPGA implementation of a hardware string matching engine for recognizing regular expressions. The architecture of the proposed regular expression matching circuit was a two-dimensional array of simple processing units. A pattern is set in the circuit before string matching, and a text to be retrieved is entered into the circuit one character by one character. The proposed circuit was designed with Verilog-HDL, and was implemented using a Xilinx Virtex6 chip. Experimental evaluations show that the proposed circuit achieves a high throughput of 2.736 Gigabits per second.

This paper is organized as follows. Section II formulates the regular expression matching problem. Section III proposes a hardware string matching engine for recognizing regular expressions. Section IV shows the result of design and implementation of the proposed machine. Some experimental results are also given. Finally, in Section V, some concluding remarks are described.

II. PRELIMINARIES

A. Regular Expressions [7]

Let $\Sigma = \{a_1, a_2, \ldots, a_s\}$ be a set of symbols (i.e., characters) called an *alphabet*. A *regular expression* R is a string on the set of symbols $\Sigma \cup \{\varepsilon, |, \cdot, *, (,)\}$, which is recursively defined as the empty character ε, a character $a_i \in \Sigma$, and (R_1), $(R_1 \cdot R_2)$, $(R_1|R_2)$, and (R_1^*), where R_1 and R_2 are regular expressions. When there is no ambiguity, we simplify our expressions by writing R_1R_2 instead of $(R_1 \cdot R_2)$. It is usual to use also the precedence order "$*$", "$\cdot$", "$|$" to remove more parentheses.

The language represented by a regular expression R, denoted $L(R)$, is a set of strings over Σ, which is defined recursively on the structure of R as follows:

- If R is ε, then $L(R) = \{\varepsilon\}$.
- If R is $a_i \in \Sigma$, then $L(R) = \{a_i\}$.
- If R is of the form (R_1), then $L(R) = L(R_1)$.
- If R is of the form $(R_1 \cdot R_2)$, then $L(R) = L(R_1) \cdot L(R_2)$, where "$\cdot$" in the right-hand side represents the concatenation of string sets.
- If R is of the form $(R_1|R_2)$, then $L(R) = L(R_1) \cup L(R_2)$.
- If R is (R_1^*), then $L(R) = L(R)^* = \bigcup_{i \geq 0} L(R_1)^i$, where $L(R_1)^i = L(R_1) \cdot L(R_1) \cdot \cdots \cdot L(R_1)$ (i times).

We say a character string w on alphabet Σ *matches* pattern P if $w \in L(P)$ where P is a regular expression on Σ. The problem of regular expression matching discussed in this paper is the problem of finding all occurrences of substrings, which match a given pattern P, where P is a regular expression on alphabet Σ.

Given a pattern P, the length of P is defined as the number of characters in Σ included in P. That is, $\cdot$, $|$ and $+$ operators and parentheses in P are not counted.

B. Extended Regular Expressions

We introduce several operations into regular expressions described in the previous section so that complicated patterns can be specified in a short form. Those operations are defined as follows. In the following, R is a (extended) regular expression, and a_i is a character in Σ.

- $R? ::= R|\varepsilon$
- $R^+ ::= R|R^2|R^3|\cdots$
- $. ::= a_1|a_2|a_3 \cdots a_{s-1}|a_s$
- $[a_{i_1}, a_{i_2}, \ldots, a_{i_k}] ::= a_{i_1}|a_{i_2}|\cdots|a_{i_k}$
- $[a_i - a_j] ::= a_i|a_{i+1}|a_{i+2} \cdots a_{j-1}|a_j$

We call $[a_{i_1}, a_{i_2}, \ldots, a_{i_k}]$ and $[a_i - a_j]$ *class* characters. Note that all extensions described above do not affect the class of languages defined by regular expressions. That is, a language represented by an extended regular expression is the regular language.

C. Finite Automata

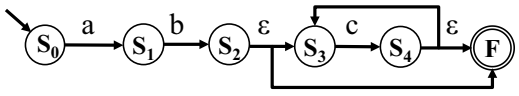
As known well, the regular language defined by a regular expression can be recognized by either a deterministic or non-deterministic finite automaton (DFA/NFA) [5]. In this study, we pay attention to the NFA and its extension.

C.1. NFA

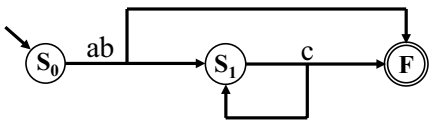
A *non-deterministic finite automaton (NFA)* M is defined as $M = (S, \Sigma, \delta, q_0, F)$, where S is a finite set of *states*, Σ is an alphabet of input character strings, $\delta : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$ is a *transition function*, q_0 is an *initial* state, and $F \subseteq S$ is a set of *final* states. Figure 1 (a) shows an NFA for regular expression abc^*.

C.2. String-Transition NFA

We extend the transition function δ for an NFA to apply to strings in $\Sigma^* - \{\varepsilon\}$, and define the extended transition function as $\hat{\delta} : S \times (\Sigma^* - \{\varepsilon\}) \rightarrow 2^S$. In this paper, this extended NFA $\hat{M} = (S, \Sigma, \hat{\delta}, q_0, F)$ is called the *string-transition NFA*. Note that a string-transition NFA is ε-free. It is easy to show that the class of languages recognized by string-transition NFAs is the regular language. Figure 1 (b) shows an NFA for regular expression abc^*.



(a) NFA for abc*.



(b) String-transition NFA for abc*.

Fig. 1. NFA and string-transition NFA for abc^*.

III. THE PROPOSED CIRCUIT

A. Overview of the Circuit

In this study, we propose a new regular expression matching hardware engine, in which a pattern is described as a regular expression. The proposed hardware engine is "*programmable*", that is, any regular expression can be set as a pattern in the matching engine without circuit reconfiguration.

As explained in C.2, for a regular expression P given as a pattern to be matched with an input text T, there is a string-transition NFA M such that $L(M) = L(P)$, where $L(M)$ and $L(P)$ are a language recognized by M and a regular language defined by P. For each state in M, the state transition is defined for not only single characters but also strings including empty strings on Σ. Let $Str(P)$ be a set of strings, each of which is used to define a state transition from some state in M. Each element in $Str(P)$ may consist of any element in alphabet Σ, and, possibly, class characters such as $[a_i - a_j]$.

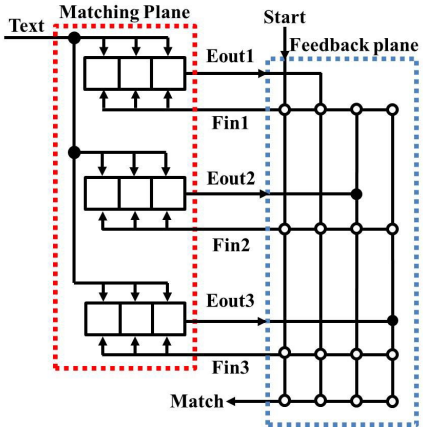


Fig. 2. The overall structure of the proposed engine.

Figure 2 shows the overall structure of the proposed circuit. The proposed circuit has a two-dimensional array structure, consisting of two subcircuits, called the *matching plane* and the *feedback plane*. The former performs the matching between the input text and each string in $Str(P)$, where P is a regular expression given as a pattern, and the latter performs the state transition of a string-transition NFA, which accepts $L(P)$. Details of those two subcircuits will be given in the following subsection.

B. The Matching Plane

The matching plane consists of several rows of one-dimensional arrays of small processing units. Each one-dimensional array is called the *string matching unit*, shown in Figure 3, which performs matching between the input text and a string in $Str(P)$. For each string w in $Str(P)$, one string matching unit is assigned to perform matching between the input text and w. Each w in $Str(P)$ is stored in some string matching unit in advance before matching.

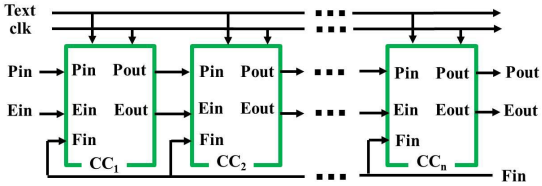


Fig. 3. The string matching unit.

Each small processing unit in the string matching unit is called the *comparison cell*, which is dedicated to compare a character in the text and a character (possibly, a class character) in the pattern. There are two types of string matching units, the *basic* string matching unit and the *class* string matching unit. The former consists of basic comparison cells and the latter consists of class comparison cells. Those two types of string matching arrays are explained below.

B.1. The Basic String Matching Unit

As described, for each w in $Str(P)$, there is a string matching unit, which performs matching between the input text and w. For the basic string matching unit, w consists of elements in Σ only, and no class characters are included.

The basic string matching unit consists of basic comparison cells, which performs matching between a character in the text and a character in w. Figure 4 shows the structure of the basic comparison cell. It consists of a register LP, a comparator CMP, a selector SEL, and a flipflop FF. LP stores a character in the pattern. LPs in basic string matching units can be also functioned as shift registers by connecting input and output terminals, P_{in} and P_{out}, of comparison cells. Each character in the pattern P is set from the outside of the circuit in advance by shifting them in LPs.

CMP is used to compare the character stored in LP and the character in the text fed from the input terminals of the matching plane. Each character in the text is broadcasted to the all comparison cells in the circuit. The result of comparison is ANDed with the internal enable signal E, and the result is stored in FF. E_{in} and E_{out} are enable input and output signals, respectively, of the comparison cell. F_{in} is a feedback transition signal from the feedback plane. The enable signal E is either provided from the outside of the basic string matching array, F_{in}, or E_{out} of the left comparison cell in the array. This selection is realized by SEL.

B.2. The Class String Matching Unit

The class string matching unit consists of class comparison cells, which performs matching between a character in the

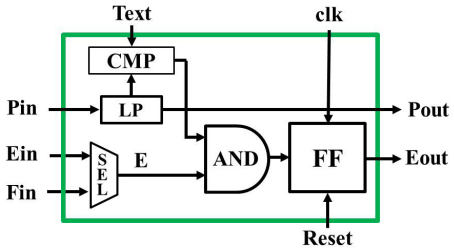


Fig. 4. The basic comparison cell.

text and a class character in the pattern. Figure 5 shows the circuit structure of the class comparison cell. It consists of a matching table MT, a selector SEL, and a flipflop FF. The MT is a random access memory (RAM), whose word width is 1 bit. The memory address of a MT is given by the input text character. Let $MT[x]$ be the value of a memory word whose address is x. We assume that a text character is represented by a 7 bit ASCII code. Let $c[i_1, i_2, \ldots, i_k]$ be a class character representing $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ in the alphabet Σ. Then $MT[x] = 1$ if a_x matches with $c[i_1, i_2, \ldots, i_k]$, otherwise $MT[x] = 0$. The output of the MT is ANDed with internal signal E and the result is stored in FF. Using the MT, matching with any class character can be realized. Note that the class string matching unit can be also used as the basic string matching unit. Comparison cells in one class string matching unit shares the RAM to implement the MTs. In FPGA implementation, MTs are implemented using Block RAMs (BRAMs).

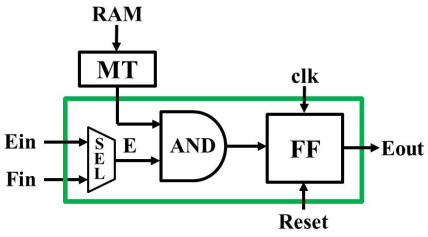


Fig. 5. The class comparison cell.

C. The Feedback Plane

The feedback plane realizes the state transition from the current state of the NFA to the next state. The feedback plane consists of n vertical signal lines, each of which is connected to E_{out} of the rightmost comparison cell in a string matching unit, and n horizontal signal lines, which is connected to F_{in} of each comparison cell in a string matching unit, where n is the number of string matching units in the matching plane. There is another vertical signal line, denoted $Start$, which is connected to the input terminal of the circuit. There is also another horizontal signal line, denoted $Match$, which is connected to the output terminal of the circuit. At each point where a vertical and a horizontal signal lines are crossed, there is a programmable switch which connects two lines. The signal line $Start$ is used to start matching. When $Start = 1$, the initial state is enabled to start matching. The signal line $Match$ shows the success of matching when $Match = 1$. Figure 6 shows the detailed structure of the feedback plane. For each switch, there is a FF, and depending on the value of FF, the switch is turned on or off.

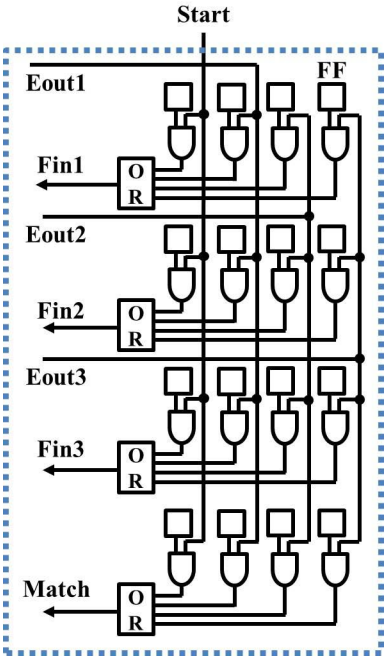


Fig. 6. The feedback plane.

D. Setting a Pattern

Now, we explain how to "*program*" the proposed matching engine when a regular expression is given as a pattern to be searched in the input text. Let P be the regular expression as a pattern, and M be the string-transition NFA for P. Let $Str(P)$ be a set of strings, each of which is used to define the transition function of M. For each w in $Str(P)$, w is set to any row of string matching units in the matching plane. For the basic string matching unit, this is done by the shift register function of LPs in the basic comparison cells. For the class string matching unit, this is done by writing memory data to BRAMs. For the memory access from the outside of the circuit, the dedicated circuit is included. Characters in string w are set in the string matching unit so that they are flush on the right. Depending on the position of the first character in w, selector SEL in each comparison cell is set appropriately so that the state transition signal F_{in} from the previous state is provided to the comparison cell having the first character in w.

For the feedback plane, depending on the transition function, switches in the plane are appropriately set. Note that for each string matching unit in the matching plane, in which string w is set, there is a state p in M such that $\hat{\delta}(p, w) = q$. For each string matching unit corresponding state p, there are a vertical signal line, denoted $E_{out}[p]$, in the feedback plane, which is connected to E_{out} of the rightmost comparison cell, and a horizontal signal line, denoted $F_{in}[p]$, which is connected to F_{in} in each comparison cell. Assume that $\hat{\delta}(p, w) = q$, then $E_{out}[p]$ and $F_{in}[q]$ is connected by turning the switch on, which is placed at the cross point of $E_{out}[p]$ and $F_{in}[q]$. Note that turning the switch on is setting 1 to its associated FF. Figure 7 shows the pattern setting for "$a(bc|de)$".

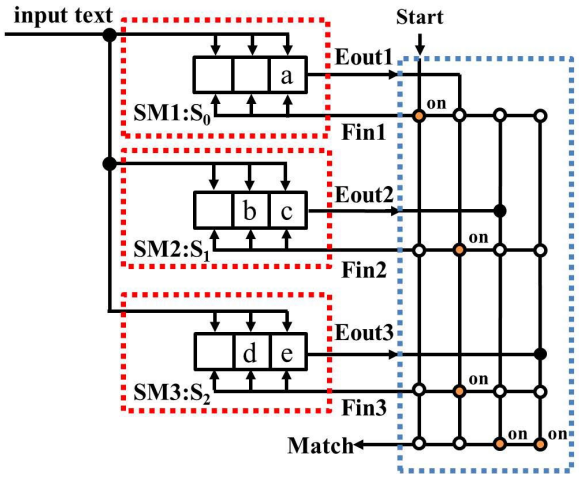


Fig. 7. Setting "$a(bc|de)$" as a pattern.

E. Suitability for FPGA Implementation

As noted, the proposed engine has a two-dimensional array structure, in which comparison cells in the matching plane and switch circuits in the feedback plane are regularly placed. Since state-of-the-art FPGAs have also a two-dimensional array structure consisting of logic blocks, it is very easy to implement the proposed engine on the FPGA chip. Furthermore, we show that block RAMs (BRAMs) can be effectively utilized to implement the class comparison cells in the class string matching units. Since state-of-the-art FPGA chips usually have several hundreds of BRAMs in the chip, area-efficient circuit implementation of the proposed engine can be achieved.

IV. EXPERIMENTAL EVALUATION

We have designed the proposed matching engine presented in this paper with Verilog-HDL, and implemented it on an FPGA board, which contains an Xilinx Virtex6 XC6VLX75-1FF784 FPGA chip (the number of logic elements = 74,496, the number of slices = 11,640). The design tool used in experiments was Xilinx ISE Design Suite Version 14.2. In the current implementation, a character in pattern and text consists of 8 bits.

Table I. Design results

Circuit size	Clock [MHz]		#slice (#usage [%])	
#state × #length	Basic	Class	Basic	Class
5 × 5	341	356	79 (0)	38 (0)
10 × 10	359	313	283 (2)	109 (1)
15 × 15	340	301	694 (5)	264 (2)
20 × 20	288	276	951 (8)	370 (3)
25 × 25	209	262	2001 (17)	449 (3)
30 × 30	222	251	2179 (18)	744 (6)

The design result was summarized in Table I. In this table, #state and #length represents the maximum number of states and the maximum length of strings in any transition in the string-transition NFA, respectively. In the design, in the matching plane, either basic or class string matching units was used. "Basic" and "Class" represents the design results of the proposed matching engines, in which basic string matching units and the class string matching units are used as the string matching units, respectively. For each case of the design, we have generated the Verilog-HDL source of the circuit, performed the logic synthesis, and then performed the place and route to get the configuration data of the circuit. "Clock" shows the clock frequency of the circuit. #slice and #usage shows the number of slices and the percentage of slice usage for each design.

From the table, we see that the implementation with class string matching units was much smaller in size than the implementation with basic string matching units. This is due to the fact that class string matching unit does not need comparators. Comparison with text characters was implemented with BRAMs, hence no slices (i.e., logic elements) were needed. Since state-of-the-art FPGAs usually equipped many BRAMs, our implementation is very suited to those FPGAs. Note that the function of class string matching units is higher than that of basic string matching units. From the performance point of view, the implementation with class string matching units was faster than the implementation with basic string matching unit. This may be explained by the fact that the circuit size of the former is smaller than the latter.

We also compared the design result of the proposed matching engine with the FPGA implementation of the systolic string matching algorithm with string-transition NFA, which we have previously proposed in [9] (in the following, we call it the *previous* algorithm). In the previous algorithm,

string matching was performed in the one-dimensional systolic array, and the state transition in the string-transition NFA was done in the NFA circuit. Note that the circuit structure of the programmable NFA circuit in the previous machine was totally different from the proposed machine in this paper. Due to the limit of space, we omit the details.

Table II. Comparison with the previous algorithm [9].

	Previous	Proposed
Clock [MHz]	191	251
#slice (#usage[%])	8237 (69)	744 (6)

We have designed and implemented the previous algorithm using the same design environment and FPGAs as used for the proposed algorithm. The design results were summarized in Table II. In the design, the maximum number of states (#state) and the maximum length of strings (#length) in one transition in the string-transition NFA were set to 30 and 30, respectively. In the design, for string matching units in the matching plane, class string matching units was used. #slice and #usage show the number of slices and the percentage of slice usage for each design, respectively. From the table, we see that the proposed engine was much smaller than the previous hardware algorithm. This is due to the fact that the number of registers used in the design of the previous algorithm was much larger than the proposed algorithm, since every comparison cells in the systolic string matching array has registers to store the text data. In addition, in the implementation of the previous algorithm, no BRAMs were used to realize comparison functions.

V. CONCLUSION

In this paper, we have proposed a new programmable regular expression matching hardware engine, in which any regular expression, possibly including class characters, can be set as a pattern without circuit reconfiguration. Since the proposed engine has a two-dimensional circuit structure, it was very suitable for FPGA implementation. We have also presented an area-efficient implementation of the proposed engine by effectively utilizing BRAMs in the FPGA chip. Experimental results showed that the proposed matching engine outperformed the previous hardware engine in both the size and the speed of the circuit. Future research includes the extension of regular expression such as quantitative specifiers (e.g., a{2,5}, which means the repetition of "a" in at least 2 and at most 5 times), and development of the software program for automatic generation of configuration data of the proposed matching engine from a regular expression given as a pattern.

ACKNOWLEDGMENTS

This research was supported in part by a Grant-in-Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS) under Grant (C) 23500066.

VI. REFERENCES

[1] J. Aoe (eds.), *Computer Algorithms: String Pattern Matching Strategies*, IEEE Computer Society Press, 1994.

[2] C.R. Clark, D.E. Schimmel, "Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns," *Proc. 2003 IEEE ICFPL*, pp.956–959, 2003.

[3] T. Ganegedara, Y.E. Yang, V.K. Prasanna, "Automation framework for large-scale regular expression matching on FPGA," *Proc. 2010 IEEE ICFPL*, pp.50–55, 2010.

[4] M. B. Gokhale, P. S. Graham, *Reconfigurable Computing*, Springer, 2005.

[5] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*, Pearson Education, 2006.

[6] B. L. Hutchings, R. Franklin, D. Cover, "Assisting network intrusion detection with reconfigurable hardware," Proc. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp.111–120, 2002.

[7] G. Navarro, M. Raffinot, *Flexible Pattern Matching in Strings*, Cambridge University Press, 2002.

[8] T. Trung Hieu, T. Ngoc Thinh, T. Huy Vu, S. Tomiyama, "Optimization of Regular Expression Processing Circuits for NIDS on FPGA," *Proc. 2011 IEEE ICNC*, pp.105–112, 2011.

[9] Y. Wakaba, M. Inagi, S. Wakabayashi, S. Nagayama, "An efficient hardware matching engine for regular expression with nested Kleene operators," *Proc. 2011 IEEE ICFPL*, pp.157–161, 2011.