A Regular Expression Matching Circuit: Decomposed Non-deterministic Realization With Prefix Sharing and Multi-Character Transition

Hiroki Nakahara[a], Tsutomu Sasao[b], Munehiro Matsuura[b]

[a]*Kagoshima University, Korimoto, 1-21-40, Kagoshima, Japan*
[b]*Kyushu Institute of Technology, Kawazu, 680-4, Iizuka, Fukuoka, Japan*

Abstract

This paper shows a compact realization of regular expression matching circuits on FPGAs. First, the given regular expression is converted into a non-deterministic finite automaton (NFA) by the modified McNaughton-Yamada method. Second, to reduce the number of the states in the NFA, prefixes for the NFA are shared. Also, the NFA is converted into the NFA with multi-character transition (MNFAU: Modular non-deterministic finite automaton with unbounded string transition). Third, the MNFAU is decomposed into the transition string part and the state transition part. The transition string part is represented by the Aho-Corasic deterministic finite automaton (AC-DFA), and it is implemented by an off-chip memory and a register. On the other hand, the state transition part is implemented by a cascade of logic cells (LCs) and the the interconnection on the FPGA. We implemented the regular expressions for SNORT (an open source intrusion detection system) on a Xilinx FPGA. Experimental results showed that, the embedded memory size per a character of the MNFAU is reduced to 0.2% of the pipelined DFA; 4.2% of the bit-partitioned DFA; 41.0% of the MNFAU(3); and 71.4% of the MNFAU without prefix sharing. Also, the number of LCs per a character of the MNFAU is reduced to 0.9% of the pipelined DFA; 15.6% of the NFA; and 80.0% of MNFAU without prefix sharing.

Keywords: Regular Expression, FPGA, IDS, NFA, MNFAU

1. Introduction

1.1. Regular Expression Matching in Network Applications

A regular expression, which consists of **characters** and **meta characters**, represents a set of strings. Various network applications (e.g., intrusion detection systems [3, 26, 9], a spam filter [28], a virus scanning system [7], and an L7 filter [14]) use **regular expression matching** to detect malicious data in incoming packets. Regular expression matching spends a considerable fraction of the total computation time for these applications. The throughput using the Perl compatible regular expression (PCRE) library [23] on a general purpose MPU is up to hundreds of Mega bits per second (Mbps) [24], which is too low for most applications. Thus, a dedicated circuit for regular expression matching is required. For network applications, since the high-mix low-volume production and the frequent update for new protocols are required, FPGAs are widely used. With the advent of FPGAs embedding dedicated high-speed transceivers for the high-speed network, we expect extensive use of FPGAs in the future.

For different users, systems with different performance and price are required. Thus, different architectures should be designed. For the IXPs (Internet exchange points) and the ISPs (Internet service providers), a high throughput, e.g., more than tens of Giga bits per second (Gbps), is the first priority, but the cost of the systems is the second priority. However, for low-end users, such as SOHO (small office and home office), a low cost system is the first priority. Since the cost for the FPGA increases with the number of LCs, reduction of the number of LCs lowers the system cost. In this paper, we propose the regular expression matching circuit that requires fewer LCs than conventional methods.

1.2. Related Work

Regular expressions are detected by finite automata (FA). In a deterministic finite automaton (DFA), for each state and each input, there is a unique transition, while in a non-deterministic finite automaton (NFA), for each state for each input, multiple transitions may exist. In an NFA, there exists ε**-transitions** to other states without consuming input characters. Floyd and Ullman [10] proposed a hardware implementation of the regular expression matching based on the NFA in 1982. After this, various FA-based regular expression matching circuits have been proposed.

Realization of regular expression matching circuits: Most of the proposed regular expression matching circuits are based on finite automata. DFA-based regular expression matchings include: An Aho-Corasick DFA (AC-DFA) algorithm on a computer [1]; a bit-partitioned AC-DFA [30]; a combination of the bit-partitioned AC-DFA and the MPU [4]; and a pipelined DFA [6]. Also, NFA-based regular expression matchings include: An algorithm that emulates the NFA by shift and AND operations (Baeza-Yates's algorithm) [2]; an FPGA realization of Baeza-Yates's algorithm [25]; an FPGA realization reduced by prefix and postfix sharing of regular expressions [15]; and a method that maps repeated parts of regular expressions into the Xilinx FPGA primitive (SRL16) [5].

Complexity Analysis of DFAs: For regular expression matching circuits based on DFAs, the area is proportional to the number of states. Moscola et al. [17] analyzed the DFA representing the regular expression *without* meta characters. Yu et al. [34] analyzed the DFA representing the regular expression *with* meta characters. Also, Dixon et al. [8] analyzed the bit-partitioned AC-DFA.

Regular Expression Matching Circuit Based on the NFA with String Transition: A conventional NFA is based on a single-character transitions [25]. Since the area is proportional to the number of states for the NFA, the reduction of the number of states also reduces the area.

In [18], we developed a regular expression matching circuit based on the NFA with string (multi-character) transition. This method drastically reduced the number of the states, and won the design contest for the eighth ACM/IEEE international conference on formal methods and models for co-design (MEMOCODE2010) [22]. We analyzed the area complexity for the NFA with string (bounded characters) transition [19]. To further reduce the number of states, we proposed the NFA with unbounded characters [20].

1.3. Xilinx FPGA

The proposed method uses the Xilinx FPGA to implement the regular expression matching circuit. Here, we introduce the Xilinx FPGA. Fig. 1 shows the structure for the Xilinx FPGA consisting of configurable logic blocks (CLBs), block RAMs (BRAMs), I/O blocks, and interconnection elements [1]. Fig. 2 shows the structure for the CLB. In the Xilinx FPGA, a CLB consists of four **SLICEs**, and a SLICE consists of two **logic cells (LCs)**. An LC consists of a four-input **look-up table (LUT)** [2] and a flip-flop. In the CLB, two types of SLICEs (SLICEM and SLICEL) exist. An LUT on the SLICEM can be configured as a shift register (SRL16) or an LUT, while an LUT on the SLICEL can be configured only as an LUT [33]. Fig. 3 shows two LUT modes of a Xilinx FPGA.

1.4. Contributions of Our Previous Works [18, 19, 20, 21]

We showed compact realizations of regular expression matching circuits. In the regular expression matching circuits, we assume that the area is proportional to the number of states in the NFA. In a conventional NFA, the NFA goes to the next state by consuming a character. By merging a sequence of states in an NFA, we have an NFA with string (in other words, multi-character) transition. In most cases, the NFA with string transition has fewer states than the conventional NFA with a character transition. However, a special technique (e.g., a DFA) is necessary to implement the string transitions. An implementation of the string transitions by using the primitive for the Xilinx FPGAs is shown in [20].

[1] Additionally, some FPGAs have DSP blocks, DLLs, and embedded processors.

[2] For modern FPGAs (Spartan VI, and Virtex 6), six or five inputs LUTs can be configured by using multiple four-input LUTs.

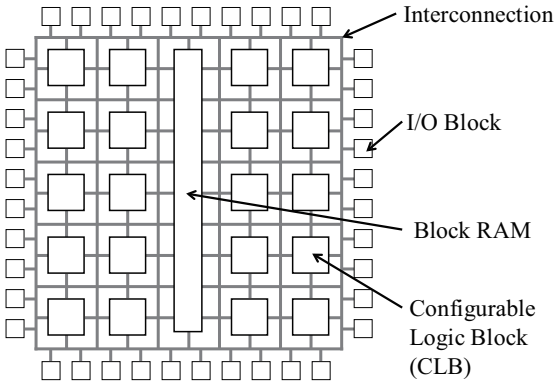


Figure 1: Architecture of the Xilinx FPGA.

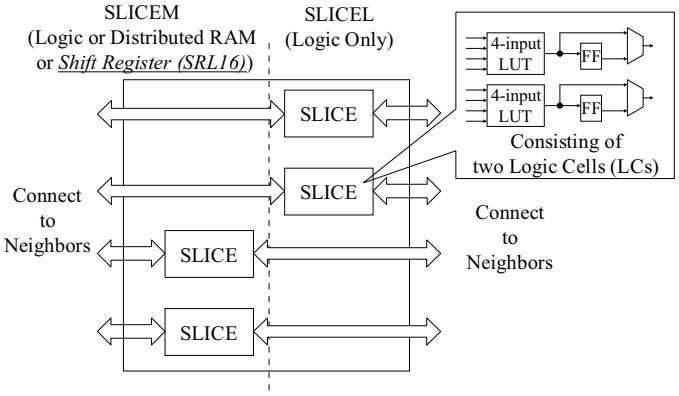


Figure 2: CLB structure of Xilinx FPGA.

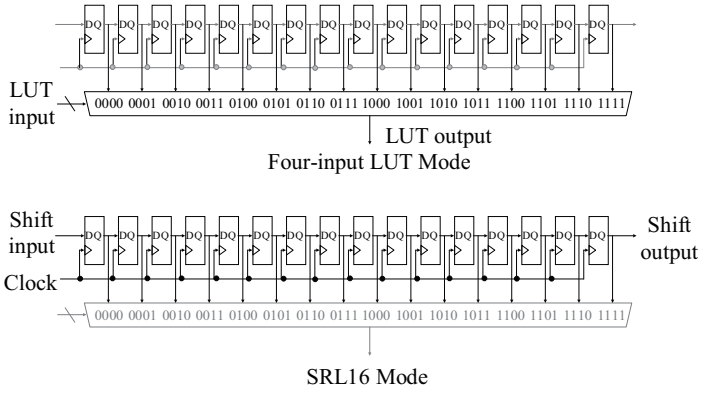


Figure 3: Two LUT modes for Xilinx FPGA.

1.5. Contributions of This Paper

This paper is an extension of previous publications [18, 19, 20, 21]. New contributions are:

1. Application of the prefix sharing to the regular expressions for the SNORT (an open source instruction detection system).
   Regular expressions in the SNORT have common prefixes, and they can be shared without using any additional logical elements [5].
2. Implementation of *all* the regular expressions for the SNORT including an extended regular expression. In the previous papers, only the subset of the SNORT were implemented.

1.6. Organization of the Paper

The rest of the paper is organized as follows: Section 2 introduces the Perl compatible regular expression (PCRE); Section 3 shows a regular expression matching circuit based on the FA. Section 4 introduces area reduction techniques for the regular expression matching circuit; Section 5 shows a regular expression matching circuit based on an NFA with string transition; Section 6 shows a design method of a regular expression matching circuit based on the NFA with string transition; Section 7 shows the experimental results; and Section 8 concludes the paper.

2. Perl Compatible Regular Expression (PCRE)

This section briefly introduces a **Perl compatible regular expression (PCRE)** which is used in this paper.

A PCRE consists of **characters** and **meta characters**. A character is represented by eight bits. The **length** of the regular expression is the number of characters. Table 1 shows the PCRE considered in this paper, where *R* denotes a regular expression. Our regular expression matching circuit does not support some meta characters as follows: anchors (\A,\Z, and \z); word boundaries (\B, and \b); continuing from the previous match (\G); and application specific flags (\U,\R, and \B). Since these meta characters are not used in network intrusions [5], we need not consider the realization of these meta characters [3].

3. Regular Expression Matching Circuit Based on Finite Automaton

A regular expression can be converted into an equivalent finite automaton. Thus, the regular expressions can be detected by finite automata. First, we introduce a conversion from a regular expression into a finite automaton. Then, we show the realization of the finite automaton on the FPGA.

[3] The XML filter [31] requires these meta characters. To realize them, additional hardware is necessary. For example, [12] built a single deterministic push down automata using a lazy approach, and realized them by a sequence of LCs with a stack.

3.1. Definitions of DFA

Definition 3.1. *A* **deterministic finite automaton (DFA)** *is defined by a five-tuple* $M_{DFA} = (S, \Sigma, \delta, s_0, A)$*, where* $S = \{s_0, s_1, \ldots, s_{q-1}\}$ *is a finite set of the states;* Σ *is a finite set of the input characters;* δ *is a transition function* ($\delta : S \times \Sigma \to S$)*;* $s_0 \in S$ *is the initial state; and* $A \subseteq S$ *is the set of accept states. Since our system treats ASCII characters, we assume that* $|\Sigma| = 2^8 = 256$.

Definition 3.2. *Let* $s \in S$*, and* $c \in \Sigma$*. If* $\delta(s, c) \in S$*, then* c *is* **a transition character from state** s **to state** $\delta(s, c)$.

To define **a transition string** accepted by the DFA, we extend the transition function δ to $\hat{\delta}$.

Definition 3.3. *Let* Σ^+ *be a set of strings, and* $\hat{\delta} : S \times \Sigma^+ \to S$ *be the extended transition function. If* $C \subseteq \Sigma^+$ *and* $s \in S$*, then* $\hat{\delta}(s, C)$ *represents a transition state of* s *with respect to the input string* C.

Definition 3.4. *Consider a DFA,* $M_{DFA} = (S, \Sigma, \delta, s_0, A)$*. Let* $C_{in} \subseteq \Sigma^+$*. Then,* M_{DFA} *accepts a string* C_{in}*, if the following relation holds:*

$$\hat{\delta}(s_0, C_{in}) \quad \in \quad A. \tag{1}$$

Let c_i be a character of a string $C = c_0 c_1 \cdots c_n$, and δ be a transition function. Then, the extended transition function $\hat{\delta}$ is defined recursively as follows:

$$\hat{\delta}(s_0, C) \quad = \quad \hat{\delta}(\delta(s_0, c_0), c_1 c_2 \cdots c_n). \tag{2}$$

By using (1) and (2), the DFA performs the string matching by repeating state transitions.

3.2. Aho-Corasic DFA (AC-DFA)

For some DFAs, **backtrackings** are necessary to detect multiple regular expressions. Thus, the matching speed tend to be slow.

Example 3.1. *Fig. 4 shows a DFA accepting two strings "ABCD" and "BCAB". In this DFA,* s_0 *denotes the initial state, and* s_4 *and* s_8 *denote the accept states. Note that, when the mismatch occurs, it backs to the initial state.*

When the text is "ABCAB", first, it goes to s_1 *by consuming "A" (Fig. 4 (a)). Second, it goes to* s_2 *by consuming "B" (Fig. 4 (b)). Third, it goes to* s_3 *by consuming "C" (Fig. 4 (c)). Fourth, for the input "A", since a mismatch occurs, it backs to the initial state. Then, it goes to* s_1 *by consuming "A" (Fig. 4 (d)). Finally, although it goes to* s_2 *by consuming "B", it cannot detect "BCAB" (Fig. 4 (b)).*

To detect "BCAB", it must perform the backtracking in Fig. 4 (c). In this case, it also must back to the text pointer to the second "B". Then, it goes to s_5 *by consuming "B" (Fig. 4 (f)). By performing state transitions repeatedly (Fig. 4 (g)-(h)), it goes to the accept state* s_8 *(Fig. 4 (i)).* ■

Table 1: Perl compatible regular expressions (PCREs) considered in this paper.

Expression	Meaning	Example
Character		
(ASCII character)	Match a single character	
\x(Hexadecimal number)	Match a single character	
Meta character (R denotes a regular expression, and ϕ denotes an empty character)		
.	Match any single character except newline (\n)	.={a,b,...,z,A,B,...,Z,0,1,...,9}
R*	Repeat R zero or more times (Kleene closure)	A*={ϕ,A,AA,AAA,...}
R+	Repeat R one or more times	A+=AA*={A,AA,AAA,...}
R?	Repeat R zero or one times	A?={ϕ,A}
R{α} R{α,} R{α,β}	Repeat R α times Repeat R α or more times Repeat R at least α and at most β times	
R_1|R_2	R_1 or R_2 (union)	
(R)	Groups regular expressions, so operators can be applied	
\(meta character)	Match a meta character as a character	\? matches "?"
[(characters)]	Set of characters	[abc] = (a|b|c)
[^(characters)]	Complement set of characters	
^R R$	Matching start from the first character Matching ends at the last character	
R_1(?=R_2) R_1(?!R_2) (?<=R_1)R_2 (?<!R_1)R_2	Lookahead (Continue matching when R_2 matches after R_1) Negation of lookahead (Continue matching when R_2 does not match after R_1) Lookbehind (Continue matching when R_1 matches before R_2) Negation of lookbehind (Continue matching when R_1 does not match before R_2)	
SNORT shorthand character class		
\d	A set of numbers (=[0-9])	
\D	Complement set of \d (=[^0-9])	
\f	Form feed	
\n	Line feed	
\r	Carriage return	
\t	Horizontal Tab	
\v	Vertical Tab	
\s	A set of white spaces (=[\f\n\r\t\v])	
\S	Complement set of \s	
\w	A set of alphabets (=[A-Za-z0-9_])	
\W	Complement set of \w	
Flag		
\i	Case insensitive (matches both lowercase and uppercase characters)	
\m	^ and $ match after and before newlines	

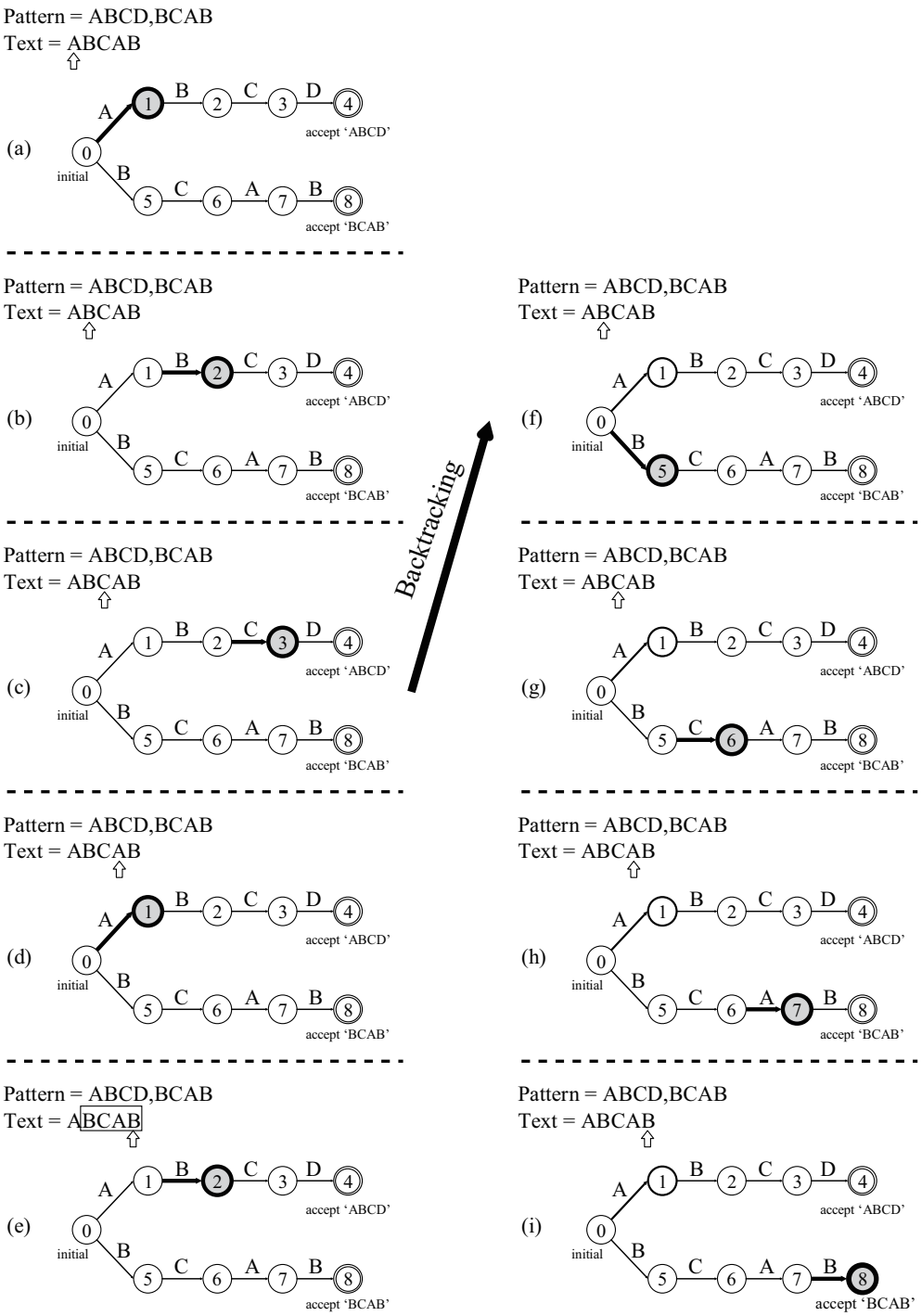


Figure 4: An example of backtracking.

To detect multiple patterns without performing backtracking, **the Aho-Corasick DFA (AC-DFA)** has been proposed [1]. To construct the AC-DFA, first, the transition strings are represented by a text tree (Trie). Next, **failure paths** that indicate the transitions for the mismatches are attached to the text tree. Since the AC-DFA stores all possible paths for all the patterns, no backtracking is necessary. By scanning the input text only once, the AC-DFA detects all matched strings.

Example 3.2. *In Fig. 4, by attaching the failure path from s_3 to s_7, we have the DFA shown in Fig. 5 that detects both "ABCD" and "BCAB" without the backtracking.*

First, the DFA goes to s_1 by consuming "A" (Fig. 5 (a)), second, it goes to s_2 by consuming "B" (Fig. 5 (b)), third, it goes to s_3 by consuming "C" (Fig. 5 (c)). In this case, it goes to s_7 by consuming "A" through the failure path (Fig. 5 (d)). Finally, it goes to the accept state s_8 by consuming "B" (Fig. 5 (e)). ■

3.3. Realization of AC-DFA

Fig. 6 shows the AC-DFA machine, where the register stores the present state and the memory stores **a state transition table** for δ. Let $q = |S|$ be the number of the states, and $n = |\Sigma|$ be the number of different characters in Σ. Then, the amount of memory to implement the DFA is $\lceil log_2 q \rceil 2^{\lceil log_2 n \rceil + \lceil log_2 q \rceil}$ bits [4]. In fact, the sizes of the memories for the AC-DFAs increase exponentially [34]. Thus, a direct hardware realization of the regular expression matching using an AC-DFA is often impractical.

3.4. Definitions of NFA

Definition 3.5. *A* **non-deterministic finite automaton (NFA)** *is defined by a five-tuple $M_{NFA} = (S, \Sigma, \gamma, s_0, A)$, where S, Σ, s_0, and A are the same as Definition 3.1, while the transition function $\gamma : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$ is different. Note that, ε denotes an empty character, and $P(S)$ denotes the power set of S.*

In an NFA, the empty (ε) input is permitted. Thus, a state for the NFA can transit to multiple states for a single input. The state transition with the ε input is **an ε transition**. In this paper, in a state transition diagram, an ε symbol with an arrow denotes the ε transition.

Example 3.3. *Fig. 7 shows the NFA for the regular expression "A+[AB]{3}D". An ε transition exists from the second state to the first state.* ■

Example 3.4. *Fig. 8 illustrates the state transitions for the NFA in Fig. 7 when the input string is "ABABD". Note that, multiple state transitions occur in certain rows, since the NFA can be in multiple states given the input string "ABABD". There is at least one path from the initial state s_0 to the accept state s_5. Thus, "ABABD" is accepted by this NFA.* ■

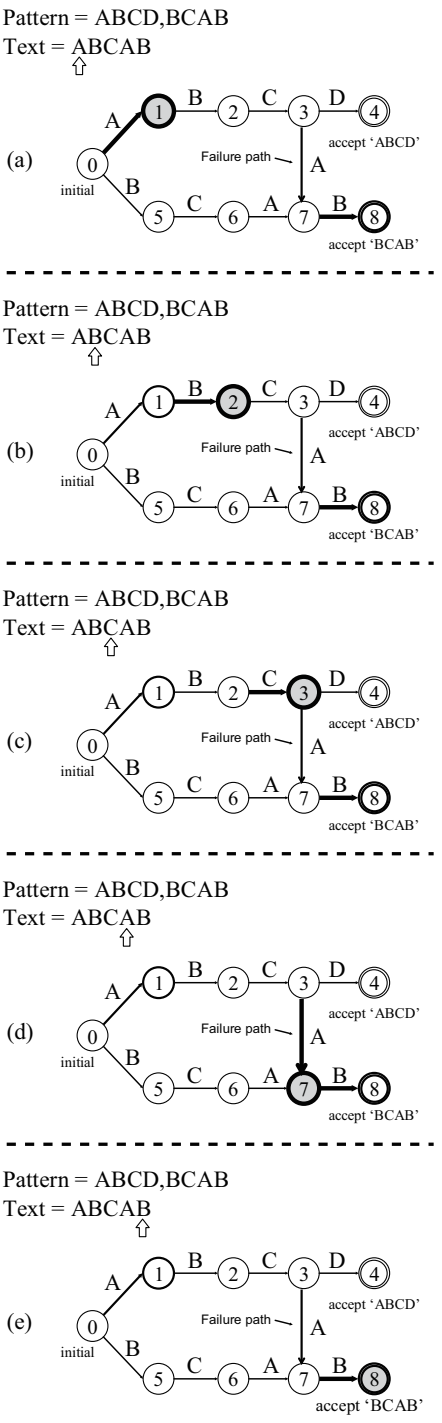


Figure 5: An example of AC-DFA.

[4] Since the size of the register in the DFA machine is much smaller than that for the memory storing the transition function, we ignore the size of the register.

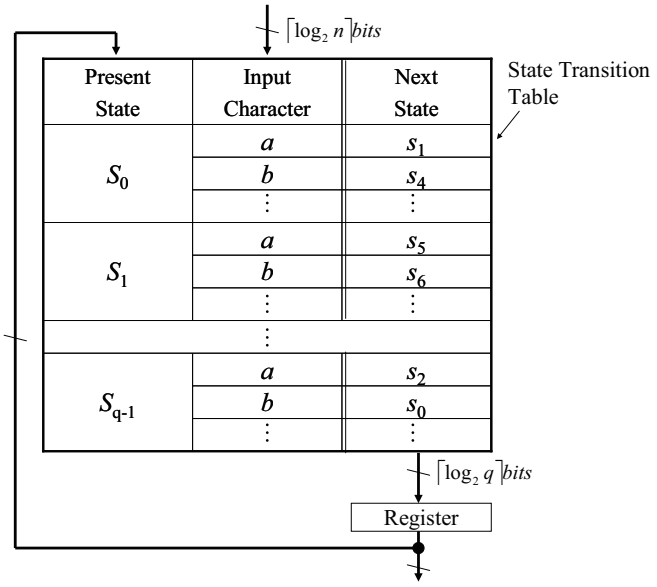


Figure 6: AC-DFA machine.

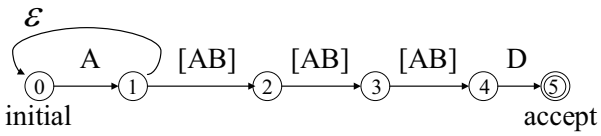


Figure 7: NFA for the regular expression "A+[AB]{3}D".

3.5. Conversion of a Regular Expression into the NFA

Several methods exist to convert a regular expression into an NFA. This paper uses **the Modified McNaughton-Yamada method** [11] that is suitable for hardware realization. The modified McNaughton-Yamada method is shown in Fig. 9, where R denotes the regular expression, and ε denotes the ε transition. By applying the Modified McNaughton-Yamada method to the regular expression repeatedly, we have an NFA.

Example 3.5. *Fig. 10 illustrates the conversion of the regular expression "A+[AB]{3}D" into the NFA by using the Modified McNaughton-Yamada method.* ■

3.6. Realization of NFA

Sidhu and Prasanna [25] proposed the realization of the regular expression matching circuit based on an NFA. In their circuit, each state is implemented by a cascade of LCs on an FPGA. Thus, the necessary number of LCs increases with the number of states. The conversion from the NFA to the circuit is shown in Fig. 11, where R denotes the regular expression. Note that, repetitions ($R\{\alpha\}$ and $R\{\alpha,\}$) can be realized by circuits shown in Figs. 11 (1) and 11 (3).

Example 3.6. *Fig. 13 shows a circuit for the NFA shown in Fig. 7.* ■

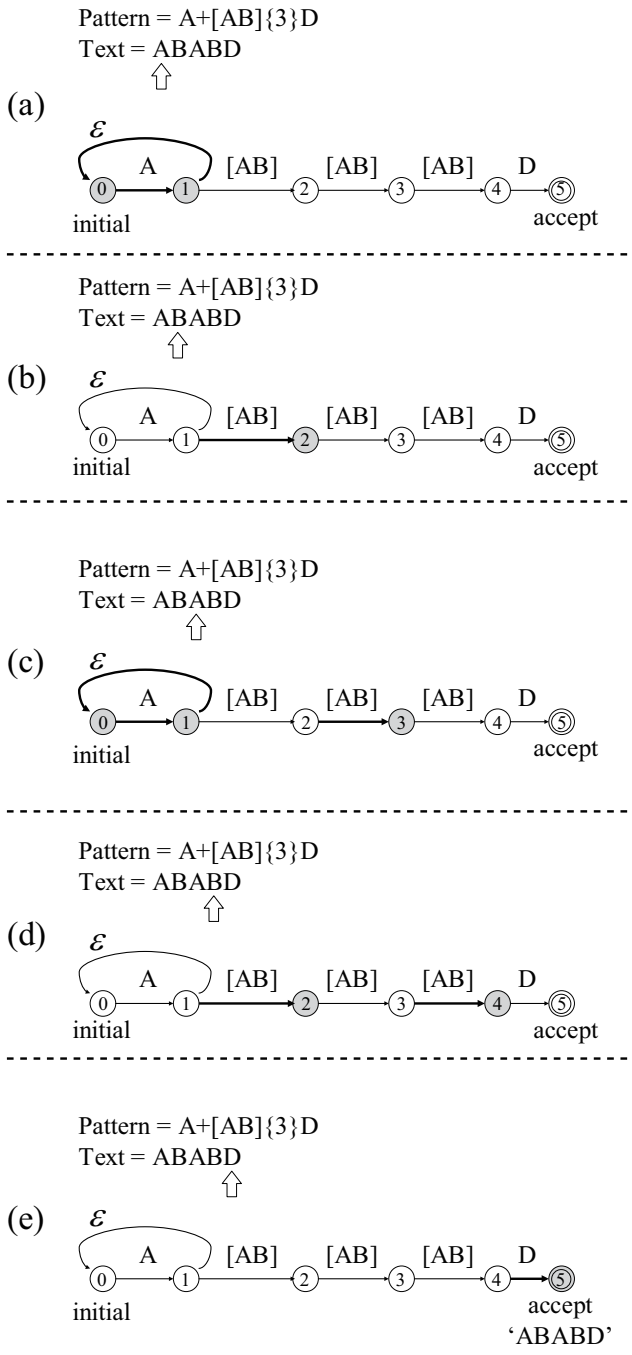


Figure 8: NFA shown in Fig. 7 accepts "ABABD".

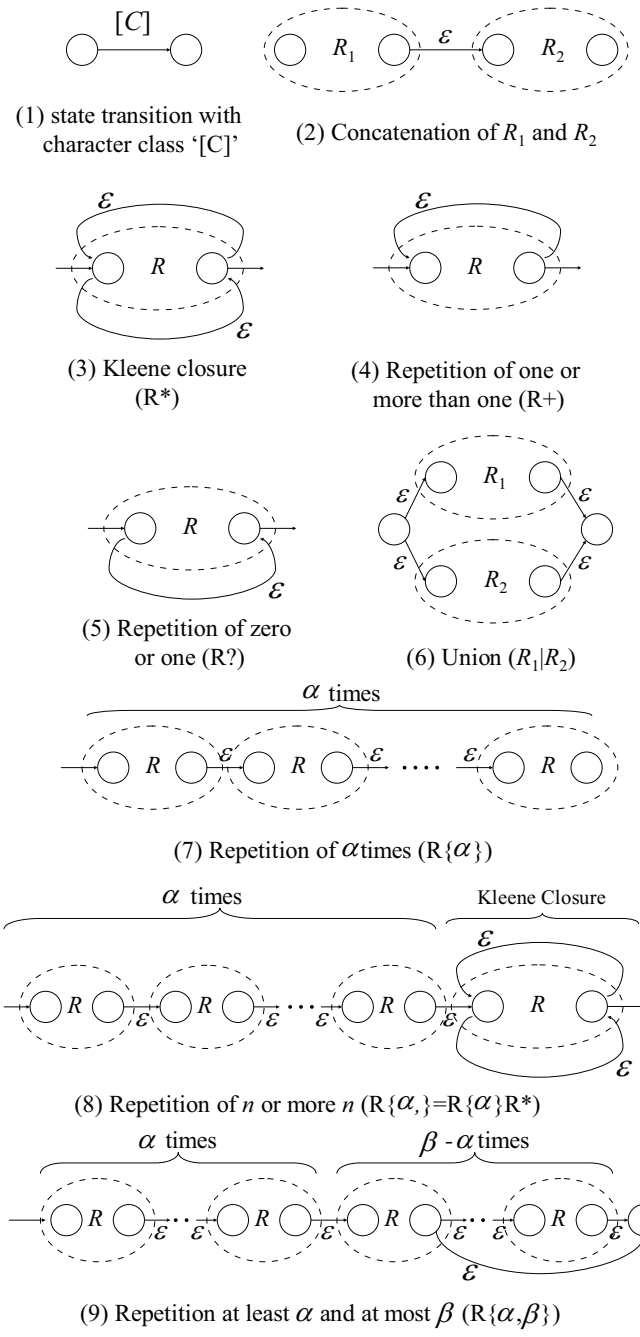


Figure 9: Modified McNaughton-Yamada Method.

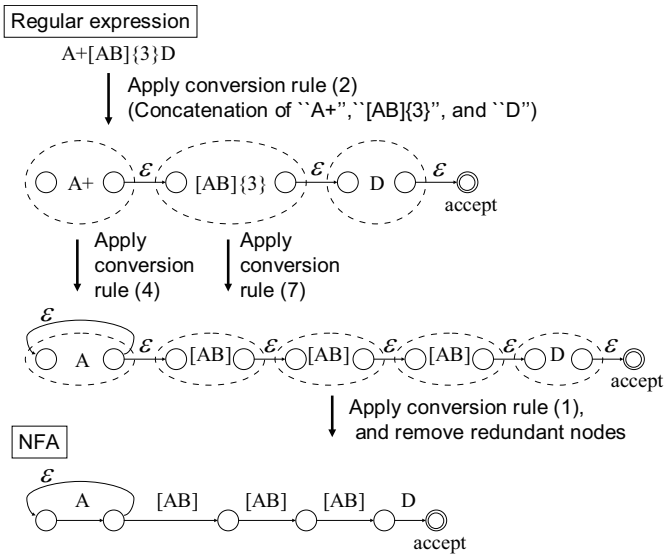


Figure 10: An example of NFA conversion by modified McNaughton-Yamada method.

3.7. Realization of Extended Regular Expressions

In this part, we briefly show the realization of extended regular expressions, which were not implemented in the previous works [18, 19, 20, 21].

3.7.1. SNORT Shorthand Character Class

The SNORT shorthand character class can be represented by a meta character "[]" (set of characters) or "[^]" (complement set of characters). Thus, the circuit shown in Fig. 11 (1) can realize a SNORT shorthand character class.

Example 3.7. *A SNORT shorthand character class "\s" can be written by "[\f\n\r\t\v]". Fig. 12 shows a circuit for SNORT shorthand character class "\s".* ■

3.7.2. Flag [5]

A case insensitive flag (\i) can be written by the set of lowercase and uppercase characters. Thus, it can be realized by the circuit for the character class shown in Fig. 11 (1). To realize the flag (\m) considering the newline, a small sequencer is attached to the regular expression matching circuit. When it detects the newline character (\n), it checks whether the matching is succeeded or not.

3.7.3. Lookahead and Lookbehind [5]

In a hardware implementation, we ignore any software related features (lookahead and lookbehind) by rewriting regular expressions. For a lookahead (lookbehind) meta character, we remove it. On the other hand, for a negative one, we remove it, but insert an inverter into the outputs for negative one.

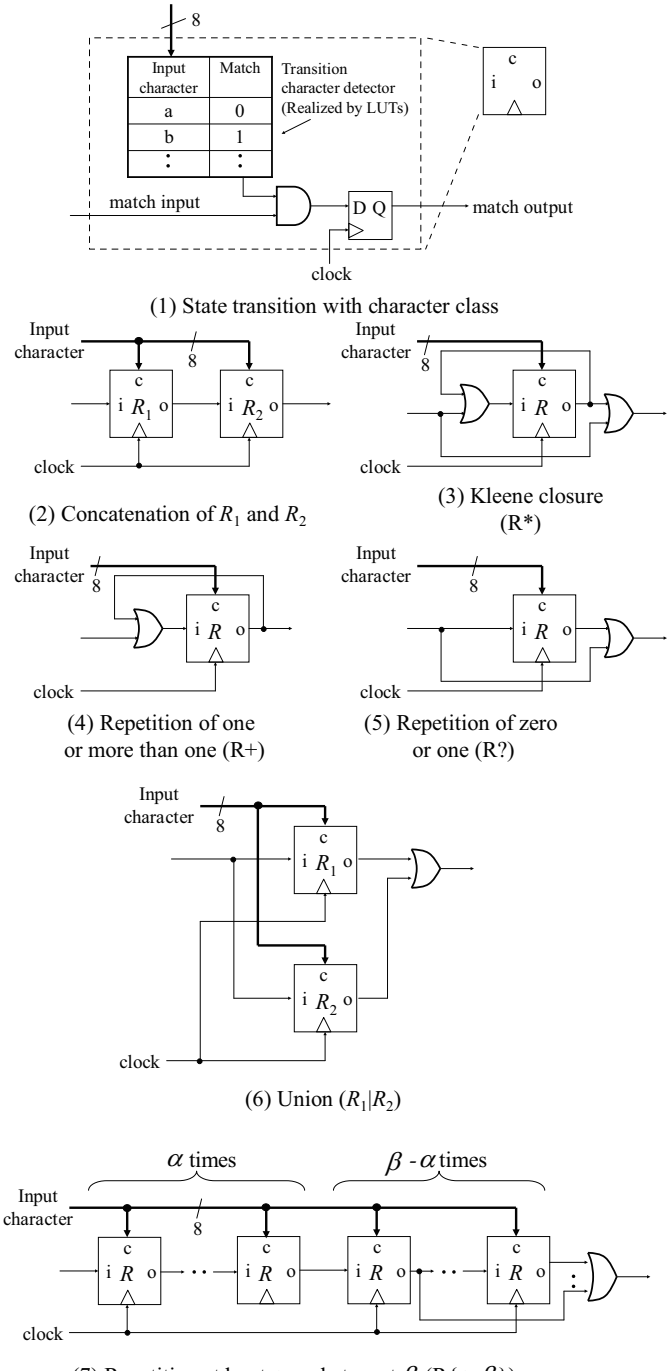


Figure 11: Circuit conversion from the NFA.

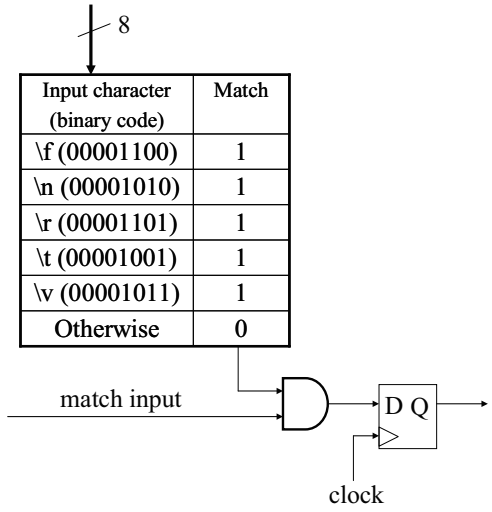


Figure 12: Circuit for the SNORT shorthand character class "\s ".

4. Area Reduction Techniques for Regular Expression Matching Circuits

4.1. Sharing of Transition Character Detector

Sidhu and Prasanna [27] realized transition character detectors by LUTs in a straightforward manner. However, by sharing them, the necessary number of LUTs can be reduced. Sourdis and Pnevmatikatos [27] realized the character detectors by a content addressable memory (CAM). Since the CAM is expensive and dissipates high power [32], we realize transition character detectors by an SRAM.

Example 4.8. *In the circuit of Fig. 13, the transition character sets "[AB]" are used three times. Fig. 14 shows the regular expression circuit, where the transition character detector is implemented by a memory.* ■

As shown in Fig. 14, **a decomposed NFA** consists of **the transition character detection part** and **the state transition part**. The transition character detection part can be realized by an off-chip memory, while the state transition part can be realized by the cascade of LCs.

4.2. Prefix Sharing

Many regular expressions used in network applications share the same prefixes and/or postfixes [15]. By sharing them, the number of states for the NFA can be decreased.

Example 4.9. *Consider the NFA accepting regular expressions "ABCD" and "ABEF". Since the prefixes "AB" can be shared, we have the NFA shown in the lower part of Fig. 15. Next, consider the NFA accepting regular expressions "ABCD" and "EFCD". Since the postfixes "CD" can be shared, we have the NFA shown in the lower part of Fig. 16. As shown in these examples, a sharing a part of regular expressions will decrease the number of states.* ■

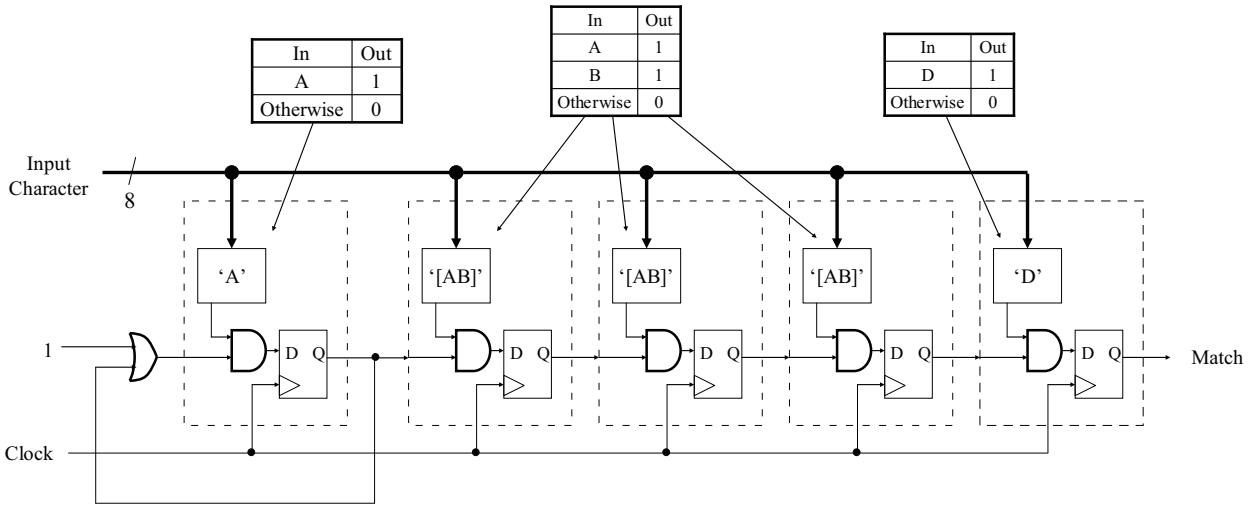


Figure 13: A circuit for the NFA [25].

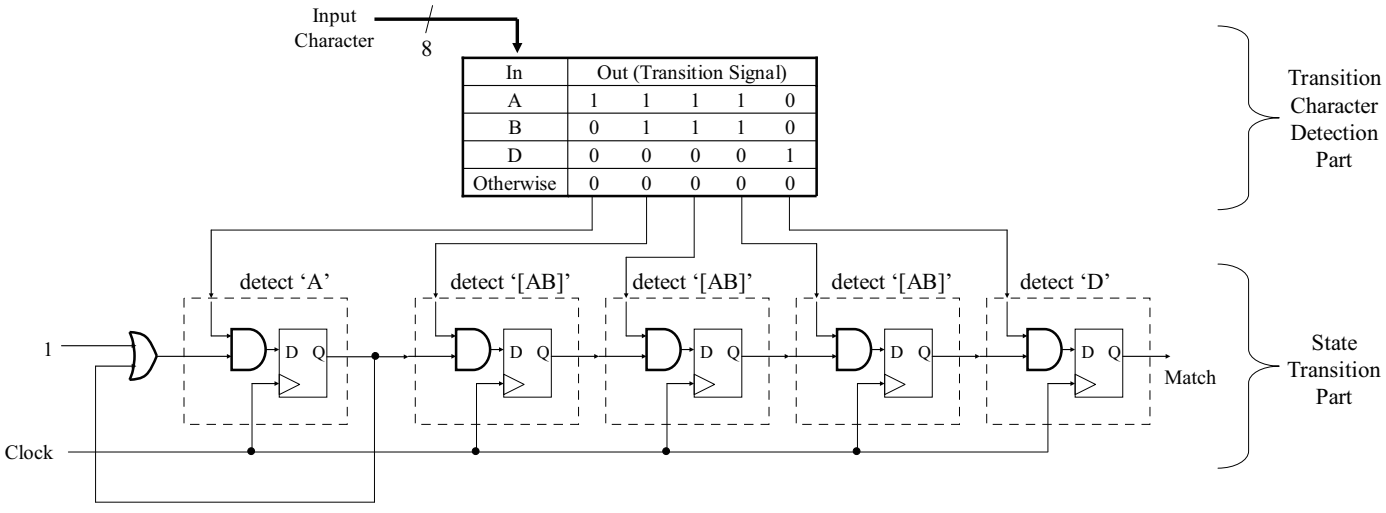


Figure 14: A circuit for the decomposed NFA [27].

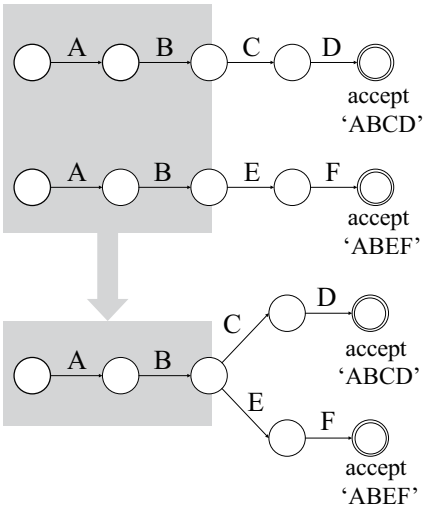


Figure 15: An example of prefix sharing.

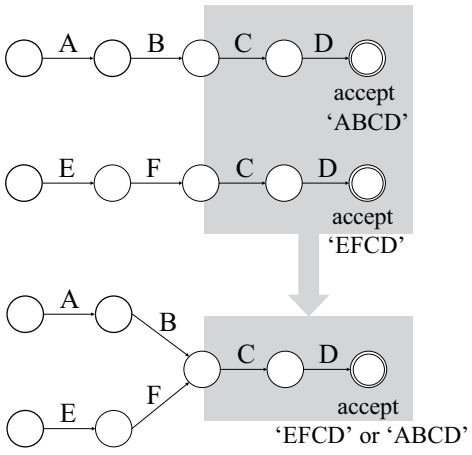


Figure 16: An example of postfix sharing.

In a circuit for a prefix shared NFA, matched regular expressions are distinguished without any additional hardware. However, in a circuit for a postfix shared NFA, an additional hardware (OR gates, AND gates, shift registers) is necessary to distinguish matched regular expressions [15].

Example 4.10. *Fig. 17 shows the regular expression matching circuit for the prefix shared NFA shown in Fig. 15.* ■

Example 4.11. *Fig. 18 shows the regular expression matching circuit for the postfix shared NFA shown in Fig. 16. To distinguish matched regular expressions, the following hardware is necessary: An OR gate (Fig. 18 (1)); shift registers (Fig. 18 (2)) that retains matched states for each regular expression; AND gates (Fig. 18 (3)) that produce the match signals.* ■

As shown in Example 4.11, although the postfix sharing reduces the number of states, it requires additional hardware to implement. Thus, in this paper, we do not share the postfix in the NFA.

5. Regular Expression Matching Circuit Based on NFA with String Transition

5.1. MNFAU

Sidhu and Prasanna [25] implemented a regular expression matching by an FPGA. However, they did not use embedded memory. Since a modern FPGA consists of LCs and embedded memories, their method wastes existing resources. Each state of the NFA is implemented by an LC of an FPGA. Thus, the necessary number of LCs increases with the number of states. To reduce the number of states, we propose a regular expression matching circuit based on a **modular non-deterministic finite automaton with unbounded string transition (MNFAU)**. To convert an NFA into an equivalent MNFAU, we merge a sequence of states. To retain the equivalence between the NFA and the MNFAU, the states are merged as follows:

Lemma 5.1. *Let $S = \{s_0, s_1, \ldots, s_{q-1}\}$ be the set of states for the NFA. Assume that a subset $S' = \{s_k, s_{k+1}, \ldots, s_{k+p-1}\} \subseteq S$, for $k \leq i \leq k+p-2$, s_i goes to s_{i+1} only. Then, S' is merged into one state of the MNFAU only if both in-degree and out-degree for s_i ($k \leq i \leq k+p-1$) are one.*

Definition 5.6. *Let $c_j \in \Sigma$ be the transition character of s_j for $j = k, k+1, \ldots, k+p$, then $C = c_k c_{k+1} \cdots c_{k+p}$ denotes* **a transition string of** S_M.

In this case, a set of states $\{s_k, s_{k+1}, \ldots, s_{k+p}\}$ of an NFA is merged into a state S_M of an MNFAU.

Example 5.12. *In the NFA shown in Fig. 7, the set of states $\{s_2, s_3, s_4, s_5\}$ can be merged into a state of the MNFAU. However, the set of states $\{s_1, s_2\}$ cannot be merged, since $e_1 \neq 0$.* ■

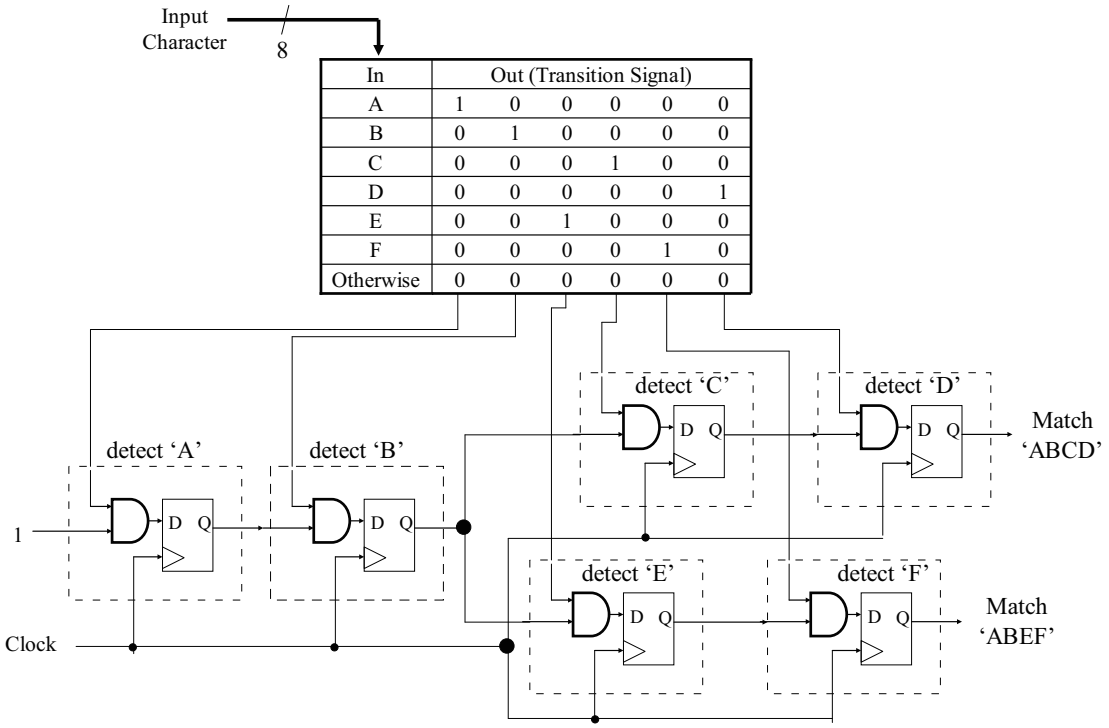


Figure 17: A circuit for the prefix shared NFA shown in Fig. 15.

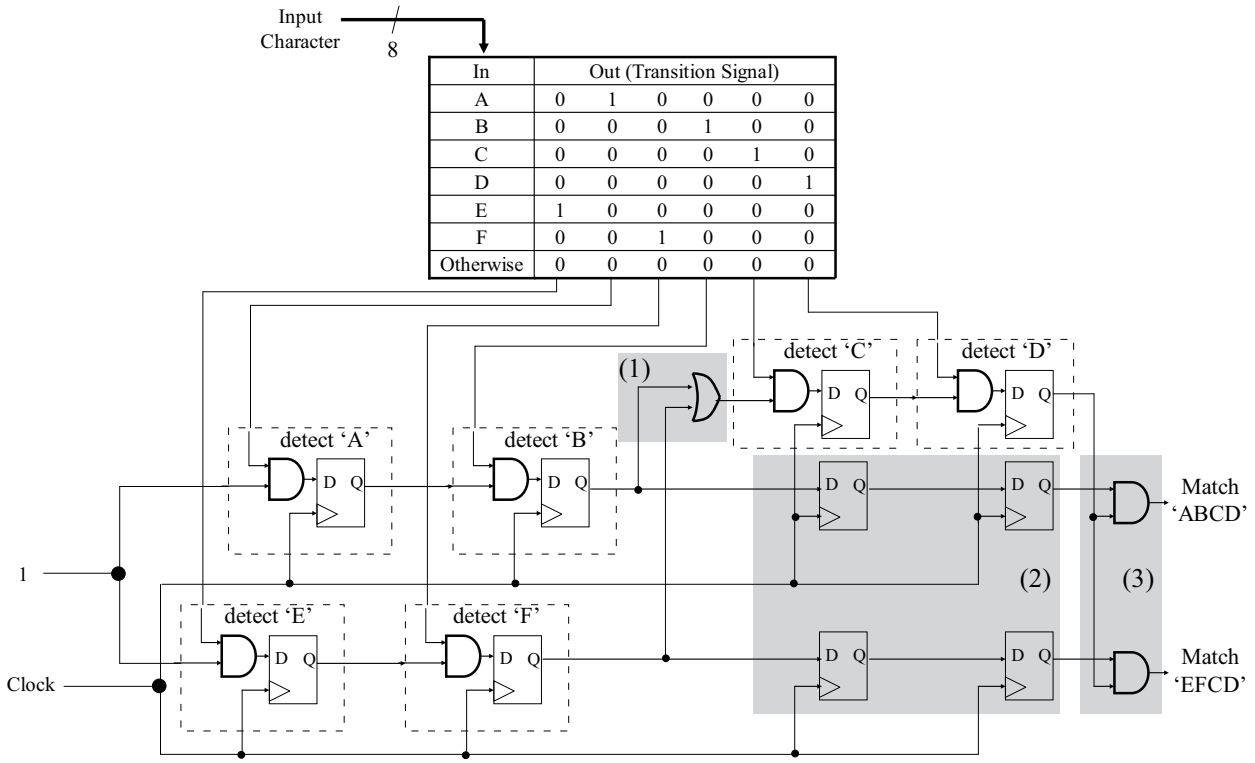


Figure 18: A circuit for the postfix shared NFA shown in Fig. 16.

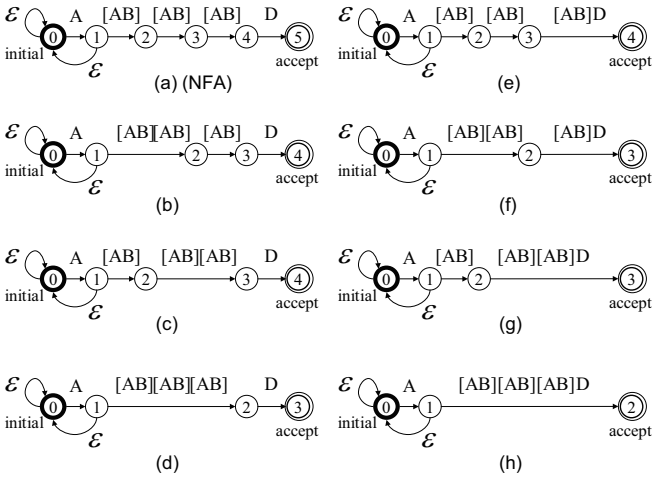


Figure 19: Possible MNFAUs derived from the NFA shown in Fig. 7.

Example 5.13. *Fig. 19 shows different MNFAUs derived from the NFA shown in Fig. 7. In the NFA, since the numbers of ε transitions inputs and outputs for three states $\{s_2, s_3, s_4\}$ are $e_i = 0$, the number of different MNFAUs are eight. In Fig. 19, the MNFAU (h) is the most compact MNFAU.* ■

As shown in Example 5.13, we can convert the given NFA into a compact MNFAU. However, there exists the restriction of the hardware realization. The next Section 5.3 shows the hardware realization for the MNFAU, and Section 6 shows the design method for the MNFAU.

5.2. Decomposition of MNFAU

Consider the MNFAU in Fig. 20. It can be decomposed into **a transition string detection circuit** and **a state transition circuit**. Two circuits are connected by $q - 1$ **transition string detections signals**. Note that, in Fig. 20, q denotes the number of states for the MNFAU, and C_i denotes the transition string for i-th state of the MNFAU. Since transition strings do not include meta characters [5], they are detected by **exact matching**. Exact matching is a subclass of regular expression matching, and the AC-DFA for it can be realized by memory and a state register. On the other hand, the state transition part treating the ε transition is implemented by the cascade of logic cells shown in Fig. 11 (1) and interconnections.

5.3. Circuit Realization of Decomposed MNFAU

5.3.1. Transition String Detection Circuit

The lengths for the transition strings for the MNFAU are different in general. To detect multiple strings with different lengths, we use the AC-DFA [1] shown in Fig. 6. Since this part tend to be large, the state transition table is implemented by an off-chip memory. However, the state register is implemented by the FPGA.

[5] However, meta characters "[]" can be used.

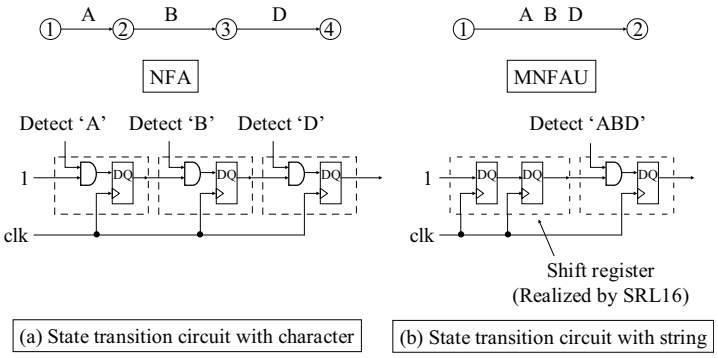


Figure 21: State transition circuits for NFA and MNFAU.

5.3.2. State Transition Circuit for the MNFAU [18].

In an NFA, each state is realized by a small machine consisting of a flip-flop and an AND gate. The right-hand side of Fig. 21 shows the state transition circuit for the MNFAU that accepts "ABD". When the AC-DFA detects the transition string ("ABD" in Fig. 21), a detection signal is sent to the state transition circuit. Then, the state transition is performed. The AC-DFA scans a character in every clock, while the state transition circuit has to wait for p clocks to perform the state transition, where p denotes the length of the transition string. Thus, a $(p - 1)$-bit shift register is inserted between small machines to synchronize with the AC-DFA. As shown in Fig. 3, a four-input LUT of a Xilinx FPGA can also be used as a shift register with up to 16 bits (SRL16). With the SRL16, we can reduce the necessary number of LUTs and flip-flops.

Example 5.14. *Fig. 21 (a) shows a state transition circuit with a character for the NFA accepting "ABD", while Fig. 21 (b) shows a state transition circuit with a string for the MNFAU accepting "ABD". In the circuit of the NFA, three AND gates and flip-flops are used, thus, the total number of LCs is three. On the other hand, in the circuit of the MNFAU, one AND gate, one flip-flop, and one shift register are used. We can realize a shift register by a SRL16. Thus, the total number of necessary LCs is two.* ■

Example 5.15. *Fig. 22 shows an example of operations for an NFA and an MNFAU.*

The lefthand side column of Fig. 22 illustrates the operation of the circuit for the NFA shown in Fig. 21. First, the transition character detection circuit reads "A", and sends a transition character detection signal to the first flip-flop. Then, the first flip-flop is activated (Fig. 21 (a)). Second, it reads "B", and second flip-flop is activated (Fig. 21 (b)). Finally, it reads "D", and the last flip-flop is activated (Fig. 21 (c)). In this way, the circuit accepts "ABD".

The righthand side column of Fig. 22 illustrates the operation of the circuit for the MNFAU shown in Fig. 21. First, the transition string detection circuit reads "A". Also, the first flip-flop is activated (Fig. 21 (a)). Second, it reads "B", and also the second flip-flop is activated (Fig. 21 (b)). Finally, the transition string detection circuit sends the detection signal to the

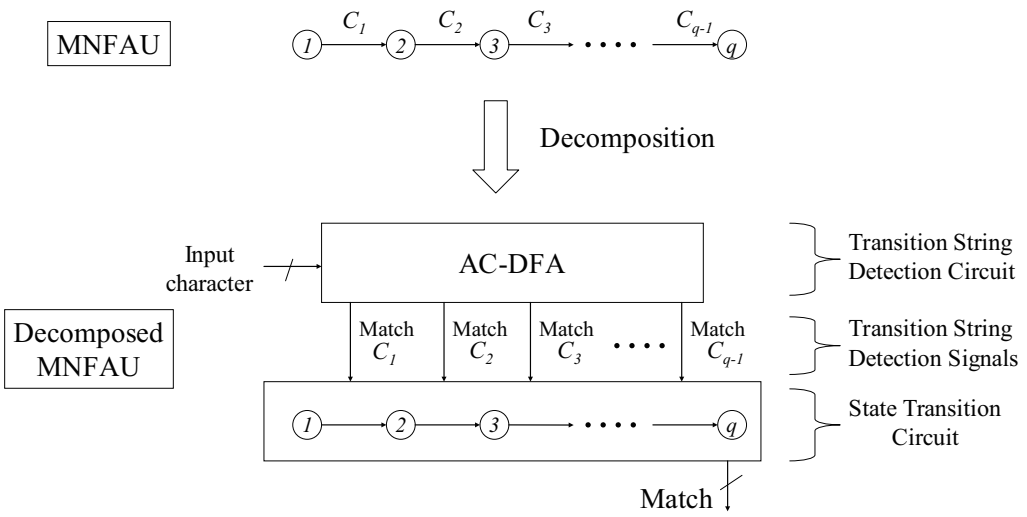


Figure 20: Decomposed MNFAU.

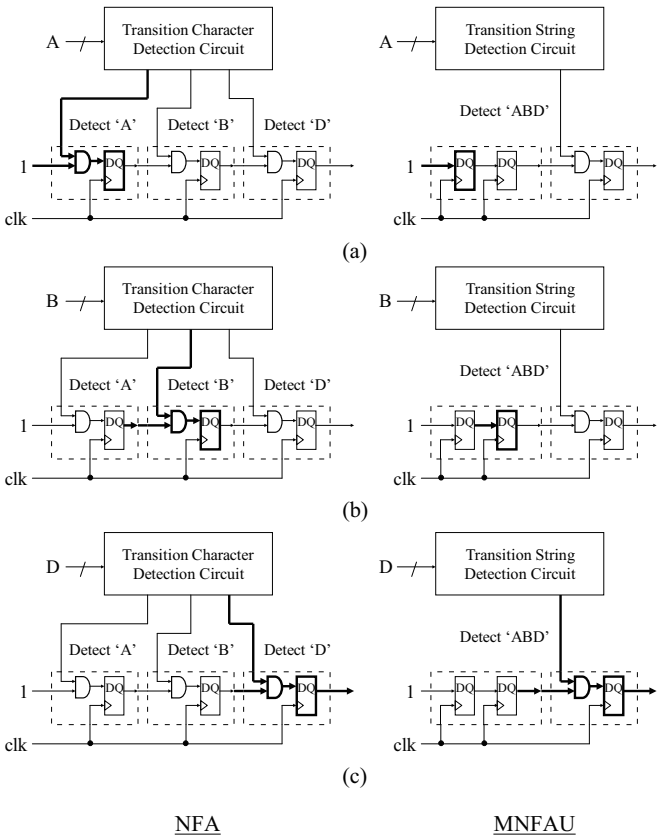


Figure 22: Operations for an NFA and an MNFAU.

last flip-flop. Then, it is activated (Fig. 21 (c)). In this way, the circuit accepts "ABD". ■

5.3.3. Realization of Decomposed MNFAU

Fig. 23 shows the circuit for the decomposed MNFAU consisting of the transition string detection part and the state transition part. The transition string part is realized by the AC-DFA machine shown in Fig. 6, while the state transition part is realized by the cascade of the LCs shown in Fig. 21.

Let R_i $(1 \leq i \leq r)$ be r regular expressions, $|S_{MNFAU_i}|$ be the number of states for the $MNFAU_i$ representing R_i, and $q_{MNFAU} = \sum_{i=1}^{r} |S_{MNFAU_i}|$ be the total number of states in the MNFAU. In Fig.23, the transition string detection circuit (AC-DFA machine) sends q_{MNFAU}**transition string detection signals** to the state transition circuit (a cascade of LCs). Thus, the number of outputs of the AC-DFA machine would be q_{MNFAU}. If all the outputs for the AC-DFA machine was implemented by an off-chip memory, then a large number of I/O pins [6] would be necessary. In this case, the use of an FPGA is impractical.

To reduce the number of I/O pins, we use the decoder that converts state numbers ($\lceil log_2 q_{AC-DFA} \rceil$ bits) of the AC-DFA into transition string detection signals (q_{MNFAU} bits), where q_{AC-DFA} denotes the number of states for the AC-DFA. The decoder is realized by the BRAM on the FPGA. In this way, the number of pins for the FPGA is reduced from q_{MNFAU} to $\lceil log_2 q_{AC-DFA} \rceil$.

Example 5.16. *Fig. 24 shows the regular expression matching circuit for the MNFAU representing the regular expression*

[6] Our experiment shows that the present version of SNORT requires a memory with $q_{MNFAU} = 10,066$.

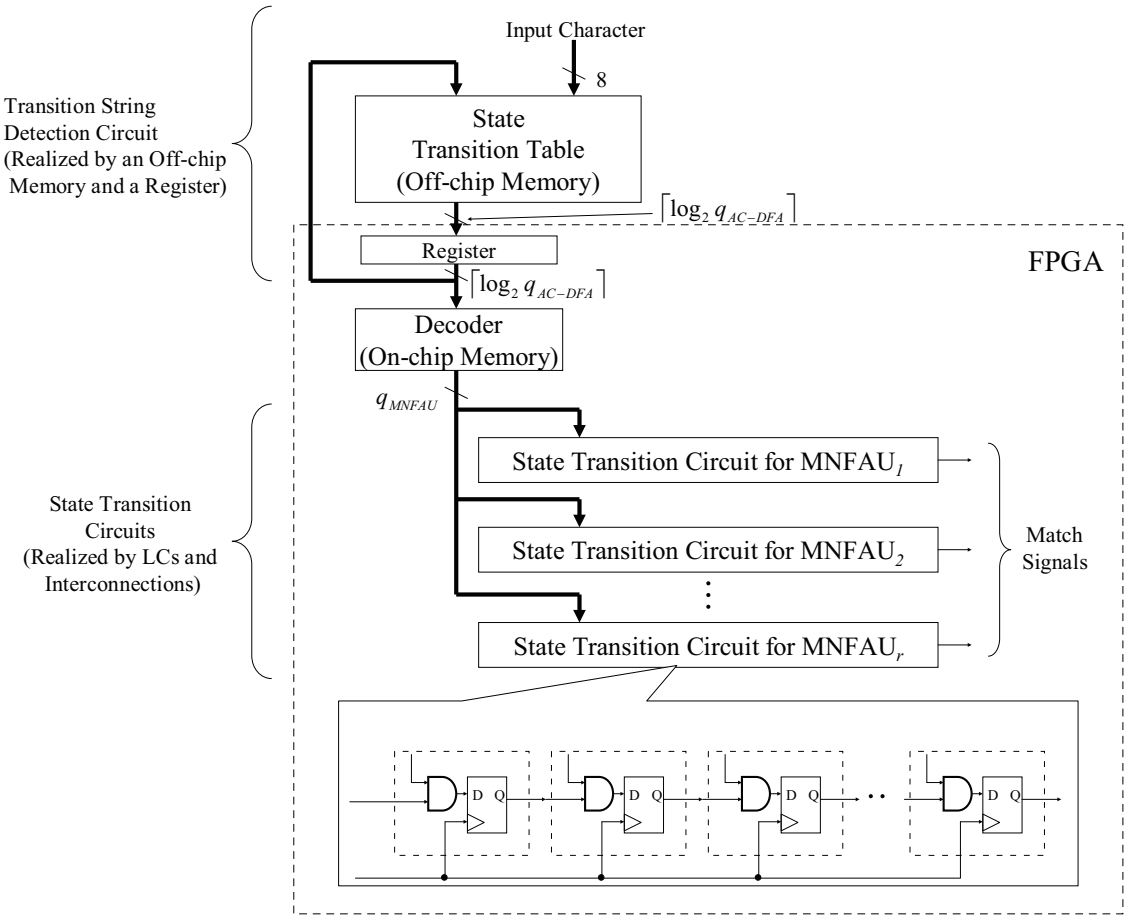


Figure 23: Realization of a decomposed MNFAU.

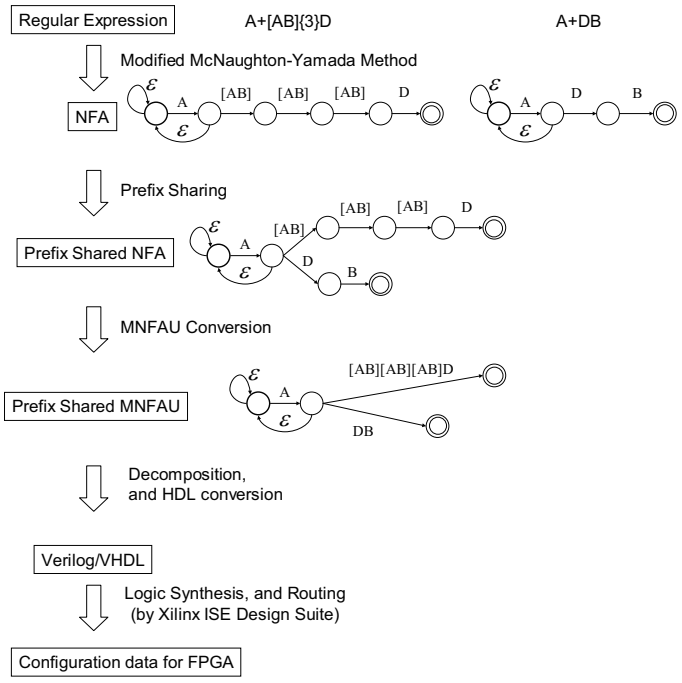


Figure 25: Design flow for regular expression matching circuit.

"A+[AB]{3}D". In this case, the number of states in the AC-DFA q_{AC-DFA} is eight. On the other hand, the number of string detection signal q_{MNFAU} is two. ■

6. Design of a Decomposed Regular Expression Matching Circuit

6.1. Design Flow

Fig. 25 shows a design flow for regular expression matching circuit based on a decomposed MNFAU. First, given regular expressions are converted into NFAs by using the Modified McNaughton-Yamada method. Second, states corresponding to the prefixes in regular expressions are shared. An algorithm for prefix sharing is shown in Section 6.2. Third, the prefix shared NFA is converted into the prefix shared MNFAU. An algorithm for the MNFAU design is shown in Section 6.3. Fourth, the prefix shared MNFAU is decomposed into the transition string detection part and the state transition part. The transition string detection part is represented by the AC-DFA. Fifth, these circuits are converted into the HDL code, and finally, the configuration data for the FPGA is generated by Xilinx ISE Design Suite.

6.2. Algorithm to Derive Prefix Shared NFA

As shown in Example 4.9, the number of states in the NFA can be reduced by sharing common prefixes. First, we define the common prefixes for two regular expressions, and corresponding states in the NFA.

Definition 6.7. *Assume that two regular expressions R_1 and R_2 have a common prefix with length w. Let $S_1 = \{s_{(1,1)}, s_{(1,2)}, \ldots, s_{(1,q1)}\}$ be the set of states for the NFA accepting R_1; $S_2 = \{s_{(2,1)}, s_{(2,2)}, \ldots, s_{(2,q2)}\}$ be the set of states for the NFA accepting R_2; $S_{PRE_1} = \{s_{(1,1)}, s_{(1,2)}, \ldots, s_{(1,w)}\}$, $S_{PRE_1} \subseteq S_1$ be a set of states corresponding to the prefix for R_1; and $S_{PRE_2} = \{s_{(2,1)}, s_{(2,2)}, \ldots, s_{(2,w)}\}$, $S_{PRE_2} \subseteq S_2$ be a set of states corresponding to the prefix for R_2. When all incoming edges and outgoing edges for both $s_{(1,i)}$ and $s_{(2,i)}$ are the same for all i ($1 \le i \le w$), the states for S_{PRE1} and S_{PRE2} can be shared.*

The problem is to share prefixes so that the resulting NFA has the minimum number of states. Since the exhaustive search to find an optimal prefix sharing is impractical, we use a greedy method to find a near optimal prefix sharing. We define the figure of merit function to find good prefixes to share.

Definition 6.8. *Let S_1 and S_2 be sets of states defined in Definition 6.7.* **The figure of merit function for the pair** (S_1, S_2) *is*

$$M(S_1, S_2) \quad = \quad w.$$

Example 6.17. *Fig. 26 shows three NFAs for a part of file transfer protocol (ftp) rule in the SNORT. Let S_1, S_2, and S_3 be the sets of states for the NFAs shown in Fig. 26 (1-3), respectively. $M(S_1, S_2)$ is eight, while $M(S_1, S_3)$ and $M(S_2, S_3)$ are six.* ■

The following algorithm tries to share two sets of states having the longest common prefix, repeatedly.

Algorithm 6.1. *(A near optimal prefix sharing of NFAs) Assume that r regular expressions $R_1, R_2, \ldots R_r$ are given. Let $S_i = \{s_{(i,1)}, s_{(i,2)}, \ldots, s_{(i,qi)}\}$ be the set of states for the NFA accepting R_i, and $\mathcal{S}_{given} = \{S_1, S_2, \cdots, S_r\}$ be the set of S_i ($1 \le i \le r$).*

1 $\mathcal{S}_{shared} \leftarrow \phi$
2 *Among $\mathcal{S}_{given}$, select a set S_{sel} with the largest numbers of states. Then, $\mathcal{S}_{given} \leftarrow \mathcal{S}_{given} - S_{sel}$.*
  3.1 *Find a set S_j in $\mathcal{S}_{given}$ that has the largest figure of merit $M(S_j, S_{sel})$.*
  3.2 *If $M(S_j, S_{sel}) > 0$, then obtain a new set of states S_{new} by applying prefix sharing to S_{sel} and S_j. Let $S_{sel} \leftarrow S_{new}$, $\mathcal{S}_{given} \leftarrow \mathcal{S}_{given} - S_j$, and go to Step 4.*
  3.3 *If $M(S_j, S_{sel}) = 0$ for all S_j, then $\mathcal{S}_{shared} \leftarrow \mathcal{S}_{shared} \cup S_{sel}$.*
4 *If $\mathcal{S}_{given} \neq \phi$, then go to Step 2.*
5 *Terminate.*

Example 6.18. *Fig. 27 illustrates of Algorithm 6.1 applied to the NFAs accepting regular expressions for a part of ftp rules of SNORT. In Fig. 27, the first NFA (A_1 in Fig. 27) accepts "SITE\sCHMOD\s[^\n]200"; the second NFA (A_2 in Fig. 27) accepts "SITE\sCHOWN\s[^\n]100"; the third NFA (A_3 in Fig. 27) accepts "SITE\sCPWD\s[^\n]100"; and the last NFA (A_4 in Fig. 27) accepts "SITE\sEXEC".*

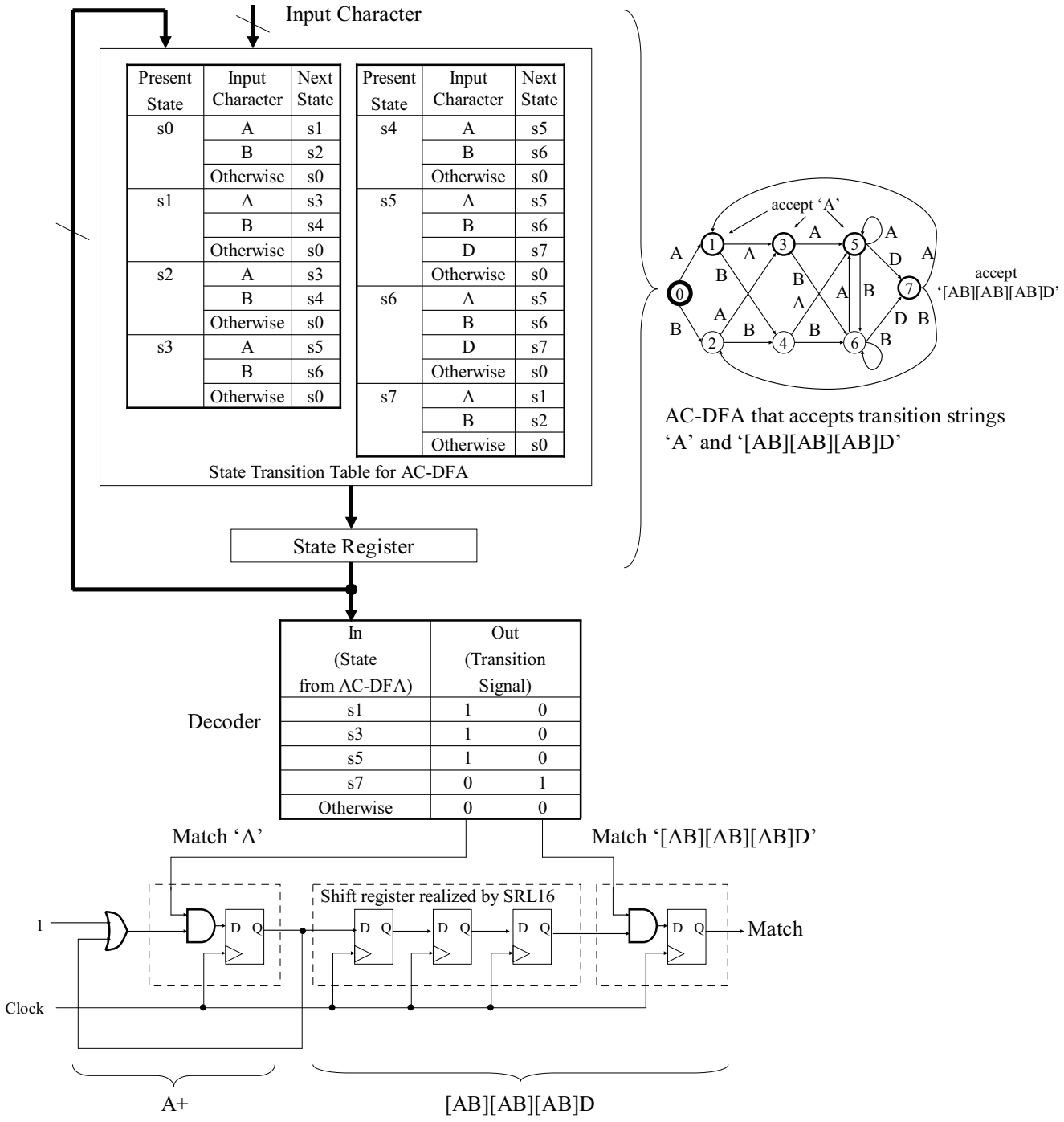


Figure 24: A circuit for regular expression "A+[AB]{3}D" based on a decomposed MNFAU.

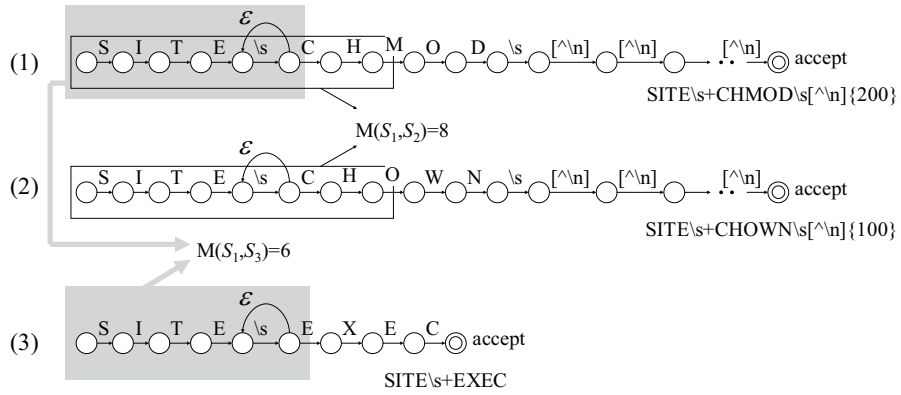


Figure 26: An example showing the figure of merit function.

1st sharing: *We select A_1 (Step 2), since A_1 has the largest number of states. Since $M(S_1, S_2) = 8$, $M(S_1, S_3) = 7$, and $M(S_1, S_4) = 6$, we choose A_2 for sharing (Step 3.1). Then, we share the states in A_1 and A_2 (Step 3.2), and obtain the prefix shared NFA (A_5 in Fig. 27).*

2nd sharing: *We select A_5 (Step 2), since A_5 has the largest number of states. Since $M(S_5, S_3) = 7$ and $M(S_5, S_4) = 6$, we choose A_3 for sharing (Step 3.1). Then, we share the states in A_5 and A_3 (Step 3.2), and obtain the prefix shared NFA (A_6 in Fig. 27).*

3rd sharing: *We select A_6 (Step 2), since A_6 has the largest number of states. Since $M(S_6, S_4) = 4$, we choose A_4 for sharing (Step 3.1). Then, we share the states in A_6 and A_4 (Step 3.2), and obtain the prefix shared NFA (A_7 in Fig. 27).*

Since all the NFAs have been shared, we terminate the algorithm. ■

6.3. Design Algorithm for the Decomposed MNFAU [21]

As shown in Example 5.13, any NFA can be converted into an MNFAU using Lemma 5.1.

The transition string detection circuit is implemented by an off-chip memory, while the state transition circuit for MNFAU is implemented by an on-chip memory of an FPGA. In order to minimize the system cost, given the size of the off-chip memory, we try to use the smallest FPGA.

The number of the states in the transition string detection circuit increases with the length of the string. On the other hand, the number of the states in the state transition circuit for the MNFAU decreases with the length of the string.

With these conditions in mind, the conversion problem from an NFA to an MNFAU is formulated as follows:

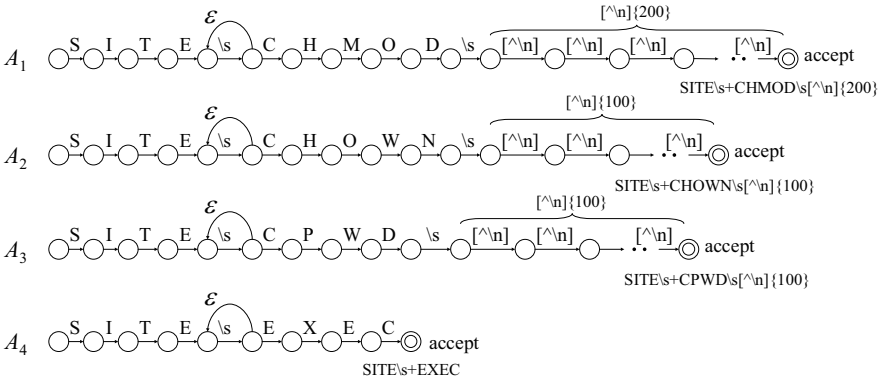
Problem 6.1. *Let $S = \{s_0, s_1, \ldots, s_{q-1}\}$ be the set of states of the NFA; $S_1 \cup S_2 \cup \cdots \cup S_u$ be a partition of S, where $S_i \subseteq S$ and $S_i \cap S_j = \phi (i \neq j)$; C_i be a transition string for a set of states S_i; $C = \{C_1, C_2, \ldots, C_u\}$ be a set of transition strings; q_{AC-DFA} be the number of states in the AC-DFA for C; $M(C) = \lceil q_{AC-DFA} \rceil 2^{8+\lceil log_2 q_{AC-DFA} \rceil}$ be the memory size of the AC-DFA for C; and $M_{off-chip}$ be the memory size for the off-chip memory. Assume that, for each $S_i = \{s_k, s_{k+1}, \ldots, s_{k+p}\}$, for $i = k, k+1, \cdots, k+p-1$, s_i goes to only s_{i+1} and both in-degree and out-degree for s_i are one. Then, find a partition S that minimizes u satisfying the memory constraint $M(C) < M_{off-chip}$, where u is the number of partitions in S.*

Let $S = \{s_0, s_1, \ldots, s_{q-1}\}$ be the set of states in the NFA, and t be the number of states for the NFA with $e_i > 0$ $(0 \leq i \leq q-1)$, where e_i be the total number of ε transition inputs and outputs in the state s_i. Then, the number of different MNFAUs is 2^{q-t-1}. Since this can be very large, an exhaustive method to find a minimum MNFAU satisfying the off-chip memory constraint $M(C) < M_{off-chip}$ is impractical. Thus, we use a greedy method to find a near minimum MNFAU.

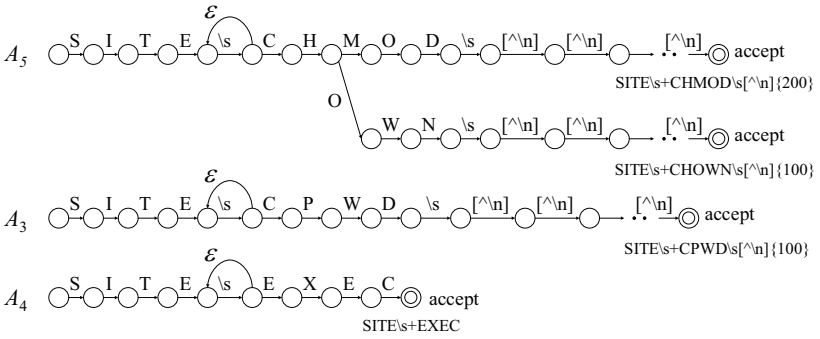
Algorithm 6.2. *(Find a near minimum MNFAU from the NFA) Let $S = \{s_0, s_1, \ldots, s_{q-1}\}$ be the set of states for the NFA, and $M_{off-chip}$ be the memory size for the off-chip memory.*

1. *Obtain a minimum partition $S = S_1 \cup S_2 \cup \cdots \cup S_u$, where $S_i \cap S_j = \phi (i \neq j)$, such that, for each $S_i = \{s_k, s_{k+1}, \ldots, s_{k+p}\}$, for $i = k, k+1, \cdots, k+p-1$, s_i goes to only s_{i+1} and both in-degree and out-degree for s_i are one. Then, obtain a set of transition strings $C = \{C_1, C_2, \ldots, C_u\}$.*
2. *Construct the AC-DFA for C. Then, obtain $M(C)$.*
3. *While $M(C) > M_{off-chip}$, perform Step 4. Otherwise go to Step. 4.*
4. *Select a set $S_i = \{s_k, s_{k+1}, \ldots s_{k+m}\}$ with the maximum number of states from S, and partition S_i into two subsets S_{i_1} and S_{i_2}, where $S_{i\_1} = \{s_k, s_{k+1}, \ldots, s_{k+\lceil \frac{m}{2} \rceil}\}$ and $S_{i\_2} = \{s_{k+\lceil \frac{m}{2} \rceil + 1} \ldots, s_{k+m}\}$. Also, obtain a set of transition strings $C = \{C_1, C_2, \ldots, C_{i\_1}, C_{i\_2}, \ldots, C_u\}$, where C_{i_1} is a transition string for S_{i_1}, and C_{i_2} is that for S_{i_2}.*
5. *Terminate the algorithm.*

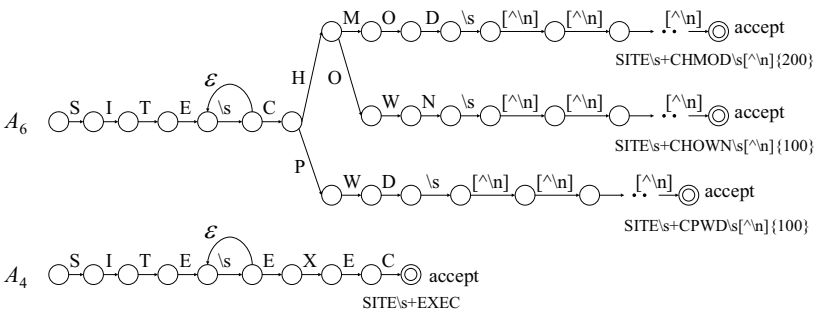
Algorithm 6.2 repeatedly bi-partitions the maximum subset of S while the memory size $M(C)$ exceed $M_{off-chip}$.



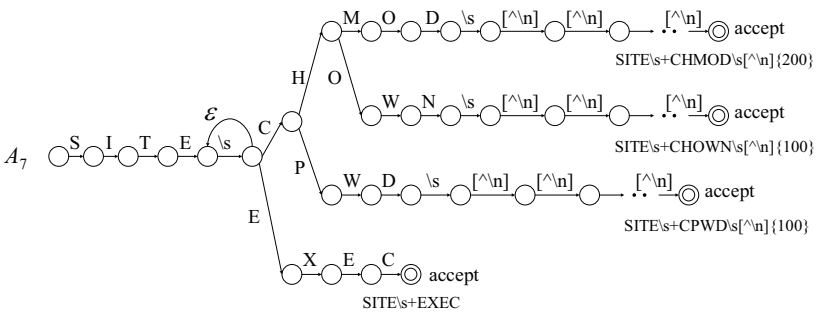
(a) Original



(b) After 1st sharing



(c) After 2nd sharing



(d) After 3rd sharing

Figure 27: An illustration of Algorithm 6.1.

Table 2: Comparison of NFA based regular expression matching circuits.

FA Type	NFA #States	AC-DFA #States	On-chip MEM (Kb)	#LC	#LC/ #Char	MEM/ #Char
NFA	661,510	0	0	690,617	1.04	0
Prefix Shared NFA	610,972	0	0	640,078	0.96	0
MNFAU	83,703	12,714	12,909	171,776	0.26	19.9
Prefix Shared MNFAU	64,387	9,443	9,930	132,136	0.20	15.3

Table 3: Comparison with other methods.

Method	FA Type	FPGA	Th (Gbps)	#LC	Off-chip MEM (Kb)	#Char	#LC/ #Char	MEM/ #Char
Pipelined DFA [6] (ISCA'06)	DFA	Virtex 2	4.0	247,000	3,456	11,126	22.22	3182.2
MPU+Bit-partitioned DFA [4] (FPL'06)	DFA	Virtex 4	1.4	N/A	6,000	16,715	N/A	367.5
Improvement of Sidhu-Prasanna method [5] (FPT'06)	NFA	Virtex 4	2.9	25,074	0	19,580	1.28	0
MNFA(3) [19] (SASIMI'10)	MNFA(p)	Virtex 6	3.2	4,717	441	12,095	0.39	37.3
MNFAU *without* prefix sharing [20] (ARC'11)	MNFAU	Spartan 3	1.6	19,552	1,585	75,633	0.25	21.4
Prefix shared MNFAU (Proposed)	MNFAU	Virtex 5	1.6	132,136	9,930	665,040	0.20	15.3

7. Experimental Results

7.1. Implementation of the MNFAU

We implemented all the regular expressions of SNORT on the S2C Inc., Virtex-5 TAI logic module board (FPGA: XC5VLX330: 207360 logic cells (LCs), total 10,368 Kbits BRAM). The total number of regular expressions is 3,533 (665,040 characters) [7]. The prefix shared MNFAU has 64,387 states, and the AC-DFA for the transition string detection has 9,443 states. This implementation requires 132,136 LCs, and an off-chip memory of 16 Mbits. Note that, the 16 Mbits off-chip SRAM is used to store the transition function of the AC-DFA, while 9,930 Kbits on-chip BRAM is used to realize the decoder. The FPGA operates at 306.3 MHz. However due to the limitation on the clock frequency for the off-chip SRAM, the system clock was set to 200 MHz. Our regular expression matching circuit scans one character in every clock. Thus, the throughput is $0.2 \times 8 = 1.6$ Gbps.

7.2. Comparison of NFA Based Regular Expression Matching Circuits

Table 2 compares four types of NFA based regular expression matching circuits: NFA based (shown in Fig. 13); prefix shared NFA based (shown in Fig. 17); MNFAU based (shown in Fig. 23); and prefix shared MNFAU based. In all the circuits, 3,533 regular expressions (665,040 characters) are implemented. In Table 2, *#LC* denotes the number of logic cells; *On-chip MEM* denotes the amount of embedded memory for the FPGA (Kbits); and *#Char* denotes the number of characters for the regular expression.

Fig. 28 compares four different implementations of a part of ftp rule on the SNORT. Table 2 shows that the prefix sharing of the NFA reduced the number of states to $\frac{\#States_{PrefixSharedNFA}}{\#States_{NFA}} = \frac{610,972}{661,510} = 92.3\%$. Since, for the regular expression of SNORT, about 10% of the prefixes are specified by *protocol* (such as "SITE\s" [8] in Fig. 28), the prefix sharing reduces prefixes (Fig. 28 (b)). On the other hand, the conversion of the NFA into the MNFAU reduced the number of states to $\frac{\#States_{MNFAU}}{\#States_{NFA}} = \frac{83,703}{661,510} = 12.6\%$. Since, about 70% of the postfixes consist of *repetition of redundant characters* (such as "[^\n]{n}" [9]), the MNFAU that merges such repetition reduces the number of states for postfixes (Fig. 28 (c)). Since the number of states for prefixes are reduced by the prefix sharing and that for postfixes are reduced by the MNFAU, the prefix shared MNFAU reduced the number of states to $\frac{\#States_{PrefixSharedMNFAU}}{\#States_{NFA}} = \frac{64,387}{661,510} = 9.7\%$ (Fig. 28 (d)) of the NFA.

7.3. Comparison with Other Methods

Table 3 compares our method with other methods. Since these methods realize different numbers of regular expressions, direct comparison is difficult. So, we compare by the **embedded memory size per a character** and **the number of LCs per a character**. In Table 3, *Th* denotes the throughput (Gbps); *#LC* denotes the number of logic cells; *MEM* denotes the amount of embedded memory for the FPGA (Kbits); and *#Char* denotes the number of characters for the regular expression. Table 3 shows that, the embedded memory

[7] As for the same device (Spartan III: XC3S4000 consists of 62,208 LCs and 1,728 Kbit BRAMs) used in [20], we can store 1,325 rules (89,625 characters) by using a prefix sharing technique and a dedicated hardware shown in Section 3.7. However, since the restriction of amount of BRAM, this devices cannot realize all regular expressions of SNORT.

[8] For ftp, "SITE" executes the subsequent command. Thus, this command is used to send a malicious data by attackers.

[9] It causes a buffer overflow by attackers.

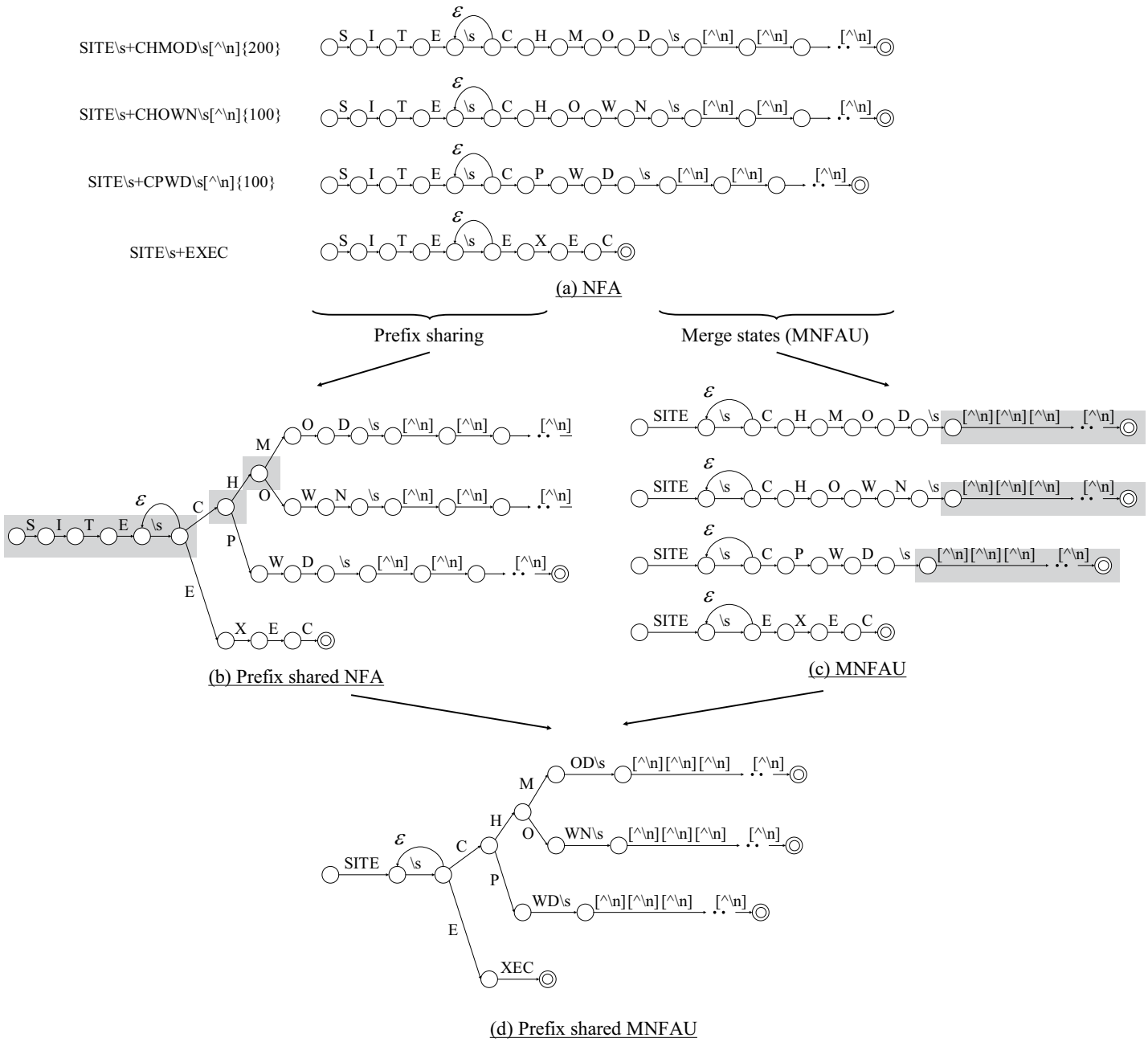


Figure 28: Four different NFAs for a set of regular expressions.

size per a character of the MNFAU is reduced to 0.2% (= $\frac{PrefixsharedMNFAU}{PipelinedDFA} = \frac{15.3}{3182.2}$) of the pipelined DFA; 4.2% (= $\frac{PrefixsharedMNFAU}{Bit-partitionedDFA} = \frac{15.3}{367.5}$) of the bit-partitioned DFA with an MPU; 41.0% (= $\frac{PrefixsharedMNFAU}{MNFA(3)} = \frac{15.3}{37.3}$) of the MNFA(3); and 71.4% (= $\frac{PrefixsharedMNFAU}{MNFAUwithoutprefixsharing} = \frac{15.3}{21.4}$) of the MNFAU without prefix sharing. Also, the number of LCs per a character of the MNFAU is reduced to 0.9% (= $\frac{PrefixsharedMNFAU}{PipelinedDFA} = \frac{0.20}{22.22}$) of the pipelined DFA; 15.6% (= $\frac{PrefixsharedMNFAU}{NFA} = \frac{0.20}{1.28}$) of the NFA; 51.2% (= $\frac{PrefixsharedMNFAU}{MNFA(3)} = \frac{0.20}{0.39}$) of the MNFAU(3); and 80.0% (= $\frac{PrefixsharedMNFAU}{MNFAUwithoutprefixsharing} = \frac{0.20}{0.25}$) of the MNFAU without prefix sharing.

8. Conclusion

This paper showed a regular expression matching circuit based on a decomposed MNFAU. First, the given regular expression is converted into the NFA by using the McNaughton-Yamada method. Second, to reduce the number of states of the NFA, the prefix sharing is applied. Third, to further reduce the number of states, the prefix shared NFA is converted into the MNFAU. Fourth, the MNFAU is decomposed into the transition string part and the state transition part. The transition string part is realized by the AC-DFA machine, while the state transition part is realized by the cascade of LCs. These circuits are connected by the decoder using BRAMs. We implemented all the regular expressions used in the SNORT on a Xilinx FPGA. Comparison with conventional methods showed that the embedded memory size per a character of the MNFAU is reduced to 0.2% of the pipelined DFA; 4.2% of the bit-partitioned DFA; 41.0% of the MNFAU(3); and 71.4% of the MNFAU without prefix sharing. Also, the number of LCs per a character of the MNFAU is reduced to 0.9% of the pipelined DFA; 15.6% of the NFA; and 80.0% of MNFAU without prefix sharing.

9. Acknowledgement

This research is supported in part by the grant of Regional Innovation Cluster Program (Global Type, 2nd Stage). Discussion with Prof. J. T. Butler was quite useful. Reviewer's comments were useful to improve the paper.

References

[1] A. V. Aho, and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Comm. of the ACM*, Vol. 18, No. 6, pp. 333-340, 1975.

[2] R. Baeza-Yates, and G. H. Gonnet, "A new approach to text searching," *Comm. of the ACM*, Vol. 35, No. 10, pp. 74-82, Oct., 1992.

[3] "The Bro Network Security Monitor," http://bro-ids.org/

[4] Z. K. Baker, H. Jung, and V. K. Prasanna, "Regular expression software deceleration for intrusion detection systems," *16th Int'l Conf. on Field Programmable Logic and Applications (FPL 2006)*, pp. 28-30, 2006.

[5] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," *Proc. IEEE Int'l Conf. on Field Programmable Technology (FPT 2006)*, pp. 119-126, 2006.

[6] B.C.Brodie, D.E.Taylor, and R.K.Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *Proc. 33rd Int'l Symp. on Computer Architecture (ISCA 2006)*, pp. 191-202, 2006.

[7] "Clam Anti Virus: open source anti-virus toolkit," http://www.clamav.net/lang/en/

[8] R. Dixon, O. Egecioglu, and T. Sherwood, "Automata-theoretic analysis of bit-split languages for packet scanning," *Proc. 13th Int'l Conf. on Implementation and Application of Automata (CIAA 2008)*, pp. 141-150, 2008.

[9] "Firekeeper: Detect and block malicious sites," http://firekeeper.mozdev.org/

[10] R. W. Floyd and J. D. Ullman, "The compilation of regular expressions into integrated circuits," it Journal of the Association for Computing Machinery, Vol. 29, Issue. 3, pp. 603-622, July, 1982.

[11] T. Ganegedara, Y. E. Yang, V. K. Prasanna, "Automation framework for large-scale regular expression matching on FPGA," *20th Int. Conf. on Field Programmable Logic and Applications (FPL 2010)*, pp. 50-55, 2010.

[12] A. K. Gupta and D. Suciu, "Stream processing of XPath queries with predicates," *Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of data (SIGMOD 2003)*, pp. 419-430, 2003.

[13] Z. Kohavi, *Switching and Finite Automata Theory, McGraw-Hill Inc.*, 1979.

[14] "Application Layer Packet Classifier for Linux," http://l7-filter.sourceforge.net/

[15] C. Lin, C. Huang, C. Jiang, and S. Chang, "Optimization of regular expression pattern matching circuits on FPGA," *Proc. of the Int'l Conf. on Design, automation and test in Europe (DATE 2006)*, pp. 12-17, 2006.

[16] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IEEE Trans. on Electronic Comput.*, vol. 9, pp. 39-47, 1960.

[17] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," *IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM2003)*, April, 2003.

[18] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching using non-deterministic finite automaton," *Proc. of Eighth ACM/IEEE Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE 2010)*, Grenoble, France, July 26-28, pp. 73-76, 2010.

[19] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching circuit based on a modular non-deterministic finite automaton with multi-character transition," *The 16th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI2010)*, Taipei, Oct. 18-19, pp. 359-364, 2010.

[20] H. Nakahara, T. Sasao and M. Matsuura, "A regular expression matching circuit based on a decomposed automaton," *7th Int'l Symp. on Applied Reconfigurable Computing (ARC 2011)*, March 23-25, 2011. Lecture Notes in Computer Science , No. 6578, pp. 16-28.

[21] H. Nakahara, T. Sasao and M. Matsuura, "A design method of a regular expression matching circuit based on decomposed automaton," *IEICE Trans. on Information and Systems*, Vol. E95-D, No. 2, pp. 364-373, 2012.

[22] M. Pellauer, A. Agarwal, A. Khan, M. C. Ng, M. Vijayaraghavan, F. Brewer, and J. Emer, "Design contest overview: Combined architecture for network stream categorization and intrusion detection (CANSCID)," *Proc. of Eighth ACM/IEEE Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE 2010)*, Grenoble, France, July 26-28, pp. 69-72, 2010.

[23] "PCRE: Perl Compatible Regular Expression," http://www.pcre.org/.

[24] H. C. Roan, W. J. Hawang, and C. T. Dan Lo., "Shift-or circuit for efficient network intrusion detection pattern matching," *Proc. Int'l Conf. on Field Programmable Logic and Applications (FPL 2006)*, pp. 785-790, 2006.

[25] R. Sidhu, and V. K. Prasanna, "Fast regular expression matching using FPGA," *Proc. of the 9th Annual IEEE Symp. on Field-programmable Custom Computing Machines (FCCM 2001)*, pp. 227-238, 2001.

[26] "SNORT official web site," http://www.snort.org.

[27] I. Sourdis, and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," *Proc. of the 12th Annual IEEE Symp. on Field-programmable Custom Computing Machines (FCCM 2004)*, pp. 258-267, 2004.

[28] "SPAMASSASSIN: Open-Source Spam Filter," http://spamassassin.apache.org/

[29] "Spartan III data sheet," http://www.xilinx.com/
[30] L. Tan, and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," *Proc. 32nd Int'l Symp. on Computer Architecture (ISCA 2005)*, pp. 112-122, 2005.
[31] W3C Recommendation, "XML path language (XPath) version 1.0. (1999)"
http://www.w3.org/TR/xpath/
[32] J. Weirong, V. K. Prasanna, "Parallel IP lookup using multiple SRAM-based pipelines," *IEEE Int. Symp. on Parallel and Distributed Processing (IPDPS 2008)*, pp. 1-14, 2008.
[33] "Using Look-up tables as shift registers (SRL16)," http://www.xilinx.com/support/documentation/application_notes/xapp465.pdf
[34] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," *Proc. of the 2006 ACM/IEEE Symp. on Architecture for Networking and Communications Systems (ANCS 2006)*, pp. 93-102, 2006.