NFA BASED REGULAR EXPRESSION MATCHING ON FPGA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

KAMİL SERT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2018

Approval of the thesis:

NFA BASED REGULAR EXPRESSION MATCHING ON FPGA

submitted by **KAMİL SERT** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by, Name, Last Name: Kamil SERT

Signature :

ABSTRACT

NFA BASED REGULAR EXPRESSION MATCHING ON FPGA

Sert, Kamil
M.S., Department of Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı

September 2018, 82 pages

String matching is about finding all occurrences of a string within a given text, which is a classical but still a popular problem. String matching algorithms have important roles in various real world areas, such as web and security applications. In this work, we are interested in solving regular expression and hence string matching problem targeting especially the network intrusion detection systems (NIDS) field.

An NIDS engine inspects both the header and the content of the packet and hence performs a type of deep packet inspection also. This inspection requires effective string matching techniques because each network packet should be checked and compared against maybe hundreds of possible malicious attacks at line speed. In this thesis, a detailed literature analysis is presented first that explains and classifies regular expression matching studies. Among these, studies exist that presents nondeterministic finite automata (NFA) based architectures and their novel mappings onto FPGA. In our study we select one such study [1] and further modify and enhance the NFA architecture already proposed. The reference study uses the modified-McNaughton-Yamada-algorithm and maps the resulting NFA into structural HDL for FPGA implementation. Our modification proposes to use a 2-character based matching structure that yields better memory utilization. With our approach, the circuit for the NFA representation needs less number of states (hence flip-flops) and LUTs to perform the 2-character regular expression matching process. Within the scope of this thesis, an extensive evaluation study is performed using the well-known Snort IDS ruleset and the worst case evaluation is done using some intuitively and synthetically created regular expressions targeting Xilinx 7-series FPGAs. Evaluation results are obtained using various performance metrics.

Keywords: regular expression matching, string matching, NFA, network intrusion detection, network security

ÖZ

FPGA ÜZERİNDE BELİRSİZ SONLU DURUM MAKİNASI TEMELLİ DÜZENLİ İFADE EŞLEŞTİRME

Sert, Kamil
Yüksek Lisans. Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Cüneyt Fehmi Bazlamaçcı

Eylül 2018, 82 sayfa

Dizi eşleştirme, bir dizinin belirli bir metin içerisinde ki tüm mevcudiyetlerini bulmakla ilgili olan klasik ancak yine de popüler bir sorundur. Dizi eşleştirme algoritmaları, web ve güvenlik uygulamaları gibi çeşitli gerçek dünya alanlarında önemli rollere sahiptir. Bu çalışmada, özellikle Ağa Sızmayı Algılama Sistemleri (ASAS) alanını hedefleyen düzenli ifade eşleştirme ve dolayısıyla dizi eşleştirme problemini çözmekle ilgileniyoruz.

Ağa sızmayı algılama sistemi ağ paketlerinin hem başlığını hem de içeriğini inceler dolayısıyla bir tür derinlemesine paket denetimi de gerçekleştirir. Bu inceleme, etkili dizi eşleştirme tekniklerinin kullanımını gerektirir, çünkü her ağ paketi belki de yüzlerce olası kötü niyetli saldırıya karşı hat hızında kontrol edilmeli ve karşılaştırılmalıdır.

Bu tezde, ilk olarak düzenli ifade eşleştirme çalışmalarını açıklayan ve sınıflandıran ayrıntılı bir literatür analizi sunulmaktadır. Bu çalışmaların arasında, belirsiz sonlu durum makinesi (NFA) temelli mimarileri ve onların FPGA üzerine eşlemlenmesini sunan çalışmalar bulunmaktadır. Kendi çalışmamızda böylesi bir çalışmayı [1] seçtik ve daha önce önerilen NFA mimarisini daha da değiştirip geliştirdik. Referans çalışma değiştirilmiş McNaughton-Yamada algoritmasını kullanır ve elde edilen sonlu durum makinesini FPGA uygulaması için yapısal donanım tanımlama diline (HDL) çevirir. Değişikliğimiz, daha iyi bellek kullanımı sağlayan 2-karakterli bir eşleştirme yapısı kullanmayı önermektir. Bizim yaklaşımımızla sonlu durum makinesi gösterimi için olan devre, 2-karakterli düzenli ifade eşleştirme işlemini gerçekleştirmek için daha az sayıda duruma (dolayısıyla iki durumlu hafıza birimine) ve arabul çizelgesine ihtiyaç duyar. Bu tez kapsamında, Xilinx 7 serisi FPGA ları hedefleyen, tanınmış Snort ağa sızmayı algılama sistemi kural seti kullanılarak kapsamlı bir değerlendirme çalışması ve bazı sezgisel ve yapay olarak oluşturulmuş düzenli ifadeler kullanılarak en kötü durum değerlendirmesi çalışması yapılır. Değerlendirme sonuçları çeşitli performans ölçütleri kullanılarak elde edilir.

Anahtar kelimeler: düzenli ifade eşleştirme, dizi eşleştirme, NFA, belirsiz sonlu durum makinası, ağ sızma algılama, ağ güvenliği

To My Family

ACKNOWLEDGEMENTS

I want to express my sincere gratitude to my supervisor Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı for giving me the honor of working with him. Without his constant support and guidance, it would be impossible to perform this work.

Last but not the least, I want to thank my family for their spiritual supports.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

String matching algorithms are used to find all occurrences of a string in a given text. These algorithms have important roles in a variety of real world application areas, such as security applications (spam filters, network intrusion detection systems, etc.), web applications (search engines, social media networking, etc.), bioinformatics, word processors, plagiarism detection, databases, etc. Many recent applications conduct regular expression matching in order to solve the string matching problem because regular expressions are more flexible in representing complex string patterns.

In this thesis, we are interested in solving regular expression matching and thereby the general form string matching problem targeting network intrusion detection systems (NIDS) in particular using a non-deterministic finite automata (NFA) based approach. Yang et al. [1]'s solution to this problem required two state registers while performing two-character (2-stride) matching in one cycle. We modified the NFA representation of the proposed architecture in [1] so that this match requires only one state register. In this case, while the limiting factor in [1] is the number of state registers, the limiting factor in our approach becomes LUT usage. To substantiate our approach, we designed several hardware modules and evaluated the proposed approach.

1.1 Overview

In computer networks, a malicious user may want to intrude into a computer system to steal information, to make unwanted interference or to make a Denial of Service (DoS) attack. In order to achieve such things, messages carrying malicious content could be sent to a victim computer. The aim of a Network Intrusion Detection System is to catch messages carrying such malicious content and this process has to be conducted at line speed. Many hardware or software based approaches were proposed in order to reach line speeds. Software-based approaches generally run on network processors or general-purpose processors. They utilize CPUs or GPUs and their throughputs are dependent on the computing power of the processors used [2][3]. Hardware-based approaches are generally implemented using field programmable gate arrays (FPGAs) [4][5][6], application specific integrated circuits (ASICs) [7], and content addressable memories (CAMs) [8]. In this work, we use an approach that runs on FPGA. FPGAs can make high-speed computations but their internal memories currently are limited to about tens of megabytes. Hence, we aim to use this limited memory more effectively in order to reach higher throughputs on the same device.

1.2 Aim of the Thesis

Network Intrusion Detection Systems (NIDSs) attempt to analyze the network traffic and identify malicious strings of information by performing a form of string matching at the heart of the system. An NIDS inspects both the header and the content of the packet and hence performs a type of deep packet inspection also. This inspection requires fast and effective string matching techniques because each network packet should be checked and compared against possible malicious attacks at line speed. There are a lot of approaches performing high-speed string matching, but increasing internet traffic still requires faster and faster string matching.

In this thesis, our aim is to obtain a non-deterministic finite automata (NFA) based high-throughput memory-efficient regular expression matching engine targeting state-of-the-art FPGA devices.

1.3 Contributions of the Thesis

We propose a new NFA architecture and its associated circuits, which makes it possible to represent two characters as one state and describe the mapping of the proposed architecture onto FPGA circuits. Our architecture requires smaller number flip flops to store the regular expressions, and in the worst case LUT usage does not exceed the previous approaches.

We perform performance evaluations for the proposed approach using regular expressions extracted Snort IDS's ruleset and test results are compared against similar studies found in the literature. Our results suggest that the proposed system performs well.

The contributions of this thesis can be summarized as follows;

1. We modify the RE-NFA architecture used in [1] and create an NFA including transitions corresponding to 2-character inputs also.

2. We propose four different modules to translate the proposed NFA architecture into FPGA circuits easily.

3. We use the same approach as in [1] for utilizing block memory resources of the FPGA device in order to implement centralized character classification. This helps us improve the resource efficiency of our design.

4. We provide worst case memory requirements of our architecture.

1.4 Thesis Outline

The thesis is organized as follows:

In Chapter 2, background information about NIDSs, existing NIDSs, regular expression matching problem and NFA based regular expression matching concepts are given. The relationship between NIDS and regular expression matching is discussed. Existing regular expression matching approaches are classified and explained.

In Chapter 3, the architecture proposed by Yang et al. [1] is summarized. The modular NFA architecture, BRAM character classification and other optimizations provided in

[1] are explained. The performance evaluation results of [1] are also discussed.

In Chapter 4, the proposed NFA based on 2-character transitions is presented and requirements of the suggested architecture that needs to be satisfied are explained. Corresponding circuit modules are defined and explained including the algorithms used to generate them.

In Chapter 5, performance evaluation of the proposed implementation is done, tools and platforms employed are given and the performance metrics used are explained. A detailed performance study including the scalability analysis is performed using various test scenarios.

In Chapter 6, conclusions and possible future work are discussed.

CHAPTER II

BACKGROUND AND LITERATURE OVERVIEW

2.1 BACKGROUND INFORMATION

2.1.1 Intrusion Detection Systems

Intrusion detection can be defined as identifying whether network packets are malicious or not by inspecting packet payloads for signatures in a given rule set (database). Defending a computer system or a computer network by identifying malicious attacks is the main purpose of a Network Intrusion Detection System (NIDS) [9]. A typical NIDS architecture is shown in ***Figure 2-1***.

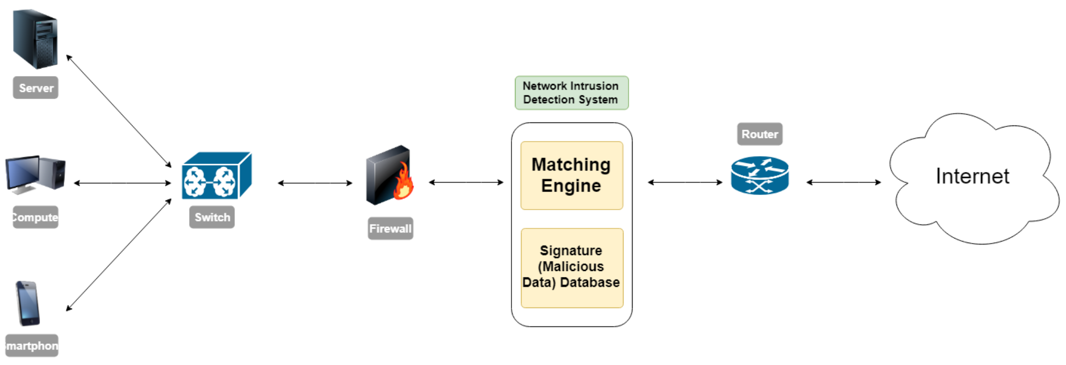

Server
Compute
martphon
Switch
Firewall
Network Intrusion Detection System
Matching Engine
Signature (Malicious Data) Database
Router
Internet

Figure 2-1: NIDS Architecture

NIDSs can be classified into two categories: anomaly-based and signature-based. Anomaly-based NIDS monitors, analyzes and searches network traffic for abnormal behaviors [10]. The most important feature of anomaly-based NIDS is the ability to detect zero-day attacks but their false positive rate is very high [11]. On the other hand, signature-based NIDS inspects packet payload to identify whether the packet contains malicious patterns (signatures) or not, hence performing deep packet inspection (DPI) also. Therefore, they generally employ string matching techniques.

Increase in the amount of network traffic and number of known-attacks results in increase in the number of signatures stored in rule sets/dictionaries such as Snort [12] and Bro [13], two well-known open source NIDSs. In that case, intrusion detection systems require more CPU time and memory. Therefore, implementations on FPGA like devices that have limited memory becomes a challenging issue. Current works aim to increase the search speed or to reduce memory to store signatures, by using for example compression techniques or new automata representations.

2.1.2 Existing IDSs

Two commonly used open source intrusion detection systems area Snort [12] and Bro [13]. Bro is an open-source anomaly-based intrusion detection system. Bro first extracts application level semantics, then analyzes and compares a variety of activities involving malicious patterns. Bro inspects both signature-based attacks and unusual activities [13]. Snort is the most widely used open-source intrusion detection system. It has the ability to perform packet logging, network traffic analysis and string matching. While doing performance analysis of our architecture we used regular expressions extracted from Snort's rule dictionary. ***Figure 2-2*** represents an example rule from Snort database. The first part of the rule (outside of the brackets) indicates that the signature corresponds to TCP packets that arrives from an external network from any port number, and that goes to HTTP ports of an HTTP server. Snort contains Perl compatible regular expressions (pcre). For the following example rule, the specified regular expression (in bold) will match those packets containing "username=" followed by 255 characters that are not '&', ';' or spaces.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-MISC Oracle iSQLPlus username overflow attempt";
flow:to_server, established; uricontent:"/isqlplus"; nocase;
pcre:"/username=[^&\x3b\r\n]{255}/si"; reference:bugtraq,10871;
reference:url,www.nextgenss.com/advisories/ora-isqlplus.txt;
classtype:web-application-attack; sid:2702; rev:1;)
```

Figure 2-2: Example SNORT signature

Snort database contains about 7.000 regular expressions in many different categories, such as blacklist, browser-firefox, file-office, malware-tools, os-linux, web-misc and protocol-icmp.

2.1.3 Pattern Matching, Regular Expressions and Finite Automata

Regular expressions are used to represent patterns for matching text. Each regular expression consists of regular characters or metacharacters, they have a textual and special meaning, respectively. For example, in the regex ab. , a and b are textual characters, that is they match only a and b characters, respectively, and '.' is a metacharacter matches every single character. Hence, this regex matches with abm, aba, abx, etc. Using metacharacters and textual characters any pattern can be identified.

Metacharacters provide flexibility to represent a specific patterns and help us to obtain easily readable regular expressions. For example, the regex initiali[sz]e matches both 'initialise' and 'initialize' words. In order to match against a search pattern, regular expression has to be translated into an internal representation. In that point one option is to utilize finite automata. We can construct a nondeterministic finite automata for the given regular expression using any NFA construction algorithm or construct DFA from the resulting NFA.

Finite Automata or Finite State Machine (FSM) is a computing model/device that recognizes/accepts a regular language, and can be used to solve string matching problem. Finite automata can be specified by a 5-tuple (Q, ∑, q, F, δ).

- Q: Finite set of states
- ∑: Set of input symbols
- q: Initial/Start state
- F: Final/accepting state
- δ: Transition function

Transition from one state to another state happens by transition function and an input symbol. There are two type of finite automata: Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA).

In a DFA, for a certain input symbol, automata goes to only one state. There is only one transition for each state and each input symbol. Hence, it is called deterministic. Also DFA cannot move to new state without any input character, that is null (ε, epsilon) transitions are not allowed for DFA. Input string is accepted if the accepting state is active. DFA has only one active state at any time.

In an NFA, for a certain input symbol, automata can goes to more than one states. That is, automata can move to different states by the same input symbol. Therefore it is called nondeterministic. NFA allows ε transitions, it can move to next state without processing an input symbol. Input string is accepted if one of the accepting state is active. In NFA, there might be more than one active states at any time.

Each NFA can be translated into a DFA. When NFA is translated to DFA, state explosion problem occurs. So, NFA requires less space than DFA.

Aim of the regular expression matching is to find all occurrences of a regex or string on input text. Normally, finite automata designed for implementing some regex only gives a match result if the input text matches completely. For example, automata for abcd regex gives a match result if and only if the input text is abcd. If input text is abcdefg or sabcdklabcd automata gives no-match output. However, if automata is used for regular expression matching problem we expect that the above automata gives a match for abcd, abcdefg and sabcdklabcd input text. That is a different search mechanism is employed in order to utilize automaton for regular expression matching.

2.2 LITERATURE REVIEW

2.2.1 Regular Expression Matching

Early deep packet inspection (DPI) techniques rely on exact string matching for attack detection [14] [15], however many recent techniques use regular expression matching [16] [17] [18] because regular expressions are more flexible in representing complex string patterns. At this point, we can categorize the relevant works broadly as NFA-based, DFA-based and hybrid designs.

NFA based implementations use less memory but their matching speed is slow. Some research have focused on utilizing parallelism to increase the NFAs matching performance [19] [20]. In [19], using multi-character decoding, the NFA technique has led to high-performance circuits with a large variety of pattern set sizes. NFAs keep a frontier of multiple states at every step during its operation while all these states are handled for every input symbol. In [20], NFA-OBDDs approach is introduced, which use ordered binary decision diagrams (OBDDs) in order to process the NFA frontier states in an efficient manner. NFAs' matching performance are unpredictable (no worst case guarantee). An efficient NFA based string matching model in BCAM (Binary Content Addressable Memory) is proposed in [21]. An efficient NFA generator using BCAMs, having fewer transistors and low latency, is built. The difference between other TCAM-based approaches and BCAM-NFA is that the latter doesn't access memory after each TCAM matching process, thus matching speed can be increased.

DFA based implementations suffer from state explosion issue. Exponential number of states may be generated when an NFA is converted to a DFA. Some methods were proposed to solve this issue to a certain extent. The main idea of D2FA [18] for example is to compress the state transition table of the DFA. The authors of [22] provide a compact representation of regular expressions that has the same throughput as of uncompressed DFAs. In [23], DFA states and transitions are grouped into three variable sized blocks, so that each block can employ different methods to optimize the performance and storage requirement. A memory-based parallel matching engine which uses the compressed state transitions is proposed in [24]. In this approach, the pointers represent the existence of transitions and they are compressed while the bit fields for storing transitions are also shared. Therefore, the memory cost can be minimized for storing these transitions. When multiple characters are processed at a time the throughput can be increased but this leads the transition table size to grow exponentially and hence increase total memory consumption. The authors in [25] applies two methods to overcome the memory explosion problem, namely top-k state extraction and transition merging.

Hybrid approaches combine the benefits of deterministic and non-deterministic finite automata. The aim of these approaches is to solve DFAs state explosion problem and NFAs performance problem simultaneously. By adjusting the degree of conversion of NFA/DFA, number of redundant states is reduced [26] [27] and in this way signatures, which may cause state explosion can be determined and avoided. Tunable-FA (TFA) [28], not a DFA, allows multiple simultaneous active states. Thereby, the total number of TFA states to watch the matching status is smaller than the number of DFA states. TFA guarantees that the number of active states is bounded by a bound factor, so unpredictable performance problem of NFA is also solved.

2.2.2 NFA-based Regular Expression Matching on GPU

In this part, we present existing NFA-based regular expression matching approaches implemented using Graphic Processor Units (GPUs).

DFA-based regular expression matching solutions suffer from state-explosion problem. Because of this problem, popularity of NFA-based architectures has increased. Using Graphic Processor Units (GPU) high clock frequency and high-throughput can be achieved. In [34], several methods are proposed in order to implement NFA on a GPU. Performance of the architecture is evaluated using regular expression sets extracted from Snort. The paper reports 29~46 times speedup and over 10 Gbps throughput.

In [32], SR-NFA (segmented regex NFA) architecture is proposed for regular expression matching. This architecture targets multi-core processors and is implemented in 4 steps. At the first step, regular expressions are compiled into nondeterministic finite automata (NFA) in a modular form. At the second step resulting NFA is partitioned, and at the third one some optimizations are done. The last step is for regular expression is matching on multi-core processors. The proposed architecture has the following advantages;

- SR-NFA is attack-resistant and has high throughput.
- It is not affected from the common issues of DFAs such as state explosion or backtracking.
- It can be constructed easily.

When compared with DFA implementations with moderate state explosion, the proposed architecture consumes 23k times less memory resources, and can be constructed and implemented 367k times faster. Performance evaluation is done on Opteron Platform, which has 8-cores and 2,6 GHz clock speed. Throughput of this architecture is about 2,2 Gbps.

2.2.3 NFA-based Regular Expression Matching on FPGA

In this part we present existing NFA-based regular expression matching approaches implemented using FPGAs.

The study in [5] is the first practical nondeterministic finite automata implementation on reconfigurable hardware. In order to implement a single character NFA module, we need one character-comparator circuit, one flip-flop and one AND gate. Using this single character module, union, concatenation and Kleene star operators (modules) can be easily constructed. NFA construction algorithm proposed in the paper, takes a regular expression as an input, then creates a syntax tree for it, and finally combines these modules in order to implement final circuit that represents the related NFA. The paper explains NFA construction both for Field-Programmable Gate Array (FPGA) and Self-Reconfigurable Gate Array (SRGA).

Another FPGA-based NFA architecture is presented in [39]. The authors describe a method for directly compiling Perl Compatible Regular Expression (PCRE) opcodes generated via Snort signatures to HDL and finally implement them on FPGA. Performance evaluation is done using Xilinx Virtex-4 FPGA device. They obtain 12,9 Gbps interface throughput. And also they achieve speedup of 353x over SW-based perl compatible regular expression execution.

In [38], a similar NFA-based technique on FPGA is presented. Multiple characters are matched per clock cycle which is, according to authors, is the key novel property of their proposal. Another novelty is an efficient range match operation implementation. In range match, [a-z] matches any character between a to z. Performance of the proposed architecture is evaluated using Snort regular expression database. With 1-character input NFA and 1,25 Gbps throughput is achieved. With 4-character input NFA 3,63 Gbps throughput is achieved. 4-character NFA is shown to consume 6% less registers and 20% more LUTs.

In paper [37] a bitmap-based architecture is presented for the Glushkov-NFA (G-NFA). This architecture can process multiple symbols per cycle in order to achieve higher throughput. Authors mentioned that their architecture is suitable for small and moderate length regular expressions. The algorithm has ability to detect the ending positions of the input strings' substring.

CAN-SCID (Combined Architecture for Stream Categorization and Intrusion Detection) is an NFA-based regular expression matching architecture [36]. This architecture utilizes a microprocessor for match results counting and for system control. They implement the architecture in 3 steps: firstly, regexes are translated into an NFA, secondly, resulting NFA is converted to a new modular NFA architecture, which has p-character transitions, and finally, in order to match with p-characters a finite-input memory machine is designed. Performance evaluation is done on Altera Stratix-3 FPGA device, and 798 Mbps throughput is achieved.

CES is an NFA-based approach implemented on FPGA [35]. CES means Character Class with Constraint Repetition (CCR) based regular expression scanner. MIN-MAX is a counting algorithm which helps to solve both ambiguity of character classes problem and overlapped match problem. CES is designed to create a novel MIN-MAX algorithm. This algorithm supports back-references also. Block RAMs are utilized to store character classes. Proposed architecture is implemented on a Xilinx Virtex-5 device and its estimated throughput is obtained to be approximately 2 Gbps.

ECD-NFA [28] is based on an Equivalence Classification, which is a type of input compression technique. Proposed approach constructs classes via compressed inputs. These inputs are represented by positive integers and they are referred as ECDs. The main idea is to classify inputs and to drive the non-deterministic finite automata via classified inputs. The architecture does not use unclassified input strings. The proposed technique has two challenges: First one is that decoding module uses too many BRAM units, shift registers and logic, and the other one is that it requires too long to synthesize and place-and-route the design. ECD-NFA technique can be parallelized on FPGA. It runs at about 460 MHz and achieves 3,68 Gbps throughput.

In [29], a regular expression matching technique targeting text analytics systems is proposed. These systems have many functions such as, start-offset reporting, group capturing. It suggests some extensions such as configuration register implementation to NFA architectures [5], [1] in order to implement the above functions. The proposed architecture eliminates state replication, avoids offset comparisons and reduces the number of offset registers in order to achieve resource-efficiency and higher clock frequency. The architecture evaluated using regular expressions extracted from text analytics and IDS areas and it is shown to achieve higher clock frequency and threefold logic resource reduction on Altera Stratix FPGA.

Memory-based non-deterministic finite automata (MX-NFA) approach [30] proposes a proper technique in order to handle character class repetitions which occurs in Snort signatures. These repetitions can cause expensive hardware cost when repetitions are handled by unrolling. In this technique, transition rules and embedded control signals are stored in a table which has a different organization in comparison to the conventional approach. Each table entry contains a transition symbol and a control signal, but does not store the state-IDs. Outgoing transitions from an active state are activated and from an inactive state are deactivated, instead of holding active states. In order to handle these repetitions MX-NFA module also implemented a count module. Performance evaluation is performed using ClamAV virus database.

Non-deterministic finite automata based regular expression matching technique on reconfigurable hardware called as ENREM is proposed in [31]. This paper designs a novel infix and suffix sharing mechanism. In order to optimize memory-cost for pattern matching circuits this mechanism and several other techniques are utilized. The proposed architecture is evaluated using Snort rules and is implemented on Xilinx Virtex-2 FPGA device. It was possible to reduce LUT usage by 42% and Flip-Flop usage by 32% when compared with the previous architectures. The implementation had throughput between 1,45 to 2,35 Gbps.

Yang and Prasanna [33] proposed a design, implementation and evaluation of a modular NFA-based high-performance REM architecture. This architecture firstly parses regular expressions to represent them in a concise token list form and then generates RE-NFA using McNaughton-Yamada NFA construction algorithm. In order to increase the performance of the proposed architecture several optimizations are done. To process multi-character inputs per clock cycle spatial stacking is utilized while constructing the engine. In order to match complex character classes, a mechanism that enables sharing of a BRAM-based character classifiers between regular expressions is proposed. Using shift-register LUTs in parallel, matching of a single-character constrained repetitions are handled efficiently. Test results show that 11 Gbps throughput can be achieved on Xilinx Virtex 5-series with this architecture. They also claim that they managed to obtain more *throughput efficiency* than the other architectures.

A Binary Content Addressable Memory (BCAM) based efficient nondeterministic finite automata (NFA) architecture is suggested in [20]. Implementation of this architecture needs less transistors and has shorter latency. With multi-character processing, scalability is achieved. The architecture is evaluated using both Snort and ClamAV signature sets.

NFA-OBDDs architecture is proposed in [19]. This architecture utilizes ordered binary decision diagrams (OBDDs) in order to process NFA frontier state sets. Experiments conducted using with regular expression sets extracted from Snort has shown that NFA-OBDDs can outperform traditional NFAs by up to three orders of magnitude.

CHAPTER III

COMPACT ARCHITECTURE FOR HIGH-THROUGHPUT REGULAR EXPRESSION MATCHING ON FPGA

In this part of the thesis, we explain the work of Yang et al. [1], which has been chosen as our main reference paper. This paper has already modified the original RE-NFA conversion approach used in [5] to get a highly modular structure, which helped them to translate an NFA into FPGA circuits easily. They have also suggested a very simple way of using the block ram (BRAM) resource of the FPGA. Because of these advantages, we prefer to use some of the ideas they have already proposed. In the paper some additional optimizations are also employed in order to get a high-throughput search engine.

3.1 ARCHITECTURE OVERVIEW

Yang et al. [1] implemented their regular expression matching engine (REME) on FPGA in three steps. First, a regular expression is parsed into a tree structure in order to be able to perform a post-order traversal of the regular expression. In the second step, a modular NFA architecture is constructed using the modified McNaughton-Yamada construction algorithm. Finally, the resulting NFA architecture is mapped into HDL for FPGA implementation. These steps are explained in following subsections.

3.1.1 Regular Expression to NFA Conversion

Given a regular expression, original McNaughton-Yamada NFA construction algorithm generates an NFA, which has many intermediate nodes (white circles) and unnecessary (ϵ)-transitions (dashed lines). An example NFA constructed by original McNaughton Yamada algorithm is shown in ***Figure 3-1***.

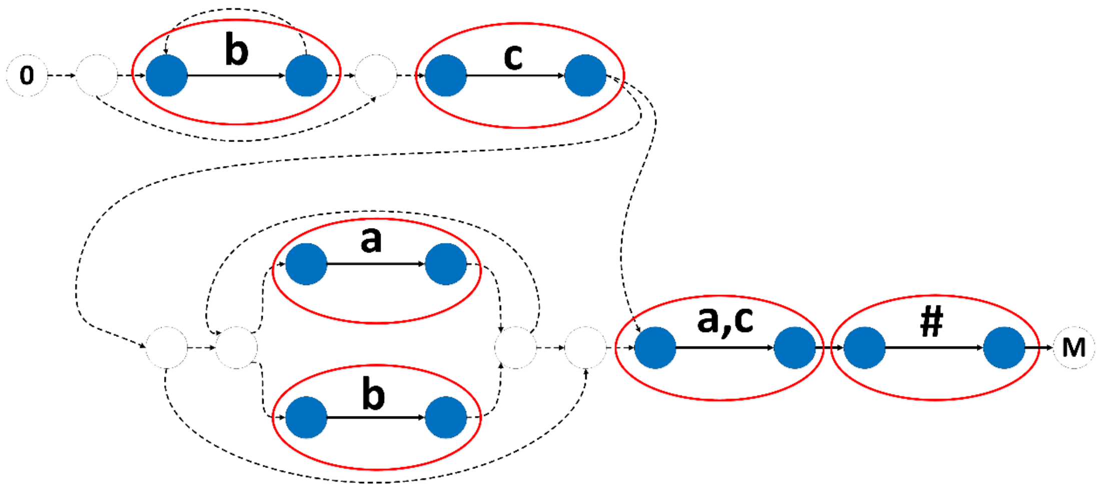


Figure 3-1: NFA representation of 'b*c(a|b)*[ac]#' for original McNaughton-Yamada construction

Yang et al. [1] modified the McNaughton-Yamada algorithm to eliminate unnecessary nodes and epsilon (ϵ)-transitions. Therefore, they reduced the memory cost and get a highly modular architecture that is easy to map onto an FPGA. The NFA constructed by the modified McNaughton-Yamada algorithm is shown in ***Figure 3-2***.

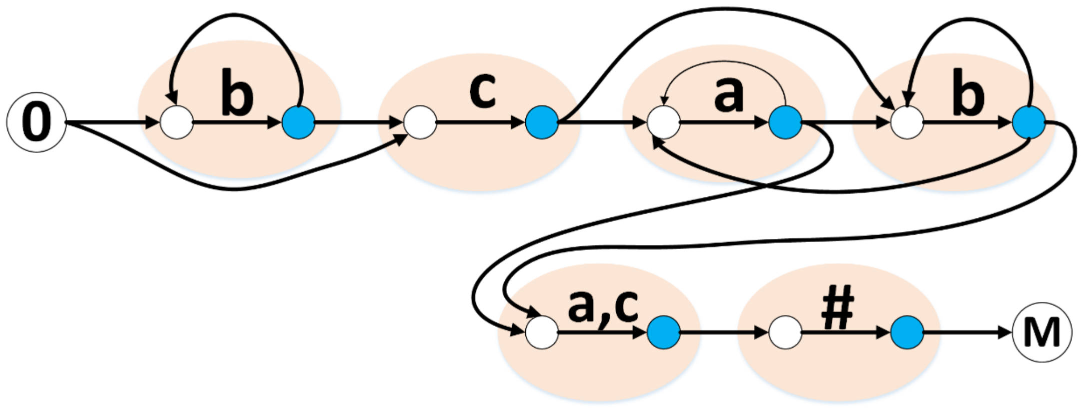


Figure 3-2: NFA representation of 'b*c(a|b)*[ac]#' for modified McNaughton Yamada algorithm

3.1.2 NFA to HDL mapping

In the resulting NFA, shaded elliptic areas are identical and they are named as *basic state block*. These basic state blocks can be defined as a single module type, e.g. an entity in VHDL. In order to translate the NFA into a circuitry, all we need to do is to connect the basic state blocks in accordance with state transitions. The circuit that corresponds to the regular expression **`b*c(a|b)*[ac]#`** is shown in ***Figure 3-3***. All of the basic blocks contain one OR and one AND gate. Rectangles show character comparison results which will be obtained from BRAM.

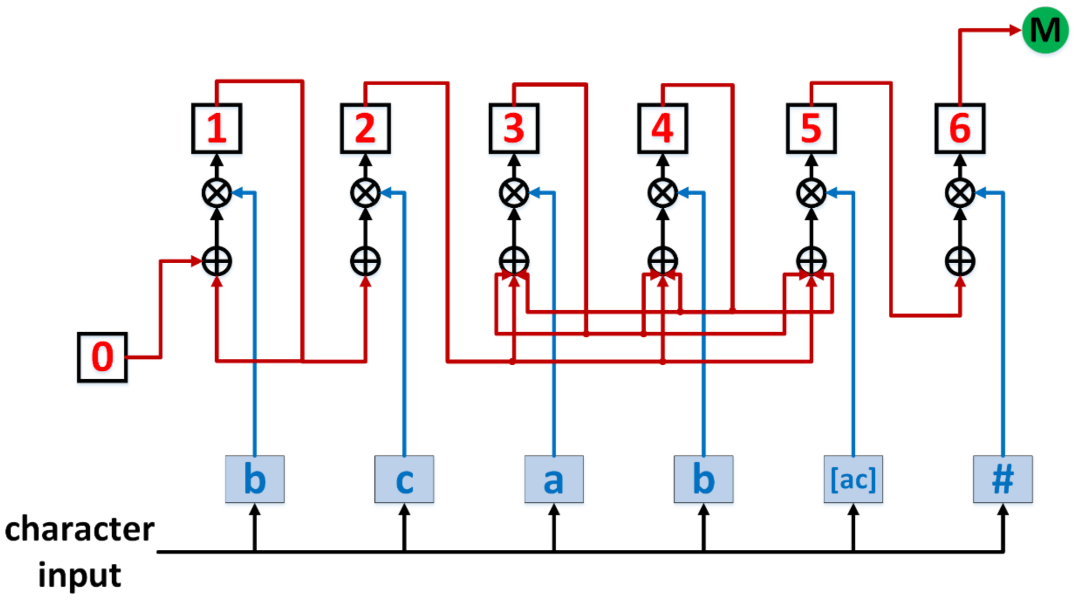


Figure 3-3: Circuit corresponding to `b*c(a|b)*[ac]#` ***regular expression***

3.1.3 BRAM-based Character Classification

In a previous work [5], character match signals were obtained via comparators such as the one in ***Figure 3-4,*** which is designed to match with character ‘a’ whose binary ascii representation is ‘01100001’. This 8-bit comparator requires two LUTs for its implementation. 4 MSBs are input to the first LUT. The LUT contains only one 1 entry corresponding to the number to be compared. 4 LSBs are processed similarly in the second LUT and the AND gate generates the final match when both parts match.

If such comparators are used for all states, FPGA LUT usage increases. Therefore, storing character match signals on BRAMs instead of comparators (hence LUTs) might be preferable, which also does not affect clock frequency badly.

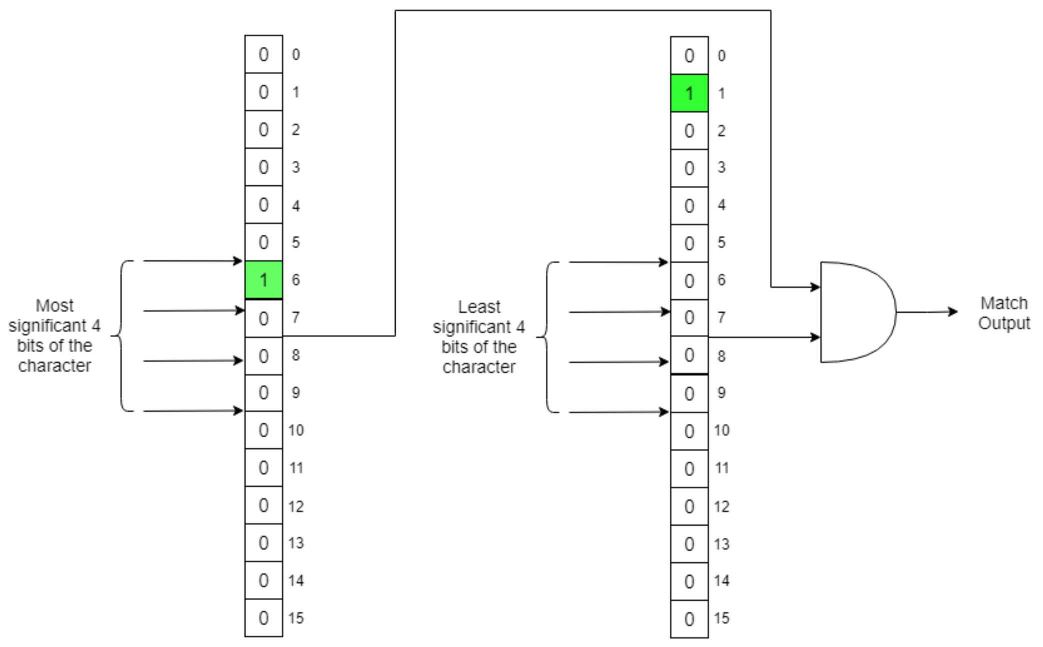

Most significant 4 bits of the character
Least significant 4 bits of the character
Match Output
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Figure 3-4: 8-bit comparator for character ‘a’ (ascii 0110 0001)

Snort signature database contains Perl Compatible Regular Expressions (PCREs). PCRE represents a group of characters as a character class and doing so any regular expression can be written in a shorter and more easily readable form. The table below shows some example PCRE character classes and corresponding standard regular expression representations.

Table 3-1: Example Character Class Representations

PCRE Character Class	Corresponding standard regular expression representation	Comment
a	a	single character 'a'
[ac]	(a | c)	a + c
[ab - e]	(a | b | c | d | e)	a + b + c + d + e
\d	(0 | 1 | 2 | 3 | … | 9)	one digit
\w	(a | b | c | … | y | z)	one word character
[0-9 a-f]	(0 | 1 | … | 9 | a | b | … | f)	one digit or any character 'a' to 'f'

For example, to implement '\d' character class we need a 10 8-bit comparators (as in ***Figure 3-4***). Such an implementation requires more LUT resources and can reduce clock frequency of the architecture. Yang et al. implemented these character classes on block RAM by storing character match signals directly in BRAM. BRAM contents corresponding to the circuit in ***Figure 3-3*** is illustrated in ***Figure 3-5***.

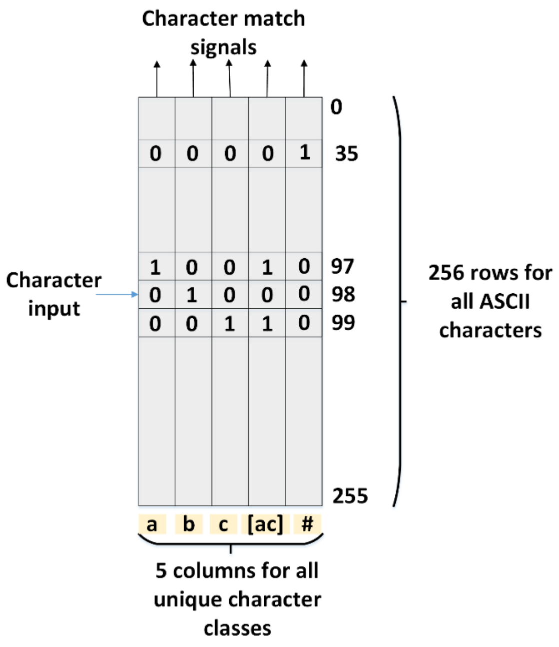


Figure 3-5: Character class representation in BRAM

In order to implement the circuit in ***Figure 3-3*** we need 5 different character match signals, i.e. there are 5 different character classes; a, b, c, # and [ac]. In the associated BRAM, there are 5 columns, each corresponding to a unique regular expression and 256 rows for each ASCII characters, and outputs a 5-bit number to be used in the circuit. For example, ASCII code of ‘b’ is 98, hence we store ‘1’ at the 98th row of the column corresponding to character class ‘b’. In order to implement character class ‘[ac]’ we store ‘1’ at the 97th and 99th row of the column corresponding to character class ‘[ac]’. If input of the BRAM is ‘a’ or ‘c’ then BRAM outputs ‘1’ from [ac] column.

Encoding simple character classes may result in some redundancy in BRAM. In order to minimize this, centralized character classification is proposed by Yang et. al., as illustrated in ***Figure 3-6***. With this architecture any block RAM output can be used by different REMEs.

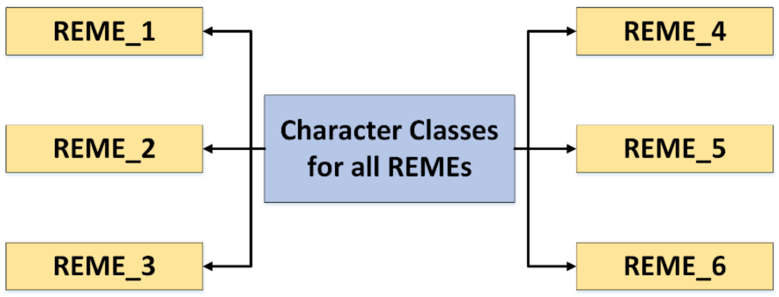


Figure 3-6: Centralized character classification for 6 different REMEs

Assume that REMEs have the following character classes;

- REME_1 → a, b, c, \w
- REME_2 → c, d, \w, [a-z]
- REME_3 → a, e, [acx]
- REME_4 → \w, [acx], [0-9]
- REME_5 → a, b, c
- REME_6 → [a-f 0-9]

We have 10 distinct classes, when these 6 REMEs are implemented. Virtex-7 FPGAs have BRAM units having 64-bits data output, therefore we can implement a maximum of 64 distinct character classes on each block RAM unit. Characteristics of the regular expressions in the set to be implemented determines the maximum number of regular expressions that can be implemented in a single BRAM block. If REME_1 and REME_5 are implemented on the same BRAM, we need a total of 4 distinct character classes. But if REME_1 and REME_4 are implemented together, in that case we need a total of 6 distinct character classes. Consequently, we can claim that grouping of regular expressions may reduce the number of BRAM units required while implementing large sets.

3.2 MULTI-CHARACTER MATCHING OPTIMIZATION

In order to get higher throughput, Yang et al. [1] proposed also a multi-character input matching architecture. Multi-character inputs are also known as *strides* in the literature. In ***Figure 3-7***, a 2-input (2-stride) character matching circuit is shown. In comparison to single character matching, this approach requires nearly the same amount of LUTs but half the amount of state registers and one extra BRAM in order to obtain the character match signal for the second character. To create this 2-input matching circuit, Yang et al. [1] used two separate 1-input matching circuits. For this, the state registers of the lower circuit are removed first and the outputs of the lower AND gates are forwarded to corresponding state registers. Then, state outputs are connected to the inputs of the OR gates in the lower circuit. In the combined circuit, the first (blue lines in **Figure 3-7**) and second (black lines in ***Figure 3-7***) characters are processed by the lower and upper of the 2-input matching circuit, respectively.

Our idea is simply to represent two consecutive characters in only one state so that register usage will be reduced by 50% for a 2-input (2-stride) character matching circuit.

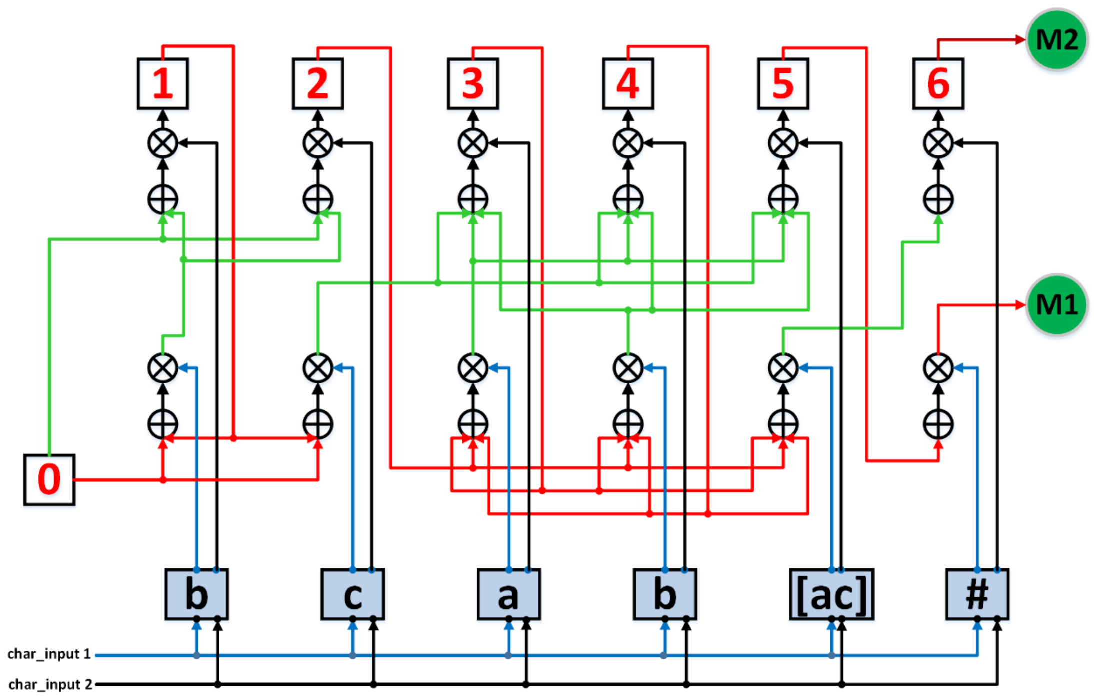


Figure 3-7: Yang's 2-stride character matching circuit

CHAPTER 4

2C-NFA ARCHITECTURE FOR FAST REGULAR EXPRESSION MATCHING

In this chapter, a detailed explanation of our new non-deterministic finite automata based architecture, where each transition represents 2-characters is given and its advantages, disadvantages and performance analysis are provided. First, we discuss the main idea and then present the design of the corresponding search mechanism required to find all occurrences of strings matching the specified RE and four circuit modules that are used to implement the NFA based circuit in question. We also present algorithms that may be used to generate and implement these circuit modules on modern day FPGAs without any structural fault.

4.1 Main Idea

State transitions on finite automata corresponds in general to actions triggered by a single character. For example; let a regular expression be **`kl(mn|op)qr`**. This expression can be matched by using the NFA in ***Figure 4-1*** appropriately. The automata goes from initial state to state-1 via single character **`k`**, from state-1 to state-2 via **`l`**, and so on. In order to store this regular expression on FPGA, we need 8 state registers except the initial one.

Our idea is to represent transitions as functions of two concatenated characters, that is transition from state-*i* to state-*j* can be triggered, for example by **`ab`** or **`xy`**. In this way, number of state registers that will be required to store a regular expression on FPGA can be reduced by half. This approach will be referred as 2C-NFA in the rest of the thesis. 2C-NFA representation for the example regular expression given above is shown in ***Figure 4-2***. In order to it, we now need only 4 state registers except the initial one.

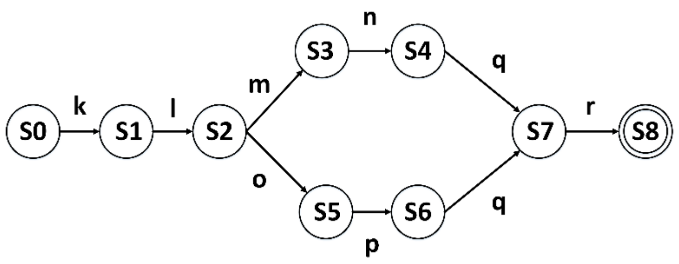


Figure 4-1: NFA used to match regexp `kl(mn|op)qr`

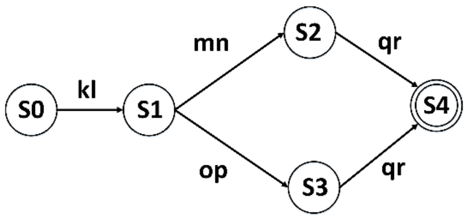


Figure 4-2: 2C-NFA used to match `kl(mn|op)qr`

4.2 Structural Construction of 2C-NFA

In this part of the thesis, we will develop the circuits and the search mechanism that are necessary to construct our 2C-NFA approach on FPGA. 2C-NFA architecture is implemented in four steps.

1. Given a regular expression, split it into sub-expressions using '(', ')' and '|' as delimiters.
2. Split each sub-expression into groups such that each group is composed of two non-star characters and including their following '*'s, if exists; where the final group can be composed of a single non-star character, if necessary.
3. Construct NFAs and their associated circuit corresponding to each group (hereafter modules) and combine them to form the final circuit.

The above steps are explained in detail below:

<u>Step 1:</u>

Any regular expression can be represented as composed of simpler sub-expressions. In this step, we identify '(', ')' and '|' as delimiters and label the remaining parts of the regular expression as sub-expressions.

Let RE be an example regular expression given as `cde*f*(g*hij|kl*m*)nop`.

RE can be re-written as the concatenation of the following sub-expressions having delimiters also in place.

$$RE = RE_1\ (\ RE_2 \mid RE_3\ )\ RE_4$$

where:

- RE_1 = `cde*f*`
- RE_2 = `g*hij`
- RE_3 = `kl*m*`
- RE_4 = `nop`

Figure 4-3 illustrates the NFA for matching RE as composed of smaller NFAs corresponding to the sub-expressions formed. This is a hierarchical form in which the unlabeled transitions correspond to epsilon transitions. Each connection between two NFAs in ***Figure 4-3*** represents a set of transitions from the final states of the previous machine to the initial state of the current machine.

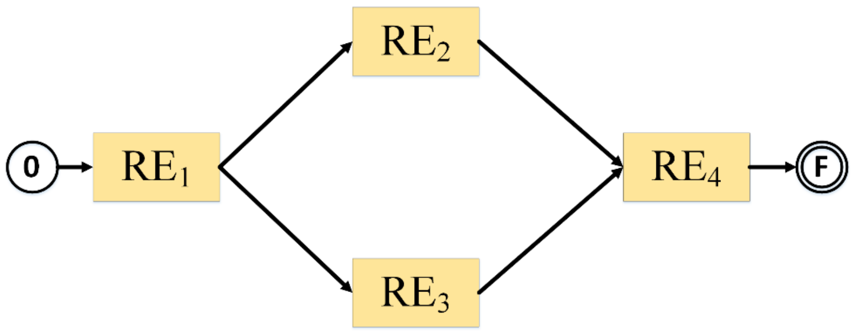


Figure 4-3: NFA for RE (hierarchical representation)

<u>Step 2:</u>

In this step, we find groups of character pairs in each sub-expression. For the given example, the groups are obtained as follows:

- RE_1 = `cde*f*` = G_1G_2 where G_1 = `cd` and G_2 = `e*f*`
- RE_2 = `g*hij` = G_3G_4 where G_3 = `g*h` and G_4 = `ij`
- RE_3 = `kl*m*` = G_5G_6 where G_5 = `kl*` and G_6 = `m*`
- RE_4 = `nop` = G_7G_8 where G_7 = `no` and G_8 = `p`

Hence RE = G_1G_2 (G_3G_4 | G_5G_6) G_7G_8. Final group may be composed of a single character.

<u>Step 3:</u>

We observe that there can be at most 6 different types of groups for any regular expression if the above splitting mechanism (steps 1 and 2) is employed. General form of expressions corresponding to these possible groups are shown below:

- **ab** → Type 1 module (T1) (example: `cd, ij, no`)

- **a** → Type 2 module (T2) (example: `p`)
- **a*b** → Type 3 module (T3) (example: `g*h`)
- **a*** → Type 4 module (T4) (example: `m*`)
- **ab*** → Type 5 module (T5) (example: `kl*`)
- **a*b*** → Type 6 module (T6) (example: `e*f*`)

The implementation details of the above module library (T1…T6) is presented in section 4.2.2.

Assuming we have the necessary modules, we choose the associated Ti for each group and combine them as illustrated in ***Figure 4-4***.

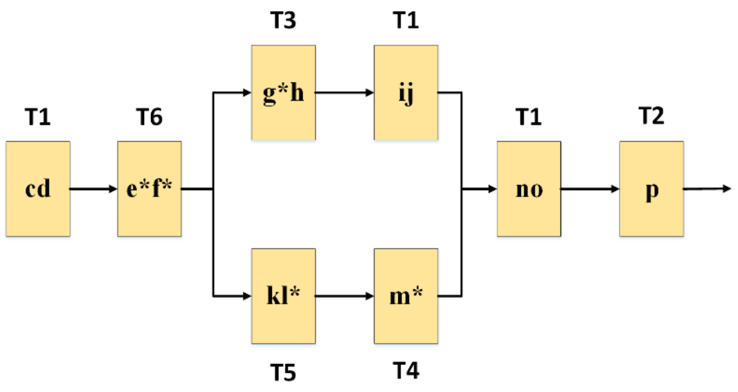


Figure 4-4: Combination of modules to implement a given regexp

4.2.1 2-Character Shifted 3-Character Window Search

Using the NFA approach, any pattern can be matched anywhere in the input stream provided that the input is presented to the engine one character at a time. However, when matching with 2C-NFA architecture, a pattern can only be matched if the first character of the string is in an odd-numbered position. Otherwise, 2C-NFA cannot match, hence providing a false negative output.

Here are some sample input streams where **`klmnop`** can be found.

- **klmnop**gdkewf
- sm**klmnop**gdkewf
- fngsythb**klmnop**gdkewf

It will be helpful to make the following definitions:

Unshifted case: The case where the first character of the pattern to match occurs in an odd-numbered index in the input stream.

Shifted case: The case where the first character of the pattern to match occurs in an even-numbered index in the input stream.

We would like to note that example input streams given above represent *unshifted* cases. In order to match a regular expression anywhere in the input stream with our 2C-NFA approach, i.e. to eliminate false negative matches, it is necessary to present a new search mechanism suitable to the new architecture.

In order to eliminate false negatives, we need three characters to check in parallel at every clock cycle while searching the input stream. Simply stated, we need to use 3 characters as input and then shift these 3 characters window by 2 characters at every iteration since two input characters are to be consumed by the engine at each clock. The idea is illustrated below.

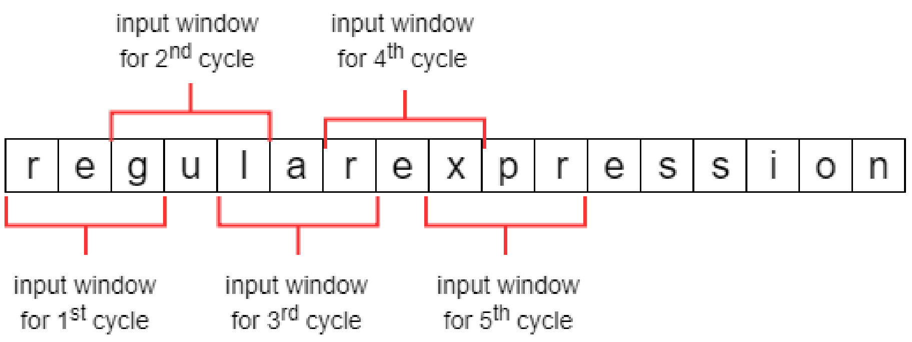

input window for 2nd cycle
input window for 4th cycle
r e g u l a r e x p r e s s i o n
input window for 1st cycle
input window for 3rd cycle
input window for 5th cycle

Figure 4-5: 2-character shifted 3-character windows

We are searching character pairs both on the left side of the window for an unshifted case and on the right side of the window for a shifted case as illustrated in ***Figure 4-6***.

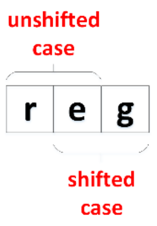


Figure 4-6: Illustration of search window

In ***Figure 4-5***, the input provided to the automata are '*reg*', 'gul', 'lar', etc. in subsequent cycles. Match results (whether there is a match or not) will be obtained from BRAMs in our architecture, the same way as in Yang [1] but we need three replicated BRAMs to obtain the required character match signals. For every character match result, we need a corresponding 1-bit signal. The circuit for obtaining character match signals is shown in ***Figure 4-7***. BRAMs take ch1, ch2 and ch3 as an address input, respectively and outputs 3 n-bit numbers, where each bit in these numbers correspond to a different character class. BRAM outputs will be connected to the corresponding character match inputs of the state logic.

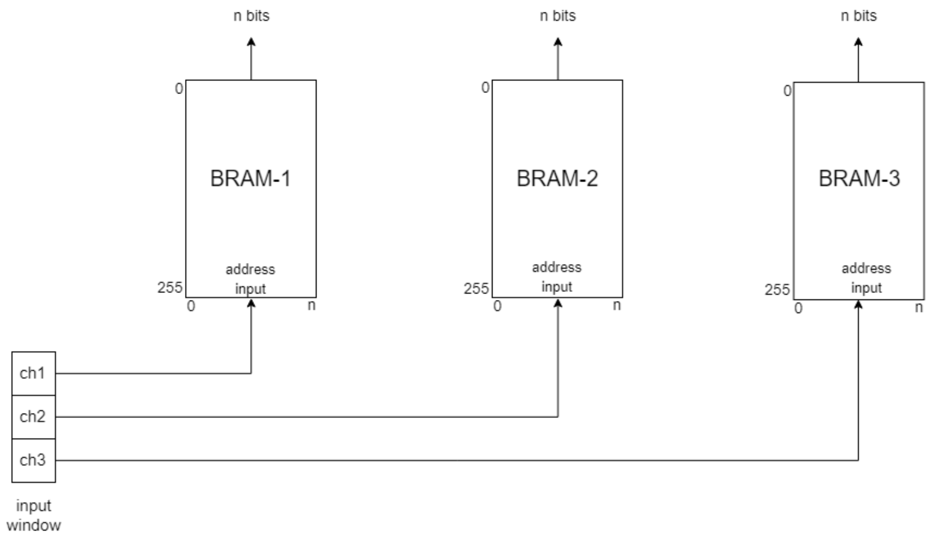


Figure 4-7: Circuit for obtaining character match signals

4.2.2 Module Design

4.2.2.1. T1 module

T1 module implements **ab** type patterns. General NFA representation that can be used to match it is shown in ***Figure 4-8***.

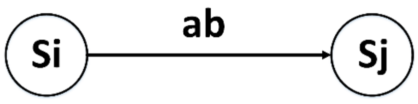


Figure 4-8: General NFA representation of the T1 module

With our search mechanism, we search for string **ab** in a 3-character window. **ab** can be found either at the right or left of the window. Hence, the NFA representation of T1 module that can be used to match ab using 2-Character Shifted 3-Character Window Search is given in ***Table 4-1*** and in ***Figure 4-9.***

Table 4-1: State transition table of the T1 module

	NS	
PS	ab-	-ab
Si	Sj	Sj
Sj	-	-

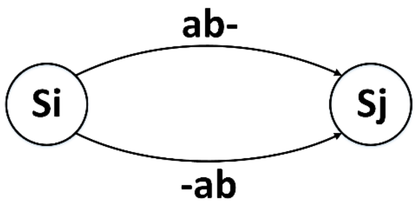


Figure 4-9: Final NFA representation of the T1 module

We need two AND gates and one OR gate to implement this module on FPGA. The resulting circuitry is given in ***Figure 4-10***.

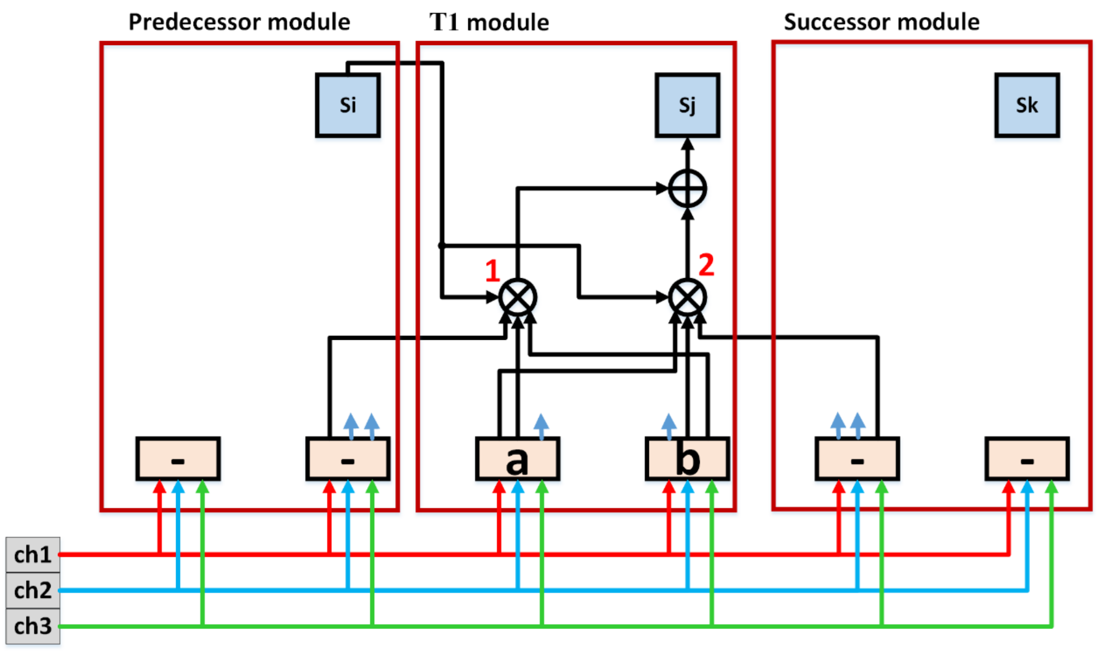


Figure 4-10: T1 module

When implementing **`ab-`**, if successor module is available, we need a connection from the successor module's left character class. In that case '`-`' stands for the character stated above, otherwise it stands for don't care (no connection from successor module can be considered as a don't care).

When implementing **`-ab`**, if predecessor module is available, we need a connection from the predecessor module's right character class. Similarly in this case also, '`-`' stands for the character stated above, otherwise it stands for don't care (no connection from predecessor module can be considered as a don't care).

Remark: The above explanations for '`-`' is valid for all modules.

Validation: We can implement the circuit to match **`cdefgh`** as an example to show that 2C-NFA architecture does not cause any false positives or false negatives. State transition table for **`cdefgh`** and the corresponding NFA are shown in ***Table 4-2*** and ***Figure 4-11***, respectively.

Table 4-2P: State transition table of* `cdefgh` *regexp

	NS					
PS	-cd	cde	def	efg	fgh	gh-
S0	S1	S1	-	-	-	-
S1	-	-	S2	S2	-	-
S2	-	-	-	-	S3	S3
S3	-	-	-	-	-	-

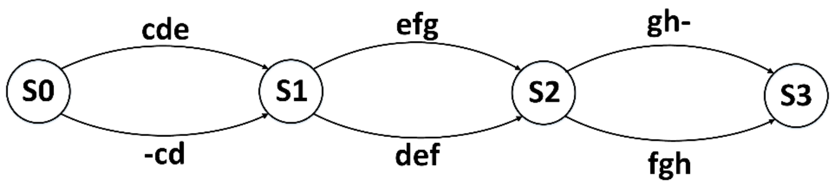


Figure 4-11: NFA representation of* `cdefgh` *regexp

We can implement this regular expression using 3 T1 modules on FPGA where the resulting circuit is given in ***Figure 4-12***.

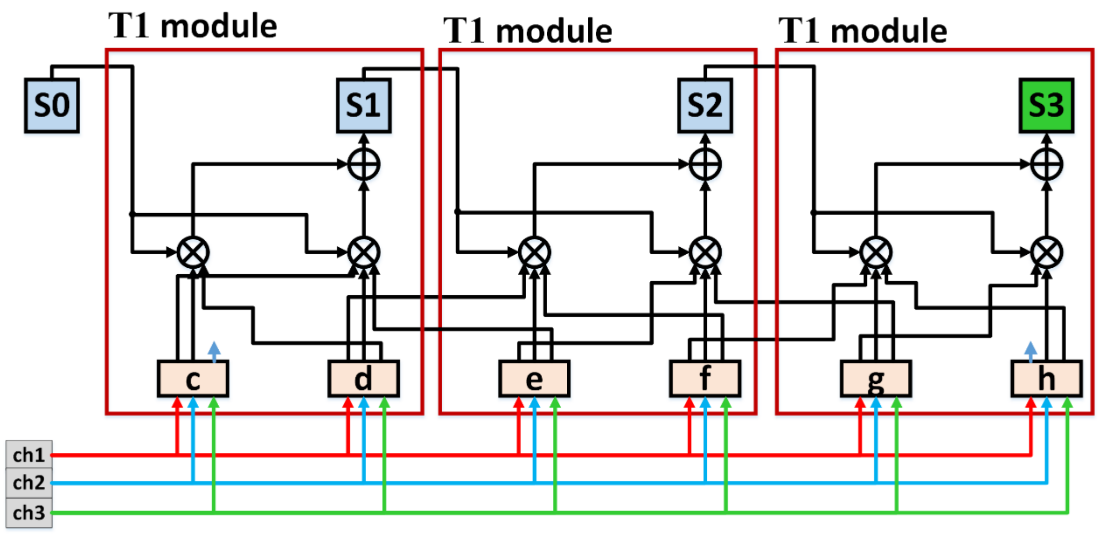


Figure 4-12: Circuit for* `cdefgh` *regular expression

Transition from S0 to S1 can only occur if the input window contains `-cd` or `cde`. Assume that the input window contains `-cd` for the ith cycle. There is a match with `-cd` and S1 will be active. In (i+1)th cycle, input window is shifted by 2-characters, and it exactly starts with character ‘`d`’. Transition from S1 to S2 occurs if the input window contains `def` or `efg`. Since the first character of the window is `d`, then we are searching for `def` in order to move to S2 state. If this happens, S2 will be active. In the (i+2)th cycle, input window starts exactly with `f`. Transition from S2 to S3 occurs if the input window contains `fgh` or `gh-`. Since the first character of the input window is `f`, we are searching for `fgh` in order to move to S3 state. If this happens, S3 will be active, i.e., `cdefgh` has been found on the input stream. Similarly, if input windows contains `cde` , `efg` and `gh-` for successive cycles then the automata will match `cdefgh` string. NFA in ***Figure 4-11*** finds only `cdefgh`. This is illustrated in ***Figure 4-13***.

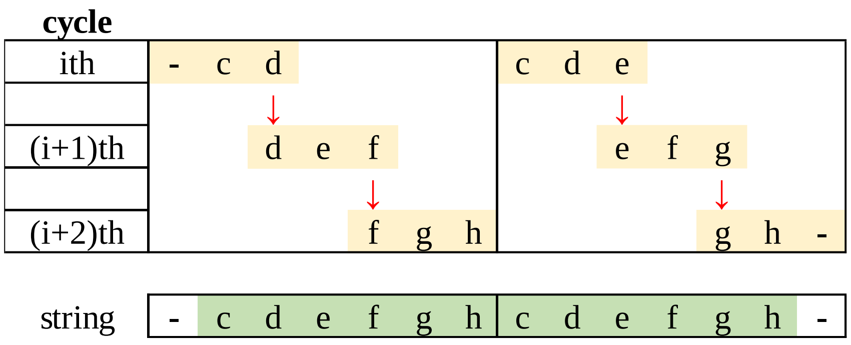


Figure 4-13: Illustration of the validation step for T1 module

4.2.2.2. T2 module

T2 module implements **`a`** type regular expression, i.e., single character. NFA representation of it is shown in ***Figure 4-14***.

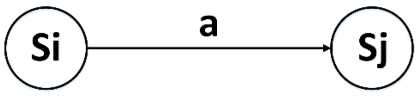


Figure 4-14: General NFA representation of the T2 module

T2 module is similar to T1 module. While T1 module is searching for **`ab`** on the left and the right of an input window, T2 module is searching for **`a`** on the left and the right of an input window. We obtained final NFA representation of T2 module as in ***Figure 4-15*** and state transition table of this module as in ***Table 4-3***.

Table 4-3: State transition table of the T2 module

	NS	
PS	a--	-a-
Si	Sj	Sj
Sj	-	-

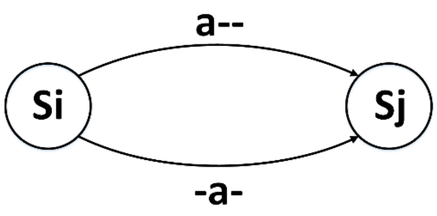


Figure 4-15: Final NFA representation of the T2 module

We need two AND gates and one OR gate to implement this module on FPGA. The resulting circuitry is given in ***Figure 4-16***.

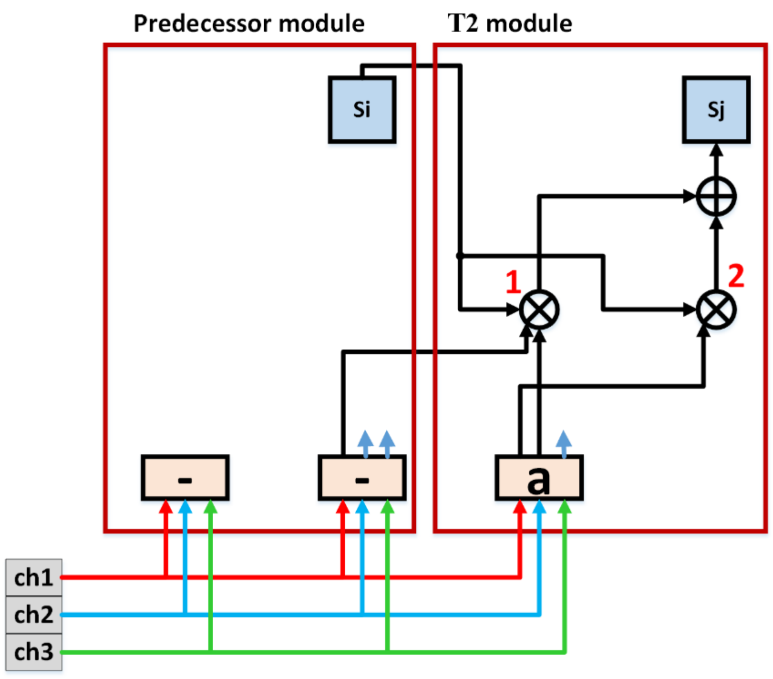


Figure 4-16: T2 module's circuitry

Validation: We will implement **`cdefg`** as an example, and using this example, we will validate that 2C-NFA architecture does not cause any false positives or false negatives. State transition table for **`cdefg`** is shown in ***Table 4-4*** and its corresponding NFA is shown in ***Figure 4-17***.

Table 4-4: State transition table of `cdefg` ***regexp***

	NS					
PS	-cd	cde	def	efg	fg-	g--
S0	S1	S1	-	-	-	-
S1	-	-	S2	S2	-	-
S2	-	-	-	-	S3	S3
S3	-	-	-	-	-	-

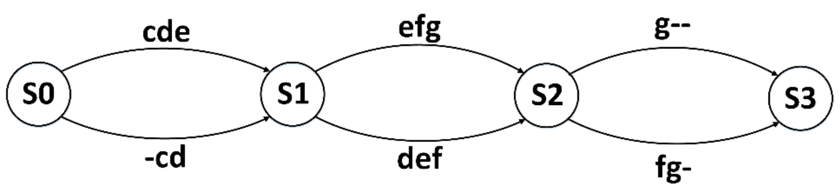


Figure 4-17: NFA representation of `cdefg` *regexp*

Above NFA finds **`cdefg`** string only, which is illustrated in ***Figure 4-18***.

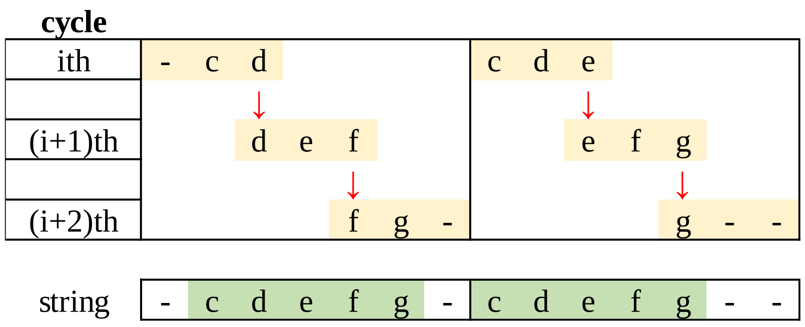


Figure 4-18: Illustration of the validation for T2 module

4.2.2.3. T3 module

T3 module implements **`a*b`** type regular expressions. General NFA representation of it is shown in ***Figure 4-19***.

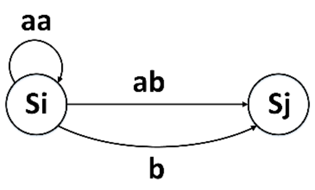


Figure 4-19: General NFA representation of the T3 module

Using **`a*b`** we can derive the following 2-character length strings; aa, ab and b. T3 module searches for these strings and they can be found at the right of a window, or at the left of a window. NFA representation of T3 module is shown in ***Figure 4-20***.

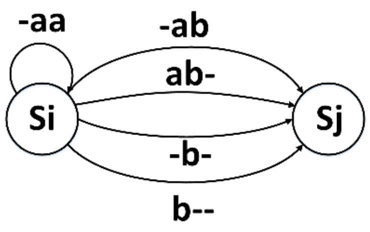


Figure 4-20: NFA representation of T3 module

`-b-` transition also contains **`ab-`** transition therefore we can only implement **`-b-`** transition string. To decrease number of inputs to LUTs, dashes on the right are not implemented (not connected to previous modules). Because of the nature of our 3-character window search mechanism transition from predecessor of the Si state (say Sh) to Sj is possible. Assuming that x and y are the character classes of the Sh state (x and y correspond to a and b, respectively), then there is a transition 'Sh to Sj via xyb'. Finally, we can form the following NFA representation for T3 module in ***Figure 4-21***. whose state transition table is given in ***Table 4-5*** while the resulting circuit is given in ***Figure 4-22.***

Table 4-5: State transition table of T3 module

	NS			
PS	xyb	-aa	-ab	-b-
Sh	Sj	-	-	-
Si	-	Si	Sj	Sj
Sj	-	-	-	-

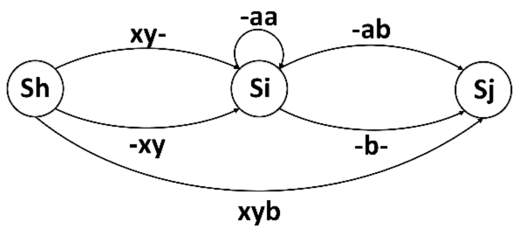


Figure 4-21: Final NFA representation of T3 module

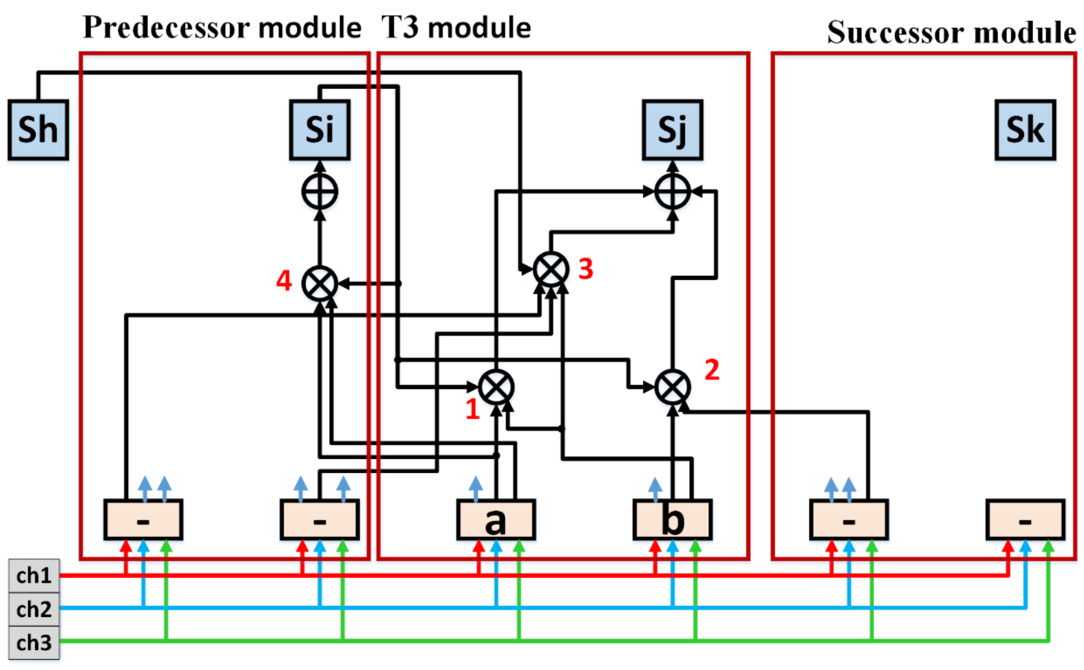


Figure 4-22: T3 module's circuitry

We need four AND gates and one OR gate to implement this module on FPGA.

Validatiom: We will implement **`cde*fgh`** regular expression as an example, and using this example, we will validate that 2C-NFA architecture does not cause any false positives or false negatives. NFA representation of **`cde*fgh`** is shown in ***Figure 4-23***. Notice that cdfgh (no e), cdefgh (single e), cdeefgh (double e), cdeeefgh (triple e), etc. matches with **`cde*fgh`**.

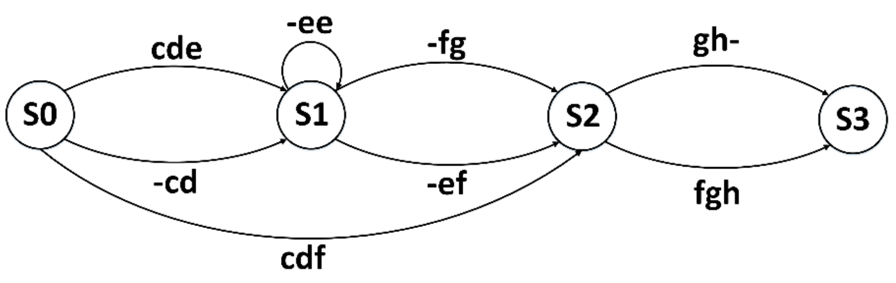


*Figure 4-23: NFA representation of 'cde*fgh' regexp*

Above NFA exactly finds those strings that only match with **`cde*fgh`** as illustrated in ***Figure 4-24.***

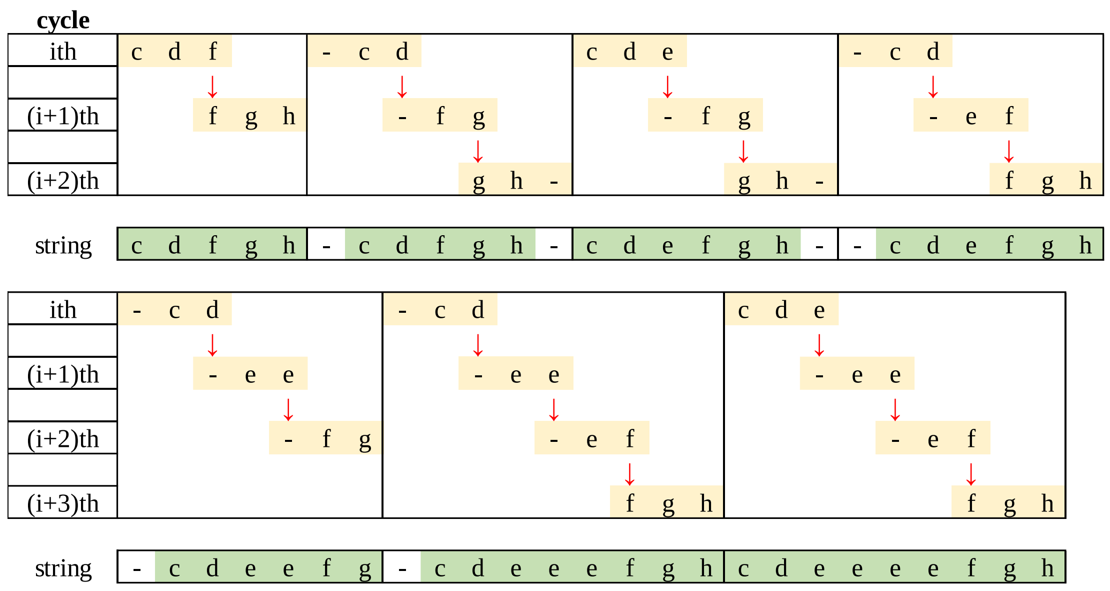


Figure 4-24: Illustration of the validation for T3 module

4.2.2.4. T4 module

T4 module implements **`a*`** type regular expression and is similar to T3 module. We need to convert some transitions of T3 module to the following transitions to implement T4 module.

- 'Sh to Sj via xyb' transition is converted into 'Sh to Sj via xy-'
- 'Si to Sj via -ab' transition is converted into 'Si to Sj via -a-'
- 'Si to Sj via –b-' transition is converted to 'Si to Sj via ---', this means that if Si is an active state then Sj becomes an active state without any character match.

NFA representation and state transition table of T4 module is given in ***Figure 4-25*** and ***Table 4-6***, respectively.

Table 4-6: State transition table of T4 module

	NS			
PS	xy-	-aa	-a-	---
Sh	Sj	-	-	-
Si	-	Si	Sj	Sj
Sj	-	-	-	-

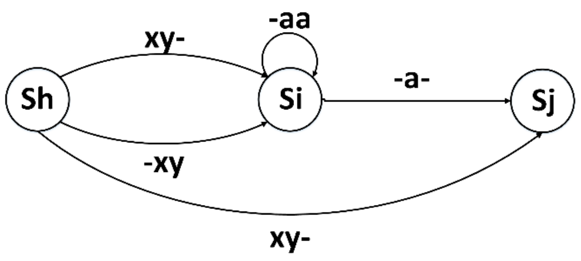


Figure 4-25: Final NFA representation of T4 module

T4 module's circuitry is shown in ***Figure 4-26***. We need four AND gates and one OR gate to implement this module on FPGA.

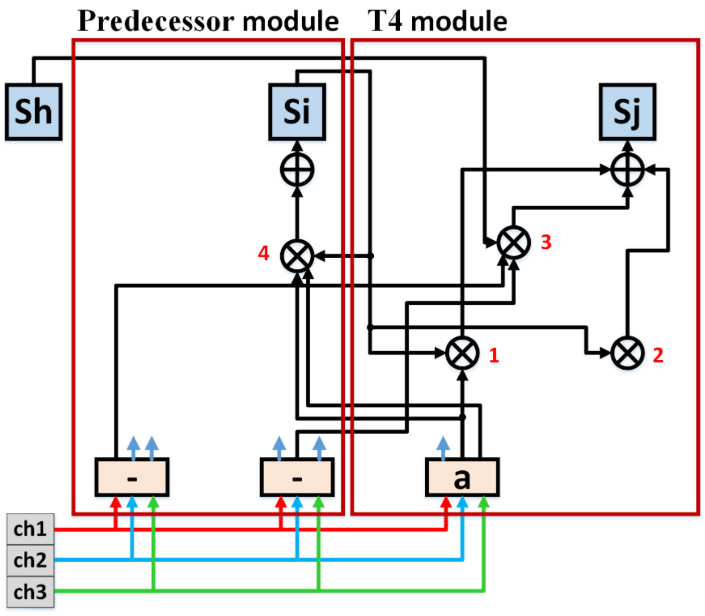


Figure 4-26: T4 module's circuitry

Validation: We will implement `cde*` as an example, and using this example, we will validate that 2C-NFA architecture does not cause any false positives or false negatives. NFA representation of `cde*` is shown in ***Figure 4-27***. Notice that cd (no e), cde (single e), cdee (double e), cdeee (triple e), etc. matches with `cde*` regular expression.

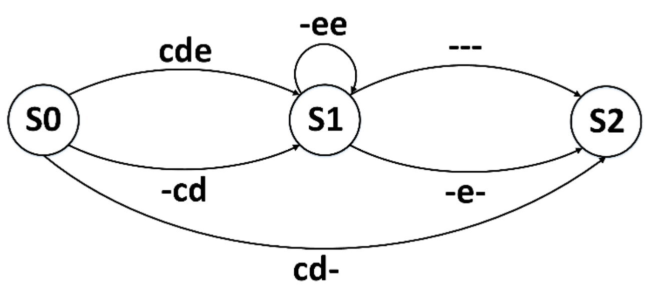
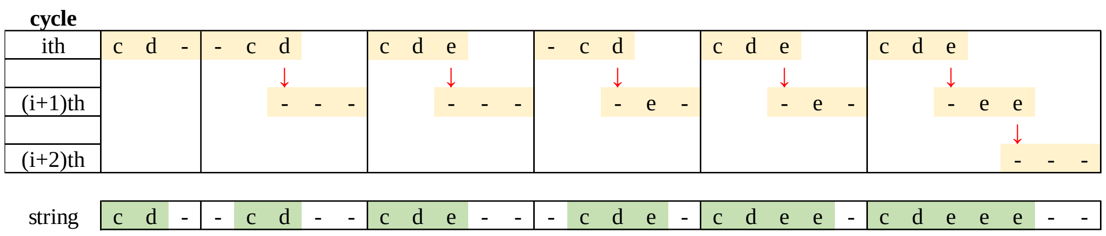


Figure 4-27: NFA representation of 'cde*' regexp

Above NFA exactly founds strings that only match with `cde*` which is illustrated in the ***Figure 4-28.***



cycle
ith
(i+1)th
(i+2)th
string
c d - | - c d | c d e | - c d | c d e | c d e
- - - | - - - | - e - | - e - | - e e
- - -
c d - | - c d - - | c d e - - | - c d e - | c d e e - | c d e e e - -

Figure 4-28: Illustration of the validation for T4 module

4.2.2.5. T5 module

T5 module implements `ab*` regular expression. General NFA representation of it is shown in ***Figure 4-29***.

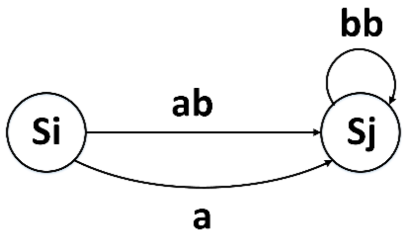


Figure 4-29: General NFA representation of the T5 module

T5 module searches for ab, a and bb strings and they can be found at the right of a windows, or at the left of a window. Therefore, we can represent this module on NFA as shown in ***Figure 4-30***.

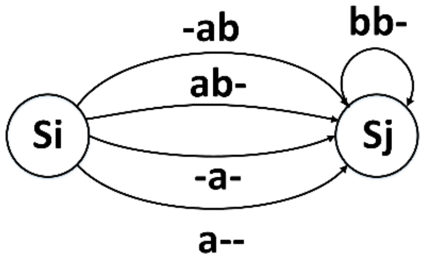


Figure 4-30: NFA representation for T5 module

`-a-` transition also contains `-ab` transition so we can only implement `-a-` transition string. To decrease number of inputs to LUTs, dashes on the left are not implemented (not connected to next modules). Due to our 3-character window search mechanism transition from predecessor of the Si state (say Sh) to Sj and transition from state Si to successor of the state Sj (say Sk) are possible. Assuming that x and y are the character classes of the Sh state and u and v are the character classes of the Sk state (x,u and y,v correspond to a and b respectively), then there are transitions 'Sh to Sj via xya' and 'Si to Sk via 'auv'. Finally, we obtain the NFA representation and state transition table of T5 module and they are given in ***Figure 4-31*** and ***Table 4-7***, respectively.

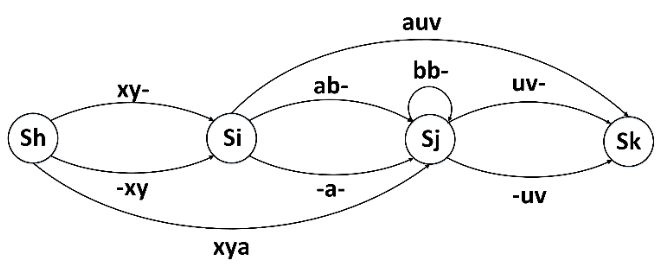


Figure 4-31: Final NFA representation for T5 module

Table 4-7: State transition table of T5 module

	NS				
PS	xya	ab-	-a-	bb-	auv
Sh	Sj	-	-	-	-
Si	-	Sj	Sj	-	Sk
Sj	-	-	-	Sj	-
Sk	-	-	-	-	-

We need five AND gates and one OR gate to implement this module on FPGA as in *Hata! Başvuru kaynağı bulunamadı.*.

Figure 4-32: T5 module's circuitry

Validation: We will implement **`cdef*gh`** as an example, and using this example, we will validate that 2C-NFA architecture does not cause any false positives or false negatives. NFA representation of **`cdef*gh`** is shown in ***Figure 4-33***. Notice that cdegh (no f), cdefgh (single f), cdeffgh (double f), cdefffgh (triple f), etc. matches with `cdef*gh` regular expression.

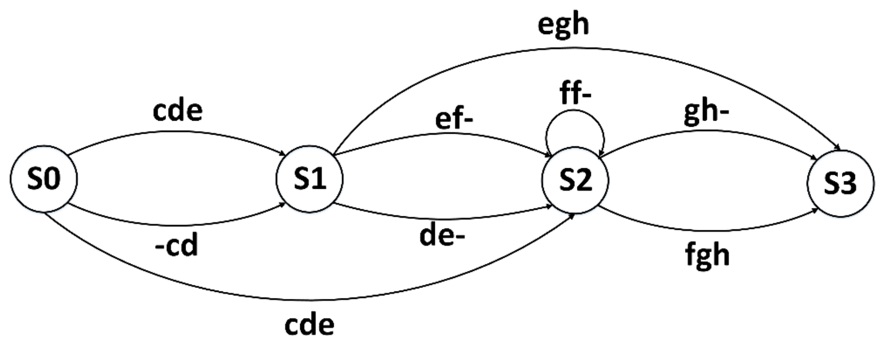
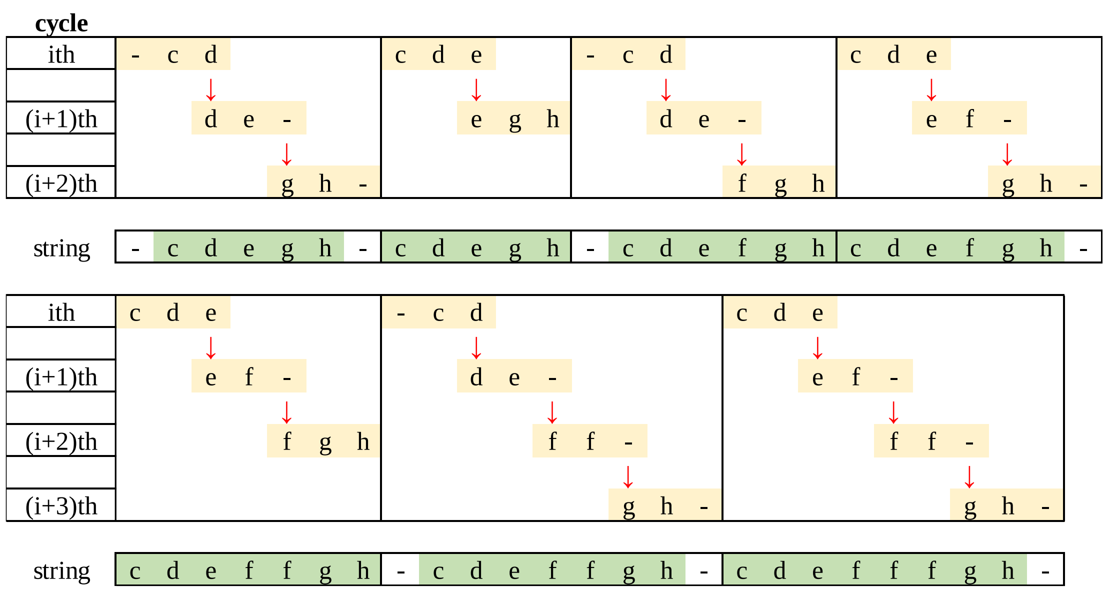


*Figure 4-33: NFA representation of 'cdef*gh' regexp*

Above NFA exactly finds strings that only match with **`cdef*gh`**. This is illustrated in ***Figure 4-34.***



cycle
ith
(i+1)th
(i+2)th
(i+3)th
string
- c d e g h - c d e g h - c d e f g h c d e f g h -
c d e f f g h - c d e f f g h - c d e f f f g h -

Figure 4-34: Illustration of the validation for T5 module

4.2.2.6. T6 module

T6 module implements `a*b*` type regular expression. General NFA representation of it is shown in ***Figure 4-35***. This module can be considered as a combination of T3 and T5 modules. So we can create the state transition table using state transition tables of them. State transition table of T6 module is given in ***Table 4-8***.

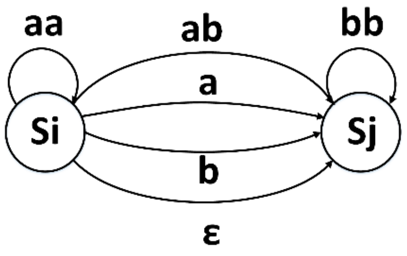


Figure 4-35: General NFA representation of the T4 module

Table 4-8: State transition table of T4 module

	NS									
PS	xyb	xya	-aa	-ab	-b-	ab-	-a-	bb-	yuv	auv
Sh	Sj	Sj	-	-	-	-	-	-	-	-
Si	-	-	Si	Sj	Sj	Sj	Sj	-	Sk	Sk
Sj	-	-	-	-	-	-	-	Sj	-	-
Sk	-	-	-	-	-	-	-	-	-	-

We can combine `xyb` and `xya` transition strings as an `xy-` string and `yuv` and `auv` transition strings as an `-uv` string. And also we can combine `'-b-'` and `-a-` as `'-(a/b)-'` in order to obtain more compact circuit. NFA representation and resulting circuit for the T4 module is given in ***Figure 4-36*** and ***Figure 4-37*** below.

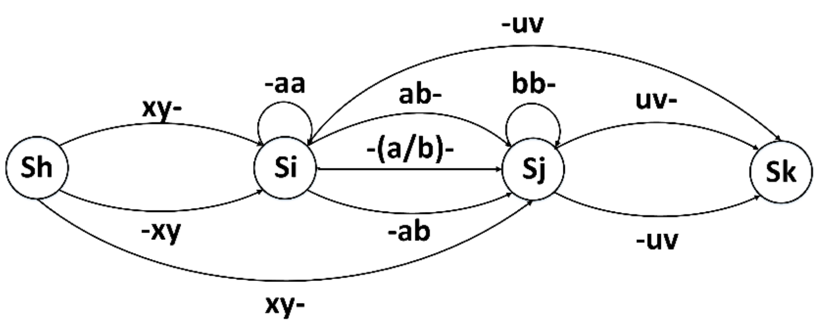


Figure 4-36: NFA representation for T4 module

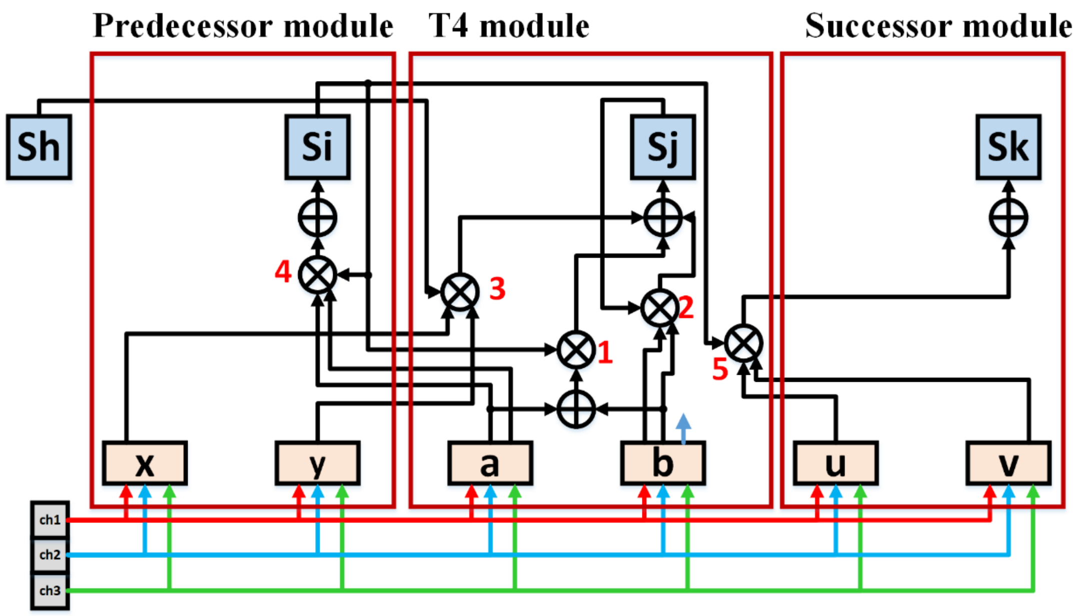


Figure 4-37: T4 module's circuitry

4.3 Module Implementations

We presented the design of the circuitry for all modules in the previous section. While combining these modules and getting the match result from the accepting state/states, we consider module positions. Consider "***S0 S1 (S2 S3 | S4 S5) S6 S7 S8***" as the regular expression under question.

We can list possible positions as follows:

• After initial state	(example: *S0*)
• In the middle	(example: *S1* to *S7*)
• In the last state	(example: *S8*)

Other possible cases with respect to OR grouping are given below:

• Case1: Before OR group, i.e. before "("	(example: *S1*)
• Case2: Inside OR group, i.e. after "(" or after "|"	(example: *S2* and *S4*)
• Case3: Inside OR group, i.e. before "|" or before ")"	(example: *S3* and *S5*)
• Case4: After OR group, i.e. after ")"	(example: *S6*)

Additionally, some connections may be discarded due to their positions. In order to explain our algorithms easily, we need to define our notations.

- *2ps*: two previous state/s
- ps: previous state/s
- *cs*: current state
- *ns*: next state
- *ls*: last state/s
- $p_{right(x)}$: xth BRAM output for the right character/s of a previous state/s (previous state of S1 is S0, previous states of S6 are S3 and S5)
- $p_{left(x)}$: xth BRAM output for the left character/s of a previous state/s
- $c_{right(x)}$: xth BRAM output for the right character of a current state
- $c_{left(x)}$: xth BRAM output for the left character of a current state
- $n_{left(x)}$: xth BRAM output for the left character of a next state
- $gate_y$: the yth gate of the module

In following sections, pseudocodes of the algorithms that can be used to form the modules are presented and explained.

4.3.1 Implementation of T1 module

T1 module has one OR gate and two AND gates. Whether the state register of this module (cs) is active or not is determined via the following equations.

$$cs = \ gate_1 \ OR \ gate_2$$

$$gate_1 = \ ps \ AND \ p_{right(1)} \ AND \ c_{left(2)} \ AND \ c_{right(3)}$$

$$gate_2 = \ ps \ AND \ c_{left(1)} \ AND \ c_{right(2)} \ AND \ n_{left(3)}$$

Equations for $gate_1$ and $gate_2$ change for different case and positions. These changes are explained in Algorithm-1. Match value is determined in lines 34-38. Assuming that we have "***S0 S1 (S2 S3 | S4 S5)***" regular expression (case 3) match value will be equal to "*S3 OR S5*". Or assuming that we have "***S0 S1 (S2 S3 | S4 S5)S6 S7 S8***" regular expression (case 1 or case 4) match value will be equal to "*S8*".

Algorithm-1: T1 Module Implementation Algorithm

```
1 determine state register values
2 cs = $gate_1$ OR $gate_2$
3       if position of the state = next to initial state then
5             if case1 or case3 then
6                   $gate_1$ = ps & $c_{left(2)}$ & $c_{right(3)}$
7                   $gate_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$
8             else
9                   $gate_1$ = ps & $c_{left(2)}$ & $c_{right(3)}$
10                  $gate_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$ & $n_{left(3)}$
11      else if position of the state = in the middle then
12            if case1 or case3 then
13                  if ps is even-sized then
14                        $gate_1$ = ps & $p_{right(1)}$ & $c_{left(2)}$ & $c_{right(3)}$
15                        $gate_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$
16                  else
17                        $gate_1$ = ps & $p_{left(1)}$ & $c_{left(2)}$ & $c_{right(3)}$
18                        $gate_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$
19            else
```

```
            if ps is even-sized then
                gate_1 = ps & $p_{right(1)}$ & $c_{left(2)}$ & $c_{right(3)}$
                gate_2 = ps & $c_{left(1)}$ & $c_{right(2)}$ & $n_{left(3)}$
            else
                gate_1 = ps & $p_{left(1)}$ & $c_{left(2)}$ & $c_{right(3)}$
                gate_2 = ps & $c_{left(1)}$ & $c_{right(2)}$ & $n_{left(3)}$
    else (position of the state = in the last state)
        if ps is even-sized then
            gate_1 = ps & $p_{right(1)}$ & $c_{left(2)}$ & $c_{right(3)}$
            gate_2 = ps & $c_{left(1)}$ & $c_{right(2)}$
        else
            gate_1 = ps & $p_{left(1)}$ & $c_{left(2)}$ & $c_{right(3)}$
            gate_2 = ps & $c_{left(1)}$ & $c_{right(2)}$

determine match value
    if case3 then
        match = ls x OR ls y…
    if case1 or case4 then
        match = ls
```

4.3.2 Implementation of T2 module

T2 module has one OR gate and two AND gates, whether state register of this module (cs) is active or not is determined via following equations.

$$cs = gate_1 \ OR \ gate_2$$

$$gate_1 = ps \ AND \ p_{right(1)} \ AND \ c_{left(2)}$$

$$gate_2 = ps \ AND \ c_{left(1)}$$

Equations for gate$_1$ and gate$_2$ change for different case and positions. These changes are explained in Algorithm-2.

Algorithm-2: T2 Module Implementation Algorithm

```
1 determine state register values
2 cs = $gate_1$ OR $gate_2$
3       if position of the state = next to initial state then
3             $gate_1$ = ps & $c_{left(2)}$
4             $gate_2$ = ps & $c_{left(1)}$
5       else if position of the state = in the middle then
6             if ps is even-sized then
7                   $gate_1$ = ps & $p_{right(1)}$ & $c_{left(2)}$
8                   $gate_2$ = ps & $c_{left(1)}$
9             else
10                  $gate_1$ = ps & $p_{left(1)}$ & $c_{left(2)}$
11                  $gate_2$ = ps & $c_{left(1)}$
12      else (position of the state = in the last state)
13            if ps is even-sized then
14                  $gate_1$ = ps & $p_{right(1)}$ & $c_{left(2)}$
15                  $gate_2$ = ps & $c_{left(1)}$
16            else
17                  $gate_1$ = ps & $p_{left(1)}$ & $c_{left(2)}$
18                  $gate_2$ = ps & $c_{left(1)}$
19
20 determine match value
21      if case3 then
22            match = ls x OR ls y…
23      if case1 or case4 then
24            match = ls
```

4.3.3 Implementation of T3 module

T3 module has one OR gate and four AND gates. Whether state register of T3 module (cs) is active or not is determined via following equations. $gate_4$ is not consider in this section, because output of it goes to previous state. Implementation of it will be explained in section 4.3.7.

$$cs = gate_1 \; OR \; gate_2 \; OR \; gate_3$$

$$gate_1 = ps\ AND\ c_{left(2)}\ AND\ c_{right(3)}$$

$$gate_2 = ps\ AND\ c_{right(2)}\ AND\ n_{left(3)}$$

$$gate_3 = 2ps\ AND\ p_{left(1)}\ AND\ p_{right(2)}\ AND\ c_{right(3)}$$

Equations for gate$_1$, gate$_2$ and gate$_3$ change for different case and positions. These changes are explained in Algorithm-3.

Algorithm-3: T3 Module Implementation Algorithm

```
1 determine state register values
2 cs = gate$_1$ OR gate$_2$ OR gate$_3$
3     if position of the state = next to initial state then
4         if case1 or case3 then
5             gate$_1$ = ps & $c_{left(2)}$ & $c_{right(3)}$
6             gate$_2$ = ps & $c_{right(2)}$
7         else
8             gate$_1$ = ps & $c_{left(2)}$ & $c_{right(3)}$
9             gate$_2$ = ps & $c_{right(2)}$ & $n_{left(3)}$
10    else if position of the state = in the middle then
11        if case1 or case3 then
12            if ps is even-sized then
13                gate$_1$ = ps & $c_{left(2)}$ & $c_{right(3)}$
14                gate$_2$ = ps & $c_{right(2)}$
15                gate$_3$ = 2ps & $p_{left(1)}$ & $p_{right(2)}$ & $c_{right(3)}$
16            else
17                gate$_1$ = ps & $c_{left(2)}$ & $c_{right(3)}$
18                gate$_2$ = ps & $c_{right(2)}$
19                gate$_3$ = 2ps & $p_{left(1)}$ & $c_{right(3)}$
20        else
21            if ps is even-sized then
22                gate$_1$ = ps & $c_{left(2)}$ & $c_{right(3)}$
23                gate$_2$ = ps & $c_{right(2)}$ & $n_{left(3)}$
24                gate$_3$ = 2ps & $p_{left(1)}$ & $p_{right(2)}$ & $c_{right(3)}$
25            else
26                gate$_1$ = ps & $c_{left(2)}$ & $c_{right(3)}$
27                gate$_2$ = ps & $c_{right(2)}$ & $n_{left(3)}$
```

```
                        gate_3 = 2ps & $p_{left(1)}$ & $c_{right(3)}$
    else (position of the state = in the last state)
        if ps is even-sized then
            gate_1 = ps & $c_{left(2)}$ & $c_{right(3)}$
            gate_2 = ps & $c_{right(2)}$
            gate_3 = 2ps & $p_{left(1)}$ & $p_{right(2)}$ & $c_{right(3)}$
        else
            gate_1 = ps & $c_{left(2)}$ & $c_{right(3)}$
            gate_2 = ps & $c_{right(2)}$
            gate_3 = 2ps & $p_{left(1)}$ & $c_{right(3)}$

determine match value
    if case3 then
        match = ls x OR ls y…
    if case1 or case4 then
        match = ls;
```

4.3.4 Implementation of T4 module

T4 module has one OR gate and four AND gates. Whether state register of T4 module (cs) is active or not is determined via following equations. gate$_4$ is not consider in this section, because output of it goes to previous state. Implementation of it will be explained in section 4.3.7.

$$cs = gate_1 \; OR \; gate_2 \; OR \; gate_3$$

$$gate_1 = ps \; AND \; c_{left(2)}$$

$$gate_2 = ps$$

$$gate_3 = 2ps \; AND \; p_{left(1)} \; AND \; p_{right(2)}$$

Equations for gate$_1$, gate$_2$ and gate$_3$ change for different case and positions. These changes are explained in Algorithm-4.

Algorithm-4: T4 Module Implementation Algorithm

```
1 determine state register values
2 cs = $gate_1$ OR $gate_2$ OR $gate_3$
3       if position of the state = next to initial state then
4             $gate_1$ = ps & $c_{left(2)}$
5             $gate_2$ = ps
6       else if position of the state = in the middle then
7             if ps is even-sized then
8                   $gate_1$ = ps & $c_{left(2)}$
9                   $gate_2$ = ps
10                  $gate_3$ = 2ps & $p_{left(1)}$ & $p_{right(2)}$
11            else
12                  $gate_1$ = ps & $c_{left(2)}$
13                  $gate_2$ = ps
14                  $gate_3$ = 2ps & $p_{left(1)}$
15      else (position of the state = in the last state)
16            if ps is even-sized then
17                  $gate_1$ = ps & $c_{left(2)}$
18                  $gate_2$ = ps
19                  $gate_3$ = 2ps & $p_{left(1)}$ & $p_{right(2)}$
20            else
21                  $gate_1$ = ps & $c_{left(2)}$
22                  $gate_2$ = ps
23                  $gate_3$ = 2ps & $p_{left(1)}$
24
25 determine match value
26      if case3 then
27            match = ls x OR ls y…
28      if case1 or case4 then
29            match = ls;
```

4.3.5 Implementation of T5 module

T5 module has one OR gate and four AND gates. Whether state register of T5 module (cs) is active or not is determined via following equations. $gate_4$ is not consider in this section, because output of it goes to next state. Implementation of it will be explained in section 4.3.7.

$$cs = gate_1 \; OR \; gate_2 \; OR \; gate_3$$

$$gate_1 = ps \; AND \; p_{right(1)} \; AND \; c_{left(2)}$$

$$gate_2 = ps \; AND \; c_{left(1)} \; AND \; c_{right(2)}$$

$$gate_3 = cs \; AND \; c_{right(1)} \; AND \; c_{right(2)}$$

$$gate_4 = 2ps \; AND \; p_{left(1)} \; AND \; p_{right(2)} \; AND \; c_{left(3)}$$

Equations for gate$_1$, gate$_2$ and gate$_3$ change for different case and positions. These changes are explained in Algorithm-5.

Algorithm-5: T5 Module Implementation Algorithm

```
1 determine state register values
2 cs = gate$_1$ OR gate$_2$ OR gate$_3$
3     if position of the state = next to initial state then
4         gate$_1$ = ps & $c_{left(2)}$
5         gate$_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$
6         gate$_3$ = cs & $c_{right(1)}$ & $c_{right(2)}$
7     else if position of the state = in the middle then
8         if ps is even-sized then
9             gate$_1$ = ps & $p_{right(1)}$ & $c_{left(2)}$
10            gate$_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$
11            gate$_3$ = cs & $c_{right(1)}$ & $c_{right(2)}$
                  gate$_4$ = 2ps & $p_{left(1)}$ & $p_{right(2)}$ & $c_{left(3)}$
12        else
13            gate$_1$ = ps & $p_{left(1)}$ & $c_{left(2)}$
14            gate$_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$
15            gate$_3$ = cs & $c_{right(1)}$ & $c_{right(2)}$
16            gate$_4$ = 2ps & $p_{left(1)}$ & $c_{left(3)}$
17    else (position of the state = in the last state)
18        if ps is even-sized then
19            gate$_1$ = ps & $p_{right(1)}$ & $c_{left(2)}$
```

```
20                          gate$_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$
21                          gate$_3$ = cs & $c_{right(1)}$ & $c_{right(2)}$
22                          gate$_4$ = 2ps & $p_{left(1)}$ & $p_{right(2)}$ & $c_{left(3)}$
23                  else
24                          gate$_1$ = ps & $p_{left(1)}$ & $c_{left(2)}$
25                          gate$_2$ = ps & $c_{left(1)}$ & $c_{right(2)}$
26                          gate$_3$ = cs & $c_{right(1)}$ & $c_{right(2)}$
27                          gate$_4$ = 2ps & $p_{left(1)}$ & $c_{left(3)}$
28
29 determine match value
30          if case3 then
31                  match = ls x OR ls y…
32          if case1 or case4 then
33                  match = ls;
```

4.3.6 Implementation of T6 module

T6 module has one OR gate and five AND gates. Whether state register of T6 module (cs) is active or not is determined via following equations. gate$_4$ and gate$_5$ is not consider in this section, because output of them are go to previous state and next state, respectively. Implementation of them will be explained in section 4.3.7.

$$cs = gate_1\ OR\ gate_2\ OR\ gate_3$$

$$gate_1 = cs\ AND\ c_{right(1)}\ AND\ c_{right(2)}$$

$$gate_2 = 2ps\ AND\ p_{left(1)}\ AND\ p_{right(2)}$$

$$gate_3 = ps\ AND\ (\ c_{left(2)}\ OR\ c_{right(2)}\ )$$

Equations for gate$_1$, gate$_2$ and gate$_3$ change for different case and positions. These changes are explained in Algorithm-6.

Algorithm-6: T6 Module Implementation Algorithm

```
determine state register values
cs = $gate_1$ OR $gate_2$ OR $gate_3$
    if position of the state = next to initial state then
        $gate_1 = cs \,\&\, c_{right(1)} \,\&\, c_{right(2)}$
        $gate_3 = ps\ (\, c_{left(2)} \mid c_{right(2)} \,)$
    else if position of the state = in the middle then
        if ps is even-sized then
            $gate_1 = cs \,\&\, c_{right(1)} \,\&\, c_{right(2)}$
            $gate_2 = 2ps \,\&\, p_{left(1)} \,\&\, p_{right(2)}$
            $gate_3 = ps\ (\, c_{left(2)} \mid c_{right(2)} \,)$
        else
            $gate_1 = cs \,\&\, c_{right(1)} \,\&\, c_{right(2)}$
            $gate_2 = 2ps \,\&\, p_{left(1)}$
            $gate_3 = ps\ (\, c_{left(2)} \mid c_{right(2)} \,)$
    else (position of the state = in the last state)
        if ps is even-sized then
            $gate_1 = cs \,\&\, c_{right(1)} \,\&\, c_{right(2)}$
            $gate_2 = 2ps \,\&\, p_{left(1)} \,\&\, p_{right(2)}$
            $gate_3 = ps\ (\, c_{left(2)} \mid c_{right(2)} \,)$
    else
            $gate_1 = cs \,\&\, c_{right(1)} \,\&\, c_{right(2)}$
            $gate_2 = 2ps \,\&\, p_{left(1)}$
            $gate_3 = ps\ (\, c_{left(2)} \mid c_{right(2)} \,)$

determine match value
    if case3 then
        match = ls x OR ls y…
    if case1 or case4 then
        match = ls;
```

4.3.7 Implementation of transitions connected to another modules' inputs

In this part we will explain how we implemented transitions (gate outputs) that are connected to another modules' inputs. In algorithm-7 we explain all such transitions. T3 and T4 modules have a transition to previous module. Hence, while implementing the previous module we should check whether next module is T3 or T4 module or not. If next module is one of them, we should connect this transition to previous module's OR gate of the state register. T5 module creates a transition from previous module to next module. Therefore, while implementing next module we should check whether previous module is T5 or not. If this happens, we should connect this transition to next module's OR gate of the state register. T6 module has transition to previous module and also creates a transition from previous module to next module. Therefore, while implementing previous module we should check whether next module is T6 module or not and while implementing next module we should check whether previous module is T6 module or not. If next module is T6 module we should connect this transition to previous module's OR gate of the state register. And also if previous module is T6 module we should connect this transition to next module's OR gate of the state register.

Algorithm-7: Algorithm to implement transitions connected to another modules' inputs

```
1   define previous_module, current_module, next module
2   define connection#1 (connection to previous module)
3   define connection#2 (connection from previous module to next module)
4
5   if successor module is type 3 or type 4 module then
6       connection#1 = cs & $n_{left(2)}$ & $n_{left(3)}$
7
8   if successor module is type 6 module then
9       connection#1 = cs & $n_{left(2)}$ & $n_{left(3)}$
10
11  if predecessor module is type 5 module then
12      connection#2 = 2ps & $p_{left(1)}$ & $c_{left(2)}$ & $c_{right(3)}$
13
14  if predecessor module is type 6 module then
15      connection#2 = 2ps & $c_{left(2)}$ & $c_{right(3)}$
```

CHAPTER 5

PERFORMANCE EVALUATION

In this chapter we evaluate the performance of the suggested architecture. We first provide information about the evaluation tools used in this work and explain signature data-sets that are employed in our evaluation. We then define a set of performance metrics and study the proposed architecture. We finally compare our results with Yang's work [1].

5.1 Evaluation Tools

We evaluate 2C-NFA using both Vivado and ISE Design Suites. Vivado and ISE are software used for analysis and synthesis of HDL designs. Both of them are produced by Xilinx. Vivado supersedes ISE because it has additional features for high-level synthesis. With these additional features Vivado creates more compact circuits then ISE for the same VHDL, Verilog or System Verilog codes.

5.2 Data Set

We analyze 2C-NFA using a real signature data-set and also performing worst-case computations. For the first part, we use part of the signature database of Snort Intrusion Detection System. For the second part, to perform worst-case computations, we generate a set of regular expressions whose length is Gaussian. In Section 5.2.1, we provide information about some of the features of the data-sets obtained from Snort's database and in Section 5.2.2 we explain how we generate the sample sets used in worst-case computations.

5.2.1 Real Data Set

We extract regular expressions from pcre (Perl compatible regular expression) field in Snort's signature database. There are more than ten thousand regular expressions in Snort. The aim of this work, is not to implement all regular expressions of Snort signatures but to propose a general architecture to implement any regular expression. Hence all regular expressions in Snort are not used but a sample set is formed and used in the tests. While selecting regular expressions from Snort to form the sample data set, we follow the following criteria:

1. Identical regular expressions stored in different rules are handled as a single one. Using the same regular expressions more than once inflates the number of regular expression matching engines.
2. We avoid choosing regular expressions that are too short or containing repetition of one or more characters. Similar to the previous criteria this also inflates the number of regular expression matching engines.
3. We avoid choosing regular expressions containing a large number of repetitions of a character or character groups. This inflates the number of states when implementing regular expression.

We used 1052 regular expressions from Snort database from 27 different categories. State-of-the-art FPGAs has BRAM units that has 64 bit output. Since character classifications can fit in only one BRAM unit, we partition 1052 regular expressions into 11 different sets and we compose 3 different sets of regular expressions using these sets. 1052-reme (regular expression matching engine) set contains all regular expressions. 719-reme set contains 719 regular expressions selected randomly and 569-reme set contains 569 regular expressions selected randomly. The aim in composing 3 different sets is to see how scalable 2C-NFA is with respect to the number of regular expressions. Number of states to implement these sets and state per regex values for these sets are given in ***Table 5-1***.

Table 5-1: Data-sets generated from Snort IDS

	# of regexes	*# of states*	*# of states/regexes*
1052-reme	*1052*	*10530*	*10.1*
719-reme	*719*	*8141*	*11.3*
569-reme	*569*	*6222*	*10,9*

5.2.2 Gaussian Distributed Data Set

In order to perform a worst-case performance analysis of the proposed architecture we generate variable length regular expressions using Gaussian distribution as follows:

- First, we randomly generate fixed size (100) sets of regular expressions using expected mean values of 10, 30, 50 and 100 and variance of 2.
- Second, we calculate their actual averages.

The sets formed are listed in ***Table 5-2***.

Table 5-2: Random data-sets generated via Gaussian distribution

	Average length of the regular expression
Random Set-1	*9.51*
Random Set-2	*29.54*
Random Set-3	*49.48*
Random Set-4	*99.50*

5.3 Performance Metrics

For evaluating 2C-NFA, we use following performance metrics:

- State count: Total number of states to implement regular expression.

- State fan-in: maximum number of the incoming transitions of any state.
- State fan-out: maximum number of the outgoing transitions of any state.
- LUT usage: Total number of LUTs used to implement regular expression matching engine.
- Slice usage: Total number of slices used to implement regular expression matching engine.
- BRAM usage: Total number of BRAM units used to implement regular expression matching engine.
- Throughput: Number of bits processed by the regular expression matching engine in one second.
- Clock frequency: Maximum clock frequency that regular expression matching engine runs properly.

When implementing regular expressions on hardware they are not created equal [1]. Therefore, state count, state fan-in and state fan-out values are different for all regular expressions. State fan-in and state fan-out values affect clock frequency of the regular expression matching engine. The slowest engine determines the overall clock frequency. Maximum fan-in value for 1052-reme, 719-reme and 569-reme is 12 for both 2C-NFA and Yang's architecture [1]. Maximum fan-out values are about 7 also for both approaches. In that case, fan-in value determines the overall clock frequency. All of the three sets reach up to 260 MHz clock frequency.

LUT usage, slice usage, BRAM usage, throughput and clock frequency 2C-NFA are given in Section 5.5.

5.4 Worst-case Memory Performance

In this part, we analyze worst-case LUT and best-case register usage for both approaches namely 2C-NFA and Yang's architecture. We need to make this analysis for two reasons:

i. HDL synthesis tools perform many optimizations to reduce the number of LUTs and registers while implementing the regular expression matching engine. In Yang's work, LUT per state value differs from 1.24 to 2.25 for different data-sets, but in our implementation LUT per state value turned out to be around 0.5.

ii. State-of-the-art FPGAs has 6-input LUTs. Our 2C-NFA implementation needs 1 LUT for the first state of the regular expression matching engine since fan-in value is 5 or 6. If fan-in value were 7 or 8, we would need 2 LUTs obviously. If two successive characters are the same, fan-in value decreases to 6, therefore we need only 1 LUT for such states.

For reasons stated above, we needed a worst-case memory performance evaluation using random-sets generated by using Gaussian distribution. While making this evaluation we did not take into consideration *union* and *Kleene star* operators because they affect Yang's architecture more in comparison to 2C-NFA. For a fair comparison we assume that neither *union* nor *Kleene star* operators occur in the regular expressions.

For Yang's architecture, in order to implement a regular expression which contains only concatenation, we need only 1 LUT and only 1 register for any state. On the other hand, in order to implement a regular expression which contains only concatenation in 2C-NFA we need 1 LUT and 1 register for the first state and 2 LUTs and 1 register for the other states. Using Table 5-2, we present the number of LUTs and registers needed to implement 100 regular expressions in Table 5-3. The same information exists also in ***Figure 5-1***.

Table 5-3: Worst-case memory performance

	Average length regexp length	***LUT usage (2C-NFA)***	***Register usage (2C-NFA)***	***LUT usage (Yang)***	***Register usage (Yang)***
Random Set1	*9.5*	*851.6*	*500.8*	*951.6*	*951.6*
Random Set2	*29.5*	*2854.1*	*1501.6*	*2954.1*	*2954.1*
Random Set3	*49.4*	*4848.5*	*2499.7*	*4948.5*	*4948.5*

Random Set4	*99.5*	*9850.9*	*5000.7*	*9950.9*	*9950.9*

2C-NFA needs one LUT less for each regular expression in comparison to Yang. Hence we might say that 2C-NFA does not require more LUTs but reduces register usage to nearly half.

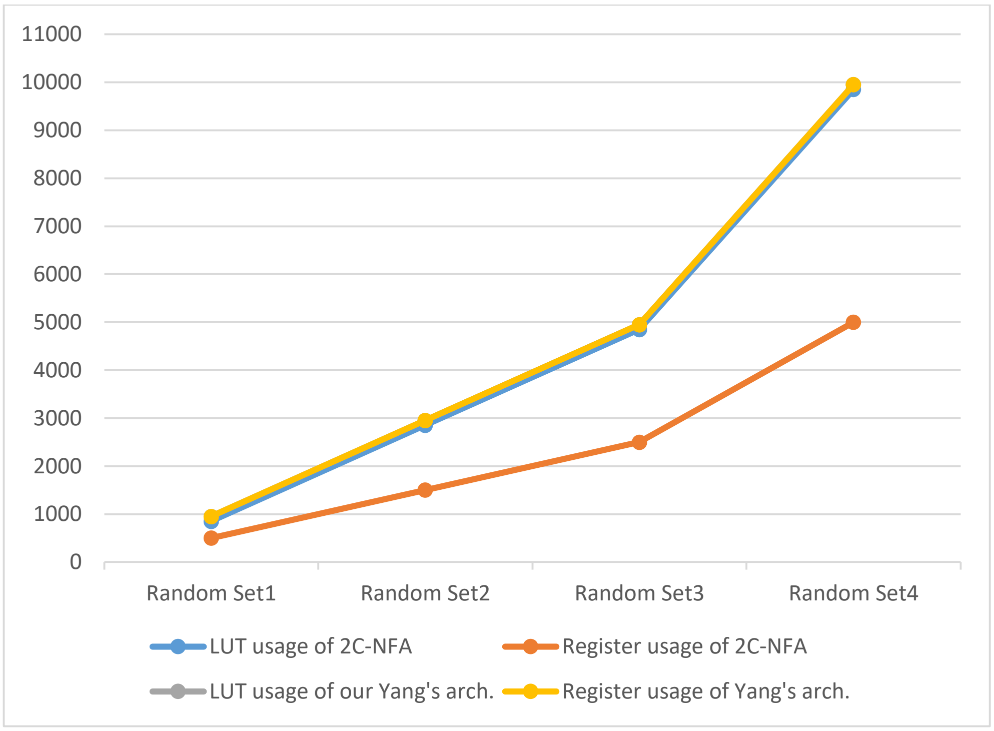


Figure 5-1: Worst-case memory performance

5.5 Implementation Results

In order to measure the other performance metrics for three sets we implemented the 1052-reme, 719-reme and 569-reme on a Xilinx Zynq-7000 series xc7z030 FPGA device. This device has 78600 6-input LUTs, 157200 registers, 19650 slices and 265 BRAM units.

While obtaining results we observed that following optimizations are done by Vivado design suite. These affect the results so we need to talk about them.

- If a state has a value of ‘1’ all the time, then any such register/flip-flop is implemented for that state. For example, initial state is always zero in 2C-NFA, hence no register is implemented for keeping this initial state.
- If the successor state of the initial has the following equation, then no LUT is implemented for state S_j ($S_{initial}$ is always ‘0’.). Only one register is implemented for S_j, and this register stores the value ‘m’.

  Sj = Sinitial & m
- While implementing two regular expressions which has a common prefix parts, these parts are implemented only once by Vivado.

  RE_1: abcd

  RE_2: abce

  S1 = S0 & a; S2 = S1 & b; S3 = S2 & c; S4 = S3 & d;

  S5 = S0 & a; S6 = S5 & b; S7 = S6 & c; S8 = S7 & e;

  S0 is always ‘1’ therefore S1 = S5 = a.

  S2 = S6 = a & b;

  S3 = S7 = a & b & c;

  We can say that these regular expressions are implemented as abc(d|e) by Vivado.
- And also when time constraints are written in the code, this forces Vivado to use less resources and to make more efficient placement & routing. Finally, we can obtain higher clock rates and more compact circuits.

First, we analyzed how 2C-NFA achieves higher throughput by using increasingly more resources. For this, we replicated 1052-reme set 1 to 7 times, 719-reme set 1 to 10 times and 569-reme set 1 to 14 times, i.e. we implemented 7 parallel circuits for 1052-reme set, 10 parallel circuits for 719-reme set and 14 parallel circuits for 569-reme set, respectively). Results are given in a ***Table 5-4***, **Table 5-5** and ***Table 5-6*** respectively.

Table 5-4: Implementation results for 1052-reme (2C-NFA)

Replication	***#LUTs***	***#Registers***	***#Slices***	***#BRAM units***	***Clock frequency (MHz)***	***Throughput (Gbps)***
x1	8916	9120	2618	33	260	4,2
x2	17820	18214	5183	66	260	8,3
x3	26733	27361	7645	99	260	12,5
x4	33530	36481	10106	132	255	16,3
x5	44380	45601	12593	165	255	20,4
x6	53311	54721	15125	198	250	24,0
x7	62166	63841	17643	231	250	28,0

Table 5-5: Implementation results for 719-reme (2C-NFA)

Replication	***#LUTs***	***#Registers***	***#Slices***	***#BRAM units***	***Clock frequency (MHz)***	***Throughput (Gbps)***
x1	6603	6717	1939	27	260	4,2
x2	13259	13433	3885	54	260	8,3
x3	19785	20149	5705	81	260	12,5
x4	26434	26865	7546	108	255	16,3
x5	33033	33581	9480	135	255	20,4
x6	39582	40297	11254	162	255	24,5
x7	46291	47013	13193	189	255	28,6
x8	52950	53729	15008	216	250	32,0
x9	59445	60445	16770	243	250	36,0
x10	69446	67289	18689	260	250	40,0

Table 5-6: Implementation results for 569-reme (2C-NFA)

Replication	***#LUTs***	***#Registers***	***#Slices***	***#BRAM units***	***Clock frequency (MHz)***	***Throughput (Gbps)***
x1	4899	5104	1430	18	260	4,2
x2	9824	10207	2826	36	260	8,3
x3	14702	15310	4188	54	255	12,2
x4	19720	20413	5645	72	255	16,3
x5	24574	25516	7003	90	250	20,0
x6	29521	30619	8413	108	245	23,5
x7	34483	35722	9829	126	245	27,4
x8	39382	40825	11192	144	248	31,7
x9	44237	45928	12484	162	248	35,7
x10	49300	51031	13914	180	245	39,2
x11	54185	56134	15212	198	243	42,8
x12	59128	61237	16614	216	240	46,1
x13	63378	66340	17253	234	230	47,8
x14	68245	71443	18582	252	230	51,5

We observe that 1052-remes run with about 260 MHz clock and achieves 4,16 Gbps throughput (***Figure 5-2***). While achieving this throughput, 11,3% of total LUTs, 5,8% of total registers, 13,3% of total slices and 12,5% of total BRAM units are used. Throughput is calculated as follows:

$$Throughput = \#characters/cycle \ * 8 \ bits/char \ \ * clock \ frequency * \#replication$$

$$Throughput = 2 \ * 8 \ bits * 0{,}260 \ GHz * 1 = 4{,}16 \ Gbps$$

When this circuit is replicated 7 times, clock frequency of the circuit decreases to 250 MHz and we obtain 28 Gbps throughput. In that case, resulting circuit uses 79% of total LUTs, 40,6% of total registers, 89,8% of total slices and 87,1% of total BRAM units. We cannot replicate it one more time because FPGAs slice resources are exhausted and are not sufficient for further parallelization.

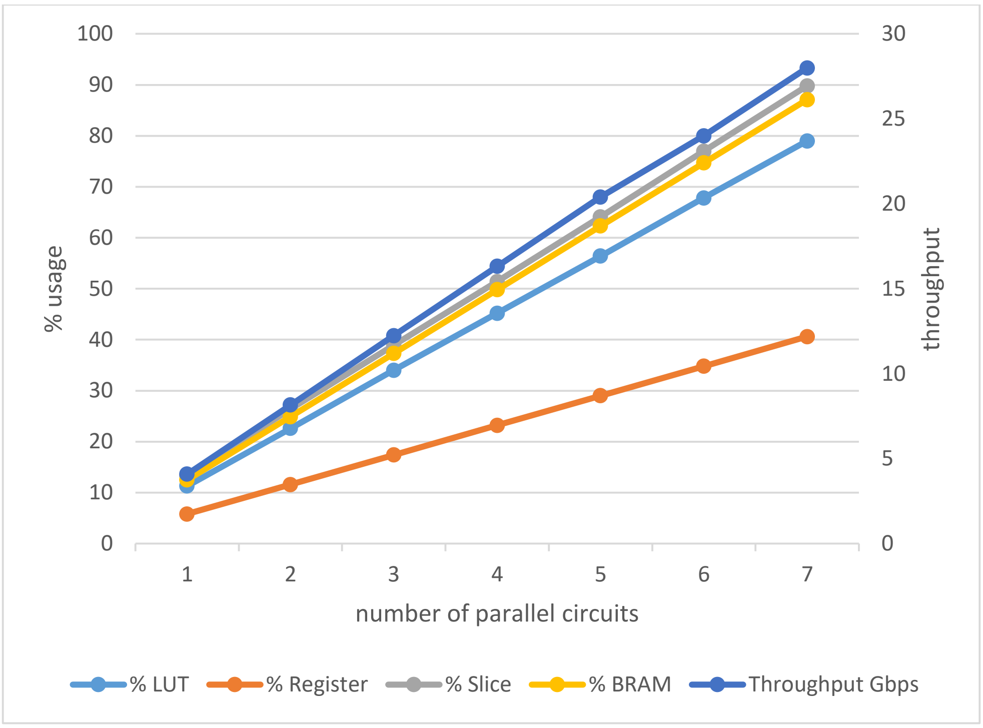


Figure 5-2: Throughput scaling of 1052-reme (2C-NFA)

We also analyze the throughput of Yang's architecture using the same 3 regular expression sets for comparison purpose. We replicate 1052-reme set for 1 to 6 times, 719-reme set for 1 to 7 times and 569-reme set for 1 to 11 times, i.e. we implement parallel circuits up to 6 times for 1052-reme, up to 7 times for 719-reme and up to 11 times for 569-reme sets, respectively). Results are presented in ***Table 5-7***, ***Table 5-8*** and ***Table 5-9***, respectively.

Table 5-7: Implementation results for 1052-reme (Yang's arch.)

Replication	***#LUTs***	***#Registers***	***#Slices***	***#BRAM units***	***Clock frequency (MHz)***	***Throughput (Gbps)***
x1	9098	13018	3343	22	260	4,2
x2	18193	26037	6679	44	260	8,3
x3	27297	39056	10036	66	255	12,2
x4	36400	52075	13372	88	252	16,1
x5	45502	65086	16719	110	252	20,1
x6	54595	78103	18948	132	250	24,0

Table 5-8: Implementation results for 719-reme (Yang's arch.)

Replication	*#LUTs*	*#Registers*	*#Slices*	*#BRAM units*	*Clock frequency (MHz)*	*Throughput (Gbps)*
x1	7237	10069	2751	18	260	4,2
x2	14483	20137	5280	36	260	8,3
x3	21764	30205	8004	54	257	12,3
x4	29956	40275	10673	72	257	16,4
x5	36198	50343	13342	90	253	20,2
x6	43434	60413	15820	108	250	24,0
x7	50756	70477	18060	126	250	28,0

Table 5-9: Implementation results for 569-reme (Yang's arch.)

Replication	*#LUTs*	*#Registers*	*#Slices*	*#BRAM units*	*Clock frequency (MHz)*	*Throughput (Gbps)*
x1	5395	7483	1839	12	260	4,2
x2	10823	14965	3961	24	260	8,3
x3	16228	22447	5820	36	256	12,3
x4	21624	29929	7776	48	256	16,4
x5	27043	37411	9967	60	256	20,5
x6	32336	44893	12026	72	250	24,0
x7	37709	52375	14295	84	250	28,0
x8	43261	59857	15447	96	250	32,0
x9	48684	67339	17424	108	250	36,0
x10	54061	74821	18608	120	250	40,0
x11	55122	82300	19052	132	243	42,8

When we implemented 1052-reme with Yang's approach, we observe that the resulting circuit runs with about 260 MHz clock, and achieves 4,16 Gbps throughput (***Figure***

5-3) 11,6% of total LUTs, 8,3% of total registers, 17,0% of total slices and 8,3% of total BRAM units are consumed to implement 1052-reme circuit. We can implement 6 parallel 1052-reme circuits on FPGA device, then clock frequency of the circuit decreases to 250 MHz and we obtain 24 Gbps throughput. In that case, it uses 69,4% of total LUTs, 49,7% of total registers, 96,4% of total slices and 49,8% of total BRAM units. This is the end of parallelization for this device similarly.

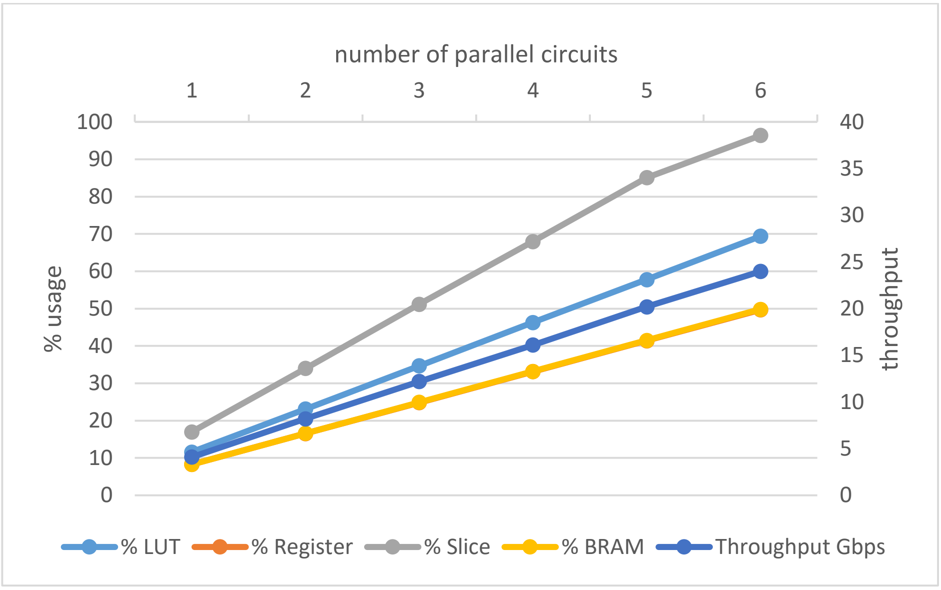


Figure 5-3: Throughput scaling of 1052-reme set (Yang's arch.)

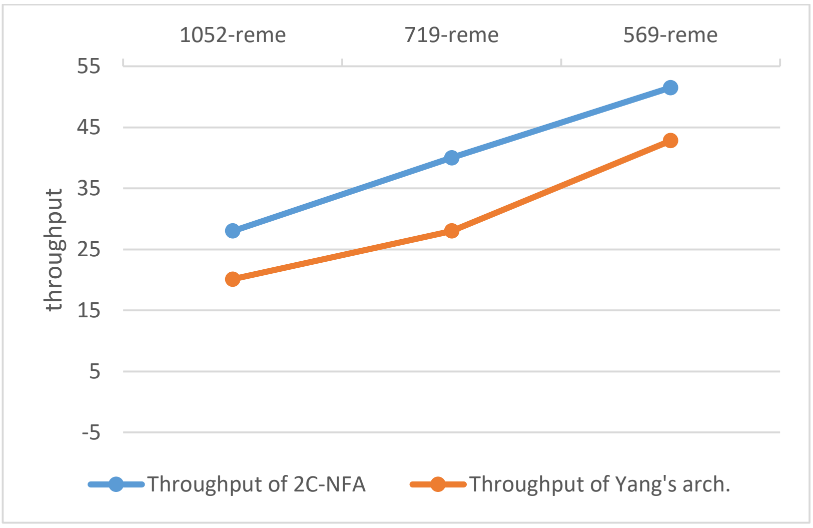


Figure 5-4: Achievable throughput for three sets

Achievable throughputs for both 2C-NFA and Yang’s architecture are shown in ***Figure 5-4***. 2C-NFA have better throughput for all sets.

Average length of the regular expressions for all sets are shown in the ***Table 5-10*** below. 569-REME and 719-REME sets are subset of the 1052-REME. We can say that 1052-REME set has shorter regular expressions than the others. If any set contains longer regular expressions this increases the memory cost of the implementation, and we can implement less number of parallel circuits therefore achievable throughput value decreases. And if any set contains shorter regexes this reduces the memory cost, thus we can obtain higher throughput with more parallelization.

Table 5-10: Average length of the regular expressions

	length (character)
569-REME	19,2
719-REME	20,0
1052-REME	17,1

In this paragraph we will discuss whether all regular expressions of the Snort database can be implemented on a single FPGA chip or not, and if this is possible what will be the throughput of such an implementation. When we consider the xc7z030 FPGA device which has 265 block RAM units each has a 64-bits data output, 265x64 = 16960 distinct character classes can be implemented on this device. In terms of BRAM capacity, we can say that all Snort rules can be implemented on a single FPGA device such as an xc7z030. Snort has about 14k regular expressions together with ‘deleted’ category contains old regular expressions, and some of the regular expressions has repeated pieces up to 1024 times. Therefore all of the Snort rules cannot be implemented on a small FPGA devices which has less resources. Nowadays, FPGA devices which has 500k LUTs and 1M registers are available. We observe that 1052-REME has 6222 states and is implemented using approximately 9000 LUT’s and registers. If average length of all of the 14k Snort regexes is 17 states as like 1052-REME, we need 238k states in order to implement these rules, i.e. we need approximately 357k LUT’s and registers. Therefore, we can say that all of the Snort rules can be implemented on a single chip if the above case is satisfied. Because of the parallelization is not possible for such an implementation, we can obtain up to 16xclock frequency throughput (2 characters x 8-bits characters x 1 replication x clock frequency).

Discussion about clock frequency will be done in this paragraph. State fan-in and fan-out values affects the reachable clock frequency. We observe that state fan-in value is 12 for both 2C-NFA and Yang’s architectures. This means that values of some states are calculated using 2 6-input LUTs. If fan-in value is between 13-18, in that case calculation is done using 3 LUTs and this process needs more time than the above case therefore clock frequency decreases. Fan-out values for both of the architectures are about 8. Also placement & routing processes affects the reachable clock frequency. Synthesizers makes optimizations to achieve higher clock frequencies when implementing the circuitry on a device.

Finally, we can summarize our results below:

- 2C-NFA architecture achieves at most 51Gbps throughput when 14 parallel 569-reme circuits are implemented. 1052-reme achieves 28 Gbps throughput with 7 replications. 719-reme achieves 40 Gbps throughput with 7 times parallelization.
- Yang’s NFA architecture achieves at most 42 Gbps throughput when 11 parallel 569-reme are implemented. 1052-reme achieves 24 Gbps throughput with 7 replications. 719-reme achieves 28 Gbps throughput with 7 replications.
- In implementation, if block RAMs are not sufficient, design tool utilizes from LUTRAMs (LUTs that can be used as memory units).
- As resource consumption increases on FPGA device, it is observed that the clock frequency decreases. Minimum clock frequency, 230 MHz, is obtained when implementing 569-reme set with our 2C-NFA architecture.

5.6 Comparison

In this part, first we compare memory costs for both architectures. LUT usage of 2C-NFA does not exceed Yang's and it nearly halves the number of registers required.

We then analyzed the number of states that are used to represent datasets. Results are given in ***Table 5-11***. We observe that 2C-NFA needs 41,7% less number of states to represent 1052-reme set and needs 43,5% less number of states to represent 719-reme and 569-reme sets.

Table 5-11: Number of states to represent datasets for both approaches

	# of regexes with 2C-NFA	# of regexes with Yang's approach
1052-reme	10530	18043
719-reme	8141	14368
569-reme	6222	10975

LUT/state and slice/state values are also analyzed. Results are presented in Table 5-12. One can easily conclude that 2C-NFA performs worse in comparison to Yang's architecture because the latter less slice resource and LUT to store a state on FPGA. Although 2C-NFA seems to be performing poor in these two resource aspects, the number of states to represent the datasets is reduced considerably in 2C-NFA. Therefore, it needs less slice resource to implement the whole dataset on the FPGA device. Consequently, we may conclude that 2C-NFA is more compact than Yang's. For example, 719-reme needs 14368*0,19=2729 slices and 8141*0,24=1953 slices to implement on FPGA device, respectively. We observe that ***Table 5-8*** and ***Table 5-5*** are consistent in this respect.

Table 5-12: FPGA Slices per State and LUTs per state values for both approaches

	Slice/State		LUT/state	
	Our app.	Yang’s app.	Our app.	Yang’s app.
1052-reme	0,25	0,19	0,85	0,50
719-reme	0,24	0,19	0,81	0,50
569-reme	0,23	0,17	0,78	0,49

Finally, we compare consumption of BRAM units. Because of the need for a 3-character window search mechanism, 2C-NFA needs 3-character match signals, therefore we need 3 BRAM units. On the other hand, Yang’s architecture needs 2 BRAM units in order to implement a character class. We create 11, 9 and 6 different character classes to implement 1052-reme, 719-reme and 569-reme sets, respectively. Hence to implement only one 1052-reme circuit, we need 33 and 22 BRAM units for 2C-NFA and Yang, respectively.

We may conclude that while the limiting factor of 2C-NFA is BRAM usage, the limiting factor for Yang’s architecture is slice usage.

1052-reme implementation achieves 28 Gbps throughput with 2C-NFA approach and 24 Gbps throughput with Yang’s architecture. Hence, we achieved approximately 16% of higher throughput in comparison to the latter for the corresponding data set. For 719-reme implementation, improvement is approximately 42% (40 Gbps instead of 28 Gbps) while for 569-reme it is approximately 20% (51 Gbps instead of 42 Gbps).

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, we proposed a i) novel ii) modular iii) compact iv) nondeterministic finite automata based v) memory-efficient vi) high-performance regular expression matching engine, which is suitable to be implemented on FPGAs. We utilized modified McNaughton-Yamada algorithm to create the matching circuit as in [1]. However, the memory-efficient nature of our approach helps us to implement more parallel circuits on a given FPGA device and therefore achieve higher throughputs.

2C-NFA does not support runtime updates. When we want to implement a new regular expression, there is a need for off-line processing to construct and add the new NFA and hence the final circuitry after which reconfiguration of the FPGA is required.

With a better grouping of characters, the number of character classes used in the architecture is reduced and as a result the number of block RAM units required is also reduced.

While FPGA technology is continuing to be improved, when more than 6-input LUTs will common in FPGAs, memory usage for our proposed 2C-NFA will be reduced even further in comparison to Yang's approach and hence even larger throughputs will be possible in the future.

As short term future work, higher throughput may be aimed by using a multi 2-character (4-character, 6-character etc.) architecture first and then performing further optimizations aiming less LUT usage.

REFERENCES

[1] Y.E. Yang, W. Jiang, V.K. Prasanna, “Compact architecture for high-throughput regular expression matching on FPGA,” *Proc. IEEE Symp. Arch. Net. Comm. Sys.*, pp. 30-39, 2008.

[2] Y.E. Yang, V.K. Prasanna, "Robust and scalable string pattern matching for deep packet inspection on multicore processors," *IEEE Trans. Paral. Dist. Sys.*, pp. 2283-2292, 2012.

[3] C-L. Lee, T-H. Yang, "A flexible pattern-matching algorithm for network intrusion detection systems using multi-core processors," *Algorithms*, vol. 10, pp. 58, 2017.

[4] O. Erdem, "Tree-based string pattern matching on FPGAs," *Comp. Elec. Eng.*, vol. 49, pp. 117-133, 2016.

[5] R. Sidhu, V.K. Prasanna, "Fast regular expression matching using FPGAs," *Proc. IEEE Symp. Field-Prog. Cust. Comp. Mach.*, pp. 227-238, 2001.

[6] I. Sourdis, J. Bispo, J.M. Cardoso, S. Vassiliadis, "Regular expression matching in reconfigurable hardware," *J. Sig. Proc. Sys.*, vol. 51, pp. 99-121, 2008.

[7] P. Dlugosch, D. Brown, P. Glendending, M. Leventhal, H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Trans. Paral. Dist. Sys.*, vol. 25, pp. 3088-3098, 2014.

[8] K. Peng, S. Tang, M. Chen, Q. Dong, "Chain-based DFA deflation for fast and scalable regular expression matching using TCAM," *Proc. ACM/IEEE Symp. Arch. Net. Commun. Sys.*, pp. 24-35, 2011.

[9] A-S. K. Pathan (ed.), *The state of the art intrusion prevention and detection*, CRC Press, 2014.

[10] R. Beghdad, "Critical study of neural networks in detecting intrusions," *Comp. & Secur.*, vol. 27, pp. 168-175, 2008.

[11] M. Gupta, "Hybrid intrusion detection system: Technology and Development," *Int. J. Comp. App.*, vol. 115, pp. 5-8, 2015.

[12] Snort. *accessed at July, 2018,* from http://www.snort.org/

[13] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Comp. Net.*, vol. 31, pp. 2435-2463, 1999.

[14] S. Wu, U. Manber, "A fast algorithm for multi-pattern searching," *Tech. R., Comp. Sci., Uni. Arizona,* 1994.

[15] N. Tuck, T. Sherwood, B. Calder, G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *Proc. IEEE Int. Conf. Comm.*, vol. 4, pp. 333-340, 2004.

[16] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," *Proc. IEEE Symp. Arch. Net. Comm. Sys.*, pp. 93-102, 2006.

[17] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, J. S. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *Proc. App. Tech. Arch. Pro. Comp. Comm.*, vol. 36, pp. 339-350, 2006.

[18] C. R. Clark, D. E. Schimmel, "Scalable pattern matching for high speed networks," *Proc. IEEE Sym. Field-Prog. Cust. Comp. Mach.*, pp. 249-257, 2004.

[19] L. Yang, R. Karim, V. Ganapathy, R. Smith, "Fast, memory-efficient regular expression matching with NFA-OBDDs," *Comp. Net*., vol. 55, pp. 3376-3393, 2011.

[20] Y. Sun, V. C. Valgenti, M. S. Kim, "NFA-based pattern matching for deep packet inspection," *Proc. IEEE Int. Conf. Comp. Comm. Net.*, pp. 1-6, 2011.

[21] S. Kumar, J. Turner, J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," *Proc. IEEE Symp. Arch. Net. Comm. Sys.*, pp. 81-92, 2006.

[22] L. Vespa, N. Weng, R. Ramaswamy, "MS-DFA: Multiple-stride pattern matching for scalable deep packet inspection," *Comp. J., vol. 54, pp. 285-303, 2011.*

[23] H. Kim, S-W. Lee, "A hardware-based string matching using state transition compression for deep packet inspection," *ETRI J.*, vol. 35, pp. 154-157, 2013.

[24] J. Yang, L. Jiang, Q. Tang, Q. Dai, J. Tan, "PiDFA: A practical multi-stride regular expression matching engine based on FPGA," *Proc. IEEE Int. Conf. Comm.*, pp. 1-7, 2016.

[25] M. Becchi, P. Crowley, "A hybrid finite automaton for practical deep packet inspection," *Proc. IEEE Int. Conf. Emer. Net. Exp. Tech,* pp. 1-12, 2007.

[26] Y-H. E. Yang, V. K. Prasanna, "Space-time tradeoff in regular expression matching with semi-deterministic finite automata," *Proc. IEEE Int. Conf. Comp. Comm.*, pp. 1853-1861, 2011.

[27] Y. Xu, J. Jiang, R. Wei, Y. Song, H. J. Chao, "TFA: A tunable finite automaton for pattern matching in network intrusion detection systems," *IEEE J. Sel. Area. Comm.*, vol. 30, pp. 1810-1821, 2014.

[28] B. Modi, G. Tripp, "A highly compressible regular expression matching circuit for network intrusion detection systems: An ECD-NFA approach," *IOSR J. VLSI Sig. Proc.*, vol. 6, pp. 50-58, 2016.

[29] K. Atasu, "Resource-efficient regular expression matching architecture for text analytics," *Proc. IEEE Int. Conf. App-Spec. Sys Arch. Proc. Proc.*, pp. 1-8, 2014.

[30] D. Pao, N. Lam, R. C. C. Cheung, "A memory-based NFA regular expression match engine for signature-based intrusion detection," *Comp. Comm.*, vol. 36, pp. 1255-1267, 2013.

[31] T. T. Hieu, T. N. Thinh, S. Tomiyama, "ENREM: An efficient NFA-based regular expression matching engine on reconfigurable hardware for NIDS," *J. Sys. Arch.*, vol. 59, pp. 202-212, 2013.

[32] Y-H. E. Yang, V. K. Prasanna, "Optimizing regular expression matching with SR-NFA on multi-core systems," *Proc. IEEE Int. Conf. Par. Arch. Comp. Tech.*, pp. 424-433, 2011.

[33] Y-H. Yang, V. K. Prasanna, "High-performance and compact architecture for regular expression matching on FPGA," *IEEE Trans. Comp.*, vol. 61, pp. 1013-1025, 2012.

[34] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, Q. Dong , "GPU-based NFA implementation for memory efficient high speed regular expression matching," *Proc. Prin. Prac. Par. Prog.*, pp. 129-139, 2012.

[35] H. Wang, S. Pu, G. Knezek, J-C. Liu, "A modular NFA architecture for regular expression matching," *Proc. Int. Sym. Field-Prog. Gate Arr.*, pp. 209-218, 2010.

[36] N. Nakahara, T. Sasao, M. Matsuura, "A regular expression matching using non-deterministic finite automaton," *Proc. IEEE Int. Conf. Form. Met. Mod. Code.*, pp. 73-76, 2010.

[37] T-H. Lee, "Hardware architecture for high-performance regular expression matching," *IEEE Trans. Comp.*, vol. 58, pp. 984-993, 2009.

[38] N. Yamagaki, R. Sidhu, S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," *Proc. IEEE Int. Conf. Field Prog. Log. App.*, pp. 131-136, 2008.

[39] A. Mitra, W. Najjar, L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," *Proc. IEEE Symp. Arch. Net. Comm. Sys.*, pp. 127-136, 2007.

[40] C. Xu, S. Chen, J. Su, S. M. Yiu, L. C. K. Hui, "A survey on regular expression matching for deep packet inspection: applications, algorithms, and hardware platforms." IEEE Comm. Sur. & Tut., vol. 18, pp. 2991-3029, 2016.