A Reconfigurable Regular Expression Accelerator for PROSITE Patterns on FPGAs

Semester Project
VLSC Lab

Zoé Baraschi

Computer Science Department
Swiss Federal Institute of Technology Lausanne
June 2019

Contents

1 Introduction

As today's technological advances give us access to ever so affordable DNA sequencing, a large interest on how to efficiently process this data has surfaced. The inspiration for our project comes from the paper "Towards in the Field Fast Pathogens Detection Using FPGAs" by E. Bezati [6]. This paper describes a solution for a fast pathogen detection embedded hardware accelerator. It can analyse up to 100 different pathogens at the same time using one single embedded accelerator. The matching process in this implementation is very fast (one nucleotide per clock cycle) but the system is not reconfigurable. This means that for every database, sets of Regular Expression circuits need to be generated and synthesised, which takes a lot of time.

We thus wanted to find a reconfigurable approach, for which the circuit did not need to be regenerated at every change of database. This would give us the possibility to use a PROSITE search in real time. Furthermore, we also wanted an approach which was as fast as a hardcoded circuit. Our aim was to replace the hard-coded regular expressions in [6] with our new implementation. To do so, we have thoroughly investigated the literature and have found that techniques used for Intrusion Detection Systems, which monitor networks or systems for malicious activity or policy violations, could be applied to finding patterns in proteins. Our first approach was to implement the architecture described in [16] using the CAL programming language [7][12]. After working on it for some time, we realised that this approach was not reconfigurable enough to our liking, since the circuit had to be regenerated when new rules were added to our rule set. We thus found a more fitting approach defined in the paper "Regular Expression Software Deceleration for Intrusion Detection Systems" by Z. K. Baker, H.-J. Jung and V. K. Prasanna [4]. The following articles are complementary to the previous paper and crucial for its understanding [11][13][17].

This report is an in-depth study of the approach described in [4] and how it can be applied to proteins, using PROSITE patterns [1], which can be viewed as specialised Regular Expressions [10] for proteins. The report follows a bottom-up approach, first giving understanding on why regular expressions are used and how to modify them in order to use them in our approach. Then, techniques on how to optimize memory storage are given. One technique in particular, the bit-splitting algorithm, has been explored in detail and implemented fully. Furthermore, software-based techniques to optimize performance, such as the use of micro-controllers, are discussed. Finally, we give insight on how this approach would be applied to hardware.

2 Architecture Implementation Details

2.1 Regular Expressions

Regular expressions [10] are sequences of characters that define a search pattern. Usually such patterns are used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation. They have been particularly usefull to find harmful and malicious sequences in code and this is why they are the subject of many studies on improving Intrusion Detection Systems.

2.2 PROSITE

PROSITE [1] is a method of determining what is the function of uncharacterized proteins translated from genomic or cDNA sequences. It consists of a database of biologically significant sites and patterns formulated in such a way that with appropriate computational tools it can rapidly and reliably identify which known family of protein (if any) the new sequence belongs to. Similary to regular expressions, PROSITE enables us to find patterns in unknown proteins and helps us to link these proteins to known ones. It has become an essential tool for protein sequence analysis. We can find the pattern rules in Table 1.

PROSITE [1]	
Pattern	Description
A	Standard IUPAC one-letter codes for the amino-acids used (c.f. Appendix A.2).
x	Position where any amino acid is accepted.
[]	Acceptable amino acids for a given position. i.e. [ABC] stands for A or B or C.
{}	Amino acids which are not acceptable at this position. i.e. {AB} stands for any amino acid except A or B.
-	Each element in a pattern is separated from its neighbor using a '-'.
x(value)	Exact numerical repetition. i.e. x(2) stands for x-x.
x(value1, value2)	Numerical range repetition. i.e. x(0,3) stands for nothing or x or x-x or x-x-x.
.	A period ends the pattern.

Table 1: Table of PROSITE patterns

2.3 Regular Expression to DFA

In order to efficiently interpret Regular Expressions at runtime, we need to convert it to an intermediate form. Regular Expressions are closely related to Nondeterministic Finite Automata (NFAs), which can then be easily converted into DFAs. The advantage of DFAs is that only one state is active at any time, so there is only one step to compute in each step. To transform a Regular Expression into a DFA, the following steps have to be made.

First, with the help of Thompson's construction [3] (Chapter 3.7.4) we are able to transform a RegEx into a NFA. The algorithm first parses the RegEx into its subexpressions. Then, the NFA is constructed from the basis and is enlarged by the inductive rules.
A quick recap of the construction rules can be found in the Appendix A.2.

The proof can be found in the book [3].

Once this is done, we do a subset construction to transform the NFA into a DFA [3] (Chapter 3.7.1). We will use the same method to transform PROSITE patterns into DFA, and illustrate this with an example.

Assume our pattern is:

x(2)-V-[LS]-C-{EPFL}(0,1).

It is equivalent to:

x - x - V - [L or S] - C - any except E,P,F or L (zero or once).

Using Thompson's construction algorithm, we get the following NFA:

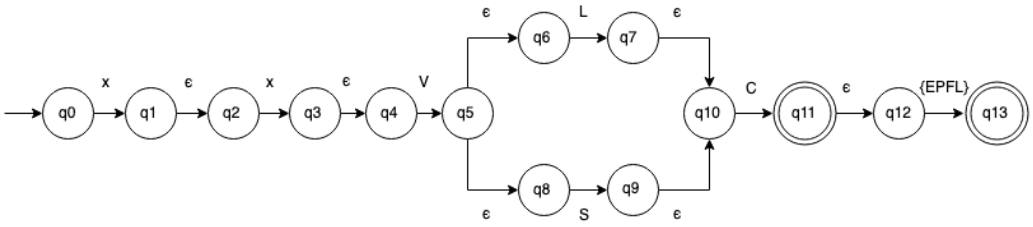


Figure 1: NFA for the PROSITE pattern x(2)-V-[LS]-C-{EPFL}(0,1).

Note that the start state is q_0 and the accepting states are the ones with the double circle. After subset construction, we get the following DFA:

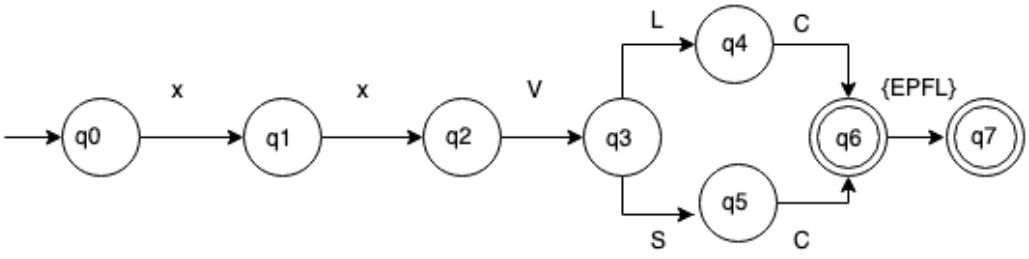


Figure 2: DFA for the PROSITE pattern x(2)-V-[LS]-C-{EPFL}(0,1).

Since the PROSITE alphabet consists of many characters, we do not show the edges leading back to the start state for simplicity. These edges represent an inputed character which diverges from the pattern sequence and lead us back to the start of our pattern search.

2.4 DFA to Modified DFA

In [4], the authors talk about modifying the DFAs in order to recognize the individual pattern segments. In "classic" regular expressions, a group of characters or regular expressions may be subject to repetition or wild-cards. i.e. for the regular expression $((c(a|b))*)|((de)+)$, the first and second group may be repeated infinitely many times, and matches would be found for strings such as $ca, de, cade, cbdede$ etc. Thus, it important to annotate patterns to receive information about which pattern segment was matched (first segment, second segment, or both segments).

With PROSITE, we do not have more than one amino acid which can be subject to operators at a time. For instance, $[AB](0,1) - E(0,1)$, matches strings a, b, ae, be, e and the empty string. This is a theoretical possibility.
In order to evaluate actual patterns, we scraped the PROSITE database and sampled 1294 patterns. Out of these, $\sim$74% had some elements of repetition. Less than 5 % had a repetition indicating the range 0 or more characters. Finally, less than 1% of patterns had more than one range repetition of 0 or more characters. These can be seen in the table below. We notice that none of these patterns have chained repetitions, and that each repetition has a sequence of characters preceding and following it. From this sample, we never reach the case illustrated in the above paragraph.

It is thus debatable whether the pattern annotation is a necessity for PROSITE patterns. The pattern sampling methodology can be found in the Appendix A.3.

Pattern	Repetitions
[GD]-x(0,10)-[FYWA]-x(0,1)-G-[LIVM]-x(0,2)-[LIVMFYD]-x(0,7)-G-[KN]-[NHW]-x(0,1)-G-[STARCV]-x(0,2)-[GD]-x(0,2)-[LY]-[FC]	x(0,10), x(0,1), x(0,2), x(0,7), x(0,1), x(0,2), x(0,2)
C-C-[SHYN]-x(0,1)-[PRG]-[RPATV]-C-[ARMFTNHG]-x(0,4)-[QWHDGENFYVP]-[RIVYLGSDW]-C	x(0,1), x(0,4)
C-C-[TGN]-[PFG]-[PRG]-x(0,2)-C-[KRS]-[DS]-[RK]-[RQW]-C-[KR]-[PD]-[MLQH]-x(0,1)-[KR]-C-C	x(0,2), x(0,1)
C-[SREYKLIMQVN]-x(2)-[DGWET]-x-[FYSPKV]-C-[GNDSRHTP]-x(1,5)-[NPGSMTAHF]-[GWPNIYRSKLQ]-x-C-C-[STRHGD]-x(0,2)-[NFLWSRYIT]-C-x(0,3)-[VFGAITSNRKL]-[FLIKRNGH]-[VWIARKF]-C	x(0,2), x(0,3)
C-x(0,1)-[ES]-S-C-[AV]-[MFYW]-I-[PS]-x(0,1)-C	x(0,1),x(0,1)
[FWV]-x(0,1)-[LIVM]-D-P-[LIVM]-D-[SG]-[ST]-x(2)-[FYA]-x(0,1)-[HKRNSTY]	x(0,1), x(0,1)
[GSA]-[ATIVS]-[LIVMYCAFST]-K-[DN]-[LIVMA]-[LIVMFYT]-[GA]-x-[GACKMSIFT]-x-G-[ALIVMF]-x(2)-[SGAQ]-[LIVMYERAKQFS]-x(0,1)-[TLIVMFYWAQ]-[ETGAS]-x(0,1)-[NDVS]	x(0,1), x(0,1)
[GA]-x(0,2)-[YSA]-x(0,1)-[VFY]-SEDT-C-x(1,2)-[PG]-x(0,1)-H-x(2,4)-[MQ]	x(0,2), x(0,1), x(0,1)

Table 2: Example of PROSITE patterns with multiple range repetition starting with zero

2.5 Aho-Corasick Algorithm and Bit-Splitting

2.5.1 Aho-Corasick

The Aho-Corasick Algorithm [2] was first introduced in [13]. The algorithm includes a preprocessing step which encodes all of the strings to be searched. The output of the algorithm is a finite state machine, which is generated into two stages. The first stage builds a tree of all strings that need to be indentified in the input stream. The second second stage consists of inserting failure edges. When a string match is not found, it is possible for the suffix of one string to match the prefix of another. Failure edges represent the transition from a partial match of one string to the partial match of another.
Let's illustrate this with an example.

Given the set of strings to be matched: {he, she, his, hers}, we build the following tree of strings needing to be identified:

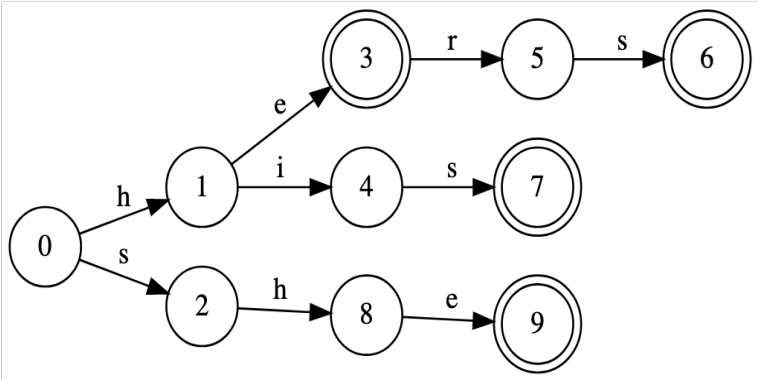


Figure 3: Tree of the strings needed to be identified for the set {he, she, his, hers}

We then insert failure edges and produce the following finite state machine:

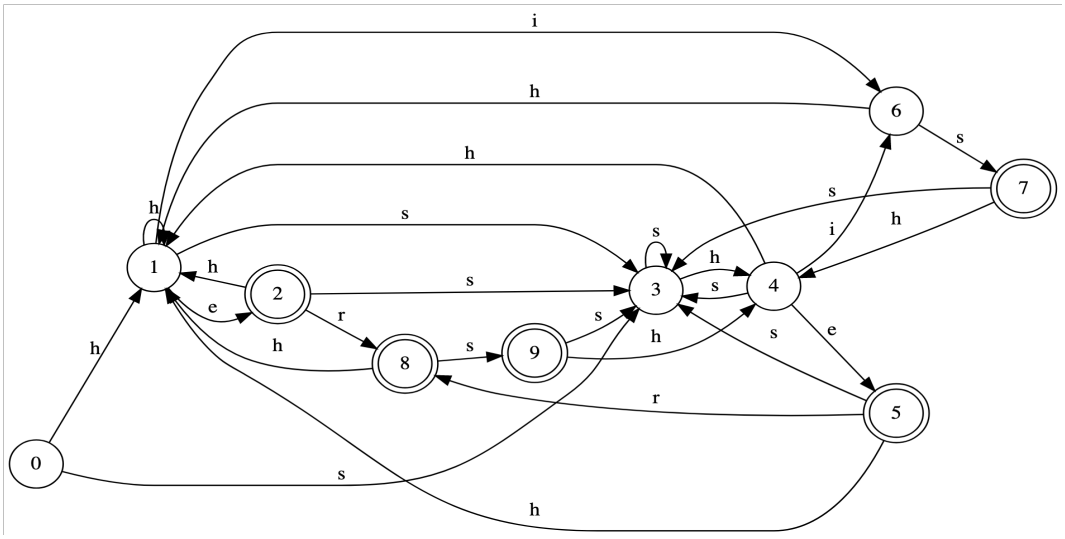


Figure 4: FSM resulting from the Aho-Corasick algorithm for the set {he, she, his, hers}

The advantage of this approach is that after the preprocessing of the strings, the algorithm always runs in time linear to the length of the input stream.

Although the Aho-Corasick algorithm has many advantages, one of its main drawbacks is the enormous amount of memory it requires. Each state's possible outgoing edge needs to be tracked, which equals to one edge per character of the DFA's alphabet. There is also a high variation of the number of pointers stored for each state. The ones which are closer to the root of the tree will have an enourmous amount of pointers, whereas the ones near the leafs will only have a few. Thus, having a fixed array of pointers equal to the maximum number of possible next states is very wasteful.

The second drawback is keeping track of the next state we are going to go to. This leads us to enter a critical loop, in which the next state could be one of many memory locations (as many as the number of characters in the DFA's alphabet). It is thus very difficult to make this fast.

To solve the problems of the Aho-Corasick algorithm, the bit-splitting method has been introduced in [13] and will be discussed in the next section.

2.5.2 Bit-Splitting

In [13], bit-splitting is defined as a procedure which consists of splitting each state machine into a new set of 8 state machines. Each state machine is responsible for only one of the 8 bits of an input character (in ASCII, 1 character corresponds to 8 bits). In [11], bit-splitting evolves by splitting each state machine into a set of 4 new state machines, each one of them being responsible for two bits. In order for the reader to easily grasp the concept of bit-splitting, we will start by illustrating the approach with 8 state machines and then go on to give an example with 4 state machines.

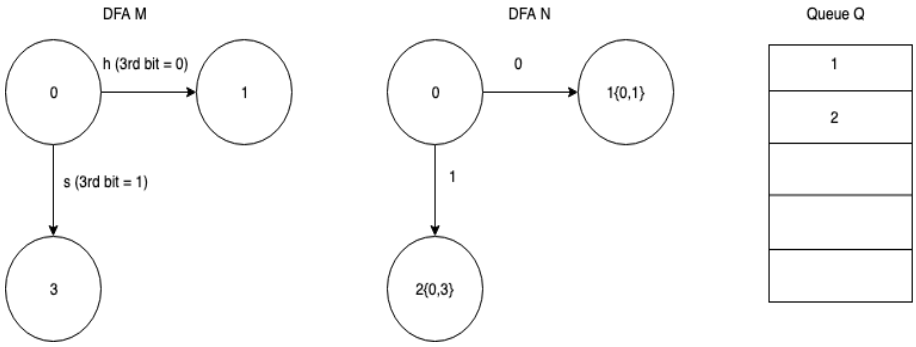


Figure 5: First Step of Bit-Splitting Algorithm for DFA 4

Let us create the automaton for bit 3. The first step of bit-splitting is to create a new DFA N, with starting state 0. We visit the starting state of the original DFA M and observe all of its outgoing edges. We then create two sets: the set of states which are reached by reading a 0 in the 3rd bit of the

label value, and the set of states which are reached by reading a 1 in the 3rd bits of the label value. For each outgoing edge, we look at the 3rd bit of the label (in [13], the MSB is bit 0) and place the target state in the appropriate set. Finally, we create two new states in DFA N, with their corresponding sets of states in M and add an edge from the starting state to each new state, with the correspnding label value. We then add the states to a queue Q of states to visit. This step can be seen in figure 5.

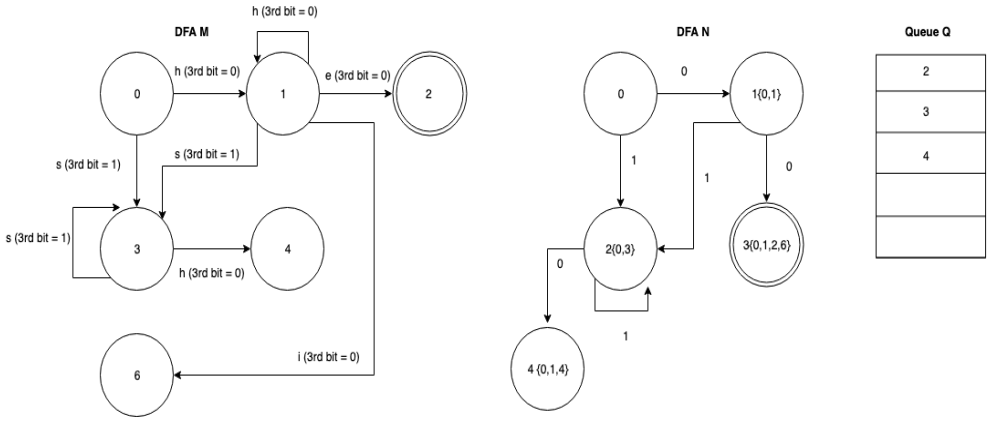


Figure 6: Second Step of Bit-Splitting Algorithm for DFA in Figure 4

The second step consists of removing a state S from the queue Q and looking at its set of states E, representing states of DFA M. For each state in E, we look at the outgoing edges and place the reachable states into sets, as in the first step. We then check if there already exists a state R in N with the same set of reachable states and the corresponding incoming label value. If not, we create a new state T in N. We then connect the state S to either T or R (if there is no existing edge between them with the same direction and label). If we created a new state, we add it to the queue Q. We repeat step 2 until there are no more states left in Q. This step can be seen in figure 6.

In the last step, we go through all of the states of N and look at its set of states in M. If there are states which are not accepting states in M, we remove them from the set of states. If the set of states contains at least one accepting state in M, then it is also an accepting state in N. The resulting DFA for bit 3 can be seen in figure 7. The code for the bit-splitting for 8 states can be found in the appendix A.4. A full implementation of the bit-splitting algorithm, for 8 and 4 states can be found here [5].

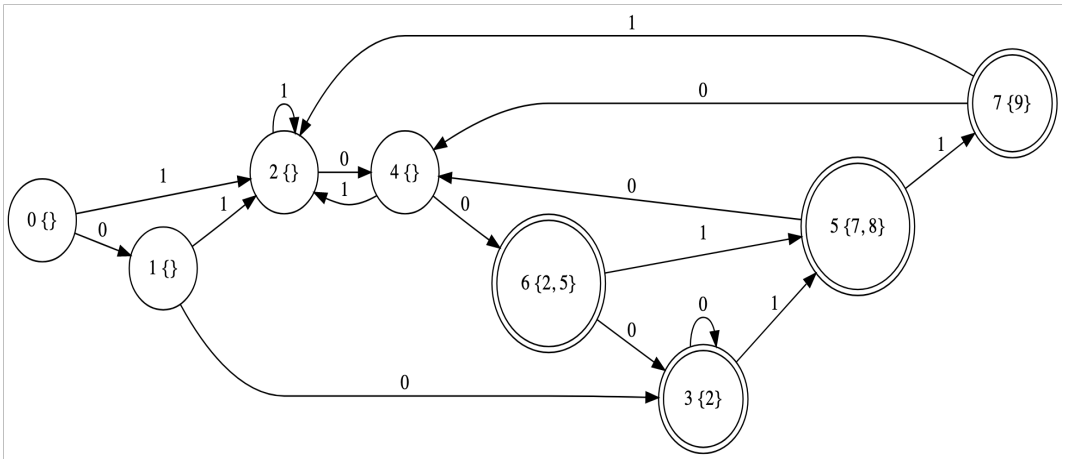


Figure 7: Result of Bit-Split Algorithm for Bit 3 of DFA in Figure 4

Similarly, we can apply this algorithm to create 4 state machines instead of 8. Each state is now responsible for 2 bits instead of 1. According to [13] and [4], this approach is optimal in terms of storage. It requires only 2^2 memory locations per possible input combination, with a total of $4 * 2^2 = 16$ locations over the 4 machines. An example can be seen in figure 8. It is the bit-split machine responsible for bits 01 of the DFA in figure 4.

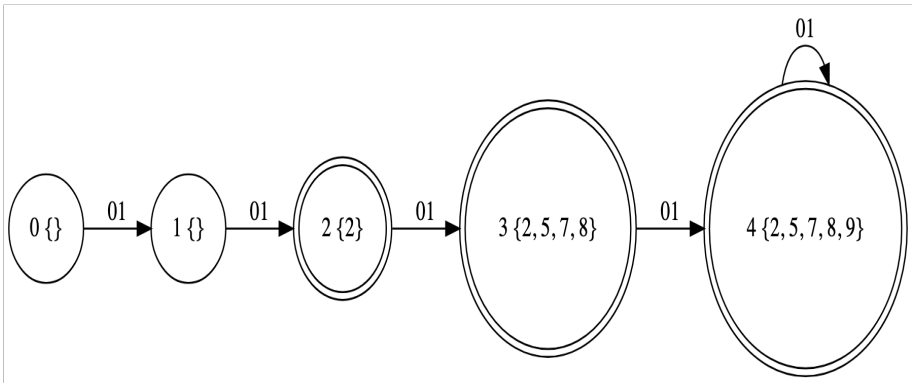


Figure 8: Result of Bit-Split Algorithm for Bits 01 (MSB = 0) of DFA in Figure 4

2.5.3 Table Filling

Once the sub-DFAs are constructed, we will load them into the rule modules of our FPGA. Each module will consist of 4 tiles, each of which will represent the state transitions for its respective sub-DFA. Each table has the state numbers, the transitions for an input and a partial match vector. The partial match vector has 1 bit per possible string to match. For instance, the partial match vector for our above example {he, she, his, hers} will consist of 4 bits. For instance, a PMV equal to 1001, will show a partial match for the strings "he" and "hers". We will illustrate the tile for the DFA of bits 01, shown in figure 8.

	00	01	10	11	PMV
0	0	1	0	0	0000
1	0	2	0	0	0000
2	0	3	0	0	1000
3	0	4	0	0	1110
4	0	4	0	0	1111

Table 3: Tile 0 for Transitions of DFA in Figure 8

For each input, the transitions are recorded with the help of a state pointer. After each transition, the PMV is outputed for each of the 4 tiles. Each output is then bit-wise ANDed and the result is a 4-bit full match vector. In our case, if the FMV is equal to 0100, then we have a full match for the string "she". The hardware implementation of this table can be seen in section 2.7.1.

2.5.4 Rule Partitioning

If we decide to construct a single state machine for all of the rules in our rule set, we will have a partial match vector with a number of bits equal to the size of the rule set, which we will have to store for each entry of each tile. This is quite a waste of storage.

A possible solution would be to divide the rule set into groups and construct DFAs for each of them. In order to choose the grouping algorithm, there are a few factors to take into consideration. The length N of the PMV determines how many strings we can handle per tile. Also, each tile can only store a fixed number S of states. We would like to make full use of the storage of both PMVs and state entries. We thus would like to maximize the number of strings, without going over N strings or S states, otherwise the group would need to be divided again in order to have a fitting number of states and this would again result in a waste of PMVs. A possible solution for static strings would be to sort them lexicographically and then divide them into groups, so that their common prefixes share states in state machines.

It is a different story for regular expressions, so it is debatable whether rule partitioning is useful when treating PROSITE patterns. It will be up to the implementing party to decide if and how to group PROSITE patterns.

2.5.5 Open Issues

Sometimes, patterns can overlap each other. For instance, if we have a regex "telephone—phonebook" and an input "telephonebook", we would like to have a match for both "telephone" and "phonebook" for the given input.

For static strings, this can be done by adding an edge from the 'e' at the end of telephone to the 'b' of phonebook, thus merging two DFAs into one. In [4], it is stated that this method cannot be applied to general regexp, because it cannot be done simply merging DFAs but would require more states that are not included in the original two DFAs. We have unfortunately not investigated further on this topic, so overlapped matches is not enforced in this solution.

2.6 Microcontroller Software

One of the main issues of transforming Regular Expressions into DFAs is the exponential state explosion which happens when an expression uses wildcards or constrained repetitions. Wildcards are usually in the form of ".$*$" for Regular Expressions and x{0,N} for PROSITE with N being any user-defined number. Constrained repetitions are in the form of a(x, y) or A{x,y}, for the respective types mentioned previously.

In order to eliminate the state explosion problems of those two types of expression components, a microcontroller-based architecture has been developed in [4]. This architecture links a DFA simulation module with a small and customized microcontroller, which allows a memory-efficient implementation of the rules of Regular Expressions without state explosion. Every regular expression is broken down into independent segments at wildcard separators and reassembled postmatch in the microcontroller. Each DFA is responsible for up to M pattern segments. For instance, in [4] each DFA is responsible for 28 pattern segments.

The microcontroller keeps track of the flags and counters required to implement the wildcards and reassembles the pattern segments. With this approach, orders of magnitude less of memory are used in comparison to implementing the full DFAs with states for wildcards and constrained repetitions.

The approach goes as follows. We first separate each rule into its components. This is done by parsing the rule and extracting the substrings between wildcards and wildcard constraints. Each pattern segment and matching wildcard function is then saved to an internal table. This table is further translated by a template-based assembler to the binary instruction codes and starting data memory configurations. After the instruction code is created, the code is "linked" to generate the tranlation tables.

2.7 System Architecture

2.7.1 Bit-Split Tile

Figure 9 shows us a hardware representation of a tile responsible for one out of four DFAs constructed from the original one using the Bit-Split algorithm. Each tile is responsible for a group of N rules. The loading tables are loaded using the configuration data and contain the state transitions as seen in section 2.5.3. Each row of the table represent a state in the DFA and each of the first four columns represent the possible next states to transition to. The last column on the left represents the value of the Partial Match Vector for a given state. The input consists of 2 out of the initial 8 bits of an ASCII character. A pointer keeps track of the current state and the Partial Match Vector of length N is outputted after each transition.

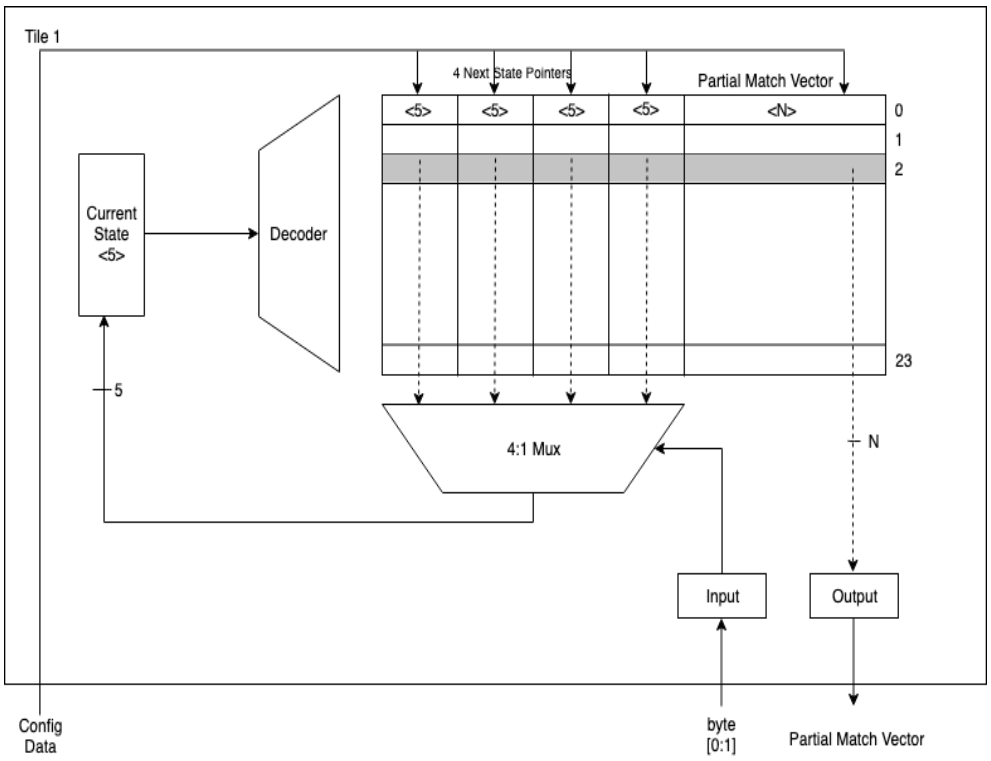


Figure 9: Bit-Split Tile 0

2.7.2 Priority Encoder

In [11], the need for a priority encoder was introduced. Since each module is responsible for a finite number of rules N, when we drastically increase the number of patterns in our database, we will scale up the number of modules and come into significant routing delays. To speed up our system, it is more important to minimize these routing delays than to optimimize the rule modules themselves. Since we can put many rule modules on a FPGA, their arrangement is critical to the overall performance. In this implementation, the critical path is the production of the Full Match Vector, produced by each module. Without a priority encoder, all FMVs will be bundled at the end of the string matching engine.

In order to reduce the total number of outgoing lines, we put a priority encoder, which chooses one module if it matches at least one pattern. If we have more than one match in one module, we can seperate it in software and it would probably be that multiple patterns have overlapped on one branch of the Aho-Corasick tree. We could have potential overlaps in multiple modules, so the patterns must be arranged so that the longest pattern is in the highest priority module. We can thus extract shorter patterns that are a substrings of the longer ones in software. The PE thus provides the encoded addresses of the matches in successive cycles. The priority encoder design can be seen in Figure 10.

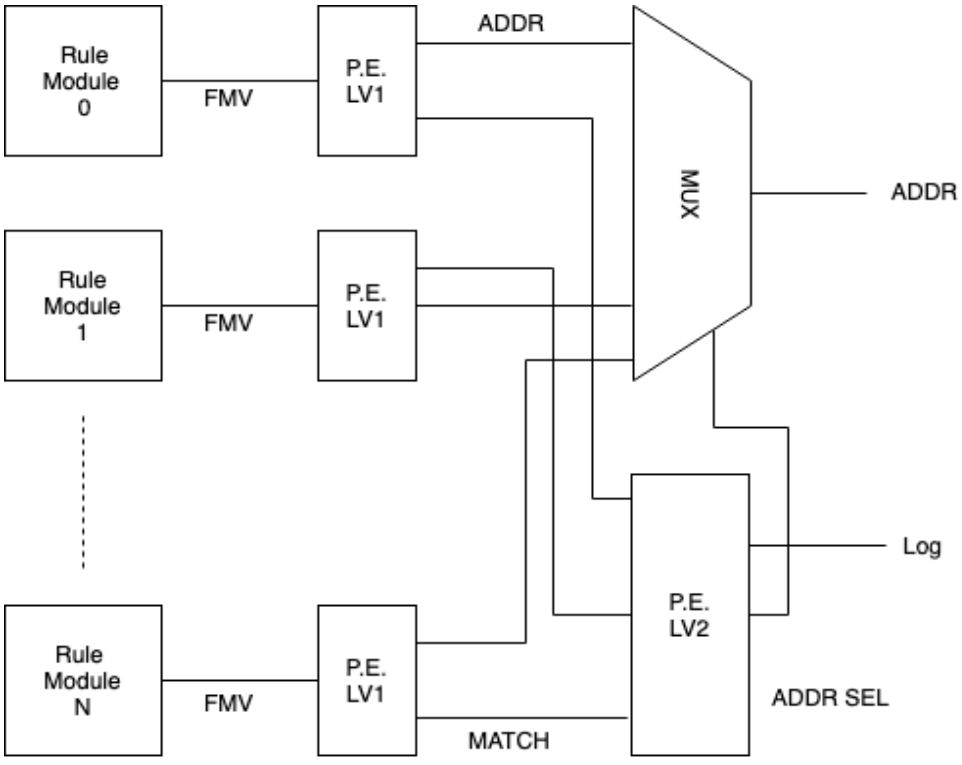


Figure 10: Priority Encoder blocks for reporting results outside of device

2.7.3 Pattern Module

Figure 11 shows us a global view of the system architecture of a pattern module, which we have discussed in this report. We can see the four Bit-Split tiles on the left, each simulating a DFA responsible for 2 out of 8 bits of an input character. Their respective Partial Match Vectors are then ANDed as described in section 2.5.3 and result in the Full Match Vector for the module. Then, the FMVs go through the priority encoder described in 2.7.2.

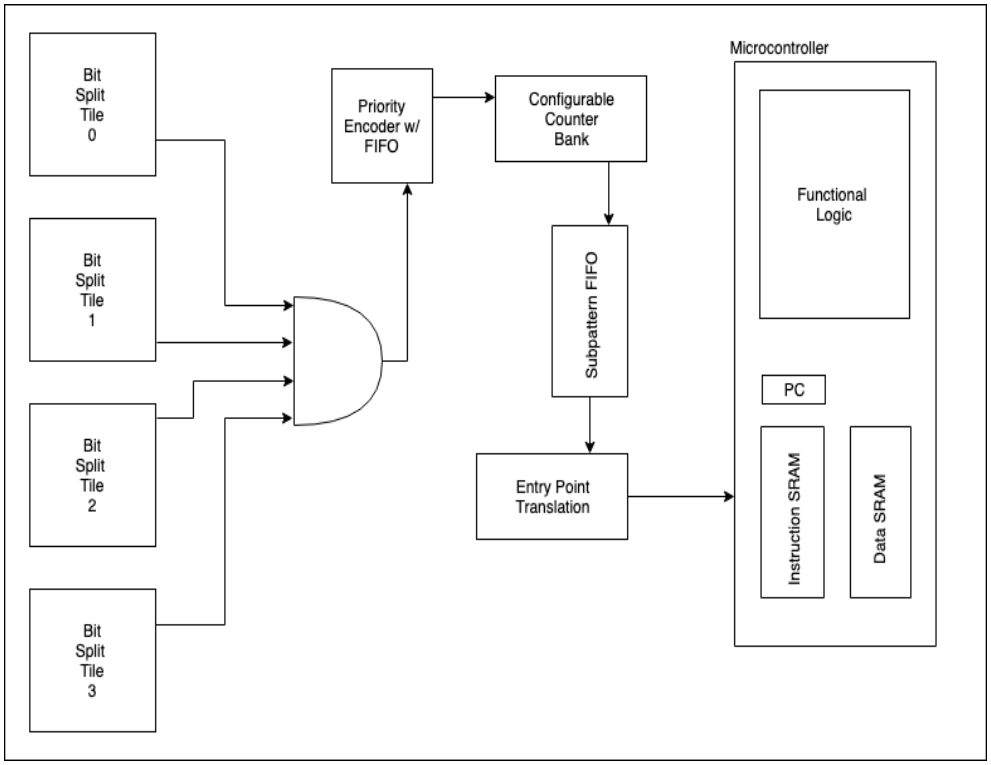


Figure 11: System Architecture of Pattern Microcontroller and Bit-Split State Machine

Since we have delays resulting from reporting match results and from the execution of the microcode, we also have a FIFO buffer which handles the delay between the detection of the match and its processing by the microcontroller. We must also preserve the state of the system when a match occurs. As the match comes from the DFA, this information is recorded in the FIFO by saving the relevant counter index for each segment. Each segment stores the counter's value at the time of the detection in memory. The microcontroller reads the current counter value and stores it in memory, following its program listing as seen in Figure 6 of [4].

3 Conclusion

The aim of this project was to find a reconfigurable approach to replace the hard-coded Regular Expressions of the implementation described in [6], in order to detect infectious pathogens and perform a PROSITE seach in real time.

We have thoroughly investigated the work done by Z. K. Baker, H.-J. Jung and V. K. Prasanna in [4] in order to apply their techniques used for Intrusion Detection Systems to our problem of matching PROSITE patterns. We have successfully understood and demonstrated how to transform and manipulate regular expressions in order to interpret them efficiently at runtime. We have also sampled over 1000 PROSITE patterns in order to investigate whether annotating parts of the patterns, as demonstrated in [4], would be useful in our implementation. Furthermore, we have understood how the Aho-Corasick algorithm is used to encode patterns, as well as its benefits and drawbacks. We have seen that the Bit-Splitting algorithm was introduced in order to fix these memory storage related problems and have implemented it fully, having not found an existing solution online.
We have studied the use of microcontrollers to further minimize state explosions caused from the conversion of regular expressions into DFAs. Additionally, we have understood the need of priority encoders in order to reduce routing delays created by the large number of pattern modules in our system. Finally, we have analysed and understood the reconfigurable hardware designs illustrated in the research.

Due to time constraints, the microcontroller and the hardware design have not been implemented in this project. Likewise, the issue of dealing with overlapped edges by merging DFAs is still left open. These topics are left to be implemented in future work.

4 Bibliography

[1] The PROSITE database of protein domains, families and functional sites - User Manual. https://prosite.expasy.org/prosuser.html. Accessed June 9, 2019.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18:333–340, 06 1975.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[4] Z. K. Baker, H. Jung, and V. K. Prasanna. Regular expression software deceleration for intrusion detection systems. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–8, Aug 2006.

[5] Z Baraschi. A java implementation of the bit-splitting algorithm. https://github.com/baraschi/bit-split. Accessed June 11, 2019.

[6] S. Casale Brunet, T. Schuepbach, N. Guex, C. Iseli, A. Bridge, D. Kuznetsov, C. Sigrist, P. Lemercier, I. Xenarios, and E. Bezati. Towards in the field fast pathogens detection using fpgas. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 463–4631, Aug 2018.

[7] Johan Eker and Jorn Janneck. CAL language report. Technical report, 2003.

[8] Python Software Foundation. python. https://www.python.org/.

[9] Pandas Governance. pandas. https://pandas.pydata.org/. Accessed June 10, 2019.

[10] J. Goyvaerts. Regular Expressions Reference. https://www.regular-expressions.info/reference.html. Accessed June 9, 2019.

[11] Hong-Jip Jung, Z. K. Baker, and V. K. Prasanna. Performance of fpga implementation of bit-split architecture for intrusion detection systems. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8 pp.–, April 2006.

[12] J.W. Janneck. Tokens? What tokens? A gentle introduction to dataflow programming. Technical report, Programming Solutions Group Xilinx, August 2007. ASTG Technical Memo — document edition 1.

[13] Lin Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 112–122, June 2005.

[14] Paul Stothard. Sequence Manipulation Suite: IUPAC codes. https://www.bioinformatics.org/sms2/iupac.html. Accessed June 9, 2019.

[15] L Richardson. beautifulsoup4 4.7.1. https://pypi.org/project/beautifulsoup4. Accessed June 10, 2019.

[16] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 227–238, March 2001.

[17] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *2006 Symposium on Architecture For Networking And Communications Systems*, pages 93–102, Dec 2006.

A Appendix

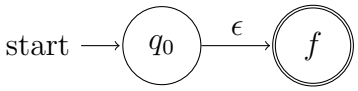
A.1 IUPAC Amino Acid Codes

IUPAC Amino Acid Codes [14]		
1-Letter Code	3-Letter Code	Amino Acid
A	Ala	Alanine
B	Asx	Aspartic acid or Asparagine
C	Cys	Cysteine
D	Asp	Aspartic Acid
E	Glu	Glutamic Acid
F	Phe	Phenylalanine
G	Gly	Glycine
H	His	Histidine
I	Ile	Isoleucine
K	Lys	Lysine
L	Leu	Leucine
M	Met	Methionine
N	Asn	Asparagine
P	Pro	Proline
Q	Gln	Glutamine
R	Arg	Arginine
S	Ser	Serine
T	Thr	Threonine
V	Val	Valine
W	Trp	Tryptophan
X	Xaa	Any amino acid
Y	Tyr	Tyrosine
Z	Glx	Glutamine or Glutamic acid

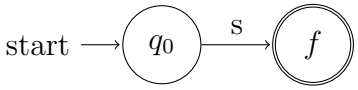
A.2 Thompson's Algorithm: Rules

Let subexpressions a and b have NFAs N(a) and N(b) respectively.

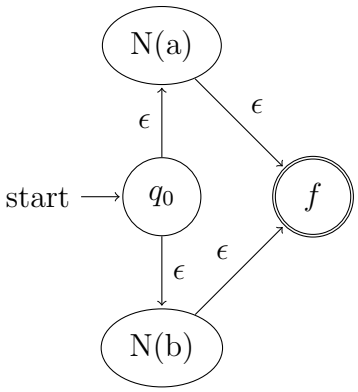
1. The empty expression ϵ is converted to the following NFA with start state q_0 and accepting state f.



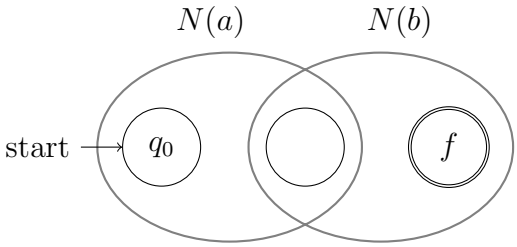
2. A symbol s of the input alphabet is converted to:



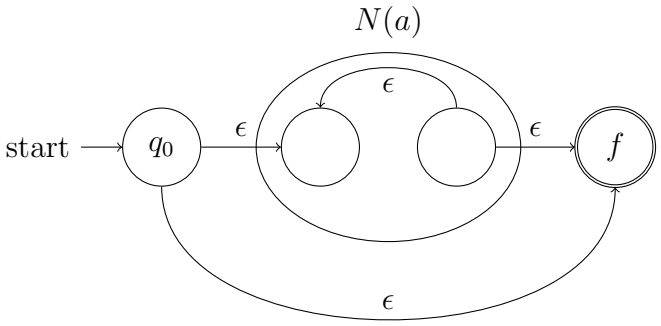
3. The union expression a|b is:



4. The concatenation expression ab is:



5. The Kleene star expression a* is:



6. The parenthesized expression (a) is converted to N(a).

The proof of this algorithm can be found in Chapter 3.7.4 [3].

A.3 PROSITE Pattern Sampling

In order to sample PROSITE patterns, we first had to find elements in the PROSITE documentation [1] which concretely defined the pattern line. We did a search for the keyword "Pattern" and had an output of 1101 pages containing proteins with their associated patterns. We scraped this search to get all the urls of these associated documentation pages. We then opened each page and extracted the patterns in them (some pages defined more than one pattern).

This was done using python [8] and Beautiful Soup [15]. We then made some manipulations using pandas [9]. The code can be found below.

```
# coding: utf-8

from bs4 import BeautifulSoup
import requests
import re
import pandas as pd

main = "https://prosite.expasy.org"
url = main + "/cgi-bin/prosite/prosite_search_full.pl?SEARCH=
    Pattern"
r  = requests.get(url)
data = r.text
soup = BeautifulSoup(data, "html.parser")

#find all patterns
patterns = []
for link in soup.find_all('a'):
    ref = link.get('href')
    if ref and "/PDOC" in ref:
        patterns.append(ref)

#loop through all pattern urls and find PA line
pa = []
for p in patterns:
```

```
    r = requests.get(main + p)
    data = r.text
    soup = BeautifulSoup(data, "html.parser")
    for x in soup.find_all('fieldset'):
        for y in x.find_all('ul'):
            z = y.find('li')
            if z and "Consensus" in z.contents[0]:
                pa.append(z.contents[2].replace('\n', ''))

#save to dataframe
df = pd.DataFrame(pa, columns=["pattern"])

#save to csv
df.to_csv("patterns.csv")

#load from csv
df = pd.read_csv('patterns.csv', index_col=0)

#find patterns containing repetitions - look for repetition symbol
pa_rep = df[df['pattern'].str.contains("(", na=False, regex=False)
    ]
print("Patterns with repetition:", (len(pa_rep)/len(df))*100, "%")

vals = pa_rep['pattern'].values
a = []
c=[]
for v in vals:
    #filter all repetitions within each pattern
    r = re.findall("([\[\]a-zA-Z]{0,27}\(.{0,7}\))-", v)
    c.append([v,r])
    for i in r:
        a.append([v,i])

#save to dataframe - one line per pattern/repetition segment
df_rep = pd.DataFrame(a, columns=['pattern', 'repetition'])

```

```
#save to dataframe - one line per pattern/ list of repetition segments
df_rep_comp = pd.DataFrame(c, columns=['pattern', 'repetition'])

vals = pa_rep['pattern'].values
b = []
d=[]
for v in vals:
    #filter all zero range repetitions within each pattern
    r = re.findall("([\[\]a-zA-Z]{0,27}\(0.{0,3}\))-", v)
    if r:
        d.append([v,r])
        for i in r:
            b.append([v,i])

#save to dataframe - one line per pattern/repetition segment
df_zero_rep = pd.DataFrame(b, columns=['pattern', 'repetition'])

#save to dataframe - one line per pattern/ list of repetition segments
df_zero_comp = pd.DataFrame(d, columns=['pattern', 'repetition'])

print("Patterns with zero range repetitions:", (len(df_zero_rep)/len(df))*100, "%")

#patterns with more than one zero range repetition
df_z_reps = df_zero_comp[df_zero_comp['repetition'].str.len() > 1]

print("Patterns with multiple zero range repetitions:",
      (len(df_z_reps)/len(df))*100, "%")
```

A.4 Bit-Splitting Implementation

```
1  /*
2      * Method which takes original graph g and applies bit split algorithm
       to return a new graph for the bit given as
3      * a parameter.
4      * As per: https://ieeexplore.ieee.org/abstract/document/1431550
5      * Note that you can choose whether 0 or 7 is the LSB by setting the
       boolean zeroLSB
6      */
7      private static Graph<ExtendedState, RelationshipEdge> getBitSplitGraph(
       Graph<State, RelationshipEdge> g,
8
       Map<String, List<Integer>> binEncoding,
9
       int bitNumber, boolean zeroLSB) {
10         //create new graph
11         Graph<ExtendedState, RelationshipEdge> bitSplit = new
       DirectedPseudograph<>(RelationshipEdge.class);
12         //iterate over states in g graph
13         GraphIterator<State, RelationshipEdge> iterator = new
       DepthFirstIterator<>(g);
14         //counter for state number
15         int stateNumber = 0;
16         //first node of g graph
17         State source = iterator.next();
18         //create new extended state corresponding to first node of bit
       split graph
19         ExtendedState state = new ExtendedState(String.valueOf(stateNumber)
       , false, "0");
20         //add first node of g to set of nodes of first node of bit split
       graph
21         state.addState(source);
22         //add state to bit split
23         bitSplit.addVertex(state);
24         //increase state number
25         stateNumber++;
26
27         //set of nodes of bit split graph to visit
28         Queue<ExtendedState> toVisit = new LinkedList<>();
29         //add initial state to queue
30         toVisit.add(state);
31
32         //loop through until there are no more states to visit
```

```
33          while (!toVisit.isEmpty()) {
34              //set of nodes with transition state 0
35              Set<State> zeros = new HashSet<>();
36              //set of nodes with transition state 1
37              Set<State> ones = new HashSet<>();
38
39              //get and remove next element from toVisit queue
40              ExtendedState currentState = toVisit.poll();
41
42              assert currentState != null;
43
44              //loop through set of states of state
45              for (State s : currentState.getSet()) {
46                  //get outgoing edged of current state
47                  Set<RelationshipEdge> outgoing = g.outgoingEdgesOf(s);
48                  //loop through outgoing edges of state
49                  for (RelationshipEdge e : outgoing) {
50                      //get edge label
51                      String label = e.getLabel();
52                      //find corresponding integer value (0 or 1) for bit
    given on input
53                      Integer labelValue;
54                      if(zeroLSB) {
55                          labelValue = Math.abs(binEncoding.get(label).get(
    bitNumber)-7);
56                      }else{
57                          labelValue = binEncoding.get(label).get(bitNumber);
58                      }
59                      if (labelValue == 0) {
60                          //add reachable state to 0 set
61                          zeros.add(e.getTarget());
62                      } else {
63                          //add reachable state to 1 set
64                          ones.add(e.getTarget());
65                      }
66                  }
67              }
68
69              if (!zeros.isEmpty()) {
70                  //check if there is an existing state for zeros state set
71                  zeros.add(source);
72                  ExtendedState zero = stateInGraph(bitSplit, zeros, "0");
73                  //if not, create and add new extended state
74                  if (Objects.isNull(zero)) {
```

```
                    zero = new ExtendedState(String.valueOf(stateNumber),
    false, "0");
                    zero.addSet(zeros);
                    stateNumber++;
                    bitSplit.addVertex(zero);
                    toVisit.add(zero);
                }
                //add edge between current state and extended state
                if (noExistingEdge(bitSplit, currentState, "0")) {
                    bitSplit.addEdge(currentState, zero, new
    RelationshipEdge(String.valueOf(0)));
                }
            }

            if (!ones.isEmpty()) {
                //check if there is an existing states for ones state set
                ones.add(source);
                ExtendedState one = stateInGraph(bitSplit, ones, "1");
                //if not, create and add new extended state
                if (Objects.isNull(one)) {
                    one = new ExtendedState(String.valueOf(stateNumber),
    false, "1");
                    one.addSet(ones);
                    stateNumber++;
                    bitSplit.addVertex(one);
                    toVisit.add(one);
                }
                //add edge between current state and extended state
                if (noExistingEdge(bitSplit, currentState, "1")) {
                    bitSplit.addEdge(currentState, one, new
    RelationshipEdge(String.valueOf(1)));
                }
            }
        }
        for (ExtendedState e : bitSplit.vertexSet()) {
            e.setAccepting();
        }

        return bitSplit;

    }
```