

An FPGA Based Approach
for Černý Conjecture Falsification

by

ÇAĞLA KOCA

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabanci University

July 2018

An FPGA Based Approach
for Černý Conjecture Falsification

APPROVED BY:

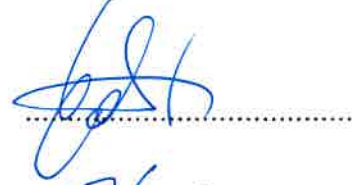
Prof. ErKay Savaş
(Thesis Supervisor)



Assoc. Prof. Hüsnü Yenigün
(Thesis Co-supervisor)



Asst. Prof. Erdiñç Öztürk



Asst. Prof. Kamer Kaya



Asst. Prof. Alper Özpınar



DATE OF APPROVAL: 30/07/18

© aęla Koca 2018
All Rights Reserved

ABSTRACT

An FPGA Based Approach for Černý Conjecture Falsification

ÇAĞLA KOCA

M.Sc. Thesis, July 2018

Supervisor: Prof. ErKay Savaş

Co-supervisor: Assoc. Prof. Hüsnü Yenigün

Keywords: FPGA, Synchronizing sequences, Černý conjecture

A synchronizing sequence for an automaton is a special input sequence that sends all states of the automaton to the same state. J. Černý conjectured that the length of the shortest synchronizing sequence of an automaton with n states cannot be greater than $(n - 1)^2$, which is known today as the Černý conjecture. This half-a-century old conjecture is still open and it is considered to be the most long-standing open problem in the combinatorial theory of finite state automata. One research line that has been pursued in the literature is to check if the conjecture holds for a fixed number of states n , by considering all automata with n states and checking if any of these automata falsifies the conjecture. This is a computationally intensive task, even for automata up to a dozen of states and only two input symbols. To accelerate the search parallel computation approaches using multicore CPUs have been tried before. In this thesis, we study the use of FPGAs to accelerate the search for an automaton falsifying the Černý conjecture. We present a design to calculate

the minimum length synchronizing sequence of a finite state automaton. The proposed design is implemented with the parallel computing capability of hardware designs while optimizing the time performance.

ÖZET

Černý Sanıtı Yanlıřlaması için APKD Tabanlı Yaklařım

Çaęla Koca

Master Tezi, Temmuz 2018

Danıřman: Prof. Dr. ErKay Savař

Eř-danıřman: Doç. Dr. Hüsnü Yenigün

Anahtar Sözcükler: APKD, Sıfırlama dizileri, Černý sanıtı

Bir sıfırlama dizisi, verilen bir özdevininim tüm durumlarını aynı duruma getirmeye yarayan bir girdi dizisidir. J. Černý, n durumlu bir özdevininim için en kısa sıfırlama dizisi boyunun $(n - 1)^2$ 'den daha uzun olamayacağı varsayımında bulunmuřtur. Bu varsayım günümüzde Černý sanıtı olarak adlandırılmaktadır. Bu yarım asırlık eski varsayım hala açıktır ve sonlu durum özdevininim kombinatoryal teorisinde en uzun süre çözülemeyen problem olarak kabul edilmektedir. Literatürde yürütölen çalıřmalardan bir tanesi, n durum sayısına sahip tüm özdevininimleri ele alarak ve bu özdevininimlerin hepsinde sanıtın doęru olup olmadığı kontrol ederek, belli bir n durum sayısına sahip özdevimlerin için Černý sanıtının doęruluęunu kontrol etmektir. Bu, sadece 2 girdili ve durum sayısı 12 veya daha az olan tüm özdevininimler için bile hesaplama açıısından yoęun bir iřlemdir. Arama iřlemlerini hızlandırmak için, çok çekirdekli CPU'lar kullanan paralel hesaplama yöntemleri daha önce denenmiřtir. Bu tezde, Černý sanıtını yanlıřlayan bir özdevininim

arayışını hızlandırmak için Alanda Programlanabilir Kapı Dizileri (APKD) kullanımı üzerine çalışılmaktadır. Sonlu bir durum özdevinimin en kısa sıfırlama dizisi boyunu hesaplayan bir tasarım sunulmaktadır. Önerilen tasarım, donanım tasarımlarının paralel hesaplama imkanlarıyla zaman performansını optimize ederek uygulanmaktadır.

dedicated to everyone who've been patient with me,
especially my family

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisor Prof. Erkey Savaş for his encouragement and worthwhile guidance during my masters studies at Sabanci University. It is my honor to be able to finish my masters studies under his supervision.

I would also like extend my sincere thanks to my co-supervisor Assoc. Prof. Hüsnü Yenigün for his contributions, support and patience along the way. His passion and broad vision for research has made a deep impression on me. It has been a great honor for me to work under his guidance.

I also would like to sincerely thank my committee members, Asst. Prof. Erdinç Öztürk, Asst. Prof. Kamer Kaya and Asst. Prof. Alper Özpınar for their time, interest and valuable comments. I would like to express my special thanks to Asst. Prof. Erdinç Öztürk for taking his time to answer my endless questions throughout my studies.

I would like to thank my friends and my lab colleagues for all the help and support during the development of the thesis work.

Most of all, I am grateful to my family for their absolute support and unconditional love. Without them and their devotion this work would never have come into existence.

Finally, I would like to acknowledge Sabanci University and Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting me with scholarships throughout my studies. This thesis was supported by TÜBİTAK under the contract 114E569.

TABLE OF CONTENTS

1	Introduction	1
1.1	Literature Survey	3
1.2	Organization	6
2	Background and Notation	7
2.1	Finite State Machines	7
2.2	FPGA	9
3	FPGA Implementation	13
3.1	Introduction	13
3.2	Proposed Structure	14
4	Experiments	29
4.1	Verification	29
4.2	Timing Results	31
4.2.1	Applied Filters	36
5	Conclusion	39

LIST OF TABLES

2.1	General feature summary	11
4.1	Timing for n= 9	32
4.2	Utilization for n=9	33
4.3	Slice Logic for n= 9	34
4.4	Memory for n=9	34
4.5	Timing for n=12	34
4.6	Utilization for n=12	35
4.7	Slice logic for n=12	35
4.8	Memory for n=12	35
4.9	Timing Comparison	35
4.10	Filter Timing Comparison	38

LIST OF FIGURES

2.1	A deterministic and fully-specified automaton with $n=4$ states	8
2.2	Černý automaton with $n=4$ states	10
2.3	Xilinx Virtex-7 FPGA VC709 Board Features	11
3.1	Block Diagram of Overall Structure	14
3.2	Block Diagram of Submodules inside <code>uart_top</code>	15
3.3	State Machine for Transferring Two Unary Automata A and B from <code>RX_serial</code> to FIFOs in <code>getMacUart</code>	18
3.4	Block Diagram of Submodules inside <code>topmodule</code>	20
3.5	General Block Diagram of Module Instantiation	21
3.6	State Diagram of State Machine in <code>cerny_pu</code>	25
4.1	Comparison of FPGA and Processor Timings for All $n = 9$ State Automata	33

Chapter 1

Introduction

The continual demand to improve performance and efficiency of software and digital hardware designs leads to a corresponding increase in system complexity. The improvements in complex software and hardware designs need testing and verification steps which are highly costly in terms of time and money. Therefore, the development of new methods and techniques for testing the functional requirements of such systems is important to optimize cost and productivity of the design process.

Once the functional requirements of the developing system are formally defined, it is possible to adapt efficient approaches to test these features quickly. Due to the discrete and sequential behavior of digital systems, formal descriptions of the functional requirements of these systems can be made using state based specification methods and finite state machines (FSM).

When a functional design structure is modeled by finite state machines, it is possible to produce high-quality test sequences for verification by analyzing the FSM [11, 16, 22]. These high-quality test sequences are generated by using a combination of some special structures, such as a distinguishing sequence, a locating sequence, and a synchronizing sequence [24]. The transitions in an FSM are labeled with an input and an output symbol. The output symbols make no difference in the context of synchronizing sequences examined in this thesis, therefore an FSM is considered as an automaton where the transitions are only labeled with an input symbol.

A synchronizing sequence (or a reset word) is a special input sequence that resets an automaton to a single state. In other words, an automaton reaches a specific state by applying a synchronizing sequence (SS) regardless of its initial state. Any state of automaton which is applied a SS results in the same single state. In practice, SS is used to reset any system whose behavior can be modeled as a finite automaton. The applications in the field of electronic circuitry and software testing take advantages from the synchronizing sequences greatly. Apart from these applications, it is also reported in the literature that SS is used for the alignment problem in production engineering [14, 28] and for the resetting of the calculation units in the bio-computing area [4].

The number of inputs that formed SS is preferred to be as short as possible in order to keep resetting time and implementation cost minimized. However for a given FSM, computing the shortest reset word is known to be NP-hard problem in general [15]. That is why, heuristic methods are proposed to compute shortest SS in the literature. On the other hand, all these methods have the complexity of $\Omega(n^2)$ where n is the number of states in automaton. Although it is a low degree polynomial, this complexity becomes problematic in terms of practical applications with large scale automata.

Another concern of synchronizing sequences is about the upper bound on the length of shortest reset words. Černý [9] claims that the length of the shortest reset word of an automaton with n states cannot be greater than $(n - 1)^2$. This is called the Černý Conjecture. After half a century, hundreds of papers, and two conferences dedicated to the investigation of the conjecture, no one was able to prove or disprove this claim. There are studies that validate the conjecture by theoretically proving it for some specific automaton classes, but it is not verified in general. The approach in these experimental studies is to check all the automata with a certain state and number of inputs one by one whether the conjecture is valid for all these automata.

Main concern is improving the performances of short SS generation methods in the literature and developing new heuristic methods for this purpose. However, high performance calculations are required for both of these main topics. Improvements on the heuristic algorithms lead to increase in complexity. Modern parallel computing methods

are used on both Graphic Processor Units (GPU) and Field Programmable Gate Array (FPGA) technologies to process the complex structures.

At the beginning of the thesis, it was known that Černý conjecture was true for all the automata with an alphabet size of 2 and a number of states of 11 or less. It is shown experimentally that Černý conjecture holds for all automata up to 12 states and with 2 inputs using CPUs [19]. As discussed, performance limitations occur due to cost, size and complexity of calculations that are used to check the correctness of Černý conjecture for 12 state automata. Thus, parallel computing techniques are planned to be used in order to speed up the checking process. Considering the high system performance, parallel processing capacity and their easy integration capabilities, FPGAs are optimal solutions for this purpose. Accordingly, the purpose of this thesis is to develop a high-performance analyzing system for computations using FPGA technologies in order to investigate whether Černý conjecture holds for all automata up to 12 states and with 2 inputs. A shortest synchronizing sequence searching design for Černý conjecture verification is implemented with the parallel computing capability of hardware designs while optimizing the time performance. The proposed design considers breadth first search (BFS) algorithm for automaton pairs to be examined in order to find lengths of synchronizing sequences.

1.1 Literature Survey

Synchronizing sequences are used in many practical applications. For instance, the alignment problem encountered in assembly lines can be solved using synchronizing sequences. In an assembly line, robots pick objects from a point and carry it to another point where it is processed. Robots can pick up objects with a certain angle and orientation. However, the objects are dropped in a random fashion and the initial orientations are not known. The object is needed to change its current orientation to a particular orientation in order to be processed by robots. This problem can be modeled with an automaton where each possible orientation of the object is represented by a state. Also, orienting operations are considered as input alphabet. A sequence of orienting operations brings

these objects to certain orientation regardless of the initial orientation. A synchronizing sequence is used to ensure that the object arrives at a single certain orientation when it passes over all obstacles no matter which orientation it started at [14, 28]. A general literature review of synchronizing sequences and other application areas of them are presented by Volkov [44].

Main questions arose from the study of automata synchronization are synchronizability of a given automaton and the length of the shortest synchronizing sequence. The problem of checking the synchronizability of partial finite automata or non-deterministic automata requires algorithms with exponential complexity [27, 34]. Meanwhile, it is possible to check the existence of a synchronizing sequence of a deterministic and fully-specified automata with n -state and p -input symbol in $O(pn^2)$ complexity [15]. The length of a synchronizing sequence is desired to be kept minimum in practical use. Since each input in the sequence has a cost, keeping the sequence short makes it cheap and efficient in applications such as assembly lines or generating test sequences based on synchronizing sequences. However, it is proved that this problem is NP-hard [15].

It is also an interesting research aspect to find upper bounds for synchronizing sequences. Černý [9] states that every synchronizing automaton with n states has a synchronizing sequence of length at most $(n - 1)^2$. There are many attempts to prove so called Černý conjecture, but verification or falsification cannot be provided so far. The surveys concerning Černý conjecture still continue to appear in the literature [3, 5, 6, 10, 21, 23, 29, 30, 33, 37]. The conjecture is verified only for some special cases of automata such as non-strongly connected automata [45], acyclic automata [40], etc.

Falsifying examples are examined in some studies for automata with certain number of states and certain number of input symbols [1, 21, 39, 40]. There are total n^pn automata with n state and p input symbol. Each one of them is checked for the length of shortest synchronizing sequence. The point that needs to be considered here is the computational complexity of the numbers of the automata to be studied. Only non-isomorphic automata need to be considered to reduce that number because the validity of Černý conjecture is identical for two isomorphic automata. Therefore, all the approaches proposed in the

literature attempt to reduce the number of automata to be analyzed using this observation.

There are studies in the literature to calculate the number of non-isomorphic directed graphs [17, 26, 32]. Since automata are essentially directed graphs, all the results derived from directed graphs are also valid for automata. Moreover, Kisiliewicz and Szykula [20, 21] analyzed all automata with 4 inputs up to 7 states; with 3 inputs up to 8 states; with 2 inputs up to 11 states. Despite all the reduction in the number, there are still 79,246,008,127,339 automata with 11 states and 2 inputs. It took about 4 CPU years to process (25 days on 64 cores) [21]. The conjecture is verified for these specific automaton classes. In other words, an automaton falsifying the conjecture is not encountered after all these studies.

Within the context of this thesis, the validation of the conjecture is examined for 12-state and 2-input automata. Since there are 12^{24} automata, generation of isomorphic automata are tried to be avoided as in the previous works in the literature.

In this thesis, all binary automata (i.e., with two input symbols) are generated by superimposing two unary automata (i.e., with one input symbol). From a given unary (a -only) automaton A and a given unary (b -only) automaton B , it is possible to create $n!$ automata, where n is the number of states. Following that, permutations of a given unary automaton B are used to generate $n!$ binary automata with a given unary automaton A . Permutations are used to solve problems in several practical areas. Tompkins examined the use of permutations in [38]. Generally, two elements of an existing permutation are exchanged in order to generate a new permutation. A permutation enumeration algorithm was formulated by S. M. Johnson [18] and H. F. Trotter [41] independently. They show that it is possible to generate all $n!$ permutations of n elements with $(n! - 1)$ exchanges of adjacent elements. Furthermore, it can be seen that permutation networks of n elements can be constructed from permutation networks of $(n - 1)$ elements. Comprehensive permutation generation methods are presented in [36]. One of them proposed by Johnson [18] is to exchange two elements that are adjacent to each other. Another algorithm that optimized Johnson's method generates permutations in minimal change order [42]. In this thesis, for permutation generation we employed the algorithm based on Johnson's

method [18].

A synchronizing sequence of minimal length for binary automata is found by using breadth first search (BFS) on the set of all input sequences. BFS algorithm is the most widely used traversal graph algorithm. The graph starts at the whole state set of the given automaton and it is formed applying input symbols until a singleton is reached. The shortest synchronizing sequence corresponds to shortest paths from the whole state set to a singleton. Since larger graph problems need more computational and memory requirements, FPGA-based high performance solutions are becoming popular to accelerate such graph algorithms. There are several existing works that propose architectures to implement BFS on FPGA [12, 25, 43]. Early works that propose architectures for graphs on the clusters of FPGAs [2, 13] have limited performance gains for bigger graphs. They used low-latency on-chip memory resources. Optimizations on architectures are implemented in later works [8, 35, 46]. Some studies proposed using off-chip dynamic random-access memory for efficient traversal of large-scale graphs [7, 46]. Betkaui’s work [7] implements an efficient reconfigurable computing solution by taking advantage of both FPGA and the parallel memory subsystem. We examine automata with 12 states and 2 input symbols and there can be $2^{12} = 4096$ nodes at most in the graph. Since our graphs are relatively small, we use only memory resources on the FPGA for implementing BFS to find shortest synchronizing sequences of automata.

1.2 Organization

The rest of this thesis is organized as follows. Chapter 2 gives background information and notation necessary for understanding the basis of the work. Chapter 3 presents the overall architecture in detail. The proposed architecture was implemented on a Xilinx Virtex-7 FPGA device for evaluation. Details of implementation verification tests and experimental results are provided in Chapter 4. Certain design choices are compared with respect to their timing performances. Utilizations of the FPGA resources are also given in this chapter. Chapter 5 concludes the thesis with the summary of work.

Chapter 2

Background and Notation

2.1 Finite State Machines

An automaton is defined by a triple $A = (Q, \Sigma, \delta)$ where, Q is a finite set of states, Σ is a finite set of input alphabet, and δ is a transition function defining the action of the letters in Σ on Q . Automata can be classified into two types, deterministic and non-deterministic automaton. The transition function is defined $\delta : Q \times \Sigma \rightarrow Q$ for deterministic automaton whereas $\delta : Q \times \Sigma \rightarrow 2^Q$ for non-deterministic automaton. Moreover, automata can be divided into two types, fully-specified and partially-specified. A is called a fully specified automaton if δ is a total function that is defined at every state $q \in Q$ and every input $x \in \Sigma$. Otherwise, A is called a partially specified automaton when δ is a partial function that may not be defined at some state $q \in Q$ and input $x \in \Sigma$. In this thesis, only deterministic and fully-specified automata are studied. Throughout this thesis, the term automaton refers to deterministic and fully-specified automaton accordingly.

Graphical representation of an automaton would be as follows: a circle represents a state of the automaton and the transition function is represented by directed and labeled edges between states. Figure 2.1 shows an example of an automaton with 4 states (q_1, q_2, q_3 , and q_4) and 2 input symbols (a and b).

The number of states ($|Q|$) is denoted by n and the number of inputs ($|\Sigma|$) is denoted by p for the rest of the thesis. At any given time, an automaton $A = (Q, \Sigma, \delta)$ is at one

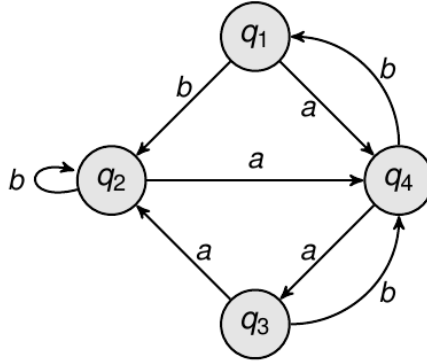


Figure 2.1: A deterministic and fully-specified automaton with $n=4$ states

of its states $q \in Q$. If an input $x \in \Sigma$ is applied when A is at state q , it changes its state to $\delta(q, x)$. Each element in Σ^* is an input sequence and the symbol ε is used to denote an empty input sequence. It is possible to extend the definitions of transition functions of automata on a state set and an input sequence as follows:

$$\forall q \in Q : \delta(q, \varepsilon) = q$$

$$\forall q \in Q, x \in \Sigma, \sigma \in \Sigma^* : \delta(q, x\sigma) = \delta(\delta(q, x), \sigma)$$

$$\forall Q' \subseteq Q, \sigma \in \Sigma^* : \delta(Q', \sigma) = \{\delta(q, \sigma) \mid q \in Q'\}$$

$Q^{(k)}$ describes all subsets of state set Q with a cardinality between 1 and k , where $1 \leq k \leq n$. It is formally defined as:

$$Q^{(k)} = \{Q' \subseteq Q \mid 1 \leq |Q'| \leq k\}$$

For a given automaton $A = (Q, \Sigma, \delta)$ and an integer k , where $1 \leq k \leq n$, the automaton $A^{(k)} = (Q^{(k)}, \Sigma, \delta^{(k)})$ is defined as follows:

$$\forall Q' \subseteq Q^{(k)}, x \in \Sigma, \delta^{(k)}(Q', x) = \delta(Q', x)$$

If there exists an input sequence $\sigma \in \Sigma^*$ that leaves the automaton $A = (Q, \Sigma, \delta)$

in one particular state such that, $|\delta(Q, \sigma)| = 1$, then A is called *synchronizing*. Any such input sequence σ is called a *synchronizing sequence* (or a *reset word*) for A . It is known that all automata are not synchronizing. There can be more than one synchronizing sequence for a synchronizing automaton.

Synchronizability of an automaton can be checked with an algorithm of $O(pn^2)$ complexity [15]. However, determining the minimum length of a synchronizing sequence for a given synchronizing automaton is an NP-hard problem [15]. It is possible to find the shortest synchronizing sequence by using algorithms with exponential complexity. Similarly, the power automaton $A^{(n)} = (Q^{(n)}, \Sigma, \delta^{(n)})$ can be used and the breadth first search is performed to find the shortest path to any singleton state. $Q \in Q^{(n)}$ is taken as a starting point which is the largest state set of automaton $A^{(n)}$ for this search. Since there are $2^n - 1$ states in $A^{(n)}$, this algorithm requires exponential time.

When Černý claims that upper bound for the shortest synchronizing sequence is $(n - 1)^2$ for an n state automaton, he also introduces an automaton class with two input symbols that achieves this bound. Such automata are called Černý automata and are defined as follows:

$$\forall q \in \{1, 2, \dots, n - 1\} : \delta(q, a) = q$$

$$\delta(0, a) = 1$$

$$\forall q \in \{0, 1, 2, \dots, n - 1\} : \delta(q, b) = (q + 1) \bmod n$$

Figure 2.2 shows an example of a Černý automaton with $n = 4$ states. The length of shortest synchronizing sequence for this automaton is $(n - 1)^2$. It can be found that ab^3ab^3a is the shortest synchronizing sequence for this automaton and it is the upper bound, $(n - 1)^2$, in Černý conjecture.

2.2 FPGA

Field Programmable Gate Array (FPGA) is set of logic blocks which are programmable with reconfigurable interconnects and input/output pads. The logic blocks can perform

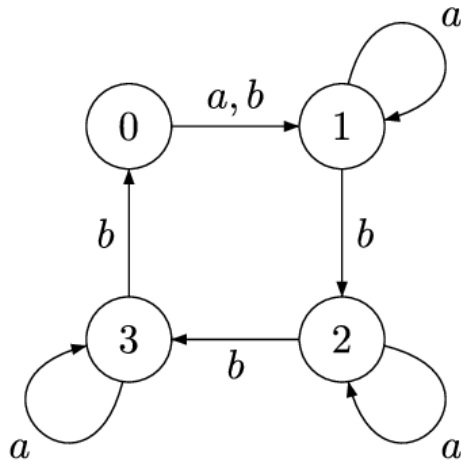


Figure 2.2: Černý automaton with $n=4$ states

simple to complex computational works to satisfy different system requirements.

The smallest configurable unit is called Configurable Logic Block (CLB) in Xilinx FPGAs. In a family Xilinx FPGAs, a CLB consists of two instances of a smaller logic unit called *slice*. Each FPGA slice contains four Look Up Tables (LUTs) and eight flip-flops (FF). The other configurable resources are DSP Slices, block RAMs (BRAM), I/O blocks and so on.

The FPGA component used in this thesis is a VC709 evaluation board, which provides a hardware environment for developing and evaluating designs targeting the Virtex-7 XC7VX690T-2FFG1761C FPGA. The Virtex-7 FPGA VC709 provides 40 Gb/s connectivity platform for high-bandwidth and high-performance applications. Figure 2.3 provides an overview of the board components [49]. A general summary of the features of FPGA VC709 is given in Table 2.1 [49]. Additionally, logic cells are the logical equivalent of a four-input LUT and a FF. The ratio between the number of logic cells and 6-input LUTs is 1.6:1. The details for each feature are described in VC709 Evaluation Board for the Virtex-7 FPGA User Guide [48].

The bitstream is a configuration file, which is used to program the logic blocks in an FPGA device. The JTAG connectivity on the VC709 board allows a host computer to download bitstreams to the FPGA using the Xilinx development tools. The proposed architecture is programmed into the FPGA by way of JTAG. JTAG configuration is provided

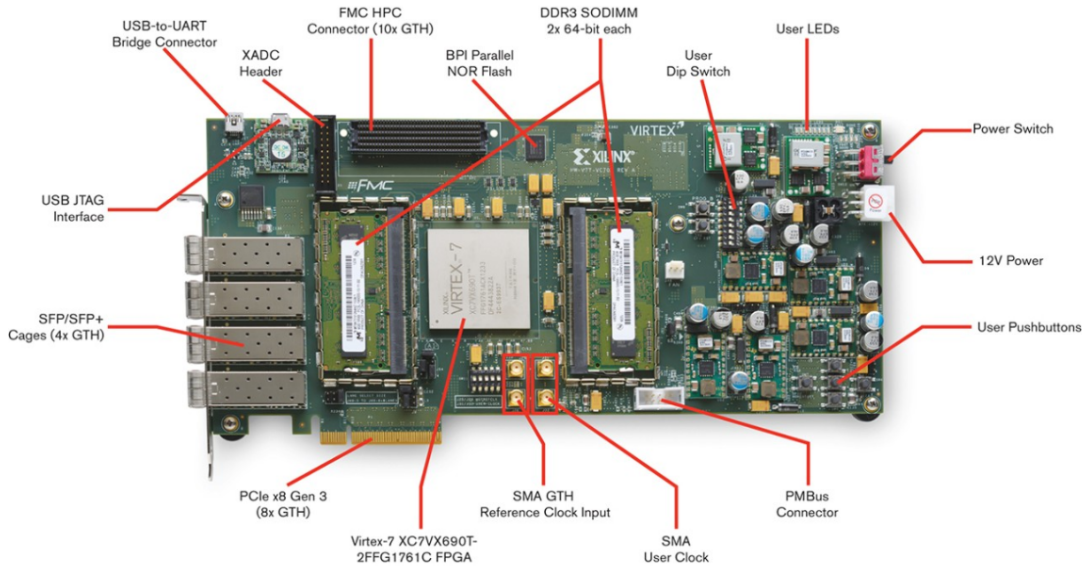


Figure 2.3: Xilinx Virtex-7 FPGA VC709 Board Features

Table 2.1: General feature summary

The number of logic cells	693120
The number of DSP Slices	3600
Memory (Kb)	52920
GTH 13.1Gb/s Transceivers	80
I/O Pins	1000

solely through a Digilent onboard USB-to-JTAG configuration logic module where a host computer accesses the VC709 board JTAG chain through a type-A (host side) to micro-B (VC709 board side) USB cable [48]. Moreover, the VC709 board contains a Silicon Labs CP2103GM USB-to-UART bridge device which allows a connection to a host computer with a USB port [48]. It allows communication between the host computer and the board. The data is sent over a serial line at a specific frequency known as the baud rate. Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both devices must operate at the same baud rate.

The block RAM (BRAM) in Xilinx 7 series FPGAs stores up to 36 Kbits of data and can be configured as either two independent 18 Kb RAMs, or one 36 Kb RAM. The write and read operations are synchronous; the two ports are symmetrical and totally independent, sharing only the stored data. The memory content can be initialized or

cleared by the configuration bitstream [47]. Each block RAM contains optional address sequencing and control circuitry to be configured as first-in/first-out (FIFO) memory with common or independent read and write clocks. It can be designed as an 18 Kb or 36 Kb FIFO [47]. The hardware design for this FPGA is expressed in RTL using Verilog hardware description language (HDL) and it is implemented with a tool suite provided by Xilinx named Vivado version 2016.4.

Chapter 3

FPGA Implementation

3.1 Introduction

In this chapter, we explain our design structure in detail. The architectural overview of the developed hardware design is depicted in Figure 3.1. The design is implemented on a Xilinx VC709 development board containing a Virtex-7. Verification of the Černý conjecture for 12-state automata is examined using the proposed implementation. The top-level interface handles a unary automaton A and a unary automaton B received from the USB-to-UART bridge. A binary automaton is created by using A and each state name permutation of B in the proposed implementation. The BFS algorithm is applied to each generated binary automata to find the shortest synchronizing sequences. The algorithm starts at a node labeled as “all states”, into which all states are encoded. The idea is to apply an input to every state simultaneously in the beginning. The process is repeated for the node that contains the resulting states. If and when the BFS reaches to a node labeled by a single state for a binary automaton (i.e., the same state is encoded), the current depth of the BFS tree is controlled. If the depth is greater than $(n - 1)^2$, this means that the conjecture is falsified. Otherwise, it is verified for the current automaton pair. Another possibility is that no node labeled by a single state is encountered during the search. In this case, it is understood that the binary automaton is not synchronizable, and it cannot falsify the Černý conjecture. Also, it is possible to finish the BFS search early by using

some filters. After finishing the BFS search of a binary automaton generated by using A and a permutation of B , the next permutation for B is considered and the analysis is repeated for the new permutation of B with same A . Each of these processes will be discussed in the following section.

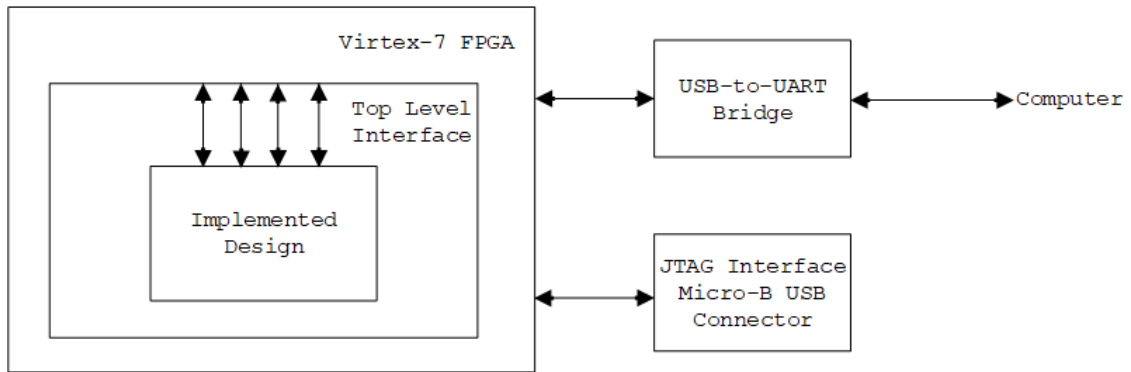


Figure 3.1: Block Diagram of Overall Structure

3.2 Proposed Structure

The proposed hardware design is expressed in RTL using Verilog HDL and was compiled using Vivado 2016.4. The design was implemented on a Xilinx VC709 development board containing a Virtex-7 FPGA device. The clock frequency is fixed at 100 MHz. This section will describe the entire FPGA design in a top-down approach. The overview of the design structure is presented as a block diagram in Figure 3.2.

The top-level module, namely `uart_top`, is the part of the design that controls UART communication with the host computer and all other functional modules in the FPGA device. UART (Universal Asynchronous Receiver and Transmitter) provides bi-directional asynchronous communication using the computer's serial ports. It provides communication by converting incoming serial data into parallel data or transmitting parallel data into serial data. Data are sent and received at baud rate of 921,600.

The module `uart_top` receives input data via the serial input line `Usb_uart_rx`, which are combined into bytes within the module `RX_serial` in Figure 3.2. Then the input bytes are transferred to module `manageMod` and the computation for the conjecture

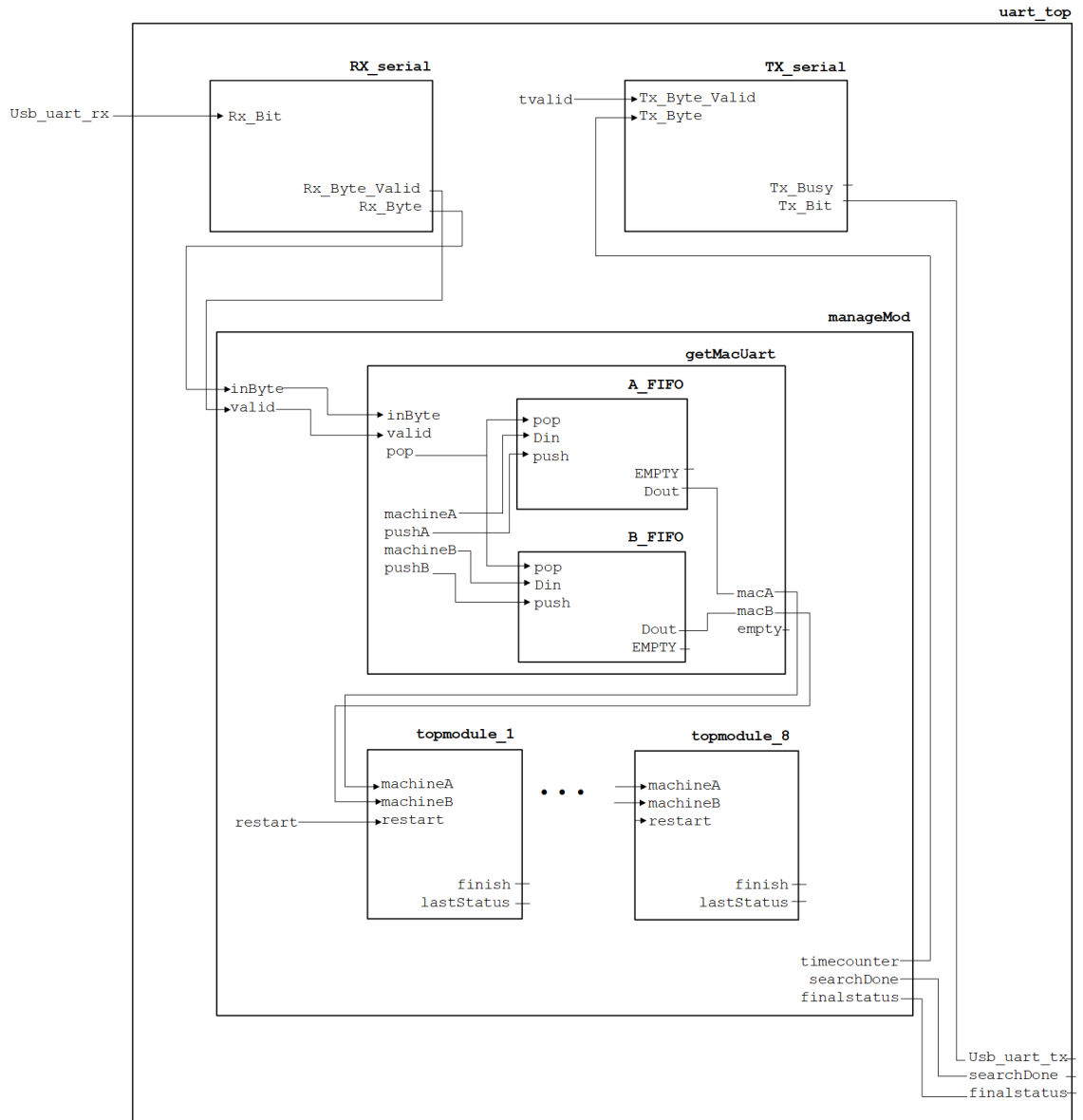


Figure 3.2: Block Diagram of Submodules inside `uart_top`

testing starts. Thereafter, `uart_top` waits for the modules to finish computation and outputs the resulting data.

The automata A and B are analyzed for finding shortest synchronizing sequences in the submodules. The analysis results are transferred to the top module with the utilization of two signals: `searchdone` and `finalstatus`. They simply indicate that i) if the search of all automaton pairs is finished or not and ii) if the Černý conjecture is verified or not. Precisely speaking, the signal, `searchdone`, stays low as the analysis continues. It is asserted high when the analysis of all the automata finishes. The signal, `finalstatus` indicates whether the conjecture is falsified. If it stays low (it is reset initially), there is no such pair that disproves the conjecture. If it is high, the conjecture is falsified. In the latter case, the result is saved and reported if it ever happens.

Additionally, the duration of analysis is measured and communicated to the top module using `timecounter`. The timing value is then sent to the host computer using the module `TX_serial` that converts it to serial data for transmission.

The module `manageMod` consists of two submodules: i) `getMacUart` and ii) eight instances of `topmodule` (although the number of instances is programmable). The module `getMacUart` contains two FIFO (first-in first-out) buffers, one for each of automata A and B . As the host computer usually sends more than one pair of automata, the FIFO buffers are used to store them before processing. The module `topmodule` is where the actual conjecture testing is performed. It implements two main functionalities: i) it first generates the permutations of B to combine two unary automata A and B into a binary automaton as described previously; and ii) every binary automaton obtained this way is tested with input sequences using breadth first search algorithm as explained in detail in the subsequent sections.

The FIFO buffers have an input signal `pop`, which is asserted to transfer an automaton to `topmodule`. As it is connected to both buffers, the pair A and B is transferred synchronously.

Each of the `topmodule` modules is designed to test one unary automata pair (A , B) concurrently. Furthermore, each `topmodule` generates the permutations of B to form

binary automata and can test 60 binary automata concurrently (again the number of binary automata is programmable in our design).

The computation usually takes different number of clock cycles in different instances of `topmodule`. An instance of `topmodule` will pop the next pair of A and B from FIFO buffers when it is done with the current pair. The instances of `topmodule` are numbered from 1 to 8 to prevent two instances from accessing the buffers at the same time when they finish in the same clock cycle. The module with smaller number will have priority of accessing the FIFO buffers.

When the FIFO in the `getMacUart` module is empty, that is, when there is no new A and B automata pair to test, the ongoing computations in submodules are allowed to complete. The `searchdone` signal is asserted high when all eight modules are done. A module can generate two outcomes: i) the conjecture is falsified (logical-1) and ii) the conjecture is verified for the currently tested automata (logical-0). The outcome is OR-ed to the signal `finalstatus`, which is initially set to logical-0. Consequently, `finalstatus` will be asserted when the conjecture is falsified. When that happens, the computations come to a stop. As long as there is no instance that falsifies the conjecture, `finalstatus` will remain low.

Furthermore, a counter variable, `timecounter` is used to measure the execution time of the conjecture testing for all the (A, B) pairs sent from the host computer to the FPGA. `timecounter` is started when the first pair of unary automata is popped from the FIFOs, and then stopped when `searchdone` becomes high. At the end, this value is output to the top module from `manageMod`.

The data coming from UART are transformed into bytes in `RX_serial` module. These bytes then are combined to obtain unary automata A and B to be stored in two separate 36 Kb FIFO buffers in `getMacUart` module, respectively. The state diagram for the process is illustrated in Figure 3.3.

The state machine in Figure 3.3 is initially in `idle` state and remains so when there is no data in `RX_serial`; i.e., `valid` signal is 0. When there is a byte in `RX_serial`, `valid` signal is asserted and the state machine will go to `initial` state. When in

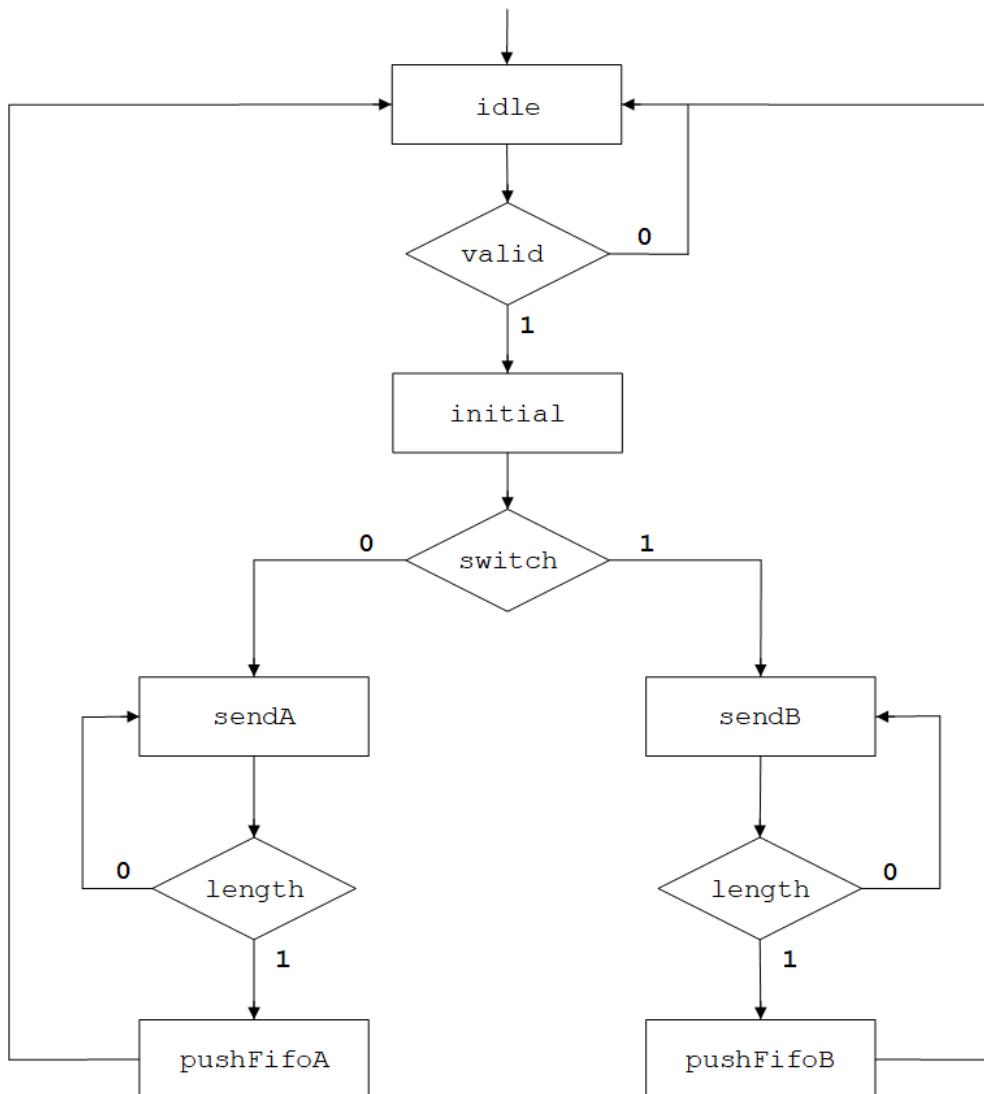


Figure 3.3: State Machine for Transferring Two Unary Automata A and B from `RX_serial` to FIFOs in `getMacUart`

initial state, `switch` variable is checked. Depending the value of `switch` the state machine can take one of the two paths. If it is asserted high, `sendA` state is reached; this indicates that this is automaton A and `A_FIFO` is selected. Otherwise, the state machine goes to `sendB` state and `B_FIFO` is selected.

The incoming data are combined into A (B) automaton in `sendA` (`sendB`) state. When all necessary bytes are received to define an automaton, `length` variable is asserted high and the state machine goes to `pushfifoA` (`pushfifoB`) state. Thereafter,

the automaton is pushed into the corresponding FIFO. The state machine transitions back to `idle` state in the following clock cycle and waits for `valid` signal. When one loop in the state machine is completed (from `idle` state back to itself), the variable `switch` is complemented.

Unary automaton pairs A and B are read from the FIFOs using the signal `pop` and transferred to `topmodule` modules. Moreover, the variable `empty` is used to indicate whether the FIFOs are full or empty. When `empty` is high, it means that there is no more automata to be read.

Each instances of `topmodule` has three inputs: i) `machineA` for automaton A , ii) `machineB` for automaton B , and iii) `restart` signal. It has two outputs: i) `finish` to indicate that the computation is finished for the current unary automata pair and ii) `laststatus`, which is set to 1 to indicate whether a counterexample that falsifies the conjecture is found. A more detailed overview of the architecture of `topmodule` is presented in Figure 3.4.

Mainly, the process inside `topmodule` consists of the following steps. Firstly, n -bit permutations of the unary automaton B are computed and with each permutation a new binary automaton is created. The permuted automata B are written into a 36 Kb FIFO (`permFIFO` in Figure 3.4). Then, the permuted automata B in `permFIFO` along with A are sent to the modules `incSearch` for conjecture testing operation.

Each binary automaton (generated by combining A and a different permutation of B) is tested in a unit named `incSearch`. There are 60 instances of `incSearch` in each `topmodule`, and therefore 60 binary automata can be tested at the same time. The instances of the execution module `incSearch` are numbered from 1 to 60 as can be seen in Figure 3.5.

Finally, `perm_FIFO` becomes empty, all ongoing testing operations are allowed to finish. When all operations are finished, the variable `finish` is asserted. Thus, it is understood that the analysis of one unary automata pair is completed. When a new unary automata pair is sent to an instance of `topmodule`, the `restart` signal is set to high resetting the buffers in the instance. Thereupon, the same operations for the previous pair

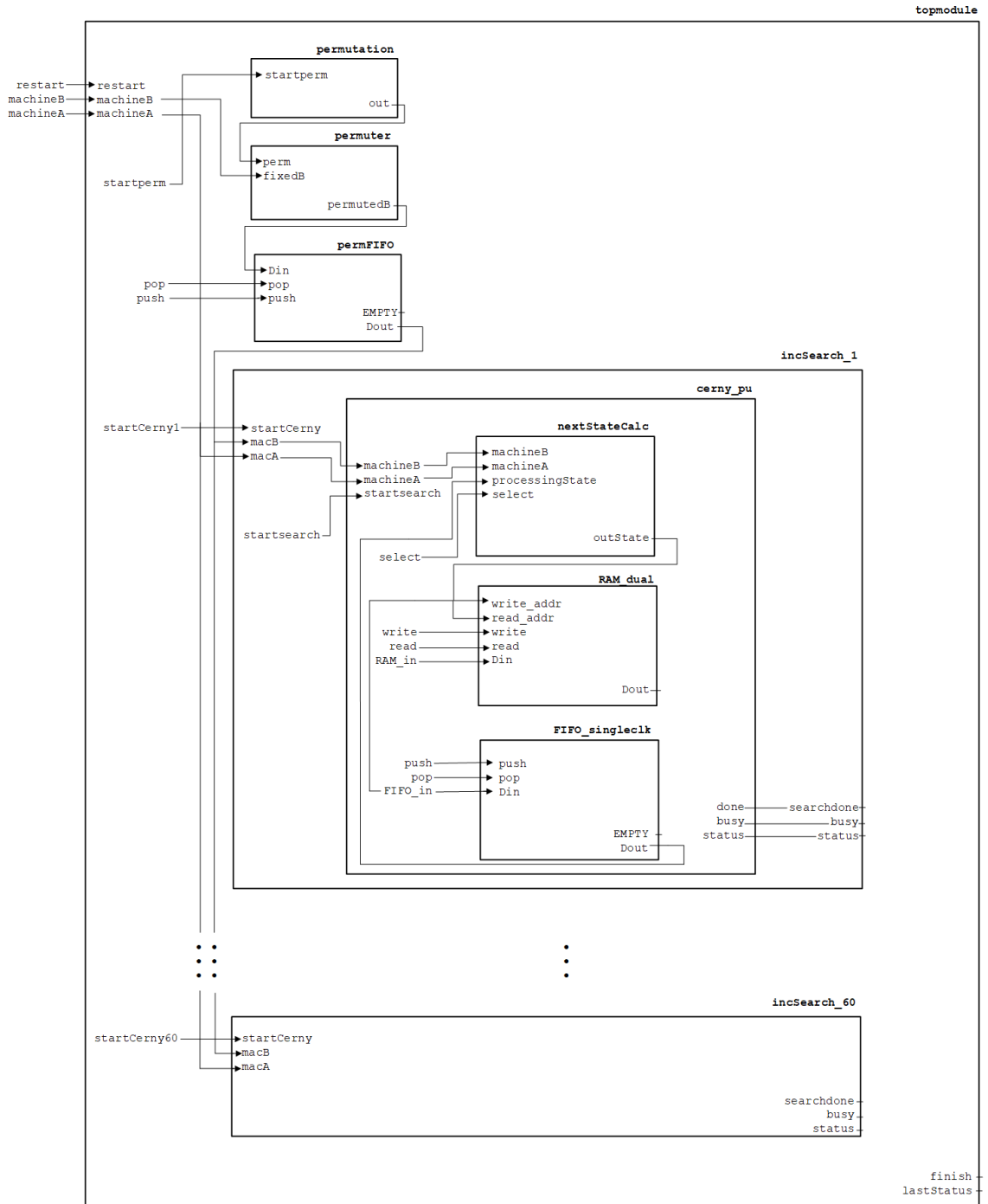


Figure 3.4: Block Diagram of Submodules inside `topmodule`

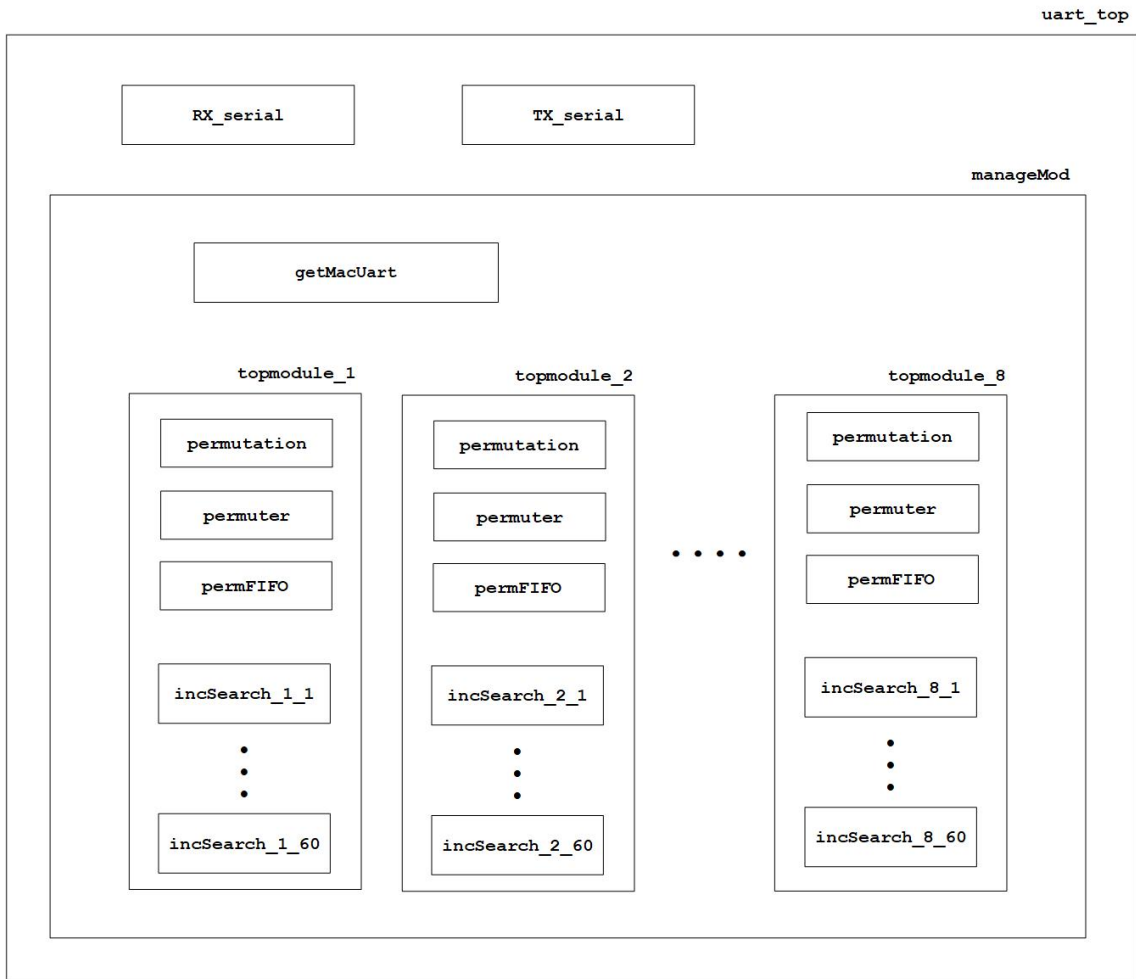


Figure 3.5: General Block Diagram of Module Instantiation

are repeated for the new pair.

The permutation of B is done in two modules: i) `permutation` and ii) `permuter`. The states of a unary automaton B can be indexed using the set of integers $\{n - 1, n - 2, \dots, 0\}$, where n is the number of states. Therefore, the permutation can be defined as a different ordering of this set. The `permutation` module generates a permutation and this permutation is applied to B in the module `permuter`.

The permutation operation starts with a specified initial permutation set and produces other orderings of states by changing the positions of elements in the initial set. The sequence for an n -state initial permutation set is in the form of $\{n - 1, n - 2, \dots, 0\}$. This form also indicates the position of the corresponding bit in the initial set. There are $n!$

permutations for n states.

For example, the set with two states has two permutations. The first one is initial set $\langle 1, 0 \rangle$. The second one can be achieved by exchanging the positions of the two elements, namely, $\langle 0, 1 \rangle$. Similarly, the set with three states has six permutations. All permutations can be obtained by exchanging 0th and 1st elements in odd numbered iterations and exchanging 0th and 2nd elements in even numbered iterations. For example, assuming the initial permutation is $\langle 2, 1, 0 \rangle$, in the first iteration we obtain the permutation $\langle 2, 0, 1 \rangle$. In the second iteration we obtain $\langle 1, 0, 2 \rangle$.

The set with 4 states has 24 permutations. When the permutations are examined, the first 6 permutations can be obtained exactly as in the three state permutation case by applying the same rule on the states in positions 2, 1, and 0. In every sixth iteration, we exchange the elements in positions 0 and 3. We perform the same operation on elements in positions 1 and 2.

The set with 5 states has 120 permutations. The permutations for five state sets can be obtained using the method used for 4-state case. The method for 4-state case is applied to all states except for the one in the position 4. In every 24th iteration, the elements in the positions 0 and 4 are exchanged and the elements in the positions 1 and 3 are also exchanged. In general, the permutations for n states can be obtained using the permutations for $n - 1$ states.

In conclusion, the process starts from the initial set and keeps producing permutations until the initial set is reached again. The set with 12 states has 479,001,600 permutations. Our design produces one permutation per clock period. As previously explained, after a new ordering is obtained in `permutation` module, the actual permutation is applied by `permuter` module. The permutations are stored in `permFIFO`. Then they are popped by `incSearch` modules to be tested in `cerny_pu` module. When the test is done for the current pair of automata, the result is output and the analysis is started with a new pair of automata.

Essentially, given automata pairs are tested for shortest synchronizing sequences in module `cerny_pu` using a breadth first search (BFS) algorithm. In the first iteration, both

A and B automata are applied all possible states and two new set of states are obtained. Each set of states is called a *state node* or shortly *node*. The initial node contains all the states. After an automaton is applied to a node, a new node is obtained, in which the same state may appear more than once. This implies that an input symbol takes more than one states into the same state. When a node that contains only one state, referred as a *singleton node*, is obtained, a synchronizing sequence is found and its length is checked. Alternatively, the process can also terminate when no new node of states is found.

The BFS algorithm starts with the initial node and obtains new nodes by applying both automata in the pair to the initial node. As we repeat the process for the new nodes we obtain a graph. We also check if a node previously occurred. If so, we do not proceed with this node. As long as a new node is obtained, the process continues and we traverse the graph.

FIFO (see `FIFO_singleclk` in Figure 3.4) and block RAM (BRAM) memory (see `RAM_dual` in Figure 3.4) structures are used to provide the graph traversal. The module `FIFO_singleclk` is used to save all generated nodes of states. A node is encoded using an n -bit node index, each of which corresponds to one state. A bit in the node is set to 1, if the corresponding state appears in the node. The nodes are mapped into `RAM_dual` locations using the node index. Consequently, if a node is obtained during the BFS algorithm, the corresponding location in `RAM_dual` is set to 1. This way, we can check if a node is generated previously and guarantee that the states in the `FIFO_singleclk` are unique.

`FIFO_singleclk` is initially pushed the value of all 1s (i.e., all 1s of n -bit), implying all states occur in the initial node. Also, the corresponding location for this initial node in `RAM_dual` is set to 1, which means the node is visited. After this point, the operations are done in a repeated fashion. A state node is read from `FIFO_singleclk` and sent to `nextStateCalc` module where it goes to new nodes of states by applying a and b inputs. The address of new state set is checked in `RAM_dual`. If the value is low, this means it is not seen before. Thus, it is written to `FIFO_singleclk` and the address in `RAM_dual` is asserted high to indicate it is seen before. If the value is high, no actions

are done for this previously visited address. The calculations are continued with new state sets that are popped from `FIFO_singleclk`.

The main termination condition for the test is to check that a node goes to a singleton, i.e. a node which contains a single 1. When a singleton is reached, the search is completed. At this point, the length of the input sequence that brings the initial state to a singleton is taken into consideration. If the length of the input sequence is more than $(n - 1)^2$, the conjecture is falsified and it sets `status` signal (see Figure 3.4) to 1. Otherwise, the conjecture is verified and `status` remains 0.

Another termination condition is when `FIFO_singleclk` is not pushed a new node as all the obtained nodes have been previously reached. If this happens, independent of whether a singleton is reached, the operation is terminated. Then, it is assumed that a synchronizing sequence is not found for this automaton. In this case, conjecture is not falsified and therefore the `status` signal remains zero.

The `nextStateCalc` module in Figure 3.4 implements the state transitions when a length one input a or b is applied to a given node. With the `select` input, A or B automaton is applied to the node and a new state node is reached. If `select` is 1, then automaton A is selected. If it is 0, automaton B is selected. Automaton A defines the states that are reached by applying the input a to all possible states. Similarly, automaton B describes the states that are reached by b input. The newly generated node of states is returned in the variable `outState`.

The state machine illustrated in Figure 3.6 explains the operations in the module `cerny_pu`. With the `reset` input, the state machine by default goes to the `idle` state. In this state, it waits for the `startsearch` signal to be asserted. Once the `startsearch` is 1 the state machine transitions to the `initial` state. In the `initial` state, the initial node (all 1s indicating all state's existence) is created and sent to the `FIFO_singleclk` module. Thereafter, the state machine goes to the `read` state in which the nodes are popped from `FIFO_singleclk`. The `read` state returns `idle` if one of the *termination conditions* is satisfied; otherwise, the state machine transitions to the `writeA` state.

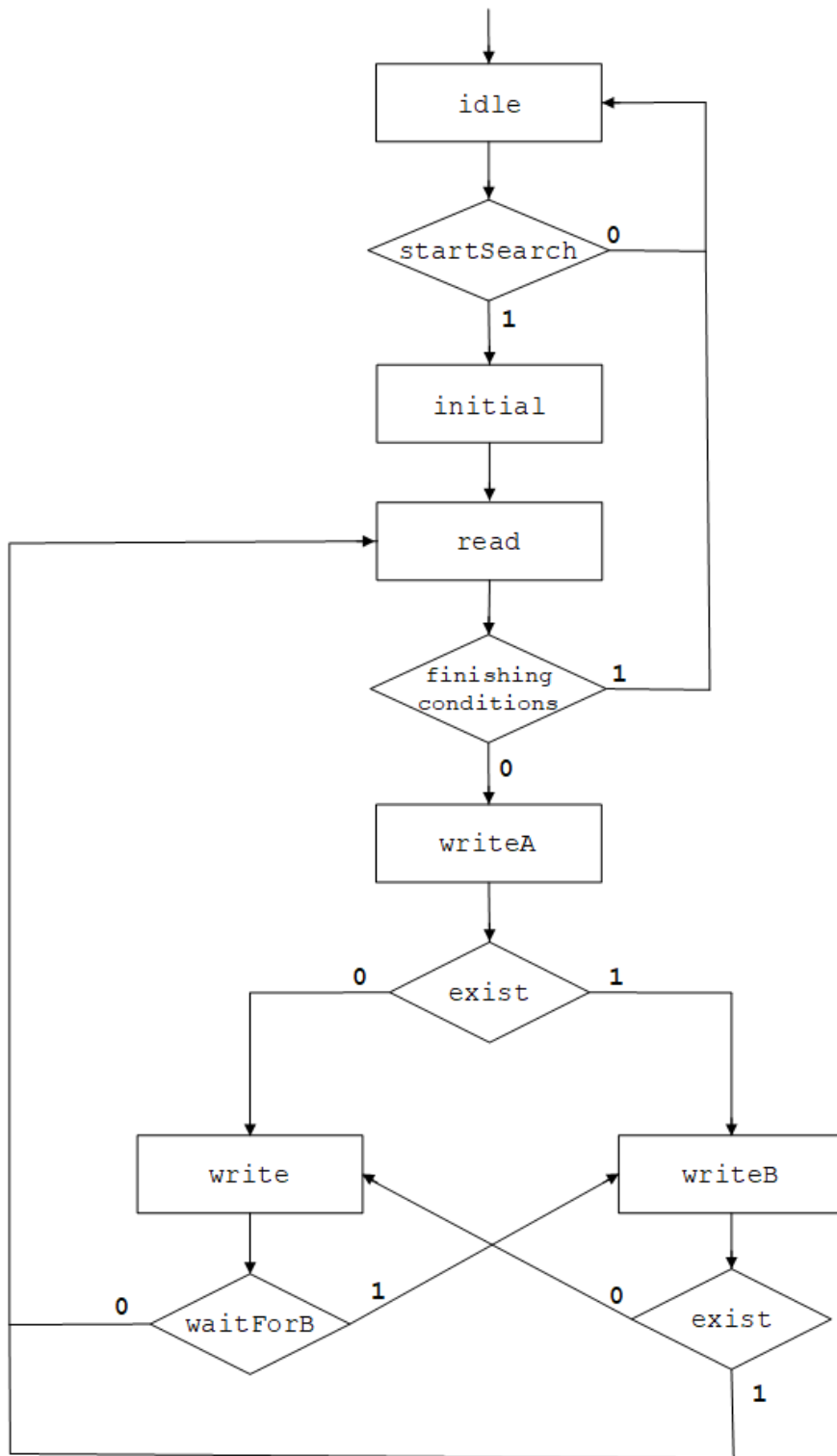


Figure 3.6: State Diagram of State Machine in cerny_pu

In the `writeA` state, the input a is applied to the current node and a new node of states is obtained. Also in this state, we check if the node is already in `RAM_dual`. If it does not, the signal `exist` signal is set to 0, the state machine goes to the `write` state, in which the newly generated node is pushed to `FIFO_singleclk`. Moreover, the corresponding address `RAM_dual` is asserted to indicate that this node is reached. In the `write` state, if a signal called `waitForB` is 1, the state machine goes to `writeB` state. Otherwise, it returns to the `read` state. If `exist` signal in `writeA` state is high, state machine transitions `writeB` state. This state is the same as `writeA` state but with input b . Instead this time if the `exist` signal is high, it returns to `read` state. Similar operations are performed when the state machine is in the `writeB` state.

A block RAM of 18 Kb is created to implement the `RAM_dual` module. The RAM used is simple dual-port mode. In this mode, independent read and write operations can be performed at the same time. In other words, reading a port and writing another port can be done simultaneously. These ports are symmetrical and independent of one another and share only the stored data.

To update or read a BRAM, the write or read addresses are provided, and the two signals `read` and `write` are asserted. When these signals are set to 1, the operations are started. Otherwise, these signals are 0 and this means that there is no reading activity or change in BRAM. After reading or writing operations are completed, `read` or `write` return to 0.

Addresses in the `RAM_dual` module represent nodes of states. When the value in an address is checked, it reveals whether this address (i.e. the corresponding set of states) has been visited before or not. Initially, the contents of all addresses have zero value. If the value becomes 1, it means that this address, in other words, this node has appeared before. In this case, no further action is taken for this node and a new node of state is generated.

FIFO memories of 18 Kb and 36 Kb are configured to implement the `FIFO_singleclk` and `permFIFO` modules, respectively. Recall that `permFIFO` module keeps automata pairs obtained via permutation while the `FIFO_singleclk` keeps track of nodes of

states as they are generated using an automata pair. As the name indicates a write operation *pushes* a new data item to the end of the buffer whereas a read operation *pops* the item from the head of the buffer. Two signals are used to activate these operations: `push` and `pop`. Both of these signals are initially assigned to 0. A push command is issued by setting `push` to 1 and a new data is pushed to the memory. A node of states given as input is written to the tail of the FIFO. `pop` is set to 1 to enable pop operation and the oldest value in the buffer is removed. When these values are 0, write and read operations are disabled.

As stated previously, the `FIFO_singleclk` content is set to 0 initially. When conjecture testing for an automata pair is started some addresses are set to 1. When the testing is finished and another one is started, the values in the RAM block does not return to the initial values immediately in one clock cycle. In other words, it is not possible to reset the values in all addresses within the same clock cycle. For this reason, it is necessary to reset the addresses one by one.

Thus, one extra FIFO and one extra RAM block are used to mitigate the overhead caused by this operation. The RAM blocks are named as `Ram1` and `Ram2`. `searchFifo` and `deleteFifo` are the names of two FIFOs. When a search for synchronizing sequence of the first automata pair is ongoing, addresses are checked in `Ram1`, and new occurrences of nodes are transferred to both `searchFifo` and `deleteFifo`. Nodes are popped from `searchFifo` to test the automata pair. When this search ends, `searchFifo` is reset but `deleteFifo` does not change.

Addresses are now checked in `Ram2` for second automata pair, and new nodes are similarly transferred to both `searchFifo` and `deleteFifo`. Values are popped from `searchFifo` for new state calculations while values popped from `deleteFifo` are used to determine addresses of `Ram1` to be reset. This way, resetting of `Ram1` is overlapped with the testing of another automata pair being in progress and no cycle is wasted for resetting a BRAM. Use of `Ram1` and `Ram2` are alternated for different iterations using a control signal `selectRam`.

In addition, `deleteFifo` is not flushed after each search and the new data inputs are

written in every search. In such a case, a larger FIFO is needed in order to store all data. Therefore, one 36 Kb memory and one 18 Kb memory are used together for the FIFO to hold the deleted addresses.

Chapter 4

Experiments

This chapter presents some implementation details and experimental results to test and validate the proposed architecture. Section 4.1 provides verification methods used to test the correctness of the design. The required resources to implement the design and the experimental results are presented in Section 4.2. In particular, certain design choices are compared against each other and the results are shown. To evaluate the performance of the proposed architecture, experimental results were obtained on a Xilinx VC709 development board containing a Virtex-7 FPGA device, using Xilinx Vivado 2016.4. Moreover, we used a Linux-based PC in addition to Virtex-7 board. We used Verilog HDL for all hardware designs and C/Python language for software components on the host computer.

4.1 Verification

The correctness of the proposed implementation is the major focal point to avoid miscalculations of synchronizing sequences and to avoid misleading verification of the Černý conjecture. The design is tested in simulation environment and hardware software co-verification environment to ensure the validity of the design. Certain modifications and corrections are applied with respect to the results of these tests. The functionality of the design is verified by the behavioral simulation and board implementation.

Xilinx Vivado simulator is used to test the steps during the implementation of the

design. A test bench is written to verify the design in simulation. Using the hardware simulator, the output waveforms are observed and the signals are also checked in detail. The traces of state transitions are monitored by examining the content of the registers. The simulator also helps in the development process and facilitates the evaluation of internal methods in modules.

Since the proposed design consists of several parallel submodules, it is not very efficient to run and check simulation with all parallel submodules. By keeping the general architecture the same, fewer number of parallel modules (e.g., `topmodule`) are used for testing. In the simulator, the design is tested by one automaton pair at a time. When the termination condition is achieved, a new automaton pair is tested. State transitions with respect to automaton pairs, popping a new automaton pair, FIFO memory usage, block RAM memory usage, finite state machine transitions are all checked in the simulation environment. The FPGA implementation is verified with simulations.

Other verification tests are applied to determine whether the design produces the expected outputs. Proposed techniques for the conjecture testing are also implemented in Python code for verification purposes. The expected results of both Verilog and Python codes are compared. The accuracy of the implementation results in hardware is tested employing a host PC computer and serial communication with UART. C code is developed at the host side to receive or sent data from the serial port. The hardware implementation of UART protocol is based on Verilog HDL language.

The computation starts when the input data is read from the UART by the FPGA. Firstly, the state transition outputs of the automaton pairs are sent to the host computer via UART. The results are saved and compared with the state transition outputs of the automata pairs obtained by the Python code. Another test is also applied for checking reset word lengths which are found by the BFS algorithm. When an automaton pair is analyzed, the paths that reach a synchronizing sequence are saved and transmitted with corresponding permutation of the automaton B . The results are compared with the software results from the Python code. If there are inconsistencies between the results for both verification tests, the architecture is investigated and the necessary updates are per-

formed. The simulation helps to keep track of the processing steps. When the results of the FPGA implementation match with the results of the Python implementation, the design is considered to be verified.

Recall that for synchronizable automata if the length of the shortest reset word is larger than Černý upper bound, the conjecture is falsified. Therefore it is important to perform the upper bound check accurately; i.e., to validate that our design would catch it when a counter example that falsifies the conjecture is found. In order to make sure of that, the upper bound is artificially set to a relatively small value with respect to the average length of reset words from the previous analysis. The falsification signal is expected to activate for automata with reset words longer than this specified bound. With artificially low bounds, the test is started and the falsification signal is observed. The signal becomes active indicating that Černý bound is exceeded as expected. In summary, the simulations and experiments on the hardware implementation show that the implementation works correctly.

4.2 Timing Results

The non-isomorphic unary automata are created outside the hardware implementation (i.e., via a software implementation). The proposed hardware architecture gets two unary automata, A and B as inputs and combines them into a binary automaton. One of the given automaton pair, automaton B is permuted and each permuted B automaton is combined with the given A automaton. In this manner, $n!$ binary automata are generated from a given automaton pair with n states. All automata are checked for reset words and the analysis time is recorded. UART communication protocol allows to send and receive data between the host PC and the FPGA device.

When the number of states is 9, the number of automaton pairs to be examined is 243,243. Firstly, each generated automata are checked for synchronizing words in the CPU of the host PC. The automaton pairs are ordered depending on the execution times from the longest to the shortest. Then the automata pairs are sent to FPGA in this order

Table 4.1: Timing for n= 9

number of topmodule module	number of incSearch module	Time (seconds)
10	40	269
11	40	248
10	45	241
12	35	259

and the elapsed time for the conjecture testing is taken in the FPGA implementation. The automaton pairs are grouped into 1020. A total of 239 automaton groups are formed. Groups are sent to FPGA via UART one by one.

The first 1020 automata pairs with the longest execution times are transmitted, and another 1020 pairs are transmitted when the computation for the first group is completed. The transmission continues until all groups are sent in this order. The execution time is measured in FPGA and sent via UART when all groups of automata are completed.

Several module configurations are tested on the FPGA for 9-state automata by changing the number of `topmodule` and `incSearch` units in Figure 3.5. Different configurations are tried to maximize the utilization of FPGA resources. Initially, we used 10 `topmodule` and 40 `incSearch` units. We observed for this configuration that the time spent on hardware implementation is more than the time spent in the software implementation in the CPU of the host computer (i.e., 269 s on FPGA vs. 247 s in PC using a single CPU core). When the execution times are inspected in more detail, the FPGA implementation is faster for the automata with high execution times, while it is slower for automata that can be tested in shorter times.

Then we tried different configurations and the execution times are enumerated for each configuration in Table 4.1. The fastest configuration is obtained when we use 10 `topmodule` and 45 `incSearch` units, that slightly outperforms the PC implementation. The execution times for the automata pair groups of 1020, which are ordered from the slowest to the fastest are illustrated in Figure 4.1.

The resource utilization of the implementation is shown in Table 4.2. In addition,

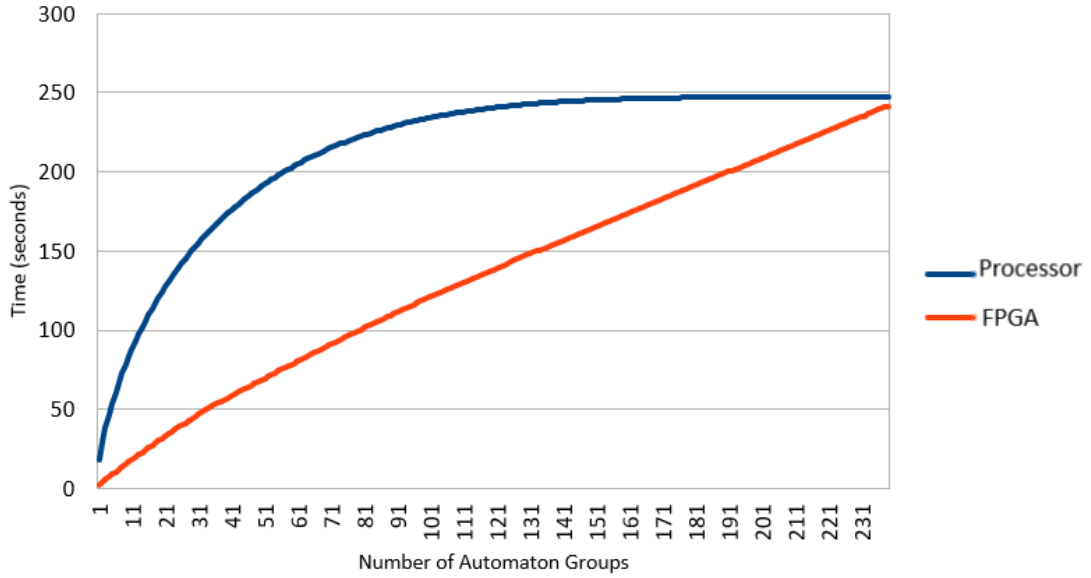


Figure 4.1: Comparison of FPGA and Processor Timings for All $n = 9$ State Automata

Table 4.2: Utilization for n=9

Resource	Utilization	Available	Utilization%
LUT	188230	433200	42.20
FF	76355	866400	8.81
BRAM	1362	1470	92.65
IO	8	850	0.94
BUFG	3	32	9.38
MMCM	1	20	5.00

utilization design information is given in Table 4.3 and Table 4.4.

Checking for synchronizing sequences for 12 state automata is done in the CPU of the host PC firstly. The automaton pairs are sorted with respect to the execution times from the longest to the shortest. Then the first 25, 50 and 100 automaton pairs taking maximum times are examined respectively in FPGA to find the length of minimal synchronizing sequences. The execution time for the conjecture testing is measured for the FPGA implementation. In order to find the shortest possible execution time, various configurations are tried by aiming maximum resource utilization. The performance comparison of the configurations is given in Table 4.5. Shortest timing is achieved with 8 `topmodule` and

Table 4.3: Slice Logic for n= 9

Site Type	Used	Available	Util %
Slice LUTs	182830	433200	42.20
LUT as Logic	182830	433200	42.20
LUT as Memory	0	174200	0
Slice Registers	76355	866400	8.81
Register as Flip Flop	76355	866400	8.81
Register as Latch	0	866400	0
F7 Muxes	1390	216600	0.64
F8 Muxes	0	108300	0

Table 4.4: Memory for n=9

Site Type	Used	Available	Util %
Block RAM Tile	1362	1470	92.65
RAMB36/FIFO	462	1470	31.43
RAMB18	1800	2940	61.22

60 `incSearch` units.

The resource utilization of the implementation is shown in Table 4.6. In addition, utilization design information is given in Table 4.7 and Table 4.8.

The performances of both the hardware and software approaches are compared in this experimental sets. The results are shown in Table 4.9.

The time spent on hardware implementation is about half of the time spent in the software implementation in the CPU of the host computer for the 100 automaton pairs.

Table 4.5: Timing for n=12

number of topmodule module	number of incSearch module	Time (seconds)		
		25 pairs	50 pairs	100 pairs
8	60	3931	5796	10213
9	50	3652	6280	10397
10	40	4489	6855	12781

Table 4.6: Utilization for n=12

Resource	Utilization	Available	Utilization%
LUT	270868	433200	62.53
FF	91130	866400	10.52
BRAM	1450	1470	98.64
IO	8	850	0.94
BUFG	3	32	9.38
MMCM	1	20	5.00

Table 4.7: Slice logic for n=12

Site Type	Used	Available	Util %
Slice LUTs	270868	433200	62.53
LUT as Logic	270868	433200	62.53
LUT as Memory	0	174200	0
Slice Registers	91130	866400	10.52
Register as Flip Flop	91130	866400	10.52
Register as Latch	0	866400	0
F7 Muxes	2528	216600	1.17
F8 Muxes	0	108300	0

Table 4.8: Memory for n=12

Site Type	Used	Available	Util %
Block RAM Tile	1450	1470	98.64
RAMB36/FIFO	490	1470	33.33
RAMB18	1920	2940	65.31

Table 4.9: Timing Comparison

Automaton pair number	FPGA	Processor
25	3931 s	4100 s
50	5796 s	8259 s
100	10213 s	21140 s

4.2.1 Applied Filters

Several filters are used on the processor to improve the performance. Such filters are not completely suitable for implementation on the proposed FPGA architecture. Some of them are applied to FPGA analysis to get better results in terms of performance. They are useful in filtering permutations.

Filter 1

It is known that if there exists a counterexample for the Černý conjecture, then there exists a counterexample where the automaton is strongly connected. Therefore, the search for a counterexample for the Černý conjecture can be restricted to strongly connected automata. For this reason, if an automaton A is not strongly connected, there is no need to consider A for the falsification of the Černý conjecture. In Filter 1, we use the following easy-to-check sufficient condition for a binary automaton $A = (Q, \{a, b\}, \delta)$ to be not strongly connected: if $\delta(Q, a) \cup \delta(Q, b) \neq Q$ then A is not strongly connected. Note that, if there exists a state $q \notin \delta(Q, a) \cup \delta(Q, b)$, then q has no incoming transition. In this case, the automaton A is not strongly connected.

Filter 2

For an automaton $A = (Q, \Sigma, \delta)$, if A is synchronizing then for any two states $q_i, q_j \in Q$ there exists an input sequence $\sigma \in \Sigma^*$ with length at most $\frac{n(n-1)}{2}$ such that $\delta(\{q_i, q_j\}, \sigma)$ is a singleton. Therefore, if there exists an input sequence σ' such that $\delta(Q, \sigma') = \{q_i, q_j\}$ where the length of σ' is at most $(n-1)^2 - \frac{n(n-1)}{2}$, then there exists a synchronizing sequence with length at most $(n-1)^2$ (simply consider using $\sigma'\sigma$ as the synchronizing sequence).

Based on this observation, during the search for the shortest synchronizing sequence using BFS, if a state set with cardinality 2 is reached at depth $(n-1)^2 - \frac{n(n-1)}{2}$ or less, then the search can be terminated. For example for $n=12$, if a state set with cardinality 2 is reached at level 55 or less during the BFS, the search is terminated.

Filter 2'

A more eager filter similar to Filter 2 has also been tried. Although $\frac{n(n-1)}{2}$ is the general upper bound for merging two states in a synchronizing automaton, it might be possible to get a better upper bound $k < \frac{n(n-1)}{2}$ for an automaton $A = (Q, \Sigma, \delta)$. Recall that in our work, a binary automaton $A = (Q, \{a, b\}, \delta)$ is constructed by considering two unary automata $A_a = (Q, \{a\}, \delta_a)$ and $A_b = (Q, \{b\}, \delta_b)$, where the names of the states of A_b are permuted.

By analyzing the transitions of A_a only, one can get an upper bound k (which might still be smaller than the general upper bound value $\frac{n(n-1)}{2}$) such that for any two states $q_i, q_j \in Q$ there exists an input sequence $\sigma \in \Sigma^*$ with length at most k such that $\delta(\{q_i, q_j\}, \sigma)$ is a singleton for any two states for any synchronizable binary automaton $A = (Q, \{a, b\}, \delta)$ that would be obtained by using $A_a = (Q, \{a\}, \delta_a)$ together with any other unary automaton $A_b = (Q, \{b\}, \delta_b)$.

To assess if such a reduced upper bound would accelerate our search for a counterexample, we have tried the upper bound of $k=60$ (instead of the general bound 55) for $n=12$. In other words, during the search for the shortest synchronizing sequence using BFS, if a state set with cardinality 2 is reached at depth 60 or less, then the search is terminated when automata with $n=12$ states are considered.

Filter 3

For a synchronizing automaton $A = (Q, \Sigma, \delta)$ with n states, if there exists an input sequence $\sigma \in \Sigma^*$ with length at most k such that $(|\delta(Q, \sigma)|) = m$, Pin [31] proved that there exists an input sequence $\sigma' \in \Sigma^*$ with length at most $2k + n - m + 1$ such that $(|\delta(Q, \sigma')|) \leq m - 1$. Also recall that for a synchronizing automaton, for any two states $q_i, q_j \in Q$ there always exists an input sequence $\sigma' \in \Sigma^*$ with length at most $\frac{n(n-1)}{2}$ such that $\delta(\{q_i, q_j\}, \sigma')$ is a singleton. Combining these two results, if $|\delta(Q, \sigma)| = 3$, for an input sequence of σ length at most $\lfloor (n^2 - 5n + 6)/4 \rfloor$, then there exists a synchronizing sequence of length at most $(n - 1)^2$. Therefore during the BFS search for $n=12$, if state set is reduced to 3 or less states at level 22 or earlier, the search can be terminated.

Table 4.10: Filter Timing Comparison

Filters	Time (seconds)
Filter 1	9714
Filter 2	7335
Filter 2'	6832
Filter 3	4029
Filter 1 & 2	6650
Filter 1 & 2'	6557
Filter 1 & 3	3373
Filter 1 & 2 & 3	3839
Filter 1 & 2' & 3	3813

Previously explained filters are added to the design respectively. The combinations of filters are also applied. The measured time performances for examining 100 pairs of automaton are stated in Table 4.10. We get the fastest timing performance when we applied Filter 1 and Filter 3 combination. The time spent on hardware implementation is about six times faster than the time spent in the software implementation in the CPU of the host computer (i.e., 3373 s on FPGA vs. 21140 s in PC using a single CPU core).

Chapter 5

Conclusion

The study presented in this thesis gives the details of the design and the implementation of a parallel computation methods for generating shortest synchronizing sequences of binary automata using FPGA. The proposed design is developed to improve the performance of experimental methods for 12-state binary automata. BFS algorithm is considered to find the length of synchronizing sequences.

The proposed implementation handles a pair of unary automaton A and unary automaton B . It creates binary automata by using A and each state name permutation of B . Each generated binary automaton is examined with BFS algorithm to find the length of a shortest synchronizing sequence. When a singleton is reached during the search, the depth of the graph is controlled. If the depth is greater than $(n - 1)^2$, it means that the Černý conjecture is falsified. Otherwise, it is verified for the current automaton pair. Another possibility is that no single-state cluster is encountered during the search. In this case, it is understood that the automaton pair is not a synchronizable, and it cannot falsify the Černý conjecture. Also it is possible to terminate search with using the filters. After finishing one search, the next permutation for B is considered and the analysis is repeated for the new permutation of B with same A .

We proposed a hardware based solution in this thesis, which mostly utilizes memory blocks on FPGA to solve the shortest synchronizing sequence problem. FPGA provides high productivity for parallel processing. Experimental results obtained using a Xilinx

Virtex-7 FPGA shows that the proposed solution achieves a performance 6 times higher than the existing one core processor performance. When we analyzed the performance results with respect to existing processor results, we observed that short time-consuming automata in software, take a long time in hardware. Since several filtering methods are applied to automaton pairs that eliminate some cases without checking them, total execution time is accelerated in CPU. However, these methods are not applicable for FPGA implementation and all cases need to be considered. Instead, some suitable filters (see Chapter 4.2.1 Applied Filters) that can finish the search early are applied to proposed design. Also, we have seen that these methods on processor are not successful on long time consuming automata. Furthermore, as our previous analysis shows that our implementation tends to have better performance with long time consuming automata.

Finally, our approach achieves 6 times better throughput than the existing one core software architectures. We expect to have much more performance gain if using hardware software mixed architectures. Long time consuming automata can be checked in FPGA part, whereas short time consuming automata can be checked in CPU part. One approach to the continuation of the project's work within this research area can be to address the 2-input and 13-state automata. However, observations we have made from our experiments and theoretical calculations show that an increase in the number of cases means dramatic increase in the total analysis time. For this reason, a new study that will target 2-input and 13-state automata systems also needs to use some theoretical methods for eliminating more automaton cases. In addition, hardware and software mixed architectures can achieve drastically better performance for this case as well.

References

- [1] Ananichev, Dmitry, Gusev, Vladimir, and Volkov, Mikhail. “Slowly synchronizing automata and digraphs”. In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2010, pp. 55–65.
- [2] Babb, Jonathan W, Frank, Matthew, and Agarwal, Anant. “Solving graph problems with dynamic computation structures”. In: *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*. Vol. 2914. International Society for Optics and Photonics. 1996, pp. 225–237.
- [3] Béal, Marie-Pierre. “A note on Cerny’s conjecture and rational series”. In: *preprint IGM 5 (2003)*.
- [4] Benenson, Yaakov, Paz-Elizur, Tamar, Adar, Rivka, Keinan, Ehud, Livneh, Zvi, and Shapiro, Ehud. “Programmable and autonomous computing machine made of biomolecules”. In: *Nature* 414.6862 (2001), p. 430.
- [5] Berlinkov, Mikhail. “The Cerny Conjecture”. In: *arXiv preprint arXiv:1204.0856 (2012)*.
- [6] Berlinkov, Mikhail V. “Extension Method and The Cerny Conjecture”. In: *CoRR abs/0909.3790 (2009)*. arXiv: 0909.3790. URL: <http://arxiv.org/abs/0909.3790>.
- [7] Betkaoui, Brahim, Wang, Yu, Thomas, David B, and Luk, Wayne. “A reconfigurable computing approach for efficient and scalable parallel graph exploration”. In: *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*. IEEE. 2012, pp. 8–15.

- [8] Ceriani, Marco, Palermo, Gianluca, Secchi, Simone, Tumeo, Antonino, and Villa, Oreste. “Exploring manycore multinode systems for irregular applications with FPGA prototyping”. In: *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE. 2013, pp. 238–238.
- [9] Cerný, Ján. “Poznámka k. homogénnym experimentom s konečnými automatmi”. In: *Mat. fyz. cas SAV* 14 (1964), pp. 208–215.
- [10] Chmiel, Krzysztof and Roman, Adam. “COMPAS-A computing package for synchronization”. In: *International Conference on Implementation and Application of Automata*. Springer. 2010, pp. 79–86.
- [11] Chow, Tsun S. “Testing software design modeled by finite-state machines”. In: *IEEE transactions on software engineering* 3 (1978), pp. 178–187.
- [12] Dai, Guohao, Chi, Yuze, Wang, Yu, and Yang, Huazhong. “Fpgp: Graph processing framework on fpga a case study of breadth-first search”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2016, pp. 105–110.
- [13] Dandalis, Andreas, Mei, Alessandro, and Prasanna, Viktor K. “Domain specific mapping for solving graph problems on reconfigurable devices”. In: *International Parallel Processing Symposium*. Springer. 1999, pp. 652–660.
- [14] Eppstein, David. “Reset sequences for finite automata with application to design of parts orienters”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1988, pp. 230–238.
- [15] Eppstein, David. “Reset sequences for monotonic automata”. In: *SIAM Journal on Computing* 19.3 (1990), pp. 500–510.
- [16] Gill, Arthur. *Introduction to the theory of finite-state machines*. McGraw-Hill electronic sciences series, 1962.
- [17] Harary, Frank and Palmer, Edgar M. *Graphical enumeration*. Elsevier, 2014.

- [18] Johnson, Selmer M. “Generation of permutations by adjacent transposition”. In: *Mathematics of computation* 17.83 (1963), pp. 282–285.
- [19] Kisielewicz, Andrzej, Kowalski, Jakub, and Szykuła, Marek. “Experiments with synchronizing automata”. In: *International Conference on Implementation and Application of Automata*. Springer. 2016, pp. 176–188.
- [20] Kisielewicz, Andrzej and Szykuła, Marek. “Synchronizing Automata with Large Reset Lengths”. In: *CoRR* abs/1404.3311 (2014).
- [21] Kisielewicz, Andrzej and Szykuła, Marek. “Generating small automata and the Černý conjecture”. In: *International Conference on Implementation and Application of Automata*. Springer. 2013, pp. 340–348.
- [22] Kohavi, Zvi and Jha, Niraj K. *Switching and finite automata theory*. Cambridge University Press, 2009.
- [23] Kudłacik, Rafał, Roman, Adam, and Wagner, Hubert. “Effective synchronizing algorithms”. In: *Expert Systems with Applications* 39.14 (2012), pp. 11746–11757.
- [24] Lee, David and Yannakakis, Mihalis. “Testing finite-state machines: State identification and verification”. In: *IEEE Transactions on computers* 43.3 (1994), pp. 306–320.
- [25] Lei, Guoqing, Li, Rongchun, Guo, Song, and Xia, Fei. “TorusBFS : A Novel Message-passing Parallel Breadth-First Search Architecture on FPGAs”. In: (2015).
- [26] Liskovets, Valery A. “Exact enumeration of acyclic deterministic automata”. In: *Discrete Applied Mathematics* 154.3 (2006), pp. 537–551.
- [27] Martyugin, Pavel V. “Complexity of problems concerning carefully synchronizing words for PFA and directing words for NFA”. In: *International Computer Science Symposium in Russia*. Springer. 2010, pp. 288–302.
- [28] Natarajan, Balas K. “An algorithmic approach to the automated design of parts orienters”. In: *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE. 1986, pp. 132–142.

- [29] Pin, Jean-Eric. *Le probleme de la synchronisation et la conjecture de Cerný*. 1981.
- [30] Pin, Jean-Eric. “Sur le monoïde de L^* lorsque L est un langage fini”. In: *Theoretical Computer Science* 7 (1978), pp. 211–215.
- [31] Pin, Jean-Eric. “Utilisation de l’algèbre linéaire en théorie des automates”. In: *1er Colloque AFCET-SMF de Mathématiques Appliquées*. 1978, pp. 85–92.
- [32] Robinson, Robert W. “Counting strongly connected finite automata”. In: *Graph theory with applications to algorithms and computer science*. John Wiley & Sons, Inc. 1985, pp. 671–685.
- [33] Roman, Adam. “A note on Černy conjecture for automata over 3-letter alphabet”. In: *Journal of Automata, Languages and Combinatorics* 13.2 (2008), pp. 141–143.
- [34] Roman, Adam and Foryś, Wit. “Lower bound for the length of synchronizing words in partially-synchronizing automata”. In: *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer. 2008, pp. 448–459.
- [35] Secchi, Simone, Ceriani, Marco, Tumeo, Antonino, Villa, Oreste, Palermo, Gianluca, and Raffo, Luigi. “Exploring hardware support for scaling irregular applications on multi-node multi-core architectures”. In: *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE. 2013, pp. 309–313.
- [36] Sedgewick, Robert. “Permutation generation methods”. In: *ACM Computing Surveys (CSUR)* 9.2 (1977), pp. 137–164.
- [37] Starke, Peter H. “Eine bemerkung über homogene experimente”. In: *Elektr. Informationverarbeitung und Kyb* 2 (1966), pp. 257–259.
- [38] Tompkins, C. “Machine attacks on problems whose variables are permutations”. In: *Proceedings of Symposia in Applied Mathematics*. Vol. 6. McGraw-Hill New York. 1956, pp. 195–211.

- [39] Trahtman, Avraham N. “Modifying the upper bound on the length of minimal synchronizing word”. In: *International Symposium on Fundamentals of Computation Theory*. Springer. 2011, pp. 173–180.
- [40] Trahtman, Avraham N. “The Černý conjecture for aperiodic automata”. In: *Discrete Mathematics and Theoretical Computer Science* 9.2 (2007), pp. 3–10.
- [41] Trotter, Hale F. “Algorithm 115: Perm”. In: *Communications of the ACM* 5.8 (1962), pp. 434–435.
- [42] Tsay, Jong-Chuang and Lee, Wei-Ping. “An optimal parallel algorithm for generating permutations in minimal change order”. In: *Parallel computing* 20.3 (1994), pp. 353–361.
- [43] Umuroglu, Yaman, Morrison, Donn, and Jahre, Magnus. “Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform”. In: *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*. IEEE. 2015, pp. 1–8.
- [44] Volkov, Mikhail V. “Synchronizing automata and the Černý conjecture”. In: *International Conference on Language and Automata Theory and Applications*. Springer. 2008, pp. 11–27.
- [45] Volkov, Mikhail V. “Synchronizing automata preserving a chain of partial orders”. In: *Theoretical Computer Science* 410.37 (2009), pp. 3513–3519.
- [46] Wang, Qingbo, Jiang, Weirong, Xia, Yinglong, and Prasanna, Viktor. “A message-passing multi-softcore architecture on FPGA for breadth-first search”. In: *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE. 2010, pp. 70–77.
- [47] Xilinx. *7 Series FPGAs Memory Resources User Guide*. 2016.
- [48] Xilinx. *VC709 Evaluation Board for the Virtex-7 FPGA User Guide*. 2016.

- [49] *Xilinx Virtex-7 FPGA VC709 Connectivity Kit*. URL: <https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html#hardware> (visited on 07/06/2018).