NTNU

Norwegian University of
Science and Technology

System for Running Replicating Non-deterministic Finite Automata Using Partial Reconfiguration

Nicholas Paulsen
Lohrmann

Master of Science in Electronics
Submission date: June 2016
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

ABSTRACT

Partial reconfiguration has many practical applications, and is an increasingly important requirement for many user markets. The cloning non-deterministic state machine can be used as fundamental design when specifying, designing and verifying systems that are reconfigurable. Earlier work done at NTNU has produced a designs for the Virtex-4 FPGA, but the FPGA was too small for this application, only fitting one reconfigurable partition. The work done in this thesis is implementing and verifying the design on the Virtex-6 FPGA. The increased size of the Virtex-6 over the Virtex-4 has made it possible to construct the design with multiple reconfigurable partitions. Using the MicroBlaze processor and the PetaLinux Linux Kernel, the designed was run using sixteen partitions, actively managing non-deterministic state machines using partial reconfiguration.

SAMMENDRAG

Partiell rekonfigurering har mange praktiske applikasjoner, og er et viktig krav for mange brukermarkeder. En klonende ikke deterministisk tilstandsmaskin kan bli brukt som et fundamental design når det skal spesifiseres, designes og verifiseres systemer som er partielt re-konfigurerbare. Tidligere arbeid gjort på NTNU har produsert et design for Virtex-4 FPGAen, men denne var for liten for dette formåletall og passet bare en re-konfigurerbar partisjon. Arbeidet gjort i denne oppgaven er å implementere og verifisere designet på Virtex-6 FPGAen. Med mer plass på Virtex-6 enn på Virtex-4 har det vært mulig å lage et system med flere re-konfigurerbare partisjoner. Med bruk av MicroBlaze prosessoren og PetaLinux Linux Kernelen, var designet kjørt med seksten partisjoner, og aktivt behandlet de ikke deterministiske tilstandsmaskinene ved bruk av partiell rekonfigurering.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 PROBLEM DESCRIPTION

The use of partial reconfiguration for specifying non-deterministic state machines was proposed and simulated by Kjetil Svarstad and Kjetil Volden in their paper "Replicating Non-Deterministic Finite State Machines as a Mechanism for Run Time Reconfiguration on FPGAs" (Svarstad & Volden, 2011). Based on the SystemC simulator from the thesis "Compiling Regular Expressions into Non-Deterministic State Machines for Simulation in SystemC" by Volden (Volden, 2011), they proposed using runtime reconfiguration as a means of solving the self-replicating nature of the non-deterministic state machine.

An FPGA implementation of the design proposed in their paper has been in development at NTNU, being the topic of study for the theses done by Daniel Blomkvist (Blomkvist, 2013) and Tormod Heimark (Heimark, 2015). They each developed parts of the system for the Virtex-4 FPGA, but were unable to complete it, as the Virtex-4 did not have enough memory to support more than one reconfigurable module. The purpose of this thesis is to continue the work done by Blomkvist and Heimark, and implement the design on the Virtex-6 FPGA and realize the design Volden and Svarstad proposed.

1.2 MOTIVATION

Following Moore's law, the decreasing size of transistors has made chip area increasingly cost efficient. As a response, the usage of hardware accelerators has become more prevalent in newer hardware designs. Running a routine on hardware versus software is many cases more time efficient, performing computations every clock cycle, while a software implementation requires several instructions per operation. With the excessive space on hardware, even marginal speedups can be beneficial to implement.

By adding partially reconfigurable hardware, like an FPGA, to the system architecture, the designer is able to implement hardware accelerators on demand. As the logic written to the FPGA during reconfiguration lies in memory, designs can be modified and improved and be loaded to devices post production.

The usage of regular expressions for high speed pattern matching has given rise to a new type of hardware accelerators. Regular expressions are search patterns used to process strings, and is commonly used in a wide range of fields. The non-deterministic state machines are the realization of regular expressions, being equivalent in what patterns they recognize, and can be constructed using algorithms. When performance is critical, software solutions do not provide sufficient implementations. Using hardware solution will improve the performance of regular expression pattern matching, and combined with runtime reconfiguration provide the flexibility regular expression implementations require.

1.3 TASK

The assignment for this is thesis is the development of a system for running non-deterministic using partial Reconfiguration on the Xilinx ML-605 Board.

The previously NTNU developed tool chain for performing dynamic/partial reconfiguration was built for a Suzaku board with the Virtex-4 FPGA. This implementation has been the foundation for this work. Prior work to this thesis included building a new development toolchain for the Virtex-6, the old implementations have been decrepitated. The previous work ended with a toolchain capable development of user modules for the MicroBlaze processor architecture. Defining reconfigurable modules for the system and building the Linux Kernel with kernel modules and applications.

The next stage of the design is to implement a complete system with reconfigurable modules, a control framework and a software bundle for hardware control. When developed, a larger example should be tested. The task can be broken down to the following components:

- Design a control framework with reconfigurable modules capable of running several non-deterministic state machines.
- The runtime reconfiguration should allocate state machines dynamically, removing unused machines and copying when needed, as opposed to static where several machines are pre-allocated to hardware.
- Build a Linux Kernel with application for user interface, and with drivers to communicate with required hardware.
- The working example should be compared to existing and alternative solutions, both with regard to performance and feasibility for integration to other systems.

1.4 REPORT STRUCTURE

The next three chapters will first present some background to the tools and design ideas utilized in this design. Following this is a chapter on previous work done in the field and some considerations taken before the design process can take place. Next, an overview of the system before diving into the details of the design. From here the report is structured to follow the design solution in a bottom up fashion, following the implementation from the smallest component first. After the complete design and design process is argued, the functional system is verified through the user interface, and some performance evaluations are made. Last is the discussion section, divided into two sections, one discussing alternative implementations and the second discussing future work that can be done to improve the design or adapt it

for other usage. The thesis is concluded in the last chapter, with an appendix attached for additional details.

2 BACKGROUND

This chapter will present some of the background material required for understanding the work done in this thesis. The first topic covered is what Field Programmable Gate Array are and what they are used for. The Virtex-6 FPGA, which is the platform used for this project, will be the subject of this study. Second, an introduction of runtime reconfiguration and how designs benefit from it, concluded by an overview of what finite automata are and what applications they have.

2.1 FIELD PROGRAMMABLE GATE ARRAY

Field Programmable Gate Arrays (FPGAs) are integrated circuits designed to be reprogrammable after production to perform specific tasks. The circuit consists of a matrix of Configurable Logic Blocks (CLBs) and reconfigurable interconnects, which allow the logic blocks to be connect in different configurations.

FPGAs differ from Application Specific Integrated Circuits (ASICs) in that they are flexible in their design, as opposed to manufactured for specific tasks. The advantages of using an ASIC is smaller form factor and unit cost, as an FPGA must be larger to allow different configuration. Advantages of an FPGA over an ASIC is faster time-to-market, due to automated layout tools and manufacturing steps, simpler design cycle, and flexible designs (Xilinx, DS150). FPGAs are also a suitable platform for ASICs prototyping.

The behavior of the FPGA is described using hardware description languages (HDLs), of which the two most common are VHDL and Verilog. Using an HDL, the designer is able to describe the circuit accurately at a high level, and then use automated tools supplied by the FPGA manufacturer to test and verify the circuit. When the design is complete, a BIT file is created and used to configure the hardware. To configure the FPGA, the BIT file is written to the configuration memory using Joint Test Action Group (JTAG), which is a communication standard for debugging embedded systems.

The difference between VHDL and computer languages like C and assembly is that it describes a concurrent system, as opposed to a sequential system. Computer programs are translated into instructions to be read on a processor, while HDL code is synthesized to become combinatorial logic and memory elements.

2.2 ML-605 DEVELOPMENT PLATFORM

The ML-605 development platform is equipped with a Xilinx's Virtex-6 FPGA. The Virtex-6 is an FPGA manufactured by Xilinx built on 40nm process technology. It is marketed for high-performance logic designers, DPS designers and embedded system designers. The board is equipped with many features, being intended as a standalone prototyping board. Some of the features are 512 MB of RAM for external memory, a 200 MHz oscillator as a clock source, USB JTAG for configuration and debugging, and several communication ports (Xilinx, UG534). The board is shown in Figure 1.

The specific model of the Virtex-6 FPGA on this board is the "XC6VLX240T". The "XC6VLX240T" contains a total of 37,680 slices. Each CLB element of the Virtex-6 FPGA contains a pair of slices, and each slice contains four logic-function generators, also known as look up tables (LUTs) and eight storage elements, in this case flip-flops. The LUTs are designed to solve Boolean functions using arrays with the result of the function stored as opposed to calculated.

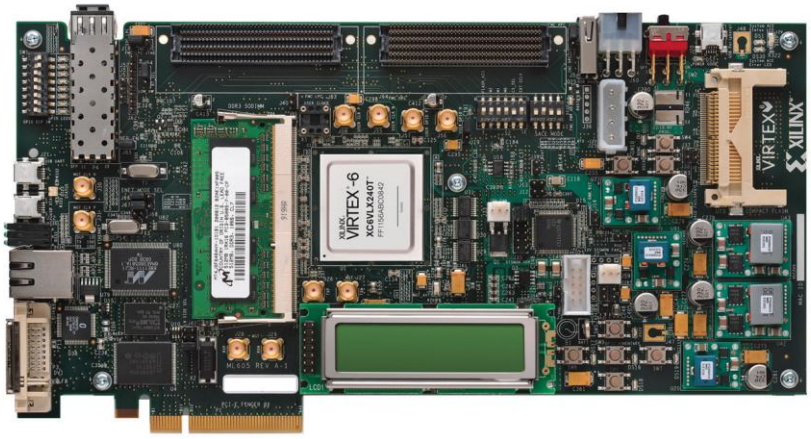

Figure 1 ML-605 Development Platform

The Virtex-6 family supports the MicroBlaze 32bit RISC soft-core processor. Being a soft-core processor means the implementation resides entirely in the general-purpose memory and logic of the FPGA. The MicroBlaze is designed to be configurable, giving the user the option to customize it for their purposes, and features a wide array of peripherals.

For the Virtex-6, the latest design software available is the ISE Design Suit, version 14.7. But as of October 2013, the tools are no longer updated, as Xilinx have transitioned to their newer software, the Vivado Design Suite. While the older design suite is usable, it does feature some bugs that will never be fixed, and does not support newer Windows operating systems.

2.3 PARTIAL RECONFIGURATION

Runtime Reconfiguration allows for even greater flexibility of the FPGA design, by allowing the designer to changes parts of the design during runtime. Adding a Hardware Internal Configuration Access Port (HWICAP) to the design allows the embedded microprocessor to read and write the configuration memory. Using partial BIT files, specific memory slots can be rewritten to change the functionality during operation of the circuit.

The greatest advantage for Partial Reconfiguration is the adaptability of the design. A typical premise where partial reconfiguration is advantageous, is a set of communications ports that supports different interface protocols. While each port support multiple interface protocols, the system is unable to predict what protocol will be used.

As opposed to having all possible interface protocols implemented on FPGA, each port has a reconfigurable partition assigned. By making each port interface a module, and storing them in memory, only the required module is written to hardware. This will reduce the amount of space needed on the FPGA, and thereby reduce cost and power consumption.

2.4 FINITE AUTOMATA

A finite automaton is a machine consisting of states and a transition function. For every input and time step, the automaton is in one of its states, as described by the transition function. For a given sequence of inputs, the automaton transitions for each of them, one by one, until it reaches the end of the input.

Automata, or state machines, are widely used in both software and hardware design. The automata serve two important functions. The first is language detection. For a certain input sequence, a word, the machine might either accept or reject the string. The words accepted are part of the automatons recognized language. The second function is procedural, triggering a correct sequence of outputs given the inputs, often used for control applications and communications.

A finite automaton is represented by 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Q is the finite set of states. Σ is the finite set of input symbols. δ is the transition function. q_0 is the start state. F is the set of accept states. Depending on the transition function the automaton can either be deterministic or non-deterministic. When the transition function uniquely defines a transition for all inputs, it is a deterministic finite automaton (DFA). If not, and an input may result in several transition, the automata is a non-deterministic finite automata (NFA).

While non-deterministic finite automata (NFAs) appear more flexible, they have the same computational power as the deterministic finite automata (DFAs). This is due to the fact that every NFA has an equivalent DFA construction. The transition between the two is done using the powerset construction.

While the two recognize the same language, for an **n**-state NFA with alphabet **Σ**, the DFA might have up to $\mathbf{\Sigma}^{\mathbf{n}}$ states (Rabin & Scott, 1959).

Regular expressions are search patterns used for string matching and are widely used. A typical example for the usage of string matching is the proofing tool in a text editor, matching the text to known words, and reporting an error when a word is rejected. For every regular expression, a NFA can be constructed that recognizes the same language. This is done using the Thompson algorithm. This relation makes the state machine a solution to solving a regular expression.

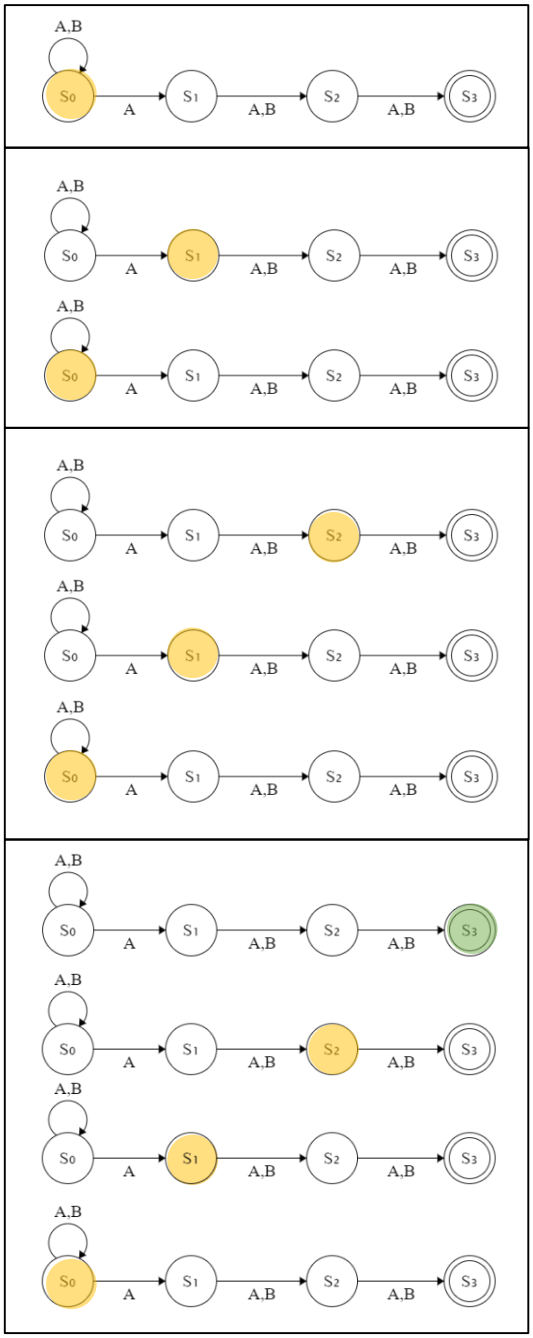

Figure 2 Running NFA with input "AAA"

Here, in **Error! Reference source not found.**, a simple example NFA is run with the input sequence "AAA". When the first character is read, "A", there are two transitions, one to state S0 and one to S1. The machine now exists in two states simultaneously, and is visualized by copying the state machine and running two machines in parallel. Upon reading the next character the machine in S1 transitions normally to S2, while the machine in S0 is copied resulting in three machines. Upon reading the last input character the initial machine reaches the accept state and the string is accepted.

While this representation is not the smartest way of running an NFA, it requires the usage of several machine and is a good application for runtime reconfiguration. When traversing NFAs in this paper, this approach is used.

3 DESIGN GOALS AND CONSIDERATIONS

This chapter will first present some of the earlier work done related to NFAs and runtime reconfiguration, in particular their usage with regular expressions, before evaluate the design goals of the system and discuss some design considerations.

3.1 EARLIER WORK

In the paper (Svarstad & Volden, 2011), the authors broke down the issue of running NFA on hardware to two components, evaluating the set of NFAs and managing NFA replication. While the first challenge is well understood and performed in hardware implementations, they propose runtime reconfiguration as a solution for the latter. They proposed an FPGA implementation that requires a framework to allow any state machine to copy itself to some other position in hardware, using available FPGA resources to execute and present all possible execution paths. While not expecting the implementation to be practical for many applications, they propose usefulness for some streaming applications and monitoring systems.

The master thesis by Tormod Heimark (Heimark, 2015), is the continuation of an ongoing design at NTNU to implement a system for running non-deterministic state machine utilizing partial reconfiguration, based on the paper by Svarstad and Volden. As opposed to using only the runtime reconfiguration to change the functionality, the system was intended to use the reconfiguration to dynamically assign clones of the running NFA. The design was implemented for a Virtex-4 FPGA, but had insufficient space to have both the HWICAP module and more than one NFA running. The resulting design was able to run the NFA statically using four automata, and partial reconfiguration was successfully performed for the one available slot, but the system was unable to perform the intended design.

Regular expressions are demanding for software implementations and do not scale well as the number of regular expressions increase. Due to the importance of regular expressions, many resources have been invested in designing solution to improve their performance (Cheng, Shuhui, Jinshu, S.M., & C.K., 2016). While the earliest work focused on software implementations, with the advances in hardware technology, solutions running NFAs on hardware have become common.

The paper “Regular Expression Matching in Reconfigurable Hardware” (Sourdis, Bisop, Cardoso, & Vassiliadis, 2008) implemented an efficient hardware implementation for matching regular expressions. Using a tool for generating synthesizable VHDL code from regular expressions, they evaluated their work to network Intrusion Detection Systems (IDS). IDSs use regular expressions to detect dangerous payload

contents, and require high speed implementations to match the speed of network traffic. Their design achieved a throughput of 2.3-3-2 Gbps on the Virtex-4 processor.

In their paper “Fast Regular Expression Matching using FPGAs” (Siduh & Prasanna, 2013), the authors used non-deterministic automata on FPGAs to achieve a processing speed of one text character per clock cycle. Comparing their solution to software approaches like grep, they argue the construction from NFA to DFA is computationally slow, and only having to construct the NFA from the regular expression before running it is many fold more efficient. Their implementation on hardware utilized flip flops as one hot coding for their design, each bit representing the current state of the NFA. For every input the corresponding bits would change to indicate what states where currently active. Next they created an FPGA mapping algorithm for placing the NFA on a self-reconfiguring gate array (Sidhu, Wadhwa, Mei, & Prasanna, 2000). The result of their labor was a system capable of constructing and running NFAs of size **n** with text of length **m** in **O(n + m)** time, and **O(n^2)** space, as opposed to the fastest DFA implementation requiring $\mathbf{O(2^n + m)}$ time and $\mathbf{O(2^n)}$ space.

3.2 DESIGN CONSIDERATIONS

There are several ways to run non-deterministic automata. Due to their relation with deterministic automata, any NFA can be constructed as a DFA using the powerset construction, with a maximum of $\mathbf{\Sigma^n}$ states for an NFA with **n** states and alphabet **Σ**. While having both the opportunity to run either the NFA or the DFA, the developer also has the choice of running them on hardware or software.

Comparing the implementation of NFAs to DFAs on hardware, the DFA would be implemented as a single large automaton, while the NFA as several potentially smaller machines. Since the difference in size between the two vary, we could potentially have DFA and NFA of the same size, making the multiple NFAs implementation impractical. While the DFA once constructed has no limitations for input strings, the NFA implementation is limited to the number of partitions that are available to the design. The more states an NFA has, the larger the DFA can potentially be. Optimization techniques do exist for hardware implementations of both NFAs and DFAs, some of them covered in the previous section.

Comparing the speed of the two in hardware, both the DFA and the NFA will be runnable in constant time, reading one symbol at each clock cycle. There are several design strategies when implementing finite automata on hardware, and while the DFA implementation is often straightforward, the NFA design requires smarter design strategies. In the case the DFA construction is too complex to be practically implemented, steps can be taken to either reduce it or make hybrid implementations with less non-determinism.

Comparing the implementations of NFAs and DFAs on software to their hardware implementations, performance is the biggest difference. While a hardware implementation is capable of reading an input every clock cycle, the software implementation needs several instructions to perform the same task. This disparity has given rise to hardware accelerators. By trading hardware size for computational speed, tasks can be performed faster. Combining this with partial reconfiguration, the hardware size requirements can be minimized.

In the software, each machine is treated as an object, and upon reading the input, each machine is transitioned individually until all machines have made their transition. While the DFAs requires large space for the transition function but little memory bandwidth, the NFAs requires large memory bandwidth as any number of state machine object might be created, exhausting the available resources and causing performance degradation.

3.3 Design Goal

The goal for this thesis is to develop a system able of running non-deterministic finite automata in conjunction with partial reconfiguration, and compare the solution to other existing implementations. While reconfigurable hardware has been used with finite state machines earlier, none have succeeded in using partial reconfiguration to actively manage copies of the non-deterministic automata.

Compared to other solution that already exist, this design is not expected to excel at any benchmark. Any performance gained by the hardware implementation is lost in the multiple partial reconfigurations that have to be performed. With these considerations in mind, the design can serve as a reference design for potential alternative solutions to problems that could benefit from active use of runtime reconfiguration.

3.4 System Specifications

- The partial reconfiguration should be used to assign automatons to reconfigurable partitions when a copy is requested, and should remove the automaton when it is rejected. Upon completing the input sequence, the accepted states are stored by the framework and relayed back to the user.
- The framework must be compatible with different NFA designs in order to accommodate different applications. The NFA design should accommodate simple regular expressions.
- The state machines should be controlled using a combination of hardware and software functionality, optimizing for ease of use and flexibility.
- The system should be constructed with an operating system that can manage resources, like hardware components and memory. The framework should be usable through the operating system by applications.

4 SYSTEM OVERVIEW

A brief system overview is given here before the implementation is discussed.

The completed system consists of three major components, the MicroBlaze system running the Linux Kernel, the framework for controlling the non-deterministic state machines and the reconfigurable partitions the NFAs can occupy. The system layout is shown in Figure 3 with four reconfigurable partitions.

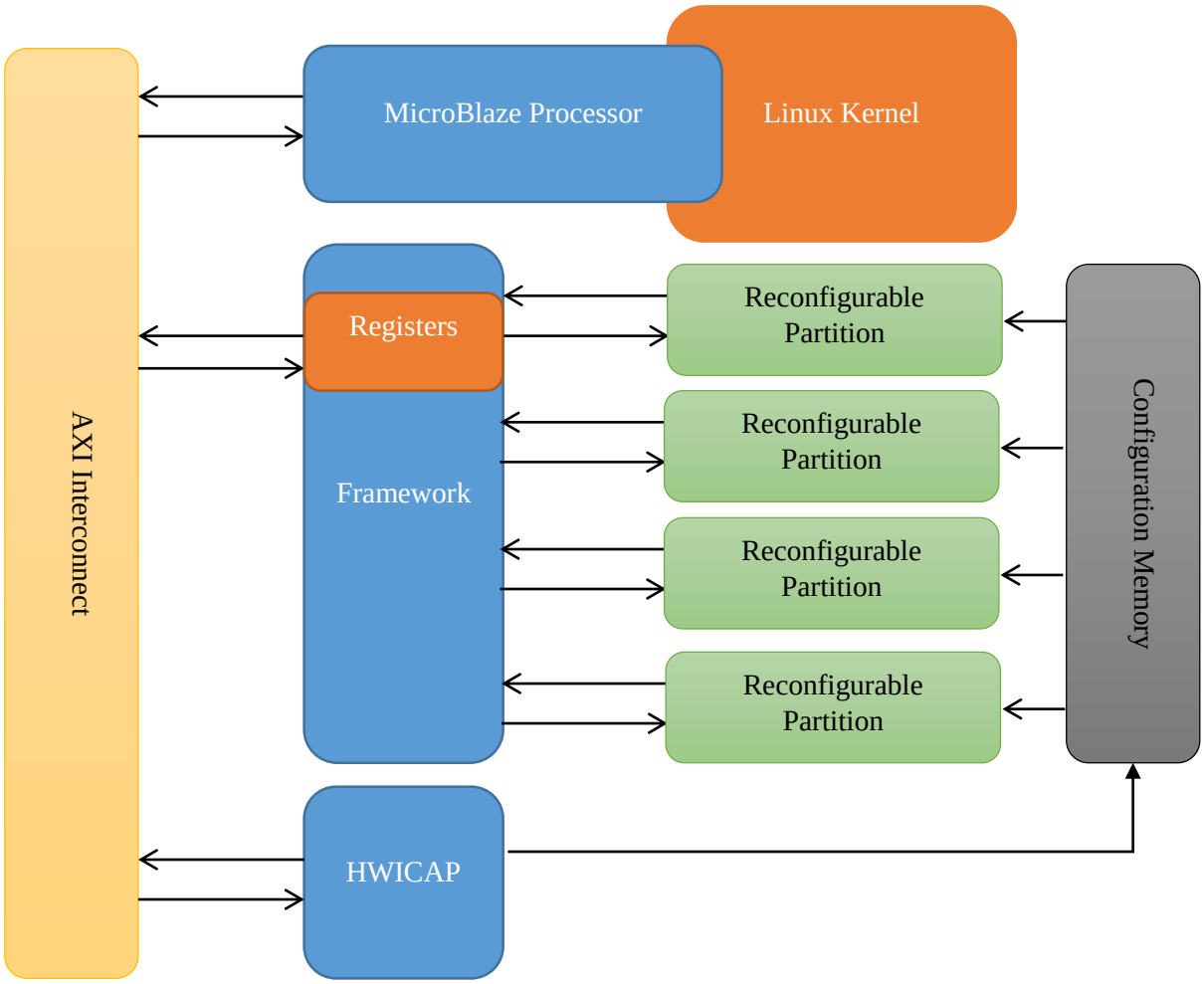

Figure 3 System Overview

The MicroBlaze is the hearth of the design, being the embedded processor, responsible for the execution of all software instructions. The Linux Kernel on top is responsible for managing the resources of the processor and acts as a layer between the user and the kernel. The user interfaces with the system through

an application. From this application, calls are made through the driver to the appropriate devices, protecting the user from illegal operations that might jeopardize the system.

The interconnections between the processor and peripherals allow for software access to the framework and HWICAP. Over the AXI interface, the software is able to read and write to registers on the framework, and write to the configuration memory through the HWICAP device.

In this overview the framework is connected to four reconfigurable partitions, but the design is scalable to connect to as many as would be required, and the final implementation features sixteen modules. The framework has no control over the HWICAP, and only the software running on the Linux Kernel can change their configuration.

5 NFA DESIGN AND IMPLEMENTATION

While the complete system should be capable of running a wide array of NFA, an analysis of the NFA and its components is crucial to designing the framework. This chapter will examine some of the common elements for NFAs and how they are implemented to hardware.

5.1 DESIGN CONSIDERATIONS

As noted earlier, any finite automaton is represented by a finite set of states, a finite set of input symbols, the transition function and a set of accept states. For non-deterministic finite automata, each transition can result in one or more states. While the design of the DFA requires no other considerations, designing the NFA requires some way of handling the addition simultaneous states. And as there is no theoretical limit to how many states can exist at any given time, restrictions have to be enforced.

When a machines transitions to more than one state, in order for the framework to properly instantiate a cloned machine, a signal buss with information as to what state the copied machine should resume from is required. Depending on how many simultaneous transitions there were, several busses would be required. On the other end of this, the machine the information is sent to needs to know that it should switch states and what state it should switch to.

Upon either accepting or rejecting the input, the machine must return that it has accepted or reject. Depending on the amount of accept states in the design, the buss width must be extended. When rejecting the input string, all outputs from the automaton can be ignored.

The framework is unable to predict what the needs every NFA might have, so some restrictions need to be set before NFA design can take place. The first restriction set is the number of transitions from any single state. The second restriction is the number of states available for the state machine. The third restriction is the number of accept states in any automaton, and the last restriction is the size of the alphabet.

5.2 VHDL IMPLEMENTATION

In order to streamline NFA constriction, a reference design file is created for the NFA. In the design file, all the connections to the framework are established. When creating a NFA with the design file, only the states, with their transitions and outputs is needed.

The following restrictions are implemented in the design file. The amount of states is limited to an array of 4 bits, giving the machine a total number of 16 states, which is sufficient for basic designs and testing. The states are not optimized, but implementing one-hot coding is possible with modification.

The design is limited to three accept states with two accept bits on the output. The alphabet for this design contains three letters and the NULL character, for a total of two bits. The NFA is only allowed to request two copies, limiting each transitions to three new states. These restrictions are unchangeable after implementation, and require some modification to both the framework and the NFA design if changed.

As the design only has 4 bits of available states and can only request two copies at any given time, two control signals are added and two 4 bit wide outputs, that contain the binary representation of the state. A matching input is added to each machine, which will be connected to the output of the machine that requested the new machine.

The complete module is show in Figure 4, and the output is defined as show in Table 1. The design features two force out busses, returning state information to the framework, and one force in, receiving information form the framework. When the run bit is set, the machines are allowed to progress to the next state.

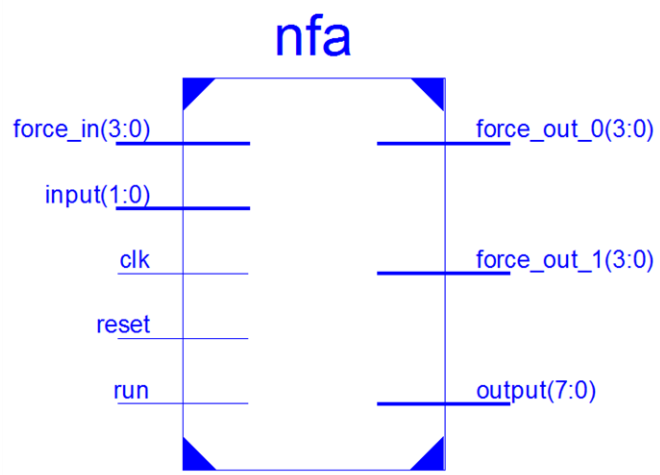

Figure 4 NFA structure

Output - Control Signals								
Bit	7	6	5	4	3	2	1	0
Signal	N/A	N/A	N/A	Accept_1	Accept_0	Copy_1	Copy_0	Delete

Table 1 Output from NFA containing control signals

5.3 NFA Synthesis

As the NFA modules are implemented as a reconfigurable module, some considerations need to be made. Each partition must contain a super set of all the pins that are to be used by the different modules. When a black box module is used, all outputs from the partition are set to logic 1, while unused ports for other modules are either logic 1 or logic 0. Unless ignored, all modules should connect to all ports to set them to correct default values.

The boundary between the static and dynamic parts of the design are called partition pins. These pins can't be bidirectional, so IOBUFs are not allowed. This is resolved by disabling -IOBUF in the Xilinx synthesis settings.

After synthesis, the NFA module requires 16 LUTs and 4 Registers, translating to 4 Slices, and a total of 2 CLBs.

5.4 Implemented NFA

Using the reference design file, the NFA shown in Figure 5 was designed. The design utilizes partial reconfiguration frequently, having several transitions with multiple outcomes. The design also uses the maximum number of accept states and has one transition that requires two copies, S0 with input B transitions to S0, S4 and S6.

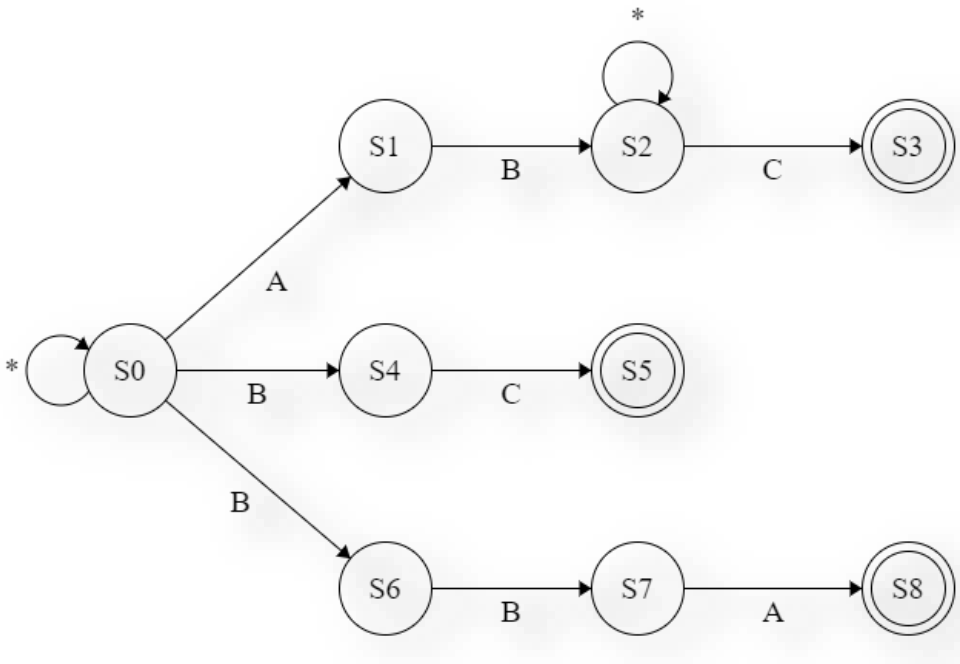

Figure 5 NFA implemented in design

The design recognizes the same language as the three regular expressions “AB.*C”, “BC” and “BBA”. While the first one looks for any string having the sequence of AB followed by the last character of the string being a C, the two second look for the any string ending in either BC or BBA.

This NFA design is meant to showcase a situation where the NFA has several simultaneous machines running. A sample run for this state machine enfolds like shown below. We see that for this string of eight character, we need eight state machines.

Input	A	B	A	B	A	B	A	C
NFA 0	S0	S0	S0	S0	S0	S0	S0	S0
NFA 1	S1	S2	S2	S2	S2	S2	S2	S2
NFA 2		S4	S1	S2	S2	S2	S2	S2
NFA 3		S6		S4	S1	S2	S2	S2
NFA 4				S6		S4	S1	**S3**
NFA 5						S6		**S3**
NFA 7								**S3**

Table 2 First Sample run of implemented design

6 FRAMEWORK DESIGN AND IMPLEMENTATION

This chapter will present the design of the framework for running the non-deterministic finite automata. Taking basis in what was presented in the previous chapter about NFAs, a framework is designed. The first part of the process is recognizing the functionality required and a partitioning between hardware and software. The hardware implementations are discussed here, while the software implementation later in the paper.

6.1 DESIGN STRATEGY AND PARTITIONING

The primary purpose of the framework is to act as control logic for the running non-deterministic state machines. As touched on in the previous chapter, running NFA requires procedures to handle how states are copied and how the new state is initialized. In addition to control logic, the framework acts as an interface between software and hardware.

Having access to both software and hardware control over the design opens the doors up to many possible design implementations. Having identified what tasks need to be performed, the design needs to be partitioned over hardware and software. The partitioning should improve flexibility and performance. While hardware implementations are often faster, software implementations have the possibility of handling complex functionality, not being limited by hardware design methods.

6.1.1 Allocation

Looking at the NFA from the previous chapter, when an NFA reaches a transition that requires the machine to be copied, the framework should be able to instantiate a new NFA in an unused partition. When the copy is requested, a control signal is registered and the address of the next state for the copied machine is received. The framework then relays this information to an unused slot, which the framework is responsible for allocating. This process is the first function required by the framework.

While the simplest method is to hardcode the allocation and use the closest partition, this is impractical to implement when each machine can have multiple copies and enforces restriction to the design space. A different implementation is assigning each partition to a first inn first out (FIFO) que that is read whenever a clone requested. When a delete is registered, that partition returns to the bottom of the FIFO. The stratagem is flexible, but requires multiple clock cycles for allocation, as to avoid simultaneous reads of the FIFO.

While this is a simple solution to implement on hardware, a software implementation is possible. While computing the FIFO would require few instructions, relaying the information back to the hardware would

require several read and writes of the software accessible registers, while the hardware can read from and write to an NFA module at every clock cycle.

6.1.2 Runtime Reconfiguration

After having allocated a partition for the new module to be copied to, runtime reconfiguration needs to be performed. While possible to implement on hardware, several requirements need to be filled. First, the framework needs access to the local memory where the partial BIT files are stored. Second, a procedure for writing the BIT files to the HWICAP must be designed. While the optimal solution, this is well beyond the scope of this project. Instead, a software solution is approached. Upon finishing the allocation, a binary string with the required partitions is returned to the software, which then performs the runtime reconfiguration.

During reconfiguration, the static region should be decoupled from the reconfigurable partition. This is due to the partition being active during the reconfiguration, and unexpected behavior may occur. The only method for ensuring the configuration is complete is having the software report back to the framework upon successful reconfiguration.

Two methods for solving this were approached, each with their own benefits. The first is utilizing a bit from the NFA design that was checked for a high or low signal. Since empty partitions always have high outputs, when a low is detected a module fills that slot.The framework then knows when and where the new machines are written to hardware, but unable to resume operations before the control bit is low. The control bit is set high when reconfiguration is performed and returned to low when completed.

The second, and implemented, solution is giving the software full control over the available modules, and relaying this information back to the hardware when reconfiguration is completed. While the first variation has the framework being more independent, and easier to debug by writing the available machines back to software, it is difficult to verify in simulations. The second approach also allows machines to be prewritten to hardware, for performance comparisons between the two.

6.1.3 NFA Control Process

During allocation and reconfigurations, the other automata cannot be run. Due to the nature of state machines, they can only process one input at a time, and the NFAs need to process the same input at the same time. In order to achieve this in conjunction with the other tasks performed, some procedurals need to be followed. The most common implementation for procedural design is a deterministic state machine. Combining the state machine with counters, the correct control signals can be set in the correct order and for the correct amount of time. The control logic is impossible to implement in software, as the procedurals are clock sensitive. Even a single tick of delay can cause unexpected behavior.

6.2 VHDL IMPLEMENTATION

6.2.1 Memory Process

To allow the framework to read input from user, a software accessible register and one control bit is reserved for this usage. When the control bit is set in software, the register is stored in an equally large memory register. The framework is then ready for processing the string, and no more writes are performed until the string is finished processing.

The memory is read sequentially using a pointer that increments on each read cycle. When the pointer reaches the end or an end of string character, it loops and the complete bit is set. When the complete bit is set the process will accept a new string to be written to memory. As the alphabet of this design is two bits wide, two and two bits are read for a maximum 16 characters per string.

This implementation has limitations in practicality, as only one string can be written to hardware at a time. While sufficient in some cases, other applications require the write to be performed in streams, reading a character at some frequency, as over a communications interface. The other limitation size of the memory. When the alphabet is increased to 8 bits, only 4 character can be stored, limiting the practical application of the design. A smarter design should be implemented here for future design.

6.2.2 NFA Control Process

The NFA control process is responsible for keeping the NFAs running synchronized and controlling the other processes in the design. This is done using a three state deterministic state machine. The three states, HOLD, RUN and ALLOC, have transitions in correspondence with the running of the system. This control process is responsible for the five control signals shown in Table 3 and the DFA used is shown in Figure 6.

read_en	Control signal for reading the memory
sm_run	Control signal for running attached NFA
reset_en	Control signal for resetting newly reconfigured NFA
request	Control signal for requesting new partitions from software
allocate	Control signal for initiating allocation process
counter_0	Procedural counter for allocation process
counter_1	Procedural counter for allocation process

Table 3 Control signals form NFA control process

Upon entering the HOLD state, the request bit is set high, signaling that the framework is waiting for resources. When the resources are available, and a string has been loaded to memory, the machine

transitions to the RUN state, which initiates the reading of memory and running of the state machines. When a copy or delete is requested form one of the machines, all operations are halted and the framework transitions to the ALLOC state. When allocation is complete, the machines transition to their new state and the automaton returns to the HOLD state. While in the HOLD state, all machines written to hardware are reset to the state they should resume from. When the complete bit is detected, the machine immediately reverts to the HOLD state and most of the circuit is reinitialized, waiting for new user input and an NFA module to be assigned to the first partition.

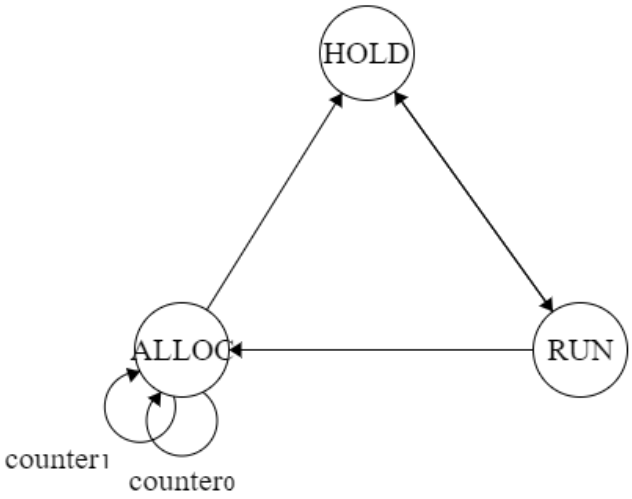

Figure 6 Control Process DFA

6.2.3 Allocation Process

While in the ALLOC state, the allocation process is initiated and progresses using the counters in the control process state machine. For each tick of the counter the NFAs are iterated over, and the deletes and first copy for each machine is processed. After traversing the NFAs once, the process is redone, this time checking the second copy signal from the machines.

While not the fastest design, it is easy to implement and causes no timing issues. The allocation process will always take the number of partitions times the number of copies a single machine can issue. This is 32 clock cycles for this reference design. While not a substantial bottleneck compared the reconfiguration, it is a substantial increase from other design techniques.

After reconfiguration is perform, a local reset must be issued to the partition, in order to put it in a known state. As part of the allocation process, the newly copied machines receive both a reset signal and the address of their new state. The reset signal is configured to put the machine to the state received, as opposed to a default state.

6.2.4 Using an AXI interface

To add the framework to the MicroBlaze processor, we need to interface it to the AXI-bus, which is used by the newer Xilinx systems. An interface can be generated using the Xilinx Platform Studio, but for this design an interface file containing both software reset, four sets software accessible registers and an interrupt port is used[1]. Although neither the interrupt or the reset is used in this design, they are available for future development. The framework component is added to the "user_logic.vhdl" of the interface, assigning the clock and reset ports and binding four inputs and outputs to the software accessible registers.

6.2.5 Outputs and Inputs

The framework has four readable and four writeable registers, all with their own purpose, listed in Table 4.

Readable Register 0	0 -> Complete bit, 1 -> Request bit, others -> N/A
Readable Register 1	Accept Value
Readable Register 2	Active and Requested modules
Readable Register 3	N/A
Writable Register 0	0 -> Write enable bit
Writable Register 1	Value written to memory
Writable Register 2	N/A
Writable Register 3	N/A

Table 4 Framework Registers

6.3 VHDL TESTBENCH

In order to verify the designed framework, a VHDL testbench is designed to observe correct operation under different stimuli. The four inputs are used as the source of stimulus, and a clock source is set on the clock input. Writing a testbench is a good way for verifying correct operation before implementation. For this testbench, there are many cases that need to tested. In general, the timing of the design needs to be verified.

Being unable to simulate runtime reconfiguration, some steps needed to be taken to imitate a hardware implementation. Having most of the control for runtime reconfiguration in software, no changes were needed to the VHDL code to make the design compatible with the testbench. When a module requests a partition, the testbench will simply report where the machine exists, despite it having been there the entire time. Having tested several strings for correct operation, both short and long, timing is compared to what

[1] http://www.xilinx.com/support/answers/51138.html

was intended in the design process. The results from the testbench are reused later when the final implementation is verified. While correct operations can be observed in the simulation, both running on hardware and with runtime reconfiguration might cause behavior unexpected by the simulator

7 IMPLEMENTATION OF MICROBLAZE SYSTEM AND RECONFIGURABLE PARTITIONS

The design of a successful system relies on two components, the framework design and the embedded processor system, with peripherals and operating system. What follows is implementation of the MicroBlaze System and reconfigurable partitions. The steps are described in detail in the Partial Reconfiguration Tutorial by Xilinx (Xilinx UG744), with the differences described below. The MicroBlaze processor is synthesized through the Xilinx Platform Studio software.

7.1 XILINX PLATFORM STUDIO DESIGN

After designing and verifying the framework, the next step to implement the design is to create a processor system that the framework can connect to. The MicroBlaze system is the only processor system available to the Virtex-6 FPGA, and it is a softcore processor, meaning it is implemented using the logic primitives of the FPGA. It is configurable and has access to a wide array of peripherals, amongst them the required peripherals for running Linux and performing partial reconfiguration.

The processor is intended to be run with Linux, and the required components for running PetaLinux on the Microblaze can be found in the “Board Bringup Guide” (Xilinx UG980). From the documentation, the minimum components needed in order to successfully run the Linux kernel are:

- External Memory Controller with at least 32MB of memory.
- Dual channel timer with interrupt connected.
- UART with interrupt connected.
- MicroBlaze with MMU support.

The design is easiest built using the Base System Builder, a simple tool for starting a processor design. In the choice between AXI and PLB the newer interface standard, AXI, is chosen, as PLB is no longer supported in FPGAs newer than the Virtex6 and Spartan6. The processor speed is set to 100 MHz, and only the needed peripherals are added. Only the minimum of peripherals are added to avoid any complications with the Linux Kernel and potential drivers. Peripherals can be added to design at a later state, if needed. After finishing the Base System Builder, the system is generated. Next, the processor is configured for “Linux with MMU”, leaving all settings at default. The system is now ready for Linux.

In order to perform partial reconfiguration, a peripheral that is able to read and write the configuration memory of the FPGA is needed. This interface, for the Xilinx devices, is the Hardware Internal

Configuration Access Port (HWICAP). An instance of the “axi_hwicap” is added, version 2.2.3, making sure to check that “Instantiate STARTUP primitive in the HWICAP core” is enabled, and the FIFO read and write depth is set to 256 bytes, as the frame size for the Virtex6 is 162 bytes. These details can be found in the technical document for this peripheral (Xilinx DS817). The maximum frequency for the “ICAP_Clk” on the Virtex-6 is 100Mhz.

Lastly, the framework is added to design. To import it to XPS, the folder containing the peripheral is placed in the “pcores” folder of the design, and added from the list of available peripherals.

Each peripheral added is assigned a memory size and an address, which are used whenever the processor needs to accesses these peripherals. After adding all the peripherals, verify the design and make sure none of the peripheral addresses are overlapping. The complete system architecture is show in figure below.

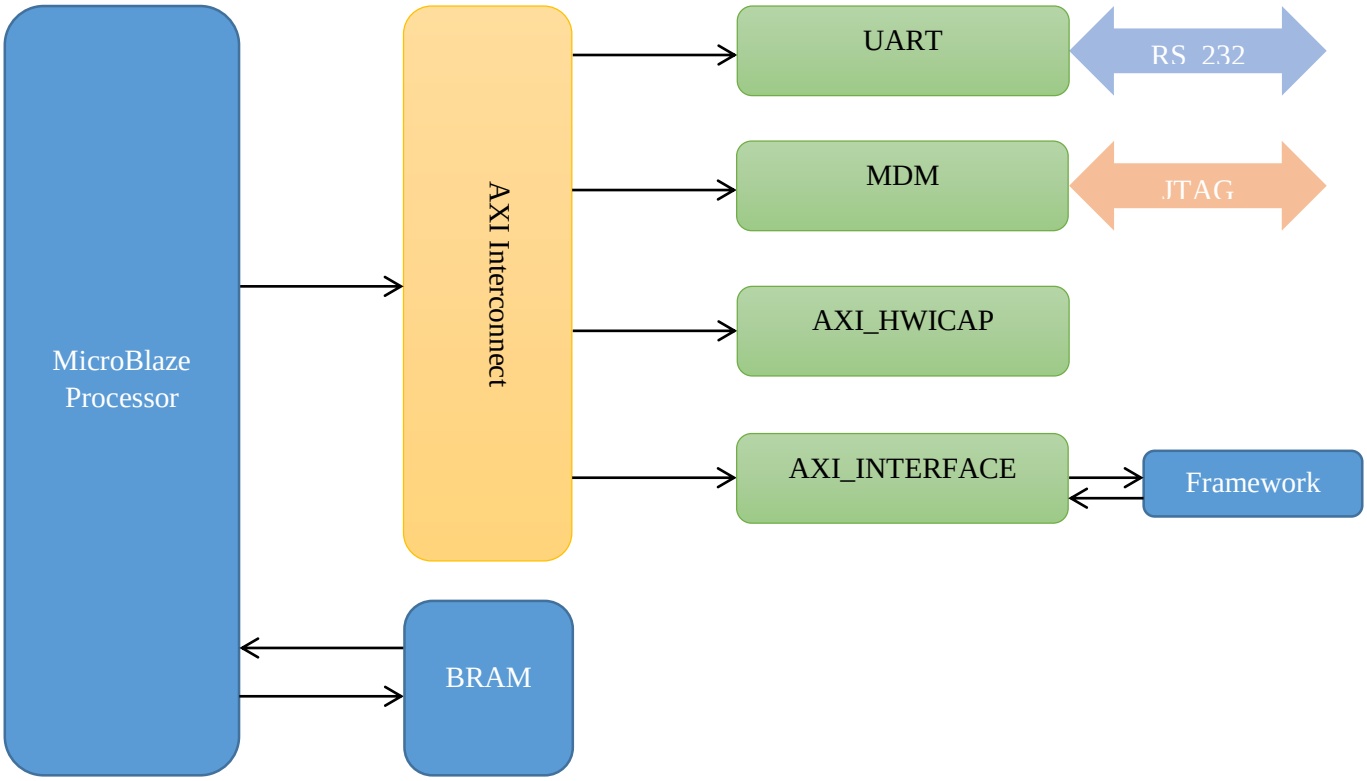

Figure 7 Complete MicroBlaze system

The last step in the XPS software is to Generate Netlist and exporting the design to SDK. When done, the system is generated as netlist files and constraint files in the implementation folder of the design, and additional constraints are placed in “system.ucf”, which is found in the data folder of the design. Additionally, a “system.xml” file, which is a description of the system, needed by the PetaLinux tools later, is generated.

7.2 PLANAHEAD IMPLEMENTATION

In order to create reconfigurable partitions, the system is imported to PlanAhead. This is the placement planner for the FPGA and the tool needed to define reconfigurable modules. A Post-synthesis project with Partial Reconfiguration is generated and the netlist files and constraint files found in the "implementation" folder of the XPS project folder, as well as the constraint "system.ucf" file fond in the "data" folder, are imported.

Note, with the Xilinx ISE tools not being updated, two bugs exist in the software. The first is a crash when reconfigurable partitions are added to the design. The simple workaround is to disable all constraints except "system.ucf", which needs to be set as target constraint file, while modules are added. After partitions have been added, the constraints can be re-enabled. The second bug happens during design runs. This is a common bug in PlanAhead when designing with partial reconfiguration. The workaround is writing "set_param funnel.forceHierarchy FALSE" in the TCL prompt before initiating runs.

In order to generate the wanted bitfiles, the reconfigurable regions need to be implemented with modules. Each NFA partition has to be defined as both black box module and as an NFA module. The synthesized NFA modules need to be imported before adding them. Next a region for each partition is defined on the FPGA that is large enough to fit the largest module used. From synthesis, the NFA module only occupies 2 CLBs.

There can only exist one reconfigurable partitions on each configuration frame of the FPGA. Every frame has a unique address, and the smallest BIT file that can be written is the size of a frame. The frames on the Virtex-6 are 40 CLBs high and 1 CLB wide. To optimize the BIT size, the partitions should occupy as few frames as possible.

The last option set before initiating the design is referencing to the "system.bmm" file using the -bm command in Translate options of the run. The file can be found in the "implementation" folder of the XPS project folder.

After the first design is run the results are promoted. When promoting a run, the implementation results are reused for other runs if they share implementations. The second run is performed and the two are verified using "Verify Configuration". If no errors are encountered the runs can then generate bitstreams.

For every bitstream generated, one BIT file containing the entire system with the modules specified, as well as the differential bitfiles that contain the reconfigurable module, one for each partition.

7.3 FINAL IMPLEMENTATION

The implemented design with NFAs occupying the partitions is shown in Figure 8. where the purple rectangles are the reconfigurable partitions. Looking at the netlist estimation, the design only utilizes 12% of available slices.

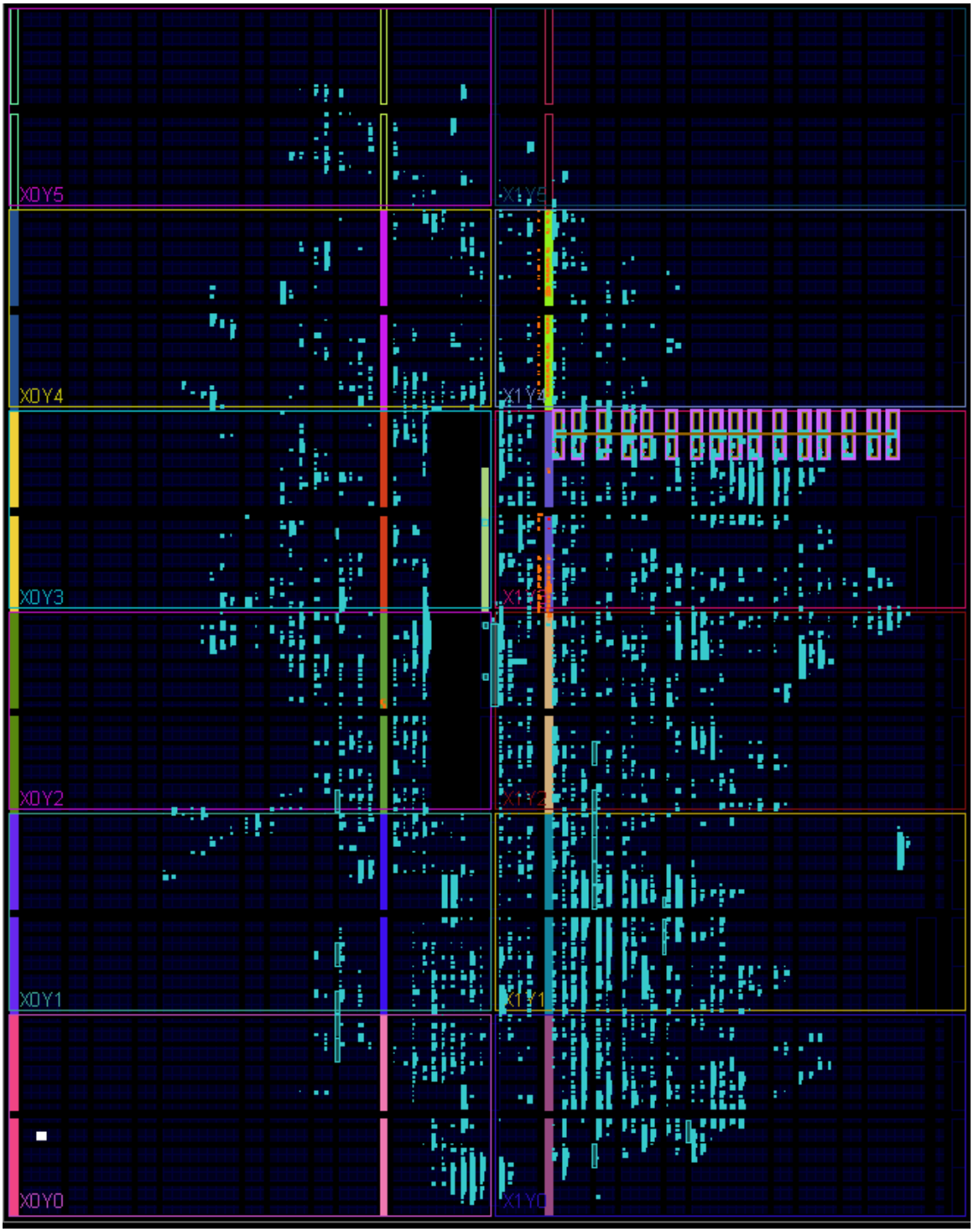

Figure 8 PlanAhead final implementation

8 IMPLEMENTATION OF LINUX KERNEL AND SOFTWARE COMPONENTS

There are two ways for us to interface with the MicroBlaze system. The first is using the Standalone Board Support Package, which is a set of software modules that can access the processor. The disadvantages of this approach is that it is not a persistent solution, only intended to run the application to the end. This limits the practical application of our design, but is a suitable for initial testing and debugging.

The second approach is to use an operating system, which gives us more design options and greater control of the system. There are several operating systems available, but Xilinx provides toolchains for building a Linux Kernel, called PetaLinux, for the MicroBlaze. This chapter will follow the implementation of the Linux Kernel and the user applications.

The Linux Kernel is an operating system kernel. The kernel is tasked with managing input and output requests from software, and translate them into instructions for the CPU and other hardware peripherals. The kernel is the layer between applications and the hardware, and can only be accessed using the functionality defined in the kernel modules.

8.1.1 PetaLinux

The PetaLinux tools supplied by Xilinx modify and compile a Linux Kernel image for the MicroBlaze architecture. The tools are only available for Linux computers, and installation and commands can be found in the PetaLinux Guide (Xilinx UG977). The tools also include a cross compiler for compiling C programs to the MicroBlaze.

There are several versions available of the PetaLinux tool, and the newer ones are recommended by Xilinx, as they have more functionality and bug fixes. Despite this, the newer versions were unable to generate a working image for this design. The last version to support the ML-605 board was the 2013.10 version, which is the one used for the development of this system.

The only needed file to generate the Linux Kernel for a system is the “system.xml” file generated by XPS. Using this file, a PetaLinux Board Support Package (BSP) can be generated using the Software Development Kit. After creating the BSP, the folder is imported to the PetaLinux project and information for both the kernel configuration and the device tree are extracted.

The process for compiling a Linux Kernel has many steps, but with tools like PetaLinux the process is simplified. The complexity of the build process is related to the flexibility of the Linux Kernel. Being

adaptable to many processor architectures and configurations, the Linux build process relies on the Device Tree to map onto a system. Every system can be uniquely defined by their device tree, and every node of the tree contains information for specific hardware components, like what parameters are needed to reference them and what driver should be used.

Before building the Linux image, PetaLinux has a myriad of settings for the kernel. These add further customizability and might have to be changed if errors are found or bad functionality is detected. While the PetaLinux tool set most of the setting when importing the BSP, one change needs to be made. Under Device Drivers -> Character Devices -> Xilinx HWICAP Support needs to be enabled.

8.1.2 HWICAP Driver

While the PetaLinux tools include all the required drivers, upon initial tests the HWICAP driver was unable to perform reconfigurations. It was found that the driver was missing some changes that had been made to the recent versions of the HWICAP hardware. Looking at the compatibility strings of the driver, it would seem the driver was intended for the OPB and XPS versions of the HWICAP, while the current design uses the AXI variation. While debugging the driver, the error was found to be caused when checking the status register of the HWICAP. Studying the technical document for the peripheral, it was found that while earlier iterations of the device had 8 bits in the status register, the newer versions only utilize three. When reading the status register, the drivers triggers a failure if the eight bit is found to be equal to zero, which is the only values it can take in the newer HWICAP design. This was changed to test the lower three bits for errors. In addition, the driver was changed to correctly bind to the axi-hwicap device and the default structure of the configuration registers was changed to Virtex-6.

8.1.3 NFA Framework Driver

The primary functionality required by the framework driver is to read and write the designated registers. There are two ways to get access to these registers on the device. The first, and simplest, is accessing the device through /dev/mem in Linux. This will directly access the given memory location. This is a quick and easy solution, but requires the address of the peripheral to be hardcoded in the software. It also offers no other functionality other than read and write, and offers no protection to the user. Reading or writing at a sensitive location might corrupt the kernel and cause crashes.

The second is using a device driver. The purpose of a device driver is to act as an interface between user space and kernel space in order to give the user access to the peripherals, and protect the user from modifying kernel space, the hierarchy shown in Figure 9. While designing a driver for a device might be beneficial, it is often time consuming, and overkill when only simple read and writes are performed. The

Linux Kernel therefore includes a generic driver called UIO. Intended only for read, writes and interrupt handles, it is well suited for simple devices.

In order to use the UIO Platform Device Driver, the compatible property of the peripheral in the Device Tree needs to be set to “generic-uio”. The hardware can now be accessed by the user space application using the commands “open()” and “mmap()”. In addition, user space application can be notified when interrupts occur by calling “select()” or “read()”, and while an interrupt signal is generated in the framework design, it is left unconnected until further development.

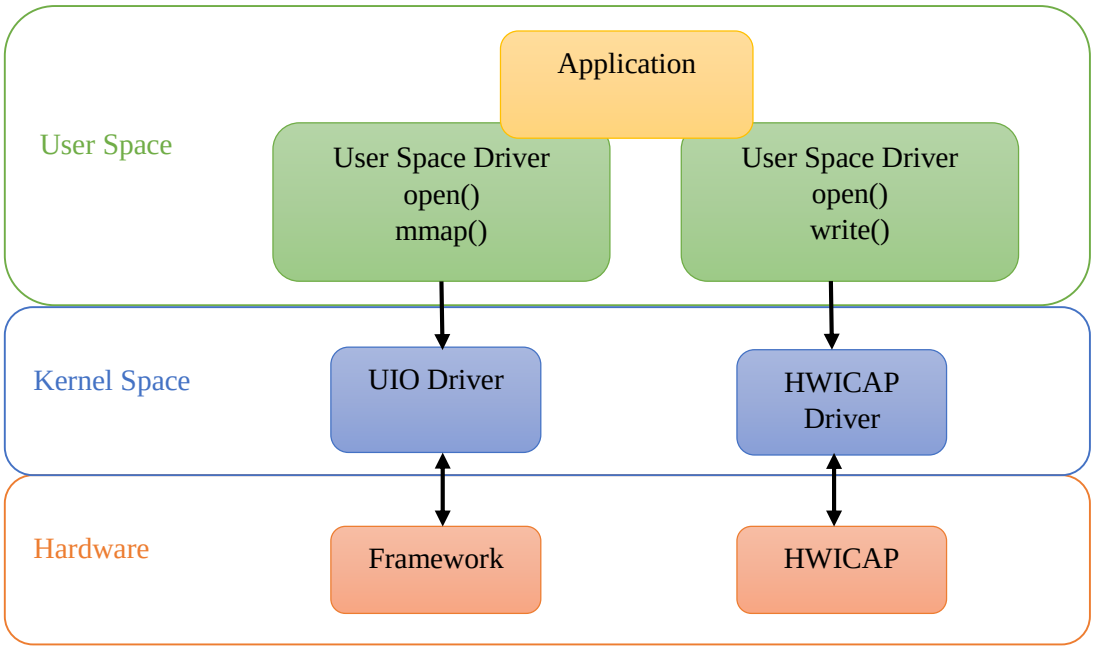

Figure 9 Linux Kernel Hierarchy

8.2 NFA Control Application

In order to perform some task, an application needs be created to interface with the hardware. The cross compiler in the PetaLinux tools support the C language. Cross Compilation is creating executable code for a different architecture than the host computer is running. As the Linux Kernel does not contain its own compiler, all the compilation needs to be done before booting the system, and the executable files need to be packaged with the kernel image. In the same packaging process, the partial bitfiles are also added to the image.

The primary task of the control application is to perform the partial configuration, accept user input and return results from the framework.

The first part of the software application is to access the framework and HWICAP. Using “open()”, each of the peripherals are bound to a file descriptor. The framework file descriptor is used with “mmap()” to bind the registers to a pointer array. By assigning values to and reading from the first four location of the array, the registers can be accessed.

The HWICAP can be accessed using the write() function with the file descriptor. In order to correctly write the partial BIT file to hardware, the header needs to be parsed. This procedural for doing this can be found in the software application of the Partial Runtime Reconfiguration Tutorial source files (Xilinx UG744). After parsed, the bitfile is written to the ICAP.

After the application has bound the correct peripherals, it requests the user for an unsigned integer to be processed. The integer is written to the framework memory, and while the complete control bit is low operations are performed. The framework requests additional modules by setting the request control bit high. This bit is checked every iteration of the while loop, and when the high bit is detected it reads from the third register to find what modules are requested. This value is checked by to an array stored in the application, and any value that is different from the value read has the corresponding module written or removed. This is done until the complete bit is read high, upon which the operations conclude and the accept value is read from the second register. After the input is entered, no other user interactions are required. The system will handle all reconfigurations until the string is completely processed. After completion, the system is internally reinitialized, and all remaining state machines are removed. The system is now ready for a new string to process.

Despite the NFA being expressed with an alphabet; the software has no functionality for translating a string input to decimal. The user will have to do this conversion prior to using the system, as only decimal inputs are processed. The software portion of the design contains many redundancies and should be rewritten if the system is to be utilized efficiently.

9 USING THE SYSTEM AND FINAL VERIFICATION

9.1 PROGRAMMING FPGA

In order to program the FPGA, the Xilinx Software Development Kit is used. In order to successfully several files need to imported to the SDK workplace. The first is the .xml file that was generated by XPS when exporting the design to SDK. This file is needed to get the correct bootloop for the processor. The bootloop is needed whenever a system is brought out of reset with uninitialized memory, as running invalid instruction might put the processor in a state that can't be recovered from. The bootloop is intended to keep the processor in a defined state until an application can be downloaded and run. The next two files are the bitfile and the block RAM map file.

After the FPGA has been programmed it can be accessed using the Xilinx Microprocessor Debugger. Through it applications can be download and run. To connect, use the "connect mb mdm" command in the XMD consol. Next, the image is downloaded using the command "dow image.elf".

To run the system, an interface needs to connected to the serial communications device through the USB to UART device by connecting a USB to the development board. The system transfers data at a baud rate of 115200Hz.

After the system has booted, the "root" user can be accessed using password "root". From here navigate to the control application located at "/bin" and excecure the application using the command "./nfa_control". When loaded the application requests the user for a string to be processed.

9.2 VERIFICATION

The system is verified using the input strings from the VHDL testbench and matching the outputs to those in the testbench. For debugging, the program is compiled with prints every time a partial reconfiguration is performed. Looking at the serial outputs, found in **Appendix A** Test Runs, correct operation is observed and the system is verified.

10 EVALUATION

As noted in the Earlier Work section, there already exists plenty of efficient NFA designs for reconfigurable hardware, and this design is not one of them. The biggest bottleneck for the design is performing the runtime reconfiguration. While other parts of the system can be optimized, the maximum throughput for the HWICAP running at 100mhz is 3,2 Gbps. The smallest differential BIT file is 12 761 bytes, or 102 088 bits. At the maximum frequency the reconfiguration takes 0.000032 seconds, or 32us. Compared to a static solution on the same hardware running at 100 MHz, processing a single character takes one clock cycle, which is 10ns.

The timing results from the verifications are shown in Table 5. The average time per symbol is approximately 190ms, which is 5 symbols per second. From these numbers the software would seem the culprit for slow runtime, as each reconfiguration is expected to only take 32us. Studying the reconfigurations performed for the string BBA, we note that 9 reconfigurations are performed, resulting in a minimum of 300ms performing reconfigurations of the total of 600ms execution time. For the string ABBABBA, 19 reconfigurations are performed, in total 608ms of 1258ms. Combining this with read and write times, this hands on approach of managing partitions is inherently slow. While speed-ups are achievable, more than half of the execution times is due to reconfigurations. Further analysis is required for optimization.

Input String	Time	Average Time per Symbol
BBA	598ms	199ms
ABC	400ms	133ms
ABBABBA	1258ms	179ms
BBABACBBABC	2092ms	190ms
ABBABBACAAACABAC	3071ms	190ms

Table 5 System performance

One of the speed limitations for this design is the layout for the runtime reconfiguration. Each partial BIT file must contain information for an entire reconfiguration frame. While the reconfiguration frame of the Virtex-6 is 40 CLBs high and 1 CLB wide, the NFA module only occupies 2 CLBs. Having some way of reducing the size of the BIT files would save time in reconfiguration.

11 Discussion and Future Work

The intended design has been implemented and verified, finally realizing the design proposed by Volden and Svartstad. This is the first working implementation of a design utilizing runtime reconfiguration to actively manage non-deterministic finite automata. Having evaluated the performance of the design, the biggest performance degradations have been in software. The frequency of which operations in hardware are interrupted is too high to reap any benefits from the hardware implementation, and a software solution would be preferred. Despite this, the system is a good example of the interworking between hardware and software, and an excellent design for showcasing the usage of runtime reconfiguration.

Heimark used in his thesis, (Heimark, 2015), a singular state design that used configuration vectors as to define states and transition function. The vectors could be loaded to each machine from software. Not being implemented in this design, the advantage of this design is the reusability of the same reconfiguration module. At the cost of configuration time, only one module for each partition would be required. The final iteration of the Virtex-4 design featured other improvements that are not implemented in this design. Before improvements are made, the performance of the design needs to be addressed and considered.

In order to optimize this design, a different application should be approached. The tasks performed on the reconfigurable hardware needs to be more independent of software interventions. An example of such a design, is a system running several different NFAs simultaneous to process strings. These modules could be managed independently, but utilizing an optimized design capable of running the NFA using other techniques than copying when several states are reached.

12 Conclusion

The task has been completed and the imagined system has been developed. Using both partial reconfiguration and the Linux Kernel, the toolchain for the design contains many steps. Hopefully the work done in this thesis will serve as guidelines for implementing designs with the older Xilinx ISE 14.7 tools or as a reference design for implementing NFAs in conjunction with runtime reconfiguration.

REFERENCES

Blomkvist, D. (2013). *Self-cloning state machines on FPGA.* NTNU, Trondheim.

Cheng, X., Shuhui, C., Jinshu, S., S.M., Y., & C.K., H. (2016). *A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms and Hardware platforms.*

Compton, K., & Hauck, S. (2002). Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys, Vol. 34, No. 2,* 171-210.

Heimark, T. (2015). *FPGA virtualization layer for non-deterministic state machines.* NTNU, Trondheim.

Rabin, M. O., & Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development 3,* 114–125.

Sidhu, R., Wadhwa, S., Mei, A., & Prasanna, V. (2000). A Self-Reconfigurable Gate Array Architecture. *10th International Workshop on Field Programmable Logic and Applications.*

Siduh, R., & Prasanna, V. (2013). *Fast Regular Expression Matching Using FPGAs.*

Sourdis, I., Bisop, J., Cardoso, J., & Vassiliadis, S. (2008). Regular Expression Matching in Reconfigurable Hardware. *Journal of Signal Processing Systems 51,* 99-121.

Svarstad, K., & Volden, K. (2011). *Replicating Non-Deterministic Finite State.*

Volden, K. (2011). *Compiling Regular Expressions into Non-Deterministic State Machines for Simulation in SystemC.* NTNU, Trondheim.

Xilinx DS817. (n.d.). LogicCORE IP AXI HWICAP.

Xilinx UG977. (n.d.). PetaLinux SDK User Guide, Getting Started.

Xilinx UG980. (n.d.). PetaLinux SDK User Guide, Board Bringup Guide.

Xilinx. (n.d.). Partial Reconfiguration User Guide UG702.

Xilinx UG744. (n.d.). Partial Reconfiguration of a Processor Peripheral Tutorial.

Xilinx. (n.d.). Virtex-6 FPGA Configuralbe Logic Block UG364.

Xilinx, DS150. (n.d.). Virtex-6 Family Overview.

Xilinx, UG534. (n.d.). ML605 Hardware User Guide.

APPENDIX A **TEST RUNS**

```
Please enter int value of binary string to be processed:
26
Wrote NFA to Comp 0
Wrote NFA to Comp 1
Wrote NFA to Comp 2
Erased NFA from Comp 1
Wrote NFA to Comp 3
Wrote NFA to Comp 4
Erased NFA from Comp 3
Erased NFA from Comp 4
Wrote NFA to Comp 5
Accept: 48
```

Figure 10 Test run, "AAB"

```
Please enter int value of binary string to be processed:
6761
Wrote NFA to Comp 0
Wrote NFA to Comp 1
Wrote NFA to Comp 2
Wrote NFA to Comp 3
Erased NFA from Comp 2
Wrote NFA to Comp 4
Wrote NFA to Comp 5
Erased NFA from Comp 4
Erased NFA from Comp 5
Wrote NFA to Comp 6
Erased NFA from Comp 3
Wrote NFA to Comp 7
Wrote NFA to Comp 8
Erased NFA from Comp 7
Wrote NFA to Comp 9
Wrote NFA to Comp 10
Erased NFA from Comp 9
Erased NFA from Comp 10
Wrote NFA to Comp 11
Accept: 196608
```

Figure 11 Test run, "ABBABBA"

```
Please enter int value of binary string to be processed:
3671448169
Wrote NFA to Comp 0
Wrote NFA to Comp 1
Wrote NFA to Comp 2
Wrote NFA to Comp 3
Erased NFA from Comp 2
Wrote NFA to Comp 4
Wrote NFA to Comp 5
Erased NFA from Comp 4
Erased NFA from Comp 5
Wrote NFA to Comp 6
Erased NFA from Comp 3
Wrote NFA to Comp 7
Wrote NFA to Comp 8
Erased NFA from Comp 7
Wrote NFA to Comp 9
Wrote NFA to Comp 10
Erased NFA from Comp 9
Erased NFA from Comp 10
Wrote NFA to Comp 11
Erased NFA from Comp 6
Erased NFA from Comp 8
Wrote NFA to Comp 12
Wrote NFA to Comp 13
Erased NFA from Comp 0
Erased NFA from Comp 1
Wrote NFA to Comp 14
Erased NFA from Comp 11
Wrote NFA to Comp 15
Wrote NFA to Comp 2
Erased NFA from Comp 14
Wrote NFA to Comp 4
Wrote NFA to Comp 5
Erased NFA from Comp 15
Wrote NFA to Comp 3
Wrote NFA to Comp 7
Erased NFA from Comp 12
Erased NFA from Comp 13
Erased NFA from Comp 3
Wrote NFA to Comp 9
Wrote NFA to Comp 10
Wrote NFA to Comp 6
Erased NFA from Comp 9
Erased NFA from Comp 10
Wrote NFA to Comp 0
Erased NFA from Comp 2
Erased NFA from Comp 7
Wrote NFA to Comp 8
Accept: 1280
```

Figure 12 Test run, "ABBABBACAAACABAC"

APPENDIX B **DEVELOPMENT ENVIRONMENT**

Lenovo X240 Laptop

- Windows 10
    - Xilinx ISE 14.7
    - PlanAhead
    - Xilinx Software Development Kit
    - Xilinx Platform Studio
- VirtualBox
    - CentOS 6.7
        - PetaLinux Tools