# A RECONFIGURABLE PATTERN MATCHING HARDWARE IMPLEMENTATION USING ON-CHIP RAM-BASED FSM

by

Indrawati Gauba

A thesis

submitted in partial fulfillment

of the requirement for the degree of

Master of Science in Computer Engineering

Boise State University

August 2010

BOISE STATE UNIVERSITY GRADUATE COLLEGE

# DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Indrawati Gauba

Thesis Title: A Reconfigurable Pattern Matching Hardware Implementation Using On-Chip RAM-Based FSM

Date of Final Oral Examination: 07 May 2010

The following individuals read and discussed the thesis submitted by student Indrawati Gauba, and they evaluated her presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

| | |
|---|---|
| Nader Rafla, Ph.D. | Chair, Supervisory Committee |
| Jennifer A. Smith, Ph.D. | Member, Supervisory Committee |
| Thad Welch, Ph.D. | Member, Supervisory Committee |

The final reading approval of the thesis was granted by Nader Rafla, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

*to Mom, Dad...*

# ACKNOWLEDGEMENT

I would like to sincerely thank my advisor Dr. Nader Rafla for his valuable guidance and support while completing my graduate education. I am grateful for his confidence in me that I could do a good job with my thesis. It has been a great pleasure, in fact, an honor to work with him.

I would also like to thank Dr. Jennifer A. Smith and Dr. Thad Welch for being on my thesis committee and guiding and encouraging me throughout my research work.

Finally, I would like to thank my family for their unwavering support and encouragement. I am grateful to my son for being patient and understanding during the entire process. Thank you all.

ABSTRACT

The use of synthesizable reconfigurable IP cores has increasingly become a trend in System on Chip (SOC) designs. Such domain-special cores are being used for their flexibility and powerful functionality. The market introduction of multi-featured platform FPGAs equipped with on-chip memory and embedded processor blocks has further extended the possibility of utilizing dynamic reconfiguration to improve overall system adaptability to meet varying product requirements. A dynamically reconfigurable Finite State Machine (FSM) can be implemented using on-chip memory and an embedded processor. Since FSMs are the vital part of sequential hardware designs, the reconfiguration can be achieved in all designs containing FSMs.

In this thesis, a FSM-based reconfigurable hardware implementation is presented. The embedded soft-core processor is used for orchestrating the run-time reconfiguration. The FSM is implemented using an on-chip memory. The hardware can be reconfigured on-the-fly by only altering the memory content. The use of a processor for reconfiguration enables SOC designers to utilize both software and hardware capability to achieve reconfiguration. This scheme of reconfigurable hardware implementation is independent of the placement and routing of the hardware on the FPGA. To demonstrate the feasibility of the proposed approach, the Knuth-Morris-Pratt (KMP) algorithm was implemented. A unique way of using memory-based FSM to reconfigure and speed up the KMP search algorithm has been introduced. With the proposed technique, the system

can reconfigure itself based on a new incoming pattern and perform a pattern search on a given text without involving a host processor.

Data extracted from test cases shows that the proposed approach made the maximum achievable frequency of the design independent of the pattern length. The number of clock cycles required to match the pattern in the worst case is equal to the pattern length plus the text length (O (m+n)).

TABLE OF CONTENTS

ACKNOWLEDGEMENT ..... vi

ABSTRACT ..... vii

TABLE OF CONTENTS ..... ix

LIST OF TABLES ..... xii

LIST OF FIGURES ..... xiii

CHAPTER 1—INTRODUCTION ..... 1

1.1 Organization ..... 3

CHAPTER 2—RECONFIGURABLE HARDWARE & CURRENT APPROACHES ..... 4

2.1 Reconfiguration Techniques ..... 4

2.1.1 Dynamic Reconfiguration ..... 5

2.1.2 Partial Reconfiguration ..... 6

2.1.3 Multi-Context Architecture ..... 8

2.1.4 Reconfiguration Using On-Chip Memory ..... 10

2.1.5 Reconfiguration Using a Self-Reconfigurable Finite State Machine ..... 10

2.2 Summary ..... 11

CHAPTER 3—RECONFIGURABLE FINITE STATE MACHINE: PROBLEM STATEMENT AND SOLUTION ..... 12

3.1 Finite State Machines (FSMs) ..... 12

3.1.1 Reconfigurable Finite State Machines ..... 14
3.1.2 Relevance of Reconfigurable Finite State Machines ..... 14
3.2 Related Work ..... 15
3.3 Proposed Approach ..... 19
3.4 Design Strategy ..... 20
3.5 Summary ..... 22
CHAPTER 4—KMP STRING MATCHING ALGORITHM ..... 23
4.1 Relevance of String Matching Algorithm ..... 23
4.2 Brute Force Search for String Matching ..... 23
4.3 KMP Algorithm ..... 25
4.3.1 Skipping Outer Iterations ..... 26
4.3.2 Skipping Inner Iterations ..... 27
4.4 Summary ..... 31
CHAPTER 5—DESIGN IMPLEMENTATION ..... 32
5.1 Design Modeling ..... 34
5.2 Design Implementation ..... 35
5.2.1 Hardware ..... 36
5.2.2 Hardware Logic Implementation of the KMP Algorithm ..... 41
5.2.3 Processor ..... 43
5.2.4 Processor Interface and Control Signals ..... 46
5.2.5 Software Implementation ..... 50
5.4 Design Synthesis and Implementation ..... 52

5.5 Summary ..... 52

CHAPTER 6—EXPERIMENTAL RESULTS AND ANALYSIS ..... 53

6.1 Simulation Testing ..... 53

6.2 Hardware Testing ..... 58

CHAPTER 7—CONCLUSION AND FUTURE WORK ..... 65

BIBLIOGRAPHY ..... 67

LIST OF TABLES

Table 4.1: Brute force search iteration result for iterations i=0 to i=7 ..... 24

Table 5.1: Translation from π[i] to FSM next state transition and output function..... 39

Table 5.2: Slave registers usage description ..... 46

Table 5.3: Control signal definitions ..... 47

Table 5.4: Control signal values for FSM state transition memory update ..... 48

Table 5.5: Control signal values for FSM output memory update ..... 49

Table 5.6: Control signal values for setting FSM output and input signal width ..... 50

Table 6.1: Clock cycle required for FSM update..... 62

Table 6.2: Performance of the implementation for various values of m with ..... 62

Table 6.3: Performance comparison for various values of m with $n=10^4$ ..... 63

LIST OF FIGURES

Figure 2.1: FPGA configuration approaches: a) Static configuration, b) Dynamic reconfiguration [8] ..... 6

Figure 2.2: Partial reconfigurable architecture ..... 7

Figure 2.3: Single context dynamically reconfigurable architecture [8] ..... 9

Figure 2.4: Multi-context dynamically reconfigurable architecture [8] ..... 9

Figure 3.1: Mealy FSM ..... 13

Figure 3.2: Moore FSM ..... 13

Figure 3.3: Transition taken during FSM reconfiguration [5] ..... 16

Figure 3.4: Reconfigurable FSM implementation [5] ..... 17

Figure 3.5: Custom reconfigurable circuit for FSM implementation [6] ..... 18

Figure 4.1: Iteration 2 and 3 ..... 26

Figure 4.2: Iteration 2 and 4 ..... 27

Figure 4.3: Pseudo code with skipped outer iteration ..... 27

Figure 4.4: State transition diagram for pattern “ababca” ..... 29

Figure 4.5: Phase 2 pattern “ababca” search execution using π[] array ..... 29

Figure 4.6: KMP algorithm phase 1: Prefix function computation [3] ..... 30

Figure 4.7: KMP algorithm phase 2: Text search [3] ..... 31

Figure 5.1: Elements and stages of XPS and EDK leading to FPGA configuration ..... 33

Figure 5.2: Spartan-3E FPGA Starter Kit Board ..... 36

Figure 5.3: KMP system block diagram ..... 37

Figure 5.4: RAM-based FSM implementation ..... 38

Figure 5.5: KMP algorithm phase 2: Pattern search ..... 41

Figure 5.6: Block diagram of KMP hardware logic ..... 42

Figure 5.7: MicroBlaze system with peripheral buses connecting user cores ..... 44

Figure 5.8: Hardware architecture of the reconfigurable KMP ..... 45

Figure 5.9: Software implementation flow ..... 51

Figure 6.1: Simulation waveform of KMP search run for pattern “ababca” ..... 55

Figure 6.2: Simulation waveform of KMP search run for pattern “ababca” with text containing two overlapped patterns ..... 57

Figure 6.3: Result of pattern search “ababca” at the terminal console ..... 60

Figure 6.4: Clock cycles vs pattern length after first match ..... 61

# CHAPTER 1—INTRODUCTION

The rising speed performance of modern FPGAs enabled them to offer possible parallelism of Application Specific Integrated Circuits (ASICs) along with flexibility. However, their increasing manufacturing cost has raised the demand to add more flexibility to System on Chip (SOC) without sacrificing performance. Reconfiguration technologies are viewed by the SOC designers as a tool to achieve this target. Dynamically reconfigurable hardware has added a new dimension to SOC design by combining the capability introducing modifications to post-fabrication functionality modification (not present in conventional ASICs) with the spatial/parallel computation style (not present in instruction set processors). These technologies allow the hardware to be customized after device fabrication to meet the instantaneous needs of a specific application.

The concept of reconfigurable hardware is not new and has been in the market for quite some time. It was first introduced by G. Estrin, B. Bussell, R. Turn and J. Bibb in 1963 [1]. These early techniques used a general purpose processor to reconfigure its computational resources for independent computations and a multiplexer to control routing between these computational resources. In the mid-1980s, the advancement of reconfigurable hardware technology led the way to the development of new techniques for FPGA reconfiguration. Since then, various techniques for achieving reconfiguration have been explored.

FPGA devices contain configurable logic blocks whose functionality is determined through the programmable configuration bits. Vendor specific tools generate a configuration bit-stream to program FPGA devices. Reconfiguration technologies manipulate the configuration bit-stream to achieve reconfiguration. Partial reconfiguration of FPGAs is based upon the partial manipulation of configuration bit-streams and downloading them to FPGA devices. Self-reconfiguration can also be realized by multi-context FPGA devices [2, 3]. These FPGA devices support dynamic reconfiguration by allowing the storage of multiple configuration bit-streams in FPGA memory and switching between them using a dedicated signal for control. A multi-context FPGA-based self-reconfigurable string matching hardware design was first presented in 1999 [3].

The advent of Static Random Access Memory (SRAM) based FPGAs has made Run Time Reconfiguration (RTR) feasible by changing the value of the configuration stream stored in SRAM cells to realize a new function. A RTR system is a heterogeneous system consisting of at least one FPGA and a configuration manager. Markus Koster and Jurgen Teich [5] proposed a hardware configuration manager design for the implementation of a RTR Finite State Machine (FSM) in 2002 [5]. The external (host) or embedded processor can also be used to achieve RTR. The use of an embedded processor in such a capacity is proposed by Joao Canas Ferreira in 2005 [17].

Modern FPGAs are complex structures. They have embedded configurable logic blocks that can be used to implement logic as well as distributed memory. Such memory blocks allow implementation of sequential blocks in such a way that it requires fewer

logic cells than traditional flip-flop based implementations. This can be utilized to implement the sequential part of the design as an FSM. The functionality of such sequential design can be reconfigured by altering the functionality of the finite state machines. Reconfigurable FSMs have given an alternative to achieving reconfiguration without the need to manipulate configuration bit-stream.

This thesis provides an in-depth discussion of a memory-based FSM implementation that is dynamically reconfigurable using an embedded processor. A hardware-software co-design is developed as a run-time reconfigurable system. The Knuth-Morris-Pratt string matching hardware is implemented to prove the feasibility of this approach [13]. This thesis provides a core framework design for a portable hardware design for run-time reconfiguration for FSM-based sequential designs.

### 1.1 Organization

The organization of this thesis is as follows. Chapter 1 briefly introduces and outlines the thesis organization; Chapter 2 reviews the current reconfiguration techniques. Finite state machines and implementation approaches of reconfigurable FSM are detailed in Chapter 3. Since KMP string matching algorithm is implemented to prove feasibility of concept, Chapter 4 provides the explanation of the KMP string matching algorithm in detail.

In Chapter 5, information gathered in previous chapters is used to develop the hardware and software co-design system for the KMP. Chapter 6 details the various tests conducted to verify the functionality and to assess the system performance. Chapter 7 concludes the thesis and discusses future work.

# CHAPTER 2—RECONFIGURABLE HARDWARE & CURRENT APPROACHES

A system which allows changing behavior or adds new features to an existing system after product manufacturing is defined as an upgradable system. Device manufacturers search for a field upgradeable system to meet the stringent time-to-market requirement. Such a system enables manufacturers to release the initial version of the product to the market, and extends the features during the product life time by system upgrades. The FPGA-based reconfigurable hardware provides a similar solution by adding reconfiguration capability to the system to meet additional product requirements.

In a static implementation strategy of FPGA-based systems, the implementation uses a fixed configuration of hardware logic resources. These resources are configured before the system operates, and maintains the same configuration throughout the operation. The techniques of reconfiguration allow the hardware logic resources to accommodate different applications or to add new features to the existing application. Much work has been accomplished on device reconfiguration techniques for FPGA-based systems. This chapter reviews some of the most common reconfiguration techniques.

## 2.1 Reconfiguration Techniques

Development of any FPGA-based system starts with entering the schematic of the hardware design using a FPGA platform specific front end tool or describing the design using a Hardware Description Language (HDL such as VHDL or Verilog). Then, a software synthesis tool is used to convert the design into a netlist of FPGA family

specific logic resources such as LUTs (Loop up table array) and flip-flops. After synthesis, a Place and Route tool performs the task of placing and routing the synthesized logic onto the target device and generates the bit-stream file. The generated bit-stream (configuration bit-stream) file is downloaded onto the FPGA to implement the design on its fabric.

### 2.1.1 Dynamic Reconfiguration

Traditional FPGA-based systems are usually configured before starting the execution of the application and referred to as statically reconfigurable. To reconfigure the FPGA, the system has to be halted till reconfiguration is done and then the FPGA needs to be restarted with the new configuration. On the other hand, a dynamically reconfigurable system allows one to run the reconfiguration and application execution in parallel. It is based on the concept of virtual memory. When the physical resources of the FPGA is much less than the overall system requirement, the system is divided into hardware segments that do not need to run concurrently and a dynamic allocation scheme is used to re-allocate the logic resources to hardware segments at run time [8]. It enhances the system flexibility and performance by dynamically loading and unloading the optimized circuit configuration during system operation. Dynamic reconfiguration is supported on FPGA devices via JTAG or vendor specific external interface ports. External intelligent hardware such as a processor or microcontroller or dedicated hardware blocks residing on the FPGA itself are used to reconfigure the device. Configurations required for reconfiguration are stored either on FPGA memory if enough memory is available or in external memory. Figure 2.1 shows the static and dynamic

reconfiguration approaches. A statically configured FPGA system, once configured cannot be reconfigured again, while a dynamically configurable FPGA system can be reconfigured multiple times to implement different functionalities.

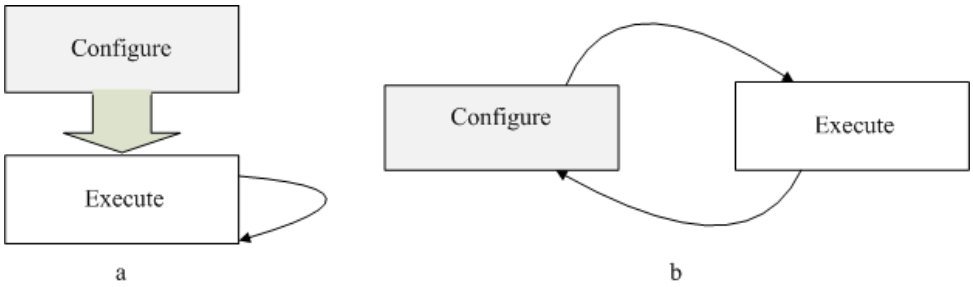


**Figure 2.1: FPGA configuration approaches: a) Static configuration, b) Dynamic reconfiguration [8]**

2.1.2 Partial Reconfiguration

Some applications require modifying only part of the circuit for reconfiguration. In such cases, partial reconfiguration can be used. Partial reconfiguration is an important feature in some FPGA architectures. It is the ability to reconfigure a portion of a FPGA while the remainder of the design is still operational. Certain areas of a device can be reconfigured while other areas remain operational and unaffected by reprogramming [8, 12]. Partial reconfiguration is done while the device is active without the need of restarting.

On Xilinx FPGA devices, Virtex series (such as Virtex II and Virtex Pro) support dynamic partial reconfiguration via an Internal Configuration Access Port (ICAP) interface [19]. The system already implemented on FPGA can internally access it. It

becomes available after an initial (externally controlled) configuration is complete and allows the designer to control device reconfiguration at run-time. The reconfiguration is controlled by an on-chip embedded processor via ICAP. The advantage of reconfiguration via ICAP is that it does not require an external configuration port. It can partially alter the configuration bit-stream. The drawback of this technique is that it is slow and it must perform a memory read operation first. Figure 2.2 shows a partial reconfigurable architecture inside FPGA memory. The first left block represents the partial configuration that needs to be loaded on the FPGA. The next block represents the configuration memory before reconfiguration takes place. The dark grey area in this block is the unused area of the configuration memory while the area in white is the configured part. The final block shows the status of configuration memory after reconfiguration.

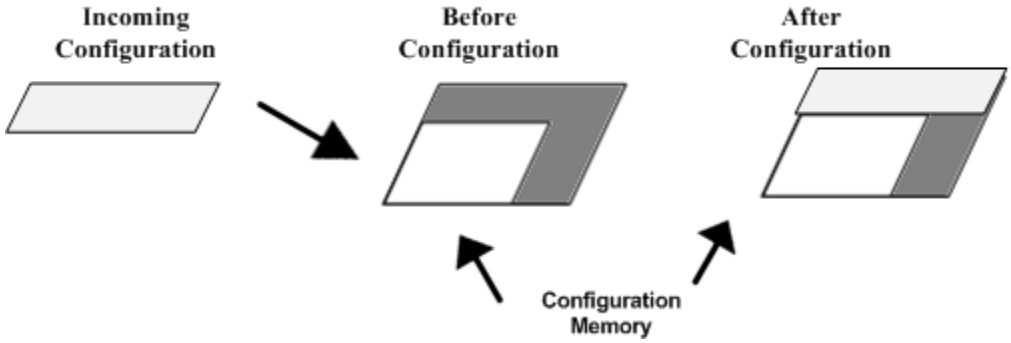


**Figure 2.2: Partial reconfigurable architecture**

There are two styles of Partial Reconfiguration: Module based and Difference based. Module-based partial reconfiguration is used when the modules are inter-dependant (have common signals) and need to communicate. To achieve inter-module communication, a special bus macro is used to provide a fixed bus for inter-design

communications and allow signals to cross over the module boundaries. At each reconfiguration, the bus macro is used to establish the unchanging routing channels between the modules to guarantee correct connections.

Difference-based partial reconfiguration is accomplished when a very small change in the design is needed to achieve the reconfiguration. Instead of storing the full configuration stream, the difference-based configuration bit-stream stores only the difference between the base configuration and the configuration needed to reconfigure the device. At first, a configuration bit-stream is generated. Then, a vendor specific software tool (such as FPGA-Editor) is used to make changes in the existing configuration to generate the difference-based configuration bit-stream. Since bit-stream differences are extremely small compared to the bit-stream of the entire device, fast switching between configurations of module(s) is possible.

### 2.1.3 Multi-Context Architecture

Self-reconfiguration can be realized by multi-context FPGA devices that allow on-chip manipulation of configuration bit-streams [3, 8]. Multi-context FPGA devices have on-chip RAM that can store multiple pre-programmed configuration contexts (configuration bit-streams). Only one context can be active at a given time. The architecture can switch quickly between different configurations using a dedicated signal that determines which configuration data should be used. Overlapping of computation with reconfiguration is allowed through background loading of configuration data during circuit operation. Figure 2.3 shows single context and Figure 2.4 shows multi-context dynamically reconfigurable architectures. The configuration memory of a multi-context

FPGA can be visualized as multiple planes of configuration memory stacked on each other. Each plane can store an individual configuration bit-stream. Since single context FPGAs have only one memory plane, they can store only one configuration, on the other hand multi-context FPGA configuration bit-stream can store several configurations on any of the available memory planes. For example, in Figure 2.4, the incoming configuration bit-stream is stored on the top memory plane.

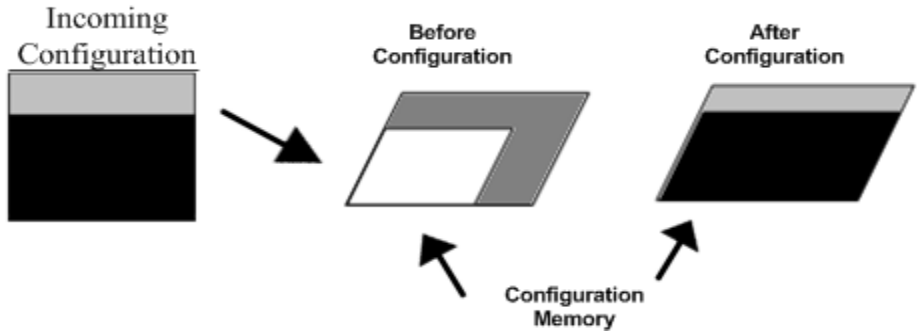


**Figure 2.3: Single context dynamically reconfigurable architecture [8]**

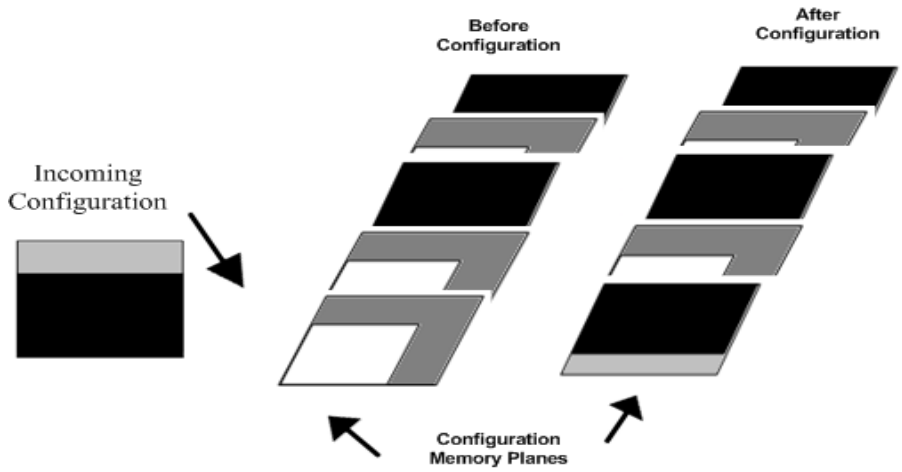


**Figure 2.4: Multi-context dynamically reconfigurable architecture [8]**

### 2.1.4 Reconfiguration Using On-Chip Memory

All the reconfiguration techniques described previously are based on manipulation of the configuration bit-stream. The on-chip memory (memory available on the FPGA device itself that can be accessed by the design) provides another alternative way to achieve reconfiguration. Instead of implementing logic that alters the configuration bit-stream, logic to control the functionality on-the-fly via altering on-chip data memory can be implemented. The self-reconfiguration can be abstracted as a set of programmable primitive logic elements (logic-lets) and a network of programmable interconnection networks [4]. These elements are an application-specific hardware block used to implement some logic functionality. For example, given a block to implement an 8-bit arithmetic unit, a 16-bit arithmetic component can be realized by connecting two of these 8-bit arithmetic units. The interconnection networks can be considered as a sort of multiplexer, which is controlled by memory bits. Two logic elements can be connected by interconnection networks. The functionality of logic-lets and interconnection between them can be altered by the memory-based lookup-up table. A problem-specific control circuit determines the functionality and interconnection between them. In this way, the control circuit performs self-reconfiguration.

### 2.1.5 Reconfiguration Using a Self-Reconfigurable Finite State Machine

Finite state machines are the most vital components of any sequential logic system. Accordingly, functionality of these systems can be altered by reconfiguring the FSM. The functionality of the FSM can be changed by simply changing the FSM's state

transitions and/or their outputs. A lot of research has been done to develop various techniques of reconfigurable FSM implementation such as F-RAM and G-RAM based FSM, Forward Transmission Expression (FTE) based FSM, RAM/ROM based hierarchical FSM etc. [4-7]. This is the most flexible way to achieve run-time reconfiguration which, can be applied to almost any FPGA platform and is explored in this research.

## 2.2 Summary

Functionality implemented on FPGA logic resources can be altered by downloading a manipulated configuration bit-stream to the device. While partial reconfiguration usually uses an external access port to download the modified configuration bit-stream to the FPGA, some FPGA devices can be partially reconfigured using an internal access port. The partial reconfiguration models require configuration bit-streams to be modified by either a host processor or by an on-chip processor. In memory-based reconfigurable models, an on-chip hardcore processor is normally used for reconfiguration. A soft-core processor can also be utilized for the same purpose. Since in this thesis a reconfigurable FSM is used to implement pattern matching hardware, the next chapter analyzes the various approaches published in the literature to implement reconfigurable FSMs.

# CHAPTER 3—RECONFIGURABLE FINITE STATE MACHINE: PROBLEM STATEMENT AND SOLUTION

This chapter describes in brief finite state machines (FSMs) and their relevance. Then, it reviews existing approaches to reconfigurable FSM implementation. A solution to avoid some of the drawbacks of existing approaches is presented. A brief design strategy of the proposed solution is also discussed.

## 3.1 Finite State Machines (FSMs)

Finite state machines model the behavior of a system or a complex object, with a limited number of modes or states, where states or modes change with circumstances. A finite state machine consists of four main elements:

1) States that define behavior and may produce actions (outputs).
2) State transitions or changes from one state to another.
3) Conditions to allow changes from one state to another.
4) Externally or internally generated input events that trigger state transitions.

FSMs can be deterministic or non-deterministic. A deterministic FSM, also known as deterministic finite automaton (DFA), has one and only one transition to the next state for each set of current state and input vectors. A non-deterministic FSM, also known as non-deterministic finite automaton (NFA), can have more than one possible next state for each pair of current state and input vectors. NFAs are typically used to

reduce the complexity of the mathematical model required to establish many important properties in the theory of computation.

For practical applications, FSMs can be broadly divided in two types: Mealy and Moore FSMs. A Moore FSM is the state machine whose output vectors are a function of the current state only and do not depend on the input vector. On the other hand, with a Mealy FSM, output vector and next state depend on both the current state and input vector. The block diagram of Moore and Mealy FSMs are shown in Figure 3.1 and Figure 3.2, respectively. For this implementation, Moore type DFA FSMs are considered.

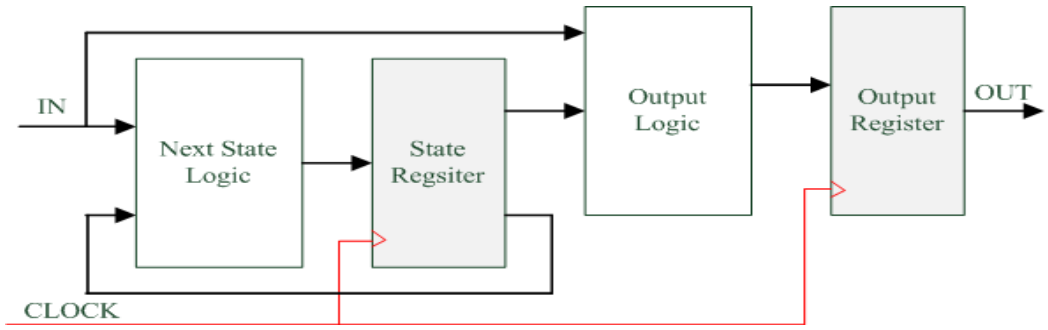


**Figure 3.1: Mealy FSM**

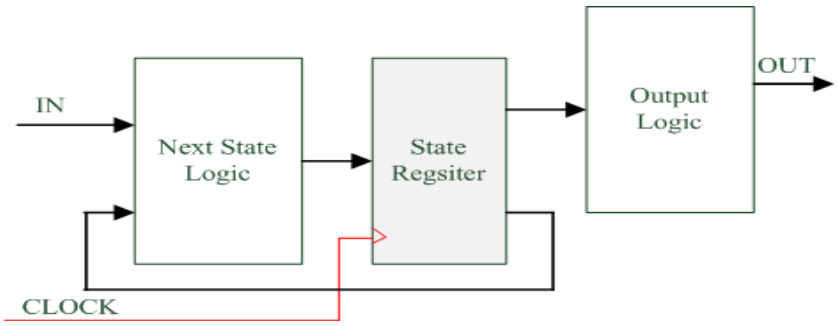


**Figure 3.2: Moore FSM**

### 3.1.1 Reconfigurable Finite State Machines

A reconfigurable finite state machine can be defined as a formal finite state machine, which has the ability to change output function and transition function or both during operation [5]. If the reconfiguration is initiated autonomously by the system itself, the FSM is called self-reconfigurable. Reconfiguration can also be initiated by external events, known as reconfiguration events.

### 3.1.2 Relevance of Reconfigurable Finite State Machines

Digital systems with statically and dynamically modifiable FSM functionality are required in a number of practical applications. They can be used in communication systems, in a crypto-processor, in embedded controllers, etc. Applications that need reconfigurable FSMs can be divided in two categories:

1. Autonomous sequential modules that are components of more complicated digital systems. For example, reconfiguration of a Mealy FSM that detects a sequence of two or more successive zeros makes it possible to change to detect successive ones instead of zeros.
2. Control circuits that require reprogrammable control units for a processor. The reconfigurable FSM allows the processor to be optimized for a particular problem that requires a specific subset of operations. For example, computations over Boolean and ternary matrices can be performed using reconfigurable control units. These units can solve different combinatorial problems such as covering of Boolean matrices, discovering subsets of vectors

with some pre-defined properties, etc. If a control unit is modeled by a reconfigurable FSM, it is possible to increase its performance by modifying the functionality of its FSM.

The design of MPEG4 natural video decoders and hardware implementation of string matching algorithms are examples of the use of reconfigurable FSM.

### 3.2 Related Work

A reconfigurable FSM can be realized by using distributed on-chip memory. An approach to implement such an FSM is presented in [5]. In this FSM implementation, two memory blocks, F-RAM and G-RAM, are used to implement the state transition and output functions of the FSM, respectively. The transition sequence required to reconfigure the initial FSM (referred to as a delta transition) is determined by a heuristic approach. The reconfiguration is realized by taking a sequence of delta transitions where, during each transition, the output and state transition function are updated until the FSM is reconfigured into the target FSM. Figure 3.3 shows the transition of a FSM used to count the number of ones in a bit-stream to a FSM that counts the number of zeros instead. Step one shows the original FSM and Step 4 shows the target FSM. The highlighted transitions are the delta transitions, which are taken to reconfigure the FSM. Reconfiguration is done in four steps by supplying the input sequence {1, 1, 0, 0}. In the first step, the FSM transitions into state S1. In the second step, the input/output function changes from 1/1 to 1/0. After this, the FSM transits into state S0; and at the final step, the input/output function changes from 0/0 to 0/1.

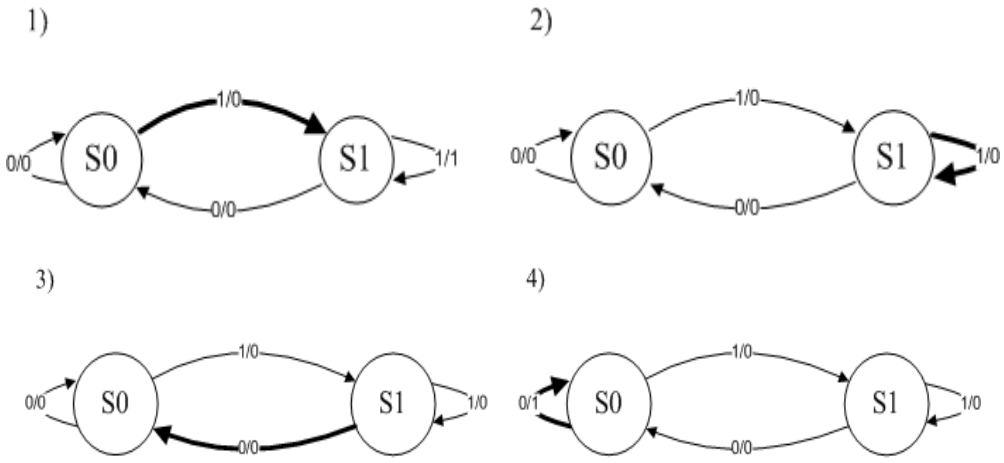


**Figure 3.3: Transition taken during FSM reconfiguration [5]**

In this approach, F-RAM and G-RAM memory blocks of the FPGA are used for FSM implementation and a hardware re-configurator is used for reconfiguration. The hardware re-configurator block is a hardware block that stores the delta transition sequences. On reception of a reconfiguration event, reconfiguration is initialized and the existing FSM is reconfigured into the target FSM. The architecture of such an FSM is shown in Figure 3.4. The input vector is connected to F-RAM and G-RAM memory blocks via the IN-MUX. In the normal mode of operation, the input vector is fed to both the F-RAM and G-RAM blocks. On reception of reconfigure event '*r*', the reconfigurator block emits delta transitions. The computed set of transitions required for reconfiguration sometimes includes dummy transitions that are not part of the target FSM [5].

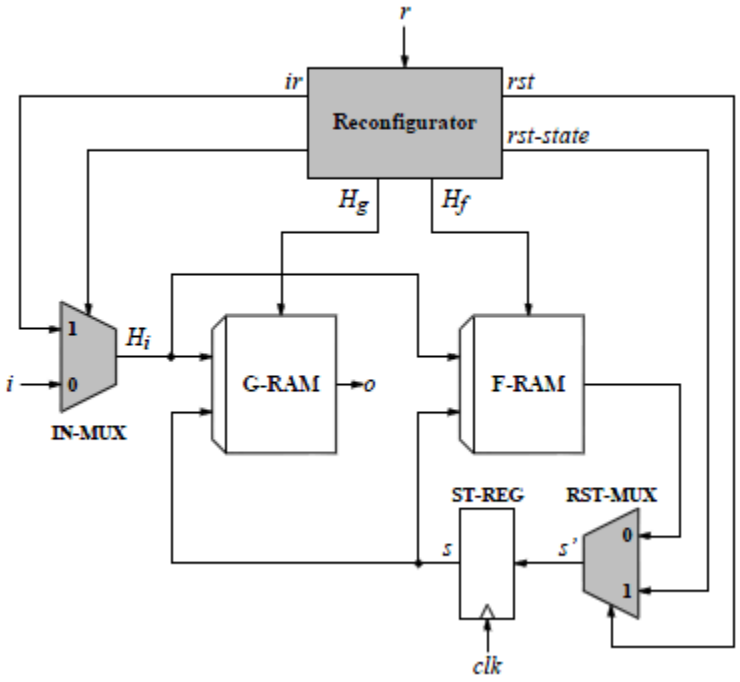


**Figure 3.4: Reconfigurable FSM implementation [5]**

Another alternative for FSM reconfiguration is the Forward Transition Expression (FTE) [6]. FTE is a Boolean expression needed for transition from the current state to the next state of an FSM. Each FTE can be optimized to utilize minimum FPGA resources. The optimized FTE is computed for each state transition. FTEs are stored in the FPGA memory and utilized for reconfiguration by the hardware re-configurator. The hardware re-configurator is a combinatorial logic block that is responsible for reconfiguration of FSM. This approach saves hardware logic cells over those required for traditional FSM implementations. The number of reconfigurations is limited and FTE needs to be calculated and optimized for each state. Also, during run-time, at each transition into a new state, a new FTE needs to be obtained from memory and then loaded onto the FPGA. In this technique, the number of reconfigurations is fixed and depends on the available

memory. The architecture of such an FSM is shown in Figure 3.5. The input vector is fed to a combinational logic block having the FTE. The comparator compares the current state with the next state. If a mismatch between the current and next state is found, a new FTE from memory is loaded into the combinatorial block.

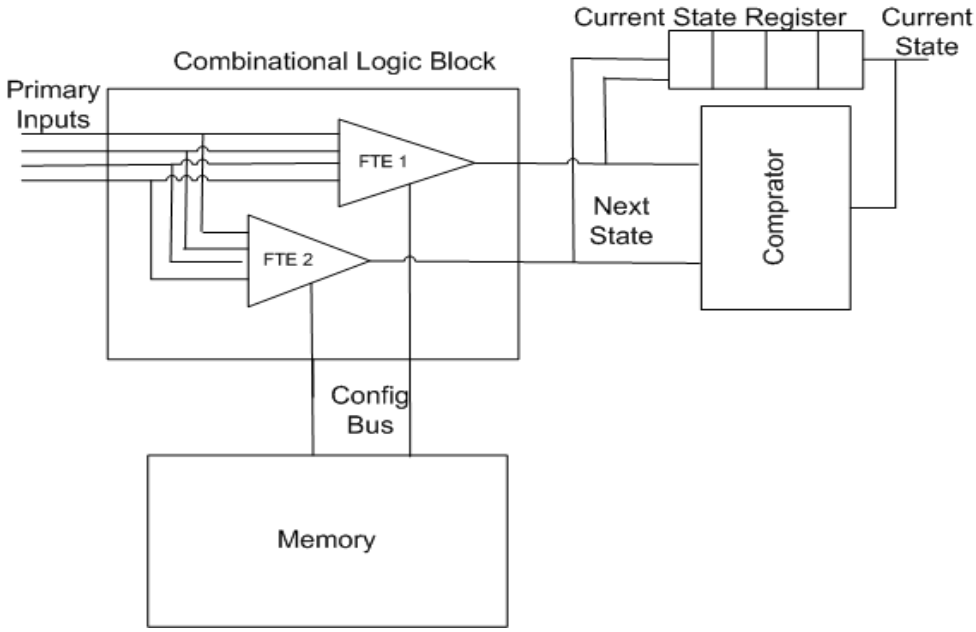


**Figure 3.5: Custom reconfigurable circuit for FSM implementation [6]**

Another approach is based on RAM/ROM hierarchical FSM implementation. Reconfiguration is achieved either by swapping pre-allocated areas on a partially dynamically reconfigurable FPGA, or by reloading memory-based cells in statically configured FPGAs [7]. In this approach, the number of reconfigurations is limited and depends on the size of the design and available chip area. To reconfigure the statically configured FPGA, a software model of the reconfigurable FSM has to be constructed and verified on the host system. Then, the bit-streams for memories such as MRAM, STRAM, and output RAM are represented as arrays. The FSM is verified on the host

system. After verification, these respective arrays are formally converted into the bit-streams for the FPGA, which can be downloaded using the dual-port capability of FPGAs. Again with this approach, a host system is required for generating configuration bit-streams.

An approach to implement reconfigurable hardware for string matching using the KMP algorithm is described in [3]. Reconfiguration is achieved via switching between multiple configuration contexts of a multi-context FPGAs. The pattern-specific back edges are constructed in the form of FSM, which is mapped onto hardware using a pre-configured template. With this approach, a specific (multi-context) feature is required for reconfiguration and a different context needs to be computed for each pattern. It also requires direct manipulation of configuration bits by the host processor. Another flaw with this technique is that the maximum achievable clock frequency of the system depends on the search pattern and increases with the pattern length.

Another method for KMP implementation using on-chip memory-based logic-lets is explained in [4]. These logic-lets can be connected in a network by altering the contents of memory-based FSMs. The FSM is computed for each specific application.

### 3.3 Proposed Approach

A new approach to reconfigurable pattern matching hardware implementation is proposed in this thesis. To develop device-independent reconfigurable hardware, a memory-based FSM is implemented and an on-chip processor is used for initiating the reconfiguration.

Traditional reconfiguration approaches require generation of configuration sequences in the form of technology dependent bit-streams. A reset is needed for a new reconfiguration to take effect. In multi-context FPGA-based reconfiguration methods, these pre-compiled configuration bit-streams need to be stored in configuration memory. Since there is a limited configuration memory space on FPGA devices, the number of reconfigurations is limited to the available configuration memory depending upon the size of configuration bit-streams. The use of an on-chip processor for reconfiguring the FSM eliminates the dependency upon a dedicated design of a hardware re-configurator and a limited number of reconfigurations. The use of the proposed approach also avoids the need of a reset to achieve reconfiguration. It also eliminates the effort exerted on generating configuration bit-streams for each configuration and saving these onto configuration memory. The proposed technique speeds up the hardware pattern search.

With the use of an embedded processor (soft-core such as MicroBlaze or hard-core processor such as Power PC), reconfiguration can be done on-the-fly [8, 23]. The speed of reconfiguration depends only on the processor clock cycles required to update the FSM memory contents. This approach is more generic than traditional reconfiguration approaches as it does not depend on device-specific features.

### 3.4 Design Strategy

The FSM is implemented using dual-port embedded RAM. The combination of current state and input vectors are used as an address to locate the RAM content. These contents consist of the next state transition and the output vector for each current state corresponding to the input vector. The FSM can be reconfigured easily by

reprogramming the RAM with a new or updated state transition table and output function table. The embedded soft-core processor (MicroBlaze) is used for reconfiguration.

This implementation strategy provides the flexibility of changing the width of input and output vectors using a multiplexer at the input and latches at the output. This strategy can be used in designs that need varying widths of input and output vectors. The multiplexers are enabled/disabled by controlling the bits stored in the memory, and updated by changing the contents of corresponding memory bits. The width of these vectors can be reconfigured by updating the contents of the memory locations providing the necessary control signals.

Several efficient software-based string matching algorithms such as Boyer-Moore Algorithm, Berry-Revindran, Suffix tree, Morris-Pratt, Knuth-Morris-Pratt (KMP) algorithm, and Colussi algorithm exist [15]. The KMP algorithm is one of the most efficient pattern matching algorithms that uses an FSM for search execution. Therefore, it is an ideal candidate for reconfigurable hardware implementation using a memory-based FSM. Also, previous hardware implementation of KMP algorithms is used as a base for performance comparisons with the proposed technique [3, 4].

The proposed implementation reconfigures itself to optimize the pattern search at each reception of a new search pattern. The delta transition required for reconfiguration is simply the difference between the existing FSM and target FSM state transition tables. Only memory contents for delta transition are needed to update and reconfigure the FSM. Unlike the approach described in [6], FTE is not needed to implement the FSM, freeing extra logic cells for design usage.

### 3.5 Summary

A reconfigurable FSM gives flexibility to change the functionality of sequential digital systems without the need of an external configuration bit-stream manipulation. Various techniques have been devised for efficiently implementing the FSM. On-chip memory-based FSM implementation is the simplest method of implementing a reconfigurable FSM. The hardware implementation of a KMP algorithm is proposed as an application of reconfigurable FSMs. The design exploits the reconfigurable feature of the memory-based FSM to self-reconfigure to adopt for each incoming search pattern instantly. Before proceeding to the implementation details of the proposed system, the next chapter details the architecture and functionality of the of KMP algorithm.

# CHAPTER 4—KMP STRING MATCHING ALGORITHM

String matching algorithms are considered ideal models for dynamically reconfigurable FSM implementations. The string matching problem consists of finding all occurrences of a pattern within a given text. This chapter gives a brief overview of a naive method of brute force string matching. Then, the KMP string matching algorithm is described in detail. Later in Chapter 5, the design and implementation of the proposed system is explained using the same test pattern. In Chapter 6, simulation waveforms of the search execution of the same test pattern are presented.

## 4.1 Relevance of String Matching Algorithm

The string matching problem has very high relevance to the field of Computer Science. Problems such as intrusion detection engines for internet network security, text processing, and pattern recognition and image matching present some examples where the string matching algorithm can be applied. Biology is another field that benefits greatly from such string matching problems. Finding patterns of DNA inside longer sequences has become central in the analysis of human genomes.

## 4.2 Brute Force Search for String Matching

A simple approach to match a pattern within a text could be implemented as a do-loop operation to check whether all the characters of the pattern match with the characters of a text string. If a pattern **P** of ***m*** character length is to be searched within a text string **T**

of ***n*** character length, the search procedure is as follows: Starting at any position *i*, the do-loop compares the characters of the pattern with the text characters until a mismatch is found. If a mismatch is found at some position, for example $i + j$, it starts searching again at position $i + 1$. This would lead to a very simple but inefficient search. Suppose a search pattern consisting of character array '*ababca*' has to be searched within the text consisting of character array '*tabacababcaxtab*', several iterations of the brute force search using loops have to be executed. Table 4.1 tabulates the iterations verses matched characters for this iterative process. Column 1 in the table lists the iteration number and row 2 lists the characters of the text string. 'X' is placed where a mismatch between text characters and pattern characters is found.

**Table 4.1: Brute force search iteration result for iterations i=0 to i=7**

| Character Number / Pattern Iterations | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **t** | **e** | **a** | **b** | **a** | **c** | **a** | **b** | **a** | **b** | **c** | **a** |
| **i=0** | X | | | | | | | | | | | |
| **i=1** | | X | | | | | | | | | | |
| **i=2** | | | a | b | a | X | | | | | | |
| **i=3** | | | | X | | | | | | | | |
| **i=4** | | | | | a | X | | | | | | |
| **i=5** | | | | | | X | | | | | | |
| **i=6** | | | | | | | A | b | a | b | c | a |

The table shows that the attempt to search the pattern at column position 4 in iteration 3, after a mismatch in iteration 2 at column 6, does not yield any match. Similarly, starting the pattern search at column position 6 in iteration 5, after a mismatch in iteration 4 at column 6, did not yield any match. It can be concluded that trying to match the character at position $i + 1$ after a mismatch is only necessary if the pattern is

such that its first $j - 1$ characters are exactly equal to $h$ - 1 (where $h < m$, $m$ is pattern length) characters starting at the second position in the search pattern itself. For example, in search pattern "aaabca", the first and second characters ('aa') are exactly similar to the second and third character ('aa') within the same pattern 'aaabc'. This pattern has to be searched from a text that contains character string "aaaabca". It can be noticed that the search pattern contains three consecutive characters of 'a' while the text contains 4 consecutive characters of 'a'. If the pattern search starts at character position '0', then the first iteration $i = 0$ will find a mismatch at the 3rd character position. Next iteration $i + 1$(search start at second character 'a') will find the pattern match. If the pattern is not such that first $j - 1$ characters are exactly equal to $h - 1$ characters starting at the second position, then trying to match the characters from position $i + 1$ in the text with the pattern would be wasteful and should be avoided. The time complexity of this algorithm is O (mn). In the worst case (if text does not contain any search pattern), $m$ x $n$ comparisons need to be performed to know that there is no match pattern, where $m$ is the length of search pattern and $n$ is the length of the text.

### 4.3 KMP Algorithm

The KMP algorithm is one of the most efficient pattern matching algorithms for exact string searches. It was conceptualized by Donald Knuth and Vaughan Pratt, and independently by J. H. Morris. They published a paper "Fast pattern matching in strings" jointly in 1977 [13]. The main features of the KMP algorithm are:

- Performs the comparisons from left to right.
- Preprocessing phase in O(m) space and time complexity.

- Searching phase in O(n+m) time complexity (independent from the alphabet size);
- Delay is bounded by log Φ(m), where Φ is the golden ratio and given by

$$\Phi = \frac{1+\sqrt{5}}{2}$$

The KMP string algorithm bypasses the re-examination of previously matched characters by employing the fact that when mismatch occurs, the pattern characters themselves embed sufficient information to determine where the next match would occur. The KMP algorithm reduces the search work of the naive method in two ways: skipping outer iteration and skipping inner iterations. To explain both, the pattern search example described in Section 4.1 is extended further as described next.

4.3.1 Skipping Outer Iterations

Some iterations can be skipped for which no match is possible. For example, if a partial match is found in an iteration, it should be overlapped with the new match to be found. As shown in Table 4.1, iteration 2 has a mismatch at the fourth position (column 6). If the search starts again from column 4 in iteration 3, a conflict in the placement of the characters is found and a mismatch occurs. Iterations 2 and 3 are shown in Figure 4.1.

***i=2: a b a***
***i=3: a b***

**Figure 4.1: Iteration 2 and 3**

It is known from iteration i=2 that T[3] and T[4] are ‘b’ and ‘a’, so they cannot match with ‘a’ and ‘b’ respectively, which iteration i=3 is trying to find. Positions then can be skipped until no conflict is found. As shown in Figure 4.2, the first pattern character ‘a’ in iteration 4 coincides with text character ‘a’.

***i=2: a  b  a***
***i=4:  a***

**Figure 4.2: Iteration 2 and 4**

The overlap of two strings *x* and *y* is the longest word that is a suffix of *x* and prefix of *y*. The number of iterations that can be skipped is the largest overlap in the current partial match. Figure 4.3 shows the pseudo code for string matching with skipped iterations [16]. Two loops 'while' and 'for' are used for pattern search. The outer ***while*** loop is for iteration and the inner ***for*** loop for pattern character comparison with the text at any iteration *i*. If a mismatch at any position *j* is found, the iterations for the overlapped characters are skipped.

```
i=0;
while (i<n)
{
        for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++);
         if (P[j] == '\0') found a match;
                  i = i + max(1, j-overlap(T[0..j-1],P[0..m]));
 }
```

**Figure 4.3: Pseudo code with skipped outer iteration**

4.3.2 Skipping Inner Iterations

Some iterations in the inner loop can also be skipped. As in the previous example in which iterations from i = 2 to i = 4 were skipped, the overlap of text character 'a' with pattern character ('a') has already been tested in the second iteration and should not be tested again in the fourth iteration. Every time an overlap occurs with the last partial match, testing a number of characters equal to the length of the overlap can be skipped. For example, suppose that text string contains characters "abababca" and the search pattern string is "ababca". If we start search at $0^{th}$ position, the first mismatch would

occurs at the 4th position. We can skip character comparisons in the outer loop by starting the search at the 2nd position in the text string. We can realize that characters "ab" at the 2nd and 3rd position in the text are equal to the characters at the 0th and 1st position in the search pattern and these characters have already been matched in the previous iteration. So we can skip comparing these two characters in the inner loop and restart searching by comparing characters from the 4th position onwards in the text string with the characters from the 2nd position onwards in the search pattern.

The KMP algorithm utilizes these two key ideas to increase the efficiency of the string search [16]. It computes, for each position *j* in the pattern, the longest prefix that is also a suffix of the first *h* characters of the same pattern. This information is stored in an integer array often referred to as function $\boldsymbol{\pi}$. This function is independent of the text (the string of characters from which the pattern is searched) and can be computed using the pattern only.

The information stored in the $\pi$ function can be represented by a state machine. Figure 4.4 shows the state transition diagram for pattern "ababca". Each node in the state diagram represents a character in the pattern and transition arrows are labeled with match and mismatch: a transition arrow connected from any node *j* to node *j+1* for match or a backward arrow to the overlap node for mismatch. For this pattern the calculated array would be

$\boldsymbol{\pi}$ [i]={***0, 0, 0, 1, 2, 0}*.**

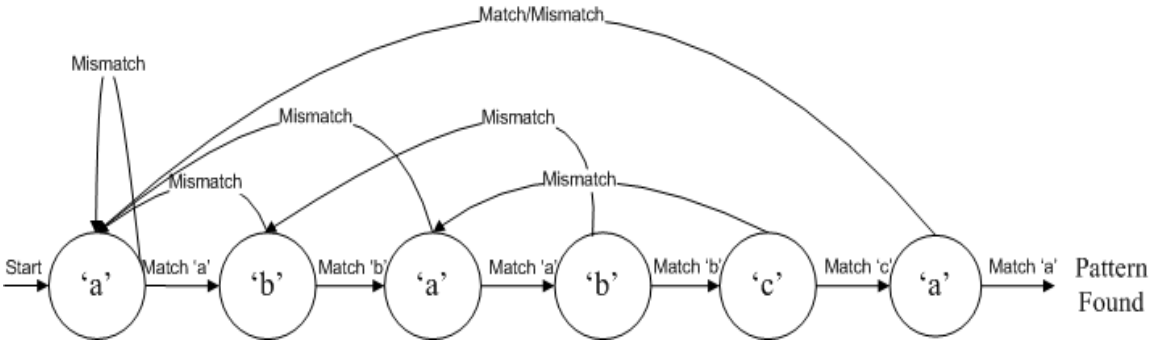


**Figure 4.4: State transition diagram for pattern "ababca"**

A string search with the KMP algorithm is done in two phases. In the first phase, the π function is computed based on the search pattern. In the second phase, the π function, computed in the first phase, is used to speed up the pattern search. For each search pattern, the π array is computed and utilized during the pattern search. At each step of the pattern search, a matcher moves from index *q* in the pattern to index *q*+1 if a match is found or else moves backward to the node π[q] as connected by the transition arrow from node *q*. The search execution for pattern "ababca" is shown in Figure 4.5. The first mismatch in iteration $i = 0$ is found at column $j = 4$ position. Since π[4] = 2, the search in iteration 2 continues by comparing the pattern at character position 2 with text character at column $j = 4$ and results in a pattern match.

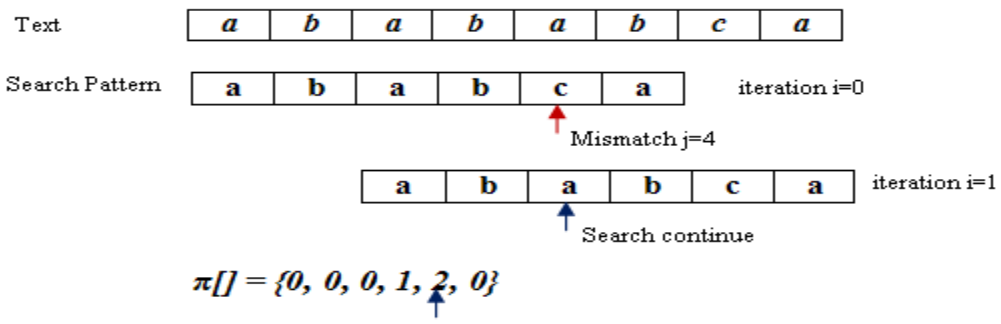


**Figure 4.5: Phase 2 pattern "ababca" search execution using π[] array**

The algorithms for both phases are listed in Figure 4.6 and Figure 4.7, respectively. The pattern to be searched is stored in array P[i] and the text string on which the search is to be performed is stored in array T[]. The function "*ComputePrefix*" computes the π function and stores it in array π[]. The array π[] is used in the procedure "*TextSearch*" to search the given text array T[] for the pattern. It can be proved that the KMP algorithm is very efficient and requires only $m+n$ iterations to perform the search [13, 16].

```
Function ComputePrefix(P)
m = length(P);
π[1] = 0;
i = 1, q = 0;
while( i < m) do
        if (P[i] ≠ P[q]) and (q == 0) then
                ++i;
                π[i] = 0;
        else if (P[i] ≠ P[q]) and (q ≠ 0) then
                q = π[q];
        else if(P[i] == P[q])
                ++i; ++q;
                π[i] = q;
        end if;
end while;
```

**Figure 4.6: KMP algorithm phase 1: Prefix function computation [3]**

```
Procedure TextSearch(P,T)
n = length(T); // length of whole text
m = length(P);
π = ComputePrefixFunction(P);
i = 0, q = 0;
while( i < n) do
        if (T[i] ≠ P[q]) and (q == 0) then
                i++;
        else if (T[i] ≠ P[q]) and (q ≠ 0) then
                q = π[q];
        else if (T[i] ==P[q]) and (q ≠ m - 1) then
                i++; q++;
        else if (T[i] ==P[q]) and (q == m - 1) then
                print "match found"
                i++; q++;
       end if
end while
```

**Figure 4.7: KMP algorithm phase 2: Text search [3]**

## 4.4 Summary

String matching algorithms are used to search all occurrences of a pattern within a string of text characters. The KMP string matching algorithm employs the observation that, at a mismatch, the pattern contains enough information to determine the location of the next possible match. It speeds up the string search by skipping the re-examination of previously matched characters. Before search execution, it builds the prefix function table based upon the specific pattern. This table, which can also be viewed as a state machine, is utilized to speed up the search execution. In the proposed implementation, the π function is converted into a state machine and implemented as a reconfigurable FSM. The next chapter describes KMP hardware implementation in detail.

# CHAPTER 5—DESIGN IMPLEMENTATION

This chapter describes the tools and techniques used in this research. The system involves hardware/software co-design. The design of both the hardware and software components is discussed in detail.

Xilinx™ provided the EDK 10.1 tool chain for design development [18]. Platform Studio (XPS), a part of Xilinx's tool set, is used for on-chip processor-based hardware logic description and XPS SDK development environment is used for software development. The hardware logic design is modeled in VHDL. FSM construction and reconfiguration is designed in software and coded in the 'C' programming language. The development stages of hardware and software components and their integration to generate the FPGA configuration bit-stream is shown in Figure 5.1.

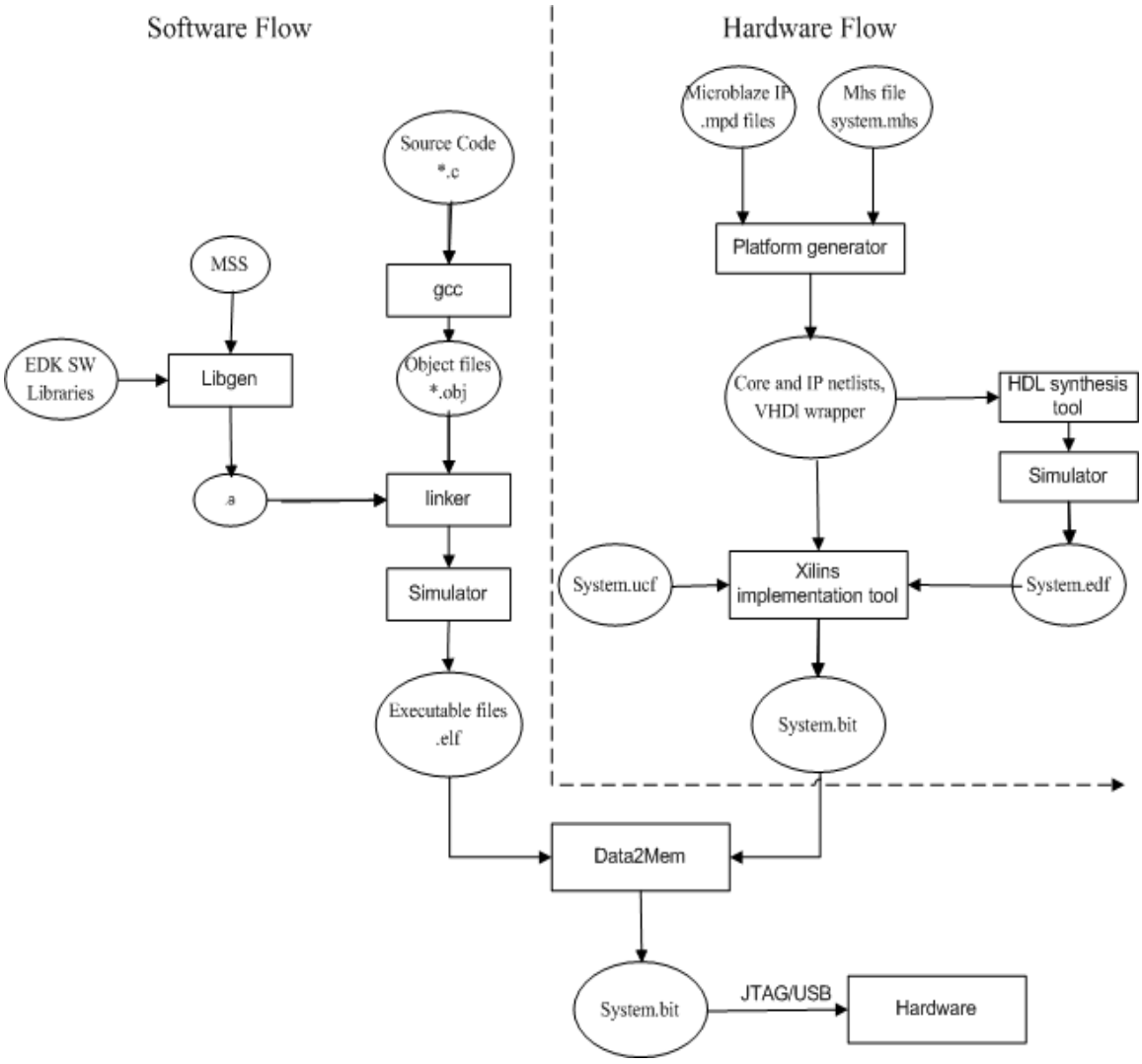


**Figure 5.1: Elements and stages of XPS and EDK leading to FPGA configuration**

The design and implementation of the KMP system is divided into two components: hardware and software. The hardware component involves processor-based system description and hardware implementation of the KMP algorithm as a user intellectual property (IP) core. The software development involves pattern specific $\pi$ function computation, conversion of the $\pi$ function into the FSM, and the software needed to update the FSM.

### 5.1 Design Modeling

The Xilinx EDK tool provides a user-interactive GUI to describe the on-chip processor-based design [19], while the XPS GUI provides options to customize the processor features and peripherals. MicroBlaze® is customized to include a universal asynchronous receiver/transmitter (UART) and LED peripherals. The UART is used to serially communicate with the host processor for test purposes, while the LEDs are used as a debugging tool for self-testing of the board. The Processor Local Bus (PLB) is chosen to integrate the KMP hardware logic with the processor system.

The FSM design for implementing a KMP finite state machine and KMP search execution logic is modeled as two separate VHDL entities. Xilinx ISE 10.1 is used for creating and synthesizing these models.

The Base System Builder (BSB), part of the XPS tool, is used to create the processor-based project [18]. It generates a MHS file (system.mhs) describing the Microprocessor Hardware Specification and a PBD file (system.pbd) representing the schematic view along with several other supporting files. A MSS file (system.mss) specifying Microprocessor Software Specification is also generated. The Import Peripheral Wizard is used to integrate the KMP hardware logic design into the processor system. The wizard creates the necessary directory structure and files needed for development. The HDL template files generated by the wizard provide an interface to hook up the top design entity of KMP logic with the processor system. The wizard also generates a software driver template header and source files to add user software logic to the designed system. These driver files are modified to include KMP phase-I software

logic and FSM creation logic. The driver file for UART communication is modified to add software logic to receive search patterns from a host system and dump debug messages on a HyperTerminal. MicroBlaze system is described as a top module. The KMP design peripheral is imported to the design through the XPS flow. The Xilinx generated software application is modified to access the KMP hardware logic. The developed embedded system is implemented on the FPGA by generating and downloading the bit-stream into the hardware board. Verification is done to prove the functionality through simulation and testing.

## 5.2 Design Implementation

The Xilinx Spartan® 3E Starter board is used for hardware implementation of the design [21]. Figure 5.2 shows a picture of such a board. In this section, hardware and software design of the proposed system is described in detail.

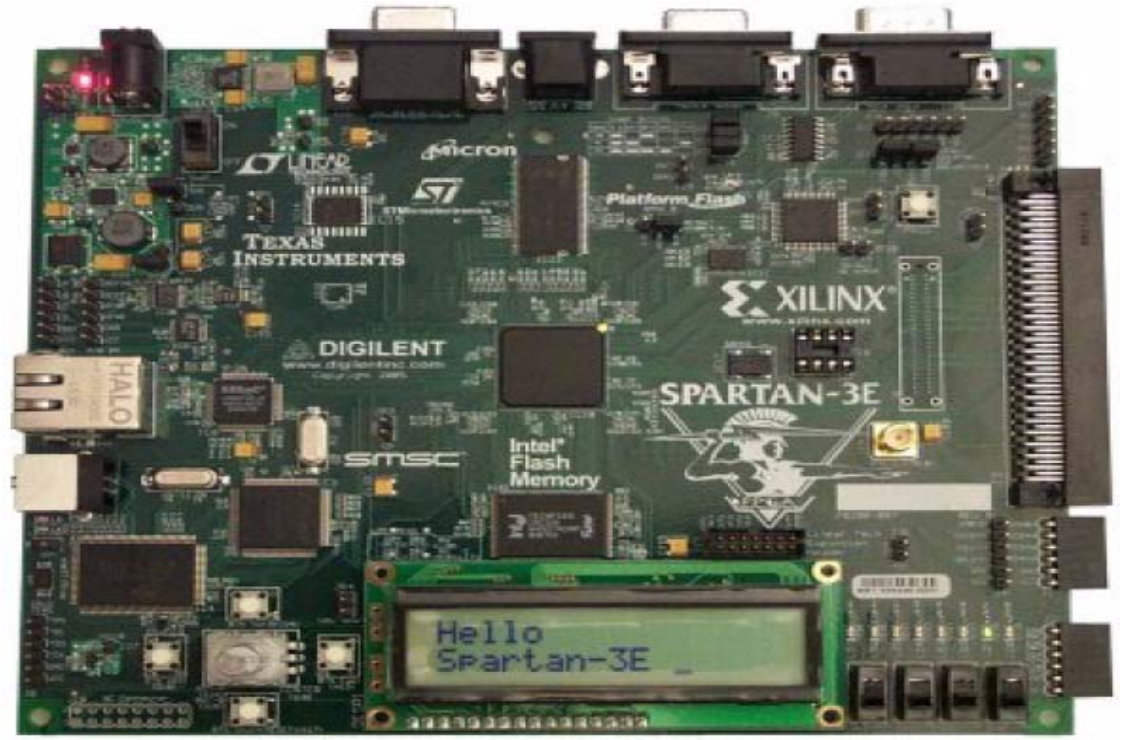


**Figure 5.2: Spartan-3E FPGA Starter Kit Board**

5.2.1 Hardware

A block diagram of the designed system is shown in Figure 5.3. The FSM and KMP logic block constitute the hardware implementation of the KMP algorithm. The KMP hardware is connected to a Processor Local Bus (PLB) via an Intellectual Property Interface (IPIF). The PLB-IPIF provides a bidirectional interface between a user-defined core and the PLB bus. The PLB bus connects peripheral devices to an on-chip processor. The RS232 is used to interface a host PC with the designed system. A customized MicroBlaze® processor core is utilized for receiving a new pattern from the host machine, execute the pattern search, debugging, and displaying the results of the pattern search.

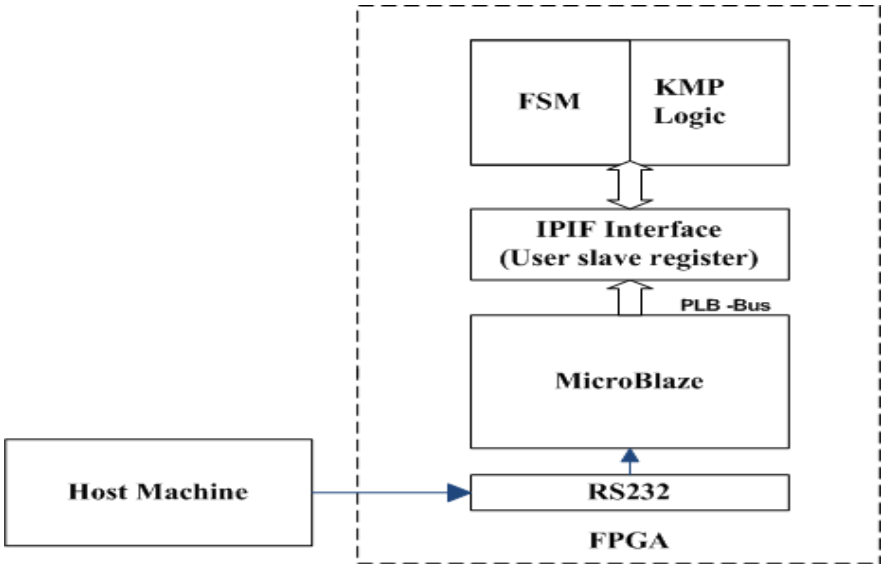


**Figure 5.3: KMP system block diagram**

The hardware implementation of the design is done in two sub phases. First, the RAM-based FSM is realized and tested using a simulation test-bench. Then, the developed FSM is used for implementing the KMP algorithm. In the second phase, the KMP hardware developed in the first phase is integrated with the MicroBlaze system.

The FSM for implementing a KMP finite state machine and KMP search execution logic is modeled as separate VHDL entities. Xilinx ISE 10.1 is used for creation and synthesis of source files.

FSMs are traditionally implemented in FPGA using state register and some combinational logic. The combinational logic receives the input vector and produces the output vector and the next state vectors. The next state vector is stored in the state register. The current state is again fed back to the combinational logic block to determine the next state transition and output vector.

As mentioned earlier, a FSM can be implemented using memory blocks. In the memory-based FSM, state vector ($S_0$, $S_1$, $S_2$… $S_n$) and input vector ($i_0$, $i_1$, $i_2$, …$i_n$) constitute a RAM address vector [5, 24]. The next state is determined by the feedback information: the present state and input vector. For this implementation, embedded block RAM is used for FSM implementation. Two sets of memory blocks, one for storing encoded state transitions (next state function table) and the other for storing the output vector are used. The block diagram for such FSM implementation is shown in Figure 5.4. Memory blocks have dual ports, where one port is synchronous read-write and the other one is synchronous read. The synchronous read-write port is used by the embedded processor to configure the new FSM state transition and output tables into a FSM memory block. A new FSM is constructed to recognize a new pattern. Also, state transition of the old FSM needs to be reconfigured. The other port of each memory block is accessed by the KMP hardware logic to run the KMP algorithm in the search execution phase.

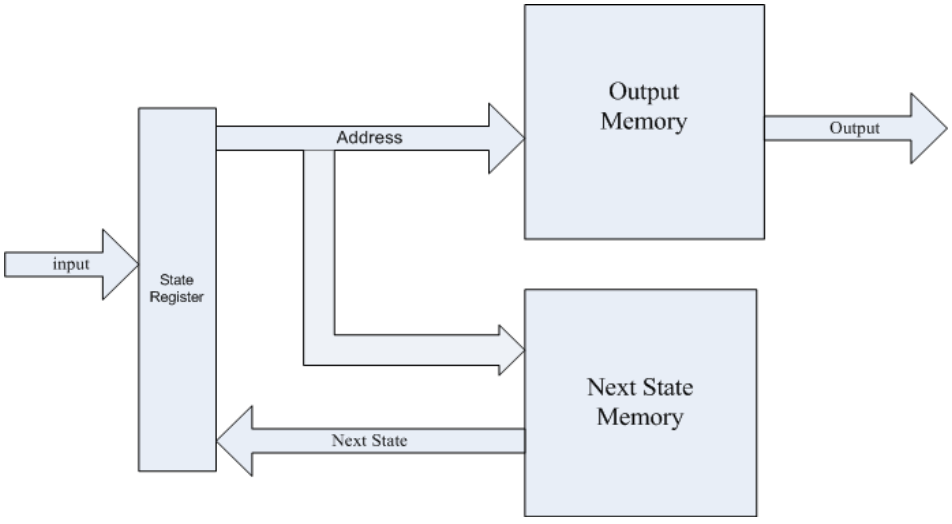


**Figure 5.4: RAM-based FSM implementation**

The FSM is modeled based on the back edges construction (π function). The FSM memory for output function is programmed in such a way that, at any stage of string comparison, the output vector represents the next pattern character. The state vectors are binary encoded to reduce the memory requirement.

As described earlier, the computed prefix function is used to compute the state transition and output functions of the FSM. Consider the pattern "ababca" as an example of which same state transition diagram of Figure 4.4 can be adapted. The calculated prefix function would be ***π[] = {0, 0, 0, 1, 2, 0}***. The length of the prefix function is equal to the pattern length. Table 5.1 shows the translation of the prefix function to the FSM state transition and output functions.

**Table 5.1: Translation from π[i] to FSM next state transition and output function**

| Pattern characters | π[i] | Current State | Next state transition (match = 1) | Next state transition (match=0) | Output function |
|---|---|---|---|---|---|
| a | 0 | 0 | 1 | 0 | a |
| b | 0 | 1 | 2 | 0 | b |
| a | 0 | 2 | 3 | 0 | a |
| b | 1 | 3 | 4 | 1 | b |
| c | 2 | 4 | 5 | 2 | c |
| a | 0 | 5 | 1 | 0 | a |

Column 3 in the table lists all the applicable states that the FSM will traverse if the input text character matches with the pattern characters. Column 4 lists the states the FSM will traverse if the input text character does not match with the pattern characters. Similarly, column 5 in the table lists the FSM output if a match is found between the

input text character and the pattern character, and column 6 lists the output if a mismatch is found. The FSM will traverse through states 0 to state 5 if a match pattern is found and the corresponding output would be the ASCII code of pattern characters. If the FSM reaches state 5 and a match is found, it transits to state 1 and the most significant bit of FSM output signal is set to '1' for one clock cycle to indicate a match is found and the rest of the bits (bits 6 to 0) outputs 0x62, the ASCII code of the second matched character. The signal '*match_addr*' contains a match address that points to the starting location of matched pattern within the text.

The match memory location is calculated by simply subtracting the state value at the current state where match is found from the text memory address counter. The FSM is designed in such a way that at every pattern match, its current state value always is $m - 1$ (pattern length - 1). The match address is then stored in a specified memory location within the block RAM. To keep a count of the number of occurrences of a match pattern, a hardware counter is implemented. The occurrence counter and memory location of match addresses are accessed by the software via user slave registers.

This arrangement avoids the hardware implementation of the $\pi$ function and the need to store the match pattern in internal memory, saving some of the FPGA logic resources. This implementation of FSM require less logic cells since the dual-port RAM block is used for storing the state transition table and output vector table. The state vectors are binary encoded to reduce the memory requirement. This design strategy saves logic cells of the FPGA device for more important sections of the designs.

5.2.2 Hardware Logic Implementation of the KMP Algorithm

The second phase of the KMP search algorithm is realized in the hardware. The algorithm for phase two logic is shown in Figure 4.6 and reproduced again in Figure 5.5. The first three lines of the KMP algorithm calculates the length of the pattern and the prefix function. The length of pattern characters and prefix function is determined in software by the on-chip processor. The 'while' loop for pattern search (lines 5-16 in the code snippet) is translated into hardware logic. The algorithm uses two counters: '*i*' to point current accessed characters position in the text array and '*q*' to point current accessed character position in the pattern during search. The counter '*i*' is implemented in the hardware. The counter '*q*' is implemented implicitly in the form of an FSM state transition. As the search progresses, the FSM outputs pattern characters stored in the FSM's output memory block and changes states based on match or mismatch.

```
Procedure TextSearch(P,T)
1 :     n = length(T); // length of whole text
2:      m = length(P);
3:      π = ComputePrefixFunction(P);
4:      i = 0, q = 0;
5:      while( i < n) do
6:              if (T[i] ≠ P[q]) and (q == 0) then
7:                      i++;
8:              else if (T[i] ≠ P[q]) and (q ≠ 0) then
9:                      q = π[q];
10:             else if (T[i] ==P[q]) and (q ≠ m - 1) then
11:                     i++; q++;
12:             else if (T[i] ==P[q]) and (q == m - 1) then
13:                     print "match found"
14:                     i++; q++;
15:             end if
16:     end while
```

**Figure 5.5: KMP algorithm phase 2: Pattern search**

The KMP phase 2 hardware logic is realized using an FSM, one comparator, and a small combinational logic block. The block diagram of the hardware logic is shown in Figure 5.6. The comparison of the text characters with pattern characters is done through an 8-bit hardware comparator. The comparator compares the FSM output vector (FSM outputs pattern characters) with the text memory output (text characters) and generates a match signal. The match signal is fed to the KMP combinational logic, which in turn controls the address counter. The address counter implements the counter '*i*' of KMP phase 2 logic and is used as an address to access text character from text memory. KMP combinational logic does not increment the address counter if there is a mismatch between a text character and a pattern character, and the FSM is not in state 0, as mentioned in line 8 (***(T[i] ≠ P[q]) and (q ≠ 0))*** of the KMP phase 2 algorithm. The match signal concatenated with the next state function forms the address vector and is used to access the FSM's state transition and output memory. The search result, which includes address locations of the matched pattern and occurrence count of pattern in text, is stored in internal memory blocks.

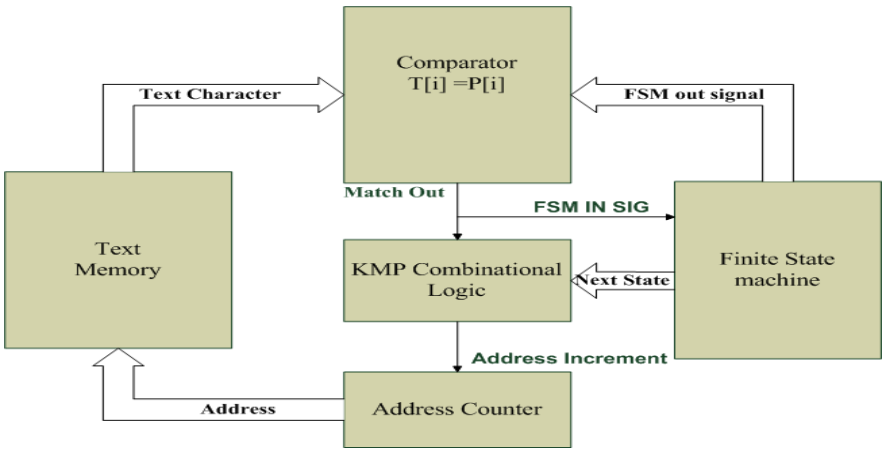


**Figure 5.6: Block diagram of KMP hardware logic**

5.2.3 Processor

As mentioned earlier, the design is implemented on a Spartan 3E FPGA. Since this particular chip does not have a built-in hard-core processor, MicroBlaze soft-core processor is used for receiving a new pattern as an input, back-edge construction, and dynamically reconfiguring the FSM. A MicroBlaze-based embedded system is comprised of a MicroBlaze soft-core processor, on-chip local memory, Standard Bus Interconnects, and on-chip Peripheral Bus (OPB) peripherals.

The MicroBlaze is a 32-bit RISC Harvard-style soft-core processor offered with the Embedded Development Kit (EDK) tool provided by Xilinx to design an FPGA-based system on-chip [19]. It is designed to deliver the highest possible performance on a single FPGA. It is highly customizable according to the application requirement. Processor instructions and local memory data are transmitted on the Local Memory Bus (LMB), which guarantees a single-cycle access to on-chip block RAM.

The MicroBlaze system architecture is shown in Figure 5.7. FPGA's on-chip block memory BRAM is connected to a processor via an Instruction Local Memory Bus (ILMB) and a Data Local Memory Bus (DLMB). An ILMB bus is used to access a processor's instruction cache and a DLMB is used to access a processor's data cache. There are two standard interfaces available to integrate customized IP cores into a MicroBlaze-based system: Processor Local Bus (PLB) and Fast Simplex Link Bus (FSL). The PLB is a part of the IBM Core Connect™ on-chip bus standard. The user core can be connected as a slave or master on the PLB bus. The FSL buses are just FIFOs (first in first out), linked to internal MicroBlaze registers. They act as buffers for point-to-point

data access at high speed. They can be used in time critical applications to provide high speed data transfer. Since the designed system does not require point-to-point data access, the Processor Local Bus (PLB 4.6 bus) is chosen to integrate the customized IP core (KMP logic) with the MicroBlaze processor system.

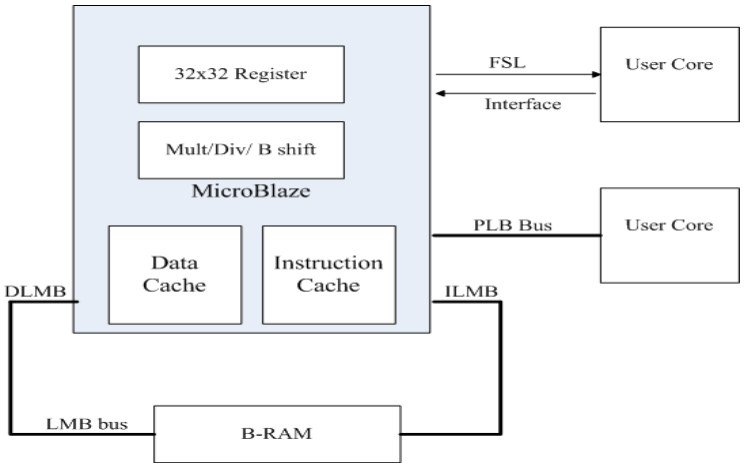


**Figure 5.7: MicroBlaze system with peripheral buses connecting user cores**

The hardware architecture of the implemented design is shown in Figure 5.8. The Processor Local Bus (PLB 4.6 bus) is chosen to integrate the customized IP core (KMP logic). Since the processor is instantiated as a top module in the system, KMP logic is connected to the PLB bus as a slave.

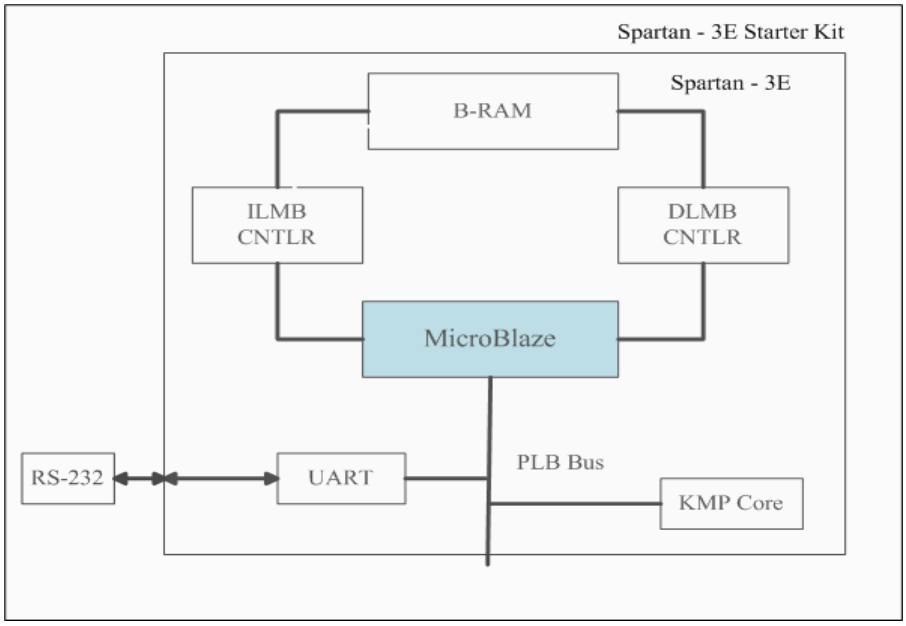


**Figure 5.8: Hardware architecture of the reconfigurable KMP**

The KMP hardware core is designed to be accessed by user accessible 32-bit wide slave registers. The number of slave registers to be used in the design is chosen during the hardware description of a MicroBlaze system [20]. For this implementation, 9 slave registers are used. Table 5.2 lists the usage of each slave register. The processor boot code, software to implement dynamic reconfiguration, and logic to construct a pattern-specific $\pi$ function are stored in the internal block RAM. No external memory is used for this implementation.

**Table 5.2: Slave registers usage description**

| Slave register | Description |
|---|---|
| 0 | Address for FSM next stage memory |
| 1 | Address for FSM output memory |
| 2 | Data for FSM next state memory |
| 3 | Data for FSM next state memory |
| 4 | Data to set FSM output signal width |
| 5 | Data to set FSM output signal width |
| 6 | Control signal for KMP system |
| 7 | Match occurrence count |
| 8 | Match address |

5.2.4 Processor Interface and Control Signals

Interface signals are defined to initiate a pattern-specific system reconfiguration and control search execution. The signals are mapped to bits of slave register 6 and asserted via setting bits. The processor initiates reconfiguration and search execution by asserting these signals. Table 5.3 lists all defined interface signals and their usage for the designed system.

**Table 5.3: Control signal definitions**

| Bit Position | Signal Name | Description |
|---|---|---|
| 0 | Configure | 1 - during FSM update<br>0 - otherwise |
| 1 | write_byte_enable | 1 - during text memory update<br>0 - otherwise |
| 2-3 | X | unused |
| 4 | we_ns | 1 - during state transition function FSM memory update<br>0 - otherwise |
| 5 | we_ns_a | 1 - during state transition memory address update<br>0 - otherwise |
| 6 | we_op | 1 - during output function FSM memory update<br>0 - otherwise |
| 7 | we_op_a | 1 - during output function memory address update<br>0 - otherwise |
| 8 | en_op_mux | 1 - during FSM output vector width setting<br>0 - otherwise |
| 9 | en_in_mux | 1 - during FSM input vector width setting<br>0 - otherwise |
| 10 | X | unused |
| 11 | FSM_reset | 1 - to reset FSM logic |

The processor initiates the reconfiguration process at each reception of a new pattern as follows. Specific signals are activated by the processor to update FSM memory for the next state and output functions. During an FSM update, the *'configure'* signal is activated to indicate a reconfiguration is in progress and the KMP core remains in reset state. After FSM update, the `'configure'` signal is de-asserted, and the KMP search process runs to find the pattern within the text.

To update the FSM memory block storing the state transition function, the processor places the starting address of the next state memory on slave register 0. Then, value 0x31 is placed on slave register 6 to activate the necessary signals for setting the memory starting address for the state transition function. Afterwards, the value 0x11 is

placed on slave register 6 to write-enable the next state memory where memory contents are updated via slave register 2. Table 5.4 lists the corresponding signals and their bit position in slave register 6.

**Table 5.4: Control signal values for FSM state transition memory update**

| Slave Register6 Bit Position | Signal Name | Next State Memory Address Update | Next State Memory Data Update |
|---|---|---|---|
| 0 | Configure | 1 | 1 |
| 1 | write_byte_enable | 0 | 0 |
| 2-3 | x | 0 | 0 |
| 4 | we_ns | 1 | 1 |
| 5 | we_ns_a | 1 | 0 |
| 6 | we_op | 0 | 0 |
| 7 | we_op_a | 0 | 0 |
| 8 | en_op_mux | 0 | 0 |
| 9 | en_in_mux | 0 | 0 |
| 10 | x | 0 | 0 |
| 11 | FSM_reset | 0 | 0 |
| **32 bit hex value** | | **0x31** | **0x11** |

The process of an output function memory update of the FSM is similar to the next state memory update. The processor first places the starting address of output function memory on slave register 1, and then places the value 0xc1 on slave register 6 to assert to the necessary signals. Afterwards, the value 0x41 is placed on slave register 6 to write enable the output function memory and memory contents are updated via slave register 3. Table 5.5 lists the corresponding signals and their bit position in slave register 6.

**Table 5.5: Control signal values for FSM output memory update**

| Slave Register6 Bit Position | Signal Name | Output Memory Address Update | Output Memory Data Update |
|---|---|---|---|
| 0 | Configure | 1 | 1 |
| 1 | write_byte_enable | 0 | 0 |
| 2-3 | x | 0 | 0 |
| 4 | we_ns | 0 | 0 |
| 5 | we_ns_a | 0 | 0 |
| 6 | we_op | 1 | 1 |
| 7 | we_op_a | 1 | 0 |
| 8 | en_op_mux | 0 | 0 |
| 9 | en_in_mux | 0 | 0 |
| 10 | x | 0 | 0 |
| 11 | FSM_reset | 0 | 0 |
| **32 bit hex value** | | **0xc1** | **0x41** |

The FSM is realized as a general purpose one, and the design gives flexibility to control the width of input and output signals. The signal width can be set by asserting appropriate control signals and placing the appropriate width value on slave register 4 (to set input signal width) or slave register 5 (to set output signal width). Table 5.6 lists all of the necessary control signals required to be set and their bit positions in slave register 6 used for setting the FSM input and output vector width. For this implementation, the signal width is set to ‘1’ by placing 0x101 on slave register 6. The output signal width is set to ‘7’ by placing 0x201 on slave register 6, since text characters and pattern are stored in 7-bit ASCII codes.

**Table 5.6: Control signal values for setting FSM output and input signal width**

| Slave Register6 Bit Position | Signal Name | Output Signal Width Setting | Input Signal Width Setting |
|---|---|---|---|
| 0 | Configure | 1 | 1 |
| 1 | write_byte_enable | 0 | 0 |
| 2-3 | x | 0 | 0 |
| 4 | we_ns | 0 | 0 |
| 5 | we_ns_a | 0 | 0 |
| 6 | we_op | 0 | 0 |
| 7 | we_op_a | 0 | 0 |
| 8 | en_op_mux | 1 | 1 |
| 9 | en_in_mux | 0 | 0 |
| 10 | x | 0 | 0 |
| 11 | FSM_reset | 0 | 0 |
| **32 bit hex value** | | **0x101** | **0x201** |

5.2.5 Software Implementation

The EDK tool set has built-in C/C++ compilers to generate the necessary machine code for the MicroBlaze processor. At reception of each pattern, pattern specific Prefix ($\pi$) function is constructed. The algorithm for computing a prefix is shown in Figure 4.5. The algorithm is implemented in 'C'. Since the MicroBlaze system has limited memory, efficient software is written to use less memory and resources. The complete software implementation flow is shown in Figure 5.9.

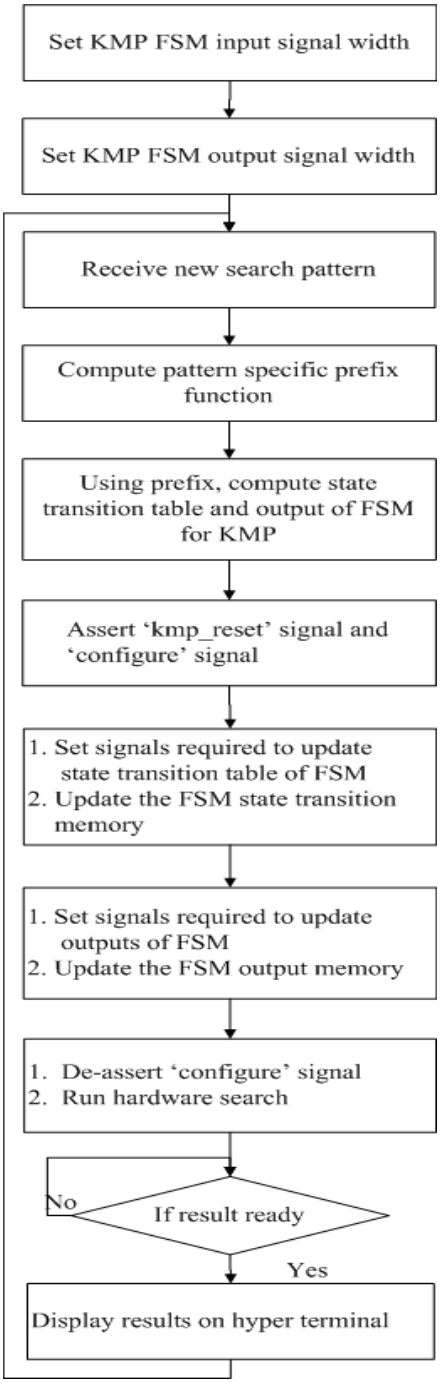


**Figure 5.9: Software implementation flow**

### 5.4 Design Synthesis and Implementation

The Base System Builder (BSB) is used in XPS to create the MicroBlaze-based project. To boot up the designed embedded processor system, both hardware and software components need to be downloaded to the FPGA and program memory, respectively. The XPS Software Development Kit combines the XPS generated hardware bit files with the XPS Software executable file into a `system.bit` file and initializes BRAMs in the bit-stream with the executable code. The generated bit-stream file is downloaded to FPGA using SDK GUI.

### 5.5 Summary

Developing a system that can reconfigure itself without involving a host processor requires an embedded processor to be utilized as a configuration manager. A platform-independent reconfigurable system is developed by employing a reconfigurable FSM. A pattern-specific $\pi$ function is needed to enable the KMP hardware to efficiently search a pattern of characters within a given text string. The $\pi$ function is converted into an FSM in such a manner that embodies the search pattern within it. Thus, configuring the FSM onto an FPGA eliminated the extra step of storing it on FPGA memory. The number of possible reconfigurations of the developed system is only limited to the number of possible write operations on the FPGA memory. Since FPGA can access its internal memory at FPGA clock speed, a significant speed improvement can be achieved in the search execution phase. The next chapter describes the result of various experiments done on the system to assess its accuracy and efficiency.

# CHAPTER 6—EXPERIMENTAL RESULTS AND ANALYSIS

This chapter describes the test procedure used to verify the design functionality. A ModelSim PE® 6.4d is used for simulation [22]. A XPS Software Development Kit is used to program the FPGA board with the configuration bit-stream.

System development is done in incremental steps. At each successive step, test cases are developed and simulation is done to verify the correct behavior. At any step, if any violation from the expected behavior is found, the design entry is modified to rectify the violation and the process is repeated until all design expectations are met.

Initially, after completing the design entry, simulation is done using several test-benches. Once the behavior of each block is verified, the design is further synthesized, and placed and routed for SPARTAN 3E FPGA. Design is further verified by downloading the design on an FPGA board. Xilinx Platform Studio 10.1 is used to generate the configuration bit-stream and the XPS Software Development Kit is used to update the generated bit-stream with the embedded software. The bit-stream is then used to program the FPGA with the developed design. The system under development is debugged via a RS232 HyperTerminal. All the above steps are described in detail in further sections.

## 6.1 Simulation Testing

Simulation testing is done in two phases. First, a designed FSM for KMP logic is implemented and simulation is done to verify the correct behavior, then the design entry

for the KMP hardware search is tested by simulation. After verifying the functionality of the hardware blocks, the design is integrated with the MicroBlaze system. XPS generated a VHDL file (user_logic.vhd) that is used for integrating the designed KMP block with the processor.

Simulation is targeted towards testing the implemented FSM and KMP logic for searching a given pattern from the text. A test-bench is designed to provide various test patterns to the implemented logic. The search patterns are also furnished by the simulation test-bench. Simulation is done for various test patterns of sizes 3 to 20 character lengths. Simulation waveform of a pattern search of one such pattern is shown in Figure 6.1. The search pattern consists of character string "ababca". The ASCII code corresponding to the characters making the pattern is '0x61, 0x62, 0x61, 0x62, 0x63, and 0x61'. The signal *'configure'* is raised until the FSM is updated, then a search is initiated at its de-assertion. As the text characters match with pattern characters, the FSM traverses through states 0 to 5. When the match pattern is found, a MSB of the FSM output signal is set to '1'. As shown in waveform, the FSM output at state '5' is 0xE1 (0x80 | 0x61), Logic operation OR of the logic 1 concatenated with zeros and the ASCII code of the first pattern character. The waveform also shows that the designed logic is capable of searching two consecutive patterns without loss of clock cycles. Signal *'match_found'* is asserted to indicate a pattern match and signal *'match_addr'* points to the location of the pattern within the text. The implemented logic continues searching for the next match.

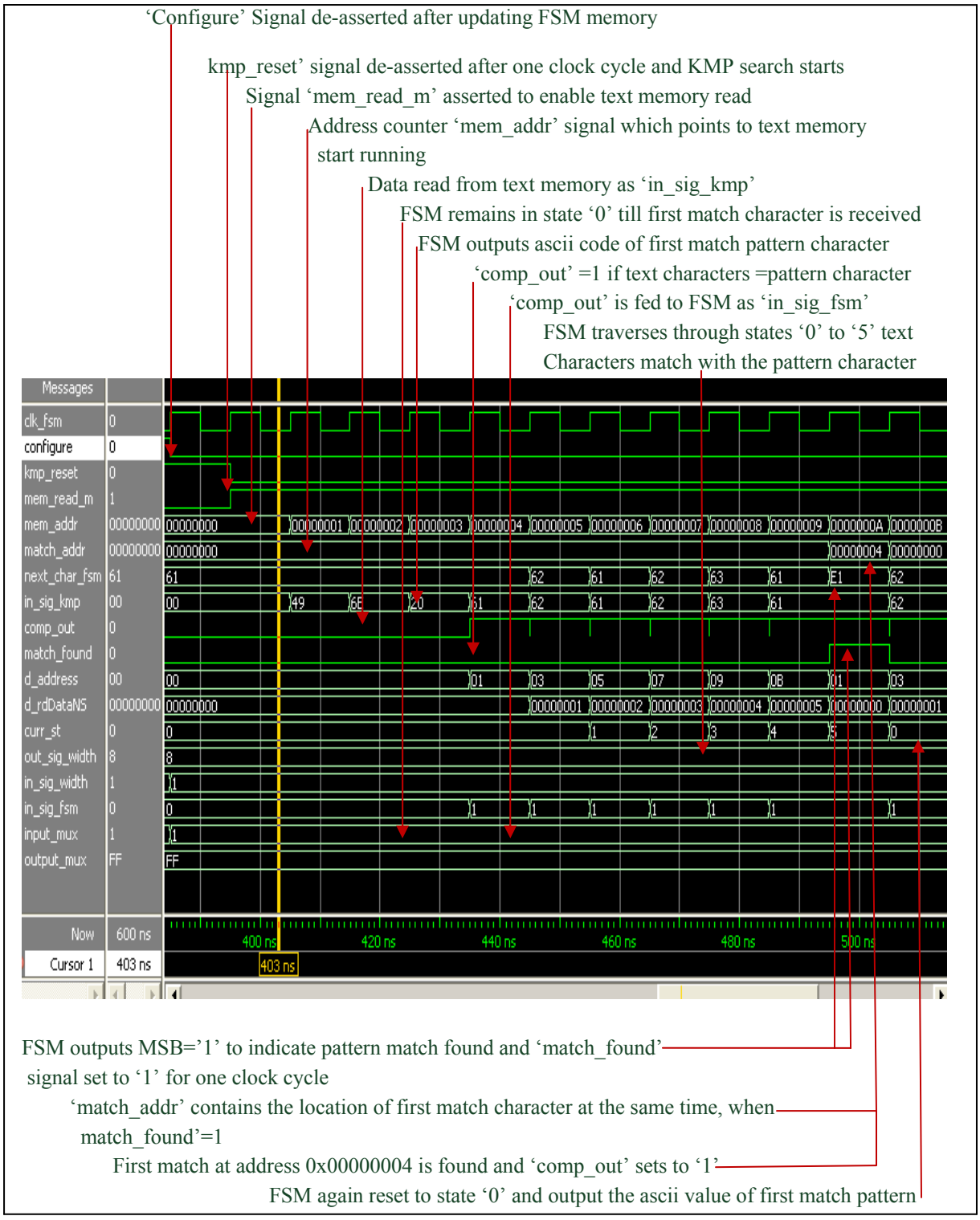


**Figure 6.1: Simulation waveform of KMP search run for pattern "ababca"**

The implemented logic is capable of searching for two consecutive patterns without any loss of clock cycle time. Figure 6.2 shows a simulation waveform of such a search execution. The text string for the test contains “In ababcababce” and the pattern to be searched is “ababca”. The simulation waveform shows that the search execution found two matches at addresses 0x4 and 0x9, which proves that the system can find two overlapped search patterns.

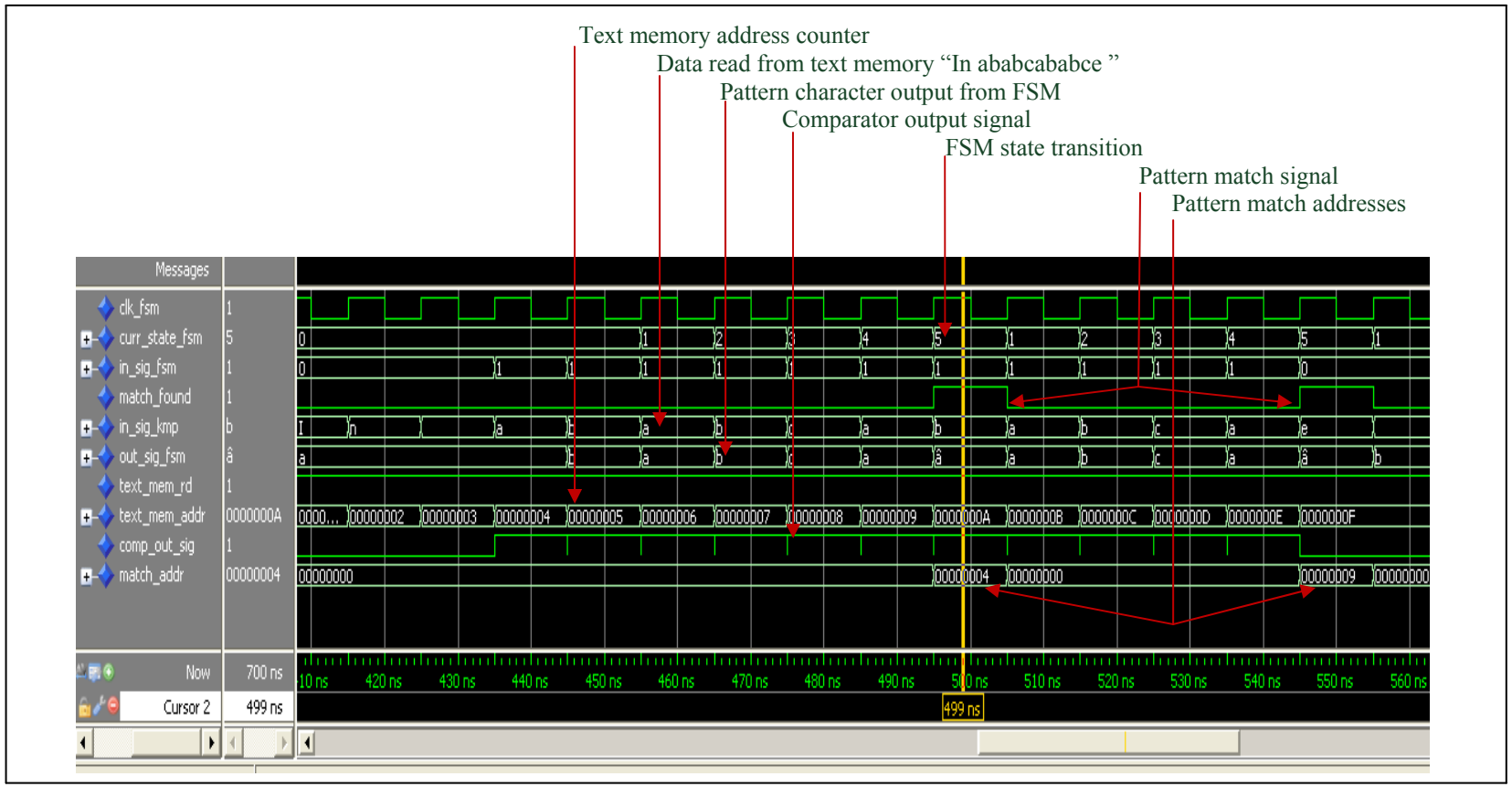


**Figure 6.2: Simulation waveform of KMP search run for pattern "ababca" with text containing two overlapped patterns**

## 6.2 Hardware Testing

To test the pattern search functionality, first a text file needs to be stored either in the board's external memory or in the FPGA's internal memory. For this experimentation, a text file is stored in the FPGA's block RAM. A VHDL source file is coded to instantiate a block 'RAM' entity using a 'RAMB16_S9' tool construct. Each FPGA device has two types of RAM: Block RAM and Distributed RAM. Block RAM is the dedicated memory inside the FPGA, which can be configured through programming. It does not consume any logic resources of FPGA. Distributed RAM is configured as RAM using FPGA logic resources. The 'RAMB16_S9' construct is used to instruct the synthesis tool to use block RAM instead of distributed RAM for implementation. This technique is used to save the FPGA logic resources. This entity instantiates a 2kx8-bit block memory. A software tool written in 'C' takes the text file as input and populates the ASCII code of text characters as an initialization code for the 'RAM' entity. This VHDL file is compiled and loaded to the FPGA along with the design source file. This procedure is followed to eliminate the need for storing and accessing external memory for testing.

An application, written in 'C', is developed to facilitate communication of the designed system on-chip with the host machine. This application used the UART peripheral of MicroBlaze® to establish serial communication with the host machine via HyperTerminal. It receives search commands and search patterns and outputs the search results back on HyperTerminal.

The FPGA board is programmed with a DSK menu command and the host system is connected to the board via UART using a USB-to-serial converter. The pattern to be

searched within the text is furnished by typing it on the HyperTerminal. The embedded software running on MicroBlaze® receives the pattern characters. It reconfigures the FSM for each instance of received pattern and runs the search. It then accesses the search results via user slave registers and then prints back the search result, count of pattern occurrences, and start locations of each pattern within the text on the HyperTerminal port.

Test results are verified using the 'Microsoft Word' application program's utility *'word count'*. Testing is done for various test patterns of sizes 3 to 20 character. A screen shot of the HyperTerminal showing results from one of these searches is shown in Figure 6.3.

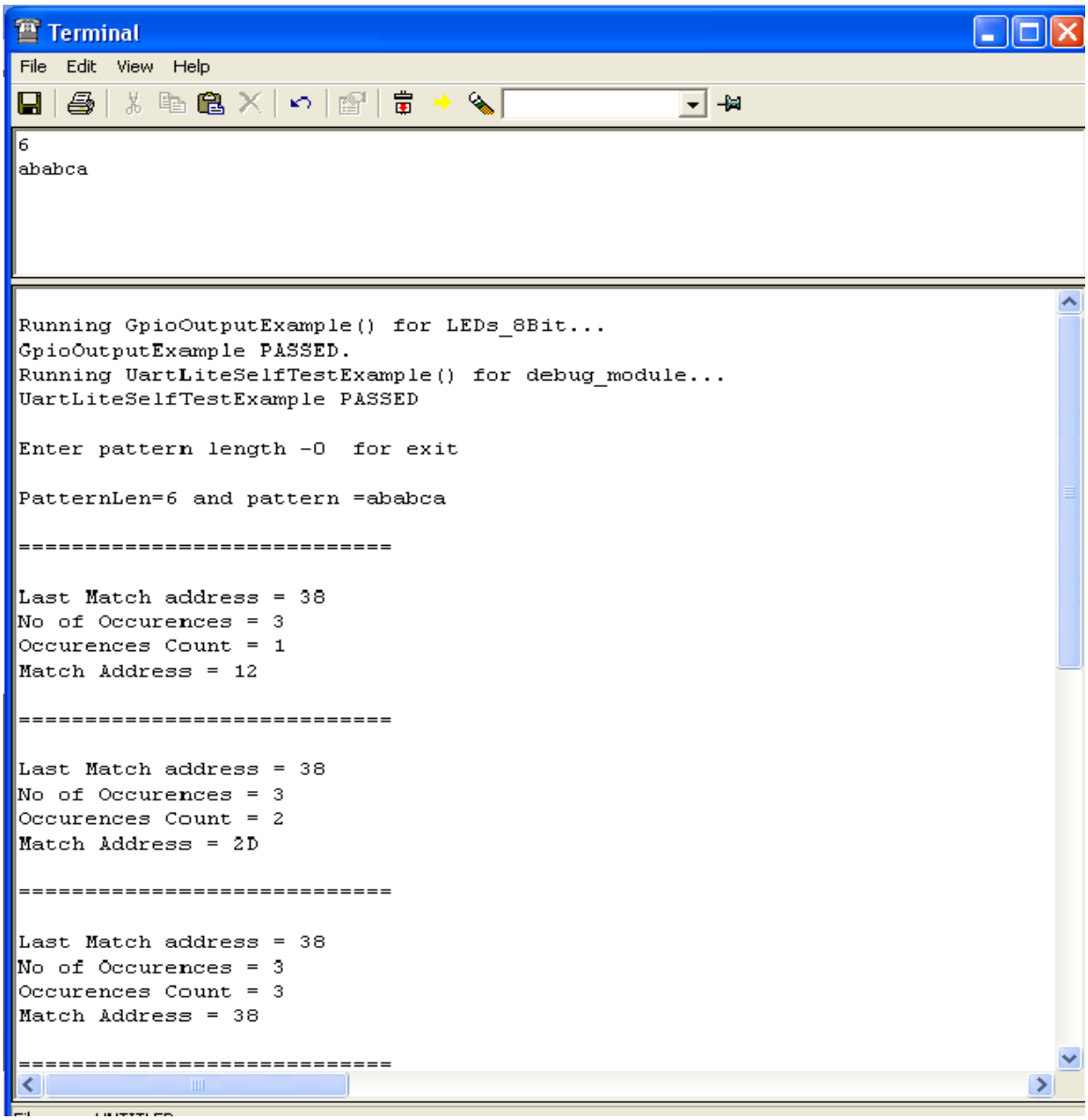


**Figure 6.3: Result of pattern search "ababca" at the terminal console**

The KMP algorithm always requires $n+m$ operations in the worst case where $m$ is the length of the pattern and n is the length of the text. Experimental results show that, with the proposed design, the number of search iterations in phase 2 search is translated into the number of clock cycles. A number of tests were executed with pattern lengths of

3 to 20 characters. In every case, the number of clock cycles to execute a search is always equal to the number of search iterations. The relationship between the number of characters and clock cycles after the first match is found is shown in Figure 6.4.

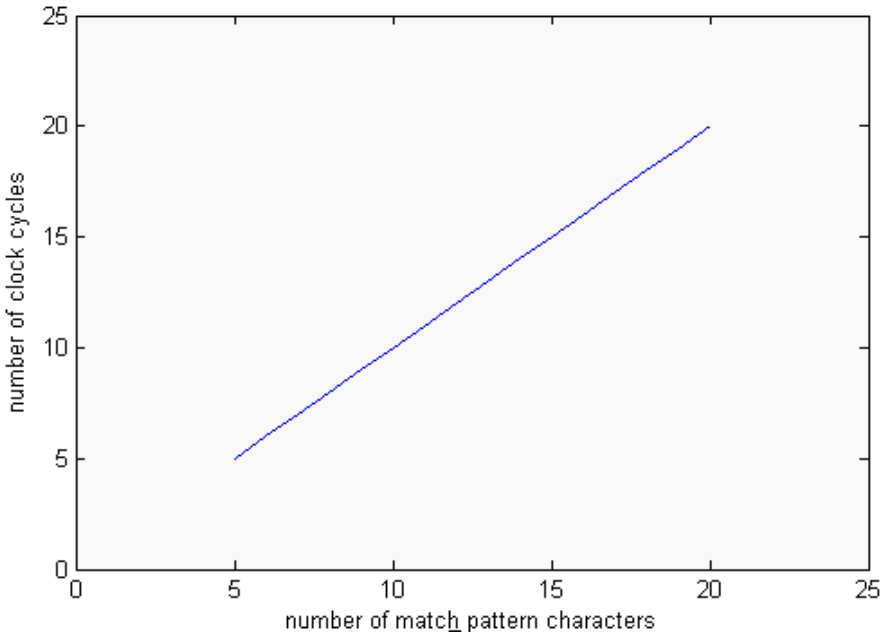


**Figure 6.4: Clock cycles vs pattern length after first match**

The time required for computing the state transition table and output vector table for the KMP finite state machine depends upon the software implementation technique and the number of clock cycles needed for execution. The time required to reconfigure the KMP finite state machine on hardware logic depends on the PLB bus communication speed. With the 'C' implementation using XPS SDK tool set, approximately 300 clock cycles are required to update the state transition function and the same amount of clock cycles to update the output function for a pattern of five characters length. The number of

clock cycles required increases by 50 cycles per increase of pattern character. These results are summarized in Table 6.1.

**Table 6.1: Clock cycle required for FSM update**

| S. No. | FSM Update function | Clock cycles |
|---|---|---|
| 1 | State transition function | 300 |
| 2 | Output function | 300 |
| 3 | Clock cycle increase per character | 50 |

The clock cycle time depends only on the target FPGA device and is independent of the pattern size, as oppose to the implementation described in [3] and reproduced in Table 6.2. The table shows the result of the search execution of pattern length $m$ within the text of $n=10^4$ characters long. Column 1 lists length of test patterns, column 2 lists the clock cycle time, column 3 and 4 ($T_M$+$T_{ME}$) lists the time required for mapping of new configurations on the hardware. $T_E$ is the search execution time in phase 2.

**Table 6.2: Performance of the implementation for various values of m with $n=10^4$ [3]**

| $m$ | $t_{clk}$ | $T_M$ | $T_{ME}$ | $T_E$ | Total time |
|---|---|---|---|---|---|
| 4 | 81.6 ns | 3.7 $\mu$s | 0.7 $\mu$s | 1428 $\mu$s | 1432 $\mu$s |
| 8 | 97.6 ns | 9.0 $\mu$s | 2.1 $\mu$s | 1830 $\mu$s | 1841 $\mu$s |
| 16 | 129.6 ns | 22.4 $\mu$s | 5.8 $\mu$s | 2511 $\mu$s | 2539 $\mu$s |

The performance of the implemented design is compared with the multi-context FPGA implementation mentioned in the literature [3]. Table 6.3 lists the reconfiguration and search execution times for various values of pattern length $m$ and text size of $10^4$

characters. Column A lists the performance with multi-context FPGA and column B list the results using the proposed approach. The time required for prefix computation and translation depends on the clock speed of the MicroBlaze® core and how and in which language the software is written. Sameer Wadhwa and Andreas Dandalis verified that the maximum achievable clock frequency is 110 MHz for a pattern size of 6 characters on Xilinx Virtex series FPGAs [4]. The maximum achievable frequency with the proposed approach is independent of pattern size and is 97.656 MHz for a SPARTAN 3E 500 FPGA. Higher speeds can be achieved with more advanced FPGAs. It is noticeable that through memory-based FSM reconfiguration, a significant improvement of performance can be obtained.

**Table 6.3: Performance comparison for various values of m with $n=10^4$**

| **Match Pattern length(m)** | **Clock Frequency $T_{CLK}$(ns)** | | **FSM reconfiguration time $T_{ME}$(μs)** | | **Phase 2 search execution time $T_E$(μs)** | | **Total Time (μs)** | |
|---|---|---|---|---|---|---|---|---|
| | A | B | A | B | A | B | A | B |
| 4 | 81.6 | 20 | 0.7 | 11 | 1428 | 204 | 1432 | 215 |
| 8 | 97.6 | 20 | 2.1 | 18 | 1830 | 208 | 1841 | 226 |
| 16 | 129.6 | 20 | 5.8 | 34 | 2511 | 216 | 2539 | 250 |

This technique requires less hardware area, as opposed to prior implementations discussed in [3] and [4], since it does not need to store the pattern in internal memory. It also reduces reconfiguration time as it only needs to update the FSM for reconfiguration as opposed to implementation ([3] and [4]), which requires an update of pattern memory and back-edge lookup memory. Since an on-chip processor is used for reconfiguration,

the host system is not required for generating bit-streams. The dynamic loading of bit-stream is also avoided in this scheme.

# CHAPTER 7—CONCLUSION AND FUTURE WORK

A new approach to FSM-based reconfigurable hardware is presented. The FSM is reconfigured on-the-fly by altering the memory contents using an on-chip processor. This approach of reconfigurable FSM is applied to implement a reconfigurable SoC for a pattern matching algorithm on hardware. The KMP phase 1 algorithm computes a pattern-specific prefix and stores it in array $\pi$. This array is used to form the state transition and output vector tables of an FSM. The FSM is utilized in a search execution phase. At any execution step, the FSM outputs the pattern character to be compared with text character. Reconfiguration is initiated by the on-chip processor at each reception of a new search pattern. Software is written to receive a new search pattern from the host system via HyperTerminal and computes its specific $\pi$ required to form the FSM. The on-chip processor is used to reconfigure the FSM implemented on the hardware by updating the state transition and output vector tables with the computed values. The design functionality was verified using simulation and tests were run on actual hardware implementation.

Results show that the implemented design increased the performance of a pattern matching application since search iterations ran at FPGA clock speed independent of the length of the search pattern. Further improvement in the performance can be done only by using an FPGA with higher clock speeds.

Employing an on-chip processor to dynamically reconfigure implemented hardware increases a system's versatility and allows the usage of low-cost FPGAs as a self-reconfigurable platform. Since no FPGA-specific feature is used, the design becomes a platform independent and portable. For example, the proposed design can be implemented on an Altera FPGA using a NIOS soft-core instead of MicroBlaze.

Factors that limit performance improvement in this FPGA-based embedded system are: 1) data transfer rate of the interface between the embedded processor and the configurable hardware block, and 2) memory bandwidth. The most important bottlenecks are the bandwidth and latency of the interface connecting the embedded processor to the user core. The other performance bottleneck is the text memory access speed, if the text is stored in external memory.

Memory size required to implement an FSM increases with the size of input vector, output vector, and number of bits needed to represent the states. Since the size of embedded memory blocks are limited, decomposition-based methods can easily be applied to reduce the memory usage in such systems.

The present implementation of pattern matching searches only for exact pattern matches. Future work can be extended to search for non exact matches. The Boyer-Moore pattern matching algorithm and its variants, which is also an FSM-based algorithm, can be implemented using the proposed reconfigurable FSM.

Another application area for the proposed technique is the efficient implementation of Cryptographic Ciphering algorithms, since these algorithms are FSM-based and can be reconfigured by altering the FSM.

# BIBLIOGRAPHY

[1] G. Estrin, B. Bussell, R. Turn, J. Bibb, "Parallel Processing in a Restructurable Computer System", *Electronic Computers, IEEE Transactions*, Volume: EC-13, Issue: 5, Publication Year: 1964, Page(s): 649 – 649, Volume: EC-12, Issue: 6 Publication Year: 1963, Page(s): 747 - 755.

[2] Julien Lallet, Sebastien Pillement, Olivier Sentieys, "Efficient dynamic reconfiguration for multi-context embedded FPGA", *Proceedings of the 21st annual symposium on Integrated circuits and system design*, 2008, Pages: 210-215.

[3] R. P. S. Sidhu, A. Mei, and V. K. Prasanna,"String matching on multicontext fpgas using self-reconfiguration", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 217–226, Monterey, CA, February 1999.

[4] Sameer Wadhwa and Andreas Dandalis, "Efficient Self-Reconfigurable Implementations Using On-chip Memory", *Lecture Notes in Computer Science; Vol. 1896, Proceedings of the Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications,* Pages: 443 - 448, 2000.

[5] Markus K¨oster, J¨urgen Teich, "(Self-) reconfigurable Finite State Machines: Theory and Implementation", *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, Page: 559, 2002.

[6] Graeme Milligan, Wim Vanderbauwhede, "Implementation of Finite State Machines on a Reconfigurable Device", *Second NASAIESA Conference on Adaptive Hardware and Systems,* (AHS 2007) 0-7695-2866-XI07, 2007.

[7] V. Sklyarov, "Reconfigurable models of finite state machines and their Implementation in FPGAs", *Journal of Systems Architecture: the EUROMICRO Journal* 47 (2002) 1043–1064.

[8] Ali Azarian and Mahmood Ahmadi, "Reconfigurable Computing Architecture: Survey and introduction", *Computer Science and Information technology, 2009 IC CSIT 2009, 2nd IEEE International Conference on Digital Object Identifier*: 10.1109.

[9] Eric J. McDonald, "Runtime FPGA Partial Reconfiguration", *IEEE Aerospace and Electronic Systems Magazine*, 2008-0723.

[10] Sad, E.M. Ah& M.K. Abutaleb, M.M, "Optimization of Reconfiguration Transitions for (Self-)reconfigurable FSM Using Decomposition", *Twenty Second National Radio Science Conference* (NRSC), March 1547, 2005, Cairo-Egypt.

[11] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, Prasanna Sundararajaran, "A Self-Reconfiguring Platform", *13th International Field Programmable Logic and Applications Conference (FPL) Lisbon*, Portugal, September 1-3, 2003.

[12] Salih Bayar, Arda Yurdakul, "Dynamic Partial Self-Reconfiguration on Spartan-III FPGAs via a Parallel Configuration Access Port (PCAP)", *Research in Microelectronic*s, 2008.

[13] Donald Knuth, James H. Morris, Jr. Vaughan Pratt, "Fast pattern matching in strings". *SIAM Journal on Computing* 6 (2): 323–350, 1977.

[14] Xilinx, "Virtex-4 FPGA Configuration User Guide", UG071 (v1.11) June 9, 2009.

[15] Christian Charras - Thierry Lecroq, "EXACT STRING MATCHING ALGORITHMS, Laboratoire d'Informatique de Rouen Université de Rouen Faculté des Sciences et des Technique, http://www-igm.univ-mlv.fr/~lecroq/string/

[16] "Knuth-Morris-Pratt Algorithm ICS 161: Design and Analysis of Algorithms Lecture notes for February 27, 1996, http://www.ics.uci.edu/~eppstein/161/960227.html

[17] Joao Canas Ferreira, Miguel M. Silva, "Run-time Reconfiguration Support for FPGAs with Embedded CPUs: The hardware Layer", *Proceedings of the 19th IEEE International Parallel and Distributed,* April 2005.

[18] Xilinx EDK Concepts, Tools, and Techniques, "A Hands-On Guide to Effective Embedded System Design", EDK 10.1.

[19] Xilinx, "Embedded System Tools Reference Manual Embedded Development Kit", EDK 10.1, September 2008.

[20] Rod Jesman, Fernando Martinez, Vallina Jafar Saniie, "MicroBlaze Tutorial – Creating a Simple Embedded System and Adding Custom Peripherals Using Xilinx EDK Software Tools", http://ecasp.ece.iit.edu/mbtutorial.pdf

[21] Xilinx Corp, "Spartan 3E Starter Kit board user Guide", March 9, 2006.

[22] ModelSim® User's Manual Software Version 6.4d.

[23] I. Gonzalez and F.J. Gomez-Arribas, “Ciphering algorithms in MicroBlaze-based embedded systems”, *Computers and Digital Techniques, IEEE Proceedings*, 2006.

[24] Benfano Soewito, Lucas Vespa, Atul Mahajan, Ning Weng, and Haibo Wang, Southern Illinois University,” Self-Addressable Memory-Based FSM:A Scalable Intrusion Detection Engine,” IEEE Network: The Magazine of Global Internetworking, Volume 23 , Issue 1 (January/February 2009).

[25] Naoto Miyamoto and Tadahiro Ohmi, “A 1.6mm2 4,096 Logic Elements Multi-Context FPGA Core in 90nm CMOS”, IEEE Asian Solid-State Circuits Conference November 3-5, 2008 / Fukuoka, Japan.