7-16-2010

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

Nader I. Rafla
Boise State University

Indrawati Gauba
Boise State University

 DOI: 10.1109/MWSCAS.2010.5548558

A Reconfigurable Pattern Matching Hardware Implementation using On-Chip RAM-Based FSM

Nader I. Rafla, Ph.D., P.E.
Electrical and Computer Engineering Department
Boise State University
Boise, Idaho
nrafla@boisestate.edu

Indrawati Gauba
Electrical and Computer Engineering Department
Boise State University
Boise, Idaho
indrawatigauba@u.boisestate.edu

Abstract — **The use of synthesizable reconfigurable IP cores has increasingly become a trend in System on Chip (SoC) designs because of their flexibility and powerful functionality. The market introduction of multi-featured platform FPGAs equipped with embedded memory and processor blocks has further expanded the possibility of utilizing dynamic reconfiguration to improve overall system adaptability to meet varying product requirements. In this paper, a reconfigurable hardware implementation for pattern matching using Finite State machine (FSM) is proposed. The FSM design is RAM-based and is reconfigured on the fly through altering memory contents only. An embedded processor is used for orchestrating run time reconfiguration. Experimental results show that the system can reconfigure itself based on a new incoming pattern and perform the text search without the need of a host processor. Results also proved that each search iteration was executed in one clock cycle and the maximum achievable clock frequency is independent of search pattern length.**

I. INTRODUCTION

Finite State Machines (FSM) is one of the most vital components of sequential digital systems. Reconfigurable FSMs can be defined as a formal FSM, which has the ability to change its output function and/or transition function during operation [1]. Existing techniques of reconfigurations have been researched and shown that it is applicable to any Field Programmable Gate Array (FPGA) platforms [1, 2, 3, 4, 5].

An approach to implement reconfigurable hardware for string matching using Knuth-Morris-Pratt (KMP) algorithm [5] is described in [6]. In this approach, a multi context FPGA is used and a different context needed to be computed for each pattern. It also required direct manipulation of configuration bits by a host processor.

In the proposed approach, the idea of reconfigurable FSM is used to implement the KMP pattern matching algorithm on hardware. An on-chip processor is used for reconfiguration instead of a dedicated hardware logic block. At each reception of a new search pattern, the FSM reconfigures itself to optimize the pattern search. This implementation avoids the need of reset, manipulation of configuration bit-stream, and any configuration memory space.

In RAM-based FSM implementation, a combination of current state and input vectors are used as an address to access FSM memory contents [3]. These contents consist of all possible next state transitions and output vectors. The FSM can be reconfigured by reprogramming the memory with new or updated state transition and output function tables.

With the used of an embedded soft or hard-core CPU, reconfiguration can be done on the fly. The speed of reconfiguration depends on the CPU clock cycle 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 such as partial reconfiguration or multi-context FGPAs [6, 7]. It also avoids the need of a dedicated hardware design for configuration [1].

II. CASE STUDY: KNUTH-MORRIS-PRATT ALGORITHM

There exist several efficient software-based string-matching algorithms for pattern matching. The KMP algorithm is one of the most efficient pattern matching algorithms that use an FSM for search execution [5]. Therefore, it is an ideal candidate for reconfigurable hardware implementation using memory based FSMs.

The KMP algorithm bypasses the re-examination of previously matched characters by employing the fact that when a mismatch occurs, the pattern characters themselves embed sufficient information to determine where the next match would occur. It computes an array, often referred to as the function $\boldsymbol{\pi}$, such that $\pi[h]=j$ where the first j characters of a pattern (P) are the longest proper prefix that is also a suffix of the first h characters of P. The π function is independent of the searchable text and can be computed from the pattern only.

The information stored in π can be represented by a state machine. Figure 1 shows the state transition diagram for pattern "ababca" where each node represents a character in the pattern. An edge is present between any two nodes i and j if $\pi[j] = i$. A transition arrow is connected from any node j to

978-1-4244-7773-9/10/$26.00 ©2010 IEEE

node *j+1* (for match) or backward to the overlap node (for mismatch). The length of the prefix function (π[i]) is equal to the pattern length. For this pattern the calculated array would be π[i]={*0, 0, 0, 1, 2, 0*}.

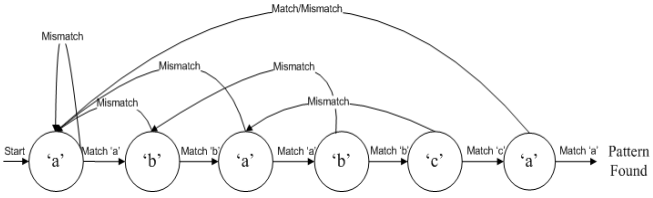

Figure 1. State transition diagram for pattern "ababca"

String search with KMP algorithm is done in two phases. In the first phase the function π is computed from the match pattern. In the second phase it is used to speed up the pattern search. At each step of computation index *q* moves to *q+1* if a match is found or else moves backward to the node π[q].

III. IMPLEMENTATION OF THE KMP ALGORITHM

The reconfigurable hardware design of KMP pattern matching algorithm is implemented using a RAM-based FSM and an embedded processor as shown in Figure 2. The FSM and KMP logic blocks implement the KMP algorithm. The KMP hardware is connected to a Processor Local Bus (PLB) via an Intellectual Property Interface (IPIF). The PLB bus connects peripheral devices to an on-chip soft-core processor MicroBlaze®. This processor is used for receiving new search patterns and performing the pattern specific prefix computation. Result of this computation is used for reconfiguring the FSM. An RS232 interface is used to connect a host PC to the designed system. This host machine is used for entering a new search pattern and for displaying the results of the pattern search on a hyper terminal.

Two Embedded RAM blocks are used for FSM implementation; one for storing encoded state transitions and the other for storing the output vectors. The block diagram for such FSM implementation is shown in Figure 3. Each memory block has dual ports. A synchronous read-write port is used by the embedded processor to access FSM state transition and output data. The other port is used by the KMP hardware logic during the execution of the KMP algorithm.

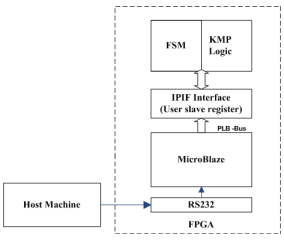

Figure 2. System block diagram

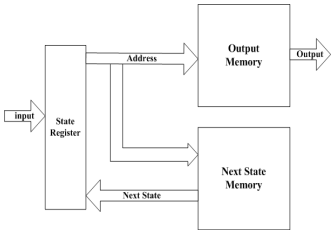

Figure 3. RAM based FSM implementation

The FSM output function is programmed in such a way that at any stage of string comparison, the output vector is the next pattern character to be compared with a text character. The FSM state transition table for the pattern "ababca" is shown in table 1. The calculated $\pi[i]$ values are used to predict next state transition after mismatch. The FSM output vector is 8 bits long where bits 6-0 contain the ASCII code of pattern characters and bit 7 is set to logic 1 only when the pattern is found within the text.

TABLE I. TRANSLATION FROM $\pi[i]$ TO FSM NEXT STATE AND OUTPUT

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	0	0	a

During string search, the text memory address counter increments to fetch the next text character from text memory. The FSM outputs the pattern characters and traverses through states 0 to 5 as the pattern characters matches with the text characters. If FSM reaches at state 5, and a match is found, it transits to state 1. The MSB of FSM output is set to '1' for one clock cycle and the rest of the bits (6:0) contain the ASCII code of the first pattern character. The location of the matched pattern within the text is calculated by simply subtracting the current state value (when match is found), from text memory address counter and decrementing it by one.

To support search of two overlapping patterns as in text string "ababcababca", a modified state diagram is used as shown in Figure 4. The state transition diagram is similar to the previous one except at the final node 'a', the transition arrow connects to second node 'b' for match instead of first node 'a' The FSM state transition table for this FSM remains the same except the last line of the table is modified to traverse the FSM into state '1' instead of state '0' if pattern match is found.

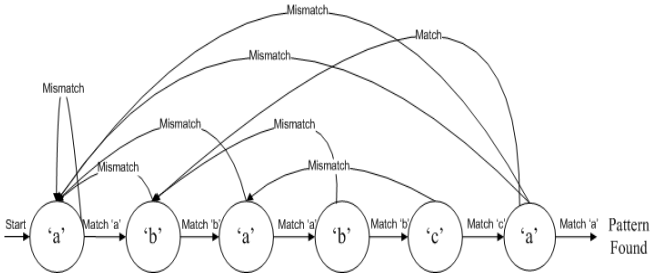

Figure 4. Modified state transitiion diagram for pattern "ababca"

The implementation of the designed system involves hardware/software co-design. The hardware partition consists of processor-based system description and the implementation of KMP algorithm while the software part involves pattern specific prefix function computation, conversion of prefix function to FSM, and software to update RAMS of FSM. Figure 5 shows the KMP hardware implementation. Comparison of text characters with pattern characters is done through a 7-bit hardware comparator that compares the FSM output vector with the text memory output and generates a match signal. This signal is fed to the KMP combinational logic which controls the text memory address counter based on the match signal and current state. The match signal

concatenated with the current state function forms an address vector for FSM memory. Since FSM memory can be accessed at FSM clock frequency, search iterations ran at the same speed. The search result, address locations of the matched pattern, and match count are stored in internal memory blocks.

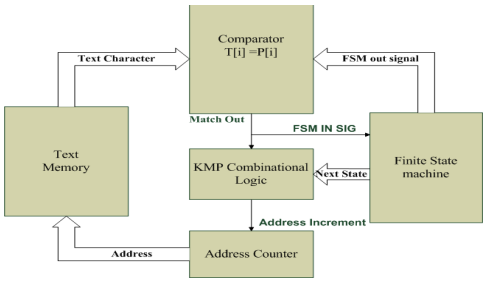

Figure 5. Block Diagram of KMP hardware logic

IV. EXPERIMENTAL ENVIRONMENT

A Xilinx Spartan 3E prototyping board [7] is used for hardware implementation of the proposed design. The EDK 10.1 Xilinx tool chain [10] is used for implementation of the reconfigurable KMP block using VHDL, while the 'C' programming language is used for coding the application that computes the pattern specific prefix function and maps it to an FSM state transition table.

Since Spartan 3E FPGA does not have a built-in hard core processor, a customized soft-core processor (MicroBlaze®) is used instead. It is customized to include UART peripheral for serial communication with a host processor.

The soft-core processor is instantiated as a top module in the system. The system also includes two separate entities for the FSM and KMP phase 2 search. The Processor Local Bus (PLB 4.6) is chosen to facilitate communication among these components. The PLB bus provides a memory-mapped interface to the user core via user accessible slave registers.

The processor boot code, software to calculate FSM state transition table and FSM memory updates, is stored in internal block RAM. No external memory is used for this implementation.

V. EXPERIMENTAL RESULTS

System behavior is verified via simulation using ModelSim PE Student Edition 6.5b [11]. Experimentation is done by generating and downloading the configuration bit-stream onto the Spartan 3E Starter board, and then testing it with various search patterns.

Test bench is designed to provide and generate various test patterns to the designed system. Simulation waveform of the pattern search of *'ababca'* is shown in Figure 6. The ASCII code corresponding to the characters making the pattern is '0x61, 0x62, 0x61, 0x62, 0x63, and 0x61'. The signal `'configure'` is raised until FSM is updated then search is initiated at its de-assertion. As the text characters match with pattern characters, FSM traverses through states 0 to 5. When the match pattern is found, MSB of FSM output signal is set to '1'. As shown in waveform, FSM output at state '5' is 0xE1 (0x80 | 0x61), Logic operation OR of the logic 1 concatinated 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 pattern within the text.

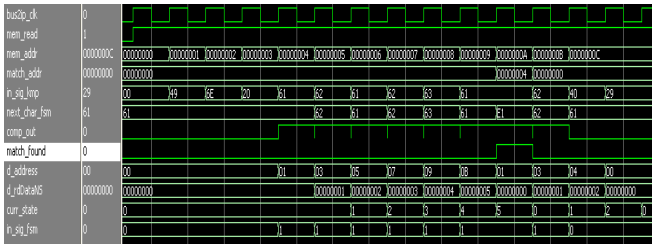

Figure 6. Simulation waveform of KMP search run for pattern "ababca"

The implemented logic is also capable of searching for overlapped patterns without any loss of clock cycle time. Figure 7 shows simulation waveform of such search execution. The text string fused for testing contains the pattern "ababcababce" and pattern to be searched is "ababca". The simulation waveform shows that search execution found two matches at addresses 0x4 and 0x9, the location of first and second pattern, which proves that system can locate overlapped patterns.

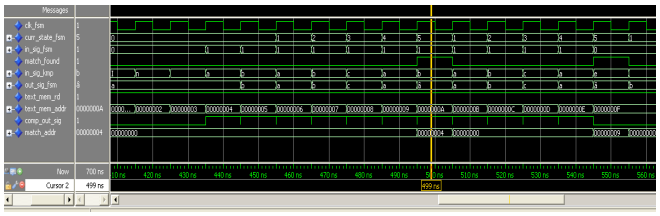

Figure 7. Simulation waveform for two overlapping patterns

To test the pattern search functionality on the physical hardware, a text file is stored in FPGA's block RAM. A VHDL source file is coded to instantiate a block 'RAM' entity using 'RAMB16_S9' tool construct. A 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 into the FPGA along with the design source files. This scheme eliminated the need of storing the text in external memory.

A small 'C' application is developed to establish serial communication between embedded processor and the host machine using UART peripheral of the soft-core. The match pattern is furnished by typing on the hyper terminal. The system prints search results back on the terminal. These results are verified using the 'word count' utility. Testing is done by searching different patterns of various sizes (3 to 20 characters) and for varying number of repetitions.

The string search with KMP algorithm requires, worst case scenario, $m+n$ iterations (m: pattern length ; n: text length) With the proposed design, each search iteration executes within one clock cycle and requires $m+n$ clock cycles, worst case. The design translates the count of search iterations into number of clock cycles.

A number of tests were executed with pattern length varying from 3 to 20 characters. In every case, a linear relationship between number of characters and clock cycles after the first match is found as shown in Figure 8.

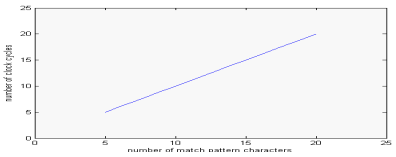

Figure 8. Clock cycles vs pattern length after hitting first match

The time required for computing the state transition and output vector tables of the FSM depends upon the software implementation technique and the FSM reconfiguration time depends on PLB bus communication speed. With the 'C' implementation using XPS SDK tools chain, approx. 300 clock cycles were consumed for updating the state transition and the same amount for the output function for the pattern of five characters length. The number of clock cycles required increases by 50 cycles per increase of a pattern character. These results are summarized in Table II.

TABLE II. 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

Performance of the implemented design is compared with the multi context FPGA implementation described in [6]. Table III lists the reconfiguration time and search execution time for various values of m and text size of 10^4 characters. The T_{CLK} is FPGA clock frequency, T_{ME} is FSM reconfiguration time and T_E is the search execution time. Columns 'A' list the performance with multi context FPGA and columns 'B' correspond to the proposed approach. The time required for prefix computation and translation is not compared since it depends on clock speed of the soft-core processor and the efficiency of software coding. As seen in the table, a significant performance improvement is obtained. The maximum operational frequency with this implementation is independent of the pattern length and is 97.656MHz for SPARTAN 3E 500 FPGA.

TABLE III. PERFORMANCE COMPARISON FOR VARIOUS VALUES OF M WITH N=10^4

m	T_{CLK}(ns)		T_{ME}(µs)		T_E(µs)		Total Time (µs)	
	A	B	A	B	A	B	A	B
4	81.6	20	0.7	11	1428	204	1428	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 other implementations since it does not need to store the pattern in internal memory. It also saves reconfiguration time since it only needs to update the FSM for reconfiguration in contrast with other approaches that require storage of pattern characters and back-edge lookup ($\pi[i]$) onto memory.

VI. CONCLUSION

A new approach to FSM-based reconfigurable hardware for KMP algorithm implementation is presented. The RAM-based FSM is reconfigured on the fly by altering its memory contents using an on-chip processor. The functionality of the designed system was verified using functional simulation and tests that were run on physical hardware.

Results show that the suggested approach increases the performance of pattern matching applications since the used clock frequency does not depend on pattern length. search iterations are also translated into clock cycles. Further improvement in the performance can be achieved by using an FPGA with higher clock speeds.

Employing an on-chip processor to dynamically reconfigure implemented hardware increases the 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 is platform independent and portable.

The present implementation of pattern matching searches only exact pattern matches. This design can be modified to search for non exact matches as well. This approach can also be applied to other efficient FSM-based algorithms.

VII. REFERENCES

[1] Markus Koester and Jürgen Teich, "(Self-)reconfigurable finite state machines: Theory and implementation," in *Proceedings of the International Conference on Design, Automation and Test in Europe Conference and Exhibition*, 2002, pp. 559-566. IEEE Computer Society Press, 2002.

[2] E.M. Sad, M.K Ahmed and M.M Abutaleb, "Optimization of Reconfiguration Transitions for (Self-)reconfigurable FSM Using Decomposition" in *Radio Science Conference, 2005. NRSC 2005. Proceedings of the Twenty-Second National*, 2005, pp.445-454.

[3] Graeme Milligan and Wim Vanderbauwhede, "Implementation of Finite State Machines on a Reconfigurable Device," in *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, 2007, pp. 386 – 396.

[4] V. Sklyarov, "Reconfigurable models of finite state machines and their implementation in FPGAs," in *Journal of Systems Architecture: the EUROMICRO Journal*, 2002, pp. 1043–1064.

[5] Donald Knuth, James H. Morris and Jr. Vaughan Pratt, "Fast pattern matching in strings," in *SIAM Journal on Computing*, 1977, pp.323–350.

[6] R. P. S. Sidhu, A. Mei, and V. K. Prasanna, "String matching on multicontext Fpgas using self-reconfiguration," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1999, pp. 217–226.

[7] Eric J. McDonald, "Runtime FPGA Partial Reconfiguration," in *Aerospace and Electronic Systems Magazine*, *IEEE*, 2008, pp. 10-15.

[8] Xilinx Corp, *Spartan 3E Starter Kit board user Guide*, March 9, 2006

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

[10] Xilinx, "EDK Concepts, Tools, and Techniques A hands on Guide to Effective Embedded System Design,", *UG683 EDK 11, Ver 11.4.*

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

[12] Mentor Graphics Corporation, "ModelSim® User's Manual," *Software Version 6.6b, 2010.*