# Fast Regular Expression Matching using FPGAs

Reetinder Sidhu
sidhu@halcyon.usc.edu

Viktor K. Prasanna
prasanna@ganges.usc.edu

Department of EE-Systems, University of Southern California,
Los Angeles CA 90089

## Abstract

This paper presents an efficient method for finding matches to a given regular expression in given text using FPGAs. To match a regular expression of length $n$, a serial machine requires $O(2^n)$ memory and takes $O(1)$ time per text character. The proposed approach requires only $O(n^2)$ space and still processes a text character in $O(1)$ time (one clock cycle). The improvement is due to the Nondeterministic Finite Automaton (NFA) used to perform the matching. As far as the authors are aware, this is the first practical use of a nondeterministic state machine on programmable logic.

Furthermore, the paper presents a simple, fast algorithm that quickly constructs the NFA for the given regular expression. Fast NFA construction is crucial because the NFA structure depends on the regular expression, which is known only at runtime. Implementations of the algorithm for conventional FPGAs and the Self-Reconfigurable Gate Array (SRGA) are described.

To evaluate performance, the NFA logic was mapped onto the Virtex XCV100 FPGA and the SRGA. Also, the performance of GNU grep for matching regular expressions was evaluated on an 800 MHz Pentium III machine. The proposed approach was faster than best case grep performance in most cases. It was orders of magnitude faster than worst case grep performance. Logic for the largest NFA considered fit in less than a 1000 CLBs while DFA storage for grep in the worst case consumed a few hundred megabytes.

## 1 Introduction

By exploiting the reconfigurability of FPGAs, significant performance improvements have been obtained over other modes of computation for several applications. This paper describes the use of FPGAs for fast regular expression matching. The problem is to find all strings in input text that match the given regular expression. The primary application of regular expression matching is in text search programs such as grep. Other important applications include lexical analysis and DNA sequence matching.

*This research was performed as part of the MAARCII project. This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-99-1-0004 monitored by Fort Huachuca.

The approach proposed in this paper constructs a Nondeterministic Finite Automaton (NFA) and uses it to process text characters. The time and space requirements of the NFA construction algorithm are respectively linear and quadratic in the length of the regular expression. The NFA can then process one text character every clock cycle. To process a text character in constant time on a serial machine requires the construction of a Deterministic Finite Automaton (DFA). DFA construction, in the worst case, requires time and space exponential in the length of the regular expression. The proposed approach significantly reduces time and space requirements by exploiting the reconfigurability and fine-grained parallelism of FPGAs.

Previous work in string matching using FPGAs has primarily focused on searching for a specific string [3] [5]. Attempts at regular expression matching using FPGAs have been based on DFAs and suffer form the same inefficiencies as serial machine implementations.

The following section reviews Finite Automata (FA) and regular expression theory which is used later. Section 3 describes the development of the NFA construction algorithm and its implementation on existing FPGAs. Section 4 describes the Self-Reconfigurable Gate Array (SRGA) and the implementation of the NFA construction algorithm using self-reconfiguration. Performance evaluation results are presented in Section 5 and the conclusion in Section 6.

## 2 Background

Below we informally review regular expressions, NFAs, DFAs, regular expression matching using FAs, and NFA construction from given regular expression. Please see [4][1] for details.

### 2.1 Regular Expressions

A regular expression is a pattern that matches one or more strings of characters. Individual characters are considered regular expressions that match themselves. $|, *, (, )$ are *metacharacters* that which are used as follows. If $r_1$ and $r_2$ are regular expressions, then $r_1|r_2$, $r_1r_2$, $r_1*$ and $(r_1)$ are also regular expressions such that: (1) $r_1|r_2$ matches any string matched by $r_1$ or $r_2$, (2) $r_1r_2$ matches any string a prefix of which is matched by $r_1$ and the rest by $r_2$, (3) $r_1*$ matches any string composed of zero or more strings matched by $r_1$,

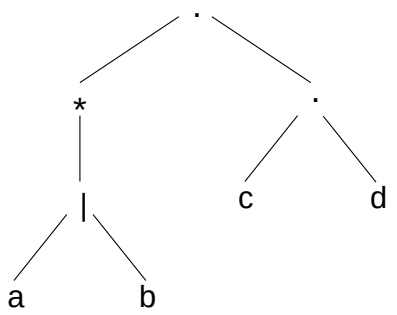


Figure 1: Syntax tree for $((a|b)*)(cd)$.

and (4) $(r_1)$ matches the same strings as $r_1$. Further, $\epsilon$ is a regular expression that matches the empty string. As an example, the regular expression $((a|b)*)(cd)$ matches all strings composed of any sequence of zero or more $a$ and $b$ characters that terminate with the characters $cd$. Figure 1 shows the syntax tree representation of the expression which is used later. An additional metacharacter $\cdot$ is used to denote concatenation in the syntax tree representation.

## 2.2 NFA and DFA

Informally, an NFA is a directed graph in which each node is a state and each edge is labeled with a single character or $\epsilon$. One state is designated the *initial* state and some states are *accepting* or *final* states. A DFA is an NFA with no edges labeled $\epsilon$ and no state has more than one outgoing edge labeled with the same character. An FA (NFA or DFA) processes an input string and either *accepts* it or *rejects* it. A string is accepted if the string characters match the labels on any path of the FA that leads from the initial state to an accepting state.

## 2.3 Regular Expression Matching using FAs

Given a regular expression, it is possible to construct an NFA from it which accepts the same strings that are matched by the regular expression. Such an NFA can therefore be used to process the input text and perform regular expression matching. On a serial machine, an NFA can be constructed from the given regular expression of length $n$ in $O(n)$ time and the NFA takes $O(n)$ memory. The NFA processes a single text character in $O(n)$ time. Thus the total time required to search through text of length $m$ is $O(mn)$ and the memory required is $O(n)$. An alternate approach for faster searching is to construct a DFA from the regular expression which takes $O(2^n)$ time and $O(2^n)$ memory. The DFA can process a text character in $O(1)$ time. Thus the above approach requires $O(2^n + m)$ time and $O(2^n)$ memory. Text searching programs such as grep employ the latter approach but use optimization techniques that in many cases significantly reduces DFA construction time and memory. However, there are always cases for which DFA construction time and memory are both exponential.

In contrast, the proposed FPGA based approach takes $O(n)$ time and $O(n^2)$ memory to construct an NFA which can process a text character every clock cycle. Thus total time required is $O(n + m)$ and the total memory required is $O(n^2)$ in all cases. The following section reviews the procedure for NFA construction from a regular expression. The procedure is used by the proposed approach as described in Section 3.

## 2.4 NFA Construction for Given Regular Expression

Below we review the construction of an NFA which matches the same strings as the given regular expression. NFAs for smaller regular expressions can be used to construct NFAs for bigger regular expressions by applying the following rules. An NFA that matches a single character $(c)$ is constructed as shown in Figure 2(a). NFAs that match the regular expressions $r_1|r_2$, $r_1r_2$, $r_1*$ can be constructed as shown in Figure 2 (b), (c) and (d) respectively. $N_1$ and $N_2$ are NFAs for the regular expressions $r_1$ and $r_2$ respectively. Each NFA has the initial state $q$ and only one final state $f$. The NFA for $(r_1)$ is the same as that for $r_1$. By parsing a regular expression into its constituent subexpressions, and applying the above rules recursively, an NFA can be constructed that matches the same strings as the given regular expression. As an example, Figure 3 shows the NFA constructed using above rules for $((a|b)*)(cd)$.

# 3 NFA Construction using FPGAs

We begin by showing in Section 3.1 how an arbitrary NFA can be implemented in logic. In Section 3.2 we describe simple logic structures that implement NFAs (shown in Figure 2) for a single character, $r_1|r_2$, $r_1r_2$ and $r_1*$. Section 3.3 presents an algorithm that reads a regular expression and uses the above logic structures to implement an NFA that matches the same strings as the given regular expression. Implementation of the above algorithm on conventional FPGAs is discussed in Section 3.4.

## 3.1 NFA Implementation in Logic

There are two standard techniques for implementing DFAs in logic [2]. One way is to use memory that stores the DFA states in a (typically binary) encoded form. Since the memory can look up only one next state every clock cycle, it would be difficult to extend the above technique to efficiently implement an NFA. The other technique is to use the One-Hot Encoding (OHE) scheme. One flip-flop is associated with each state and at any time exactly one flip-flop stores a 1, signifying the current (or active) state. Combinational logic associated with each flip-flop ensures that the 1-bit from the active flip-flop is transferred in the next clock cycle to only

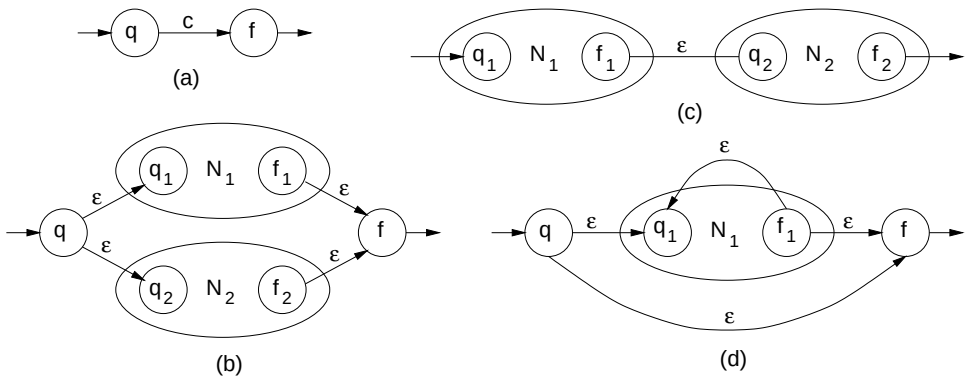


Figure 2: NFAs for (a) single character, (b) $r_1|r_2$, (c) $r_1r_2$, and (d) $r_1*$.

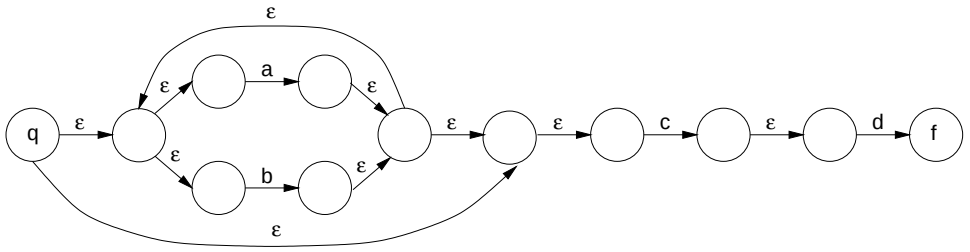


Figure 3: NFA for $((a|b)*)(cd)$.

one flip-flop (the input to the DFA determines which one). Below, we extend the OHE technique to enable direct implementation of NFAs in logic.

As mentioned in Section 2.2, one of the two differences between an NFA and DFA is that a state in an NFA can have more than one outgoing edge with the same label. In implementation, this simply translates into allowing a particular flip-flop output to be connected to the input of more than one flip-flop. The implementation is trivial and enables multiple transitions and multiple active flip-flops every clock cycle. The other differences is that an NFA can have edges labeled $\epsilon$ (empty string) while a DFA cannot. Such an edge transfers the status of its source state to its destination state without waiting for the next character to be processed. In implementation, this can be efficiently achieved by connecting the input (as opposed to output) of the source flip-flop to the input of the destination flip-flop. Figure 4(a) shows a simple NFA and Figure 4(b) shows its implementation using the techniques described above.

## 3.2 Logic Structures for NFA Implementation

The following two observations help understand the logic structures described below: (1) As stated above, and $\epsilon$ transition translates to a connection from the input of the source flip-flop to the input of the destination flip-flop. Therefore, if all transitions from the source flip-flop are $\epsilon$ transitions, the flip-flop does not have to be implemented and can be eliminated. (2) As can be seen from Figure 2, only $\epsilon$ transitions from the accept states of the NFAs are used to construct bigger NFAs. Thus based on observation (1), the flip-flops corresponding to the accept states of the NFAs in Figure 2 do not have to be implemented and can be eliminated.

Figure 5(a) shows the logic structure that implements the NFA (shown in Figure 2(a)) that matches a single character. The flip-flop corresponding to the accept state has been eliminated based on observation (2). The output is 1 only when the flip-flop stores a 1 and the input character matches the character stored in the inside the comparator. Figure 6 shows an efficient implementation of the comparator. Figure 5(b) shows the logic that implements the NFA for $r_1|r_2$ (shown in Figure 2(b)). Both the initial and final state flip-flops are eliminated based on observations (1) and (2) respectively. The only logic required is an OR gate to combine the outputs from $N_1$ and $N_2$. Figure 5(c) shows the implementation of the NFA for $r_1r_2$ (shown in Figure 2(c)). As the latter figure shows, only three edges are required. Thus the logic consists of just three wires. Figure 5(c) shows the logic that implements the NFA for $r_1*$ (shown in Figure 2(d)). As in case of $r_1|r_2$, the initial and final state flip-flops are eliminated. An OR gate is used to combine the two inputs to the initial state of $N_1$. Another OR gate combines the two inputs to generate the accept output.

The above logic structures can be combined to construct an NFA for a given regular expression. To complete the con-

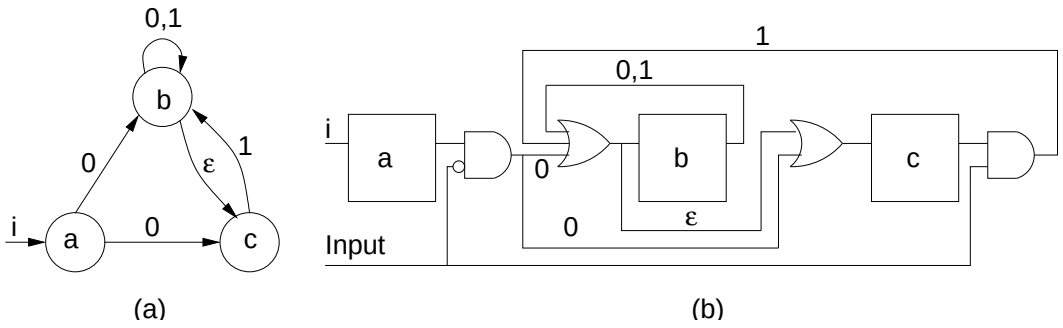


Figure 4: (a) Simple NFA. (b) Implementation in logic.

struction of such an NFA, the i and o ports of the top level logic structure need to be connected as follows. The o port is connected to a flip-flop representing the final state of the NFA. When the NFA processes text characters, a 1 in this flip-flop for any character would signify a regular expression match at that character. An NFA typically processes a single string and indicates if a match occurs or not. However, we require the NFA to match strings beginning at any position in the input text. To do so, from a theoretical point of view, the regular expression $(a_1|a_2|...|a_j)*$ needs to be prefixed to the regular expression of interest—$a_1, a_2, ..., a_j$ are all the characters of the input text. In practice, this can be efficiently implemented by simply setting the $i$ port of the top level logic structure to be permanently high. As an example, Figure 7 shows the implementation using logic structures of the NFA shown in Figure 3.

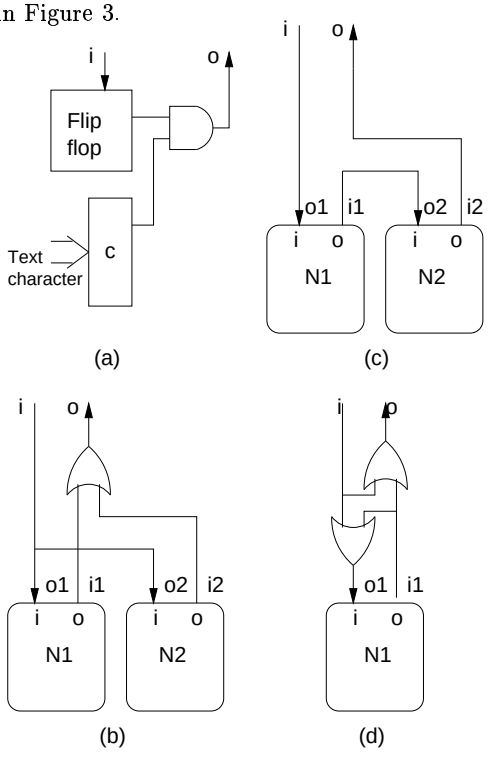


Figure 5: Logic structures for (a) single character, (b) $r_1|r_2$, (c) $r_1r_2$, and (d) $r_1*$.

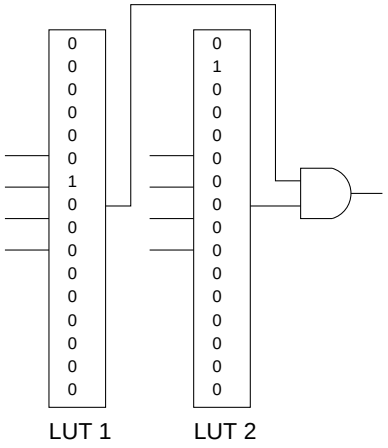


Figure 6: Comparator for ASCII character a (01100001). Output is one only when input to LUT1 is 0110 and to LUT2 is 0001.

## 3.3 NFA Construction Algorithm

Figure 8 shows the algorithm. It accepts the regular expression in postfix form which is easily obtained by postorder traversal of the syntax tree of the regular expression. For example, the postfix form of the regular expression in Figure 1 is $ab|*cd\cdot\cdot$. The postfix form eliminates the need for the ( and ) metacharacters and simplifies the algorithm.

The algorithm uses a stack data structure and relies on the following placement and routing subroutines. place_char, place_|, place_· and place_* respectively place the logic structures for a character, and the metacharacters $|, *, \cdot$. They all return a pointer reference (p) to the placed structure. route1 and route2 create bidirectional connections between the logic structures the pointers to which are supplied as arguments—route1(p, q) connects the o output of q to the i1 input of p and the o1 output of p to the i input of q. Similarly route2(p, q) creates connections to i2 and o2 ports of p. The last two lines of the algorithm create connections to the i and o ports of the logic structure for the last symbol in the regular expression. The o port is connected to the input of a flip-flop and the i port is connected to a permanently high signal.

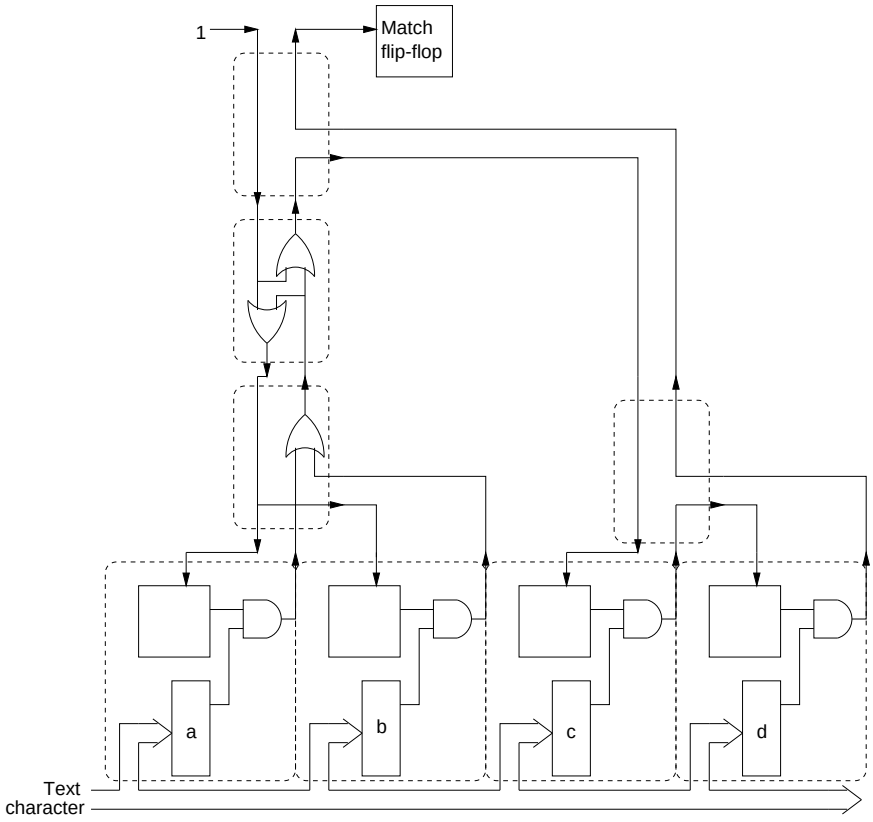


Figure 7: Implementation of NFA for $((a|b)*)(cd)$. Dashed boxes indicate logic structures.

Assuming that the placement and routing subroutines take constant (O(1)) time, the algorithm executes in $O(n)$ time where $n$ is the length of the regular expression. The constructed NFA can process one character each clock cycle (O(1) time). To be able to process one character in O(1) time on a serial machine, a DFA has to be constructed for the input regular expression. Doing so takes $O(2^n)$ time. In practice, optimized DFA construction techniques can be quite fast in many cases but still require exponential time in the worst case. In contrast the proposed approach is very fast ($O(n)$ time) in all cases.

The area of the NFA constructed using the proposed approach would depend upon the placement and routing subroutines. In Section 4.2 we show an implementation of these which can construct an NFA in $O(n^2)$ area. On the other hand, on a serial machine a DFA takes $O(2^n)$ memory. Again, optimized techniques can often reduce the exponential memory requirement but not always. In contrast, the proposed approach requires only $O(n^2)$ in all cases.

## 3.4 FPGA Implementation

We now discuss the implementation of the NFA construction algorithm for an FPGA architecture. The crucial fact to keep in mind is that the NFA mapping time needs to be as small as possible. NFA mapping time is composed of the time required to construct the NFA, generate configuration bits for the NFA logic and configure the FPGA with the generated bits. Reducing this time is important because the NFA can be constructed only at runtime when the user inputs the regular expression. Thus the NFA mapping time is a part of the overall execution time and therefore it should be minimized.

The NFA construction algorithm can be implemented as a program that reads a regular expression and outputs the NFA logic in the form of an HDL description, or a technology mapped netlist, or a placed and routed netlist, or directly in the form of configuration bits. In the interests of minimizing the NFA mapping time, direct generation of configuration bits would be preferable because then the synthesis, placement and routing steps can be bypassed. However, doing so is not practical for following two reasons. The bitstream formats of commercial FPGAs are proprietary. Also, even if the formats were known, any error in generating configuration bits could potentially permanently damage the chip. Thus, the preferred approach would be to implement the NFA construction algorithm as a program that outputs the NFA logic as a placed and routed netlist and use vendor tools for generating configuration bits. For Xilinx FPGAs, another approach would be to write the program in Java and use Jbits [9] for generating configuration bits.

For details on the placement of the logic structures and routing between them, please see Section 4.2 which describes the algorithm implementation for the SRGA architecture. Implementation for other architectures can be done along similar lines.

```
for(i=0; i<regexp_len; ++i)
{
  switch(regexp[i])
  {
    case char: place_char(regexp[i], &p);
               push(p);

    case    |: place_|(&p);
               pop(&p1);
               route1(p, p1);
               pop(&p2);
               route2(p, p2);
               push(p);

    case    .: place_.(&p);
               pop(&p1);
               route1(p, p1);
               pop(&p2);
               route2(p, p2);
               push(p);

    case    *: place_*(&p);
               pop(&p1);
               route1(p, p1);
               push(p);
  }
}
pop(&p);
route_input_high(p);
route_output_ff(p);
```

Figure 8: NFA construction algorithm.

Section 5 presents the performance evaluation results of the above approach for the Xilinx Virtex FPGA architecture. As can be seen from the results, the NFA logic is quite fast, but the NFA mapping time still forms a significant portion of the overall execution time.

# 4 NFA Construction using the SRGA

We describe in the following sections how the NFA mapping time can be further reduced by using *self-reconfiguration*. Using self-reconfiguration, the NFA construction algorithm can be implemented as configured logic on the device itself. The logic reads the input regular expression, directly generates the configuration bits for NFA logic and configures the device with them, all without any external intervention. All three components of the NFA mapping time—NFA construction, configuration bits generation and device configuration—are significantly reduced using self-reconfiguration. We start with a description of the SRGA device.

## 4.1 Self-Reconfigurable Gate Array (SRGA)

In most cases, configuration of an FPGA, whether at compile time or at runtime, is performed externally. Much greater performance gains and a high degree of flexibility can be obtained if the device can generate configuration bits at runtime and use them to modify its own configuration—the ability of a device to do so is what we call *self-reconfiguration*.

The SRGA is a multicontext device [7][8] which, as the name implies, has been designed to support efficient self-reconfiguration. The key device features that allow it to do so are: (1) Single cycle context switching and (2) Single cycle random access to the configuration memory. Using these two features self-reconfiguration can be performed quickly as shown by the simple example below.

The problem is to configure an AND gate but at compile time, we don't know where on the logic cell array the AND gate is to be configured—the location is known only at runtime. Since we cannot configure the AND gate at compile time, we configure on one of the contexts logic which will in turn configure the gate (as shown in Figure 9). At runtime, the coordinates of the logic cell which is to be configured as an AND gate are sent to the SRGA. The logic configured on the device reads these coordinates. It then computes the address of the configuration memory locations corresponding to the logic cell—bits written into these locations control the functionality of the logic cell. The configured logic writes bits that configure the logic cell as an AND gate into the above configuration memory locations. Finally, it switches to the context on which the gate was configured.

Please see [6] for a detailed description of the SRGA device and its capabilities, including row and column routing using self-reconfiguration.

## 4.2 NFA Construction using Self-Reconfiguration

Figure 10 shows the algorithm that reads a regular expression and constructs the corresponding NFA using self-reconfiguration. The algorithm is based on the one shown in Figure 8 and it operates in a similar manner—the basic difference is that the above algorithm explicitly places and routes the NFA logic. Figure 11 shows the simplified block diagram of the datapath and control logic that implements the algorithm. The placement and routing performed by the algorithm is explained below. For clearer understanding, please also see the step by step NFA construction by the algorithm for $((a|b)*)(cd)$ shown in Figure 12.

To perform placement and routing, it is necessary to specify the logic cell in which a particular logic structure is (or is to be) configured. The counters row and col, and the registers row1, col1, row2 and col2 are used for this purpose. Note that the stack pushes or pops a row, column pair in a single operation.

The algorithm places the NFA logic as a binary tree, similar to the syntax tree of the regular expression. At the leaf nodes are the logic structures for the characters and all of them are placed in the same row (row 0). The column is specified by the column counter col which is incremented each time one is placed. Placement in the same row simplifies broadcasting the text characters to the comparators.

Logic structures for the metacharacters form the non-leaf

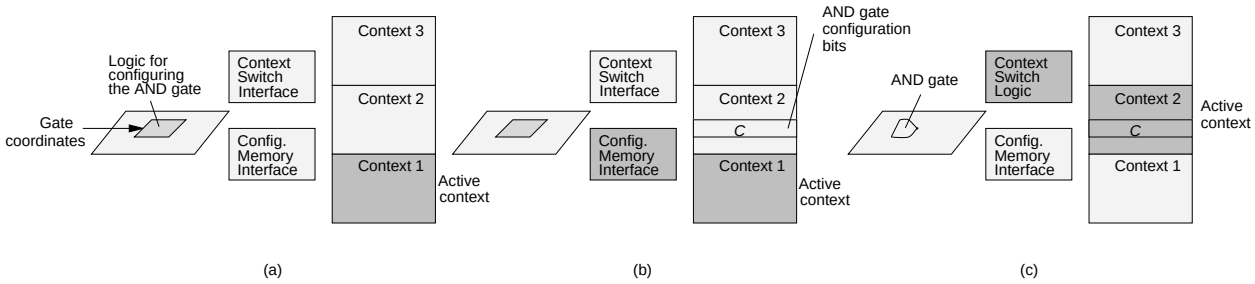


Figure 9: An illustration of self-reconfiguration using the SRGA.

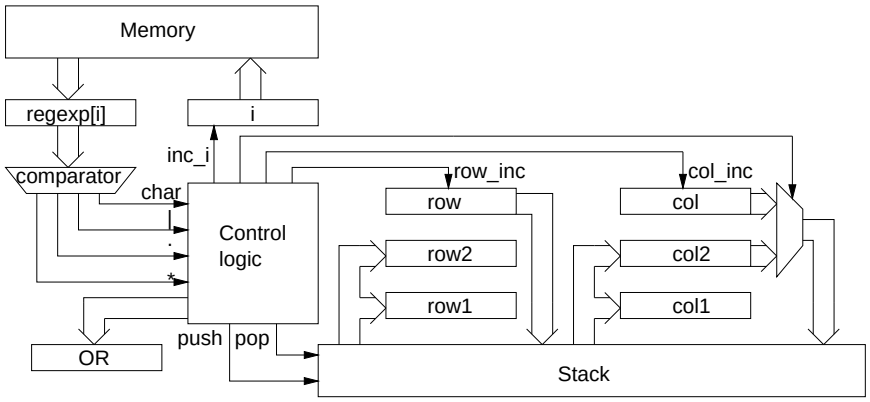


Figure 11: Datapath and control logic for NFA construction algorithm.

nodes are each is placed on a separate row. The row is specified by the row counter which is incremented each time one is placed. Note that the logic structure for one of the child nodes of | or · is always in the preceding row. Thus by placing the logic structure for | or · in the same column as the other child node, connections to the former and latter child nodes require only row and column routing operations respectively. The logic structure for the only child node of $*$ is always in the preceding row. Thus the connection can be established by simply placing the logic structure for $*$ in the same column.

The numbers at the end of each line in Figure 10 indicate the number of clock cycles required to complete the operation on that line. Counter increments occur in parallel with other operations and so take zero clock cycles. As described in [6], any row or column routing operation on the SRGA takes 4 clock cycles.The row_route and col_route operations create bidirectional connections. Also, it takes 2 clock cycles to switch to the context with the routing logic and back to the current context. Thus each routing operation takes 10 clock cycles. The time for the placement operations is composed of the time to configure the logic structure in a logic cell (1 bit per clock cycle) and 2 clock cycles for the context switching. The time for the character logic structure also includes the 32 clock cycles required to configure the comparator as shown in Figure 6.

The above algorithm thus configures the logic structures for char, |, · and $*$ in 47, 45, 37 and 28 clock cycles respectively. We use the above numbers to estimate NFA construction time using the SRGA in Section 5.4.

# 5 Performance Evaluation

In this section, we compare the performance of FPGA implementations based on the proposed approach with the performance of the Unix text search program grep. The comparison criteria are the space needed (memory for software and area for FPGA), and the time required for FA construction and text character processing.

To compare performance, the regular expression we match using both approaches is $(a|b)*a(a|b)(a|b)...(a|b)$ which has $k$ occurrences of $(a|b)$ at the end (henceforth written as $(a|b)*a(a|b)^k$). It matches any sequence of $a$'s and $b$'s in which the $(k+1)$th character from the right is $a$. Any DFA constructed for the above expression would have at least $2^k$ states. We vary $k$ to study performance variation with regular expression length.

## 5.1 Software Performance

The software performance evaluation was carried out on a machine with an 800 MHz Pentium III Xeon processor with 2 GB RAM running Linux (Red Hat 6.2). The regular expression matching program used was GNU grep version 2.4, which is distributed with the OS. The times reported are user

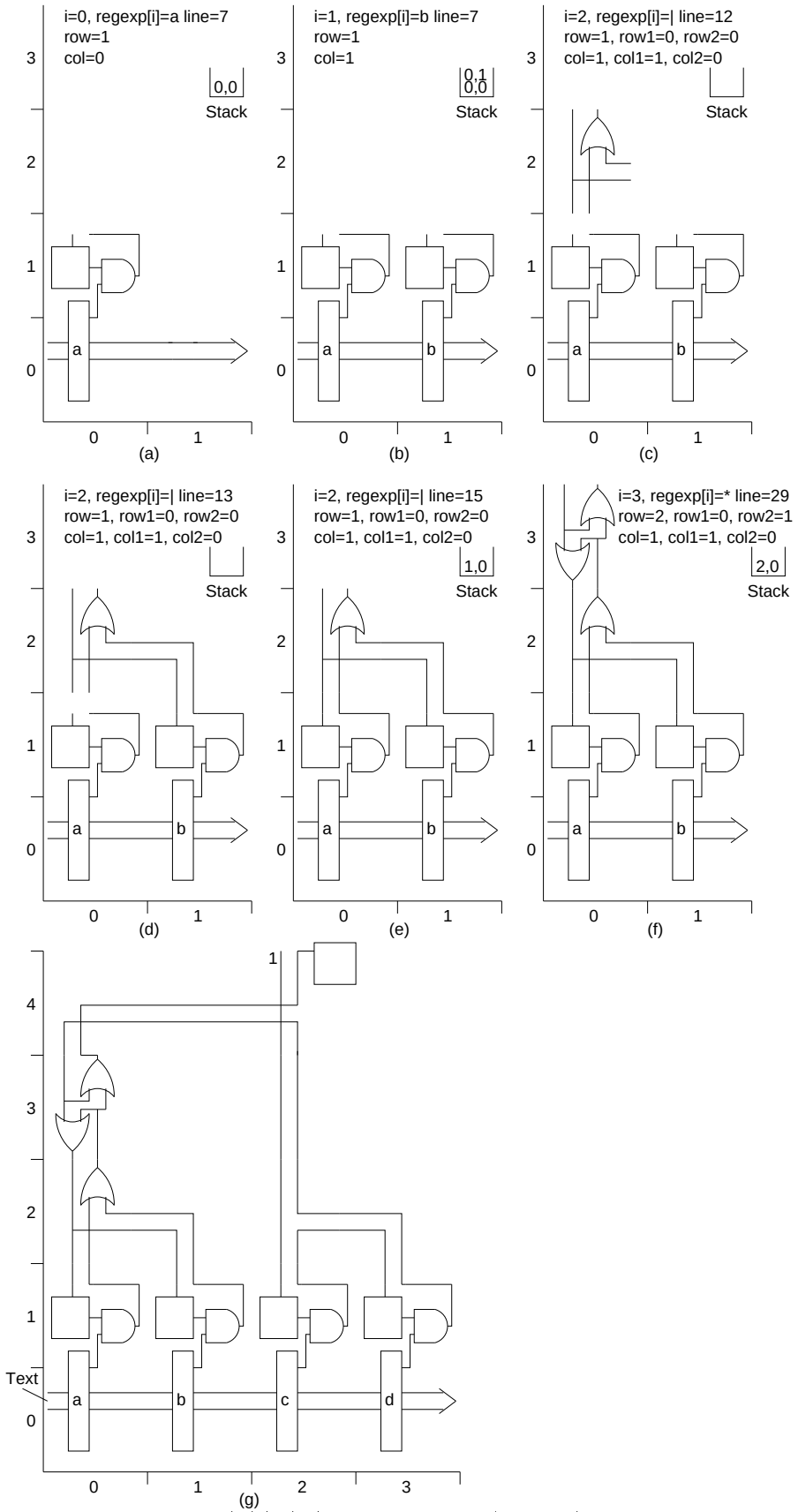


Figure 12: NFA construction for $((a|b)*)(cd)$ (postfix form: $ab|*cd\cdot\cdot$). Each figure shows NFA construction immediately after the algorithm statement indicated by line has been executed.

```
 1 row=1; col=0; i=0;                                [1]
 2 while(i<regexp_len)
 3 {
 4   switch(regexp[i])
 5   {
 6     case char: place_char(regexp[i], col); [46]
 7                push(0, col);               [1]
 8                ++col;                      [0]
 9
10     case    |: pop(&row1, &col1);          [1]
11                pop(&row2, &col2);          [1]
12                place_|(row, col2);         [22]
13                route_row(col2, col1);      [10]
14                route_col(row, row2);       [10]
15                push(row, col2);            [1]
16                ++row;                      [0]
17
19     case    .: pop(&row1, &col1);          [1]
20                pop(&row2, &col2);          [1]
21                place_.(row, col2);         [14]
22                route_row(col2, col1);      [10]
23                route_col(row, row2);       [10]
24                push(row, col2);            [1]
25                ++row;                      [0]
26
27     case    *: pop(&row2, &col2);          [1]
28                place_*(row, col2);         [26]
29                push(row, col2);            [1]
30                ++row;                      [0]
31
32   }
33   ++i;
34 }
35 pop(&row, &col);                            [1]
36 route_input_high(row, col);                 [3]
37 route_output_ff(row, col);                  [3]
```

Figure 10: NFA construction algorithm.

| $k$ | Text file size (bytes) | CPU time | Maximum memory |
|---|---|---|---|
| 8 | 2560 | 0.01 s | 1 MB |
| 9 | 5632 | 0.05 s | 1 MB |
| 10 | 12288 | 0.15 s | 1.9 MB |
| 11 | 26624 | 0.50 s | 2.2 MB |
| 12 | 57344 | 2.22 s | 3.0 MB |
| 13 | 122880 | 16.11 s | 4.4 MB |
| 14 | 262144 | 82.88 s | 7.5 MB |
| 15 | 557056 | 345.33 s | 13 MB |
| 16 | 1179648 | 1383.55 s | 26 MB |
| 17 | 2490368 | 5499.60 s | 54 MB |
| 18 | 5242880 | 21900.36 s | 111 MB |
| 19 | 11010048 | 87309.38 s | 229 MB |

Table 1: Results for text search and complete DFA construction (worst case).

space CPU times and the memory reported is the maximum memory used by that invocation of grep.

GNU grep uses clever techniques to reduce time and space requirements. Essentially, it constructs the DFA for the given regular expression in a "lazy" manner—only those parts of the DFA are constructed which are required to process the input text characters. For instance, if none of the characters in the regular expression occurs in the text, the DFA is not constructed at all. Therefore, to force grep to construct the entire DFA, the input text must contain patterns that exercise all possible DFA transitions. For example, for the regular expression $(a|b) * a(a|b)^2$, we create a file containing the four lines "aaa", "aab", "aba" and "abb"[1].

| $k$ | Text file size (bytes) | CPU time | Maximum memory | Time per character |
|---|---|---|---|---|
| 8 | 2560 | 0.00 s | 580 KB | – |
| 9 | 5632 | 0.00 s | 580 KB | – |
| 10 | 12288 | 0.00 s | 580 KB | – |
| 11 | 26624 | 0.00 s | 580 KB | – |
| 12 | 57344 | 0.00 s | 580 KB | – |
| 13 | 122880 | 0.005 s | 580 KB | – |
| 14 | 262144 | 0.01 s | 580 KB | – |
| 15 | 557056 | 0.03 s | 580 KB | 53.86 ns |
| 16 | 1179648 | 0.04 s | 580 KB | 33.91 ns |
| 17 | 2490368 | 0.08 s | 580 KB | 32.12 ns |
| 18 | 5242880 | 0.17 s | 580 KB | 32.42 ns |
| 19 | 11010048 | 0.34 s | 580 KB | 30.88 |

Table 2: Results for text search and minimal DFA construction (best case).

Such files were created for regular expressions ranging from $k = 8$ to $k = 19$. The program grep was then invoked once for each regular expression with the corresponding file supplied as input text. The results, summarized in Table 1, demonstrate the exponential blowup in both time and memory that can occur in DFA based regular expression matchers. As $k$ is increased, time required and memory consumed quickly reach unacceptable levels. It should be clear from the above discussion that times obtained are composed of the worst case DFA construction time (since the complete DFA is constructed) plus the time required to search $(k + 2)2^k$ text characters.

We now determine time and memory requirements for best case DFA construction (only one transition computed). We do this by repeating the above experiment with text files of the same sizes as before, but with each file containing $2^k$ repetitions of a single pattern instead of $2^k$ patterns. Table 2 shows the results obtained. The assumption that the DFA construction takes negligible time is justified by the almost constant time per text character in all cases. Also, all the grep invocations use the same amount of memory. Therefore, we

[1] In general, for $(a|b) * a(a|b)^k$, the number of lines in the text file would be $2^k$. Each line would have $k+2$ characters (the $(k+2)$th character being a newline) resulting in a file size of $(k+2)2^k$ bytes.

| $k$ | DFA size | Construction time |
|---|---|---|
| 8 | 420 KB | 0.01 s |
| 9 | 420 KB | 0.05 s |
| 10 | 1.32 MB | 0.15 s |
| 11 | 1.62 MB | 0.50 s |
| 12 | 2.42 MB | 2.22 s |
| 13 | 3.82 MB | 16.11 s |
| 14 | 6.42 MB | 82.87 s |
| 15 | 12.42 MB | 345.30 s |
| 16 | 25.42 MB | 1383.51 s |
| 17 | 53.42 MB | 5499.52 s |
| 18 | 110.42 MB | 21900.19 s |
| 19 | 228.42 MB | 87309.04 s |

Table 3: Software performance results for DFA construction (worst case). Best case requires negligible time and memory. Time per text character is 30.88 ns.

| $k$ | Configuration bit generation | FPGA configuration | NFA construction |
|---|---|---|---|
| 8 | 20 ms | 1 ms | 21 ms |
| 9 | 38 ms | 1 ms | 39 ms |
| 10 | 31 ms | 1 ms | 32 ms |
| 11 | 33 ms | 1 ms | 34 ms |
| 12 | 30 ms | 1 ms | 31 ms |
| 13 | 28 ms | 1 ms | 29 ms |
| 14 | 32 ms | 1 ms | 33 ms |
| 15 | 33 ms | 1 ms | 34 ms |
| 16 | 33 ms | 1 ms | 34 ms |
| 17 | 36 ms | 1 ms | 37 ms |
| 18 | 36 ms | 1 ms | 37 ms |
| 19 | 30 ms | 1 ms | 31 ms |
| 28 | 38 ms | 1 ms | 39 ms |

Table 4: NFA construction times.

obtain the DFA construction time and memory requirements by subtracting values in Table 2 from corresponding values in Table 1. Also, the time required to process a text character is estimated to be equal to the time per character for $k = 19$. Table 3 summarizes the software performance results.

## 5.2 FPGA Performance

The FPGA performance results were obtained using Xilinx Foundation tools running on a 450 MHz Pentium III and the target device was the Virtex XCV100 FPGA which is one of the smallest Virtex FPGAs. It has a $20 \times 30$ CLB array . The NFA construction for the Virtex architecture is based on the algorithm shown in Figure 10. The basic difference is that by utilizing the richer interconnect of Virtex, logic structures for two metacharacters (instead of one) can be placed in a single row of logic cells[2]. Also, the required routing along rows and columns can be performed using the longlines, 12 of which are present along each row and column. Doing so simplifies and speeds up the row_route and col_route operations.

We obtain the NFA size, NFA construction time, and time per text character for $(a|b) * a(a|b)^k$, $k$ ranging from 8 to 19, as follows. The logic structures shown in Figure 5 were described in VHDL. Using these, VHDL descriptions of the above regular expression were written and synthesized. NFA simulation was performed to verify proper operation. Next, the floor plan editor was used to manually place each NFA as it would have been by the algorithm described above. These were then routed and used to determine the NFA size and the time required to process a text character. The NFA construction time, as described in Section 3.4, consists of the times required for the NFA construction algorithm, configuration bit generation, and FPGA configuration. Since a very small amount of computation is required by the NFA construction algorithm, we assume its execution time to be quite small compared to the time required for the latter two tasks. Thus the NFA construction time is estimated to be the sum of the configuration bit generation and FPGA configuration times as shown if Table 4. Table 5 summarizes all the results.

| $k$ | NFA area | Construction time | Time per text character |
|---|---|---|---|
| 8 | $10 \times 7$ CLBs | 21 ms | 10.70 ns |
| 9 | $11 \times 8$ CLBs | 39 ms | 11.68 ns |
| 10 | $12 \times 8$ CLBs | 32 ms | 11.99 ns |
| 11 | $13 \times 9$ CLBs | 34 ms | 12.17 ns |
| 12 | $14 \times 9$ CLBs | 31 ms | 12.69 ns |
| 13 | $15 \times 10$ CLBs | 29 ms | 12.32 ns |
| 14 | $16 \times 10$ CLBs | 33 ms | 12.70 ns |
| 15 | $17 \times 11$ CLBs | 34 ms | 11.89 ns |
| 16 | $18 \times 11$ CLBs | 34 ms | 12.55 ns |
| 17 | $19 \times 12$ CLBs | 37 ms | 13.06 ns |
| 18 | $20 \times 12$ CLBs | 37 ms | 13.24 ns |
| 19 | $21 \times 13$ CLBs | 31 ms | 14.98 ns |
| 28 | $30 \times 16$ CLBs | 39 ms | 17.42 ns |

Table 5: NFA area, NFA construction time and time per text character for the FPGA implementation.

## 5.3 Performance Comparison

**FA size** The space requirement (memory for software and area for FPGA) is determined by the FA size. For software, in the best case scenario, the DFA requires a small, constant amount of memory. However, as shown in Table 3, the exponential blowup of the DFA size can con-

[2]A logic cell consists of an LUT and a flip-flop and thus corresponds to one-fourth of a Virtex CLB.

sume hundreds of megabytes in the worst case.

In contrast, using the proposed approach the area of the NFA constructed on an FPGA grows only quadratically with regular expression length *in all cases*. Thus space requirements are dramatically reduced compared to the worst case software requirements but are somewhat larger than the best case requirements. However, in practice, NFAs for long regular expressions can fit into a small amount of logic. For example the expression $(a|b) * a(a|b)^{28}$ which has a length of 118 can fit into the XCV100 (see the last row of Table 5). NFAs for much longer regular expressions can be constructed on bigger devices.

**FA construction time** In case of software, the DFA construction time in the best case scenario is negligible. However, it increases exponentially with regular expression length in the worst case. As shown in Table 3, DFA construction can take a few hours even on an 800 MHz microprocessor.

In contrast, using the proposed approach, an NFA can be constructed on an FPGA in less than a tenth of a second *in all cases*. Thus construction time is several orders of magnitude lower than the worst case software time, but slightly higher than the best case software time.

**Time per text character** As explained in Section 2.3, once constructed, DFAs as well as NFAs can process each text character in a constant amount of time. As described in Section 5.1 above, this time in case of software is 30.88 ns.

The times per text character using the proposed approach vary with the length of the regular expression and are shown in Table 5. All these times are significantly lower than the time required by software running on an 800 MHz microprocessor. However, the time required increases with regular expression length (the increase is roughly linear). Assuming the linear increase holds, the time required by the FPGA to process a text character will be lower than the time required by software even for regular expressions hundreds of characters in length.

As an example, finding matches to $(a|b) * a(a|b)^8$ in a 2MB file using grep will take 64.76 ms to 74.76 ms while consuming 580 KB to 1 MB memory, depending on the contents of the text file. In comparison, the FPGA implementation will always take 43.44 ms and occupy 70 CLBs. For $k = 19$ in the above example, grep would require 64.76 ms to 87309.1 s time and 580 KB to 229 MB of memory. The FPGA implementation would need 62.42 ms time and 273 CLBs.

To summarize, for text files above a certain size, the FPGA implementation always performs better than a software regular expression matcher. The improvement can be several orders of magnitude in both time and space requirements. For smaller files, in some cases, the FPGA performance may be slightly slower than software. This happens for cases where DFA construction time is smaller than the NFA construction time and the text file is too short for the faster FPGA processing to make a difference. Speedups can be obtained even for these cases by using the SRGA as described below to reduce the NFA construction time to less than 1 ms.

## 5.4 SRGA Performance

The SRGA architecture discussed in Section 4 has been described in Verilog and synthesized using a standard cell library for a 0.18 $\mu$ process. Placement and routing of the design are in progress. Below, we determine NFA size and estimate the NFA construction time. The time required to process a text character (and a more precise NFA construction time) will be reported after the SRGA design is complete.

| $k$ | NFA size | NFA construction time |
|---|---|---|
| 8 | 19 × 22 | 166.7 $\mu$s |
| 9 | 21 × 24 | 184.3 $\mu$s |
| 10 | 23 × 26 | 201.9 $\mu$s |
| 11 | 25 × 28 | 219.5 $\mu$s |
| 12 | 27 × 30 | 237.1 $\mu$s |
| 13 | 29 × 32 | 254.7 $\mu$s |
| 14 | 31 × 34 | 272.3 $\mu$s |
| 15 | 33 × 36 | 289.9 $\mu$s |
| 16 | 35 × 38 | 307.5 $\mu$s |
| 17 | 37 × 40 | 325.1 $\mu$s |
| 18 | 39 × 42 | 342.7 $\mu$s |
| 19 | 41 × 44 | 360.3 $\mu$s |

Table 6: NFA size and NFA construction times for the SRGA implementation.

The NFA construction algorithm, along with the number of clock cycles required to process each character of the regular expression are described in Section 4.2. Using these, we determine the number of clock cycles required to construct the NFA for $(a|b) * a(a|b)^k$ to be $259 + 176k$. Also, based on available SRGA timing information, we conservatively estimate the clock period of the NFA construction logic (shown in Figure fig:algolog) to be 100 ns. Product of clock period and number of clock cycles provides the NFA construction times which are summarized in Table 6. The table also shows NFA logic size.

Compared with NFA construction times for the FPGA implementation, the times for the SRGA are about two orders of magnitude smaller. This is due to self-reconfiguration which enables the device to generate configuration bits and configure itself with them. The faster NFA construction should speed up the proposed approach compared to a software regular expression matcher even for small text files. We expect to demonstrate this once the SRGA design is complete.

# 6 Conclusion

In this paper we have shown how to efficiently perform regular expression matching using FPGAs. The proposed approach takes $O(n+m)$ time and $O(n^2)$ space to find matches to a regular expression of length $n$ in text of length $m$. In contrast, the fastest serial machine algorithm requires $O(2^n+m)$ time and $O(2^n)$ space.

The key techniques that make the proposed approach efficient are the use of an NFA (instead of the DFA used on serial machines) for regular expression matching, and the fast construction of such an NFA. Based on theoretical definitions of DFA and NFA, we have described how an NFA can be directly implemented in logic by extending the OHE technique of DFA implementation. Also, based on theoretical techniques for NFA composition, we have developed a simple, fast algorithm that constructs an efficient NFA implementation for the given regular expression. We believe that the proposed approach has not been explored before on other parallel machines because the reconfigurability and fine-grained parallelism of FPGA devices is essential for the efficiency of the NFA implementation and construction techniques.

The other important contribution of the paper is to show how NFA construction can be performed very quickly using self-reconfiguration. We have developed an implementation of the NFA construction algorithm as configured logic on the SRGA which can construct an NFA in a few hundred $\mu$s.

Finally, we have shown in this paper detailed performance comparisons between software and FPGA implementations that justify the space and time performance claims made. The implementation results have shown that for most cases, the FPGA implementation is faster than grep for regular expression matching. The improvement can be several orders of magnitude in both time and space requirements. The FPGA implementation can be slightly slower for small files due to the time required for NFA construction. We have shown how this time can be reduced by about two orders of magnitude by constructing the NFA on the SRGA using self-reconfiguration.

It was observed that software regular expression matchers use techniques that help avoid the worst case memory and time requirements in many cases. However, no matter what techniques are used, for any DFA based matcher there will always be cases that cause exponential blowup in the time and memory required. We have demonstrated that the proposed approach overcomes this problem.

# 7 Acknowledgement

The authors thank Bharani Thiruvengadam for his work on FPGA performance evaluation.

# References

[1] A. V. Aho. *Handbook of Theoretical Computer Science, Volume A Algorithms and Complexity*, chapter 5. MIT Press/Elsevier, 1990.

[2] S. Golson. State machine design techniques for verilog and vhdl. *Synopsys Journal of High-Level Design*, pages 1–48, September 1994. www.synopsys.com/news/pubs/JHLD/JHLD-099401.

[3] B. Gunther, G. Milne, and L. Narasimhan. Assessing document relevance with run-time reconfigurable machines. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 10–17, Napa, CA, April 1996.

[4] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[5] R. P. S. Sidhu, A. Mei, and V. K. Prasanna. String matching on multicontext FPGAs using self-reconfiguration. In *FPGA '99. Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 217–226, Feb. 1999.

[6] R. P. S. Sidhu, A. Mei, S. Wadhwa, and V. K. Prasanna. A self-reconfigurable gate array architecture. In *FPGA '99. Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, Aug. 2000.

[7] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon. A first generation DPGA implementation. In *FPD'94 - Third Canadian Workshop of Field-Programmable Devices*, pages 138–143, May 1995.

[8] Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A time-multiplexed FPGA. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 22–28, Napa, CA, April 1997.

[9] Xilinx Inc. JBits. www.xilinx.com/products/jbits/index.htm.