# Partial Character Decoding for Improved Regular Expression Matching in FPGAs

Peter Sutton
*School of Information Technology and Electrical Engineering*
*The University of Queensland*
*Brisbane, Queensland, 4072, Australia*
*p.sutton@itee.uq.edu.au*

## Abstract

*High-speed string pattern matching in hardware is required in many applications including Network Intrusion Detection applications. Regular expressions are one method to implement such matching and are often built in FPGAs using non-deterministic finite automata (NFAs). To obtain high throughputs it is necessary to process many bytes in parallel. This paper extends the modular NFA construction method of Sidhu and Prasanna to handle the processing of many bytes in parallel. The paper also introduces the concept of partial character decoding in which character match units are shared but the number of signals needed to be routed around the FPGA is reduced over previous shared-decoder approaches. With these approaches, throughput over 5Gbps is achieved for the full default Snort rule-set (23401 literals) in a Xilinx Virtex-2 6000 FPGA. Throughputs over 40Gbps are achieved on smaller rule-sets. Suggestions to improve performance are also given.*

## 1. Introduction

There has been much recent work on the topic of accelerated textual pattern matching, and in particular, regular expression matching in FPGAs. Much of this has been in the context of Network Intrusion Detection System (IDS) applications - often using the rules from the Snort IDS [1] as a basis for testing.

A common and efficient method for implementing regular expression matching in hardware is the non-deterministic finite automaton (NFA). Since Sidhu and Prasanna [2] demonstrated how FPGAs can efficiently implement NFAs, a number of authors have presented variations on their approach which give better performance.

The work presented here extends the original modular building-block approach of Sidhu and Prasanna [2] by presenting regular expression building blocks suitable for processing an arbitrary number of bytes in parallel.

The paper also extends the work of Clark and Schimmel [3] who used a shared character decoder to more efficiently implement NFA regular expression circuits. Our results show that in most circumstances a partial decoding approach will give better results than a full 8-to-256 decoding of each character.

A program has been implemented which converts Snort rules into regular expressions and, from these, constructs a structural VHDL description of an NFA circuit which matches the given patterns whilst processing multiple bytes in parallel. The construction method uses the extended NFA building blocks and also uses the partial decoders proposed in this paper.

The remainder of this paper is organised as follows. Section 2 presents some background information. Section 3 describes the modular multi-byte NFA building blocks and Section 4 describes the partial character decoding approach. Section 5 describes the results from the synthesis of a number of example regular expression engines. Some discussion of the results and a description of future enhancements is presented in Section 6 followed by some conclusions in Section 7.

## 2. Background Information

This section presents background information on regular expressions, NFAs, NFA implementation in hardware, the Snort Network IDS tool, and other hardware approaches to Network IDS.

### 2.1. Regular Expressions

A regular expression is a pattern which describes a string (or strings) of characters. A regular expression consists of both characters (from some alphabet) and meta-characters which have special meaning. A regular expression *r* can be one of the following:

- a single character, or *literal* from the alphabet of interest - this matches exactly that character;

- a *concatenation* of two regular expressions, $r_1r_2$ - this matches regular expression $r_1$ immediately followed by $r_2$;
- an *alternation* of two regular expressions, $r_1|r_2$ - this matches either $r_1$ or $r_2$;
- the *closure* (or *star*) of a regular expression, $r_1$* - this matches 0 or more occurrences of $r_1$ ($r_1$* is the same as $\varepsilon|r_1|r_1r_1|r_1r_1r_1|...$ where ε is the empty (zero-length) string); or
- a parenthesized regular expression, ($r_1$) - this matches $r_1$ (parentheses are used for overriding precedence).

More complex regular expression syntaxes with many more meta-characters (e.g. . + [ ] ? etc.) are available. In all cases, such an expression can be converted to one using only the meta-characters above (though often at the cost of greatly increased pattern length)[1].

## 2.2. Non-Deterministic Finite Automata (NFAs)

A non-deterministic finite automaton (NFA) can be thought of as a directed graph whose nodes are states [5]. There is a *start* state and one or more *accepting* or *final* states. Transitions between states are labelled by any symbol from the alphabet or by ε, the empty string. The NFA *accepts* a string $a_1a_2...a_n$ if there is a path from the start state to an accepting state with labels $a_1$, $a_2$, … $a_n$. (The label ε may also appear any number of times.)

A deterministic finite automation (DFA) is an NFA in which there are no paths labelled with ε and no two outgoing paths from any state are labelled with the same symbol. At most one state of a DFA is active, whereas an NFA may have any number of active states. Any NFA can be converted to an equivalent DFA, though in the worst case this will take $O(n^2)$ time and $O(n^2)$ space where $n$ is the number of states in the NFA.

Further details on NFAs can be found in [4].

## 2.3. NFAs and Regular Expressions

NFAs are suited to regular expression matching because there is a direct mapping from any regular expression to an NFA [5] in $O(n)$ time where $n$ is the length of the regular expression. A software implementation of an NFA can match a character in $O(n)$ time[2]. In hardware, however, multiple state transitions can be considered in parallel so a hardware NFA can match a character in $O(1)$ time.

The implementation of regular expressions in hardware was first examined over 20 years ago when Floyd and Ullman [5] showed that an NFA regular expression circuit can be implemented efficiently using a programmable-logic array (PLA) architecture. More recently, Sidhu and Prasanna [2] showed that NFAs were an efficient method for implementing regular expressions in FPGAs. Both approaches involve a simple conversion process using one-hot encoding of the states.

As described in the following section, a number of researchers have examined regular expression matching in FPGAs.

## 2.4. Regular Expression Matching in FPGAs

Since the work of Sidhu and Prasanna [2] a number of authors have presented enhancements that improve pattern matching performance. Some of these are discussed below.

Hutchings et al. [6] report on a JHDL-based module generator. This uses the modular approach of Sidhu and Prasanna, but also supports additional regular expression meta-characters "?", "." and "[]". The module-generator also recognises and shares hardware between patterns with common prefixes. Characters to be matched are broadcast to all character match units, but a pipelined broadcast tree is implemented to improve performance.

Moscola et al. [8] take the approach that in many cases a DFA equivalent to an NFA would actually use fewer states. Because a DFA is always in a single state, such an approach enables the state to be captured in a small number of bits, which is ideal for swapping pattern matching contexts in and out of hardware. Their work processes a single byte at a time but uses multiple engines in parallel to accelerate matching.

## 2.5. Snort

Snort [1] is an open-source Network Intrusion Detection System (IDS). Snort is configured with a set of rules which describe packets of interest and the action which should be taken upon receipt of such packets. The rules can specify both packet header information (e.g. source and/or destination IP addresses, port numbers etc.) as well as packet content information to be matched against.

An example Snort rule (with some detail omitted) is:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 515
(msg:"EXPLOIT LPRng overflow"; flow:
to_server,established; content: "|43 07 89
5B 08 8D 4B 08 89 43 0C B0 0B CD 80 31 C0 FE
C0 CD 80 E8 94 FF FF FF 2F 62 69 6E 2F 73 68
0A|"; …)
```

1. Such a conversion may not lead to an efficient implementation but it will lead to correct one.
2. Software implementations of regular expression matching will typically use an equivalent DFA which can match any character in $O(1)$ time. Note however that the conversion process from an NFA to a DFA may be $O(n^2)$ in both time and space.

The rule specifies the action to take (`alert`), the protocol (`tcp`), the source and destination IP addresses and ports, some characteristics of the connection (`flow:` option) and the content to be matched (`content:` option), in this case expressed as a sequence of hexadecimal character values.

Snort rules are often used as example patterns for FPGA-based pattern matching implementations because they represent a large real-world set of patterns.

### 2.6. Other Approaches for Network IDS on FPGAs

Network IDS implementations on FPGAs need not be based on NFAs or even regular expressions. Other approaches have included (but are not limited to) DFAs [8], variations on the Knuth-Morris-Pratt (KMP) algorithm [9], parallel Bloom filters [10], and content-addressable memory (CAM) based approaches [11],[12].

This paper concentrates on the issue of efficiently implementing regular expression matching in FPGAs rather than Network IDS per se. Network IDS, in the form of Snort rules, is used as the example domain.

## 3. Regular Expression NFA Building Blocks

Building on the work of Sidhu and Prasanna [2], a building-block approach to the construction of NFA circuits which match regular expressions against multi-byte inputs is presented.

Regular expressions describe patterns in a given alphabet of characters. In Network Intrusion Detection applications, the alphabet is the set of all possible byte values (ranging numerically from 0 to 255 if considered as an unsigned value). Implementations of regular expression engines (in both software and hardware) usually consider one incoming symbol at a time.

As network speeds accelerate, Network Intrusion Detection applications have increasing performance requirements, e.g., needing to scan network traffic at throughputs of many giga-bits per second (Gbps). Such throughputs are relatively easy to obtain in hardware, *provided* that more than one byte of the input stream can be scanned simultaneously in parallel.

The easiest way to get greater performance (throughput) from a pattern matching circuit is to extend it to handle multiple-bytes in parallel. Sidhu and Prasanna's single byte matching unit (Fig. 1 (a)) can be extended to handle multiple bytes in parallel as shown in Fig. 1 (b) and (c). In each case, the flip-flop holds the NFA state bit.

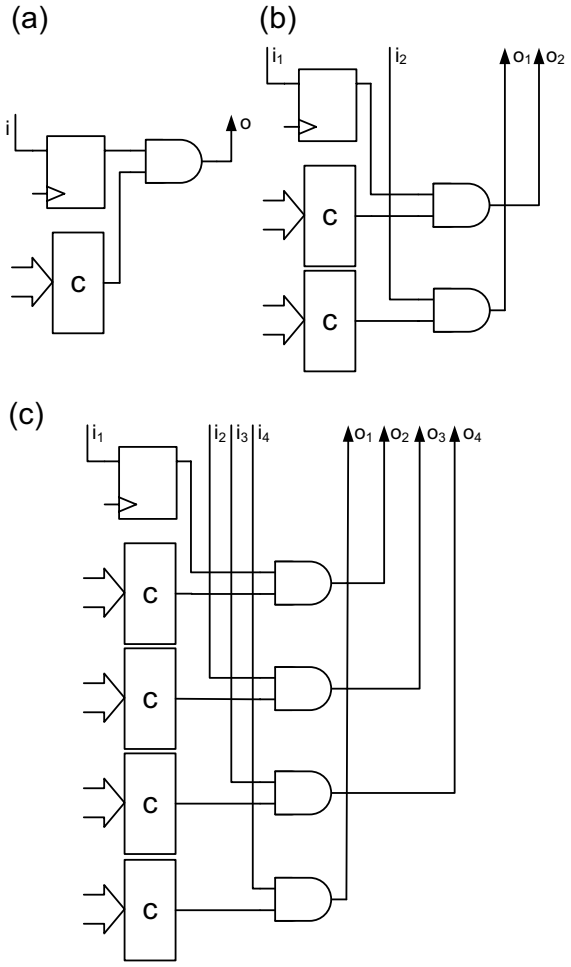


**Figure 1. Single character NFA building blocks for (a) single byte matching, (b) two bytes at a time, and (c) four bytes at a time. Each "C" block is a character match unit.**

For the single-byte case, a '1' stored in the flip-flop means that if the current character passes the character match unit "C", then we enable the subsequent state (i.e. output "o" will be high).

The multi- (n) byte case extends the byte matching unit to have n character match units - one examining each byte of the n-byte parallel input data. If the flip-flop holds a '1', output $o_2$ will be high if byte one matches the given character - i.e. the circuit is indicating to the next stage that it should look for match on byte two. If $i_2$ is high, $o_3$ will be high if byte two matches the required character. Similarly, if input $i_n$ is high, $o_1$ will be high if byte n matches the required character. The $o_1$ signal will be the $i_1$ input to a subsequent stage and will be latched by its flip-flop. Effectively this means that if the last byte of our group of bytes matched the required character, we'll be looking for the next match in the first byte of the *next* group of bytes.

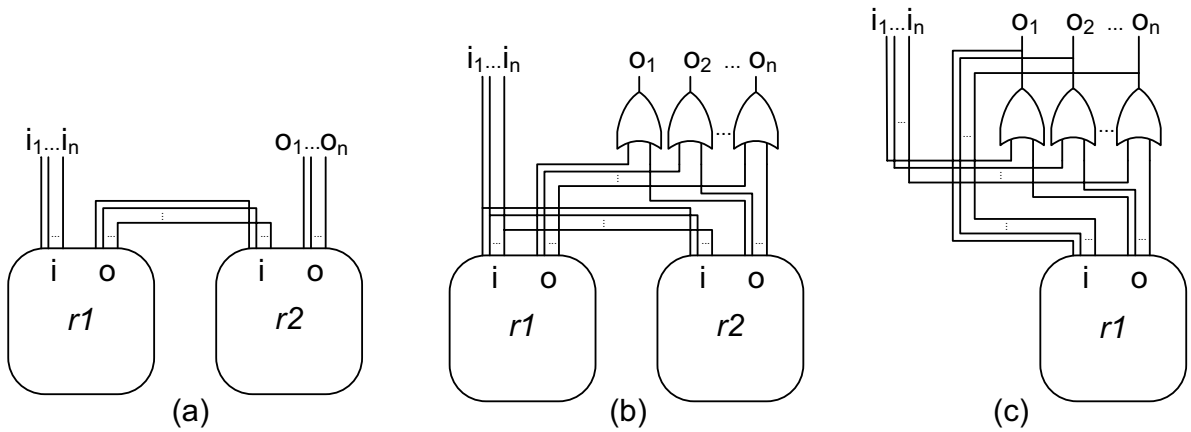


**Figure 2. Multi-byte regular-expression NFA building blocks for combination elements: (a) concatenation: *r1r2*, (b) alternation: *r1|r2* and (c) star: *r1**

The resulting circuit from grouping such blocks will be similar to the multi-byte approach presented by Cho et al. in [14] but does not break up the pattern into N-byte chunks (repeated at each of the N possible offsets). Sourdis and Pnevmatikatos [7] also present a multi-byte approach in which the pattern is repeated at different offsets. The approach presented here allows for a building-block approach to the construction of an NFA and is conceptually cleaner.

In addition to the multi-byte character matching block, the single byte concatenation, alternation and star blocks of Sidhu and Prasanna can be extended to multiple bytes as shown in Fig. 2. In the case where n=1, the circuits reduce to those shown in [2].

A tool has been implemented in Tcl [13] which reads patterns in the format of Snort rules, forms regular expression parse trees, and then generates structural VHDL describing an NFA circuit which matches the given patterns. The final alternation (i.e. or combination) of all the patterns in the rule-set uses a pipelined or-tree so as not to be a bottle-neck. The number of bytes to be considered in parallel is an input parameter to the tool.

Character match units generated by the tool take advantage of shared partial decoders as described in the following section.

## 4. Partial Character Decoding

Sidhu and Prasanna's approach (and that of many others) distributes each input character to each character match unit in the circuit and replicates comparators. Clark and Schimmel [3] describe a shared character-decoder approach. An 8-to-256 decoder is used to generate a single-bit match signal for each character. These single-bit signals are then distributed to the appropriate character match units, removing the overhead of distributing an 8-bit character to each character match unit and performing redundant character comparisons. This approach saves logic resources but does require a large number of global signals to be distributed around the FPGA.

Our work extends this approach by considering *partial shared character decoding*. Partial decoding means that instead of decoding all 8 bits of each character, we decode groups of bits within each character. In this work, we considered breaking the 8 bits of each character into 1,2,3,4 and 8 groups as shown in Fig. 3 (One group of 8 bits is not partial decoding - we consider this decoding option to enable comparison between this approach and partial decoding.)

Shared partial decoding allows trade-offs between the amount of logic shared amongst the whole circuit (i.e. the number and size of the character decoders) and the number of signals which have to be routed around the FPGA.

As shown in Fig. 3, the number of signals which need to be distributed around the FPGA reduces as the number of decoding groups increases. With full 8-to-256 decoding for each character, up to 256 signals need to be distributed around the FPGA[3] from our shared decoder block (though each match unit will only need one of these signals). With four 2-to-4 decoders, for example, only 16 signals need to be distributed around the FPGA (though on average, each signal will be needed in 16 times more places than with 8-to-256 decoding). If multiple (n) bytes are being processed in parallel, the number of signals to be distributed must be multiplied by n.

Partial decoding does increase the complexity of the character match units. With 8-to-256 decoding, the character match unit (shown as blocks labelled "C" in Fig. 1) becomes just a wire (as shown in Fig. 4(a)). With partial decoding, an AND gate is required in each character match unit. Fig. 4 shows the required character match units (i.e. "C" blocks) for each of the partial decodings considered. It is hypothesized that the cost of this is negligible given that the output

3. Fewer signals (and decoders) are needed if the pattern-space does not use all 256 possible characters.

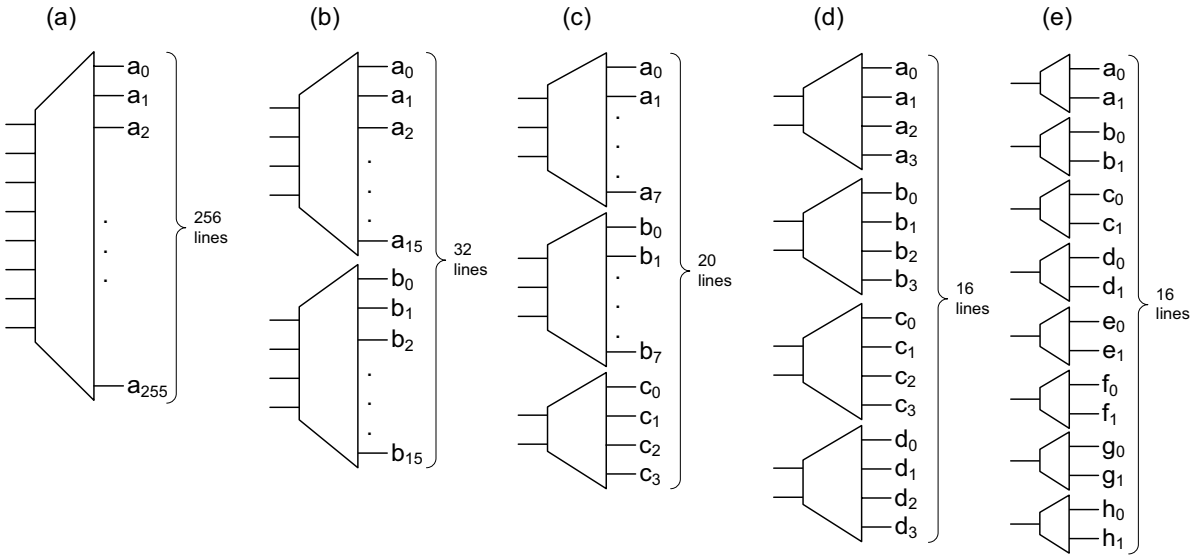


**Figure 3. Character decoding options considered in this paper: (a) Each character is fully decoded. (b) Each character is broken into two groups of four bits each. (c) Each character is broken into three groups of 3,3,2 bits. (d) Each character is broken into four groups of 2 bits each. (e) Each character is broken into individual bits.**

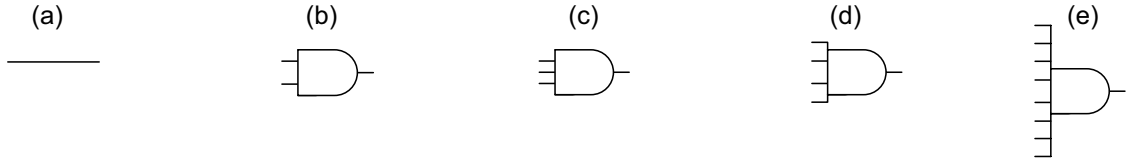


**Figure 4. Character match units required for each of the five decoding options shown in Fig. 3**

of the character match unit is ANDed with the current NFA state bit or other input (as shown in Fig. 1).

## 5. Experimental Results

In this section we present the results of combining partial-character decoding with the modular construction of multi-byte regular expression NFAs. A number of rule sets are considered. We chose to target the Xilinx Virtex-2 6000 (speed grade 5) and Xilinx Virtex-E 2000 (speed grade 7) since those devices are used as examples in a number of related studies.

### 5.1. Rule-sets

A number of rule-sets were considered in the experiments. These were subsets of the Snort 2.0.0 rule-set as follows:

- web-attacks - the set of rules in the file web-attacks.rules
- sql - the set of rules in the file sql.rules
- web-iis - the set of rules in the file web-iis.rules
- default - the set of rules enabled by default in the 2.0.0 rule-set (i.e. those rules included in the snort.conf file)

Various characteristics of these rule-sets are shown in Table 1. The "number of content patterns" refers to the total number of `content:` or `uri-content:` strings in the rule-set. The "number of literals" is the total number of characters in each of the content patterns for that rule-set. The smaller rule-sets were chosen based on their relative size (an approximate doubling of the number of literals from set to set).

**Table 1: Experimental rule-set characteristics**

| Name | Number of rules | Number of content patterns | Number of literals |
|---|---|---|---|
| web-attacks | 47 | 47 | 450 |
| sql | 43 | 49 | 1034 |
| web-iis | 116 | 130 | 2016 |
| default | 1554 | 2260 | 23401 |

A number of artificial rule sets of various sizes and lengths (with low inter-pattern commonality) were also tested but results for these are not presented as little is added to the data presented below.

### 5.2. Methodology

Structural VHDL was generated (by the tool described earlier) for each of the circuits under test. The design was passed through a standard Xilinx ISE 6.1 tool flow (xst, ngdbuild, map, par) with default options. The only constraint specified was the target clock period, which was chosen to be 1.25 times the predicted clock period reported by the synthesis tool. This resulted in some designs being overconstrained (failing to meet the timing constraint) and some underconstrained (easily meeting the timing constraint). In all cases, the clock period (and hence maximum frequency) reported is the "Actual" value reported by the place-and-route tool. The throughput reported (in Gbps) is the maximum frequency obtained (in MHz) multiplied by 8 multiplied by the number of bytes being processed in parallel divided by 1024.

For all of the rule-sets except the largest ("default"), VHDL was generated, synthesized, placed and routed for each combination of:

- the two devices being considered;
- the five decoding options; and
- the processing of 1,2,3,4,5,6,7,8,10,12,14 and 16 bytes in parallel.

For the large "default" rule set, only the processing of 1,2 and 3 bytes in parallel was considered for some of the five decoding options and only on the larger Virtex 2 device. Some combinations were beyond the memory capacity of the computer concerned even though the device was not fully utilized.

### 5.3. Performance

Tables 2, 3, 4, and 5 show the results for the various rule-sets on the Virtex-2 6000 device. The tables list the throughput (in Gbps) for the given combination of number of bytes processed in parallel and the style of character decoding. The number in parentheses in each table entry is the proportion of 4-input lookup-tables (LUTs) utilized in the device. For each given number of bytes processed in parallel, the decoding option which produced the greatest throughput is shaded light gray. Table 6 shows the results for the web-attacks rule-set on the Virtex-E 2000 device. Other results for this device are omitted.

A discussion of the results is presented below.

## 6. Discussion

The most important conclusion to draw from the experimental results is that there is no obvious relationship between the style of character decoding and the best performance (throughput). For any given rule-set and number of bytes to be processed in parallel, any one of the character decoding styles could produce the best performance. The best choice also depends on the device chosen - for the web-attacks

**Table 2: Gbps throughput (LUT utilization%) for web-attacks rule-set on Virtex-2 6000**

| Bytes in parallel | (a) 8-to-256 | (b) 2 Decoders | (c) 3 Decoders | (d) 4 Decoders | (e) 8 Decoders |
|---|---|---|---|---|---|
| 1 | 2.69 (0.5%) | 2.68 (0.5%) | 3.06 (0.5%) | 2.68 (0.5%) | 3.08 (2%) |
| 2 | 5.35 (0.6%) | 5.36 (0.6%) | 5.43 (0.8%) | 4.61 (0.7%) | 5.21 (3%) |
| 3 | 7.51 (0.8%) | 8.63 (1%) | 7.88 (1.3%) | 7 (1.1%) | 8.6 (3.4%) |
| 4 | 10.5 (1.2%) | 10.6 (1.4%) | 8.81 (1.7%) | 8.43 (1.6%) | 11.3 (4%) |
| 5 | 13.7 (1.7%) | 13.7 (1.8%) | 11.0 (2.2%) | 13.5 (2.2%) | 13.0 (4.8%) |
| 6 | 14.4 (2.1%) | 13.3 (2.1%) | 14.0 (2.7%) | 15.3 (2.7%) | 16.0 (5.4%) |
| 7 | 16.8 (2.5%) | 16.8 (2.6%) | 16.5 (3.3%) | 16.7 (3.2%) | 18.2 (6.2%) |
| 8 | 20.6 (3%) | 19.3 (3%) | 19.5 (3.7%) | 19.4 (3.8%) | 20.8 (6.3%) |
| 10 | 22.2 (3.8%) | 23.9 (3.9%) | 25 (4.7%) | 23.5 (4.9%) | 25.1 (8%) |
| 12 | 27.1 (4.6%) | 27.2 (4.8%) | 30.0 (5.7%) | 28.0 (5.8%) | 30.5 (8.9%) |
| 14 | 27.7 (5.4%) | 35.1 (5.7%) | 34.7 (6.7%) | 36.5 (6.9%) | 35.4 (10.2%) |
| 16 | 38.6 (6.2%) | 39.9 (6.7%) | 40.1 (7.5%) | 40.1 (7.8%) | 38.9 (10.8%) |

**Table 3: Gbps throughput (LUT utilization%) for sql rule-set on Virtex-2 6000**

| Bytes in parallel | (a) 8-to-256 | (b) 2 Decoders | (c) 3 Decoders | (d) 4 Decoders | (e) 8 Decoders |
|---|---|---|---|---|---|
| 1 | 3.31 (0.7%) | 2.76 (0.7%) | 2.85 (0.7%) | 2.83 (0.7%) | 3.3 (3.2%) |
| 2 | 5.15 (0.8%) | 5.26 (0.8%) | 4.86 (1%) | 5.33 (1.1%) | 4.29 (4%) |
| 3 | 7.91 (1%) | 8.5 (1.1%) | 7.71 (1.4%) | 7.88 (2%) | 6.59 (5.6%) |
| 4 | 11.0 (1.4%) | 10.3 (1.5%) | 10.7 (1.8%) | 11.4 (2.5%) | 8.84 (6.1%) |
| 5 | 12.0 (2%) | 13.7 (2%) | 13.1 (2.4%) | 13.1 (2.8%) | 10.8 (7.7%) |
| 6 | 14.2 (2.6%) | 14.6 (2.5%) | 13.9 (2.9%) | 18.0 (3.5%) | 13 (8.8%) |
| 7 | 16.6 (3%) | 16.7 (2.9%) | 16.8 (3.4%) | 20.6 (3.8%) | 15.1 (10.1%) |
| 8 | 19.2 (3.7%) | 19.1 (3.4%) | 19.1 (3.9%) | 22.9 (4.4%) | 16.2 (11.1%) |
| 10 | 23.5 (4.7%) | 23.3 (4.3%) | 23.5 (5%) | 25.1 (5.4%) | 20.6 (12.7%) |
| 12 | 29.9 (5.6%) | 29.9 (5.4%) | 23.0 (5.9%) | 30.9 (6.4%) | 23.7 (15.6%) |
| 14 | 35.7 (6.6%) | 35.8 (6.3%) | 36.0 (7.2%) | 35.7 (7.6%) | 32.9 (15.5%) |
| 16 | 39.0 (7.6%) | 39.2 (7.4%) | 38.8 (8.2%) | 39.7 (8.5%) | 32.9 (19.4%) |

**Table 4: Gbps throughput (LUT utilization%) for web-iis rule-set on Virtex-2 6000**

| Bytes in parallel | (a) 8-to-256 | (b) 2 Decoders | (c) 3 Decoders | (d) 4 Decoders | (e) 8 Decoders |
|---|---|---|---|---|---|
| 1 | 2.76 (2.3%) | 3.19 (2.3%) | 3.31 (2.2%) | 2.78 (2.3%) | 3.12 (4.6%) |
| 2 | 5.25 (2.6%) | 5.11 (2.6%) | 4.43 (3.2%) | 4.45 (4.2%) | 4.68 (5.6%) |
| 3 | 7.04 (3%) | 8.13 (3.7%) | 6.7 (5%) | 6.31 (6.3%) | 8.99 (9.1%) |
| 4 | 9.68 (4.2%) | 8.93 (5.3%) | 9.03 (6.5%) | 8.79 (8.2%) | 10.3 (10.5%) |
| 5 | 12.7 (5.9%) | 11.1 (7.1%) | 11.3 (8.4%) | 11.2 (9.7%) | 13.4 (14.7%) |
| 6 | 15.1 (7.6%) | 12.8 (8.5%) | 13.3 (10.3%) | 13.0 (11.5%) | 15.4 (16.7%) |
| 7 | 17.5 (9.1%) | 15.1 (10.1%) | 14.3 (11.9%) | 14.7 (13.2%) | 18.4 (19.5%) |
| 8 | 17.9 (10.9%) | 21.8 (11.8%) | 19.5 (13.9%) | 17.0 (15%) | 19.9 (21.6%) |
| 10 | 24.6 (14%) | 21.2 (15.3%) | 22.3 (18.1%) | 21.0 (19.1%) | 23.4 (26.6%) |
| 12 | 23.2 (16.5%) | 25.5 (18.8%) | 24.7 (21.8%) | 24.9 (22.5%) | 26.0 (31.4%) |
| 14 | Failed | 30.1 (22.7%) | 35.1 (25.8%) | 30.3 (26.3%) | 35.4 (36.6%) |
| 16 | Failed | 39.2 (26.7%) | 38.2 (30.1%) | 34.7 (30.5%) | 39.2 (41.4%) |

**Table 5: Gbps throughput (LUT utilization%) for default rule-set on Virtex-2 6000**

| Bytes in parallel | (a) 8-to-256 | (b) 2 Decoders | (c) 3 Decoders | (d) 4 Decoders | (e) 8 Decoders |
|---|---|---|---|---|---|
| 1 | Failed | 2.36 (20.2%) | 2.58 (20.2%) | 2.66 (20.3%) | 2.52 (23.5%) |
| 2 | Failed | 3.92 (22.7%) | 3.73 (36.4%) | 3.56 (41.8%) | 2.77 (38.9%) |
| 3 | Failed | 5.12 (28.5%) | 5.13 (61.1%) | 2.96 (61.3%) | Did not fit |

**Table 6: Gbps throughput (LUT utilization%) for web-attacks rule-set on Virtex-E 2000**

| Bytes in para-llel | (a) 8-to-256 | (b) 2 Decoders | (c) 3 Decoders | (d) 4 Decoders | (e) 8 Decoders |
|---|---|---|---|---|---|
| 1 | 1.55 (0.9%) | 1.67 (0.9%) | 2.08 (0.9%) | 1.76 (0.9%) | 1.85 (3.5%) |
| 2 | 3.37 (1.1%) | 3.03 (1.1%) | 3.44 (1.4%) | 3.38 (1.2%) | 3.28 (5.3%) |
| 3 | 4.56 (1.5%) | 4.9 (1.8%) | 4.89 (2.3%) | 4.59 (2%) | 4.66 (6%) |
| 4 | 6.18 (2.1%) | 6.52 (2.4%) | 5.79 (3.1%) | 4.98 (2.9%) | 6.05 (7%) |
| 5 | 6.67 (2.9%) | 7.19 (3.1%) | 6.42 (3.9%) | 7.2 (3.9%) | 7.68 (8.4%) |
| 6 | 7.65 (3.7%) | 8 (3.8%) | 8.21 (4.9%) | 7.86 (4.7%) | 8.76 (9.5%) |
| 7 | 8.97 (4.4%) | 9.77 (4.6%) | 9.64 (5.8%) | 8.83 (5.6%) | 9.74 (10.8%) |
| 8 | 10.1 (5.2%) | 9.83 (5.3%) | 11.1 (6.6%) | 9.86 (6.7%) | 10.3 (11.1%) |
| 10 | 12.4 (6.6%) | 12.8 (6.9%) | 13.1 (8.3%) | 12.5 (8.4%) | 13.9 (14%) |
| 12 | 15.3 (8.1%) | 15.0 (8.5%) | 15.4 (10%) | 15.7 (10.2%) | 16.3 (15.7%) |
| 14 | 17.6 (9.5%) | 17.2 (10.1%) | 17.3 (11.7%) | 19.5 (12.1%) | 18.0 (18%) |
| 16 | 20.0 (11%) | 21.0 (11.8%) | 20.6 (13.3%) | 20.1 (13.8%) | 20.2 (18.9%) |

rule-set (and also the other rule-sets, although these results are not shown) the best decoding option differed for the Virtex-E and Virtex-2 devices considered. This is likely to be because a device's balance between logic and routing resources may determine the most appropriate decoding approach for any given rule-set.

It should be noted that in all but a few cases, the full character-decoding option (8-to-256 decoding) did not produce the best performance. In most cases, however, it did produce the best logic utilization.

As expected, throughput improves as the number of bytes processed in parallel increases. For the three smaller rule-sets, processing 16 bytes in parallel gave throughputs (for the best decoding option) at or near 40Gbps. The best performance achieved for the default rule-set was 5.13 Gbps when processing 3 bytes in parallel with three shared character decoders (3-to-8, 3-to-8 and 2-to-4). Higher performance is likely to be possible.

### 6.1. Further Advantages of Partial Decoding

Although not applied here, partial decoding of characters would also allow improved implementation of pattern ranges (including case-insensitivity). A reduced number of signals and associated AND logic will be needed to match a range of characters, e.g., in the 4 decoder case, only 3 of the 4 signals are needed to decode a pattern range in which one of the 4 groups is wild (i.e. two of the bits of the character are wild).

### 6.2. Further Optimizations

A number of further optimizations are possible, including many that others have reported on. Clark and Schimmel [3] implement *prefix tree optimization* in which circuitry is eliminated for patterns with common prefixes. This occurred in our work only to the extent that the synthesis tool recognised common logic. Adding the capability to the circuit generator might reduce synthesis time and give improved results.

Little attempt was made at pipelining in this work (other than the final or-combination). As shown in [6], extensive fine-grained pipelining can produce even better performance (at the cost of latency).

An extension to support further regular expression meta-characters may also reduce generated circuit size as also shown in [6].

## 7. Conclusions

This paper has presented an extension to the modular regular-expression to NFA construction method of Sidhu and Prasanna in order to handle multiple bytes in parallel in an equivalent modular fashion. The paper has also introduced the idea of shared partial character decoding and demonstrated that full shared character (8-to-256) decoding is usually not the best approach to achieve the best throughput. The ideal partial decoding option has been experimentally shown to be dependent on the FPGA device concerned, the number of bytes to be processed in parallel and the patterns to be implemented.

A program has been written which converts Snort rules to regular expressions and then applies the multi-byte modular NFA construction method to produce circuits which match given patterns using a particular decoding method and processing a given number of bytes in parallel. The circuits generated are able to match against the complete default Snort rule set at a throughput of over 5Gbps in a Xilinx Virtex-2 6000 device. Throughputs over 40Gbps are achieved on smaller rule-sets. Future work will involve incorporating further optimizations into the circuit generation program.

## References


[1] http://www.snort.org. (Accessed 1 June 2004)

[2] Sidhu, R. and Prasanna, V.K., "Fast Regular Expression Matching using FPGAs," in *Proceedings of 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'01), April 2001.

[3] Clark, C.R. and Schimmel, D.E., "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns," in *Proceedings of FPL 2003*, LNCS 2778, pp 956-959, Sep. 2003.

[4] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, MA, 1979.

[5] Floyd, R.W. and Ullman, J.D., "The Compilation of Regular Expressions into Integrated Circuits", *Journal of ACM*, vol. 29, no. 3, pp 603-622, July 1982.

[6] Hutchings, B.L., Franklin, R. and Carver, D., "Assisting Network Intrusion Detection with Reconfigurable

Hardware", in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'02), April 2002.

[7] Sourdis, I. and Pnevmatikatos, D., "Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System", in *Proceedings of FPL 2003*, LNCS 2778, pp 880-889, Sep. 2003.

[8] Moscola, J., Lockwood, J., Loui, R.P. and Pachos, M., "Implementation of a Content-Scanning Module for an Internet Firewall", in *Proceedings of 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'03), April 2003.

[9] Baker, Z.K. and Prasanna, V.K., "Time and Area Efficient Pattern Matching on FPGAs", in *Proceedings of FPGA'04*, February 2004.

[10] Dharmapurikar, S., Krishnamurthy, P., Sproull, T.S. and Lockwood, J.W., "Deep Packet Inspection Using Parallel Bloom Filters", *IEEE Micro*, pp 52-61, Jan/ Feb 2004.

[11] Gokhale, M., Dubois, D, Dubois, A., Boorman, M., Poole, S. and Hogsett, V., "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology", in *Proceedings of FPL 2002*, LNCS 2438, pp 404-413, Sep. 2002.

[12] Li, S., Torresen, J. and Soraasen, O., "Exploiting Reconfigurable Hardware for Network Security", in *Proceedings of 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'03), April 2003.

[13] http://www.tcl.tk. (Accessed 1 June 2004)

[14] Cho, Y.H., Navab, S., Mangione-Smith, W.H., "Specialized Hardware for Deep Network Packet Filtering", in *Proceedings of FPL 2002*, LNCS 2438, pp 404-413, Sep. 2002.