# Reconfigurable Context-Free Grammar Based Data Processing Hardware with Error Recovery *

James Moscola, Young H. Cho, and John W. Lockwood

Washington University in St. Louis
Department of Computer Science and Engineering
St. Louis, Missouri 63130
{jmm5, young, lockwood}@arl.wustl.edu

## Abstract

*This paper presents an architecture for context-free grammar (CFG) based data processing hardware for reconfigurable devices. Our system leverages on CFGs to tokenize and parse data streams into a sequence of words with corresponding semantics. Such a tokenizing and parsing engine is sufficient for processing grammatically correct input data. However, most pattern recognition applications must consider data sets that do not always conform to the predefined grammar. Therefore, we augment our system to detect and recover from grammatical errors while extracting useful information. Unlike the table look up method used in traditional CFG parsers, we map the structure of the grammar rules directly onto the Field Programmable Gate Array (FPGA). Since every part of the grammar is mapped onto independent logic, the resulting design is an efficient parallel data processing engine. To evaluate our design, we implement several XML parsers in an FPGA. Our XML parsers are able to process the full content of the packets up to 3.59 Gbps on Xilinx Virtex 4 devices.*

## 1 Introduction

Recent studies in network intrusion detection system (NIDS) show that a single field programmable gate array (FPGA) device can search for several thousand worm signatures over networks exceeding 1 Gbps in bandwidth [2, 3, 5, 7, 8]. Such real-time pattern detection can prevent network attacks. However, naive pattern searching methods are susceptible to false positive identification. On high speed networks, even a small percentage of incorrect identification can surmount to an unacceptable amount of data.

This paper presents a hardware architecture that uses context-free grammars (CFG) to increase the pattern recognition accuracy. CFGs provide a higher level of expressiveness than regular expressions by defining the semantics of the patterns within the structure of their language. The contextual information of patterns can then be used to reduce the number of false positive identifications. A method for error detection and recovery ensures that useful information can be extracted from a data stream that doesn't perfectly conform to the grammar.

A compiler automatically generates the CFG hardware using Lex and Yacc style grammars as input. The structure of the grammar is determined using algorithms for software predictive parser generation. The generated hardware is a highly pipelined and parallel engine that recognizes patterns and the semantics of streaming data.

## 2 Hardware Architecture

Using technologies developed for CFGs, we describe an effective way to augment fast pattern recognition hardware with the ability to parse data and recover from erroneous inputs while collecting useful data. Our design consists of a tokenizer that is generated based on a token list, a parsing structure that is generated from the production list of a grammar, and an error detection and recovery unit.

*This research was sponsored by the Air Force Research Laboratory, Air Force Materiel Command, USAF, under Contract number MDA972-03-9-0001. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL or the U.S. Government.

## 2.1 Tokenizer

The first task of language interpretation is tokenization. To minimize the amount of routing required to the tokenizers, each 8-bit character is decoded to a single bit line [6].

### 2.1.1 Regular Expression Chain

The first method we present for detecting regular expression tokens consists of chains of pipelined AND gates as shown in figure 1a. At each stage of the pipeline, the output is asserted when the decoded character bit and the output of the previous stage are both asserted. Such a chain is capable of detecting specified strings in data streams [5].

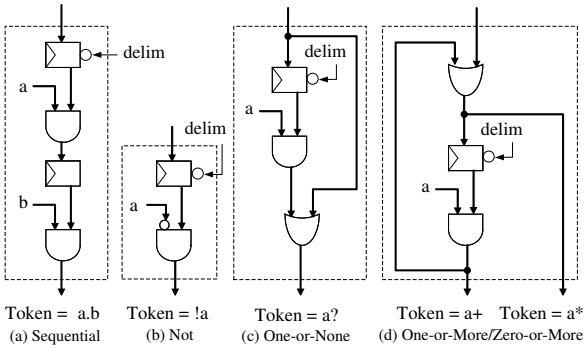


**Figure 1. Constructing regular expressions**

As a stream of data enters the hardware, token delimiters effectively hold the detection of next pattern. Therefore, the delimiter detection output is inverted and connected to the enable signals of the first registers in the token detection chains. It is necessary that only the first register of each token chain is stalled. Otherwise, two partial tokens separated by a delimiter could be recognized as a complete token.

Other regular expression functions such as *Not*, *One-or-None*, *One-or-More*, and *Zero-or-More* can also be represented in the hardware logic as shown in figure 1. Our hardware generator instantiates combinations these elementary logic templates to build regular expression detectors.

### 2.1.2 Pipelined Character Grid

Instead of forwarding the decoded characters to each stage of the pipelined chains, the decoded characters can be buffered into a pipelined grid structure as shown in figure 2. Given such a grid of decoded characters, one can construct string detectors that are less than or equal to the length of the pipeline [2]. Though the commong grid structure can save resources over the regular expression chain, it is not flexible enough to allow the detection of full regular expressions without incurring a lot of additional logic.

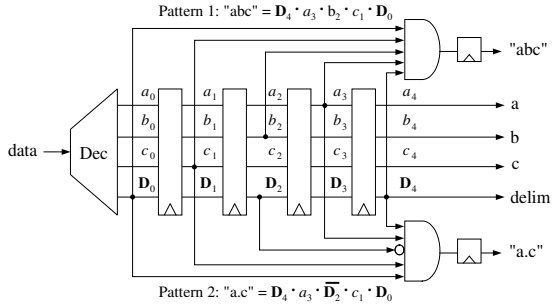


**Figure 2. A pipelined character grid**

## 2.2 Grammar Parser

Once tokenization has occurred, the next step in language interpretation is to parse the tokens according to the grammar. By modularizing the parser and the tokenizer, we eliminate the need to create multiple copies of a tokens logic for each of its contexts as in our previous work [4]. Instead, we create a single instance of each token in the tokenizer, and use a compact parsing structure to maintain the context of the grammar. Adding the parsing structure allows for an overall savings in the space required by a grammar. This is especially true for grammars that use the same token in many different contexts of the grammar.

The hardware logic for the parser is determined from the production list of the grammar. Tokens are represented in the parser using a primitive similar those shown in figure 1a where each token consists of a single register and a single AND gate. The inputs to each of the AND gates are the outputs of the tokenizer. The output of each AND gate represents a token transition and is routed to the input of other token registers. Token transitions are determined from the production list using the well known Follow set algorithm [1]. The Follow set algorithm traverses through the production list to find sets of tokens that can follow any given symbol in the grammar. The resulting sets can then be used to map the grammar onto the hardware. An example grammar is shown in figure 3. The corresponding hardware parser is shown in figure 4.

As the CFG hardware processes a data stream, the parser receives a stream of token signals from the tokenizer. These signals allow the parser to traverse the grammar and maintain the context of the data stream.

| No. | | Production |
|---|---|---|
| 1 | E → | if C then E else E \| go \| stop |
| 2 | C → | true \| false |

**Figure 3. Grammar for if-then-else statement**

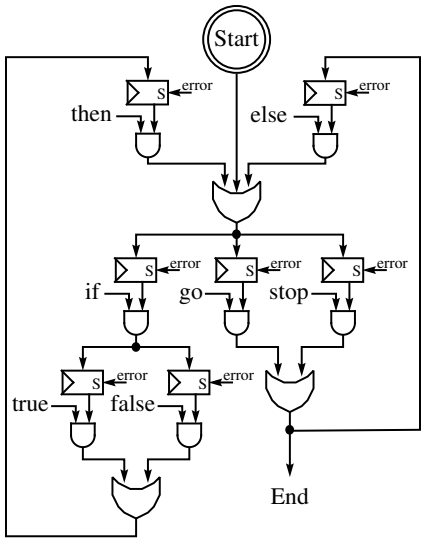


**Figure 4. Parser for if-then-else statement**

### 2.3 Error Detection and Recovery

Our initial design [4] requires the input data sets conform to the rules in the CFG. However, such a constraint can make the design ineffective in applications that deal with unreliable input data. Therefore, we must employ methods for error detection and recovery. Adding the ability to detect errors as they occur and gracefully recovery from them helps to ensure that the parser can correctly detect and tag tokens even after an error has occurred in the data stream.

To design an error detection module, we first have to define what an error is in terms of the parser architecture. The parser architecture consists of a group of registers, one for each context of each token. Once the grammar has started matching, there should always be at least one active register in the parser architecture until the grammar reaches some ending state. The absence of an active register in the parser architecture indicates that an error has occurred.

Errors occur in the grammar when the parser receives a token from the tokenizer that it wasn't expecting — that is, the token that the parser has received is not in the follow set of any of the currently active tokens. This can occur for several different reasons including: the expected token was malformed and therefore not detected by the tokenizer; the expected token was missing; or extraneous tokens were inserted into the data stream prior to the expected token.

As an example, when the grammar in figure 4 receives the *start* signal, the registers associated with "if", "go", and "stop" all become active. If the first token received from the tokenizer is the "if" token, the active signal is passed from the "if" register to both the "true" and "false" registers. The active signals from both the "go" and "stop" registers are not passed along and these registers become inactive. If the second token received from the tokenizer is the "then" token, both of the active registers, the "true" and the "false" registers, are prevented from forwarding the active signal and both become inactive. At this point, there are no active registers in the parser architecture because an "if" followed by a "then" does not conform to the grammar in figure 3, hence an error has occurred. To detect these errors in the grammar the output of each register is forwarded to a pipelined NOR tree.

The pipelined NOR tree is optimized for implementation on an FPGA by building each stage of the pipeline out of a series of 4-input OR gates and inverting the final output. The number of pipeline stages required can be calculated as $Stages = \lceil \log_4(num\_tokens) \rceil$. The number of LUTs required can be calculated as $LUTs = \sum_{i=1}^{Stages} \left\lceil \frac{num\_tokens}{4^i} \right\rceil$.

Since we cannot determine the cause of the error, the error signal is forwarded as a *set* signal to all of the registers in the parser as shown in figure 4. With all registers active, the parser can recover the grammar when the next token arrives.

## 3 Implementation

With the increasing popularity of XML on the Internet, high-speed XML parsers are needed now more than ever. The parser presented in this paper is well suited for such an application. In our previous work [4], we illustrated how our architecture can be used to create an XML parser for routing XML-RPC messages. We continue that example here so differences in the architectures can be easily compared.

Using our automatic VHDL generator, we generated hardware for five different size XML parsers utilizing both tokenizer architectures discussed in section 2.1. The grammars ranged in size from 300 to 3000 bytes.

### 3.1 Area and Performance

The hardware for each of the XML parsers was synthesized and placed and routed on the Xilinx Virtex 4

LX200 -11 chips. For synthesis, we used Synplicity's Synplify Pro v8.1. Placing and routing was completed using version 7.1 of the Xilinx back-end tools.

| | 8-bit chain | | | | 8-bit pipeline | | | |
|---|---|---|---|---|---|---|---|---|
| # of Bytes | Freq (MHz) | BW (Gbps) | LUTs | LUTs/ Byte | Freq (MHz) | BW (Gbps) | LUTs | LUTs/ Byte |
| 300 | 433 | 3.46 | 314 | 1.05 | 448 | 3.59 | 174 | 0.58 |
| 600 | 486 | 3.89 | 581 | 0.97 | 408 | 3.27 | 290 | 0.48 |
| 1200 | 426 | 3.41 | 1089 | 0.91 | 457 | 3.65 | 524 | 0.44 |
| 2100 | 408 | 3.26 | 1798 | 0.86 | 406 | 3.25 | 867 | 0.41 |
| 3000 | 289 | 2.32 | 2473 | 0.82 | 383 | 3.06 | 1204 | 0.40 |

**Table 1. Device utilization for XML parsers**

Results for the CFG parser architecture with error recovery utilizing both varieties of the tokenizer are shown in table 1. It is clear from looking at table 1 that the pipelined architecture is superior to the chained architecture in both size and speed. For small grammars, the chain architecture is comparable in speed, but requires much more space in the form of LUTs. The graph in figure 5, shows that the difference in the space requirement between the two tokenizers increases as the size of the grammar increases.

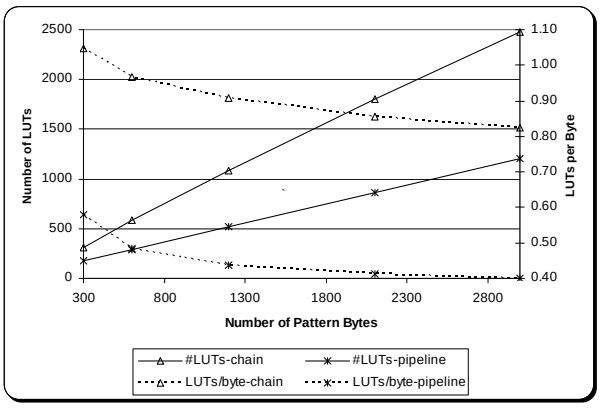


**Figure 5. LUTs vs. the number of bytes**

The number of LUTs per byte achieved by both architectures decreases as the size of the grammar increases. This is because as the size of the grammar increases, all of the encoder and decoder logic required by the architectures becomes a smaller and smaller percentage of the overall logic required. This means that the number of LUTs per byte for both architectures should asymptotically approach some value representing the minimum space requirements achievable by the architecture as shown in figure 5. The pipelined architecture is far more space efficient than the chain architecture requiring only .40 LUTs/byte for the 3000 byte grammar. This is less than half the size of the chain architecture which achieves .82 LUTs/byte for the same grammar.

## 4 Conclusions

In this paper we expanded on our previous work of using CFGs to generate hardware based pattern matching engines that can also provide contextual meaning for tokens. Two different methods for tokenization were explored with the pipelined character grid being both faster and more space efficient than regular expression chains. We also augment the architecture with the ability to detect and recovery from errors. Our architecture is capable of processing an 8-bit wide data stream at over 3.5 Gbps on a Xilinx Virtex 4 FPGA. The architecture is also capable of achieving a space requirement as low as .40 LUTs/byte.

## References


[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles and Techniques and Tools*. Addison-Wesley, 1986.

[2] Z. K. Baker and V. K. Prasanna. A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004. IEEE.

[3] Y. H. Cho and W. H. Mangione-Smith. Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2005. IEEE.

[4] Y. H. Cho, J. Moscola, and J. W. Lockwood. Context-Free-Grammar based Token Tagger in Reconfigurable Devices. In *International Workshop on Semantics enabled Networks and Services (SeNS)*, Atlanta, GA, Apr 2006. IEEE.

[5] Y. H. Cho, S. Navab, and W. H. Mangione-Smith. Specialized Hardware for Deep Network Packet Filtering. In *12th Conference on Field Programmable Logic and Applications*, pages 452–461, Montpellier, France, September 2002. Springer-Verlag.

[6] C. R. Clark and D. E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 956–959, Lisbon, Portugal, 2003.

[7] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*, Stanford, CA, August 2003. IEEE Computer Society Press.

[8] H. Song and J. W. Lockwood. Efficient Packet Classification for Network Intrusion Detection using FPGA. In *ACM International Symposium on FPGAs*, Monterey, CA, February 2005. ACM.