Master Thesis

FPGA-accelerated pattern matching over streams of events

Author(s):
Woods, Louis

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005906519

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master Thesis

FPGA-Accelerated Pattern Matching over Streams of Events

by

Louis Woods

March 2nd 2009 - August 31st 2009

Supervision:
Dr. Jens Teubner
René Müller
Prof. Gustavo Alonso

Information and Communication Systems Research Group
Institute for Pervasive Computing

Department of Computer Science
Swiss Federal Institute of Technology (ETH), Zurich, Switzerland
ETH-Zentrum, IFW
CH-8092 Zürich
Abstract

High performance computing, for example as required in the financial sector, is constantly confronted with growing data volumes and an increasing need for fast results. Due to their highly parallel nature and despite their lower clock frequencies, field-programmable gate arrays (FPGAs) can outperform conventional CPUs on certain tasks. One such task with significant potential for FPGA acceleration is pattern matching with regular expressions. In this thesis we investigate different techniques for implementing regular expression engines on the FPGA. We begin with the most common application of regular expressions, namely pattern matching in texts, e.g. as can be performed by grep — the popular Unix text search utility. Later we generalize the problem of matching patterns in texts to pattern matching in streams of items that can have arbitrary data types. As an example application, we employ our regular expression engines to do click stream analysis on a network stream. For specifying such general purpose regular expression engines, we use a declarative language that combines elements from stream processing languages with the widely recognized core syntax of regular expressions. Finally, we have developed a compiler that can translate the aforementioned language into actual circuit specifications, e.g. VHDL code, that can be used to configure an FPGA with the desired pattern matching engine.
Zusammenfassung

Preface

This thesis is submitted for partial fulfillment of the requirements of the degree Master of Science in Computer Science at the Swiss Federal Institute of Technology (ETH) Zurich. The thesis was derived during a six month project from March 2nd to August 31st 2009 in the Information and Communication Systems Research Group led by Prof. Gustavo Alonso at the Institute of Pervasive Computing at the Department of Computer Science of the Swiss Federal Institute of Technology Zurich.
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3 Implementing the Previous-Expression</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Differences between GRe and REs</td>
<td>24</td>
</tr>
<tr>
<td>3.4.1 Simultaneously Satisfiable Predicates</td>
<td>24</td>
</tr>
<tr>
<td>4 Dispatching Events to Multiple Replicated GRe</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Motivation</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Solution Statement: Dispatcher</td>
<td>28</td>
</tr>
<tr>
<td>4.3 Replicating GRe</td>
<td>28</td>
</tr>
<tr>
<td>4.4 Content-Addressable Memory (CAM)</td>
<td>29</td>
</tr>
<tr>
<td>4.5 Updating Content-Addressable Memory</td>
<td>30</td>
</tr>
<tr>
<td>4.5.1 CAM Memory Types</td>
<td>30</td>
</tr>
<tr>
<td>4.5.2 Buffering Event Instances in a FIFO Queue</td>
<td>30</td>
</tr>
<tr>
<td>4.5.3 Detailed Timing Analysis</td>
<td>31</td>
</tr>
<tr>
<td>4.6 Allocating Free GRe</td>
<td>33</td>
</tr>
<tr>
<td>4.7 Determining when a GRE is Free</td>
<td>34</td>
</tr>
<tr>
<td>5 Click Stream Pattern Matching</td>
<td>37</td>
</tr>
<tr>
<td>5.1 Click Stream Analysis</td>
<td>37</td>
</tr>
<tr>
<td>5.2 Click Stream Pattern Examples</td>
<td>38</td>
</tr>
<tr>
<td>5.2.1 Stream Declaration</td>
<td>39</td>
</tr>
<tr>
<td>5.2.2 Predicate Declarations</td>
<td>39</td>
</tr>
<tr>
<td>5.2.3 Pattern Declarations</td>
<td>40</td>
</tr>
<tr>
<td>5.3 Regular Expression Engine Generator</td>
<td>40</td>
</tr>
<tr>
<td>5.4 Integration with ISE and EDK</td>
<td>43</td>
</tr>
<tr>
<td>5.4.1 Integration with ISE Project</td>
<td>43</td>
</tr>
<tr>
<td>5.4.2 Integration with EDK Project</td>
<td>44</td>
</tr>
<tr>
<td>5.4.3 Network Interface</td>
<td>45</td>
</tr>
<tr>
<td>5.5 Open Issues and Future Work</td>
<td>46</td>
</tr>
<tr>
<td>6 Measurement</td>
<td>47</td>
</tr>
<tr>
<td>6.1 Measurement Units</td>
<td>47</td>
</tr>
<tr>
<td>6.2 State Explosion &amp; Resource Consumption</td>
<td>47</td>
</tr>
<tr>
<td>6.3 Multilevel Web Form Example</td>
<td>49</td>
</tr>
<tr>
<td>6.4 Timing Constraint Violations</td>
<td>50</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>7.1 Regular Expression Matching in FPGAs</td>
<td>51</td>
</tr>
<tr>
<td>7.2 Pattern Matching in Streaming Systems</td>
<td>51</td>
</tr>
<tr>
<td>7.3 Future Work</td>
<td>52</td>
</tr>
<tr>
<td>A Regular Expression Syntax</td>
<td>53</td>
</tr>
<tr>
<td>B Glossary of Abbreviations</td>
<td>55</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In Section 1.1 of this introductory chapter, we first give some background information and a brief overview of the area of topics and technologies that we will be concerned with in this thesis. Then, in Section 1.2 we describe the main objectives we intend to complete with this work. Related work is covered in Section 1.3 and we sketch the outline of the thesis in Section 1.4.

1.1 Overview

A field-programmable gate array (FPGA) is an integrated circuit that can be configured (programmed) and re-configured after manufacturing. It can be used to implement any logic function that could be performed by a hardwired circuit, but has the great advantage that its functionality can be updated after shipping. Compared to applications running on a CPU, applications implemented on an FPGA can benefit from massive parallelism, fast on-chip communication, and application-specific hardware designs, potentially allowing for superior performance. Nevertheless, developing applications for FPGAs is considerably more difficult and time-consuming than writing software for a general purpose processor.

Regular expressions date back to the 1950s and have their origins in theoretical computer science, that is to say in automata theory and formal language theory. Today, regular expressions are supported by various scripting and programming languages as well as operating systems and search tools, such as grep on Unix. A regular expression can be processed using a finite automaton, which makes them suitable for FPGAs, since state machines can be implemented directly in hardware not requiring external memory and the like. The inherent parallelism of FPGAs can then be exploited, e.g. to process multiple regular expressions concurrently.

In recent years, complex event processing (CEP) applications dealing with data streams, i.e. data composed of a continuous, real-time sequence of items, have been given substantial attention to in both industry and the academic community. In many of those applications, pattern matching over data sequences has become a key requirement, e.g. in finance there is great interest to detect patterns in the market that indicate a beneficial moment for trade.
1.2 Problem Statement

A main objective of this thesis was to build a compiler that can transform regular expressions into circuits on an FPGA. More precisely, the compiler should output circuit specifications, e.g. in the hardware description language VHDL, that could then be used to configure an FPGA using the appropriate tools. Since there are many well-understood ways to implement regular expression engines in software, a first milestone was to apply these techniques to hardware, adapting them where necessary. In a second phase, it was desired to clarify whether more hardware-tailored methods for implementing regular expression engines exist, and potential optimizations should be investigated.

Another major goal of this thesis was to examine how these regular expression engines could be applied in streaming systems. This included further development of the circuits, making them more general to process arbitrary data streams, as well as elaborating and analyzing potential new problems. A particular focus was to support a concrete use case that would demonstrate the regular expression engines embedded in a real application and show how the inherent parallelism of FPGAs could be exploited. Finally, resulting performance characteristics should be evaluated and relevant metrics and limitations discussed, e.g. latency and space-performance trade-offs.

1.3 Related Work

1.3.1 Regular Expressions in Hardware

*Non-deterministic finite automata* (NFA) were introduced by Rabin and Scott \[13\] in 1959, who also proved that for every non-deterministic finite automaton there exists an equivalent *deterministic finite automaton* (DFA). Thompson \[18\] described one of the first methods, known as Thompson’s algorithm, to convert regular expressions into NFAs. We will discuss Thompson’s algorithm in detail in Chapter 2. A method for converting NFAs to equivalent DFAs can be found in \[9\]. Floyd and Ullman \[7\] first studied the implementation of NFAs in hardware showing that an $N$-state NFA requires at most $O(N)$ circuit area to implement. Sidhu and Prasanna \[15\] later presented an NFA-based implementation of regular expressions in FPGAs. Yang et al. \[19\] have slightly modified the architecture used in \[15\] to a more modular structure and they also propose an elegant way to perform multi-character input matching. To our knowledge there have been only a few approaches to implement regular expression engines in hardware as DFAs. One example, however, can be found in \[12\]. In the area of Network Intrusion Detection Systems (IDS), e.g. SNORT and BRO, there has been quite some research for implementing regular expressions on the FPGA \[10\]. There have been several proposals to optimize constraint repetitions \[16, 3, 6\], e.g. in \[16\] shift registers and counters are used to efficiently deal with single-character repetition blocks. Clark and Schimmel \[4\] first used a 8-to-256 ASCII character pre-decoder to share the character comparators among the states of their NFA implementations and thus reducing hardware resources.
1.4 Outline of the Thesis

1.3.2 Stream Processing

The syntax for declaring the schema of an event stream that we will use in Chapter 3 has been adopted from StreamBase [17] and their StreamSQL language. In [20] a declarative pattern matching language standard has been proposed. Our syntax for defining predicates and patterns over those predicates strongly resembles parts of that proposal. Nevertheless, our language is less comprehensive and therefore a lot simpler. Software implementations of the language proposed in [20] have been presented by [1] and [5]. In [1] a so called NFA is introduced that works with buffers for storing all the overlapping matches. In [5] an extension to MySQL has been developed which is based on the same ideas as in [1]. In Glacier [11], a component library and compositional compiler that transforms continuous queries into logic circuits, content-addressable memory (CAM) was used to implement the grouping operator. In Chapter 4 we will make use of CAM to build an event dispatcher.

1.4 Outline of the Thesis

In Chapter 1 we have provided a short introduction to FPGAs, regular expressions and CEP applications. We have described the major goals of this thesis and have pointed out related work from the various associated fields of research.

In Chapter 2 we analyze different techniques for implementing regular expression engines on the FPGA. In particular, we compare implementations as deterministic finite automata (DFAs) versus non-deterministic finite automata (NFAs). We give insight on how to compile regular expressions and translate them to hardware circuits and also present some possibilities for optimization.

In Chapter 3 we extend our regular expression engines for utilization in CEP applications. We show how to decouple symbol decoding from pattern matching and why this leads to more flexible and space efficient regular expression engines. Additionally, this chapter covers a declarative language to specify general purpose regular expression engines and discusses implementation details of this language.

In Chapter 4 we tackle a new kind of problem that may arise when doing pattern matching over event streams. Namely, executing the regular expression engine on a selected set of items in the stream. We propose a solution to this problem by introducing a piece of hardware that we will call event dispatcher.

In Chapter 5 we demonstrate our regular expression engines in action. We present an example application where we will do real-time analysis of click stream traffic and detect click patterns on a per-user basis. In this application we combine many of the concepts discussed in this thesis and show how the FPGA can directly access the network interface and process Ethernet frames at wire speed.
In Chapter 6, we discuss the results of the experiments that we have carried out to evaluate the circuits generated by our compiler. In these experiments we measured FPGA resources, such as flip-flops and look-up tables, consumed by our pattern matching engines and performed some scalability tests.

In Chapter 7, we conclude this thesis with a brief summary of the most important aspects of our work and an outlook to potential future work.
Chapter 2

Regular Expressions in FPGAs

The focus of this chapter is on methods for compiling regular expressions to hardware circuits. Section 2.1 gives a brief introduction to regular expressions and regular languages. We have developed a regular expression compiler that generates regular expression engines (REs) specified in the hardware description language VHDL. Section 2.2 explains some basics of this compiler, such as parsing regular expressions. A regular expression can be implemented as deterministic finite automaton (DFA) or as non-deterministic finite automaton (NFA), see Section 2.3. Our compiler supports both representations, as we discuss in Section 2.3 and 2.4 respectively. Section 2.5 compares the two approaches. An optimization for NFAs, called prefix sharing, is presented in Section 2.6 and a technique to increase throughput is described in Section 2.7. Finally, in Section 2.8 we highlight the difficulty to extend regular expressions with backreferences in hardware.

2.1 Regular Languages and Regular Expressions

A regular language \[ \Sigma \] is a set of finite sequences of symbols from a finite alphabet. For example the regular language \( L = \{a, ab, abb, \ldots\} \) over the alphabet \( \Sigma = \{a, b\} \) contains all strings starting with an ‘a’ followed by zero or more ‘b’s. A regular expression is a declarative way of expressing the strings contained in such a language, e.g. the language \( L \) above can be described by the regular expression \( a \cdot b^* \). Here, and throughout this chapter, we can think of the symbols in regular expressions as characters. However, in Chapter 3 we will show how our REs can be generalized to handle any kind of symbols. In the example application, discussed in Chapter 5 for example, symbols will represent different http-requests to a web server and our REs will detect different click patterns of web users.

From theoretical computer science it is known that the words of a regular language can be recognized by a deterministic finite automaton (DFA) or by a

\[ \text{concatenation} \]
non-deterministic finite automaton (NFA). Whereas a DFA never has more than one concurrently active state, in an NFA several or even all states can be active in parallel. In software implementations of REs this may lead to performance problems when too many states are active at the same time. On the FPGA this is not the case. In the following sections we will investigate both variants, REs implemented as DFAs and as NFAs, respectively.

All operators of regular expressions can be reduced to three basic operations on languages: concatenation (\(-\)), union (\(|\)) and Kleene closure (\(\ast\)). Any other regular expression operator (see Appendix A for a more comprehensive list) can always be reduced to a combination of these three basic operators. For instance, the regular expression \(a^+\) is equivalent to \(a \cdot a^\ast\). Therefore, we will focus especially on these three fundamental operators in the following. The regular expression operators relevant for this chapter are listed in Table 2.1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\cdot)</td>
<td>Concatenation</td>
<td>First matches item on the left, then matches item on the right.</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
<td>Union</td>
</tr>
<tr>
<td>(\ast)</td>
<td>Kleene closure</td>
<td>Matches the previous item zero or more times.</td>
</tr>
<tr>
<td>(\cdot)</td>
<td>Kleene plus</td>
<td>Matches the previous item once or more.</td>
</tr>
<tr>
<td>({n})</td>
<td>Exactly</td>
<td>Matches the previous item exactly (n) times.</td>
</tr>
<tr>
<td>([\ ])</td>
<td>Symbol class</td>
<td>Matches a single symbol out of all the possibilities offered by the symbol class.</td>
</tr>
<tr>
<td>.</td>
<td>Wildcard</td>
<td>Matches any single symbol.</td>
</tr>
</tbody>
</table>

Table 2.1: Regular expression operators used in this chapter

2.2 A Regular Expression to VHDL Compiler

As mentioned in the introduction to this chapter, we have developed a compiler that can translate a regular expression to VHDL code defining a finite automaton that can recognize the language specified by the regular expression. Our compiler can generate REs either as DFAs or as NFAs. In this section the part of the compiler that is independent of either representation is explained. As in compilers for programming languages, it is helpful to convert the regular expression into an abstract syntax tree (AST) for further processing. Our compiler is written in Java and we used the lexical analyzer JLex\(^2\) together with the LALR parser generator CUP\(^3\) to generate the parser. This parser converts a given regular expression into an AST. In Figure 2.1 the AST for the regular expression \(a \cdot b\{2\}\) is displayed, as an example.

An AST is a convenient intermediate representation for the regular expression. Applying the Visitor design pattern\(^4\) we can traverse the tree and perform arbitrary operations on it. For example, we use a visitor to rewrite those nodes of the tree that do not need extra optimization later. Recall that any regular

\(^2\)http://www.cs.princeton.edu/ appel/modern/java/JLex/
\(^3\)http://www2.cs.tum.edu/projects/cup/
2.3 Deterministic Finite Automaton (DFA)

Expression operator can be reduced to a combination of the three basic operators: concatenation (\(\cdot\)), union (\(|\)\)), and Kleene closure (\(\ast\)). The AST in Figure 2.1 contains a constraint repetition (\(b^2\)), which is unrolled by our rewrite-visitor. The resulting AST is depicted in Figure 2.2.

2.3 Deterministic Finite Automaton (DFA)

From the AST, derived in Section 2.2, we will now construct a DFA. However, directly building a DFA from the AST is not trivial. Therefore, we will first generate an equivalent NFA using a slightly modified version of Thompson’s algorithm [18]. Then we will convert the NFA to a DFA applying the Subset Construction Algorithm [9].

2.3.1 Thompson’s Algorithm

Thompson’s algorithm [18] is a modular and straightforward way to transform a regular expression into an NFA. The NFA for a regular expression is built up from partial NFAs, each corresponding to a subexpression. The algorithm specifies a few transformation-rules for converting single symbols and the three basic regular expression operators: concatenation (\(\cdot\)), union (\(|\)\)), and Kleene closure (\(\ast\)). Figure 2.3 depicts the NFA recognizing the symbol ‘a’. S is the start state and F the accepting (finishing) state, indicated by the double circle.

The concatenation of two subexpressions can be realized by connecting their corresponding NFAs with an \(\epsilon\)-transition, as illustrated in Figure 2.4. An \(\epsilon\)-transition is one that does not consume a symbol. The start state (\(S'\)) of the first child NFA becomes the start state of the new higher level NFA and the accepting state (\(F''\)) of the second child NFA will be the new finishing state. Notice, we did not draw any transitions inside the child NFAs, which are expressed through the dashed boxes. This is because the NFAs being concatenated can be treated as black boxes. For the construction to work, all we need to know about the child NFAs are their start and finishing states.

The union of two subexpressions is created similarly to the concatenation. Nevertheless, here new start and accepting states (\(S,F\)) must be created, see Figure 2.5. The new start state has two \(\epsilon\)-transitions, one to each start state.

Figure 2.1: AST for \(a \cdot b^2\)  
Figure 2.2: Unrolled AST for \(a \cdot b^2\)
8 Regular Expressions in FPGAs

![Figure 2.4: Concatenation of two NFAs](image)

of the respective child NFA. Observe, the non-determinism that is introduced here: when \( S \) is activated, \( S' \) and \( S'' \) will be active at the same time.

![Figure 2.5: Union of two NFAs](image)

Finally, the construction rule for the last of the three fundamental regular expression operators, Kleene closure, is given in Figure 2.6. Again, new start and finishing states are introduced and several \( \epsilon \)-transitions added.

![Figure 2.6: Kleene closure of an NFA](image)

### 2.3.2 Applying Thompson’s Algorithm

To apply Thompson’s algorithm to the AST from Section 2.2, we can again make use of the Visitor design pattern [8]. A visitor traverses the AST and recursively converts every node of the AST to an NFA, starting with the leaf nodes. At an inner node the visitor first visits the children of that node receiving a corresponding child NFA for each child. From these child NFAs a new bigger NFA can be created, according to the rules of Thompson’s algorithm until at the root node, one large NFA remains representing the entire regular expression.

### 2.3.3 Modified Thompson’s Algorithm

As explained in Section 2.1 the three basic operators (·, |, and *) suffice to express any regular expression. Therefore, the transformation-rules presented in Subsection 2.3.1 are complete. Nevertheless, as an optimization, we added two extra transformation-rules to Thompson’s algorithm, concerning symbol
2.3 Deterministic Finite Automaton (DFA)

Classes \(\text{class}^4\) and wildcard symbols. In our AST, besides representing single symbols, leaf nodes can also represent symbol classes or wildcard symbols. The NFA for a symbol class is depicted in Figure 2.7. It is similar to the NFA for a single symbol, but has a transition for every symbol in the symbol class. In Figure 2.8 we show the NFA for the wildcard operator. Here we have introduced a new any-transition. This transition needs to be dealt with separately when converting from NFA to DFA, see Subsection 2.3.4.

\[ \begin{array}{c}
\text{Start} \\
\ \ \ S \\
\ a \ \\
\ b \ \\
\ c \\
\ F \\
\end{array} \]

Figure 2.7: NFA for \([abc]\)

\[ \begin{array}{c}
\text{Start} \\
\ \ \ S \\
\ a \\
\ \ F \\
\end{array} \]

Figure 2.8: NFA for wildcard operator

2.3.4 NFA to DFA Conversion

The next step is to convert the generated NFA to an equivalent DFA. The subset construction (or powerset construction) is a standard method for doing so. Roughly speaking, the subset construction creates a new state for any possible permutation of active states and thereby eliminates \(\epsilon\)-transitions and all other ambiguous transitions, see [9] for more details. Remember, in the previous section one of our optimizations to Thompson’s algorithm was to introduce a special any-transition; a single transition that would match any symbol, as opposed to having a distinct transition for every symbol in the alphabet. This change has consequences for the subset construction. Consider the NFA in Figure 2.9.

\[ \begin{array}{c}
\text{Start} \\
\ \ \ S_0 \\
\ a \\
\ S_1 \ \\
\ S_2 \\
\end{array} \]

Figure 2.9: NFA with any-transition (.)

\[ \begin{array}{c}
\text{Start} \\
\ \ \ S_0 \\
\ a \\
\ S_1, S_2 \\
\ S_2 \\
\end{array} \]

Figure 2.10: DFA with transformed any-transition (\(\sim a\))

On the one hand, by consuming the symbol ‘a’ the states \(S_1\) and \(S_2\) can be reached. On the other hand, consuming any symbol other than ‘a’ leads us to the state \(S_2\). We thus need to make the right modifications to the subset construction, for it to produce the DFA presented in Figure 2.10.

2.3.5 DFA as Switch-Case Statement

As a last step, the DFA needs to be turned into VHDL code. This step is straightforward, since the DFA obviously is a normal finite state machine (FSM) that can be written as one VHDL switch-case statement. Every state of the DFA produces a new case-block, see Listing [2.1]5.

\[ \text{Character classes}^4 \text{ in regular expressions speak; we prefer “symbol classes” as the more general term}\]

\[ \text{The datavalid signal will be explained in Section [2.4.1] and can be ignored for now.}\]
case current_state is
  when STATE_1 =>
    if ((datavalid = '1') and (symbol = 'a')) then
      next_state <= STATE_2;
    elsif ((datavalid = '1')) then
      next_state <= STATE_3;
    end if;
  when STATE_2 =>
    ...
end case;

Listing 2.1: DFA as switch-case statement

Within each case-block every outgoing transition of the respective state is emitted in the form of an if-condition. If the state has a converted any-transition, i.e. a transition that matches any symbol that was not matched by any of the explicit transitions, than that transition can be emitted last as an else-block, like the one on line five in Listing 2.1.

2.3.6 State Explosion

DFAs have one major disadvantage compared to NFAs. Namely, DFAs can suffer from severe state explosion; in the worst case a DFA can have up to $2^n$ states compared to a corresponding NFA with only $n$ states, see [9] for a complete proof. The following example comes close to this upper limit. Consider the language $L$, which is defined as the set of all strings over an alphabet $\{0,1\}$ with a '1' at the $i$th position starting from the end of the string. This language can be described by the following regular expression $(0|1)^* \cdot 1 \cdot (0|1)^{\{i\}}$. Figure 2.11 illustrates the NFA that can recognizes this language with $i = 2$ for presentability reasons. This NFA has four states and the number of states grows linearly with respect to $i$, e.g. $n = 1 + (i + 1)$, where $n$ is the number of states. The equivalent DFA that accepts this language, depicted in Figure 2.12 needs to remember the last $i$ (2 in this case) symbols it has read. Since there are $2^i$ permutations of a bit-string with length $i$ the DFA must contain at least $2^i$ different paths, e.g. $(q_0 q_1 q_1 q_0 q_0 q_0 q_{100}, q_0 q_1 q_1 q_1 q_0 q_{101}, q_0 q_1 q_1 q_0 q_{110}, q_0 q_1 q_1 q_{111})$ in the example. Thus, the number of states of a DFA that can recognize the language above grows exponentially relative to $i$, i.e. $n = 1 + (2^{i+1} - 1)$, where $n$ is again the number of states.

Figure 2.11: NFA for: $(0|1)^* \cdot 1 \cdot (0|1)^{\{2\}}$

In Chapter 3 we will show another type of state explosion due to multiple symbols matching at the same time. This will be possible in that scenario, because the symbols will be representing predicates defined over attributes of tuples and overlapping matches will be allowed.
2.4 Non-Deterministic Finite Automaton (NFA)

NFAs do not suffer from state explosion. On the FPGA, unlike in software, we can process all concurrently active states in parallel. In fact, the inherent parallelism of FPGAs is one of their main advantages over commodity systems running on a sequential processor which we want to exploit. Thus, in this section we show how the Thompson’s algorithm, explained in the previous section, can be directly mapped to hardware. The result will be an RE implemented as NFA. The construction we are proposing here is inspired by the work of Yang et al. [19].

2.4.1 Modular NFA Construction

As in Section 2.3 we will start by generating circuits corresponding to subexpressions and then connect these circuits appropriately until ultimately we have implemented the entire regular expression. The key element in this construction is a single building block that represents a state in the NFA. This building block is configurable and can be connected to other building blocks. Constructing the NFA now breaks down to instantiating and configuring as many building blocks as there are states in the NFA, and wiring them according to Thompson’s algorithm.

In Listing 2.2 the entity declaration of the building block is given. It has five input ports. There is a clock and reset port since our building block is a synchronized entity. Through the activation port the building block can be activated. A building block can only consume a symbol once it has been activated, i.e. when the preceding building block has accepted its symbol. Notice that the width of the activation port can be configured by the parameter widthA. This is because it is possible that multiple building blocks can activate the current one as is the case for example with the union operator (see below). As will be further explained in Chapter 3, we separate symbol decoding from regular expression matching. A symbol decoder translates every incoming symbol to an individual wire which can than be connected to the building block. A building block is configured to consume a specific symbol by connecting the building block to the appropriate wire from the decoder. Additionally, we need a datavalid signal that indicates that a symbol was indeed decoded since in our scenario a new symbol will not necessarily be decoded in every clock cycle. Finally, also the
width of the symbol port can be configured with the parameter \texttt{widthS}. The purpose thereof is to support symbol classes, as we will show below.

```vhdl
entity state is
generic (  
  widthA : positive;  
  widthS : positive  
);
port (  
  clock : in std_logic;  
  reset : in std_logic;  
  activation : in std_logic_vector(widthA-1 downto 0);  
  datavalid : in std_logic;  
  symbol : in std_logic_vector(widthS-1 downto 0);  
  output : out std_logic  
);
end state;
```

Listing 2.2: Basic building block for constructing an NFA

How a building block can be configured to accept a single symbol, say a character ‘a’, is shown in Figure 2.13. The activation signal is permanently set to logic high, because we have only one building block, i.e. any character ‘a’ matches the regular expression independently of other previous matches. The bar below the building block represents the symbol decoder (see Chapter 3) responsible for generating the \texttt{datavalid} and \texttt{symbol} signals. Whenever the building block matches, its \texttt{output} signal is set to logic high. The match is stored in a flip-flop for one clock cycle.

Instead of having the building block consume a single symbol, it is possible to configure the building block to accept an entire class of symbols. The building block in Figure 2.14 is configured to accept any of the symbols ‘a’, ‘b’ or ‘c’. This is accomplished by setting the parameter \texttt{widthS} to the number of symbols in the class and then routing the relevant signals from the symbol decoder to the \texttt{symbol} port. The building block matches if any of the signals at the \texttt{symbol} port is a logic high. Obviously, we could also realize a symbol class by a union of building blocks configured for single symbols. However, this is a far more space efficient way of doing it.

![Figure 2.13: Accepting ‘a’](image)

![Figure 2.14: Accepting [a-c](image)

Given the building block configuration of Figure 2.14 a negated symbol class, e.g. 
\([\hat{a-c}]\) is straightforward to implement. All we need to change is the condition when the building block matches. Instead of matching any of the symbols in the class the condition inside the building block now needs to make sure that none of the signals at the \texttt{symbol} port carry a ‘1’. Our compiler makes this
configuration by selecting a different architecture for the building block entity at compile time.

At last, also the wildcard operator (.), which matches any symbol, can be accomplished by configuring a single building block. In this case the compiler sets the signal at the symbol port permanently to '1'. The effect is that when the building block is activated the matching condition now depends solely on the datavalid signal, i.e. any decoded symbol matches.

Next we will look at operators that involve more than one symbol. For reasons of presentation we have omitted the clock, reset, and the datavalid port of the building blocks in Figure 2.15, 2.16, and 2.17. The concatenation of two symbols is illustrated in Figure 2.15. Two building blocks are connected sequentially. When the first signal in the concatenation was matched by the building block on the left hand side, that building block’s output signal will be '1' during the next clock cycle. The building block on the right hand side is activated by the output signal of the preceding building block, i.e. the building block is activated when ‘a’ was matched and can now consume ‘b’.

The union operator is depicted in Figure 2.16. Both building blocks in the union are activated by the preceding building block for symbol ‘x’. Note how the non-determinism is handled in parallel here; the output signal from the first building block is fanned out and routed to the activation ports of both building blocks inside the union concurrently. When a building block follows a union, as the building block for symbol ‘y’ does in the figure, then all of the building blocks inside the union can potentially activate that building block, i.e. need to be connected to the activation port of that building block. This is where parameter widthA comes into play; the width of the activation port needs to be set to the number of building blocks in the union. In the example, widthA would be set to two for the building block accepting ‘y’.

The most complex operator is Kleene closure for which an example is given in Figure 2.17. In the example the building block for ‘a’ is initially activated by the building block for ‘x’. Once the building block inside the Kleene closure has matched a symbol for the first time, it can activate itself. That is why the
output signal of the building block for ‘a’ is connected to its own activation port. However, this is not yet the whole story since the regular expression inside a Kleene closure does not need to be matched at all; recall Kleene closure means zero or more occurrences. Therefore, it is necessary that building blocks that can activate the building block in the Kleene closure also can activate building blocks directly following the Kleene closure, as it is done for the building block matching ‘y’ in the example.

In this section we have presented how the basic operators of regular expressions can directly be translated to circuits on the FPGA. In our examples the operators concatenation (·), union (|), and Kleene closure (∗) only contain simple symbols. However, implementing more complex regular expressions, e.g. \( x \cdot (a|b)^* \cdot y \), is straightforward and just a matter of connecting the right output signals to the proper activation ports of the involved building blocks. Thus, in \( x \cdot (a|b)^* \cdot y \) the output signal of the building block for ‘a’ can activate the building block for ‘b’ and vice versa, since the union a|b is inside the Kleene closure. In Section 2.4.2 we give more insight into how our compiler can figure out which building blocks to connect.

### 2.4.2 Activation Signal Routing

Our compiler determines how to connect the building blocks in performing two additional preprocessing steps on the AST, which represents the regular expression. In the first phase a visitor assigns all of the output signals to the relevant nodes and in the second phase all activation signals are assigned. Observe that we need to assign signals only to the leaf nodes, because only those will later be translated to building blocks; all inner nodes represent regular expression operators, such as concatenation (·), union (|) and Kleene closure (∗). Figure 2.18 (a) shows the AST for the regular expression \( x \cdot (a|b)^* \cdot y \) with output signals \( S_0 \) to \( S_3 \). While traversing the entire AST the visitor generates and assigns a new output signal, whenever the visitor encounters a new leaf node.

Now that the output signals for all of the leaf nodes are defined, a second visitor can link those output signals to the activation port of the right building blocks; i.e. leaf nodes. The visitor traverses the AST in preorder. While going down in the tree the visitor propagates the activation signals to be assigned to the leaf nodes at the bottom. Going up in the tree the visitor collects new output signals, which in return will be assigned to the next leaf nodes when going down again. As opposed to the first visitor, this one needs to consider the inner nodes as well, since they determine how output and activation signals are propagated. In Figure 2.18 (b), in addition to the output signals, the activation signals have been added. The set of activation signals is initialized to {‘1’}. The first node visited (node 1) represents a concatenation; the set of activation...
signals is forwarded to the node’s left child (node 2). This is the leaf node representing the building block matching symbol ‘x’. The activation signals from the set received are assigned. In this case this is just ‘1’. Note that the building block for accepting ‘x’ is the first building block in our circuit. This building block should always be active, as motivated in Subsection 2.4.1, therefore the activation signal ‘1’ make sense here. Node 2 then returns its output signal to its parent (node 1) which creates a new set of activation signals \{S_0\} and forwards them to its right child (node 3). Unlike the concatenation operator, the union operator (node 4) forwards the set of activation signals to both children and then collects the output signals of both children to create the new set of activation signals \{S_1, S_2\} which is propagated up the tree to node 3 and from there down again to node 7.

### 2.5 DFA- vs. NFA-Approach

In this chapter we have examined the internals of our regular expression compiler. We have shown techniques for implementing REs on the FPGA both as DFAs and as NFAs. This section discusses advantages and disadvantages of either representation.

The main advantage of DFAs over NFAs in software implementations, namely that only one state can be active at any given point in time, does not, at first glance, bring any obvious benefits for implementations on the FPGA. Conversely, NFAs do not suffer from state explosion which can be a great advantage, especially in terms of space efficiency. Nevertheless, there is one aspect of the NFA-approach that could have a negative impact on performance. Recall, that a building block can be activated by several other building blocks. If there are a lot of those building blocks, e.g. as would be the case for a large union, then this will lead to an equally large logic OR-operation inside the building block being activated. This becomes a problem once the logic OR-operation cannot be evaluated in a single clock cycle anymore, since proper evaluation would then require pipelining and, as a consequence, increase the latency of the building block.

Generally speaking, an NFA is often simpler and more resembling the regular expression it has been derived from than its DFA-pendant. This can have positive effects in terms of both optimization and debugging. In the following two
sections we show two optimizations that can be easily applied to NFAs. One is prefix sharing (Section 2.6) and the other multiple symbol processing (Section 2.7).

### 2.6 Prefix Sharing with NFAs

When multiple regular expressions share a common prefix, one NFA could be generated for the prefix. The output signal of that NFA could then be fanned out and routed to all of those NFAs that represent the remaining part of the respective regular expression sharing that prefix. In Figure 2.19 two NFAs with prefix sharing for the regular expressions \( a \cdot b \cdot c \) and \( a \cdot b \cdot d \) are depicted.

![Figure 2.19: NFAs with prefix sharing for regular expressions \( a \cdot b \cdot c \) and \( a \cdot b \cdot d \)](image)

Implementing prefix sharing in a DFA scenario would be a lot more complex and less modular since multiple states would need to be linked resulting in a larger DFA comprising all regular expressions sharing the common prefix.

### 2.7 Multiple Symbol Processing with NFAs

Yang et al. [19] proposed a convenient method to make their NFAs process more than one symbol per clock cycle. Figure 2.20 illustrates an NFA representing the regular expression \( P \cdot O \cdot S \cdot T \). The dashed boxes denote the building blocks corresponding to the subexpression, where each subexpression matches one of the symbols ‘P’, ‘O’, ‘S’, or ‘T’. Each building block consists of some combinatorial logic, indicated by the circles, that determines whether the block matches or not and a flip-flop to store the result of the combinatorial logic for one clock cycle. This circuit needs four cycles to match the entire regular expression.

![Figure 2.20: NFA for regular expression \( P \cdot O \cdot S \cdot T \), one symbol per clock cycle](image)

In order to process more than one symbol per clock cycle, the combinatorial part of the NFA is replicated. We need as many replicas as we want to process symbols per cycle, e.g. in Figure 2.21 an NFA is displayed that can process four symbols per cycle. Please note, we have modified the way the signals are connected.
Assume the next four symbols to process are “POST”. Then ‘P’ will be matched by the top NFA, in Figure 2.21, ‘O’ will be matched by the second NFA, ‘S’ by the third, and ‘T’ by the fourth. The path of activation signals starts in the upper left corner and propagates down to the lower right corner. On this path there is only combinatorial logic. Thus, the entire four symbols can be processed in one clock cycle.

![Figure 2.21: NFA for regular expression P·O·S·T, four symbols per clock cycle](image)

Now imagine that the next eight symbols to process are “XPOS” and “TYYY”. In this case, it is the second NFA from the TOP that matches ‘P’. After one clock cycle the subexpression P·O·S has been matched and this result is stored in the second flip-flop from the right. A cycle later the combinatorial block for matching ‘T’ of the top NFA is activated by this flip-flop and since the first symbol in “TYYY” is ‘T’, this block matches.

With the method just explained, we trade space for throughput. Theoretically, with enough space, an arbitrary high throughput could be achieved. However, in practice this method only scales up to a certain degree. Since every combinatorial block has a processing delay, we can only chain a limited amount of these blocks together to fit in one clock cycle.

### 2.8 Backreferences

In some software implementations of regular expressions, e.g. in the programming language Perl, regular expressions are extended by a special language construct, called backreferences. Backreferences allow the extraction of parts of a string that previously has matched some subexpression. The matching substring can then be referenced in the remaining part of the regular expression. Consider the regular expression /(a|b)\1/ The backreference is represented by \1 and it refers to the substring that has matched the subexpression in parenthesis (a|b), see [14] for more details on this syntax. The language defined by this regular expression contains the strings “aa” and “bb” but not “ab” nor “ba”.

Once regular expressions are extended with backreferences, the languages defined by those regular expressions are not necessarily regular anymore. For example, the extended regular expression /\(\ast\)\1/ defines the language L = \{ww | w ∈ ∑∗ \}, which is neither regular nor context-free, as can be shown with the Pumping Lemma [9]. In fact, in [2] it is proved that the problem of deciding

---

6Note, we use the PCRE syntax here, e.g. the slashes indicate a regular expression and the centered dot (·) for the explicit concatenation has been omitted.
whether a string contains a substring matched by a regular expression extended with backreferences is NP-complete, i.e. there is no polynomial-time algorithm for matching regular expressions with backreferences if $P \neq NP$ holds.

Nonetheless, backreferencing is often supported in software implementations of REs, as mentioned above. Supporting backreferences in hardware REs, however, is problematic. On the one hand, techniques for implementing backreferences, such as backtracking, are complex and imply excessive memory usage. On the other hand, conceivable real-time constraints can no longer be guaranteed once regular expressions are extended with backreferences, e.g. the constraint that an RE can process one symbol per clock cycle.
Chapter 3

General Purpose Regular Expression Engines

So far, we have considered the symbols in regular expressions to be equivalent to ASCII characters. However, pattern matching can be applied not only to textual data. In event stream processing interesting applications involving pattern matching exist, as we will exemplify further in Section 3.1. In this chapter we examine how the REs from Chapter 2 can be generalized for pattern matching over arbitrary types of data. We present a syntax for describing such general purpose regular expression engines (GREs) in Section 3.2. The implications for the compiler and resulting circuits on the FPGA are explained in Section 3.3 and in Section 3.4 we discuss differences between GREs and REs.

3.1 Pattern Matching over Event Streams

Complex Event Processing (CEP) is a concept that aims at inferring meaningful higher-level events from one or multiple streams of event data. Pattern matching is one technique for identifying such so-called complex events. As an example, imagine a store where the goods are equipped with RFID tags. RFID readers at different locations in the store continuously produce events, which represent some action related to the store’s products. Consider the following three simple events: \( A = \text{“product X has been removed from the shelf”}, \) \( B = \text{“product X has been payed for at the cash register”}, \) and \( C = \text{“product X has left the store”}. \)

The pattern \( A \cdot [\neg B] \cdot C \) defines a complex event: after product X was removed from the shelf (event A), zero or more events other than \( B \) (\( [\neg B] \)) have occurred, before product X finally has left the store (event C). Thus, the complex event that could be inferred from this pattern is a theft, since product X, which was never paid for\(^1\), has left the store. This is just one example from many other CEP applications, which include algorithmic stock trading, security monitoring such as network intrusion detection, and credit card fraud detection. In the following sections we demonstrate how our REs from the previous chapter can be made applicable for such scenarios.

\(^1\)\( [\neg B] \) ensures that event \( B \) did not occur, i.e. the RFID reader at the cash register never detected product X.
3.2 GRE Specification Language

The classical syntax for regular expressions, i.e. Perl Compatible Regular Expressions (PCRE), is designed for textual data. As we aim at deriving more general purpose regular expression engines, we need some new language constructs for specifying these GREs. In this section an SQL-like syntax is presented that will allow us to do so. The syntax is strongly related to parts of the proposed language standard in [20]. Nevertheless, our language is less comprehensive and therefore a lot simpler.

3.2.1 Input Stream Declaration

Instead of assuming a stream of ASCII characters, we are now trying to handle a stream of any kind of data. Nevertheless, we need some information on how this data is structured. Listing 3.1 shows an example of an input stream declaration. In this stream each tuple represents a trade handled by some derivative exchange, such as for example Eurex.

```sql
CREATE INPUT STREAM Trades (  
  Seqnr int,  -- sequence number  
  Symbol string(4),  -- valor symbol  
  Price int,  -- stock price  
  Volume int  -- trade volume
)
```

Listing 3.1: Stream declaration

The tuples in the input stream have four attributes: a sequence number of type `int` (32-bit), a valor symbol of type `string(4)` (32-bit), and two more attributes again of type `int`. Therefore the entire width of the tuples in this stream is 128 bits.

3.2.2 Predicate Declarations

The patterns we are going to create in the following example will not be defined directly on tuples, but rather on predicates specified over the attributes of these tuples. Listing 3.2 shows how these predicates can be defined in a special DEFINE-clause.

```sql
DEFINE  
  PredA AS (Trades.Symbol = "ABBN")  
  PredB AS (Trades.Price < 50)  
  PredC AS (Trades.Price < 50 AND Trades.Volume > 10000)  
  PredD AS (Trades.Price <= PREV(Trades.Price))
```

Listing 3.2: Predicate declarations

For instance, on line two the predicate `PredA` is defined as the condition whether the attribute `Trades.Symbol` of a tuple is equal to the string “ABBN”. For any tuple with matching attribute this predicate is satisfied. More complex predicates are also possible. On line four the predicate `PredC` is defined over both attributes, `Trades.Price` and `Trades.Volume`, combined by a Boolean operator.

---

3.2 GRE Specification Language

On line five the predicate \( \text{PredD} \) is even made dependent on the attribute of a tuple previously seen — as long as \( \text{Trades.Price} \) does not increase for successive tuples in the stream this predicate is satisfied.

Please note that nothing prevents us from declaring predicates that can be satisfied simultaneously by the same tuple, e.g. \( \text{PredA1 AS (Trades.Symbol = "ABBN")} \) and \( \text{PredA2 AS (Trades.Symbol = "ABBN")} \). We will discuss the consequences of this fact in detail in Subsection 3.4.1.

3.2.3 Pattern Declaration

With the predicates declared it is now necessary to define a pattern over these predicates, which is done in a separate PATTERN-clause. In Listing 3.3 we show how GREs for a stream of ASCII characters can be written in our GRE specification language. This GRE represents the regular expression \( a^* b^+ c^+ \).

```plaintext
CREATE INPUT STREAM CharStream (Character char -- 8-bit character stream)
pattern pat1 (PredA PredB* PredC+)
  define PredA AS (CharStream.Character = 'a')
  define PredB AS (CharStream.Character = 'b')
  define PredC AS (CharStream.Character = 'c')
```

Listing 3.3: Pattern declaration with predicates representing ASCII characters

Since the symbols in the pattern, \( \text{PredA}, \text{PredB}, \text{and PredC} \), are now multi-character identifiers, we need to insert whitespace between the identifiers to make the concatenation explicit.

The predicates, which for better understanding represent characters here, can easily be replaced by predicates with completely different meanings, e.g. with the predicates from Listing 3.2.

Our compiler allows multiple patterns to be compiled in one step. Besides this being convenient, since the predicates and input stream only need to be defined once, it also enables potential space saving optimizations, such as prefix sharing (see Section 2.5). Multiple patterns can simply be listed one after the other, as is done in Listing 3.4.

```plaintext
pattern pat1 (PredA PredB* PredC+)
pattern pat2 (PredB [^PredA PredC])
  define PredA AS (CharStream.Character = 'a')
  define PredB AS (CharStream.Character = 'b')
  ...
```

Listing 3.4: Multiple pattern declaration

3.2.4 Large Variable-Width Tuples

Up to now we have assumed the width of tuples to be fixed and reasonably narrow, e.g. 128-bit wide tuples. In our demo application, in Chapter 5, the entire payload of a TCP packet will be treated as a tuple, which brings some
new complications. First, not all of the tuples have the same width, and second, the tuples can be several thousand bytes wide. The network delivers the TCP packet sequentially in fixed width chunks, e.g., 32-bit chunks. In this case, the assumption that one tuple can be processed every clock cycle is wrong and our circuits need to be prepared for handling tuples arriving sequentially in fragments over several clock cycles. An example GRE specification for handling these kinds of tuples is given in Listing 3.5.

```
CREATE INPUT STREAM TCP (Payload varstring(4) -- 32-bit chunks of variable width tuples)
PATTERN pat1 (INDEX PAGE1* PAGE2+)

INDEX AS (TCP.Payload = /^GET /index\.(htmls?|php)/)
PAGE1 AS (TCP.Payload = /^GET /page1\.(htmls?|php)/)
PAGE2 AS (TCP.Payload = /^GET /page2\.(htmls?|php)/)
```

Listing 3.5: varstring(4)

To deal with variable width fields of tuples a new data type, `varstring`, is introduced. Varstring stands for variable length string. The 4 in parenthesis (see line two of Listing 3.5) indicates that this field needs to be processed sequentially in 32-bit chunks (4×8-bit). When this data type is used a special *end-of-string* (EOS) signal indicates when the end of the field has arrived. In Listing 3.5 we also show the kinds of predicates that can be defined over fields of type `varstring`. For example, a predicate might be satisfied by a regular expression that matches a variable length string. As in Perl, the two slashes are used to denote start and end of a regular expression. In the example we are trying to detect different kinds of HTTP requests by running appropriate REs on the payloads of the TCP packets arriving at a web server.

### 3.3 GRE implementation on the FPGA

In this section we explain how the language constructs, just described, can be translated to circuits on the FPGA. Observe, that the finite state machine implementing the RE is actually independent of the data represented by the symbols, i.e. the automaton is the same whether the symbols represent characters or some other predicate. Therefore we propose to decouple the part that defines the symbols from the REs by introducing a new component which we will call *predicate decoder* in the following.

#### 3.3.1 Decoder for Character-Predicates

We now show how the predicate decoder fits together with our REs. Consider the regular expression $a \cdot b \cdot a \cdot b$ and think of the symbols as characters. The NFA corresponding to this regular expression working directly on character data is depicted in Figure 3.1. The eight wires at the bottom carry the 8-bit ASCII code of the character, e.g. character ‘a’, of which the ASCII code is 97, would be represented as 01100001. Notice that the eight wires need to be routed to every basic building block of the NFA.
At this point we abstract from the ASCII characters. We assume that they are converted to predicates, as is done in Listing 3.3. The predicate decoder has one output signal for every predicate defined. In the case of a character predicate decoder this means that we have an 8-to-256 decoder, which in effect is a demultiplexer, decoding from binary to one-hot format. Figure 3.2 illustrates this decoder.

Note that now the character is only decoded once and then a single wire is routed to the appropriate building blocks. Thus, our REs become more space-efficient with the decoder. More importantly, now the REs are decoupled from the symbol definition and we can simply replace the decoder for ASCII characters by some other predicate decoder.

### 3.3.2 Decoder for Complex Predicate

Besides increasing space efficiency the main benefit of using a decoder is that the REs now are decoupled from the data that they are performing matches on. Whether characters or other kinds of data shall be matched is now a matter of the decoder, e.g. instead having line 97 of the decoder representing the character ‘a’ it can just as well represent the predicate \( \text{Trades.Price < 50 AND Trades.Volume > 10000} \). Evaluation of this predicate is straightforward; a Boolean AND on two conditions on two different attributes of the tuple, can be directly translated to VHDL as such.

### 3.3.3 Implementing the Previous-Expression

Line 5 of Listing 3.2 defines a predicate that is fundamentally different, from the predicates defined above. The predicates in line 2 to 4 can all be evaluated by looking at the current tuple only. To evaluate predicate \( \text{PredD} \), we additionally need information about the previous tuple. More generally, if the the predicate contains the statement \( \text{PREV(Tuple.Attribute, n)} \), where \( n \) refers to the \( n \)th tuple before the current one, then a buffer needs to store at least the last \( n \)
tuples, so this predicate can be evaluated. The Xilinx IP Core Generator Tool provides a wizard for creating first-in first-out (FIFO) memory queues. This core exactly implements the ring buffer behavior that we need to evaluate the PREV-expression. Note that the size of this buffer cannot be unbounded. Therefore the compiler needs to check that PREV-expressions do not refer to tuples which are not in the buffer anymore. Such kinds of semantic checks are typically performed by a compiler in a so called semantic analyzer phase.

3.4 Differences between GREs and REs

The syntax introduced in Section 3.2 allows us to specify GREs that can process more than just textual data. Note that GREs can be specified that are semantically equivalent to the REs from Chapter 2 (see Listing 3.3), but it is also possible to write GREs with subtle semantic differences compared to plain ASCII text REs. This section addresses these differences.

3.4.1 Simultaneously Satisfiable Predicates

It is possible to write predicates, using the syntax from Section 3.2, that may be satisfied simultaneously by a single tuple, e.g. the predicates \( A \) \( \text{AS} \) (\( \text{Trades.Price} > 10 \)) and \( B \) \( \text{AS} \) (\( \text{Trades.Price} < 100 \)) both are satisfied for some tuple from the \text{Trades} stream with attribute, say \( \text{Trades.Price} = 50 \). Notice, this is not possible in the REs dealing only with ASCII characters, since every ASCII code unambiguously defines a single character, e.g. ASCII code \( 0x61 \) always decodes to character ’a’, and no other character but ’a’. To demonstrate the consequences for GRE implementations as NFAs and DFAs respectively, consider the regular expression \((AB|BA)\). In Figure 3.3 the corresponding NFA is given.

![NFA for (AB|BA) with simultaneously satisfiable predicates](image)

Fortunately, the NFA does not need to be modified for handling simultaneously satisfiable predicates; it already covers all possible cases successfully: for example, if the first tuple satisfies predicate \( A \) and \( B \) simultaneously \( q_1 \) and \( q_2 \) are both active concurrently. If the tuple satisfies only predicate \( A \) or \( B \) then only \( q_1 \) or \( q_2 \) are activated accordingly. In any case, the result is correct.

Unfortunately, for the DFA implementation to work properly we need to make some changes to the automaton obtained form the Subset Construction Algorithm [9], since this algorithm does not consider simultaneously satisfiable predicates. The corrected DFA corresponding to \((AB|BA)\) with simultaneously satisfiable predicates is depicted in Figure 3.4.

It is a requirement by definition for a DFA to have not more than one active state at a given point in time. To be able to fulfill this property with simulta-
neously satisfiable predicates, extra states and transitions need to be added. To be precise, a new state (q3) and transition need to be added for the case where both predicates are satisfied at the same time. The reason is that if the first tuple satisfied only predicate A, then the next tuple needs to satisfy predicate B and vice versa. However, if the first tuple satisfied both predicates, A and B, then it suffices for the following tuple to satisfy either predicate, A or B.

As a consequence the DFA implementation can suffer not only from the state explosion described in Section 2.3.6 but also from an additional state explosion due to simultaneously satisfiable predicates. The severity of this state explosion depends on the number of predicates that can be satisfied simultaneously.

For a set of n predicates that can simultaneously be satisfied we need a transition leading to a separate state for any possible subset of these n predicates, i.e. we need a state for every member of the power set of those n predicates, except of course the empty set ∅. Thus, the number of states is equivalent to the sum of binomial coefficients n over k minus one \( ^n \sum_{k=0}^{n-1} \binom{n}{k} = 2^n - 1 \). Again, we have an exponential state explosion.

Note, this state explosion is independent of the regular expression and occurs only due to simultaneously satisfiable predicates. Also, the additional states potentially need to be added to every state of the DFA retrieved from the Subset Construction Algorithm [9]. The DFA in Figure 3.4 is already optimized in that respect, as q1 and q2 only have one outgoing transition.

Figure 3.4: DFA for (AB|BA) with simultaneously satisfiable predicates
Chapter 4

Dispatching Events to Multiple Replicated GREs

When doing pattern matching over event streams, as opposed to pattern matching over textual data, there are a number of new aspects that need to be given attention to. In the previous chapter we have illuminated some of these issues, such as simultaneously satisfiable predicates (see Subsection 3.4.1). In this chapter we focus on an entirely different problem that can arise in event stream pattern matching applications. In an event stream, different sources can produce the same type of event, e.g. source 1 can generate an event X and source 2 can also generate that same event X. To do meaningful pattern matching on such streams it is often necessary to view the events originating from one source isolated from events produced by other sources. We will further motivate why this is important in Section 4.1. In Section 4.2 we propose an event dispatcher with replicated GREs as a solution to this problem. All of the technical details concerning the implementation on the FPGA are covered in Sections 4.3 through 4.7.

4.1 Motivation

Recall the CEP scenario presented in Section 3.1, where the products in a store equipped with RFID readers were generating events and the event pattern $A \cdot [B]^* \cdot C$ was intended to detect theft. What we have omitted in that section is how a pattern matching engine could deal with multiple products. Obviously, if product X is removed from the shelf and next product Y leaves the store, this sequence of events cannot be interpreted as a theft, i.e. for the pattern to make sense, events belonging to one product need to be isolated from events related to other products. However, the events corresponding to different products are interleaved in the event stream and we need a way to separate these events and process them appropriately.

In the example application of Chapter 5 we will have a similar situation to the one above. There we will present a device for analyzing click stream traffic on websites. This device is capable of detecting click patterns on a per-user basis while the users are concurrently clicking on the links of a website.
4.2 Solution Statement: Dispatcher

In applications where multiple sources, e.g. users or products (see previous section), are concurrently generating events which are merged into a single event stream, it helps to view these events as event instances of some event type. We can then write event instances as pairs (ID, Event) consisting of an identifier referring to the source that produced the event instance, and the type of the event instance. Thus, recapitulating the example of the RFID-store from Section 3.1, a stream of event instances could look as the one visualized in Figure 4.1.

\[(\text{ID: ProductX, Event: LeaveShelf})\]
\[(\text{ID: ProductX, Event: CashRegister})\]
\[(\text{ID: ProductY, Event: LeaveStore})\]
\[(\text{ID: ProductX, Event: LeaveShelf})\]

Pattern Matching Engine

Figure 4.1: Stream of event instances

The GREs from Chapter 3 operate on event types and are not concerned with source-IDs. To match patterns on a per-source basis we need to do two things: First, we have to replicate the GREs, such that every source can be processed by a separate GRE. Second, incoming event instances need to be forwarded to the appropriate GRE based on their source-ID. Replication of the GREs is described in Section 4.3, and a hardware implementation of the dispatcher responsible for the forwarding of event instances is explained in Section 4.4.

The number of available sources may outrange the number of GREs that we can put on an FPGA. Therefore, the dispatcher also contains management logic for keeping track of GREs that are not currently in use and allocating those GREs when required. That way, the maximum number of GREs that can be put onto one FPGA is an upper bound for how many different sources can be processed in parallel, but does not strictly limit how many sources can be processed in total over time.

4.3 Replicating GREs

For designs that can be expressed as a repetition of some subsystem, VHDL provides a generate statement which allows us to conveniently define these kinds of regular structures. Thus, replication of GRE entities is straightforward, as shown in Listing 4.1.
4.4 Content-Addressable Memory (CAM)

The number of GREs to be generated is specified by the constant \texttt{NUM\_GRES}, which is set by the compiler. Instead of individual signals we now use vectors to connect the input- and output-ports, e.g. on line four of Listing 4.1 each component of the vector \texttt{result} corresponds to the output signal of one replicated GRE.

4.4 Content-Addressable Memory (CAM)

For the implementation of the dispatcher, we propose using content-addressable memory (CAM), which can be generated with the Xilinx IP Core Generator. With conventional memory, one specifies an address to retrieve the content residing at that address. CAM works the other way around: content is specified for which an address is retrieved. One could compare a CAM with a hash table, however, with look-up times of one clock cycle only. Figure 4.2 depicts an example of a CAM, as we will use it for our dispatcher implementation. The left column contains the keys and the right column the values (addresses). The addresses are encoded in one-hot-encoding\footnote{The Xilinx IP Core Generator also provides the possibility of binary address encoding.} which is useful, as will become clear later. The keys can be any kind of fixed length bit-string, e.g. in Chapter 5 we will use IP addresses concatenated with port numbers as identifiers.

\begin{verbatim}
for i in 0 to NUM\_GRES-1 generate
begin
    GRE\_instance: GRE
    port map (... , output => result(i) , ...);
end generate;
\end{verbatim}

Listing 4.1: Generate statement to replicate GREs

In Figure 4.3 we show how the CAM can be used to dispatch incoming event instances to the appropriate GRE. The box on the left in Figure 4.3 represents an event instance with an identifier part (ID) and an event type part (Event). Note that the event type is routed to all of the GREs, whereas the identifier is forwarded to the dispatcher, which manages the CAM. The \texttt{datavalid} signal of every GRE is connected to a different bit of the one-hot-encoded CAM address. When, for example, \texttt{IDx} is looked-up in the CAM, the returned address will be \texttt{0010}, i.e. three GREs will have their \texttt{datavalid} signal set to logic low, and only the relevant GRE will have its \texttt{datavalid} signal set to logic high and actually process the event instance.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
IDw & 0001 \\
IDx & 0010 \\
IDy & 0100 \\
IDz & 1000 \\
\hline
\end{tabular}
\caption{CAM}
\end{figure}

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node (ID) at (0,0) {ID};
\node (Event) at (1,0) {Event};
\node (Dispatcher) at (2,0) {Dispatcher};
\node (GRE1) at (3,0) {GRE};
\node (GRE2) at (4,0) {GRE};
\node (GRE3) at (5,0) {GRE};
\node (GRE4) at (6,0) {GRE};
\draw[->] (ID) -- (Dispatcher);
\draw[->] (Dispatcher) -- (Gre1);\draw[->] (Dispatcher) -- (Gre2);\draw[->] (Dispatcher) -- (Gre3);\draw[->] (Dispatcher) -- (Gre4);
\end{tikzpicture}
\caption{CAM used as dispatcher}
\end{figure}
4.5 Updating Content-Addressable Memory

In Section 4.2 we have mentioned that our dispatcher should be able to handle more sources than there are GREs on the FPGA, provided that event instances from these sources arrive at different times. Therefore, the entries of the CAM need to be updated on the fly and cannot be preloaded. When an event instance arrives from some source that has no CAM entry, the dispatcher needs to fetch the address of a free GRE and insert it into the CAM together with the source-ID. Then the source-ID can be looked-up once more, this time, resulting in a match. Unfortunately, writing entries to the CAM takes longer than looking-up entries. The latency of the write operation depends on the memory type of the CAM implementation, which is discussed in the following subsection.

4.5.1 CAM Memory Types

The Xilinx IP Core Generator lets you configure the CAM using one of two memory implementations:

- SRL16 based CAM with a 16 clock cycle write operation and a one clock cycle look-up operation.
- Block RAM based CAM with a two clock cycle write operation and a one clock cycle look-up operation.

With a write latency of only two clock cycles we can achieve a higher performance in terms of speed when using the Block RAM implementation. However, when Block RAM becomes scarce on the target device the SRL16 implementation is an interesting alternative. In our compiler, we support both options.

4.5.2 Buffering Event Instances in a FIFO Queue

The write latencies of the CAM raise a problem since we can no longer guarantee processing an event instance in one clock cycle if that event instance triggers a write operation. Instead of operating our circuit at a lower frequency, we propose to buffer incoming event instances. A buffer can be generated with the Xilinx IP Core Generator in the form of a FIFO queue. Figure 4.4 depicts the setup with the dispatcher and the buffer in place.

Rather than forwarding the event instances directly to the dispatcher, they are inserted into the buffer first. By means of an empty signal the buffer lets the dispatcher know when it has new event instances to process (empty = ‘0’). The source-ID of the first event instance in the buffer is forwarded to the dispatcher. We make use of the first-word fall-through (FWFT) option provided by the Xilinx IP Core Generator. This feature enables us to look ahead to the next
4.5 Updating Content-Addressable Memory

An event instance available from the buffer without issuing a read operation, which means that we can check whether an event instance matches some entry in the CAM without having to remove that event instance from the buffer. This is essential, due to the delay caused by the write operation of the CAM. Once the dispatcher is done with an event instance, the dispatcher’s next signal issues the read operation of the FIFO queue, causing the current event instance to be discarded and the next event instance to become available for look-up in the CAM.

4.5.3 Detailed Timing Analysis

In this subsection we give more low-level insight in terms of write and look-up operations of the CAM and the exact interaction between dispatcher and buffer. Figure 4.5 shows the waveforms of relevant signals in the case of a failed look-up with succeeding write operation to the CAM. In this example we are considering the Block RAM implementation of the CAM. We comment the waveforms for every clock cycle below.

Figure 4.5: Mismatch followed by write operation, memory type: Block RAM

**Cycle 1** The buffer is still empty, i.e. the FIFO_EMPTY signal is set to logic high.

**Cycle 2** The buffer now contains an event instance ready to be processed by the dispatcher. Since we make use of the first-word fall-through feature of the FIFO queue, we do not need to issue a read operation to the buffer and the event instance can immediately be looked-up in the CAM.

**Cycle 3** A look-up in the CAM takes one cycle. As the CAM_MATCH signal is still logic low, the look-up must have failed, i.e. we need to write the source-ID of the event instance to the CAM. Therefore the dispatcher issues the write operation of the CAM (CAM_WE = ‘1’).

**Cycle 4** The CAM indicates that it is busy (CAM_BUSY = ‘1’). Depending on the memory type the CAM will be busy for only one clock cycle (Block RAM) or for 15 cycles (SRL16). By synchronizing the dispatcher with the CAM_BUSY signal it is easy to support both memory types.

**Cycle 5** After the source-ID of the event instance has been written to the CAM (CAM_BUSY is set to logic low again), the look-up operation can be repeated. We need to wait an additional clock cycle for the result.

**Cycle 6** This time, the look-up succeeds (CAM_MATCH = ‘1’) and we can request the next event instance from the buffer (FIFO_NEXT = ‘1’).
Notice, the second lock-up of the event instance’s source ID is not strictly necessary, since we already know the result of the lookup — we have to specify the result (address) when we issue a write operation. However, this address is in binary format and we need the result in one-hot-encoding, i.e. among other things we would need to convert the address separately, whereas the CAM does this for free. We decided that spending one extra clock cycle on write operations in favor of a cleaner and less complex design is acceptable. Unfortunately, this is not yet the whole story. With the design presented so far the waveforms for three successive matches in the CAM would look as depicted in Figure 4.6.

As shown in the figure, processing an event instance takes two clock cycles when we request the next event instance from the buffer only once the current one has matched an entry in the CAM. This is so, because we need to wait one clock cycle for the read operation of the FIFO queue to terminate and then an additional clock cycle for the look-up of the next event instance.

To achieve processing times of only one clock cycle per matching event instance, we need to issue the read operation of the FIFO queue before we actually know if the event instance has a match in the CAM. However, the problem with this is that at the time when a potential mismatch could be detected, the buffer has already discarded the relevant event instance and we could no longer write its source-ID to the CAM. To solve this we temporarily save the current event instance for one clock cycle, so that we are still able to write it to the CAM in case of a mismatch. The waveforms for three successive matches in the CAM with this new design are depicted in Figure 4.7.

Now that we have derived all of the relevant timing aspects, we can draw some conclusions about choosing a reasonable depth for the buffer. Event instances with matching CAM entries can be processed in a single clock cycle. Depending on the memory type, processing a mismatching event instance takes $t_w = 2 + 2 = 4$ (Block RAM) or $t_w = 16 + 2 = 18$ (SRL16) clock cycles. The worst-case with respect to buffer depth is when every event instance in the buffer needs to be written to the CAM. Obviously, we cannot support this, since this would require an infinitely deep buffer. Nevertheless, we can make the following simplification: Let $N_{GREs}$ be the number of available GREs on the FPGA. Then after $N_{GREs}$ succeeding mismatches all GREs would most likely be busy,
4.6 Allocating Free GREs

To insert an entry into the CAM we need to specify a key-value pair where the key is the source-ID of the event instance and the value is the binary-encoded address of some free GRE. This implies that we need to be able to identify a free GRE and retrieve its address. As we will further clarify in the next section, each GRE has an is_free signal indicating whether the GRE is currently in use or not. If the GRE is free, it can potentially be allocated to process event instances from the new source-ID. Thus, there can be multiple free GREs and our task is to select one of those GREs — it does not matter which one. We introduce a new entity, which we call priority decoder in the following. This entity is responsible to check if there are any free GREs and to select one of those GREs if there are several. An example is given in Figure 4.8.

![Priority decoder with four GREs](image)

The priority decoder has an input line for the is_free signal of every GRE. The has_free output signal informs whether there is a free GRE available. For \( n \) GREs the priority decoder has \( \log_2(n) \) additional output lines (address) for the binary-encoded address of the GRE that was selected by the priority encoder. Notice that the Xilinx IP Core Generator only lets you specify CAMs of a depth \( n \), such that \( n \) is a power of two, i.e. we need exactly \( \log_2(n) \) lines to be able to encode all \( n \) addresses.

Since it is irrelevant which one of the free GREs is allocated, the priority decoder will always select the GRE with the lowest address, i.e. in Figure 4.8 the priority decoder selects the second GRE with address "01" instead of the third GRE with address "10", which is also free.

The way we code the priority decoder in VHDL matters. In particular, the part describing how the priority decoder finds a free GRE has consequences for the synthesized circuit. In Listing 4.2 we show a naive implementation of a priority decoder.

```vhdl
1 address <= "00" when is_free(0) = '1' else
2   "01" when is_free(1) = '1' else
3   "10" when is_free(2) = '1' else
4   "11" when is_free(3) = '1';
```

Listing 4.2: Naive priority decoder implementation
Though behavioral simulation of this VHDL code produces correct results, the corresponding synthesized circuit has some issues. As the VHDL code specifies, first the is_free signal of GRE 1 is checked. If GRE 1 is not free the is_free signal of GRE 2 is checked and so on. The problem with the synthesized circuit is that the is_free signals of GREs with higher addresses pass through more gates than the ones with lower addresses. The number of gates the is_free signal needs to propagate grows linearly with the address of the corresponding GRE, i.e. if GRE 16 is the only free GRE then its is_free signal needs to be processed by 16 gates. Every gate induces some delay for processing. If the longest path of a signal becomes too long, than the priority decoder will no longer be able to compute a result within one clock cycle.

A better way to build the priority decoder is displayed in Listing 4.3. This VHDL code produces a circuit in which all of the signals pass through a tree-structure.

```
for lvl in 0 to LOG2_GRES-1 generate
  for i in 2**lvl to 2**(lvl+1)-1 generate
    local_hit(2*NUM_GRES-i-1) <=
      local_hit(2*NUM_GRES-(2*i)-1) or
      local_hit(2*NUM_GRES-(2*i+1)-1);
    min_addr(2*NUM_GRES-i-1) <=
      min_addr(2*NUM_GRES-(2*i+1)-1) when
      local_hit(2*NUM_GRES-(2*i+1)-1) = '1' else
      min_addr(2*NUM_GRES-(2*i)-1);
  end generate;
end generate;
address <= min_addr(2*NUM_GRES-2);
has_free <= local_hit(2*NUM_GRES-2);
```

Listing 4.3: Logarithmic priority decoder with log2(NUM_GRES) levels

At the leaves of this tree, we have the is_free signal and the address of every GRE. On each level of the tree, a child node is compared with its neighbor. If either of the two child nodes has a free GRE in its subtree, local_hit is set to '1' for the current node. min_addr stores the address of the free GRE. In case both child nodes have a free GRE in their subtree, the lower address of the two GREs is stored. This construction results in a logarithmic increase of the longest path in relation to the number of GREs, as opposed to a linear increase. Thus, this implementation scales a lot better than the one from Listing 4.2.

4.7 Determining when a GRE is Free

As stated above, every GRE uses its is_free output port to tell the outside world whether it is currently in use or not. In this section we will have a closer look at how exactly each GRE can determine if it is free. We need to distinguish between a GRE that allows overlapping matches and one that does not — an option that can be selected in our compiler. For the GRE that allows overlapping matches, we can say that it is free only when no states are active other than the initial ones, which are always active. A GRE that does not allow overlapping matches will be reset immediately after a match has occurred, i.e. after a match, only the initial states will be active and the GRE will become free.
If we leave it at this, we have no guarantee that a GRE, once allocated, will ever be released again, and it is very likely that we would soon run out of GREs to process event instances from new sources. Therefore, we have equipped our GREs with a lease timer. When a free GRE is allocated and associated with a new source-ID, the timer is started. If the GRE is still allocated once the timer expires, the is_free signal of the GRE is automatically set to '1'. Clearly, with this mechanism in place, it cannot be avoided that some long running pattern matches may go undetected, but it is a compromise we have to make in order to avoid the system from becoming congested.

The implementation of the lease timer is very simple. It is, in fact, nothing but a counter that is decreased by one every clock cycle. Our FPGA (Xilinx XUPV5-LX110T) is running at 100 MHz clock frequency, i.e. the clock period is 10ns. Therefore, decrementing the counter 100,000,000 times is equivalent to one second. The lease time is a parameter that needs to be chosen carefully according to the requirements of the target application. If the lease time is chosen too short, a lot of long running pattern matches will be missed. If, however, the lease time is chosen too long, the system is vulnerable to potential Denial of Service attacks. Since selecting a reasonable lease time is very application dependent, we allow the user to specify this parameter via the interface to the compiler.
Chapter 5

Click Stream Pattern Matching

In the previous chapters the explanations about the crucial parts of our work have been rather theoretical. In this chapter we assemble the pieces introduced until now and show how our GRE compiler can be used to build an actual application. In this example application we will do real-time analysis of click stream traffic and detect click patterns on a per-user basis. The FPGA will be connected to a network via its network interface, so that the circuits on the board can operate directly on the Ethernet frames at network speed. In Section 5.1 we will further explain the setup and discuss why this is an interesting application. Then, in Section 5.2 we illustrate a couple of useful click stream patterns together with the source code to generate appropriate pattern matching engines. To ease source code compilation, we have developed a graphical wizard that guides the user through the compilation process. This wizard, which we call Regular Expression Engine Generator, is presented in Section 5.3. We explain the integration of the compiler’s output files into ISE and EDK projects in Section 5.4, and finally we discuss some open issues with respect to future work in Section 5.5.

5.1 Click Stream Analysis

A main goal in web usage mining techniques is to discover a user’s browsing behavior. A click stream is then the sequence of page views that are accessed by the user. We will use our Regular Expression Engine Generator to generate circuits that can detect specific click patterns in these streams.

To experiment with this application, we have set up a small network, which is illustrated in Figure 5.1. A web server and a few clients communicate over a managed switch. We have configured the switch to mirror the port of the web server. The FPGA, configured with the pattern matching engine, is connected to the mirrored port. This way, the FPGA can eavesdrop on the entire traffic between web server and clients. On the FPGA the Ethernet frames are decoded and potential TCP payloads are processed by the circuits of the pattern match-
Click Stream Pattern Matching

The end system is connected to the FPGA via serial cable. When a pattern is matched, one possibility is to let the pattern matching engine interrupt the Microblaze processor running on the FPGA. An interrupt handler could then, for example, alert the end system.

![Application setup diagram](image)

Figure 5.1: Application setup

For this example application to work properly, we make some simplifying assumptions, e.g. that users can be identified by their IP addresses, or that browsers do not make use of their web caches, since caching would prevent a number of clicks from triggering HTTP requests to the web server. Hence, this application is perhaps less realistic than the CEP applications mentioned earlier, e.g. in Section 3.1. Nevertheless, for the purpose of demonstrating the potential of our GREs, this application is satisfactory.

We have deliberately chosen this application as an example because it combines many of the concepts we have previously presented. Just to name a few, we will use regular expressions inside the predicate declarations to detect HTTP requests, we will need to process four characters per clock cycle, since the network delivers 32 bits of data every cycle, and we will make extensive use of prefix sharing. In addition, it is very easy to generate real test data for this application.

5.2 Click Stream Pattern Examples

In this section we want to give some idea of the kinds of click stream patterns that can be detected by the circuits generated with our compiler. As an example we will consider a multilevel web form such as are typically seen in online flight reservation systems. A schematic representation of the web form is depicted in Figure 5.2.

This form consists of three subforms, which are all on separate web pages. Depending on the options selected by the user at the first level (F1), the second level (F2) will be skipped. After the user has completed the third level (F3), a confirmation-page (C) is displayed. From the confirmation-page the user can go back to any of the previous levels and modify his data. Once the user clicks the confirm-button on the confirmation-page, one of three different thankyou-pages (T1–T3) will be displayed.

---

2http://www.xilinx.com/support/documentation/sw Manuals/mb ref_guide.pdf
5.2 Click Stream Pattern Examples

5.2.1 Stream Declaration

Before we can define patterns, we have to specify a stream declaration, see Listing 5.1. We name this stream TCP. It has only one attribute of type varstring(4) which is called Payload. As the name implies, this attribute represents the payload of a TCP packet. The network interface on the FPGA delivers the Ethernet frames in 32 bit chunks. Therefore, we use a four character wide varstring (4 × 8 bits). Notice that variable length TCP packets are perfectly supported by the varstring data type.

```
CREATE INPUT STREAM TCP ( Payload varstring(4) )
```

Listing 5.1: Stream declaration for multilevel web form

5.2.2 Predicate Declarations

The predicates over which our patterns will be defined should represent different kinds of user clicks. The predicate declarations are given in Listing 5.2.

```
F1 AS (TCP.Payload = /^(GET|POST) /form1.html/)  
F2 AS (TCP.Payload = /^(GET|POST) /form2.html/)  
F3 AS (TCP.Payload = /^(GET|POST) /form3.html/)  
C AS (TCP.Payload = /^(GET|POST) /confirm[1-3].html/)  
T AS (TCP.Payload = /^(GET|POST) /thankyou[1-3].html/)  
O AS (TCP.Payload = /^(GET|POST) /[^.]*.html.*HTTP/1.[01]/)
```

Listing 5.2: Predicate declarations for multilevel web form

We are interested in TCP payloads that start either with “GET” or “POST”, according to the HTTP protocol. For example, clicking the submit-button on one of the subforms would trigger a POST request, whereas the links on the confirmation-page, taking the user back to one of the subforms, would fire GET requests.

Since it will be irrelevant in our patterns which one of the three thankyou-pages is displayed to the user, we can declare a single predicate representing all three thankyou-pages. On line five in Listing 5.2, we do exactly this with the help of a character class declaration ([1-3]).

The last predicate (O), on line six in Listing 5.2, is supposed to catch any other GET or POST request issued by the user which is not covered by the five previously declared predicates. Note that we could have also written this predicate as follows: O AS (TCP.Payload = /^(GET|POST) /[^.]*.html/). However, it would then always be satisfied simultaneously with one of the other
predicates. With attributes of type `varstring` the predicate decoder stops on the first satisfied predicate, i.e. declaring predicate `O` as is done on line six in Listing 5.2 ensures that this predicate can only be satisfied, when none of the other five predicates have previously been satisfied.

### 5.2.3 Pattern Declarations

With the predicates declared, it is now time to define some patterns over these predicates. In Listing 5.3 three patterns of interest are defined. We comment the three patterns below.

1. **pattern1** `(F1 F2? F3 C T)`
2. **pattern2** `(F1 F2? F3 C ([F1-F3]+ C)+ T)`
3. **pattern3** `([F1-C]+ 0)`

Listing 5.3: Predicate declarations for multilevel web form

- **pattern1** On the confirmation-page the user clicks the confirm-button without going back to make any changes to his data. Note, subform 2 is optional in the pattern `(F2?)`.
- **pattern2** This time, the user goes back at least one time to one of the subforms, before confirming. Notice how we use the range operator in the symbol class `([F1-F3])`. Which predicates are included in the range is dependent on the order in which the predicates are declared, i.e. in this case `[F1-F3]` is equivalent to `[F1 F2 F3]`.
- **pattern3** The last pattern detects users that start filling out the web form but then leave and request some other page on this website.

The three example patterns above are slightly simplified. There are a number of special cases which we have ignored for readability, e.g. the case where the user reloads the confirmation-page. Also, **pattern3** only partially detects what we actually would like it to recognize. To be precise, this pattern does not detect cases where the user closes the browser or where the user clicks away to some other website. In fact, detecting the absence of some symbol or event, as in this case, is not something that regular expressions are very good at.

### 5.3 Regular Expression Engine Generator

As noted in the introduction to this chapter, we have built a graphical wizard that guides the user through the compilation process. We refer to this graphical user interface as *Regular Expression Engine Generator* or simply *wizard* in what follows. The wizard consists of four different dialog pages. On the first page the user can specify a location where the generated files should be saved. On the second page the source code that describes the pattern matching engine can be submitted to the compiler. For example, the source code defining the three patterns discussed in the previous section could be entered as shown in Figure 5.3.

3In HTTP version 1.0 and 1.1 GET and POST requests are followed by the version of the HTTP protocol, i.e. “HTTP/1.0”. Here, we do not support HTTP 0.9, where this is not the case.
The third dialog page is displayed in Figure 5.4. First of all, an implementation for the GREs can be selected, i.e. we can generate GREs as DFAs or as NFAs (see Chapter 2 for further information). We support both variants for experimental purposes. Note that whatever implementation the user selects, it does not affect the REs that are part of the predicate decoder. We implement those REs always as NFAs. This is because we have developed more enhancements for the NFA implementation, such as multiple symbol processing in one clock cycle, or prefix sharing. There are three other options on the third dialog page. The option Prefix sharing is applied to all generated NFAs, i.e. the REs of the predicate decoder as well as the GREs, when implemented as NFAs, can be optimized by sharing prefixes. The only obvious reason to deactivate this option is to conduct experiments. The option Overlapping matches controls whether a GRE will be reset, after it has detected a match. Deactivating this option may result in missing some overlapping matches, but can increase the availability of free GREs. Finally, with the Debugging signals option a number of internal signals that might be useful for debugging are routed to output ports of the top-level entity, e.g. the signal debug_bufferfull indicates if the event instance buffer is full.

On the fourth dialog page, depicted in Figure 5.5, the dispatcher can be configured. Number of replicas determines the depth of the CAM and how many times the GREs will be replicated. Source-ID width sets the width of the CAM and should resemble the bit length of the source-IDs. In our case, Source-ID width is set to 32 bits, since our source-IDs are IP addresses. Buffer depth
defines the depth of the event instance buffer, i.e. the FIFO queue. _Lease timeout_ specifies the maximum time that a GRE may be occupied by some source. However, we must remember that if _Overlapping matches_ from dialog page three is deactivated, the GRE will also be free when a match occurs. Lastly, a memory type (see Subsection 4.5.1) for the CAM implementation can be selected.
5.4 Integration with ISE and EDK

When the user clicks the finish button of the Regular Expression Engine Generator, several VHDL files and two XCO files are generated in the directory that was specified by the user. In this section we will explain how to proceed with these files in order to get the pattern matching engine running on the FPGA.

5.4.1 Integration with ISE Project

The Integrated Software Environment (ISE) is the Xilinx design software suite consisting of a plethora of useful tools. ISE is especially important for two reasons: on the one hand, we can simulate the generated pattern matching engine inside ISE and, on the other hand, ISE can generate the IP cores for which our compiler has generated XCO files. An XCO file is a log file that records customization parameters used to create an IP core.

Integrating the generated files into an ISE project is straightforward and can be achieved by following the three steps below:

1. Use the Project Navigator to create a new ISE project. Create the project in the same location where the output files of the compiler were saved.

2. Once the project has been created, right-click on the device in the Sources window and select Add Source. Add all VHDL and XCO files (see below) that were generated by the Regular Expression Engine Generator.

3. Change to Behavioral Simulation, add a new VHDL Test Bench, and run the simulation. When the simulation is run for the first time, ISE will prompt twice to regenerate the buffer IP core and the cam IP core respectively. Say yes both times.

Running the simulation is a convenient way to generate the IP cores. However, we could of course, also generate the cores by double-clicking Regenerate All Cores under Design Utilities in the Processes window. Note that generating the IP cores will also produce the netlists which we will require in order to run the pattern matching engine on the FPGA, as we will explain in Subsection 5.4.2.

Below, we list all of the files that are generated by the Regular Expression Engine Generator with a brief description.

- **pattern_matching_engine.vhd** This file contains the top-level entity representing the entire pattern matching engine. All of the other files below implement internal components of this entity.

- **buffer.impl.xco** This file is the log file containing customization parameters for the FIFO queue IP core representing the event instance buffer.

- **dispatcher.vhd** VHDL implementation of the dispatcher, responsible for managing the CAM and the event instance buffer.

- **cam.impl.xco** This file is the log file containing customization parameters for the content-addressable memory (CAM) IP core.

- **predicate_decoder.vhd** VHDL implementation of the predicate decoder. This entity may comprise instances of regular expression engines if regular expressions were used to define the predicates.
character_decoder.vhd  8-to-256 ASCII character decoder used in the predicate decoder when predicates are defined with regular expression.

gre.vhd  VHDL implementation of the general purpose regular expression engine (GRE). Depending on the compiler options, implemented as NFA or DFA.

nfa_state.vhd  VHDL implementation of a basic building block representing a state of the NFA.

priority_decoder.vhd  VHDL implementation of the logarithmic priority decoder for determining free GREs.

source_code.txt  The source code entered into the Regular Expression Engine Generator is backed up in this text file. This file does not need to be integrated into an ISE or EDK project.

ast.txt  This text file contains an ASCII text representation of the abstract syntax tree generated by the parser. This file does not need to be integrated into an ISE or EDK project.

5.4.2 Integration with EDK Project

To actually configure the FPGA with our pattern matching engine, we use the Embedded Development Kit (EDK), which is yet another software suite by Xilinx. EDK allows us to integrate the pattern matching engine as a peripheral, and have it communicate with the processor on the FPGA over the Processor Local Bus (PLB\[^4\]. To create peripherals EDK provides a special wizard\[^5\]. In this wizard we can submit the VHDL files generated by our Regular Expression Engine Generator. For the IP cores, we need to add the netlists generated by ISE (see Subsection 5.4.1), i.e. buffer_impl.ngc and cam_impl.ngc.

In our example application, we use the local processor on the FPGA to communicate with the end system. As mentioned in Section 5.1 one possibility is that the pattern matching engine triggers an interrupt when a pattern has been matched. On the processor we are then running a C program with a special interrupt handler that can meaningfully process the interrupt and communicate back to the end system over the serial port, e.g. write a message to a console running on the end system as displayed in Figure 5.6.

Alternatively, we could have a C program periodically poll some result register on the FPGA. This makes sense, for example, when we are not interested in every individual match, but rather want to know how many times some pattern matches. In this case, a particular circuit on the FPGA would take care of the counting and directly update the result register while the C program would readout this register, say, once every second.

Besides communicating with the local processor our circuits should also be able to directly access the network interface of the FPGA and decode incoming Ethernet frames. How this can be achieved is the topic of the next subsection.

\[^4\] In our case, this bus is actually called mb_plb, since our board has a MicroBlaze processor.

\[^5\] Tutorials on how to create peripherals can be found at http://www.fpgadeveloper.com.
5.4 Integration with ISE and EDK

5.4.3 Network Interface

To access the network interface, we need to use the Tri-Mode Ethernet Media Access Controller (TEMAC). In our case this is the XPS_LL_TEMAC core which supports the Xilinx hard silicon Ethernet MAC on our FPGA (XUPV5-LX110T). This core incorporates the applicable features of the physical layer and the Media Access Control (MAC) sublayer of the data link layer based on IEEE Std. 802.3-2002.

By integrating the XPS_LL_TEMAC core into our system, we can directly access Ethernet frames coming in from the network. Nevertheless, we still need to decode the frames in order to get hold of relevant TCP payloads corresponding to HTTP requests to our web server, e.g. we need to check that the Ethernet frame contains an IP packet with the destination address of our server and that the IP packet contains a TCP packet with the destination port set to 80, the standard port for the HTTP protocol. As an example, in Listing 5.4 the VHDL code for extracting the TCP header size from an Ethernet frame is listed.

Listing 5.4: VHDL for decoding TCP header size

```vhdl
if tcp_packet and (rx_data_length = C_TCP_HEADER_SIZE) then
    tcp_header_size := rx_llink_din(15 downto 12);
end if;
```

When the Boolean variable `tcp_packet` is true, we have decoded the Ethernet frame so far that we can be sure that this frame contains a TCP packet. Note, our circuit retrieves a 32 bit chunk of the Ethernet frame every clock cycle. The signal `rx_data_length` indicates the position of the current 32 bit chunk within the frame. The constant `C_TCP_HEADER_SIZE` refers to the offset of the 32 bit chunk that contains the TCP header size from the start of the Ethernet frame. Decoding the header size is important, since it determines at which offset the payload of the TCP packet will start.

http://www.xilinx.com/products/ipcenter/xps_ll_temac.htm
5.5 Open Issues and Future Work

In Subsection 5.4.3 we discussed how we had to manually write explicit VHDL code responsible for the decoding of the Ethernet frames. Only when various conditions were fulfilled, was the payload of a TCP packet processed by the pattern matching engine. A question that may be asked is whether the decoding of the Ethernet frames could not be handled by the predicate decoder instead, since an Ethernet frame could be viewed as a very large tuple. Figure 5.7 depicts this interpretation of an Ethernet frame. The gray areas indicate regions of the Ethernet frame that are irrelevant for the decoding.

![Figure 5.7: Ethernet frame interpreted as very large tuple](image)

Let us assume that we can write a stream declaration, such that the relevant attributes can properly be identified in the Ethernet frame. Then we could use these attributes in our predicate declarations, as exemplified in Listing 5.5.

**Listing 5.5: Decoding of the Ethernet frame moved to predicate decoder**

<table>
<thead>
<tr>
<th>ethertype</th>
<th>protocol</th>
<th>dstip</th>
<th>dstport</th>
<th>http</th>
</tr>
</thead>
</table>

|   |   |   |   |
|---|---|---|

F1 AS ( frame.ethertype = X"0800" -- is IPv4 packet
\[ 2 \] AND frame.protocol = X"6" -- is TCP packet
\[ 3 \] AND frame.dstip = X"COA80001" -- is 192.168.0.1
\[ 4 \] AND frame.dstport = X"50" -- is port 80
\[ 5 \] AND frame.http = /^ GET .../ -- is form1.html )

Defining predicates this way, we move the decoding of the Ethernet frames from a separate, “hand-crafted” circuit to the predicate decoder. However, this approach still has some issues that we point out in the following paragraphs.

As we have explained earlier, the Ethernet frames come in 32 bit chunks. When the attributes inside a predicate declaration are spread over several such chunks, this implies that we need to validate those predicates incrementally. Therefore, our compiler would need to generate some sort of state machine to keep track of satisfied conditions over multiple clock cycles. Notice, when evaluating an attribute of type `varstring` against a regular expression, it is the state machine of the regular expression engine that “remembers” previous 32 bit chunks.

A different problem is that Ethernet frames are not of a fixed size. The `varstring` data type allows us to deal with a single variable length attribute for which we know its starting position. Having multiple variable length attributes in one stream declaration means that the attributes are not necessarily at fixed positions. For example, when the TCP header contains extra, optional fields this moves the payload of the TCP packet further back within the Ethernet frame. As a result, we would need a more sophisticated way of expressing stream declarations than what we have proposed in Chapter 3.
Chapter 6

Measurement

We have conducted a number of experiments on the circuits generated with our compiler. Besides measuring the FPGA resource consumption of these circuits, we have also tested the scalability of our pattern matching engines with respect to timing constraints. In this chapter we present the results of our evaluation. In Section 6.1 we first give an overview of the measurement units relevant in our experiments. Then, in Section 6.2 we compare DFAs that suffer from state explosion with equivalent NFAs. In Section 6.3 we measure the required FPGA resources for the multilevel web form patterns of Chapter 5 considering different degrees of replication and in Section 6.4 we incorporate timing constraints.

6.1 Measurement Units

In the experiments of the following sections we will be concerned with the measurement units explained here. These are flip-flops (FF), look-up tables (LUTs), slices, and Block RAM. All of our measurements have been performed on the Xilinx University Program XUPV5-LX110T Development System. A flip-flop is an electronic circuit that serves as a one bit memory. Look-up tables are used to encode Boolean logic functions. An n-bit look-up table can encode any n-input Boolean function. Our FPGA has 6-bit look-up tables. The configurable logic blocks (CLB) inside an FPGA are composed of so called slices. A slice on our FPGA consists of four look-up tables, four flip-flops, and a carry chain. Finally, our FPGA ships with 148 units of Block RAM with each block providing 36 Kbit of memory.

6.2 State Explosion & Resource Consumption

In Chapter 2 we showed that the DFA corresponding to the regular expressions $(0|1) \ast \cdot 1 \cdot (0|1) \{i\}$, for $i \in \mathbb{N}$, suffers from state explosion, i.e. the number of states grows exponentially with respect to the $i$ in the constraint repetition. In this section we present the results of an experiment where we measured the resources required to implement the above regular expression. We compared

\footnotesize{http://www.xilinx.com/uni/xupv5-lx110t.htm
http://www.xilinx.com/support/documentation/user_guides/ug190.pdf}
the following three implementations: NFA, DFA with one-hot-encoded states (One-hot DFA), and DFA with binary-encoded states (Binary DFA). The goal of this experiment was to see if the state explosion of DFAs would lead to a proportional increase in consumption of FPGA resources.

![Figure 6.1: State explosion: flip-flops (FFs)](image_url1)

The graph in Figure 6.1 depicts the number of flip-flops used to implement \((0|1)^*1(0|1)^i\), for \(i \in \{1,2,\ldots,10\}\). The amount of flip-flops used by the NFA and the Binary DFA are negligible. However, the number of flip-flops required by the One-hot DFA explodes with increasing \(i\) of the constraint repetition. This is not surprising since in the DFA only one active state needs to be saved and \(n\) bits suffice to binary-encode one of \(2^n\) different states. With one-hot-encoding, however, saving one of \(2^n\) states requires \(2^n\) bits.

![Figure 6.2: State explosion: look-up tables (LUTs)](image_url2)
The graph in Figure 6.2 shows the number of look-up tables used to implement $(0|1)^*1(0|1)^i$, for $i \in \{1, 2, \ldots, 10\}$. Again, the few look-up tables used by the NFA are insignificant. In contrast, the number of look-up tables grows exponentially relative to $i$ for both the One-hot DFA and the Binary DFA. Increasing the number of states causes the number of state transitions to increase as well. All of these transitions need to be implemented in logic, i.e., with look-up tables. Therefore, the binary-encoded states of the Binary DFA have little impact on the consumption of look-up tables.

Finally, the graph in Figure 6.3 displays the number of occupied slices required to implement $(0|1)^*1(0|1)^i$, for $i \in \{1, 2, \ldots, 10\}$. The number of occupied slices is correlated with the number of used registers and the number of used look-up tables. The Binary DFA occupies almost as many slices as the One-Hot DFA since it uses look-up tables similarly excessive.

Our measurements show that state explosion in DFAs causes a proportional increase in consumption of FPGA resources. Implementing the DFA as a binary-encoded state machine does reduce the number of required registers but cannot significantly affect the number of required look-up tables.

### 6.3 Multilevel Web Form Example

In the previous section we looked at FPGA resources required for a single regular expression. In this section we show the resources consumed by an entire pattern matching engine, including an event dispatcher, an event instance buffer, a predicate decoder, and replicated GREs. For this experiment we used the pattern matching engine of the example application described in Chapter 5. We compiled the source code listed in Listing 6.1 with the following compiler options: `implementation = NFA, prefix sharing = true, source-ID width = 32, buffer depth = 32, CAM memory type = Block RAM, and number of replicas ∈ \{16,32,64,128\}.`
CREATE INPUT STREAM TCP {
Payload varstring(4)
}
PATTERN pat1 (F1 F2? F3 C T)
PATTERN pat2 (F1 F2? F3 C (([F1-F3]+ C)+ T)
PATTERN pat3 (([F1-C]+ O)
DEFINE F1 AS (TCP.Payload = /^(GET|POST)\/[\w]+/i)
F2 AS (TCP.Payload = /^(GET|POST)\/[\w]+/i)
F3 AS (TCP.Payload = /^(GET|POST)\/[\w]+/i)
C AS (TCP.Payload = /^(GET|POST)\/[\w]+/i)
T AS (TCP.Payload = /^(GET|POST)\/[\w]+/i)
O AS (TCP.Payload = /^(GET|POST)\/[\w]+/i)

Listing 6.1: Pattern matching engine for multilevel web form, see Chapter 5

In Table 6.3 the results of our measurements are displayed. We measured the number of flip-flops, look-up tables, occupied slices and blocks of BRAM used for 16, 32, 64, and 128 replicated GREs.

<table>
<thead>
<tr>
<th>Replicas</th>
<th>Flip-flops</th>
<th>Look-up tables</th>
<th>Slices</th>
<th>Block RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1,113 (1%)</td>
<td>1,731 (2%)</td>
<td>726 (4%)</td>
<td>5 (3%)</td>
</tr>
<tr>
<td>32</td>
<td>2,010 (2%)</td>
<td>3,081 (4%)</td>
<td>1,163 (6%)</td>
<td>5 (3%)</td>
</tr>
<tr>
<td>64</td>
<td>3,803 (5%)</td>
<td>5,838 (8%)</td>
<td>1,969 (11%)</td>
<td>9 (6%)</td>
</tr>
<tr>
<td>128</td>
<td>7,388 (10%)</td>
<td>11,214 (16%)</td>
<td>4,203 (24%)</td>
<td>17 (11%)</td>
</tr>
</tbody>
</table>

Table 6.1: Multilevel web form pattern

Not surprisingly, resource consumption increases linearly relative to the number of replicas. With 128 replicas this pattern matching engine occupies almost one quarter of the slices available on our FPGA. However, chip space is not the only limiting factor regarding the scalability of our approach to handle event instances of multiple event sources. Timing constraint violations, discussed in the next section, are another issue.

6.4 Timing Constraint Violations

Replicating GREs for the purpose of processing multiple concurrent event instances from different event sources makes sense only to a certain degree, as the measurements of the previous section have already indicated. Next to the problem of extensive resource consumption there is the problem of fulfilling timing constraints. When we attempted to configure our FPGA with the pattern matching engine from Listing 6.1 we could only scale-up to 64 replicas of the GREs without violating timing constraints. One reason for this is that the data paths in the priority decoder become too long once the tree that determines the next free GRE exceeds six levels \(\log_2(64) = 6\). When the data paths are too long, the signals cannot propagate through the entire tree within on clock cycle anymore.
Chapter 7

Conclusion

7.1 Regular Expression Matching in FPGAs

In the first part of this thesis, we discussed techniques to implement regular expression engines on the FPGA. The two main approaches, namely implementations as deterministic finite automaton (DFA) and as non-deterministic finite automaton (NFA), are both supported by the regular expression compiler that we have built. In either approach the inherent parallelism of FPGAs can be exploited in order to evaluate multiple regular expressions concurrently. Nevertheless, in the NFA implementation even multiple active states of the automaton can be processed in parallel. This may lead to substantially lower area cost for those regular expressions where the corresponding DFAs suffer from state explosion, as we showed in Section 6.2.

The throughput with both DFA and NFA implementations is one symbol per clock cycle, i.e. 100,000,000 symbols per second on an FPGA with a 100 MHz clock. However, area-throughput trade-offs are possible — for NFAs we presented an elegant method, in Section 2.7, to increase throughput by replicating the combinatorial part of the NFAs.

7.2 Pattern Matching in Streaming Systems

The second part of this thesis was devoted to applying our regular expression engines to streaming systems. We showed that by decoupling predicate (symbol) decoding from pattern matching the problem of supporting streams of items with arbitrary data types was reduced to the problem of generating the appropriate predicate decoder, i.e. the regular expression engines could be left unmodified.

A further analysis of pattern matching in Complex Event Processing (CEP) applications revealed that regular expression matching by itself would not suffice to fulfill the needs of many applications. In those applications, typically a number of sources would generate events that were all merged into the same event stream. The challenge then was to enable concurrent regular expression matching on a per-source basis in a mixed stream of events from different sources. As a solution to this problem, we proposed an event dispatcher that distributed events to replicated general purpose regular expression engines (GREs). Our experiments showed that we could process up to 64 concurrent sources without violating timing constraints.
Finally, we integrated the circuits generated by our compiler into an example application where network traffic was analyzed to detect click stream patterns of web users. The purpose of this application was to demonstrate the generality of our regular expression engines as well as their ability to process a network stream at wire speed.

7.3 Future Work

As already mentioned in Chapter 5, it could be further experimented with very large, variable length tuples. It would also make sense to adapt the GRE specification language, introduced in Chapter 3, to better support these kinds of tuples in a universally applicable manner. For further details, see Section 5.5.

In the experiments discussed in Chapter 6 we were not able to scale-up beyond 64 replicated GREs without violating timing constraints. In Section 6.4 we suggested that one cause of this problem could be the priority decoder. Once the tree for determining the next free GRE exceeds six levels ($\log_2(64) = 6$), the data paths in the tree become too long for the signals to propagate through the entire tree within one clock cycle. However, because the priority decoder only needs to be accessed when new items are inserted into the content-addressable memory (CAM), a pipelined version of the priority decoder, e.g. requiring two clock cycles to look up a free GRE, would probably have a minor negative effect on overall throughput.
Appendix A

Regular Expression Syntax

The table below lists some of the most common regular expression operators, e.g. as used to write *Perl Compatible Regular Expressions (PCRE)*.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Concatenation</td>
<td>First matches the item on the left, then matches the item on the right.</td>
</tr>
<tr>
<td></td>
<td>Union</td>
<td>Either matches the item on the left, or the item on the right.</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Kleene closure          Matches the previous item zero or more times.</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>Kleene plus            Matches the previous item once or more.</td>
</tr>
<tr>
<td></td>
<td>?</td>
<td>Optional               Makes the preceding item optional.</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Caret                   Matches at the start of the string the regular expression is applied to.</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>Dollar                  Matches at the end of the string the regular expression is applied to.</td>
</tr>
<tr>
<td></td>
<td>a,b,c</td>
<td>Character               Matches a single specific character.</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wildcard</td>
<td>Matches any single character.</td>
</tr>
<tr>
<td>[</td>
<td>Character class</td>
<td>Matches a single character out of all the possibilities offered by the character class.</td>
</tr>
<tr>
<td>]</td>
<td>Negated character class</td>
<td>Match a single character not listed in the character class.</td>
</tr>
<tr>
<td>[</td>
<td>Range</td>
<td>Matches a single character defined by the range in the character class.</td>
</tr>
<tr>
<td>(</td>
<td>Grouping</td>
<td>Groups a part of a regular expression together.</td>
</tr>
<tr>
<td>{n}</td>
<td>Exactly</td>
<td>Matches the previous item exactly n times.</td>
</tr>
<tr>
<td>{m,}</td>
<td>AtLeast</td>
<td>Matches the previous item at least m times.</td>
</tr>
<tr>
<td>{n}</td>
<td>AtMost</td>
<td>Matches the previous item at most n times.</td>
</tr>
<tr>
<td>{m,n}</td>
<td>Between</td>
<td>Matches the previous item between m and n times.</td>
</tr>
<tr>
<td>\</td>
<td>Escape</td>
<td>Escapes an operator and uses the next character literally.</td>
</tr>
<tr>
<td>\1</td>
<td>Backreference</td>
<td>Allows reusing part of a regular expression match.</td>
</tr>
</tbody>
</table>
Appendix B

Glossary of Abbreviations

- $\epsilon$-transitions: a transition that does not consume an input symbol
- AST: abstract syntax tree
- BRAM: block RAM
- CAM: content-addressable memory
- CEP: complex event processing
- CLB: configurable logic block
- CPU: central processing unit
- DFA: deterministic finite automaton
- EDK: embedded development kit (Xilinx software)
- FF: flip-flop
- FIFO: first in first out
- FPGA: field-programmable gate array
- FSM: finite state machine
- FWFT: first-word fall-through
- GRE: general purpose regular expression engine
- grep: global/regular expression/print
- ISE: integrated software environment (Xilinx software)
- LALR parser: lookahead-LR-parser
- LR parser: parser for context-free grammars
- LUT: look-up table
- MAC: media access control
- NFA: non-deterministic finite automaton
- PCRE: perl compatible regular expressions
- PLB: processor local bus
- RE: regular expression engine
- RFID: radio frequency identification
- TEMAC: tri-mode Ethernet media access controller
- VHDL: VHSIC hardware description language
- VHSIC: very high speed integrated circuit
Bibliography


[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


