ABSTRACT

KRISHNAMOORTHY, REMYA. Hardware Implementation of an XML Parser. (Under the direction of Dr.Yannis Viniotis).

EXtensible Markup Language (XML) is fast emerging as the most preferred language for data exchanges between heterogeneous web servers and databases. This in turn has propelled the deployment and use of “XML-aware” networking equipment like routers, switches and appliances. With network speeds touching tens of Gbps, the heavily CPU-intensive, software-based XML processing methods fail to deliver the required throughput, posing a severe bottleneck to network performance. The need for exploring alternative solutions for parsing and validating XML data has thus assumed prime importance today. The thesis presents an efficient hardware implementation of an XML parser, which ensures that the incoming XML is “well-formed” and “valid”. This involves checking of the document for syntactical correctness and verifying it against the corresponding XML Schema for validity. The system delivers a peak throughput of 1.2Gbps.

Hardware Implementation of an XML Parser

by
Remya Krishnamoorthy

A thesis submitted to the Graduate Faculty of
North Carolina State University
In partial fulfillment of the
Requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2008

APPROVED BY:

_____
Dr.Yannis Viniotis
Chair of Advisory Committee

_____ _____
Dr. Michael Devetsikiotis Dr. Paul Franzon

DEDICATION

To my matha, pitha, guru and daivam…

These are Sanskrit words which translate to mother, father, teacher (advisor) and God respectively. The phrase "matha, pitha, guru, daivam" is also a basic tenet in existence from the time of the Vedas and beyond and represents the hierarchy in which one should respect these entities.

BIOGRAPHY

Remya Krishnamoorthy was born in Thiruvanathapuram, India, in June 1983. She graduated from University of Kerala, India with a Bachelors degree in Electronics and Communications Engineering in June 2004. After undergraduate studies, she worked with Huawei Technologies India Pvt Ltd., Bangalore, for two years. In August 2006, she joined North Carolina State University as a graduate student in the Computer Engineering program. In Summer of 2007, she interned at Cisco Systems, Raleigh in the Service Routing Group. While working towards the Masters degree, she worked on her thesis under the guidance of Dr. Yannis Viniotis.

ACKNOWLEDGMENTS

I express my foremost and deepest gratitude to my advisor, Dr.Yannis Viniotis, for his unstinted support and encouragement throughout the course of my graduate studies at NCSU. I am grateful to him for believing in my capabilities, and for doing the best he could to help me pursue my interests. He has provided me invaluable guidance and direction, and has been a constant source of inspiration for making me achieve what I have.

I sincerely thank Dr. Michael Devetsikiotis, for agreeing to be on my committee and also for giving me the right guidance which helped me decide my research interests. I am also grateful to him for helping to arrange a system and cubicle, where I completed most of my thesis work. I owe my foundations in ASIC Design to Dr. Paul Franzon and I would like to thank him profusely for being on my committee.

I sincerely thank Meeta for the great support she has been throughout the course of the thesis and for her valuable time, patience and suggestions. This thesis has benefited greatly through those discussions and doubt-clearing sessions. I would also like to thank Ravi Jenkal for managing to find time to help me with all synthesis issues and queries. I greatly value his help in this regard.

The university and the department are true centers of learning, offering just about any kind of environment or facilities that one could wish for. I am thankful for the same, with a special mention to the NCSU Wolfline Night service, which ensured the timely completion of the thesis!

I am forever indebted to my parents for giving me the opportunity and freedom to travel this far for my higher education, for their undying faith in me and the wonderful advice and support when most needed. I am also thankful to my sister and brother-in-law for their encouragement and understanding nature throughout my graduate studies. I thank each and every one of my dearest friends I have left behind back home who,

though separated by thousands of miles, have stayed close to heart offering reassurance and strength. Any amount of thanks would not be sufficient for Abhilash, my dear friend and non-academic advisor at NC State for just being there at all times. Finally, I thank the almighty for blessing me with this great opportunity and help me make the best of it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

XML AND XML SCHEMA

This chapter introduces the concepts of XML and XML Schema, and describes their functions and varied uses. Some light is shed on SOA and the growing importance of XML in networking applications.

1.1 Introduction to XML

The following sections outline the evolution of XML as a widely used and accepted markup language, and illustrate with an example the underlying concepts of the same.

1.1.1 History and Evolution

XML was created so that richly structured documents could be transmitted over the internet. After the invention of the internet in 1973, the Standard Generalized Markup Language (SGML) was developed by IBM professionals in 1974, for defining a list of specifications for documents transmitted over the internet. Many application languages were developed based on SGML, the most popular one being the Hyper-text Markup Language (HTML), which was invented in 1993. In the mid-90s, there was rapid growth in the field of web development, and there was a huge surge in demands to support more and more features in web pages. HTML then grew in an unorganized and uncontrolled fashion to cater to the ever-increasing requirements. Features had to be added to support images, animations, multimedia, ecommerce and much more on web-pages. As a consequence, HTML became increasingly display-centric, and there emerged a clear deviation from the original intent to keep content and presentation facets of a markup language separate from each other. This fuelled the formation of a Working Group under the auspices of the World Wide Web Consortium (W3C) to develop a specification

language that could provide the necessary flexibility and scalability to support all necessary features, while keeping the content completely separate from the display and presentation. The E**X**tensible **M**arkup **L**anguage or XML was the solution announced by the W3C XML Working Group in 1996. The committee was chaired by Jon Bosak of Sun Microsystems with active participation of an XML Special Interest Group, also organized by the W3C [20].

The difference between HTML and XML is probably the first question that would arise in the minds of a web developer new to XML. To begin with, the only feature common between the two is the appearance of code in the form of structured tags. XML is like SGML in that it is not itself a markup language, but instead a specification for defining a markup language. On the other hand, HTML is an application, a markup language that is developed based on the specifications of SGML or XML (XHTML). HTML comes bound with a fixed tag set and tag semantics, and does not provide room for arbitrary structure. XML on the other hand, offers unlimited flexibility and freedom to the author of the document in terms of structure and content. The primary goal of HTML is to display data and to focus on how it looks, whereas that of XML is to describe data and to focus on what data is. Thus it would be apt to say that XML and HTML complement each other, with XML being used to structure and describe the web data and HTML to format and display the same data [21].

1.1.2 Definition and example

XML stands for Extensible Markup Language. It is recommended by the W3C and the recommendation specifies both the lexical grammar, and the requirements for parsing. XML is an open standard and can be best defined as “a cross-platform, software and hardware independent tool for transmitting information” [15]. By doing away with the constraints imposed by display-centered languages, there is unlimited scope to structure, store and send data as per one’s requirement.

Figure 1.1 shows an XML document with a customized structure and description of data. As can be seen, the document syntax is simple and self-describing. In order to view this information in a webpage, the display format must be specified in some other language whose function is to receive and display data. The example basically describes the details of an MS Thesis. The starting of an element in an XML is specified with the starting tag "<" character. The top-most element, which in this case is 'MSThesis' is the parent of all other elements in the document and is called the "**root element**". The first "**child element**" of the root is 'department'. It has two "**attributes**", called 'name' and 'university' to specify details. The contents of the 'department' element are the 'student', 'advisor' and 'committee' elements. Each of these leaf-level child elements are associated with some "**content**" which describes the value attributed to them. The XML-specific terms such as element, child, content and attributes are explained in more detail in the following section.

```
<?xml version="1.0"?>
<MSThesis>
  <department name="ECE" university = "NCSU">
    <student>Remya</student>
    <advisor>Dr.Viniotis</advisor>
    <committee>Dr.Franzon</committee>
    <committee>Dr.Devetsikiotis</committee>
  </department>
 </MSThesis>
```

Figure 1. 1 A sample XML document

1.1.3 Key terms and concepts

To understand the uses and functions of XML, it is necessary to gain a preliminary understanding of the parts that comprise a document. A concise list of these is given below with reference to the example illustrated in Section 1.1.2:

1. XML Declaration – <?xml version="1.0"?>

   This is always the first line in an XML document. It describes the XML version number and optionally, the character encoding that is used in the document. By default, the encoding is the ISO-8859-1(Latin-1/West European) character set. The declaration is a mandatory requirement in all XML documents [1,2].

2. Root Element – This is the element at the top-most level of the hierarchy and is the parent of all other elements in the document. The root node is unique within a given XML.

   In the above example, the root element is <MSThesis>

3. Child and Parent elements (Nested elements) – A tag pair nested within another is a child element. The enclosing element is called the parent element. This parent-child structure is called a nested elements hierarchy.

   In the example, the "<department>" element is a child of the "<MSThesis>" element. The "<department>" element in turn, is the parent of "<student>", "<advisor>" and "<committee>" child elements. If an element does not have children or further levels of hierarchy, it is called a leaf-level child.

4. Attributes – When defining an element, it is possible to specify a unique characteristic associated with it using attributes. Each attribute has a name and value associated with it. An attribute is unique within an element.

   In the above example, the element <department name="ECE" university = "NCSU"> has two attributes "name" and "university" associated with it which in turn have values "ECE" and "NCSU" respectively.

5. Content – All child elements and values which occur between the starting and ending tags of an element constitute the content of the element. The leaf-level elements can have only a value as their content.

   In the example above, the root node <MSThesis> has the "<department>" element as part of its contents. The content of leaf-level elements <student>, <advisor> and <committee> however, is made up only of their corresponding values.

The terms and concepts listed above would be used very frequently in the discussions that follow and are the foundation required to develop XML documents of any degree of complexity.

1.2 Uses and importance of XML

The following sections elaborate on the main uses of XML in today's world, and how important it is from the networking perspective.

1.2.1 Uses

Most of the uses of XML stem from the fact that XML is platform independent. This enables storing and sharing data without being concerned about the compatibility with different applications, servers, operating systems or browsers. The following is a list of the varied uses that XML can be put to [23]:

1. XML for documentation and data-sharing – The flexibility and freedom in defining the content and structure of an XML makes it human readable and understandable. XML can be written and shared in plain-text format, which provides a hardware and software independent way of sharing data.
2. XML for Database Development – XML can be used to store data in files or databases. When used for this purpose, the main concern is that the data must be processed easily irrespective of the application or platform being used. XML is independent of hardware, software or the application, making this an easy task.

Any required application can be used to store and retrieve data while a generic application can display the retrieved data.

3. XML for Web development – XML helps to completely separate the content and display facets in web development. The content to be displayed can be contained in multiple XML documents, while the display can be taken care of by HTML. XML can further be embedded within HTML documents in data-islands. This provides effective content management capabilities, since the content managers need not even know the existence of XML. Any change to the content will not require any change in the HTML document, software or the application, making this an easy task. Any preferred application can be used to store and retrieve data while a generic application can display the retrieved data.
4. XML for the internet – The independence with respect to platform, application and technology offered by XML, has resulted in XML technologies being adopted in web services and service oriented architecture, as the new standard for application development and integration. Such services offer rapid interoperability and seamless service re-use by establishing a standard data format and a standard interface. The utility of XML in transmission of data and services over the internet could well be considered to be the prime reason why XML has become indispensable for communication and networking today. This is also the area of maximum interest from the point of view of this thesis and hence, further elaboration of the main concepts is provided in Section 1.2.2.

1.2.2 Importance in networking

This section is further broken down into sub-sections, each describing different application areas of XML in networking.

1.2.2.1 Web-services and SOA

In a typical business environment, there exist a number of disparate applications developed using a wide range of technologies. Since each of these applications may belong to different domains and developed using different technologies, the requests and responses to these applications would need coding of proprietary interfaces and usage of tools specific to each. Due to this, establishing and maintaining communication between these applications would become extremely difficult and resource consuming when the number of such applications is large [4]. Using web services and SOA, the simple invocation of a service's URL would suffice in order to use its capability in another application. Web services and SOA rely heavily on XML for providing this very functionality because of the inherent property of XML to be platform and application independent. Requests and responses are exchanged in the form of XML messages, which are translated back to the native format at the application end using appropriate protocols. The ability to remotely access an application by just sending a request over the internet is the key concept of Web Services [6].

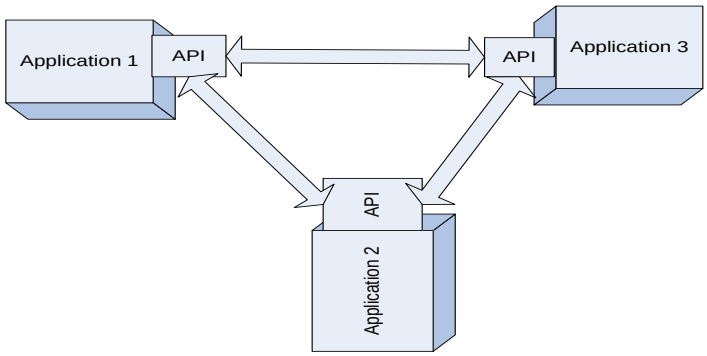

Figure 1. 2 Typical Scenario before SOA

The importance of web-services is further illustrated in Figures 1.2 and 1.3. Figure 1.2 shows the situation before the emergence of XML and SOA, wherein different

applications had to interface with each other using their proprietary APIs (Application Programmable Interface). In Figure 1.3, the APIs are non-existent. A shared web-service layer now resides on top of the application layer which helps the services to easily exchange data and services without any knowledge of the technology or platform underneath.

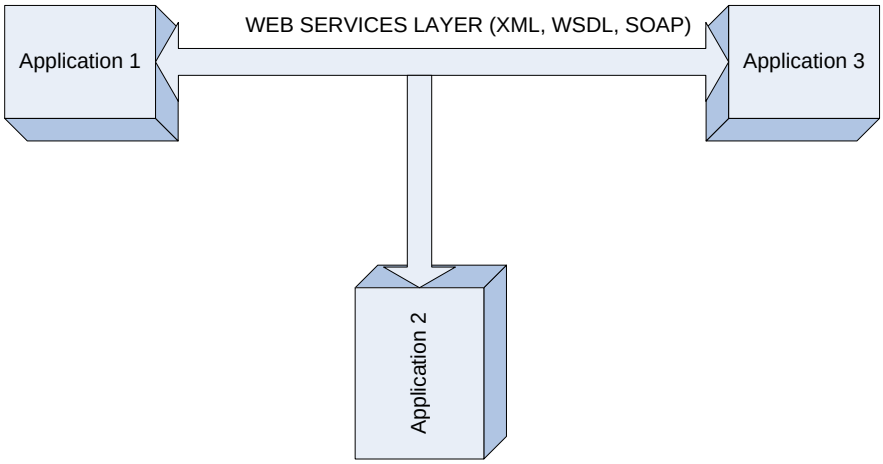

Figure 1. 3 Role of web-services in networking applications

Service Oriented Architecture is often used in conjunction with Web Services and it is used to imply a powerful framework using which applications connect, irrespective of their platform and interfaces. Instead of depending on object instantiation and method invocation to provide interoperability between applications, XML-based standard web services are created to exchange XML documents between applications using internet-based application layer protocols [7].

1.2.2.2 Content-based routing

As the name suggests, Content-based routing refers to the process of routing documents based on their content, rather than based on the header information alone. The uses of content-based routing range from load balancing based on transaction number, prioritizing messages based on the users and destinations and so on. When XML

documents are routed in this manner, the routing decisions are taken based on the evaluation of XPath expressions [12]. For example, to properly route XML-RPC (XML-Remote Procedure Call) messages based on the requested service, the parser identifies which service has been requested by the remote client. A simple XML-RPC request is shown in Figure 1.4. As can be observed, the service name is represented as a string in <methodname> tag. This may be the name of a file that has to be executed, a script to be run or a method to be invoked. The parser has to detect these values using XPATH/XQuery evaluation and forward it to a backend switch which can then route packets to their correct destination.

```
<?xml version="1.0"?>
<methodCall>
  <methodName>getTransactionId</methodName>
  <params>
    <param>
      <value>1.2.3.4</value>
      </param>
    </params>
  </methodCall>
```

Figure 1. 4Sample XML-RPC message

1.2.2.3 Other uses

Using XML over a network provides a bunch of additional uses and features such as:

1. Message Verification using XML Schema – The XML Schema specifies the valid elements and attributes in the XML. It also lays down the constraints on the values of these elements and attributes. By validating an incoming XML against its schema, security of data transmission can be ensured.
2. Transforming XML documents to other formats using XSLT – XSL stands for Extensible Stylesheet Language. XSLT stands for XSL Transformations. It is

basically a language that can be used to transform XML to other document types that could be recognized by standard browsers or user interfaces.

3. Providing Data Security using XML Encryption algorithms – The XML Encryption Recommendation by the W3C describes a process for encrypting data and representing the result in XML either as an XML element or element content. The result of this process is the inclusion of XML "EncryptedData" elements that contain the confidential data.

1.3 Introduction to XML Schema

The following sections introduce the XML Schema illustrating its role in an XML implementation. The key features and concepts are listed and briefly explained:

1.3.1 Definition and example

The XML Schema language is also referred to as XML Schema Definition (XSD). The W3C released the Schema Recommendation in May 2001 [16]. XML Schemas provide a consistent way to validate XML. The predecessor to XML Schema was Document Type Definitions (DTD). A DTD defines the legal building blocks of an XML document, defining the document structure with a list of legal elements and attributes. XML Schema is an XML based alternative to DTDs. They include more features than the DTDs like the support for namespaces and customized datatypes. Future versions of XML will rely on XML Schema for defining XML document types. The benefits of an XML Schema will be further discussed in Section 1.4.2.

In a nutshell, the XML Schema grammar specifies a language that constrains and documents corresponding XML, and thus specifies the valid elements and attributes in the document [3]. It has inherent capability to define numerous features and constraints to completely customize an XML instance.

An example of an XML instance and its corresponding schema definition is shown in Figure 1.5 and Figure 1.6 respectively. The following can be observed from this example:

```
<?xml version="1.0"?>
<MSThesis xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com
thesis.xsd">
  <XML dept="ECE" university = “NCSU”>
    <student>Remya</student>
    <advisor>Dr.Viniotis</advisor>
    <committee>Dr.Franzon</committee>
    <committee>Dr.Devetsikiotis</committee>
  </XML>
 </MSThesis>
```

Figure 1. 5 Sample XML document “thesis.xml”

1. Each element and attribute in the original XML is defined in the Schema along with some valid features and values within the “<xsd:element>” and “<xsd:attribute>” tags respectively.
2. The child elements, as well as the attributes possible for an element must be specified within “<xsd:complextype>” or “<xsd:simpletype>” tags corresponding to the enclosing element. The “<MSThesis>” and “<XML>” elements have corresponding complexTypes “ThesisType” and “XMLType” because they contain other elements they contain nested elements. The other elements (student, advisor, committee) have simple types (not shown in example), because they do not contain other elements.

3. The <xsd:schema> tag encloses the entire document and is the root element of the schema. It specifies the namespace which the elements and datatypes used in the schema come from. The attributes “xmlns”, “xmlns:xsi” and “xsi:schemaLocation” in the root element in the XML, bind the XML to the specified schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="MSThesis" type="ThesisType"/>

 <xsd:complexType name="ThesisType">
  <xsd:sequence>
   <xsd:element name="XML" type="XMLType"/>
</xsd:complexType>

<xsd:complexType name=XMLType">
  <xsd:sequence>
   <xsd:element name="student" type="xsd:string"/>
   <xsd:element name="advisor" minOccurs="1" type=“xsd:string”/>
   <xsd:element name="committee"  type="xsd:string" maxoccurs=“2”/>
  </xsd:sequence>
  <xsd:attribute name="dept" type="xsd:string"/>
  <xsd:attribute name="univ" type="xsd:string"/>
 </xsd:complexType>

</xsd:schema>
```

Figure 1. 6 Schema corresponding to thesis.xml - “thesis.xsd”

1.3.2 Key Concepts

To understand the features provided by the schema, a basic understanding of some of the terms and concepts is warranted.

1. Element Types – These are components of a schema, beginning with the “<xsd:” declaration. They restrict the name, content and attributes of the XML instances. There are a total of 30 possible element types in a schema. Of these, the <xsd:element> and <xsd:attribute> elements are used to define the actual elements and attributes in the XML and hence, the same can be directly found and matched

from the corresponding XML. Others, like <xsd:complextype>, <xsd:sequence>, <xsd:simpleType> etc, are used to add additional features and specifications for each of these elements and attributes. Yet others may be used for documentation, grouping and so on. A detailed discussion of the same is carried out in Section 3.2.1.

2. Content Type – The tags that are defined within a set of enclosing tags form the content of the enclosing element. There is a unique list of content type elements that are possible within an enclosing element type. For instance, in the above example, the <xsd:attribute> is a valid content element within <xsd:complextype>. However, it is not valid within an <xsd:element>.
3. Attribute Type– The attributes that can be specified when defining an element type form the list of valid attribute types for that element. Again, the valid attributes for each of the 30 possible element types are different. When applied correctly, these provide unbounded scope for defining and restricting the structure of the XML. Based on the above example, “minOccurs” and “maxOccurs” are valid attribute types for <xsd:element>. These specify the minimum and maximum number of times respectively, that the element can occur within its enclosing element. In the example shown, the “<advisor>” element must occur at least once whereas the “<committee>” element can occur a maximum of two times.

   A discussion and analysis of the 30 element types is provided in Section 3.1.2.3.

1.4 Uses and Importance of XML Schema

The following sections highlight the main functions, advantages and disadvantages of the XML Schema.

1.4.1 Importance of the Schema

The most important function of XML Schema is to define the valid elements and attributes in an XML document. Using the "<xsd:element>" and "<xsd:attribute>" tags, every element and attribute in an XML is defined and described in the corresponding schema. If an element or attribute in an XML is not present in the Schema, the document is invalid. Again, there are numerous features in the XML Schema recommendation to specify constraints and restrictions on the elements and attributes. It describes for example how many elements can occur and in what order they are valid within a particular parent element. There are provisions to specify datatypes, the range of valid values, defaults values and so on.

1.4.2 Advantages

XML Schemas win over DTDs with their support for datatypes and namespaces. Moreover, since they follow the XML syntax, the developer need not follow a different coding style for defining the rules. Some of the advantages are listed here:

1. XML Schemas used when creating large applications, greatly simplify modularization, resource allocation, testing and deployment.
2. XML Schemas help to secure data communication, by helping the sender of the document best describe the data in a way that the receiver will understand. The information to decipher the structure could be made available only to the intended receiver.
3. XML Schemas are extensible because they are written in XML. This means that it is possible to use a schema in other schemas, define customized data types using the existing ones, and reference multiple schemas in the same document.

1.4.3 Drawbacks

From a practical perspective, the biggest hurdle for using the XML Schema is the learning curve. XML Schemas are incredibly precise and also verbose. Schema description introduces a noticeable amount of overhead to using XML for development. Again, XML validation against the Schema is time-consuming as it also requires parsing XML. However, notwithstanding the disadvantages, XML Schemas today are an indispensable part of XML development.

1.5 Thesis Organization

The organization of the rest of the thesis is as follows. In Chapter 2, we discuss XML Parsing in detail, and include an analysis of the different parsing models which could be used. The goal of the thesis then takes shape. In Chapter 3, we summarize the pre-design phase, listing the different features selected for implementation, the designs that were considered, the final selected design methodology and how it would help achieve the goal better. In Chapter 4, we explain the design of hardware stage by stage with appropriate block diagrams and the functions performed by each block. In Chapter 5, we present the design in more detail with the pin interfaces and architecture of each module. A snapshot of the simulation output obtained for each stage is also shown. In Chapter 6, we discuss the verification methodology adopted, the list of test cases that were executed and finally, we summarize the results for the synthesized design. In Chapter 7, we conclude the thesis and identify areas which have scope for further improvements.

CHAPTER 2

XML PARSING

In this chapter we explain the concept and importance of XML Parsing, the different approaches to the problem and the logical trend towards Hardware Parsing. This background is then used to outline the goal of the thesis.

2.1 Importance of XML parsing

XML Parsing is required to read, update, create or manipulate an XML document. The receiver of an XML document would have to ensure that the XML conforms to the syntactical constructs and grammar that are specified by the W3C to render an XML valid. Again, an XML should conform to the Schema associated with it and allow only those elements and attributes that are defined in the same. To use the information contained in the document, the specific node must be identified which contains the information. It is obvious that each of the afore-mentioned functions is possible only by traversing through the XML document, extracting and validating its contents. This process is XML Parsing. The terms “parsing” and “processing” are often used in conjunction with each other, since it is obvious that either one loses its relevance without the other. The aim of XML parsing is to “process” the document, and it is impossible to process the document without “parsing” it. Figure 2.5 helps to get a better perspective of this process by illustrating the role XML processors play in networking applications.

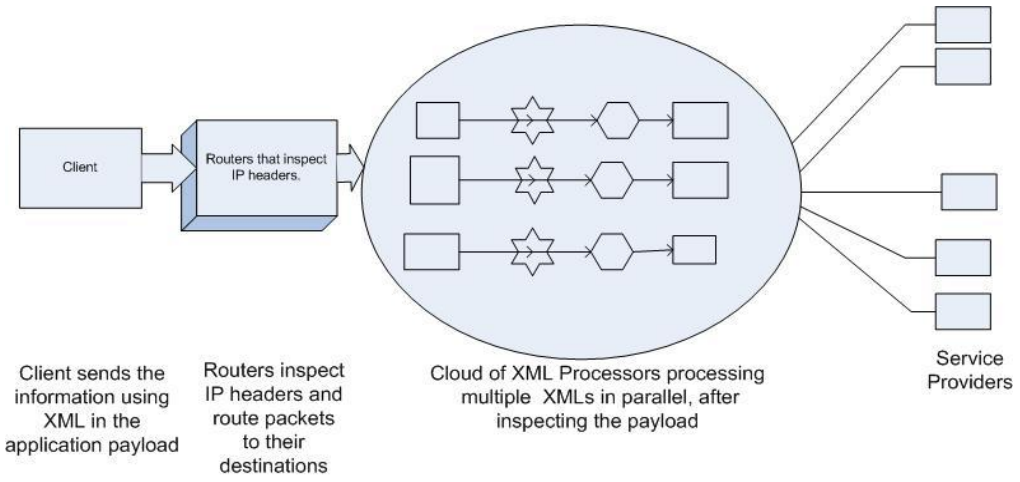

Figure 2. 1 Role of XML parsers in networking applications

In the figure, a client sends data packets over the network consisting of an XML payload. The packet passes through multiple IP routers which inspect the IP headers and route packets to the intended server. The server then offloads processing onto dedicated XML processors, which validate the document and extract the necessary information from the same. Using this routing information, the packet undergoes a last hop to its final destination.

Efficient XML parsing is fundamental to network servers today. As XML payloads become more popular, more and more traffic on the server will be in XML. With more and more XML-aware devices in the network and the rapid acceptance of web-services, the need for efficient parsing becomes paramount, since it directly impacts network performance.

2.2 Stages in XML parsing

Broadly, the complete parsing and processing of an XML document can be thought of as comprising three stages, which are detailed in the following sub-sections. Though it would make logical sense to follow these stages in the given order, different

parsers may choose to implement the functionalities either in parallel, or in a different order. The first two stages are of maximum importance from the thesis perspective, because as will be become clear, the thesis presents a hardware implementation of these stages.

2.2.1 Well-formedness Checking

This stage deals with the syntactic checking of the document. There is a set of rules to be checked in order to certify that the document is well-formed. These rules are explained with the help examples.

1. XML documents must begin with an XML declaration. While it is not mandatory, it is good practice to include it anyway. It can look as simple as shown in Figure 2.2.

```
<?xml version='1.0'?>
```

Figure 2. 2 The XML version element

2. XML documents must have a unique root element. The root element was defined in Section 1.2.3. It encloses all other elements in the document and its name must be unique within the document.

```
<?xml version="1.0"?>
<myxml>
 <firstlevel>
   <myxml>
        123
  </myxml>
</firstlevel>
</myxml>
```

Figure 2. 3 XML with duplicate root element

The example shown in Figure 2.3 is not acceptable since the inner nested element has the same name as the root element.

3. XML elements must be closed. The open and close tag characters are highlighted in red in the XML instance shown in Figure 2.4. There must be no open elements in a well-formed XML document.

```
<myxml attribute1= “123” attribute2= “abc” >
```

Figure 2. 4 XML instance illustrating open and close tags

4. Start tags must have matching closing tags. The starting and ending tags are highlighted in bold in Figure 2.5. It can be noted that the closing tag need have only the element name preceded with the ‘/’ character and need not repeat the attribute declarations of the starting tag.

```
<myxml attribute1= “123” attribute2= “abc” >
        Demo of closing element
</myxml>
```

Figure 2. 5 XML instance illustrating closing of an element

5. XML elements are case-sensitive. Figure 2.6 shows an XML element that has starting and closing tags with the same name but different case. This would not be accepted by an XML Validator.

```
<myxml attribute1= “123” attribute2= “abc” >
        Demo of case-sensitivity
</myXML>
```

Figure 2. 6 XML instance illustrating case-sensitivity of element names

6. XML elements must be properly nested. The example shown in Figure 2.7 illustrates three levels of nested elements. It can be noted that at the second level, there are two elements present, and both have corresponding ending tags. However, the nesting is improper since the </second_level1> closing tag has to occur before start of the <second_level2> start tag.

```
<myxml>
  <first_level>
        <second_level1>
              <third_level>
              </third_level>
        <second_level2>
        </second_level1>
        </second_level2>
  </first_level>
</myxml>
```

Figure 2. 7 XML instance illustrating nesting of elements

7. XML attributes must be always quoted. If an attribute is present, it must have a value, even if it is an empty string. In the example shown in Figure 2.8, even

though attribute2 has a value, it is not valid, since the value is not quoted. On the other hand, attribute1 is valid though its value is an empty string.

```
<myxml attribute1= “” attribute2= myattribute >
```

Figure 2. 8 XML element illustrating quoting of attributes

8. Element names must obey the following XML naming conventions:

    a. Names must start with letters or the “_” character. Names cannot start with numbers of punctuation characters.
    b. After the first character, numbers and punctuation characters are allowed.
    c. Names cannot contain spaces.
    d. Names should not contain the “:” character as it is a “reserved” character.
    e. Names cannot start with the letters “xml” in any combination of case.
    f. The element name must come directly after the “<” without any spaces between them.
    g. XML preserves white space within text.

9. Entities must be used with special characters. There are five special characters that have been identified by the W3C and entities must be used when any of these characters are to be used in the document. For example, if the starting tag character “<” needs to be used in the document, it must be replaced with its corresponding entity which is “<”.
10. Empty elements (<img>, <p> and
) must use end tags like
</br> or closed start tag like
.

11. XML Parser can ignore code within CDATA declaration. XML CDATA starts with "<![CDATA["and ends with"]]>". CDATA is just used to add a comment in a document and does not serve any functionality.

2.2.2 Schema Validation

The validation of the XML document against the corresponding XML Schema is performed in this stage, and it is an integral part of XML parsing. A validating parser checks the document against a schema to check that the elements and attributes that occur in a document are valid. The names and contents of the elements and attributes extracted from the XML document must be checked for validity using the XML Schema. Each element and attribute that is valid in the XML finds a definition in the corresponding Schema, along with other specifications. It is the responsibility of the Schema Validator to check the adherence of the document to each of these rules. The main rules to be validated can be summarized as follows:

1. Check if the particular element or attribute is valid in the document.
2. Check if the element under consideration is a child element.
3. Check the order and number of child elements at any particular level of hierarchy in the document.
4. Check if the contents of the elements and attributes conform to the specified data-type.
5. Check if the contents of the elements and attributes conform to the specified valid range of values.
6. Check if a particular element or attribute is an empty element or can include text. Also check for default and fixed values for the same.

2.2.3 XPATH/XQuery

XPath and XQuery languages provide functions which can be used for navigating through an XML document and extracting required information from it. XPath 2.0/XQuery 1.0 share the same data model and operate on the abstract, logical structure (data model) of an XML document, rather than its surface syntax.

Specifically, XPath makes it possible to refer to individual parts of an XML document and provides random access to XML data for other technologies. It is an expression language for addressing portions of an XML document, or for computing values (strings, numbers, or boolean values) based on the content of an XML document. XQuery uses XPath syntax and is used to query XML data - not just XML files, but anything that can appear as XML, including databases.

2.3 XML parsing in software

In most existing scenarios, the server that receives the XML payload will run JAVA or other software to process it. The packet is offloaded to these XML offload engines, which would parse and return the document in the required format or after extracting the necessary information. The following sub-sections explore the most commonly used parsing models in software, present a detailed analysis of the main bottlenecks faced and suggest possible solutions for the same.

2.3.1 Parsing models

This section introduces and briefly describes the two parsing models on which most existing software parsers are based [17].

The first category of parsing models are the **push and pull parsers** that simply read a XML document and return the data and structure of the document (e.g., SAX and

StAX). Both are event-driven parsers because they return events that the developer has to handle. Push parser implementations like SAX (Simple API for XML) return the data of the whole document in one stream and cannot be stopped. Because a SAX parser generates a transient flow of events, the XML input processing steps (parsing, recognizing, extracting, and mapping) must be performed in a single cycle. Figure 2.9 is an illustration of a SAX parser.

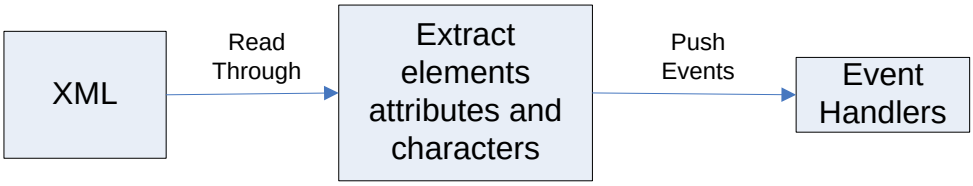

Figure 2. 9 Conceptual illustration of a push -parsing model

Pull-parsers, on the other hand, read a small amount of a document at once. The application drives the parser through the document by repeatedly requesting the next piece, which is analogous to "pulling" the information when required. StAX (Streaming API for XML) is a pull-parser specification for Java. The illustration of the StAX parser is shown in the Figure 2.10.

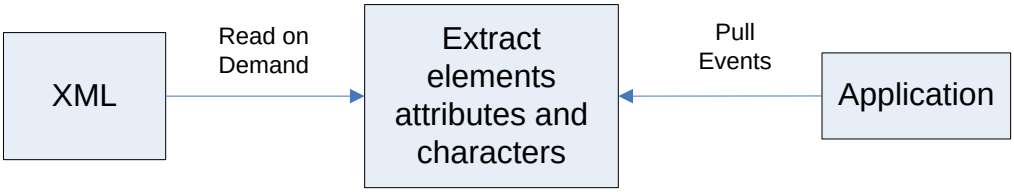

Figure 2. 10 Conceptual illustration of a pull-parsing model

The second type of XML parsers are **object model parsers** (e.g., DOM and Apache AXIOM), which not only read the data but also construct an in-memory representation of the document, which can be altered. First, the parser creates a tree-like data structure, called a DOM (Document Object Model) tree that models the XML source document. Then the application walks through the tree, searching for relevant information to

extract and further process. This last cycle can be repeated as many times as necessary, since the tree persists in memory. An illustration of a DOM parser is shown in the Figure 2.11.

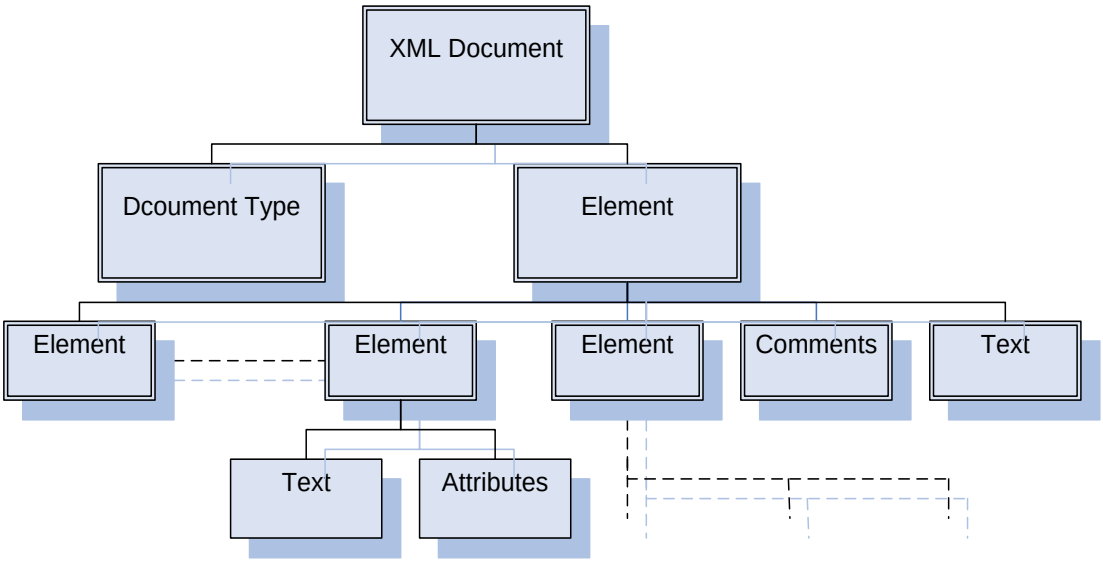

Figure 2. 11 Conceptual illustration of an object-model parser

Since DOM parsers mostly use SAX parsers to read in the documents, it is clear that the object model of a document is always built completely. This is a performance limitation if only data at the beginning of a document needs to be read and altered. New approaches make use of StAX pull-parser implementations to overcome this limitation and only build the tree representation of a document until the last node that was requested. Therefore, there is no need to read the complete document.

Numerous efficient implementations of these two parsing modes exist for XML parsing in non-real time applications [13,14]. However, they fail to scale up to meet performance requirements in a number of application areas in networking. These issues are further discussed in Section 2.3.2.

2.3.2 Bottlenecks in software parsing

Verbosity and performance continue to be the core issues in software parsers. Of these, verbosity is a cost that comes with many of XML's core benefits that include platform independence and human readability. Since bandwidth and disk storage are both becoming less of a concern rapidly and since verbosity has something to do with the physical structure of the document, it can be resolved to some extent. Performance, however, directly relates to improving XML processing efficiency or rather to XML parsing. Observations regarding performance issues suggest that it is not the text parsing and extracting information as such that is slow with XML, but it is the checking for well-formedness, Unicode encoding conversions and the like which make XML parsing very CPU-intensive[19]. With reference to the stages in XML Parsing that were presented in Section 2.2, this means that it is the first two stages, namely, Well-formedness Checking stage and the Schema Validation stage that are the most resource consuming, since they involve repeated iterations through the document and numerous checks for validity and semantic correctness.

Many different implementations and improvisations on the object model and push-pull model parsers have been tried and tested, but the need for better performance continues to plague the software parsers [11]. While the Object model implementations become too memory intensive when used in cases when there is no need to modify XML data, the push-pull models are inefficient in the case when modification of XML is required. It is thus clear that software parsers cannot keep up with the multi-gigabit network performance expected in enterprise data centers. As XML becomes the language of choice in data transmission over the internet and the number of XML-aware network routers and switches increases, there is a need, like never before to explore alternatives to software parsing that can scale up to the high-speed networks of today.

2.3.3 Possible solutions

Since the motivation behind moving away from the software parser implementations is now clear, we explore some of the possible solutions in the following sections.

2.3.3.1 Hardware Offload

Traditionally, it has been observed that whenever there is an emerging performance bottleneck, the first alternative considered is custom hardware, typically an ASIC (Application Specific Integrated Circuit) or FPGA (Field Programmable Gate Array) for enhanced processing capability [22]. Dedicated ASICs performing deep content inspection and pattern matching are a prime reason why web switches and intrusion detection systems (at layer 7) can keep up with multi-GB networks. Since XML payloads are being increasingly used in Content-based routing scenarios, it is natural to look for ways to make use of ASICs to solve XML's performance issue. Other advantages of hardware offload in addition to the promise of better performance include:

1. Reducing workload of application systems
2. Accelerating processing
3. Providing specialized services
4. Enabling centralized services such as load balancing and accounting.

2.3.3.2 Mixed hardware/software offload

An improvement over pure hardware offload could be Mixed Hardware/Software Offload. One idea is to obtain the best of both worlds by moving some of the computationally intensive and resource consuming tasks to hardware and implement the other functions using software applications. Similarly, non-real time processing could be

carried out by software. Yet another possibility is to perform a pre-evaluation on the incoming document using software, to check the complexity of the document. The pre-evaluation stage determines the improvement that could be obtained by hardware offload. Based on this evaluation, the document is offloaded to hardware or software parsers. The chief advantage of mixed parsing is that an optimal utilization can be accomplished of both hardware and software systems capabilities, by distributing the tasks in the correct manner between them.

The thesis basically presents a hardware solution to the actual validations required in the Well-formedness and schema validation stages. At the same time, much of the complex parsing of the schema is left to the Schema pre-processor, which would basically run a software application. Software processing can be afforded for this non-real-time function and it generates the pre-processed schema which is the starting point for the hardware to perform schema validation. Hence, taking into account the software pre-processing module would qualify the current implementation as a mixed parsing solution rather than a pure hardware accelerator.

2.4 XML parsing in hardware

The emergence of dedicated hardware to validate, process and route XML traffic started during 1999-2002. Initially, different vendors focused on specific areas such as speed, security and routing. The early players in the field were Data Power [18] (now owned by IBM) which had speed as the top priority, Forum Systems which aimed to tackle data security, and Sarvega (now owned by Intel) which focused on intelligent routing. The term “XML Appliances” is used to collectively refer to these hardware implementations. Some alternative terminology that has been widely used to refer to XML Appliances based on their key functionality is given below:

XML accelerators — Devices that typically use custom hardware or software built on

standards-based hardware to accelerate processing. This hardware typically provides a performance boost between 10 and 100 times in the number of messages per second that can be processed.

XML security gateways (also known as XML firewalls) – Devices that support the general security standards. These appliances typically offload encryption and decryption to specialized hardware devices.

In essence, the XML Appliances offload the XML processing from the application servers which could get overtaxed by this heavily resource consuming task. After performing the required validation, translation and other required processing, the requests contained in the XML payload are forwarded to the appropriate server [8, 9, 10].

2.5 Thesis problem statement

The thesis presents a mixed parsing solution to combat the performance bottleneck faced in XML parsing. As we had discussed in Section 2.3.2, most of the complexity in the parsing process is introduced in the validation of the XML before actually extracting information from it. Hence, rather than focusing on a complete end-to-end implementation, the idea was to implement the validating stages in XML parsing, with a throughput goal of 1Gbps. The system would check that the incoming XML document is valid in terms of well-formedness and content. In terms of the stages in XML parsing discussed in Section 2.2, the thesis presents an implementation of the first two stages, namely Well-formedness checking and Schema Validation. Existing implementations for XPath/XQuery processors could be interfaced with this design to come up with a complete parser that can process documents at Gigabit rates.

The process involved a detailed analysis of the stages involved in the parsing process and the functionalities therein. The most important and time-consuming part of the thesis would be the extensive study of the possibilities and features in the validation

stages and segregating the rules to be checked based on their importance, complexity and hardware offload efficiency. The features that would be most relevant and important from a proof of concept perspective were selected for implementation. The ultimate goal was to come up with a successful, synthesizable design implementing the selected features, with throughput efficiency as the main objective.

CHAPTER 3

REQUIREMENTS ANALYSIS AND DESIGN CONSIDERATIONS

In this chapter, we describe in detail the set of requirements for a complete XML Validator, the subset of features selected for implementation and some ideas considered before finalizing on the design.

3.1 Features Implemented

This section describes the process of selecting features for implementation. All features that define the main functionality sought by the validation stages were completely implemented. However, numerous options and features are available for use in XML documents that would be preferable in a commercial product, but would not add much value towards helping the thesis objective. Hence, while analyzing those features, the focus was not to implement as many as possible, but rather to implement those features that would be the most throughput intensive and complex. A careful analysis of these features in the well-formedness checking stage and the schema validation stage, followed by their eventual selection or elimination is now presented.

3.1.1 Well-formedness Checker

The rules that qualify an XML document to be "well-formed" have been explained in detail in Section 2.2.1. Most of the rules for well-formedness are fundamental to XML validation and hence they were all implemented without exclusion. A summary of these rules and some constraints on the same in the current implementation is presented in

Table 3.1 and the rules that are not considered are listed in Table 3.2.

Table 3. 1 Main functions and implementation constraints for the well-formedness checker

Rule	Constraints
XML documents must begin with an XML declaration. <?xml version='1.0'?>	Only for the "version" attribute is supported in the current implementation
XML documents must have a unique root element	None
XML elements must be closed	None
Start tags must have matching closing tags	None
XML elements are case-sensitive	None
XML elements must be properly nested	16 levels of nesting hierarchy supported
XML attributes must be always quoted.	Implemented support for up to 8 different attributes for each element.
Element and attribute names must start with letters or underscore and have no spaces or the colon character.	None
Element names must not start with "xml" in any combination of case.	None
Entities must be used with special characters	None
Empty must be supported with closed start tag like 	None
Attributes in an element must be unique	None

Table 3. 2 Validations not covered by well-formedness checker

Rule	Reasons
XML CDATA starts with <![CDATA["and ends with"]]>	CDATA is equivalent to information elements in other languages and does not support any important feature or requirement. The relatively simple task of validating syntactical correctness of CDATA elements would not affect design time, complexity or performance, but would increase the coding and verification time significantly.

3.1.2 Schema Validation

The Schema Validator stage validates the elements and attributes in the XML with respect to their name and value. Besides this core functionality, it is the Schema that gives enormous flexibility to the developer in terms of defining the XML's structure and imposing restrictions on the content. Each of the available options in XML Schema are explored in as much detail as would be practical within the thesis scope. The reasoning behind selecting or rejecting a feature and the eventual evolution of the final list of selected features is then presented.

3.1.2.1 Main functions

The basic functions of the schema have been discussed in Section 2.2.2. Due to their functional importance, all the rules are validated in this regard. Table 3.3 presents a summary of the main functions implemented in the Schema Validator stage and the constraints, if any.

Table 3. 3 Main functions and implementation constraints in the schema validation stage

Function	Constraints
Element name is valid in the document	Length of the name is restricted to 32 characters
Attribute name is valid in the document	Length of the attribute name is restricted to 32 characters
Check if the element is a valid child element.	Implemented with support up to 16 levels of hierarchy.
Check the order of occurrence of child elements	Support "sequence" ordering of elements. This means that all the child elements specified in the schema must be present and also present in the specified order in the XML

Table 3. 3 Continued

Check data types and values of the elements	Implemented with constraints. Datatypes supported : string, integer, decimal, date Restrictions supported: maxlength, minInclusive, totdigits, and fracdigits. These are explained in a later section.
Check data types and values of the attributes	Implemented with constraints. Datatypes supported : string, integer, decimal, date Restrictions supported: maxlength, minInclusive, totdigits, and fracdigits. These are explained in a later section.

3.1.2.2 Analysis of element types

XML Schemas can comprise of up to 30 different elements for defining an XML and the Schema has to be carefully parsed to extract the necessary information. An analysis of these elements is carried out in the following sub-sections.

3.1.2.2.1 Basic element types

This group lists the main element types that are used to define the most functional aspects of an XML. This relates to defining the elements and attributes, defining the hierarchy of elements and specifying restrictions on the same. The list of these elements is presented in Table 3.4.

Since the functionality of each of these elements is very important, the thesis implements all of them. The complexType and simpleType nodes are always attached to a parent <xsd:element> and do not exist on their own. Hence, these elements are assumed to be flattened out by the pre-processor and their contents directly included within the element they belong to.

Table 3. 4 Basic element types in the schema

Element Type	Details
Attribute	Defines an attribute
complexType	Defines a complex type element and specifies the elements and attributes that can occur within the parent element.
Element	Defines an element
Restriction	Defines restrictions on a simpleType, simpleContent, or a complexContent
simpleType	Defines a simple type and specifies the constraints and information about the values of attributes in a leaf-level elements.
schema	Defines the root element of a schema

3.1.2.2.2 Documentation related

The elements in this group used to provide comments or information in the schema and are listed in Table 3.5.

Table 3. 5 Documentation-related element types

Element Type	Details
Annotation	Specifies the top-level element for schema comments
appInfo	Specifies information to be used by the application (must go inside annotation)
documentation	Defines text comments in a schema (must go inside annotation)

Since these elements are used only for providing documentation related functionality and play no part whatsoever in XML validation, they can safely be omitted from the design.

3.1.2.2.3 External reference/extensions

The elements in this subset are listed in Table 3.6 and are either used for accessing elements and attributes that are outside the current schema document, non-schema elements or for extending a schema element with the properties of another element. The

"simpleContent" and "complexContent" nodes are used only to define extensions and impose restrictions on them.

References to external namespaces and documents are difficult to support in hardware, especially so at run-time. In the current implementation, these references are resolved by a pre-processor and all the necessary information is present in the loaded schema. Similarly, all extensions are also flattened out by the schema pre-processor and are included in the appropriate locations in the final schema.

Table 3. 6 References/Extensions related element types

Element Type	Details
Any	Enables the author to extend the XML document with elements not specified by the schema
anyAttribute	Enables the author to extend the XML document with attributes not specified by the schema
extension	Extends an existing simpleType or complexType element
attributeGroup	Defines an attribute group to be used in complex type definitions
group	Defines a group of elements to be used in complex type definitions
complexContent	Defines extensions or restrictions on a complex type that contains mixed content or elements only
import	Adds multiple schemas with different target namespace to a document
include	Adds multiple schemas with the same target namespace to a document
notation	Describes the format of non-XML data within an XML document
redefine	Redefines simple and complex types, groups, and attribute groups from an external schema
simpleContent	Contains extensions or restrictions on a text-only complex type or on a simple type as content and contains no elements

3.1.2.2.4 Identity related constraints

This group of elements validates the uniqueness of a particular element within a containing element. The list of all such elements is given in Table 3.7.

Table 3. 7 Identity related element types

Element Type	Details
Key	Specifies an attribute or element value as a key (unique, non-nullable, and always present) within the containing element in an instance document
unique	Defines that an element or an attribute value must be unique within the scope
keyref	Specifies that an attribute or element value correspond to those of the specified key or unique element
Field	Specifies an XPath expression that specifies the value used to define an identity constraint
Selector	Specifies an XPath expression that selects a set of elements for an identity constraint

The implementation of any of these nodes at best requires the implementation of a binary search algorithm to parse through the nodes and check that the particular element is not repeated. This would be trivial but time-consuming and hence identity constraints are ignored for the purpose of the thesis. It would definitely be implementable without affecting performance too much at any later stage.

3.1.2.2.5 Model group constraints

The elements in this group are listed in Table 3.8 and they specify the number and order of elements to be present within a constraining element. There are essentially three categories of model groups, and implementing each of them would require a different set of rules to be checked for while validating the element as described in the table.

Table 3. 8 Model group element types

Element Type	Details
All	Specifies that the child elements can appear in any order. Each child element can occur 0 or 1 time
Sequence	Specifies that the child elements must appear in a sequence. Each child element can occur from 0 to any number of times
Choice	Allows only one of the elements contained in the <choice> declaration to be present within the containing element

It was decided to implement the most frequently used constraint, namely the "sequence" constraint, and in the current design, all the element hierarchies present in the XML will be assumed to be of this type. The "choice" and "all" constraints would require sorting of the elements so as to select one value from among a list. Implementing these features would add no value to the thesis in terms of innovation, but would considerably increase the time for completion and hence, these constraints were given lower priority for implementation.

3.1.2.2.6 Lists/Unions

The elements in this category are used to specify that the contents of an element can belong to a list or may be a union of values. Their functions are explained in Table 3.9.

Table 3. 9 Element types specifying lists/unions

Element Type	Details
List	Defines a simple type element as a list of values
union	Defines a simple type as a collection (union) of values from specified simple data types

Some additional checking and, hence, combinational logic would get added to the design if these nodes are chosen. Since it does not really contribute anything in terms of intellectual property, and also because they are very rarely used in reality, their validation is not handled in the current implementation.

3.1.2.3 Complexity introduced by the schema

Though we have now analyzed the possible element types in an XML Schema document, more complexity and decision-making is involved because of further rules governing possibilities of elements and attributes for each of these basic element types. For instance, only certain elements are permitted at the global level, and further, each of these elements have defined for them:

1. “content-types” - which specifies the list of nested elements possible for the particular enclosing element and
2. “attribute-types” - which define the attribute possibilities for the element type.

A detailed analysis of the content-types and attribute-types for each of these 30 elements would both be unwarranted and beyond the scope of this thesis. However, for clarity, one such element is considered and the content-types and attribute-types listed out. An analysis is then carried out to demonstrate the process of selection and rejecting features.

Explanation with example

Let us consider the node <xsd:element>, which contains information related to an element in the XML.The content-types possible for the element are:

1. annotation
2. Complextype
3. Simpletype
4. key
5. keyref

6. unique

```
<to>
  <name>
    NCSU
  </name>
</to>
```

Figure 3. 1 XML instance definition

```
<xsd:element name = "to" type= "toType"/> ----------→ 1
    <xsd:annotation>                      ----------→ 1.1
        <xsd:documentation>               ----------→1.1.1
                This defines a simple element
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType name = "toType"> ----------→1.2
        <xsd:sequence>                ----------→1.2.1
            <xsd:element name= "name" type= "xsd:string">→1.2.1.1
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Figure 3. 2 XML schema definition corresponding to Figure 3.1

Consider the XML instance definition shown in Figure 3.1.The corresponding definition of the "<to>" element in the schema is shown in Figure 3.2. The element "<to>" in the XML is defined within an "<xsd:element>" declaration in the schema. This element type contains <xsd:annotation> and <xsd:complexType> as contents. These are represented as 1.1 and 1.2 respectively in the example. The contents that are to be considered are now selected based on the analysis carried out in the section 3.1.2.2. Thus,

among the content-types encountered, 1.2 would be selected and 1.1 rejected. In summary, from the 6 possible content types that are possible for the <xsd:element>, the ones that would be selected for implementation for the are:

1. <xsd:simpletype> and
2. <xsd:complextype>

Now, the attribute possibilities within an <xsd:element> in the schema and the analysis for the same are presented in Table 3.10.

Table 3. 10 Attributes possible within the <xsd:element> and their analysis

Attribute	Function	Selected/ Rejected	Reason
Default/fixed	Specify a default or fixed value for an element if a value is not present.	Rejected	This increases the coding and verification time, since writing back to the memory has now to be taken care of, while this adds no innovation to the design.
Form/ abstract/ block/final/ref/ substitutiongroup	Different attributes all related to reference to namespaces or substitutions	Rejected	Parser does not support references or extensions
nillable	Specifies whether it is ok for that an to be empty.	Rejected	Based on the default/fixed attribute implementation
Type	Specifies the complex or simple type that defines the element value and attributes or to the datatype that the value of the element belongs to.	Selected	Very important for functionality
id	The unique id of the element	Rejected	Identity constraints are not supported

Table 3. 10 Continued

maxOccurs/ minOccurs	Specifies the maximum or minimum number of times that an element can occur within an enclosing element.	Support only value of 1 for both of these attributes	1. Not one of the main functions. 2. Undue wastage of clock cycles and throughput 3. Increases the complexity in terms of the number of possibilities to be considered, and the number of values to be remembered.

A similar kind of analysis was carried out for each of the selected element types before coming up with the final list of features in the schema.

3.1.2.4 Implemented functions

Table 3.11 summarizes the discussion and lists the implemented features in the current design, the features indirectly implemented using the pre-processor and the features not considered at all.

Table 3. 11 Summary of implemented functions in Schema validation stage

Elements supported directly during parsing	Elements whose functionality is supported indirectly by pre-processing	Elements not considered
<xsd:element> <xsd:attribute> <xsd:restriction> <xsd:sequence>	<xsd:extension> <xsd:include> <xsd:import> <xsd:complexcontent> <xsd:simplecontent> <xsd:group> <xsd:attributegroup> <xsd:notation> <xsd:redefine> <xsd:any> <xsd:anyattribute>	<xsd:annotation> <xsd:documentation> <xsd:appinfo> <xsd:schema> <xsd:key> <xsd:unique> <xsd:keyref> <xsd:field> <xsd:selector> <xsd:all> <xsd:choice> <xsd:list> <xsd:union>

3.2 Designs considered

This section presents an overview of the designs and ideas considered during the evolution of the final design selected for implementation. The pros and cons of all ideas are also discussed.

3.2.1 Pre-processor based design

This design was based on a pre-processing stage which would take in the XML document and re-arrange the same in an internal memory, such that each new starting tag would start on a new line.

The motivation behind this design was that the identification of the starting tag character is fundamental to all requirements in the Well-formedness checker stage, and so, by eliminating the need to check for and match the '<' character over and over again, the complexity is greatly reduced. The idea is that only the first character in every location in memory would need to be checked for a '<' character. By widening the memory width to 128bytes, the number of bytes read in one read would increase, and hence the throughput could be increased.

However, a basic flaw in this design was the dependence of the throughput on structure of the document. The ultimate aim is to achieve a throughput of 1Gbps for the system. But the presence of the pre-processing stage would defeat the idea of achieving an absolute throughput of 1Gbps. Instead, we would have to consider the average throughput. This is because, even if 128bytes of actual data are being read in to the pre-processor, the pre-processor re-aligns the bits in memory, and in the process introduces mask bits for the empty locations. If a starting tag is read in as the 2nd byte in a memory read, the pre-processor inserts 127 bytes of mask characters, and the throughput for that particular cycle would be 1/127th of the peak throughput. Due to this inconsistency and inability to predict the position of the starting tags in a sample document, the average

throughput could never reach 1Gbps. The worst case throughput will in fact be (1/127)th of 1Gbps.

Before the design was scrapped, different work-around methods were considered. The difference between the worst and peak throughput could be reduced by decreasing the memory width. However, that would again lead to a decrease in throughput because of decreased number of bytes processed in a single read. Hence, achievement of a 1Gbps throughput looked elusive with this design from all angles.

3.2.2 4-byte design

Once the pre-processor had been decided against, an efficient pattern matching algorithm was thought to be inevitable. The characters read in every memory access have to be checked to match different pre-identified sequences to confirm the well-formedness of the document.

However, since the pattern lengths and content were variable, it is clear that larger the number of characters in one read, the more complicated the design would be, as the number of possible sequences would increase proportionally. For instance, consider that the memory is 64-bytes wide. Assuming a worst case of 16 nested elements, each with an element name that is 1 byte long, there can be a maximum of 16 starting tag characters in one read. Thus even after identifying the starting tag characters, priority decoders would have to be used to decide the first starting tag. The complexity of identifying possibilities for the remaining characters that follow would be a different story altogether. It was clear that the design would get out of hand unless the number of characters read in was reduced. The 4-byte design was basically an attempt to use this concept and limit the number of bytes per read to 4 and evaluate if the number of states required was within reasonable limits. In this manner an FSM was attempted and the number of possibilities listed out. However, the complexity was still too high and it was

evident that coding and verification would get out of hands. Hence, this design was decided against too.

3.3 Selected design

The following sub-sections describe the selected design in some detail and discuss the advantages and drawbacks of the same.

3.3.1 Concept and explanation

The current design involves reading the XML document into the system one byte at a time from memory and checking it for correctness against the characters that are valid for that read. The valid characters are determined by the design logic based on the previous character. Once sufficient bytes have been read so as to have extracted an element name, attribute name, element content, or attribute content, the data is passed to the schema validator block, which then checks the for conformance to the pre-processed schema. Once this completes, the next byte is read in from the memory.

The concept of average throughput is explained here for a fair understanding of the design concept. The schema validation involves complex tasks such as determining the next element expected and matching the attribute with the list of valid attributes. Hence there may be instances when it would require multiple iterations for validation. Due to this, it is inevitable to include pipeline stalling wherein execution is stalled for a few clock cycles until the validation is completed. Once the byte read in is passed on to the schema validation stage, a backpressure signal is asserted on the well-formedness and memory access stages for as long as the schema validation takes to complete processing. Thus, there are some cycles in which the stages process one or more bytes, and some cycles in which they do not process any bytes at all. The throughput can only be calculated in terms of the total number of bytes processed with respect to time.

Average throughput is defined as: Total number of bytes processed/Total number of clock cycles.

3.3.2 Strengths

The main strengths of the design have been identified as below:

1. Simplicity – The design is aimed at high performance, which is mostly achievable only by reducing complexity. By handling only one byte at a time, most of the document is processed even as it is read in. Multiple bytes read in would increase the number of possibilities, the need for priority decoders, and the need to verify if a particular combination has been left out. A simple design directly implies less effort but greater clarity in terms of coding and verification.
2. Clock speeds – Higher clock speeds, and hence higher throughput can be achieved using this design, as the complexity of the combinational logic is reduced.
3. Pre-processor in Schema Validator – Since the pre-processing of the schema need not be done at real-time, it can be assumed that the schema is available in the required pointer format which is specified in the design. One significant advantage of the same is that we can make use of the non-real time processing advantage of the schema, and completely do away with the need for variable-length pattern matching in this stage. This would have been the most complex part of the design if not for the pre-processor and it would have been the major bottleneck as far as throughput was concerned. It would have necessitated a robust pattern matching algorithm and multiple parses through the schema in order to validate the XML successfully.
4. Minimal buffering requirements – In any design that involves reading in more than one byte at a time, there would be the need to buffer the previous read contents, so as to predict the combinations possible in the current read. The one-byte design helps minimize this requirement.

5. Memory requirement – The memory that will be consumed by the schema is much lesser than the memory that would be required if the whole document were to be stored.

3.3.3 Drawbacks

Some of the possible drawbacks of the design are listed here:

1. Memory requirement scalability – The number of locations needed in the memory is directly proportional to the number of nested levels supported in the XML, or the number of attributes possible for an element, as will become clear in later sections. Thus for complex XML documents, the memory would have to be large enough to allow for as many backpressure cycles as may be required. Since this is an on-chip memory, it would affect the area and power parameters of the chip.
2. Pre-processing the Schema – Extensive schema parsing, possibly carried out in software is required by the design in order to convert the schema document into the customized format required by the system.

CHAPTER 4

DESIGN ARCHITECTURE AND DESCRIPTION

This chapter will introduce the hardware design, with a top-level block diagram. Each of the core modules is then described and the design and functionality of each is explained.

4.1 High level block diagram and explanation

Figure 4.1 shows the high-level block diagram of the implemented design. As observed, there are two main pipelined stages, namely the Well-formedness Checker stage (WF) and the Schema Validator stage (SV). There are two memory modules corresponding to each of these modules. The memory connected to the WF stage (WF memory) contains the XML document and the on-chip Schema Validator memory feeds the pre-processed schema to the SV stage when required. Again, the WF memory is dynamic in that it keeps reading the bytes that make up the XML one at a time and constantly replaces the processed bytes with new ones. The Schema memory on the other hand is populated by the pre-processor well before the start of processing and is not changed until the entire document has been processed.

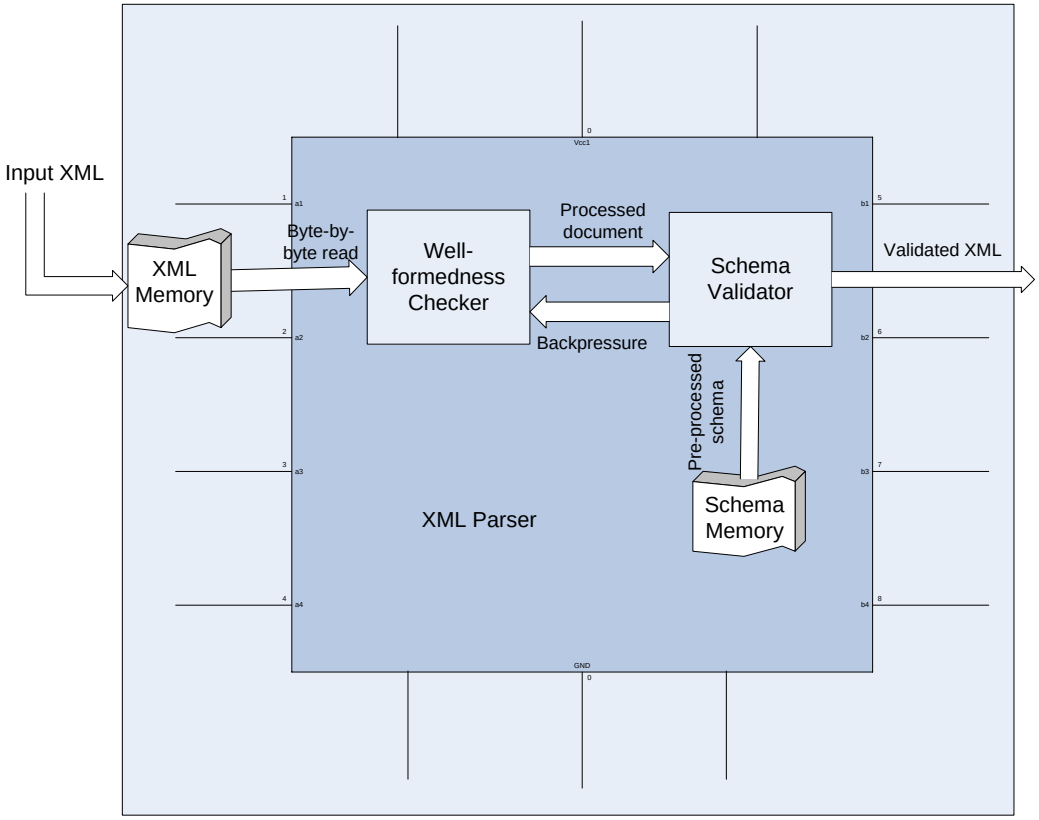

Figure 4. 1 High-level block diagram of the system

XML bit stream from the server is read one byte at a time, into the 8 - bit wide Well-formedness checker memory. At every clock cycle, one byte of data is fed to the WF stage. The data read in is checked for conformance to the well-formedness rules for XML documents defined by the W3C. The set of characters that are valid in a particular read from memory are pre-determined by the logic in the WF stage. The incoming character is then compared with the valid set of characters. Once validated, the set of possibilities for the next byte is determined by the WF stage. After extracting names or contents of elements and attributes, the data is passed on to the SV stage. SV action is requested in the following conditions:

i. Check whether the extracted element is a valid one at that particular location in the XML

ii. Check whether the extracted element content conform to the restrictions and constraints imposed on it.
iii. Check whether the extracted attribute is valid for the particular element.
iv. Check whether the extracted attribute content conforms to the restrictions imposed on it by the schema.

In each of these cases, the SV stage executes the appropriate logic and validates the data with information from the XML schema. Success or failure of validation is signaled to the WF stage. If more than one iteration or memory read is needed by the Schema Validator to complete its validation, backpressure is exerted on the WF stage. When this happens, the entire pipeline operation is stalled for as many cycles as the SV stage takes for processing. This process continues until the end of the document is reached. If no errors have been flagged by either stage until that time, the document is valid. At any point of execution, when an error is encountered, further processing or validation is not carried out and the document is discarded.

4.2 Memory Organization

The following sections elaborate on the memory organization in the design. The XML and Schema memories which are the main memory modules, as well as other memory structures used in the design are described here.

4.2.1 XML Memory

The XML memory is designed to be 1-byte wide and 8 words long. Data is written into the memory locations every clock cycle in a sequential manner, replacing the existing data with new one on looping back after 8 cycles to the starting location. Data is also read from the memory every clock cycle until backpressure is exerted from the SV stage, in which case the pipeline is stalled. This process continues until the memory is full, which implies that the read and write locations equal each other. Since the length of

the memory is 8 locations in our implementation, this situation happens when backpressure is exerted for 8 cycles.

The length of the memory has been chosen arbitrarily and just serves to demonstrate the intended functionality. The memory length denotes the number of cycles for which the system can remain unaffected by backpressure exerted by the schema validator stage. Naturally, the longer the memory, the better the throughput of the system. Thus, the length is something that can be chosen depending on the complexity of documents that the parser would handle and the available or affordable memory. If the parser would be handling complex XML documents with a large nested element hierarchy for instance, backpressure would be exerted by the SV stage more frequently, and a relatively big memory would be advisable.

4.2.2 Schema Memory

The schema memory is an on-chip implementation and stores the XML Schema corresponding to the XML under consideration, in the customized structure and format that can be understood by the design. The organization of information in the Schema memory is slightly complex and the schema pre-processor is responsible for the same. The pre-processor parses the actual schema document, extracts the required elements from it as also includes the information contained in extensions and reference elements need to be included at appropriate locations. All this information is then filled into a 512-bit wide memory structure in the pre-defined format.

The length of the memory in the current implementation is 128 words. The length is arbitrary and is proportional to the number of different elements and attributes that can occur in the XML. Hence, based on the complexity of the schema and the XML processed, it may be necessary to increase the length further. Again, in the current design, the length of element and attribute names is restricted to 32 characters. If needed, longer names can be supported by increasing the memory width. Since the memory is

implemented as an on-chip solution, the commercial viability of the memory dimensions would not be a concern [21].

Each location in the memory corresponds to a different element or attribute and each byte stores specific details about these elements and attributes. The memory organization has a heavily pointer-based structure. An element stored in memory contains pointers for the start and end locations of the nested elements possible within it, as also the start and end locations for the attributes that can be associated with it. The structure of the memory is now explained bit-wise so that the role of the pre-processor and the internal organization becomes clear. Table 4.1 shows the bit-wise break up of one location in the Schema memory.

Table 4. 1 Bit-wise organization of the schema memory

Byte position	Function	Default value	Comments
0:31	Element name/Attribute name	'< '	The starting tag character('<') is the default character in all locations
32	Datatype	03 –String	One of 4 values 00-string, 01- integer,02 – decimal, 03- date
33:34	Maximum length	4000	Since only 64 characters are supported in our design, the default used is 64.
35:41	MinInclusive	0	
42	Total Digits	18	
43	Frac Digits	0	
44:47	Start location pointer for nested elements	'<'	
48:51	End location pointer for nested elements.	'<'	
52:55	Start location pointer for the attributes	'<'	
56:59	End location pointer for the attributes	'<'	

Table 4. 1 Continued

60:63	Pointer to the parent location	'<'	This is the pointer to the parent location of the current element at any point during the execution. This could be modified during execution when the same element exists within different parents.

4.2.3 Other memory structures

Some other memory structures that have been used in the design are described in this section.

4.2.3.1 Element stack

In order to check the proper nesting of elements in the incoming XML, an element stack is maintained in the Well-formedness checker stage which stores elements as and when they are extracted from the document. When a closing tag for an element is found, it is checked with the last element entered in the stack. The two element names must match if the nesting of elements is proper. In the thesis, a 16-level nesting hierarchy is supported and a maximum element name length of 32. Hence the element stack is a register array that is 256 bit wide and 16 words long.

4.2.3.2 Attribute stack

The attribute stack is maintained in the Well-formedness stage and its purpose is to check the uniqueness of an attribute in an element. This is done by repopulating the stack with the attributes extracted, every time a new element is found. An error is flagged if an attribute is found that matches one that already exists in the stack. A maximum of 8

attributes are supported in an element and the maximum attribute name length is 32. Hence, the attribute stack used is 256 bits long and 8 words long.

4.3 Well-formedness checker

The well-formedness Checker (WFC) stage is responsible for implementing the functions that have been identified in Section 3.1.1. The fundamental validation requirement in this stage can be described as checking that a particular byte, name or value is valid at a particular location in the XML. This is accomplished with the help of an extensive state machine that analyses the incoming character to decide a list of possibilities for the next character. It also governs the actions to be taken on encountering specific characters at each of these states. This kind of detailed prediction has been made possible because the XML specification has clearly marked out the structure of the XML and the criteria for well-formedness. For example, on detecting a starting tag character, the FSM can safely assume that the start of the new element has been reached. The detection of a space character immediately following a starting tag can be flagged invalid. Similarly, in each state, by examining the character read in, the next state, as well as the actions to be taken is determined and the necessary validations carried out.

Figure 4.2 presents a closer look at the Well-formedness checker stage. As seen in the figure, the WF stage has been broken down into six main modules:

1. FSM Controller
2. Element Name Validator
3. Element Content Validator
4. Attribute name validator
5. Attribute content Validator
6. Closing element name Validator

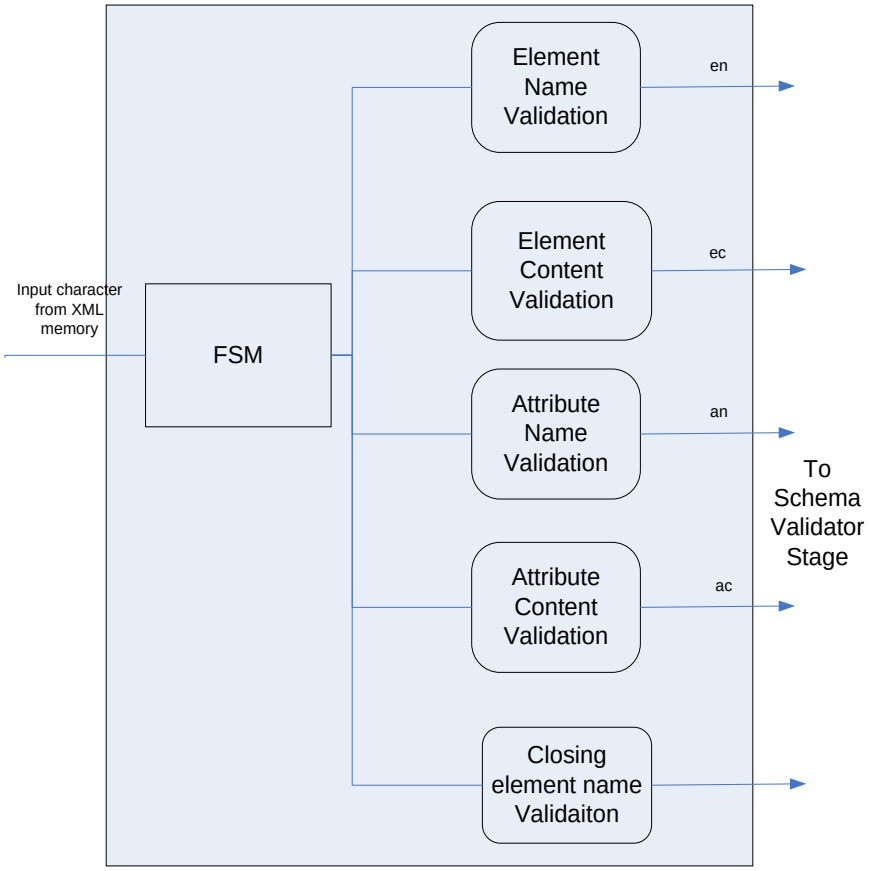

Figure 4. 2 Block diagram of the well-formedness checker stage

The functions of each of these modules are now explained in detail.

4.3.1 FSM Controller

The FSM or Finite State Machine module generates and maintains the state information for the system. The remaining modules in WF become active or inactive depending on the FSM state. Moreover, a unique set of validations and checks are carried out in each state and the well-formedness of the character being processed is determined.

Since we deal with only one character at a time, complex pattern matching algorithms to determine correctness and conformance can be completely avoided. The state transitions in the FSM used in the implementation is shown in the Figure4.3. Since the FSM controls the working of all other modules in the WFC, a detailed discussion on

the action taken during each state is now presented.

State 1: This is the initial state of the FSM in which the system expects a starting tag character to detect the start of the document or a new element. Once processing has started, this state is reached every time an element declaration has closed and so the characters read in this state constitute the element content.

State 2: This state is reached after a starting tag has been found. Thus, all characters read in this state are part of an element name. On encountering a space, slash or closing tag character, the end of the element name can be assumed

State 3: This state is reached after an element name has been extracted. The characters read in this state constitute an attribute name. On encountering an equal to character, the end of an attribute name can be assumed.

State 4: This state is reached after an attribute name has been extracted. Either a single or double quote must now be present to contain the attribute value.

State 5: This state is reached after a single quote has been detected in state 4. The characters read in this state constitute the attribute value. A closing single quote denotes that the value has been extracted.

State 6: This state is reached after a double quote has been detected in state 4. The characters read in this state constitute the attribute value. A closing double quote, when found, denotes that the value has been extracted.

State 7: This state is reached when a closing element has been encountered. This refers to detection of a starting tag character followed by a slash ('</'). The characters that follow, until encountering a closing tag character, make up the closing element name. Once the closing element name has been extracted, the same can be used to check proper nesting of elements.

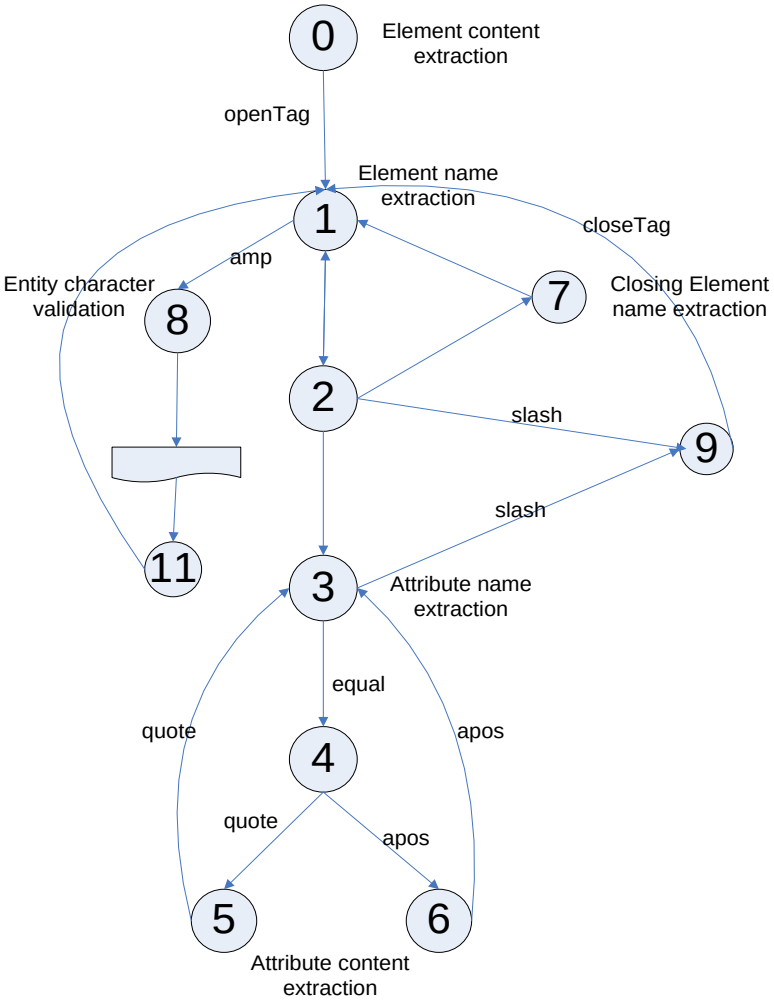

Figure 4. 3 The FSM state transitions

State 8: This state is reached whenever the ampersand character is read as part of an element or attribute's value. This denotes the presence of an entity character. The entity must conform to one of the 5 pre-defined special characters. Entities have been pre-defined for 5 special characters in XML as shown in Table 4.2. Depending on the character encountered state 8, control is passed to states 10 to 18 which deal exclusively with checking that the entity declarations are correct [5].

Table 4. 2 Special characters and their corresponding entities

Character	Entity Element
<	<
>	>
&	&
‘	'
“	"

State 9: This state is reached when an empty element is encountered. This means that a slash character has been encountered in an element declaration, which must hence be followed by a closing tag character.

4.3.2 Element name validator

This module executes when the FSM is in State 1. The logical sequence of functions of this block can be summarized as:

1. Check each incoming byte for conformance to XML element name specifications.
2. Check that the element name does not start with the letters “xml” in any combination of case.
3. On detecting a space or closing tag character, pass the element extracted to the schema validator stage and signal the successful extraction of an element. Simultaneously, add the element to the element stack.
4. On receiving confirmation from the schema validator stage that the element name is valid at that particular location in the XML, clear flags and registers related to this module and proceed to the next validation.

4.3.3 Element content validator

This module is executed when the FSM is in state 0. Again the sequence of functions performed by this module can be summarized as follows:

1. Check the incoming character for validity and extract the element content.

2. Check that entity elements are used with special characters and are defined correctly.

3. On detecting an open tag, signal the successful extraction of the content and send the extracted element content to the schema validation stage to check that the value adheres to the restrictions imposed on it,

4. On receiving confirmation from the schema validator that the element value conforms to all constraints, clear all flags and registers related to this module and proceed to the next validation.

4.3.4 Attribute name validator

This module is initiated in state 2 and its functions are almost similar to those of the element name validator module. They can be summarized as follows:

1. Check the incoming character for validity as per the attribute name specifications

2. On finding an “equal to” character, alert the schema validator of the extraction of an attribute name and add the attribute to the attribute stack, after checking for uniqueness.

3. On receiving confirmation from the schema validator that the attribute name is valid, clear all registers and flags related to this module and proceed to the next validation state.

4.3.5 Attribute content validator

This module is active in state 5 or state 6 of the state machine. The functions performed by this module are:

1. Check the incoming character for validity and populate the attribute content register with the value.

2. Check that entity elements are used with special characters.

3. On finding the closing quote for the attribute, signal the successful extraction of the content and send the extracted attribute content to the schema validation stage to check that the value adheres to the restrictions imposed on it.

4. On receiving confirmation from the schema validator that the attribute value conforms to all constraints, clear all registers and flags related to this module and proceed to the next validation.

4.3.6 Closing element name validator

This module is executed in state 7 of the state machine, when a closing element has been encountered. The functions of this module can be summarized as follows:

1. Check that the closing element name adheres to XML element name specifications.

2. Check that the element is closed with a closing tag character.

3. When a correct match has been found, delete the last entry in the element stack.

4.4 Schema Validator

The schema validation stage is responsible for checking if the element and attribute names sent from the WF stage are valid in the given XML at the particular location. It also checks that the element and attribute values adhere to the constraints set on them in the schema. The different blocks that make up the Schema Validator are shown in Figure 4.4. As shown, the schema validator is made up of four modules, each dealing with a different validation requirement. A detailed explanation of these modules is now provided.

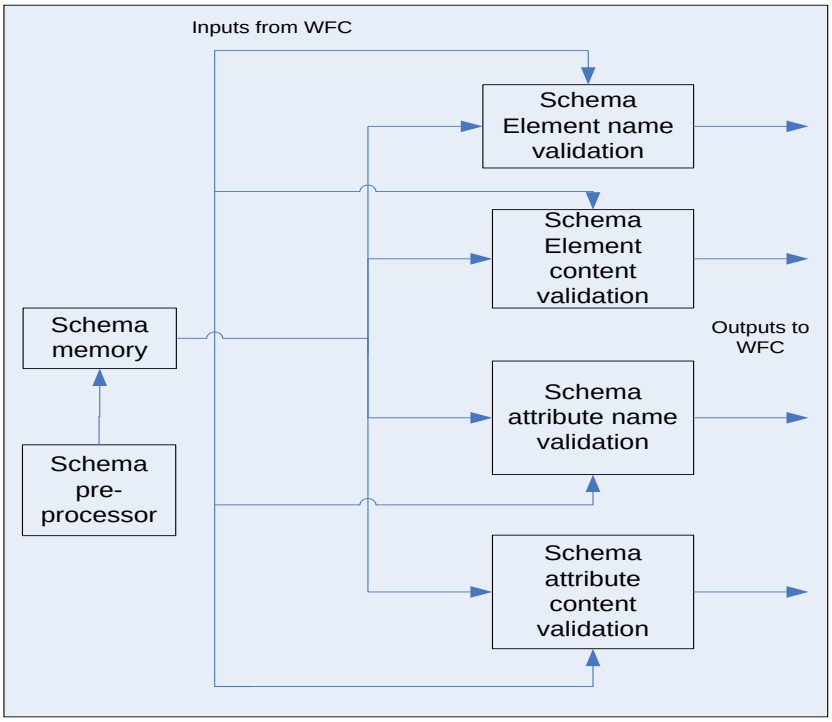

Figure 4. 4 Block diagram of the schema validator

4.4.1 The Schema pre-processor

A lot of effort and time spent in performing the complicated task of extracting information from the schema is avoided by offloading this function to the pre-processor which generates the detailed pre-processed schema. The comprehensive pointer structure helps to predict the next element that should be expected from the WF stage. It also gives information about the range of memory locations to be checked for an attribute match. In a nutshell, when an element or attribute name is received by the Schema Validator, there is no need to check each and every memory location for a possible match. Instead, the specific range of locations to be checked is determined by the logic and the validity can be ascertained by direct comparison with the element names at those locations. Again, specific bits in each memory location represent the various constraint limits and specifications for the contents of the attributes or elements they belong to. Hence, by

leaving the complicated task of parsing the actual XML schema to the pre-processor and dealing directly with the extracted information in our proprietary format, the schema validator design is greatly simplified.

4.4.2 Schema element name validation stage

This module is activated after an element name has been extracted by the well-formedness checker. The WF stage validates that the name is correct syntactically. The onus is now on the SV stage to check matching of the element name with that in the current location in the schema memory. If a match is found, the location of the next element is determined by the logic. This module also extracts the constraints on the element under consideration and the location of its possible attributes. All these functions can ideally be completed in one clock cycle. However, if the current element is at the end of one level of hierarchy, computing the next element pointer in the parent level could take multiple cycles of reading and comparing elements from the memory. In this situation, backpressure would have to be exerted on the preceding stages.

4.4.3 Schema attribute name validation stage

The main function of this module is to check that the attribute that was extracted by the WF stage is valid in the XML and within the current enclosing element. Each element location in the schema memory contains the start and end locations of the attributes that are valid within it. On receiving an attribute, the SV stage first compares the name with that of the attribute at the starting location. On a mismatch, the attributes at all successive locations until the end location are checked until a match is found. Once the end location of attributes has been reached and a match has still not been found, it is ascertained that the current attribute is not valid.

4.4.4 Schema element content validation stage

The principal function of this module is to check that the element content adheres to data type specifications and other constraints that have been specified in the schema. In the current implementation, four main datatypes are supported. These are the string, integer, decimal and date datatypes. The datatype corresponding to the current element is part of the bit-structure of each element in the schema memory.

Again, each of these data-types allows only a subset of characters to be part of the element content and some such as the date datatype, even specify a unique format for the content. Thus, the preliminary validation performed by this module is to check the contents against the character specifications of the corresponding datatype. Once that is done, the constraints on the contents have to be evaluated. Again, a total of four constraints have been implemented in the design and not all of them are valid across all datatypes. The table summarizes the data types and the constraints valid for each. These constraints are now explained in some detail:

1. maxlength – Specifies that the content should not be limited to the number of characters specified. By default, up to 4000 characters can be allowed in the content space for an element or attribute. In the current design however, the default has been limited to 64 for ease and practicality of implementation and testing.
2. totDigits – This constraint specifies the total number of digits that are valid in an integer or decimal value. The XML specification requires that only a maximum of 18 digits need to be supported and the implementation follows the same rule.
3. fracDigits – This refers to the maximum number of digits that can occur after the decimal point in a decimal data type. The constraint on the total digits is 18 by default and hence 17 for fracDigits.

4. minInclusive – This represents the minimum value that the integer or decimal data type should have to be valid. The default for this constraint is 0.

Table 4. 3 Data-types supported and their constraints

Code	Datatype	Constraints
00	String	maxlength
01	Integer	minInlcusive totDigits
02	Decimal	minInclusive totDigits fracDigits
03	Date	None

4.4.5 Schema attribute content validation stage

The functions of this module are mostly similar to that of the element content validation module, the only difference being that the content being considered belongs to an attribute rather than an element. Since there is no other difference in the functionality implemented, the details are not repeated here.

CHAPTER 5

DETAILED DESIGN AND SIMULATION

This chapter explains the implementation details of all the main modules in the design. The pin-interfaces, data and control flow diagrams and simulation snapshots are provided.

5.1 Memory Read module

The memory read module is responsible for reading in XML data from the main servers, releasing one byte of data at a time to the WFC stage.

5.1.1 Pin Interfaces

The main interfaces to the memory read module are listed and explained in tabular form in Table 5.1.

Table 5. 1 Pin interfaces for the memory read module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN		Asynchronous system reset
backPressure	IN	1	This signal tells the module that reading has to be stalled until the schema validator stage completes processing.
wfcErr	IN	1	The error signal from the WF stage. If it is asserted, further reading of data is stopped.
schemaErr	IN	1	The error signal from the schema validator stage. If it is asserted, reading of data is stopped.
readBus	OUT	8	Contains the next byte of data from the XML document.

5.1.2 Detailed Architecture

The architecture and overall working of the memory module is now explained with the help of the pin interfaces described in Section 5.1.1. The module first reads XML data from an external memory. This data is then stored in the 8-bit wide and 8-word long XML memory. The read and write addresses to this memory are generated every clock cycle based on the input signals. These addresses are ideally incremented every clock cycle, the only exception being the condition when the 'backPressure' signal is asserted and the memory read needs to be stalled. The addresses loop back to the start location on reaching the end of the memory. A better understanding of the sequential flow of events could be obtained by analyzing the simulation output for this module shown in Figure 5.1.

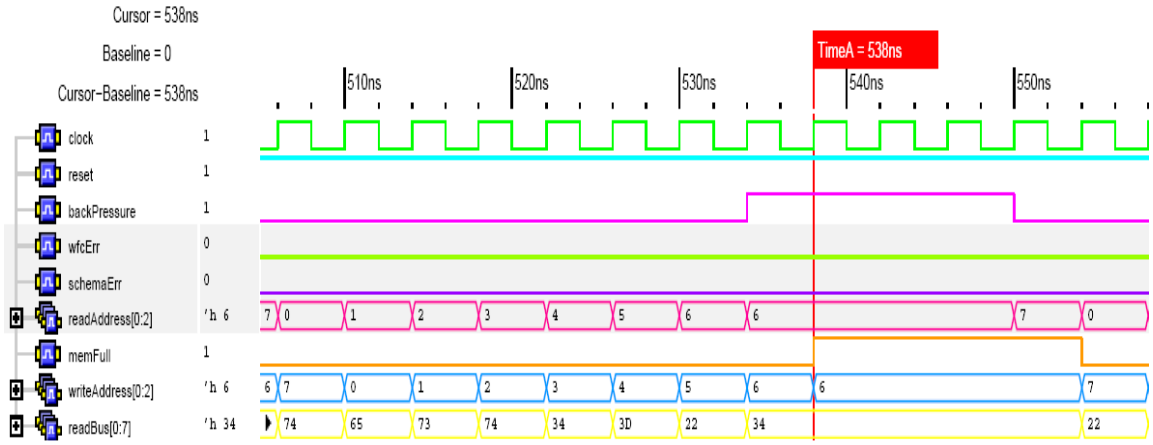

Figure 5. 1 Simulation output for the memory read module

The 'readAddress' and 'writeAddress' registers are shown in the same figure. It can be seen that when the 'backPressure' flag is asserted, the readAddress is not incremented further until the flag is de-asserted again. When the read and write locations coincide, the 'memFull' flag is set. After this happens, writeAddress is not incremented further. It resumes only after the readAddress has advanced one location and the flag is de-asserted.

If either one of the error flags, 'wfcErr' or 'schemaErr' is asserted, reading is completely stopped and the 'stopExec' signal is asserted to signify disqualification of the XML being processed and declaring it invalid.

5.2 Well-formedness checker

This is the first main functional stage in the implementation. The functions performed by this block have been explained in Section 4.3. The implementation details of the core modules are now explained in the following sub-sections.

5.2.1 Element name validation

The element name validation module performs the basic well-formedness checks while extracting an element from the incoming XML.

5.2.1.1 Pin Interfaces

The important interfaces to the Element name validation module are shown in tabular form in Table 5.2

Table 5. 2 Pin interfaces for the element name validation module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
charRead	IN	8	Contains the XML character read in from memory
curr_state	IN	5	The current state of the FSM, maintained by the controller module.
elmNameValid	IN	1	Confirmation signal from the Schema validator stage signaling that the element name under consideration is valid.
En	OUT	1	Asserted by the module after the element name has been extracted
elmName	OUT	256	Contains the 32-character element name extracted by this module which is free from well-formedness errors.

Table 5.2 Continued

elmNameErr	OUT	1	Error flag raised by the module to signal that there is a well-formedness violation in the element name.

5.2.1.2 Detailed architecture and Simulation Results

The operation of the module is now explained with the help of the pin interfaces descried in Section 5.2.1.1 and the simulation output shown in Figure 5.2.

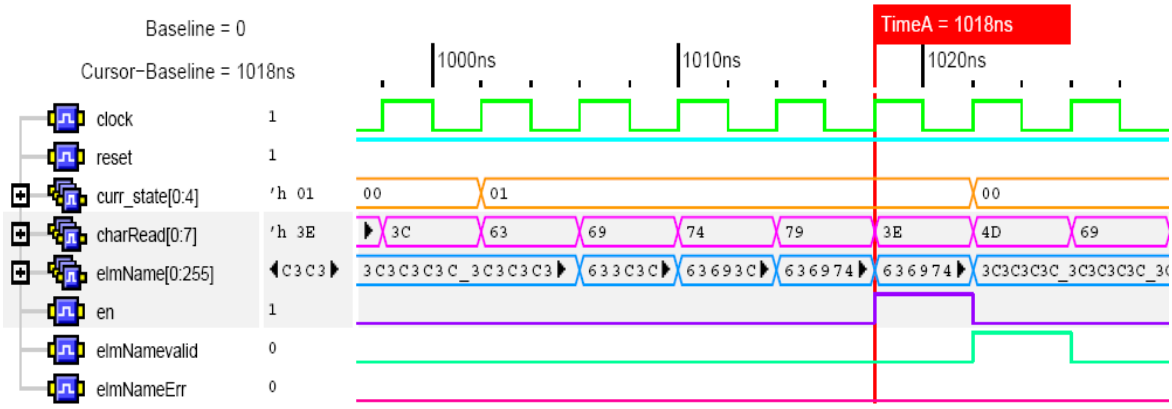

Figure 5. 2 Simulation output for the element name validation module

The module recognizes the start of an element when the 'curr_state' maintained by the FSM changes to 1. Element name validations are then performed on the 'charRead' every clock cycle and once validated, the character is stored in the 'elmName' register. A maximum length of 32 characters is allowed for the element name. When the 'curr_state' changes and no errors have been found, the 'en' flag is asserted to signal the successful completion of element extraction. The contents of 'elmName' are then passed to the schema validator stage. On receiving the 'elmNameValid' flag from the schema validator stage, the 'en' flag is de-asserted and all the registers are cleared. On finding any error, the module asserts the error flag 'elmNameErr'. All these signals are shown in the timing diagram Figure 5.2.

5.2.2 Element content validation

This module checks that the characters that comprise the content of an element are valid. Checks for entity characters in the content are also involved. Finally, the element content is extracted and sent to the next stage for constraints validation as per the schema.

5.2.2.1 Pin Interfaces

The main interfaces of this module are shown in Table 5.3.

Table 5. 3 Pin interfaces for the element content validation module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
charRead	IN	8	Contains the XML character read in from memory
curr_state	IN	5	The current state of the FSM, maintained by the controller module.
elmValueValid	IN	1	Confirmation signal from the Schema validator stage signaling that the element value under consideration is valid and satisfies the required constraints.
ec	OUT	1	Asserted by the module after the element value has been extracted
elmValue	OUT	256	Contains the 64-character element value extracted by this module which is free from well-formedness errors.
elmValueErr	OUT	1	Error flag raised by the module to signal that there is a well-formedness violation

5.2.2.2 Detailed architecture and Simulation Results

The functioning of the module is now explained with the help of the pin interfaces information given in Section 5.2.2.1 and the simulation output shown in Figure 5.3. The module kick-starts when the ‘curr_state’ input passed from the controller

module has a value of 0. The characters read thereon are checked for validity to be part of element contents and if found valid, are populated in the elmValue register. A maximum of 64 characters has been supported and the 'elmContscount' register keeps track of this limit. When an 'openTag' is detected, the curr_state changes denoting the end of content characters. The 'ec' flag is then set to signal to the next stage that the 'elmValue' register contains the element contents that are ready for the next set of validations. The schema validation stage sets the 'elmValueValid' flag on successful validation and this leads to the de-assertion of the 'ec' flag and clearing of all registers in this module. Any errors found would lead to the setting of the 'elmValueErr' flag.

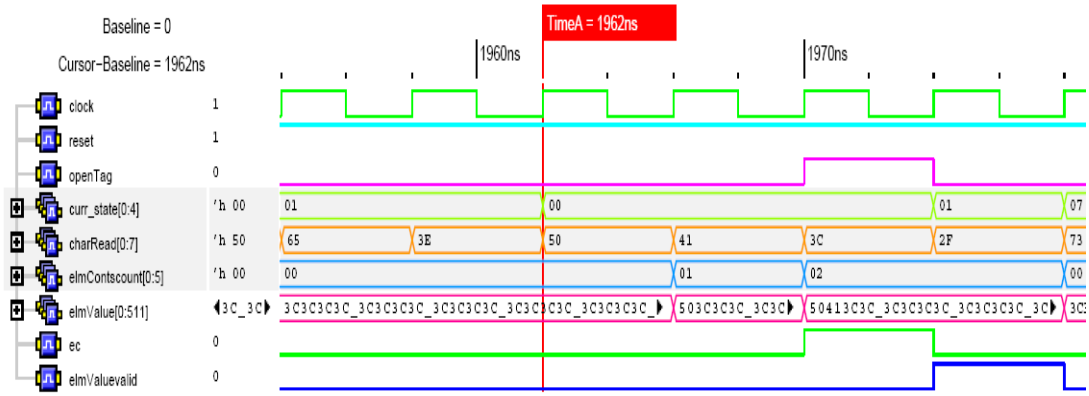

Figure 5. 3 Simulation output for the element content validation module

5.2.3 Attribute name validation

This module is responsible for performing well-formedness checks on the attribute names in the XML.

5.2.3.1 Pin Interfaces

The main input and output interfaces of this module are shown in Tale 5.4.

Table 5. 4 Pin interfaces for the attribute name validation module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
charRead	IN	8	Contains the XML character read in from memory
curr_state	IN	5	The current state of the FSM, maintained by the controller module.
attrNameValid	IN	1	Confirmation signal from the Schema validator stage signaling that the attribute name under consideration is valid within the particular element.
an	OUT	1	Asserted by the module after the attribute name has been extracted successfully.
attrName	OUT	256	Contains the 32-character attribute name extracted by this module which is free from well-formedness errors.
attrNameErr	OUT	1	Error flag raised by the module to signal that there is a well-formedness violation in the attribute name.

5.2.3.2 Detailed architecture and Simulation Results

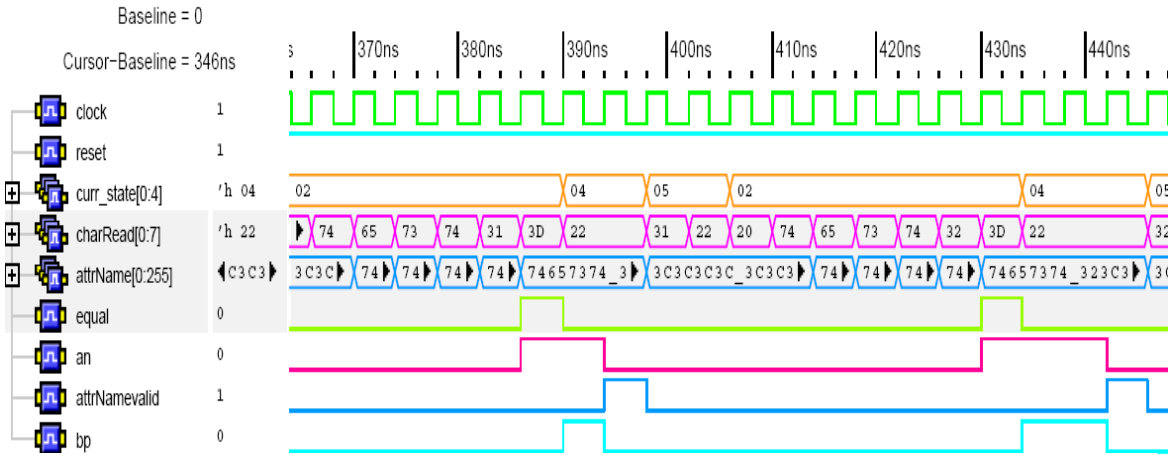

Figure 5. 4 Simulation output for the attribute name validation module

The operation of the module can be better understood by observing the signal transitions in the simulation output shown in Figure 5.4. This module becomes active

when the 'curr_state' maintained by the controller block has a value of 2. The XML data passed in through the 'charRead' input is then checked for the standard syntactical correctness with respect to an attribute name. If found valid, the data is stored in the 'attrName' register. The 'equal' flag is asserted when the "equal to (=)" character is encountered. This denotes the end of the attribute name and the 'an' flag is then asserted, signaling to the schema validator stage that the attribute name is ready for further validation. When the schema validator completes successful valiadation, it asserts the 'attrNameValid' flag, on receiving which the 'an' flag is de-asserted and all other registers in the module are cleared. Any errors found will result in the setting of the 'attrNameErr' flag.

5.2.4 Attribute content validation

This module is responsible for checking the well-formedness of the attribute contents in an XML.

5.2.4.1 Pin Interfaces

The main interfaces to the attribute content validation module are listed in Table 5.5.

Table 5. 5 Pin interfaces for the attribute content validation module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
charRead	IN	8	Contains the XML character read in from memory
curr_state	IN	5	The current state of the FSM, maintained by the controller module.
attrValueValid	IN	1	Confirmation signal from the Schema validator stage signaling that the attribute value under consideration is valid and satisfies the required constraints.

Table 5. 5 Continued

ac	OUT	1	Asserted by the module after the attribute value has been extracted
attrValue	OUT	256	Contains the 64-character attribute value extracted by this module which is free from well-formedness errors.
attrValueErr	OUT	1	Error flag raised by the module to signal that there is a well-formedness violation in the attribute contents.

5.2.4.2 Detailed architecture Simulation Results

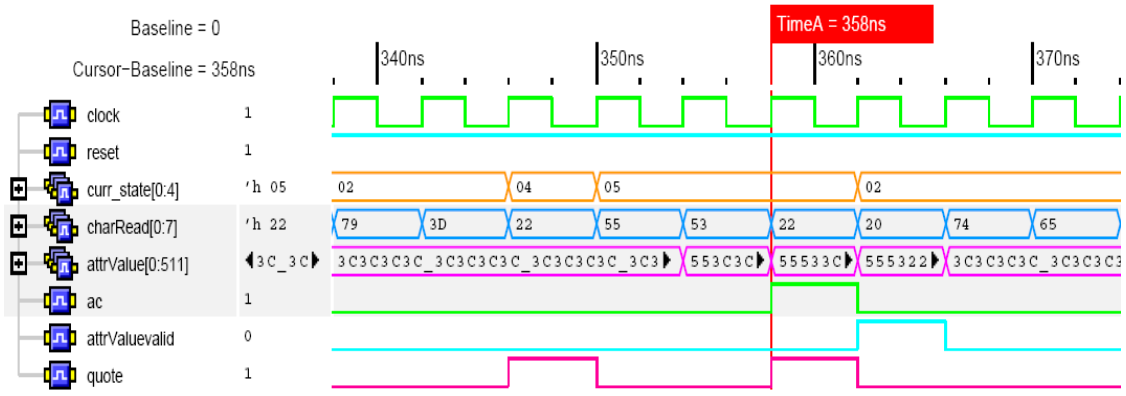

Figure 5. 5 Simulation output of the attribute content validation module

For a more in-depth understanding of the sequential flow of events in the module, the simulation waveforms shown in Figure 5.5 can be used. This module is active when the 'curr_state' maintained by the controller module is in state 5 or state 6. When in this state, the characters read in through the 'charRead' input correspond to the value of an attribute present in the XML. The contents are validated for basic well-formedness checks. The proper use of entity characters is also checked in this module. If the character is valid, the 'attrValue' register is populated with the value byte –by-byte, until the closing quote or apostrophe is detected in the document. When this happens, the 'ac' flag is raised to denote that attribute content validation has been successfully completed by the well-formedness stage and the value is present in the 'attrValue' register for further validation by the next stage. A maximum of 64 characters is allowed for storing

content in this implementation. Once the schema validator completes checking of the value against the constraints and restrictions imposed on it, the 'attrValueValid' flag is set. The 'ac' flag is then de-asserted and all registers used by the module are cleared and reset to their original values. Any errors if found in this module will cause the 'attrValErr' flag to be set.

5.2.5 Controller module

This module is comprised of the state machine which determines the functioning of all other modules in this stage. The transition of states is based on a detailed analysis of the possibilities of occurrence of characters in an XML document.

5.2.5.1 Pin Interfaces

The main inputs and outputs to this module are described in tabular form in Table 5.6.

Table 5. 6 Pin interfaces for the controller of the well-formedness checker stage

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
charReadIn	IN	8	Contains the XML character read in from memory
elmNameErr	IN	1	The error signal from the element name validation module.
elmValErr	IN	1	The error signal from the element content validation module.
attrNameErr	IN	1	The error signal from the attribute name validation module.
attrValErr	IN	1	The error signal from the attribute content validation module.
bp	IN	1	The back pressure signal from the schema validator stage.
curr_state	OUT	5	The current state of the FSM which decides the operation of all other modules.
charRead	OUT	8	The character read in from memory and stored in the controller.

Table 5. 6 Continued

wfcErr	OUT	1	The error signal representing the well-formedness stage as a whole, formed by the individual error signals from the different modules.

5.2.5.2 Detailed architecture and Simulation results

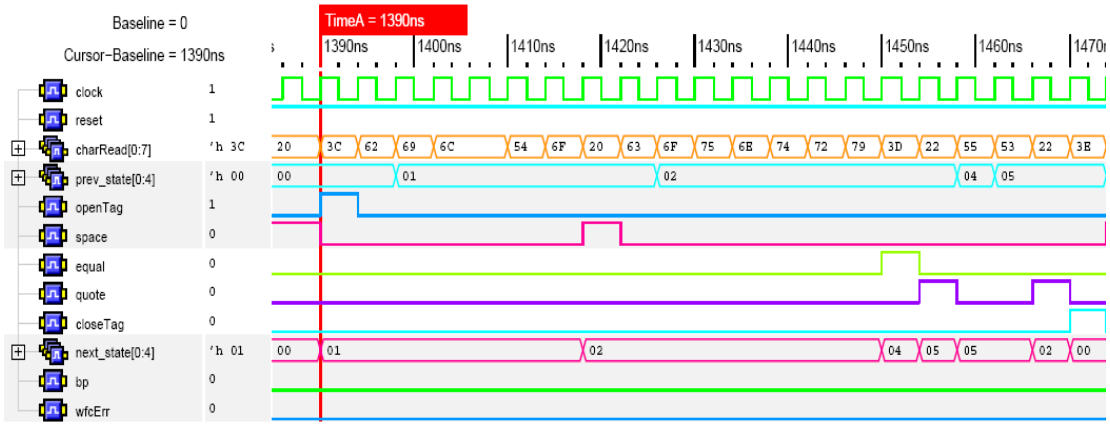

Figure 5. 6 Simulation output for the controller in well-formedness checker stage

The controller is the most important module in the design since everything else runs based on the outputs from this module. The state machine has been designed by analyzing the possible sequence of characters in an XML document and the transitions of the important signals in the module can be seen in the simulation output shown in Figure 5.6. The valid characters in each state have been carefully defined and the conditions to be satisfied for transitioning to the next state have been identified. Basically, there are 5 important characters that determine the state transitions. These can be identified as 'openTag', 'space', 'equal', 'quote' and 'closeTag' in Figure 5.6. The names of these signals are self-explanatory. The character read in from memory is stored in the 'charRead' register in the controller. Based on this character, the 'next_state' is updated. This value is reflected in the 'curr_state' output signal. On encountering the 'bp' signal, the FSM maintains its current state and no operation is performed. The error signal 'wfcErr' is generated based on the values of the error signals from each of the other modules, namely

'elmNameErr', 'elmValErr', 'attrNameErr' and 'attrValErr'. In the situation when any one of these is set, an error is flagged by the controller to signal a well-formedness violation.

5.3 Schema Validator

This stage is responsible for checking the element and attribute names and contents against the corresponding schema and verifying their conformance to the same. The implementation of the modules that make up this stage is now explained in some detail.

5.3.1 Schema memory

The schema memory is an on-chip implementation of registers that store the details of the schema corresponding to the XML being processed in a pre-defined structure. The implementation details are explained in the following sub-sections.

5.3.1.1 Pin Interfaces

The main interfaces to this memory module are shown in Table 5.7 below.

Table 5. 7 Pin interfaces to the schema memory module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
parentElmPtr	IN	32	Contains the pointer to the parent location in memory.
nextElmPtr	IN	32	The location from where the next element is to be fetched from memory
currAttrPtr	IN	32	The location from where the next attribute is to be fetched from memory.
parentElmConts	IN	512	The data to be written back at the parent location. This is basically the same element that was extracted from the location earlier, but with the parent location updated in the last 32 bits.
memElmRead	OUT	512	The details of the element stored at the requested location in memory

Table 5. 7 Continued

memAttrRead	OUT	512	The details of the attribute stored at the requested location in memory.

5.3.1.2 Detailed architecture and simulation results

The schema memory is implemented as an asynchronous read, synchronous write memory. The pre-processed schema is stored in the memory in the required proprietary format well before the start of simulation. The memory would have to be modified only if it is necessary to write back the location of the parent of a particular element that was processed. Figure 5.7 shows the operation of the module with the help of the simulation outputs obtained. As can be seen, new data gets populated on the 'memElmRead' and 'memAttrRead' data buses asynchronously with change in the 'nextElmPtr' and 'currAttrPtr' respectively.

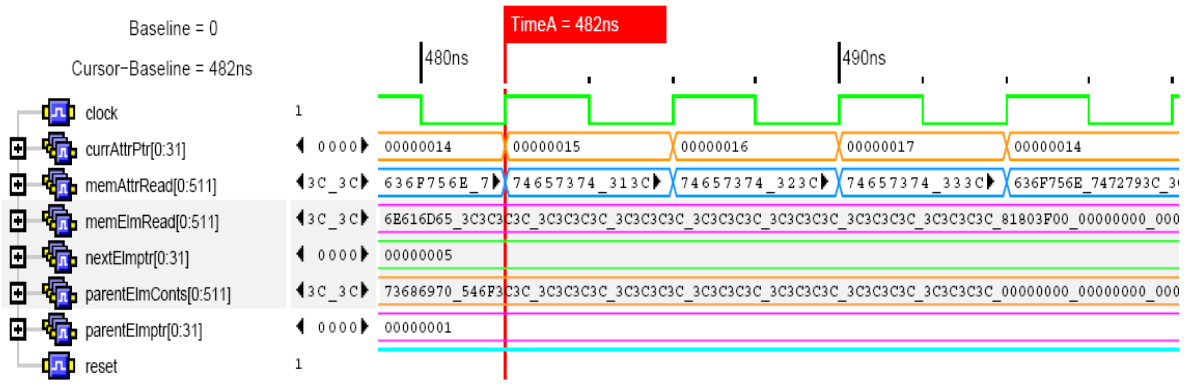

Figure 5. 7 Simulation output for the schema memory module

5.3.2 Schema element name validation

This module is responsible for validating that a particular element name is valid in the XML and that the element is valid at that particular location in the document.

5.3.2.1 Pin Interfaces

A summary of the important interfaces to this module are listed in Table 5.8.

Table 5. 8 Pin interfaces to the schema element name validation module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
en	IN	1	The signal from the WF stage requesting element name validation
elmName	IN	256	The element name extracted from the XML by the WF stage.
memElmRead	IN	512	The details of the element read from the schema memory.
elmNameValid	OUT	1	Signal to denote success or failure in validating the element name.
WE	OUT	1	Write enable signal used to request a write to the schema memory.
elmNameValErr	OUT	1	Error signal to denote validation error.
nextElmPtr	OUT	32	The pointer to the next element that should be read from the schema memory.
schemaElmConts	OUT	512	The details of the element read from memory are stored in this register.

5.3.2.2 Detailed architecture and Simulation Results

The implementation of this module can be better understood using the simulation output diagram shown in Figure 5.8, in addition to the pin interfaces explained in the previous section. The module becomes active on receiving the 'en' signal from the WF stage denoting that element name validation is requested. The 'elmName' input contains the element name that has been extracted from the document. This module also receives the element details from the schema memory in the 'memElmRead' input, whose first 32 bytes comprise the element name. The 'elmState' register denotes the state maintained by

the state machine in this module. The function of the state machine is to calculate the next location in the schema memory from where elements need to be read. This calculation is done based on the signals 'nestedElmPresent' and 'lastChildElm' which as their names suggest, specify different scenarios in which elements may exist in the document. Comparison of the element names is done only when the FSM in this module is in state 0. Else, it means that the next location has not yet been determined and hence, backpressure would be exerted on the previous stages. If in state 0, the first 32 bytes of the 'memElmRead' input is compared with the contents of the 'elmName' register to check for a match. If a mismatch is found, either the element name or its location in the document is invalid and the 'elmNameValErr' error signal is set.

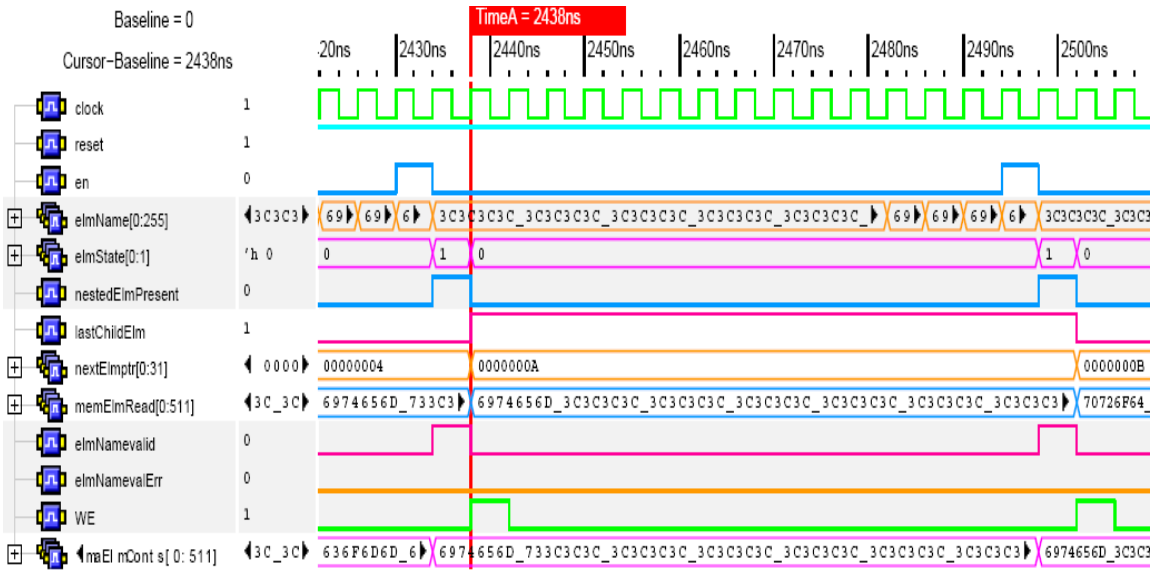

Figure 5. 8 Simulation output for the schema element name validation module

5.3.3 Schema element content validation

The function of this module is to check that the contents of the element under consideration conform to all the constraints imposed on them by the schema.

5.3.3.1 Pin Interfaces

The main interfaces to this module are shown in Table 5.9

Table 5. 9 Pin interfaces to the schema element content validation module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
Ec	IN	1	The signal from the WF stage requesting element content validation
elmValue	IN	512	The element contents extracted from the XML by the WF stage.
schemaElmConts	IN	512	The details of the element passed on by the Schema element content validation module.
elmValValid	OUT	1	Signal to denote success or failure in validating the element value.
elmValuevalErr	OUT	1	Error signal to denote validation error.

5.3.3.2 Detailed architecture and Simulation Results

The details in implementation of this module can be understood better by analyzing the simulation output obtained for this module shown in Figure 5.9. The 'ec' flag is raised by the well-formedness stage to indicate that the contents are now ready for schema validation. The extracted element content needs to be checked for all the constraints that are applicable for the same based on its data-type. As explained in Section 4.2, a detailed pre-defined memory structure tells this module where exactly to look for specific information required in the 512-bits wide 'schemaElmConts' that is received. After identifying the correct data-type the element belongs to, the other constraints are now checked one by one and applied on the contents of 'elmValue' input. As shown in the waveforms, the 'totdigits', 'maxLength', 'minInclusive' and 'fracDigits' are internal registers used to perform these validations. Corresponding error signals 'totDigitsErr', 'maxLengthErr', 'minInclusiveErr' and 'fracDigitsErr' have also been defined for these constraints. The snapshot shown below shows the identification of a valid decimal and a

valid integer type by the module represented by the 'validDec' and 'validInt' signals respectively. When the validation is successfully completed, the 'elmValValid' flag is set by the module. In case of any error found, the 'elmValuevalErr' flag is set.

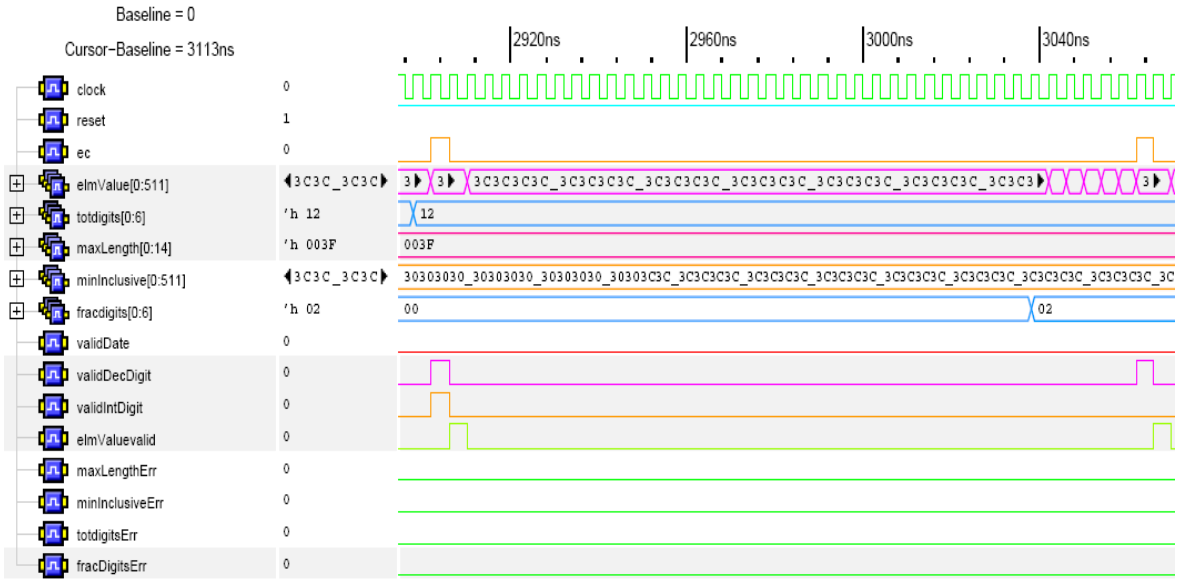

Figure 5. 9 Simulation output for the schema element content validation module

5.3.4 Schema attribute name validation

This module checks if a particular attribute is valid within the element under consideration in the XML. The details of implementation are presented in the following sub-sections.

5.3.4.1 Pin Interfaces

The important interfaces to this module are shown in Table 5.10

Table 5. 10 Pin interfaces to the schema attribute name validation module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
an	IN	1	The signal from the WF stage requesting attribute name validation

Table 5. 10 Continued

attrName	IN	256	The attribute name extracted from the XML by the WF stage.
memAttrRead	IN	512	The details of the attribute read from the schema memory.
attrNameValid	OUT	1	Signal to denote success or failure in validating the attribute name.
attrNameValErr	OUT	1	Error signal to denote validation error.
currAttrPtr	OUT	32	The pointer to the next attribute that is to be read from the schema memory.
schemaAttrConts	OUT	512	The details of the attribute read from memory are stored in this register.

5.3.4.2 Detailed architecture and Simulation Results

The implementation of this module can be better understood using the simulation output diagram shown in Figure 5.10 in addition to the pin interfaces explained in the previous section. The module becomes active on receiving the 'an' signal from the WF stage denoting that attribute name validation is requested. The 'attrName' input contains the attribute name that has been extracted from the document. This module also receives the attribute details from the schema memory in the 'memAttrRead' input, whose first 32 bytes comprise the attribute name. The starting and ending locations within which the attributes for the current element lie in the schema memory are known from the schema element name validation module. Using these locations as the limits, the value of the 'currAttrPtr' is calculated by the module using a sequential increment method. On each read from memory, the contents of 'attrName' are matched with the first 32 bytes of the 'memAttrRead' input. If there is a mismatch, the attribute location is incremented by 1 and the next location is read. This continues until the end location of valid attributes is reached. If the last attribute also returns a mismatch, it is confirmed that either the attribute is not valid or does not exist for the particular element under consideration. The

'attrNameValErr' flag is then set. On the other hand, if a match is found, the 'attrNameValid' flag is set by the module.

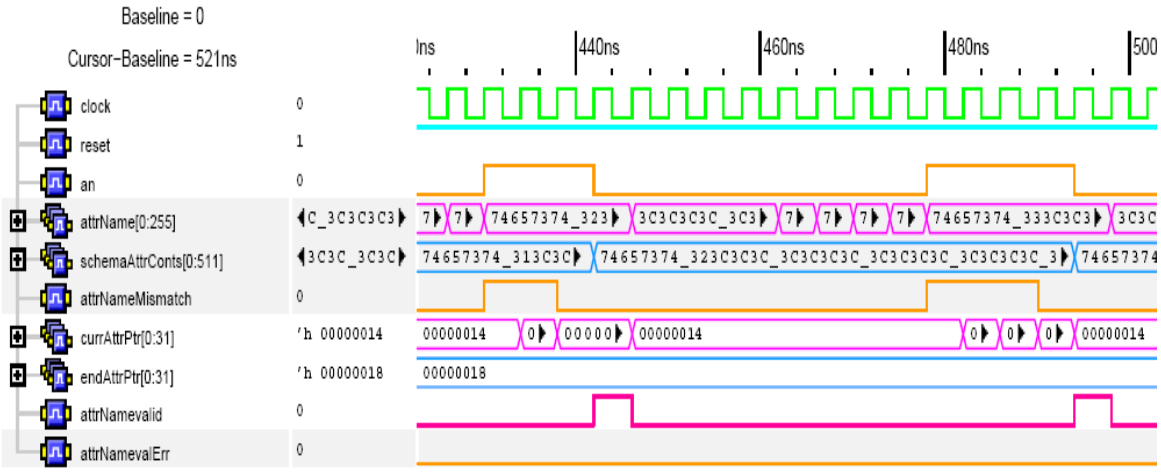

Figure 5. 10 Simulation output for the schema attribute name validation module

5.3.5 Schema attribute content validation

The function of this module is to check that the contents of the attribute under consideration conform to all the constraints imposed on them by the schema.

5.3.5.1 Pin Interfaces

The main interfaces to this module are shown in Table 5.11

Table 5. 11 Pin interfaces to the schema attribute content validation module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
ac	IN	1	The signal from the WF stage requesting attribute content validation
attrValue	IN	512	The attribute contents extracted from the XML by the WF stage.
schemaAttrConts	IN	512	The details of the attribute passed on by the Schema attribute name validation module.

Table 5. 11 Continued

attrValValid	OUT	1	Signal to denote success or failure in validating the attribute value.
attrValueValErr	OUT	1	Error signal to denote validation error.

5.3.5.2 Detailed architecture and Simulation Results

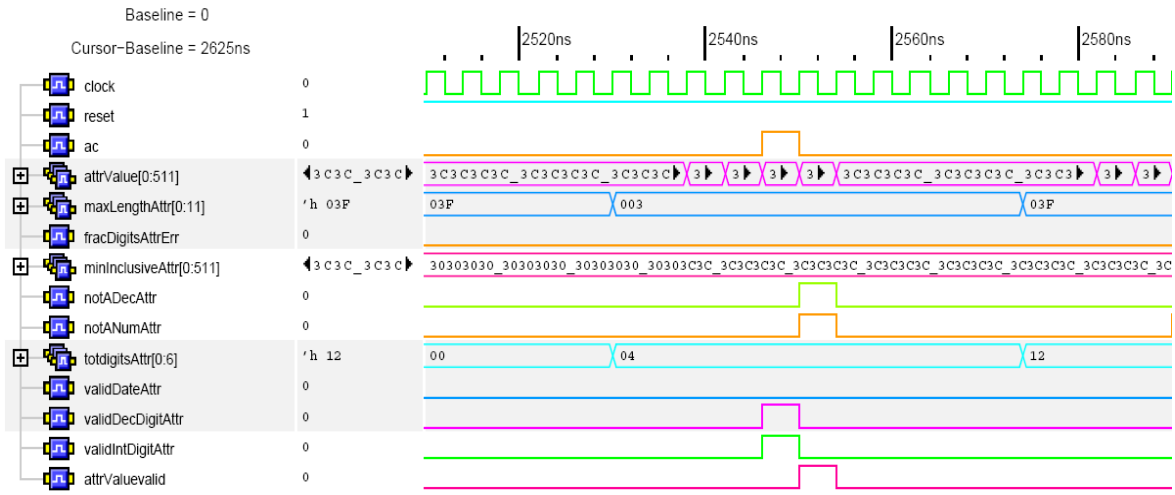

Figure 5. 11 Simulation output for the schema attribute content validaiton module

The details in implementation of this module can be understood better by analyzing the simulation output obtained for this module shown in Figure 5.11. The 'ac' flag is raised by the well-formedness stage to indicate that the contents are now ready for schema validation. The extracted attribute content needs to be checked for all the constraints that are applicable for the same based on its data-type. As explained in Section 4.2, a detailed pre-defined memory structure is used by this module to identify where exactly to look for specific information required in the 512-bits wide 'schemaAttrConts' value that is received. . After identifying the correct data-type for the attribute, the other constraints are now checked one by one and applied on the contents of 'attrValue' input. As shown in the waveforms, the 'totdigitsAttr', 'maxLengthAttr', 'minInclusiveAttr' and 'fracDigitsAttr' are internal registers used to perform these validations. Corresponding

error signals 'totDigitsAttrErr', 'maxLengthAttrErr', 'minInclusiveAttrErr' and 'fracDigitsAttrErr' have also been defined. The snapshot shown in the figure shows the identification of a valid decimal and a valid integer type represented by the 'validDecAttr' and 'validIntAttr' signals respectively. When the validation is successfully completed, the 'attrValValid' flag is set by the module. In case any error is found, the 'attrValuevalErr' flag is set.

5.3.6 Schema controller

The primary function of this module is to generate the back pressure signal and interface with the controller of the first stage.

5.3.6.1 Pin Interfaces

The important interfaces in this module are listed in Table 5.12.

Table 5. 12 Pin interfaces to the schema controller module

Pin Name	Direction	Width	Description
Clock	IN	1	System clock
Reset	IN	1	Asynchronous system reset
an	IN	1	The signal from the WF stage requesting attribute name validation
en	IN	1	The signal from the WF stage requesting element name validation
attrNameMismatch	IN	1	Signal from the attribute name validation module denoting that a mismatch has occurred and hence, multiple memory reads may be required to complete validation
elmState	IN	3	The current state of the FSM in the element name validation module.
attrNameValErr	IN	1	Error signal to denote validation error for the attribute name.
elmNamevalErr	IN	1	Error signal to denote validation error for the element name.
attrValuevalErr	IN	1	Error signal to denote validation error for the attribute value.

Table 5. 12 Continued

elmValuevalErr	IN	1	Error signal to denote validation error for the element contents.
bp	OUT	1	Back pressure signal
schemaErr	OUT	1	Denotes a schema validation error.

5.3.6.2 Detailed architecture and simulation results

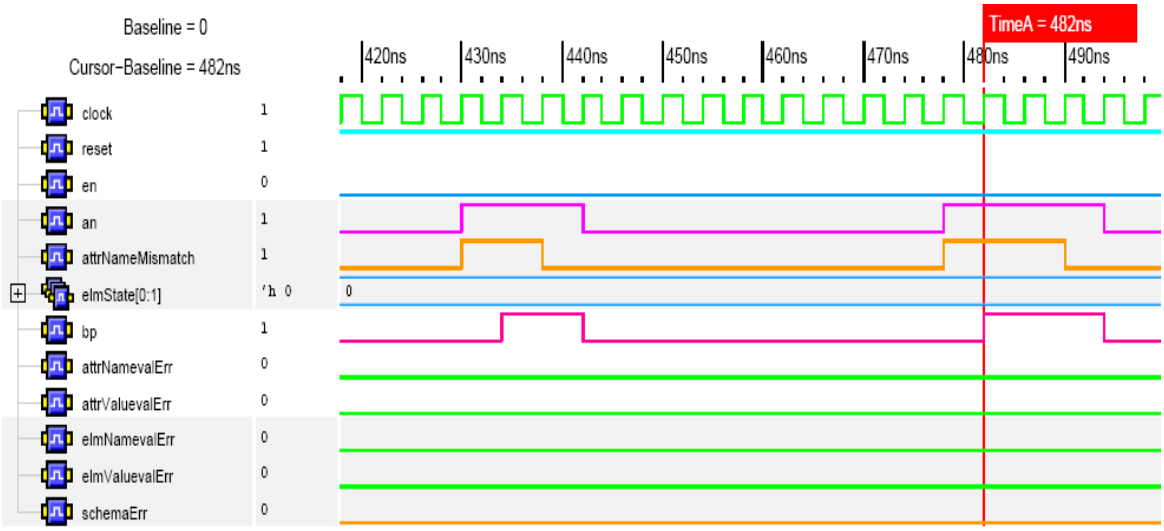

Figure 5. 12 Simulation output for the schema controller module

Backpressure is generated by the schema validator in one of the following two cases:

1. When element name validation has been requested by the WF stage, but the SV stage is still in the process of calculating the next element location. In this case, 'elmState' would be greater than 0.
2. When attribute name validation has been requested and a mismatch has occurred. In this case, multiple iterations are required to complete processing. 'attrNameMismatch' flag is set in this condition.

It is the function of the schema controller to detect the occurrence of either of these conditions using the input signals mentioned above and generate the 'bp' or backpressure

signal. The generation of this signal would stall the entire pipeline for as many clock cycles as the schema validation stage takes to come out of either of the above two conditions. This module also generates the 'schemaErr' error signal on finding that any of the internal modules has reported an error. In other words, 'schemaErr' is set if any of 'attrNameValErr', 'elmNamevalErr', 'attrValuevalErr' or 'elmValuevalErr' is found to be set. These signals are shown in the simulation output shown in Figure 5.12.

CHAPTER 6

VERIFICATION AND RESULTS

This chapter first describes the verification strategy adopted, the test cases covered and their results. The verification of the design by synthesis and back-annotation of the Verilog net list is then explained. Finally, the performance and results of the system are presented.

6.1Verification Strategy

Functional verification is an integral part of ASIC development and there are two main strategies that can be adopted. The first one is to supply directed test cases to the design and manually verify the functionality by simulation and checking the generated waveforms. The second strategy is more extensive in its approach and involves the creation of a golden model of the design which takes the same inputs as the design and generates the expected output. The results of this model are then compared with the actual outputs obtained. This method provides scope for randomization of inputs and as much automation of the process as possible. However, the first strategy can deliver good results too if the test cases run are extensive and exercise just the right conditions.

The former method, namely directed testing using simulation is used on this design. This decision was largely due to the fact that formal verification methods would not be very effective for the nature of this design. For instance, the schema stored in the schema memory varies with each XML document and hence, has to be changed and loaded into the memory before the simulation starts for a new document, eliminating the possibility of continuous random testing. The idea therefore, was to employ a detailed, step-by-step, manual verification methodology. The functioning at each stage of

implementation was checked by verifying the operation of each module as soon as coding was completed for the same. Some error-free XMLs of different complexity and structure were then run on the completed design to find any errors in coding or implementation. After this, a comprehensive list of test cases was identified and sample XMLs which exercised all these constraints and cases were run on the design. The input XMLs had known errors in them and so, the waveforms could be observed to check for an error at the exact location where it was inserted. Appropriately named error signals have been defined for every validation check that was covered. This made deciphering the occurrence of an error from the waveforms a simple task. The last set of test cases was to exercise the boundary conditions in the design.

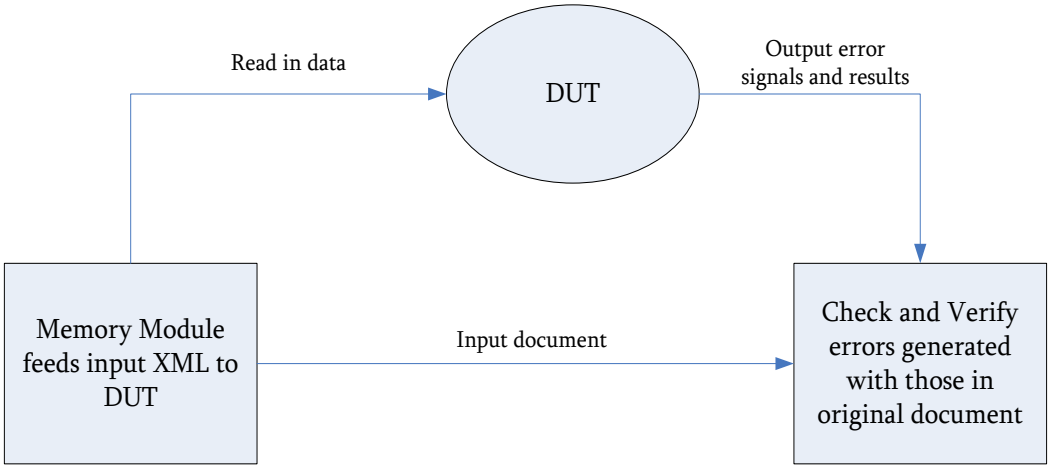

Figure 6. 1 High-level block diagram of the testbench architecture

The basic test bench architecture is shown in Figure 6.1. As can be observed, the Device Under Test (DUT) reads in the XML data from memory. The data is then validated by both stages of the pipeline and the error signals are sent out. The errors generated are checked with those in the original document and verified for correctness. The simulation output for the test bench is shown in Figure 6.2 and this helps to present a better perspective about the process taking place. The error signals from the DUT are labeled 'schemaErr' and 'wfcErr' corresponding to the two stages in the pipeline. In the

scenario shown, the schemaErr is asserted. The 'stopExec' signal is then set to signal the halting of further read of data and disqualify the document as being invalid.

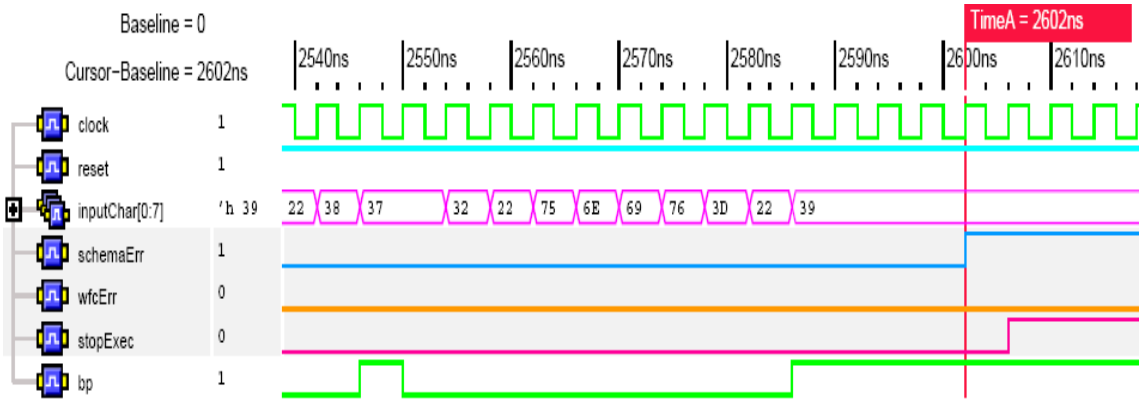

Figure 6. 2 Simulation result showing the error condition and execution halt

6.2 Test Cases Executed

This section describes the test cases that have been executed on the design and the results for the same. The test cases have been described module-wise for better clarity.

6.2.1 Element Contents Validation

The test cases executed for the element contents validation are listed in Table 6.1.

Table 6. 1 Test cases for element contents validation

Test-case Id	Test-case description (includes both positive and negative cases where applicable)	Result
1	Element value is a valid integer	Y
2	Element value is a valid date	Y
3	Element value is a valid decimal	Y
4	Element value is a valid string	Y
4	Calculation of number of decimal points in a decimal data-type	Y
5	maxLength constraint satisfied for decimal data-type	Y
6	maxLength constraint satisfied for integer data-type	Y
7	maxLength constraint satisfied for string data-type	Y
8	maxLength constraint satisfied for date data-type	Y
9	minInclusive constraint satisfied for decimal data-type	Y

Table 6. 1 Continued

10	minInclusive constraint satisfied for integer data-type	Y
11	totdigits constraint exceeded for integer data-type	Y

6.2.2 Attribute Contents Validation

The test cases executed for the attribute contents validation are listed in Table 6.2.

Table 6. 2 Test cases for attribute contents validation

Test-case Id	Test-case description (includes both positive and negative cases where applicable)	Result
1	Attribute value is a valid integer	Y
2	Attribute value is a valid date	Y
3	Attribute value is a valid decimal	Y
4	Calculation of number of decimal points in a decimal data-type	Y
5	maxLength constraint satisfied for decimal data-type	Y
6	maxLength constraint satisfied for integer data-type	Y
7	maxLength constraint satisfied for string data-type	Y
8	maxLength constraint satisfied for date data-type	Y
9	minInclusive constraint satisfied for decimal data-type	Y
10	minInclusive constraint satisfied for integer data-type	Y
11	totdigits constraint exceeded for integer data-type	Y

6.2.3 Attribute Name Validation

The test cases executed for the attribute name validation are listed in Table 6.3.

Table 6. 3 Test cases for attribute name validation

Test-case Id	Test-case description (includes both positive and negative cases where applicable)	Result
1	Attribute name mismatch	Y
2	Attribute name starts with '.' Or '-'	Y
3	Attribute name does not contain invalid characters	Y
4	Attribute name starts with numbers	Y

6.2.4 Element Name Validation

The test cases executed for the element name validation are listed in Table 6.4.

Table 6. 4 Test cases for element name validation

Test-case Id	Test-case description (includes both positive and negative cases where applicable)	Result
1	Element out of sequence	Y
2	Element occurs more than once	Y
3	Element name mismatch	Y
4	Element name starts with 'xml' in different combination of case	Y
5	Element name starts with '.' Or '-'	Y
6	Element name contains invalid characters	Y
7	Element name starts with numbers	Y

6.2.5 Closing Element Name Validation

The test cases executed for the element name validation are listed in Table 6.5.

Table 6. 5 Test cases for closing element name validation

Test-case Id	Test-case description (includes both positive and negative cases where applicable)	Result
1	Closing Element out of sequence	Y
2	Closing Element occurs more than once	Y
3	Closing Element name mismatch	Y
4	Closing Element name starts with 'xml' in different combination of case	Y
5	Closing Element name starts with '.' Or '-'	Y
6	Closing Element name contains invalid characters	Y
7	Closing Element name starts with numbers	Y

6.2.6 Well-formedness validations

The test cases executed for the checking well-formedness of the document are listed in Table 6.6.

Table 6. 6 Test cases for well-formedness validation

Test-case Id	Test-case description (includes both positive and negative cases where applicable)	Result
1	Attributes must be quoted properly	Y
2	Root element must be unique within the document	Y
3	Attributes must be unique within an element	Y
4	Elements must be nested correctly	Y
5	Elements must be closed properly	Y
6	XML declaration must be present and should adhere to correct syntax.	Y
7	Entity characters are used correctly	Y

6.2.7 Boundary conditions validation

The test cases executed for the checking the boundary conditions of the design are listed in Table 6.7.

Table 6. 7 Test cases for boundary conditions

Test-case Id	Test-case description (includes both positive and negative cases where applicable)	Result
1	Element content more than 64 characters	Y
2	Element name more than 32 characters	Y
3	Attribute name more than 32 characters	Y
4	Attribute content more than 64 characters	Y
5	Closing element name more than 32 characters	Y
6	XML document with up to 16 levels of hierarchy	Y
7	XML element with up to 8 different attributes	Y
8	XML memory is filled after 8 cycles of backpressure	Y

6.3 Logic synthesis and results

Synthesis is used to describe the process of reducing the RTL code to an optimized gate-level representation [5]. Synthesizing is an iterative process and begins with defining timing constraints for each block of the design. The synthesis tool reads the RTL code of the design and using the timing constraints, synthesizes the code to the structural level,

thereby producing a mapped gate-level net list. Synthesis usually exposes errors that are not found by simulation and these may be due to simple coding omissions such as initialization of registers or unintentional timing arcs. It may also be found that the critical path in the design includes a lot of logic, in which case it would be advisable to try to simplify it for better performance. Different tools and standard cell libraries are available for performing synthesis.

Synopsys Design Vision and the OSU 180nm Standard Cell library were used in synthesizing this design. The constraints for synthesis were set to maximize timing optimization and relax other constraints such as area. After the design was synthesized without errors, the Verilog net list generated was used for re-simulation using the original inputs to cross-verify that the generated hardware does indeed implement the originally intended functionality. After incremental compile, the design runs at a maximum frequency of 151Mhz. This translates to a throughput of 1.2Gbps for the best case input document. Analysis of important performance parameters such as area, power and timing were also carried out. The results for the same are presented in in Table 6.8.

Table 6. 8 Synthesis results

Operating Conditions Library Temperature Voltage	 osu018_stdcells_slow 125.00 1.62
Area	8295281um^2
Power (memory module excluded from analysis)	11.1325mW
Clock Period	6.6ns
Throughput	1.212Gbps

CHAPTER 7

FUTURE WORK AND CONCLUSION

This chapter presents the scope for carrying on this work and concludes the thesis.

7.1 Future Work

This section describes the improvisations and future work that could be carried out on the design. The suggestions are organized in the following sub-sections.

7.1.1 Feature Additions

A detailed analysis of the requirements was carried out in Chapter 3.1 and a list of features for implementation was finally selected. This leaves scope for supporting many additional features for completeness of the parser. The desirable additions would be the following:

1. Support for all features and constraints possible in Schema:
    a. Implementation of all data-types that are defined in the Schema recommendation, with validation of constraints applicable to each of them. This is desirable so that the parser could be more generic in nature and take in different schemas irrespective of the data-types they contain. Since a combination of the most complex and commonly used data-types has been selected for the current implementation, the performance would not be majorly affected by supporting more of these features.
    b. Another desirable feature would be the support for the 'minOccurs' and 'maxOccurs' which specify the minimum and maximum number of times respectively that an element can occur within a particular enclosing

element. These features are commonly used in documents and the reasons for not implementing the same are as follows:

- These cannot be counted as one of the principal functions of the schema validation stage.
- Checking for the maxOccurs results in undue wastage of clock cycles and hence of throughput.
- Checking for maxOccurs also increases the complexity of the design in terms of the number of possibilities to be considered, and the number of values to be remembered and compared in the stack.

2. Increase the number of levels of nesting possible in the element and the number of attributes possible within an element.

   In the current implementation, a maximum of 16 levels of nesting are supported and a maximum of 8 attributes are possible within an element. These numbers were chosen to be logically large enough to test the robustness of the design. However, for use in practical applications, increasing these numbers would be desirable so as to claim support for XML documents with a very high degree of complexity. We could also look at relaxing the constraints on the length of element and attribute names and content. In this implementation, names are restricted to a length of 32 bytes, and content length to 64 bytes.

7.1.2 Implementation and optimizations

1. FPGA Implementation – An FPGA implementation of the design could be attempted to check the compatibility of the design on existing FPGA platforms.
2. Area and Power Optimizations – The schema memory is at present implemented as a huge on-chip array of registers which take up significant area and power. Off-chip implementation was ruled out considering the huge size of the memory and the number of interfaces and memories that would be required if implemented

separately. Moreover, area and power optimizations were not the focus of this design and there could be scope for improvements on the same.

3. The "Average throughput" of the system for the best, common and worst-case scenarios can be determined only by carrying out a huge number of test runs with XMLs of all sizes and complexity. Only as much testing has been carried out on the design as was required to verify the design thoroughly and ensure its robustness.

7.2 Conclusion

The XML explosion in businesses today and its rapid adoption for different networking applications justifies any and all effort spent in trying to come up with XML parsing solutions with increased efficiency and innovation. XML parsing continues to suffer from the performance bottleneck when implemented in software and the thesis basically evaluates the practicality of offloading the validation parts of the process onto hardware. Though it is premature to quote an "average throughput" figure for the design, a peak throughput of 1.2 Gbps has been obtained for the tests conducted. The robustness of the design has also been tested by thorough functional verification. The current implementation proposes a purely hardware implementation for the well-formedness checker and the use of a non-real-time software pre-processor to help tackle the complexity involved in the schema validation. It is thus inferred that an efficient mixed-parsing solution may well be the answer to the problems in XML parsing today.

REFERENCES

[1] "XML: The complete reference", by Heather Williamson. *Published 2001, Osborne/McGraw-Hill*

[2] "Mastering XML" by Ann Navarro, Chuck White, Linda Burman. *Published 1999, Sybex*

[3] "The XML Schema complete reference" by Cliff Binstock, Dave Peterson, Mitchell Smith, Mike Wooding, Chris Dix, Chris Galtenberg. *Published 2002, Addison-Wesley.*

[4] "Fast SOA" by Frank Cohen. *Published 2006, Morgan Kaufmann.*

[5] "Logic synthesis using Synopsys" by Pran Kurup, Taher Abbasi. *Published 1997, Springer.*

[6] "Web Services: European Conference, ECOWS 2004, Erfurt, Germany, September 27-30, 2004, Proceedings (Lecture Notes in Computer Science)" by Liang-Jie Zhang. *Published 2004, Springer-Verlag New York, Inc*

[7] "Boosting the SOA with XML Networking" by Silvano Da Ros. *The Internet Protocol Journal - Volume 9, Number 4.*

[8] "Implementation of network application layer parser for multiple TCP/IP flows in reconfigurable devices" by James Moscola, Young H. Cho, John W. Lockwood, *in Proceedings of the 16th International Conference on Field Programmable Logic and Applications. (FPL), Madrid, Spain, Aug 2006, pages 1-4.*

[9] "Xml accelerator engine" by Jan van Lunteren, Ton Engbersen, Joe Bostian, Bill Carey, Chris Larsson, *in The First International Workshop on High Performance XML Processing, 2004, pages 1-6.*

[10] "Reconfigurable Context-Free Grammar based Data Processing Hardware with Error Recovery", by James Moscola, Young H. Cho, John W. Lockwood, *in Proceedings of*

International Parallel & Distributed Processing Symposium (IPDPS/RAW), Rhodes Island, Greece, 2006, pages 1-4.

[11] "A compiler-based approach to schema-specific XML parsing" by Kenneth Chiu and Wei Lu, *in Proc. Workshop on High Performance XML Processing, May 18, New-York,, USA, 2004, pages 1-3.*

[12] "Mesh-based content routing using XML" by Alex C. Snoeren, Kenneth Conley, and David K. Gifford, *in Proceedings of the eighteenth ACM symposium on Operating systems principles, Banff, Canada, Dec. 2001, pages 160-173.*

[13] "XML screamer: an integrated approach to high performance XML parsing, validation and deserialization" by Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, *in WWW'06, 15th International Conference on World Wide Web, Proceedings, pages 93–102. ACM Press, 2006.*

[14] "An evaluation of binary xml encoding optimizations for fast stream based xml processing" by Roberto J. Bayardo, Daniel Gruhl, Vanja Josifovski, Jussi Myllymaki, *in Proceedings of WWW/2004. pages 17–26.*

[15] http://www.w3schools.com/xml/

[16] http://www.w3.org/XML/

[17] http://www.xml.com/pub/a/2007/05/09/xml-parser-benchmarks-part-1.html

[18] http://www-306.ibm.com/software/integration/datapower/

[19] http://xml.com/pub/a/98/10/guide0.html?page=2#AEN84

[20] http://webdesign.about.com/od/xml/a/aa060401a.htm

[21] http://www.eetimes.com/editorial/1998/coverstory9802.html

[22] http://www.idealliance.org/proceedings/xml04/papers/299/RandomAccessXML.html

[23] http://www.itwriting.com/xmlintro.php