ABSTRACT

KUTTAPPA, SANCHITH. XPath Hardware Accelerator. (Under the direction of Dr. Yannis Viniotis).

eXtensible Markup Language (XML) and XML Path Language (XPath) are increasing in importance significantly. XML has become the language for structuring, storing and sending information between diverse sources, thus, becoming the language of choice for data exchange and data storage. XML is independent of software, hardware and applications, proving to be extremely versatile and flexible. XML and XPath implement a non-trivial grammar. Parsing this grammar proves to be a high overhead for the CPU. This thesis presents hardware offload architecture; the XPath Offload Engine (XPOE) which essentially offloads the XPath string functions along with encoding-to and decoding-from UTF-8, which burn up a large number of CPU cycles. An analysis has been done which shows a reduction in the CPU overhead by as much as 88%.

XPATH HARDWARE ACCELERATOR

by
SANCHITH KUTTAPPA

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

2007

Approved By:

Dr. Gregory Byrd

Dr. Mihail Sichitiu

Dr. Yannis Viniotis
Chair Advisory Committee

BIOGRAPHY

Sanchith Kuttappa was born in Bangalore, India in July 1980. She graduated from Visveswaraiah Technological University, India, with a Bachelors degree in Electronics and Telecommunications Engineering in June 2002. After undergraduate studies, she worked with Infosys Technologies Ltd., Bangalore, for two years. In August 2004, she joined North Carolina State University as a graduate student in the Computer Engineering program. In Summer and Fall of 2005, she interned at Qualcomm Inc., RTP in the Processor Design Group. In Summer of 2006, she interned at Ericsson IPI, Raleigh in the Dataplane Group. While working towards the Masters degree, she worked on her thesis under the guidance of Dr. Yannis Viniotis.

ACKNOWLEDGEMENTS

I sincerely thank my advisor, Dr.Yannis Viniotis for his invaluable guidance and support through my graduate studies. I am grateful to him for providing me with the opportunity and encouraging me in pursuing my interests. I thank him for his constant support, encouragement and patience.

I am very grateful to Dr. Gregory Byrd, Dr. Paul Franzon and Dr. Mihail Sichitiu for agreeing to be on my thesis committee and for the valuable feedback regarding the thesis document.

I'd like to mention my family, who were always encouraging and supportive at all times. I would also like to mention Nitesh and all my friends, without whose support I would never have been able to succeed in my endeavors.

Table of Contents

Page

List of Figures vii

List of Tables...viii

1 Introduction and Literature Review..1

1.1 Importance of XML/XPath…………………………………… 1
1.1.1 What is XML?... 1
1.1.2 Strengths of XML……………………………………… 2
1.1.3 Real world examples of XML…………………………… 3
1.1.4 Importance of XPath……………………………….. 4
1.2 Current Implementations of XML/XPath..5
1.2.1 Architecture of an XML Processor... 5
1.2.2 Architecture of XPath 7
1.2.3 Examples of XPath Implementation 9
1.3 Maximizing performance via hardware offload.......................................12
1.3.1 Issues with software processing..12
1.3.2 Current approaches to challenges faced....................................13
1.3.3 Industry examples 14
1.3.4 XPath Offload Engine...15
1.4 Thesis organization...16

2 XPath...17

2.1 Introduction – XPath/XQuery 17
2.1.1 XPath Design Aspects 18
2.2 XPath Grammar and its complexity..18
2.2.1 Lexical Analysis...19
2.3 XPath Expressions and Functions 20
2.3.1 XPath Expression and Function format..20
2.3.2 XPath Function types 22

2.3.3 XPath String Functions 23
2.4 Requirements for parsing XPath expressions/functions 26

3 Processor overhead savings analysis 27

3.1 Software approach to XPath function implementation 27
3.2 High level block diagram for hardware offload 30
3.3 Processor savings analysis 33
3.3.1 Savings analysis for UTF-8 Encoding and Decoding 33
3.3.2 Savings analysis for each of the string functions 37

4 Design Architecture 53

4.1 Introduction 53
4.2 Block diagram and explanation 54
4.2.1 Read Input Memory 55
4.2.2 Read Main Memory 56
4.2.3 Function Execute 57
4.2.4 Output Memory Write 59
4.3 Design Pipeline 60

5 Design Module Description 62

5.1 Introduction 62
5.2 Module Read Input Memory 62
5.2.1 Pin interface 63
5.2.2 Architecture 64
5.3 Module Read Main Memory 65
5.3.1 Pin interface 65
5.3.2 Architecture 67
5.4 Module Function Execute 69
5.4.1 Pin interface 69
5.4.2 Architecture 71
5.5 Module Output Memory Write 72
5.5.1 Pin interface 72
5.5.2 Architecture 74
5.6 Controller 75

6 Verification and Additional Design Details 78

6.1 Test environment description 78
6.2 Test architecture 80

6.2.1 Module Packet Generator 80
6.2.2 Module Reader 81
6.3 Verification Test Plan 83
6.3.1 Feature tests 84
6.4 Test results summary 84
6.5 Additional Design Details 85

7 Future Work and Conclusion 86

7.1 Feature Additions 86
7.2 Verification 87
7.3 FPGA implementation and Architecture optimizations 88
7.3.1 Area optimizations 88
7.3.2 Timing optimizations 88
7.4 Conclusion 89

Bibliography 90

List of Figures

Page

1.1 Generic Software Stack for XML Processing 5
1.2 Schematic of a typical XML Parser 7

3.1 High level block diagram showing system level interaction 31
3.2 Plot of fraction of XPath functions offloaded onto hardware to the savings achieved 51
3.3 Plot of fraction of XPath functions offloaded onto hardware to the speedup achieved 52

4.1 Block Diagram of XPOE 54
4.2 Read Input Memory Implementation Diagram 55
4.3 Read Main Memory Implementation Diagram 57
4.4 Function Execute Implementation Diagram 58
4.5 Memory Write Implementation Diagram 60

5.1 Interface waveform for Input Memory 64
5.2 Internal counters and Read Main Memory interface timing 65
5.3 Timing relation with Main Memory 68
5.4 Timing relation with Read Input Memory Interface 68
5.5 Internal counters and Function Execute Interface timing 69
5.6 Timing relation with Read Main Memory Interface 71
5.7 Internal counters and Memory Write interface timing 72
5.8 Timing relation with Function Execute Interface 74
5.9 Internal counters and output interface 75
5.10 Flowchart for XPOE 76
5.11 State Machine Diagram 77

6.1 Block diagram of testbench 79
6.2 Architectural Block Diagram of the Reader Module 82

List of Tables

Page

3.1 Format of the different octet types in UTF-8 encoding 28
3.2 Cycles required for an all-software approach for encoding function........... 34
3.3 Cycles required for an all-software approach for decoding function........... 35
3.4 Cycles required for a hardware-offloaded approach 36
3.5 Cycles required for an all-software approach: codepoints-to-string 38
3.6 Cycles required for a hardware-offloaded approach: codepoints-to-string . 39
3.7 Cycles required for an all-software approach: string-to-codepoints 40
3.8 Cycles required for a hardware-offloaded approach: string-to-codepoints . 41
3.9 Cycles required for an all-software approach: codepoint-equal 42
3.10 Cycles required for a hardware-offloaded approach: codepoint-equal 43
3.11 Cycles required for an all-software approach: substring 44
3.12 Cycles required for a hardware-offloaded approach: substring.................. 45
3.13 Cycles required for an all-software approach: starts-with and ends-with ... 46
3.14 Cycles required for a hardware-offloaded approach: starts-with and ends-With 47
3.15 Cycles required for an all-software approach: substring-before and substring-after 48
3.16 Cycles required for a hardware-offloaded approach: substring-before and substring-after 49
3.17 Comparison of the savings achieved for each of the functions 49
3.18 Comparison of fraction of string functions to the savings achieved............ 50
3.19 Comparison of fraction of string functions to the speedup achieved 51

5.1 Interface with the system 63
5.2 Interface with the input section of the memory... 63
5.3 Interface with the Read Main Memory module... 63
5.4 Interface with the system 66
5.5 Interface with the Read Input Memory module... 66
5.6 Interface with the Main Memory module 66
5.7 Interface with the Function Execute module 67
5.8 Interface with the system 69
5.9 Interface with the Read Main Memory module... 70

5.10 Interface with the Memory Write module 70
5.11 Interface with the system 72
5.12 Interface with the Function Execute module 73
5.13 Output Interface of the Memory Write module 73

6.1 Interface with the system 80
6.2 Interface with the DUT 80
6.3 Interface with the system 81
6.4 Interface with the DUT 81
6.5 Feature tests 84
6.6 Summary table of the test results.. 84

Chapter 1

Introduction and Literature Review

This chapter provides the motivation and the main concepts behind XML, XPath, XQuery and hardware offloading.

1.1 Importance of XML/XPath

The following sections will talk about XML, XPath and their growing versatility and importance in the real world.

1.1.1 What is XML?

XML (eXtensible Markup Language) is a markup language much like HTML. However, XML was not designed to do anything it was only created to structure, store and to send information. XML is designed to describe data and to focus on what data is. According to the W3C Recommendation [1], XML describes a class of data objects called XML documents and partially describes the behavior of

computer programs which process them. XML documents are made up of storage units which contain either parsed or unparsed data, and these units are called entities. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure.

XML documents use a self-describing and simple syntax, like the example below which is a note to John from Sarah, stored as XML:

```
<note>
<to>John</to>
<from>Sarah</from>
<heading>Reminder</heading>
<body>Don’t forget, meeting’s changed to 9:00am!</body>
</note>
```

The note has a header and a message body. It also has sender and receiver information. But still, this XML document does not DO anything. It is just pure information wrapped in XML tags. Someone must write a piece of software to send, receive or display it [1]. XML tags are not predefined; the author defines his own tags and his own document structure. The basic syntax for one element in XML is: <name attribute="value">content</name>

1.1.2 Strengths of XML

The following points emphasize the strengths of XML and show why XML is going to be everywhere [1].

1. One of the most time-consuming challenges for developers has been to exchange data between computer systems and databases that contain data in incompatible formats over the Internet. Converting the data to XML can greatly reduce this complexity and create data that can be read by many different types of applications.

2. XML is going to be the main language for exchanging financial information between businesses over the Internet. A lot of interesting B2B applications are under development.
3. XML provides a software- and hardware-independent way of sharing data, as XML data is stored in plain text format. It also makes it easier to expand or upgrade a system to new operating systems, servers, applications, and new browsers.
4. XML can also be used to store data in files or in databases. Applications can be written to store and retrieve information from the store, and generic applications can be used to display the data.
5. Since XML is independent of hardware, software and application, data can be made available to other than only standard HTML browsers. Data can be made available to all kinds of "reading machines" (agents), and it is easier to make your data available for blind people, or people with other disabilities.
6. XML can be used to create new languages. The Wireless Markup Language (WML), used to markup Internet applications for handheld devices like mobile phones, is written in XML.

1.1.3 Real world examples of XML

The following paragraphs talk about XML's versatility and flexibility in the real world. There are a large number of applications that use XML and the rapid increase in these numbers is shown by examples from the IT/Software industry and the Telecom industry.

In the IT industry we see that the processing of XML documents has entered the mainstream of software application development. System and software configuration files, UI specifications and many office documents (such as spreadsheets, presentations, etc.) are being specified as XML. This is in addition to the exchange of XML messages in web-service oriented applications that are

used for enterprise application integration and financial interactions such as B2B transactions [4]. Hence, as XML is being used more and more as a common tool for all data manipulation and data transmission, XML will be extremely important to the future of the Web.

The increase in the number of applications based on XML messages and documents and the increased importance of XML processing is felt most significantly in the telecommunications industry with its ever expanding communications demand. XML is a powerful tool for describing service definition, activation, provisioning, and billing in next-generation communications infrastructure. It is also the technology of choice for related data interchange activities. Some of the applications and areas that XML is being used in are VoIP (Voice over IP) and its associated protocol SIP (Session Initiation Protocol), NOTIFY- SUBSCRIBE applications, videoconferencing and network games.

1.1.4 Importance of XPath

As increasing amounts of information are stored, exchanged, and presented using XML, the ability to intelligently query XML data sources becomes increasingly important. One of the great strengths of XML is its flexibility in representing many different kinds of information from diverse sources. To exploit this flexibility, an XML query language must provide features for retrieving and interpreting information from these diverse sources [2].

XPath 2.0/XQuery 1.0 (XML Path Language/XML Query Language) are designed to be flexible enough to query a broad spectrum of XML information sources, including both databases and documents. XPath makes it possible to refer to individual parts of an XML document and provides random access to XML data for other technologies. XPath 2.0/Xquery 1.0 operates on the abstract, logical structure (data model) of an XML document, rather than its surface syntax. XPath 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. The XPath language is based on a tree

representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by a variety of criteria [1].

The utility of XPath suggests that its role in the repertoire of XML processing may well expand greatly in the future. Therefore we see that as XPath is fundamental to a lot of advanced XML usage, for example, sophisticated XSLT transformations, the importance of XPath is linked to the importance of XML.

1.2 Current Implementations of XML/XPath

This section elaborates on the essential components that make up XML and XPath and how they are processed.

1.2.1 Architecture of an XML Processor

The W3C Recommendation defines an XML Processor as a software module which reads XML documents and provides access to their content and structure. This processing work is done on behalf of another module called the application [1]. Figure 1.1 shows the software stack for XML processing in detail [4].

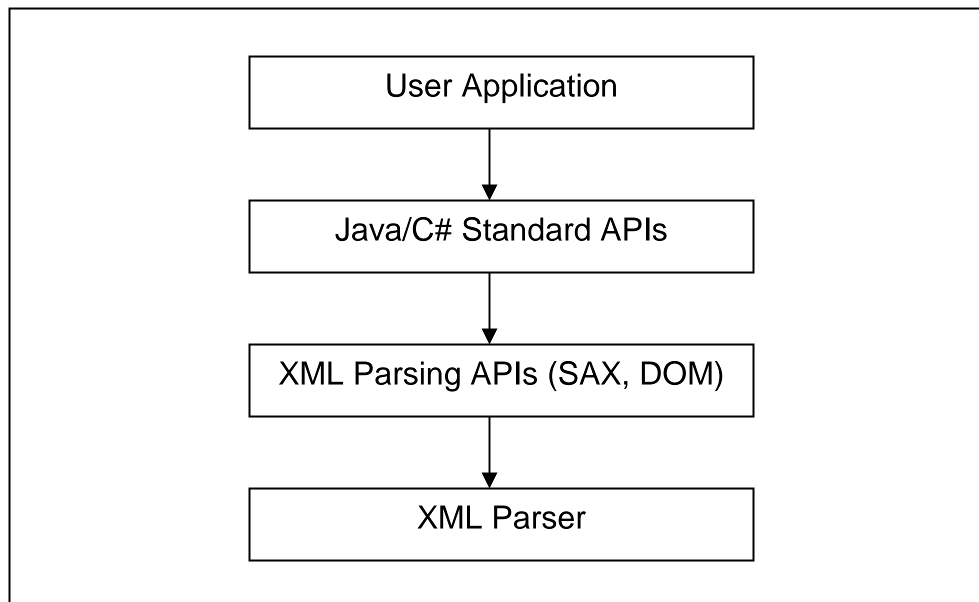

Figure 1.1: Generic Software Stack for XML Processing

To manipulate an XML document, you need an XML parser. The parser loads the document into your computer's memory. Once the document is loaded, its data can be manipulated by treating the XML document as a tree. The following points describe in detail the function of the different modules of the parser and thus show why XML parsing forms the most CPU intensive section of the software stack. Figure 1.2 depicts the architecture of a typical XML Parser. Its architecture is composed of five individual modules [4].

1. **Decoder:** This module reads in UTF-8 or UTF-16 (UTF representation is defined in section 3.1) characters from the input stream and converts them to the character format of the host programming language. In addition, this module makes sure that the inputs characters are part of a valid XML character set.
2. **Scanner:** This classifies the characters in the XML input stream to locate (namespace qualified) element and attribute names, element content, attribute values and the like. In essence this implements the lexical analyzer corresponding to the XML grammar that searches for characters such as <, >, =, :, etc., to tokenize the XML document.
3. **Parser:** This is the module that checks for well-formedness of an XML document. Well-formedness requires that the begin- and end-element tags are properly nested and matched. This requires the implementation of a stack that contains all the begin-element names that have been seen so far but that have not been matched by a corresponding end-element tag.
4. **Validator:** Validation is an optional part of XML processing. It is often required that an XML document conform to a particular XML schema (schema defines the XML document structure and its legal building blocks) particularly in cases involving transactions between untrusted parties.
5. **API Implementor:** This layer provides the interface that is required by the calling application. In most cases, the same underlying parsing engine can be used to do the tokenization, well-formedness checking and validation while the API implementor can be customized to generate events, build a tree or supply nodes on demand.

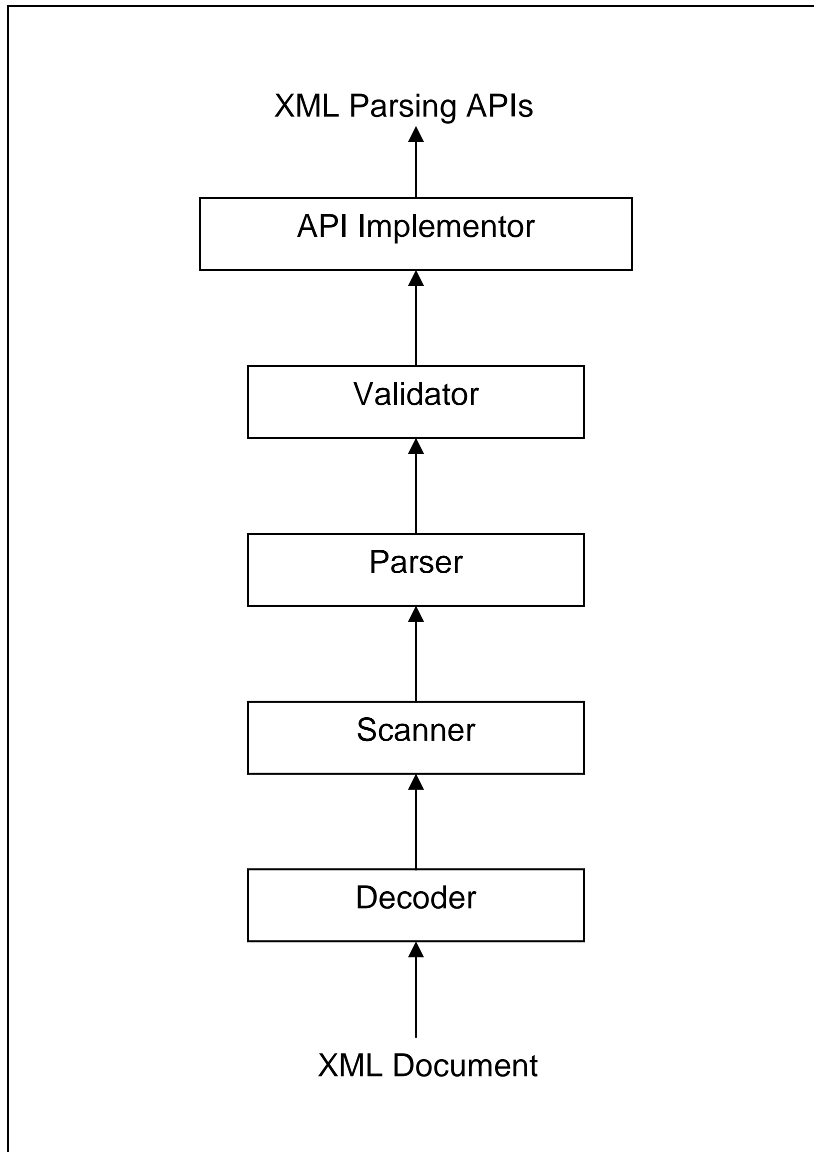

Figure 1.2: Schematic of a typical XML Parser

1.2.2 Architecture of XPath

The following are the main points the define XPath:

- XPath is a syntax for defining parts of an XML document.
- XPath uses path expressions to navigate in XML documents.
- XPath contains a library of standard functions.
- XPath is a W3C Standard.

XPath is a language for finding information in an XML document. XPath is used to navigate through elements and attributes in an XML document, and thus its primary purpose is addressing parts of an XML document. In support of this it provides basic facilities for manipulation of strings, numbers and booleans. XQuery is built on XPath expressions. Compact, non-XML syntax is used to facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document [2].

XPath includes over 100 built-in functions. There are functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, Boolean values, and more. XPath uses path expressions to select nodes or node-sets in an XML document. The node is selected by following a path or steps. In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document (root) nodes. XPath models XML documents as trees of nodes. The root of the tree is called the document node (or root node). The primary syntactic construct in XPath is the expression. An expression is evaluated to yield an object, which has one of the following four basic types [2]:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of characters)

Expression evaluation occurs with respect to a context. The context consists of:

- a node (the context node)
- a pair of non-zero positive integers (the context position and the context size)
- a set of variable bindings
- a function library
- the set of namespace declarations in scope for the expression

One important kind of expression is a location path, which selects a set of nodes relative to the context node. The result of evaluating an expression that is

a location path is the node-set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter sets of nodes. A location path can be absolute or relative, and in both cases it consists of one or more steps, each separated by a slash [2]:

- An absolute location path: /step/step/...
- A relative location path: step/step/...

Each step is evaluated against the nodes in the current node-set. A step consists of:

- an axis (defines the tree-relationship between the selected nodes and the current node)
- a node-test (identifies a node within an axis)
- zero or more predicates (to further refine the selected node-set)

The syntax for a location step is: axisname::nodetest[predicate]

An axis defines a node-set relative to the current node. A predicate filters a node-set with respect to an axis to produce a new node-set. For each node in the node-set to be filtered, the PredicateExpr is evaluated with that node as the context node, with the number of nodes in the node-set as the context size, and with the proximity position of the node in the node-set with respect to the axis as the context position; if PredicateExpr evaluates to true for that node, the node is included in the new node-set; otherwise, it is not included.

1.2.3 Examples of XPath Implementation

We will use the following XML document to illustrate the concepts discussed so far:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<bookstore>

<book>
    <title lang="eng">Harry Potter</title>
    <price>29.99</price>
</book>
```

```
<book>
   <title lang="eng">Learning XML</title>
   <price>39.95</price>
</book>

</bookstore>
```

Example of nodes in the XML document above:

- <bookstore> (document node)
- <price>29.99</price> (element node)
- lang="eng" (attribute node)

Some path expressions with and without predicates and the result of the expressions:

Path Expression	**Result**
bookstore	Selects all the child nodes of the bookstore element
/bookstore	Selects the root element bookstore
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
//book[position()<=3]	Selects the first three book elements
//title[@lang='eng']	Selects all the title elements that have an attribute named lang with a value of 'eng'
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Another example of how a function can be implemented can be shown by taking the string-join function as an example:

- fn:string-join($arg1 as string, $arg2 as string) returns a string

This function will return a string created by concatenating the members of the $arg1 sequence using $arg2 as a separator.

Assume an XML document:

```
<doc>
  <chap>
    <section>
    </section>
  </chap>
</doc>
```

With the <section> as the context node, the expression:

- fn:string-join(for $n in ancestor-or-self::* return name($n), '/')
- returns " doc/chap/section "

Later sections will show how string-join and various such functions are offloaded onto hardware.

We can take the example of the Microsoft XML parser to show how the above XPath examples are implemented in software. Microsoft's software includes integrated XPath support. Developers can load XML into a tree structure, and then query the structure with XPath to extract the data they are after. The software loads the XML document and then the selectNodes() function is used to select nodes from the XML document:

```
xmlDoc.load("books.xml")                    // loads the XML doc "books.xml"
xmlDoc.selectNodes(path expression)         // selects/filters nodes based on the
                                               path expression
```

- xmlDoc.selectNodes("/bookstore/book") - selects all the book nodes under the bookstore element.
- xmlDoc.selectNodes("/bookstore/book[0]") - selects only the first book node under the bookstore element.

- xmlDoc.selectNodes("/bookstore/book/price/text()") - selects the text from all the price nodes.
- xmlDoc.selectNodes("/bookstore/book[price>35]/price") - selects all the price nodes with a price higher than 35.

1.3 Maximizing performance via hardware offload

1.3.1 Issues with software processing of XML/XPath

The two main issues with the software processing of XML and why XML processing is CPU intensive are:

1. The emphasis on human readability of XML documents (where the nature or number of tags is not fixed).
2. The use of UTF representation of data.

XML processing typically involves the use of an off-the-shelf parser program that has to perform lexical analysis, tokenization and character integrity checks on the XML stream. In addition it has to convert UTF characters to Unicode, verify well-formedness of the document and optionally validate against an XML Schema or Document Type Definition (DTD). Finally there is the stage that may construct a tree, perform transformations, bind to Java or C# objects and call in to the application program. All this consumes a fair amount of CPU cycles such that most of the time taken in a service-oriented application is spent in parsing XML [4]. The exchange of XML messages and the querying of XML documents and databases using XPath/XQuery are going to become equally commonplace and will also burn up a large number of CPU cycles. Thus, XML parsing and querying is a severe bottleneck in XML processing and manipulation of XML documents and databases.

A solution which eases this bottleneck is to transfer some aspects of XML parsing onto dedicated hardware, an approach called hardware offloading. With

this approach, there is a definite increase in performance as the hardware would process data much faster than software. The more tasks that are offloaded, the faster the data processing becomes.

The next section talks about the work being done in this area and about available XML Accelerators that offload XML parsing and some basic XML processing onto hardware. Further sections also exemplify hardware offloading by citing some industry proven techniques.

1.3.2 Current approaches to challenges faced

There have been several different special purpose XML accelerators from companies such as DataPower (IBM), Tarari and Sarvega (Intel) to alleviate the problems faced by a purely software approach to XML processing. These have typically concentrated on narrow processing problems [4].

Some examples of XML accelerators in the industry:

- Tarari Hardware XML Processor
- Datapower XA35 XML Accelerator
- Sarvega XML Content Router
- Reactivity 400 Series of XML enables networking products.

There were two main groups of engineers that worked on tackling the issues faced with XML processing:

1. One group focused on large volumes of XML transformations - created specialized software or ASICs that performed transformations up to 100 times faster than basic software solutions.
2. The second group focused on high-speed XML processing and security – created highly optimized applications that secured and integrated XML across many use cases.

Alternative terminology that describe more specific functionality of XML appliances:

- *XML Accelerators* - are devices that typically use custom hardware or software built on standards-based hardware to accelerate processing. The 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) are devices that support the WS-Security standards. These appliances typically offload encryption and decryption to specialized hardware devices.
- *XML Enabled Networking* – is an abstraction layer that exists alongside the traditional IP network. This layer addresses the security, incompatibility and latency issues encumbering XML messages, web-services and service-oriented architectures (SOA).
- *Integration Appliance* – (also known as application routers) are devices that are designed to make the integration of computer systems easier.

1.3.3 Industry examples

Through industry examples this section elaborates on how hardware offloading has resulted in performance enhancements. The examples cited are TCP/IP and Encryption/Decryption algorithms.

- TCP/IP Offload Engine (TOE)

TCP/IP processing speeds have not kept up with rapid growth in processor speeds. Processor speeds have increased from 60MHz to 3GHz. However, the rate of increase in the case of TCP/IP has not been in parallel with the scaling of the CPU clock speed. TCP/IP is memory intensive; hence memory and I/O subsystems become limiting factors. A significant amount of software is required to implement the various features of TCP/IP. TCP/IP processing involves a great deal of memory for storing connection state information. Hence, TCP/IP in software incurs a large overhead and involves significant amount of memory.

Despite all efforts to increase TCP/IP performance via software modifications, the performance benefits that result fall behind the improvement achieved by hardware offloading. Terminator [7], a TCP offload engine developed by Chelsio Communications, Inc. highlights the advantages of TCP hardware offloading. During performance evaluation, the throughput achieved with offload was found to be almost four times than that without offload.

- Encryption / Decryption Algorithms

Secure Socket Layer (SSL) handshakes require extensive compute resource. Security algorithms for encryption and decryption impose a significant performance penalty while protecting data traveling over the Internet. SSL is known to slow down an application or Web site considerably. The time required to establish a session and then encrypt and decrypt the data, degrades performance as all of it heavily use processor cycles.

To combat the high computational cost of the RSA public-key encryption algorithm several companies have developed offload devices that use encryption offload hardware to accelerate processing, such as, RSA-offload only devices, Hardware Security Modules and Internet Hardware Devices.

IBM designers have responded to the problems of security and responsiveness through the use of cryptographic accelerator hardware. This accelerator offloads encryption/decryption tasks from the central processor.

1.3.4 XPath Offload Engine (XPOE)

XPath is used extensively by XML parsing software and its importance has been elaborated on in Section 1.1. As seen in previous sections several aspects of XML processing such as XML parsing have been offloaded onto hardware and have proven successfully the great increase in performance and the increased CPU cycles savings. We have also seen how hardware offloading boosts processing speed in TCP and Security algorithms. In anticipation of the growing importance of XPath in the processing of XML and related languages, it would be

a good idea to make advancements which would allow for faster processing of XPath. I chose to focus on those aspects of XPath2.0 that form the basic building blocks of an XPath expression and thus essentially XQuery1.0. These form the most common subset of the XPath processing problem: that of XPath functions.

How many XPath functions to offload is a complexity-performance tradeoff. If we offload all the functions, the complexity increases but so does the performance. To demonstrate the benefits of hardware offloading it is proposed to offload XPath string functions which involve encoding-to and decoding-from UTF-8 representation onto hardware. Chapter 3 demonstrates why these functions were chosen and through detailed analysis shows how such a limited offload can reduce CPU utilization significantly.

1.4 Thesis organization

The organization of the rest of the thesis is described. Chapter 2 and its subsections introduce XPath and its components, expression formats and function types. Chapter 3 presents an analysis which describes the potential reduction in CPU utilization via hardware offload. Chapter 4 presents the architecture of the hardware which would be doing the offloaded processing. Chapter 5 presents a detailed description of all the modules involved in the design. Chapter 6 explains the software simulation and verification and also includes additional design features. Chapter 7 completes the thesis by identifying areas which have scope for further improvements.

Chapter 2

XPath

This chapter briefly describes the structure of XPath expressions and functions and explains its components.

2.1 Introduction

XPath became a W3C Recommendation 16 November 1999. XPath was designed to be used by the XML-based style sheet language for transforming XML documents into other formats, by the XML-based language that is used to create hyperlinks in XML documents and other XML parsing software. XPath is a language for finding information in an XML document, and is used to navigate through elements and attributes in an XML document [2]. A detailed description of XPath is given in section 1.2.2.

2.1.1 XPath Design Aspects

Two important design aspects of XPath are that it is *functional* and that it is *typed*. These two aspects play an important role in XPath Semantics [2].

1. XPath is a functional language. XPath is built from expressions, rather than statements. Every construct in the language is an expression and expressions can be composed arbitrarily. The result of one expression can be used as the input to any other expression, as long as the type of the result of the former expression is compatible with the input type of the latter expression with which it is composed. Expressions can be composed of functions in various forms. As mentioned in section 1.2.2 XPath includes over 100 built-in functions, such as functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, boolean values, and more [2]. When such functions are used as part of path expressions, the nodes or node-sets in an XML document that are selected by the expression will depend on the result returned by the functions.

2. XPath is a typed language. XPath supports *static type analysis*. Static type analysis infers the output type of an expression based on the type of its input expressions. In addition to inferring the type of an expression for the user, static typing allows early detection of type errors, and can be used as the basis for certain classes of optimization.

The semantics and grammar of XPath is discussed in further sections.

2.2 XPath Grammar and its complexity

Grammar can be defined as a set of rules by which a language can be constructed, essentially the syntax. It can be thought of as a tool that defines the form or structure of the language enabling accurate interpretation by all users.

Parsing is the process of analyzing a sequence of tokens in order to determine its grammatical structure with respect to a given formal grammar. Parsing is

usually conducted in two stages, first identifying the meaningful tokens in the input, and then building a tree from those tokens. Parsing too follows the same set of rules that make up the grammar.

The grammar of XPath uses the Extended Backus-Naur Form (EBNF) notation [12]. The version of Unicode that is used to construct expressions is implementation defined. It follows the UTF-8 or UTF-16 charset. The UTF-8 charset is the same as ASCII for 00-7F and hence UTF-8 charset is assumed for this implementation. Sections 2.2.1 discusses the reasons why XPath grammar could be considered as significantly complex to parse.

2.2.1 Lexical analysis

From a lexical perspective, the following points contribute to the complexity of XPath grammar [2]:

1. Path Expression Syntax

   Path expressions are composed of a node name, followed by a forward slash ("/") to represent hierarchy, followed by another node name and so on till the desired level of hierarchy has been reached. The parser needs to be able to track the forward slash delimiters. The characters till the forward slash need to be buffered separately, as they form one node name and the characters after the forward slash form the next node name and are to be buffered separately.

2. Variable String Lengths

   As the string length is not limited, this places a requirement on the parser that it should be able to buffer several characters at a stretch. The need for this buffer is eliminated in the XPOE by writing these characters directly into memory, with no intermediate storage required.

3. Variable Number of Input Arguments

The number of input arguments to some XPath functions is not limited and hence poses a similar problem as that of the variable string length.

4. Depth of Hierarchy

The depth of hierarchy contained in a path expression is not limited. Sometimes the hierarchy can get really deep. This leads to buffering problems and cumbersome parsing.

2.3 XPath Expressions and Functions

As explained in section 1.2.2 the primary syntactic construct in XPath is the expression. An expression is evaluated to yield an object, which has one of the following four basic types; node-set, boolean, number or string. Expression evaluation occurs with respect to a context, for example, a node (the context node), a pair of non-zero positive integers (the context position and the context size), a function library, etc. The function library consists of a mapping from function names to functions. Each function takes zero or more arguments and returns a single result.

2.3.1 XPath Expression and Function Format

One important kind of expression is a location path. A location path selects a set of nodes relative to the context node. The result of evaluating an expression that is a location path is the node-set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter sets of nodes. Using the example of location paths we will see how functions can be used as part of expressions:

- **child::para[position()=last()-1]** selects the last but one *para* child of the context node.

The **last** function returns a number equal to the context size from the expression evaluation context.

- **/child::doc/child::chapter[position()=5]/child::section[position()=2]** selects the second *section* of the fifth *chapter* of the *doc* document element.

  The **position** function returns a number equal to the context position from the expression evaluation context.
- **child::para[attribute::type='warning'][position()=5]** selects the fifth *para* child of the context node that has a type *attribute* with value *warning*
- **child::chapter[child::title='Introduction']** selects the *chapter* children of the context node that have one or more title children with string-value equal to *Introduction.*

A location step has three parts:

1. An axis, which specifies the tree relationship between the nodes selected by the location step and the context node,
2. A node test, which specifies the node type and expanded-name of the nodes selected by the location step, and
3. Zero or more predicates, which use arbitrary expressions to further refine the set of nodes selected by the location step.

The syntax for a location step is the axis name and node test separated by a double colon, followed by zero or more expressions each in square brackets. For example, in child::para[position()=1], child is the name of the axis, para is the node test and [position()=1] is a predicate. The node-set selected by the location step is the node-set that results from generating an initial node-set from the axis and node-test, and then filtering that node-set by each of the predicates in turn. The initial node-set is filtered by the first predicate to generate a new node-set; this new node-set is then filtered using the second predicate, and so on. The final node-set is the node-set selected by the location step. The axis affects how the expression in each predicate is evaluated and so the semantics of a predicate is defined with respect to an axis.

A predicate filters a node-set with respect to an axis to produce a new node-set. For each node in the node-set to be filtered, the predicate expression is evaluated with that node as the context node, with the number of nodes in the node-set as the context size, and with the proximity position of the node in the node-set with respect to the axis as the context position; if predicate expression evaluates to true for that node, the node is included in the new node-set; otherwise, it is not included.

A function call expression is evaluated by using the function name to identify a function in the expression evaluation context function library, evaluating each of the arguments, converting each argument to the type required by the function, and finally calling the function, passing it the converted arguments. It is an error if the number of arguments is wrong or if an argument cannot be converted to the required type. The result of the function call expression is the result returned by the function.

Another example of how a function can be implemented was shown in section 1.2.3 and is reiterated here for clarity:

- fn:string-join($arg1 as string, $arg2 as string) returns a string

This function will return a string created by concatenating the members of the $arg1 sequence using $arg2 as a separator.

2.3.2 XPath Function Types

XPath implementations contain a function library that includes functions used to evaluate expressions. Each function in the function library is specified using a function prototype, which gives the return type, the name of the function, and the type of the arguments. The function library includes the following types of built-in functions that are required for XPath2.0/Xquery1.0.

- Functions on Numeric Values
- Functions on Strings
- Functions on Boolean Values
- Functions on Durations, Dates and Times

- Accessor Functions
- Error and Trace Functions
- Functions for anyURI
- Functions related to QNames
- Functions on Nodes
- Functions on Sequences
- Context Functions

2.3.3 XPath String Functions

In this implementation we concentrate on offloading the string functions onto hardware. Chapter 3 will present the motivation for offloading string functions onto hardware and an analysis of the processor overhead savings that was achieved by offloading specific string functions. The following paragraphs will talk briefly about the string functions that were chosen to be offloaded.

- **fn:codepoints-to-string(*int,int,...*)**

Returns a string from a sequence of code points

Example: codepoints-to-string(84, 104, 233, 114, 232, 115, 101)

Result: 'Thérèse'

Each of the input arguments to this function is an integer, which can have any value from 0 to 2097151 in decimal or from 0x000000 to 0x1FFFFF in hex. This function takes each one of the input arguments and encodes them to their respective char UTF-8 notation. These encoded characters will then together form the result string represented in UTF-8. Only character numbers between 0x000000 to 0x1FFFFF (21 bits) can be represented in UTF-8 hence the restriction on the range of the input numbers. Sections 3.3.1 and 3.3.2 will further illustrate this concept and show the significant savings that can be achieved by offloading such functions onto hardware.

- **fn:string-to-codepoints(*string*)**

Returns a sequence of code points from a string

Example: string-to-codepoints("Thérèse")

Result: 84, 104, 233, 114, 232, 115, 101

The input argument to the above function is a string in UTF-8 character representation. Each character of the input string is decoded from UTF-8 to give its equivalent character number. "Character number", also known as "code point" or “code position" is used to mean a non-negative integer that represents a character in some encoding. The result is a sequence of character numbers representing the individual characters of the input string, decoded from their UTF-8 representation to character numbers. Sections 3.3.1 and 3.3.2 demonstrate that string functions that involve encoding to and decoding form UTF-8 character representation benefit from significant CPU cycles savings.

- **fn:codepoint-equal(*comp1,comp2*)**

Returns true if the value of comp1 is equal to the value of comp2

The input arguments to this function are strings. The two strings are compared byte for byte. If there is a mismatch false is returned else true.

- **fn:compare(*comp1,comp2*)**

Returns -1 if comp1 is less than comp2, 0 if comp1 is equal to comp2, or 1 if comp1 is greater than comp2

Although the compare function seems similar to the codepoint-equal function above, the difference is that instead of returning just a True or False corresponding to a match or mismatch, in case of a mismatch the two strings are compared to check which string is greater. In such cases the two strings are compared code-point to code-point based on a specified collation rule.

A collation is a specification of the manner in which character strings are compared and, by extension, ordered. When values whose type is string is compared (or, equivalently, sorted), the comparisons are inherently performed according to some collation (even if that collation is defined entirely on code point

values). Collations can indicate that two different code points are, in fact, equal for comparison purposes (e.g., "v" and "w" are considered equivalent in Swedish). Strings can be compared codepoint-by-codepoint or in a linguistically appropriate manner, as defined by the collation.

When the Unicode code point collation is used, this simply involves determining whether arg1 contains a contiguous sequence of characters whose code points are the same, one for one, with the code points of the characters in arg2. All collations support the capability of deciding whether two strings are considered equal, and if not, which of the strings should be regarded as preceding the other as required in functions such as compare().

- **fn:substring(*string,start,len*); fn:substring(*string,start*)**

Returns the substring from the start position to the specified length. Index of the first character is 1. If length is omitted it returns the substring from the start position to the end

Example: substring('Beatles',1,4)

Result: 'Beat'

Example: substring('Beatles',2)

Result: 'eatles'

This function takes in a string in UTF-8 representation and start and length numbers are arguments. Although the string does not need to be decoded to return a substring, we however need to identify character boundaries as each character can be anywhere from one to four bytes. Identifying the number of octets per character will allow us to locate the character at the start index, and the substring starting at this index up to the length number of characters will be the result string of the function.

- **fn:translate(*string1,string2,string3*)**

Converts string1 by replacing the characters in string2 with the characters in string3.

Example: translate('12:30','0123','abcd')

Result: 'bc:da'

The translate function also requires that the UTF-8 characters be decoded to their code points to be able to find all the instances of occurrence of characters of string2 in string1 and then replace each one of these with each corresponding character from string3. As this function involves decoding three input strings it benefits greatly from the CPU cycles savings achieved through offloading the decoding/encoding functions onto hardware as shown in sections 3.3.1 and 3.3.2. In functions that involve character counting such as substring, string-length and translate functions, what is counted is the number of XML characters in the string (or equivalently, the number of Unicode code points).

2.4 Requirements for parsing XPath Functions

An XPath string function essentially consists of a function call through the function name, a set of input arguments of a particular type and a return value of set of values of a corresponding type. Based on the XPath string functions, the following requirements apply.

- Pre-compile and pre-load XPath expressions
- When an appropriate XPath function that can be offloaded is identified by the compiler, the pointer to each input argument (located in main memory) along with corresponding argument lengths is written into the input section of the memory. Once this is done an opcode corresponding to the particular function, and an interrupt signal asking the accelerator to take over are issued by the CPU to the XPOE.
- UTF-8 character set will be used for character representation
- Unicode code point collation is used for string and substring matching and comparisons.

Chapter 3

Processor overhead savings analysis

This chapter briefly discusses the drawbacks of XPath function implementation in software. It then presents an analysis which demonstrates how offloading specific XPath functions onto hardware will allow us to achieve significant processor cycle savings.

3.1 Software approach to XPath function implementation

This section elaborates on why XPath functions and hence XPath expressions are considered to be CPU intensive. XPath functions form part of path expressions that are used to filter required XML nodes or XML data from an XML document or database. We have seen the structure of XPath expressions in Chapter 2. Therefore, as XPath functions take XML data as arguments, the software implementation of these functions and expressions are a cumbersome task, owing to the following reasons:

1. Use of UTF representation of data.

XML uses UTF-8 or UTF-16 for representation of data and hence XPath and XQuery work on data that is represented in UTF. UTF or Unicode Transformation Format is a variable-length character encoding for Unicode. Unicode is an industry standard designed to allow text and symbols from all of the writing systems of the world to be consistently represented and manipulated by computers. UTF is able to represent any universal character in the Unicode standard, yet the initial encoding of byte codes and character assignments for UTF-8 is consistent with ASCII.

UTF-8 uses one to four bytes per character, depending on the Unicode symbol. Only one byte is needed to encode the 128 US-ASCII characters (Unicode range U+0000 to U+007F). Two bytes are needed for Latin letters with diacritics and for characters from Greek, Cyrillic, Armenian, Hebrew, Arabic, Syriac and Thaana alphabets (Unicode range U+0080 to U+07FF). Three bytes are needed for the rest of the Basic Multilingual Plane and four bytes are needed for characters in other planes of Unicode.

The table below summarizes the format of these different octet types. The letter x indicates bits available for encoding bits of the character number.

Table 3.1: Format of the different octet types in UTF-8 encoding

Num of free bits	Char. number range (hexadecimal)	UTF-8 octet sequence (binary)
7	0000 0000-0000 007F	0xxxxxxx
(5+6)=11	0000 0080-0000 07FF	110xxxxx 10xxxxxx
(4+6+6)=16	0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
(3+6+6+6)=21	0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Examples

- The character sequence U+0041 U+2262 U+0391 U+002E "A<NOT IDENTICAL TO><ALPHA>." is encoded in UTF-8 as follows:

  --+------------+--------+---

  41 E2 89 A2 CE 91 2E

  --+------------+--------+---

- The character sequence U+D55C U+AD6D U+C5B4 (Korean "hangugeo", meaning "the Korean language") is encoded in UTF-8 as follows:

  ------------+-------------+------------

  ED 95 9C EA B5 AD EC 96 B4

  ------------+-------------+------------

- The character sequence U+65E5 U+672C U+8A9E (Japanese "nihongo", meaning "the Japanese language") is encoded in UTF-8 as follows:

  -----------+-------------+------------

  E6 97 A5 E6 9C AC E8 AA 9E

  -----------+-------------+------------

2. Variable string lengths

The arguments to the string functions are of variable string length. As each string can have an extremely small or an extremely large number of characters, this poses a problem for the parser when it comes to buffering so many characters at a stretch.

3. Variable number of arguments

The arguments to the string functions are variable in number. Although some string functions have a fixed number of arguments other functions can have a variable number of arguments, making it difficult for the parser to optimize each function.

Section 3.2 elaborates on the high level block diagram of the XPath Offload Engine (XPOE), which allows the CPU savings, discussed in section 3.3.

3.2 High level Block diagram for hardware offload

The high level block diagram which shows the interaction between the CPU, the memory and the XPOE is shown in Figure 3.1.

1. Memory

The memory is shared between the CPU and the XPOE. It is implemented as a 32-bit memory. The motivation behind choosing a 32-bit data width is the fact that UTF-8 uses a maximum of 4 bytes to represent each character.

The memory is divided into 3 distinct sections. Different information is stored at pre-decided memory locations. The first section is the main memory section that is read and written to by the CPU and can only be read by the XPOE. The remaining part of the memory outside of the main memory is divided into the input and output section of the memory.

As mentioned in section 2.4 when an appropriate XPath function that can be offloaded is identified by the compiler, the pointer to each input argument (located in main memory) along with corresponding argument lengths is written into the input section of the memory. The result or the output of the function execution is written into the output section of the memory by the XPOE. The result string will be stored at a predefined location in the output section of the memory. A counter is run while this result value is being stored. Once the entire value is stored, the value of the counter serves as the length of the result string. The pointer to the result and the length of the result are stored at the start of the output section.

When control returns to the CPU, the software can then easily use the string pointer which points to the start of the result string along with its length to obtain the result value. The CPU is aware of all these addresses and thus knows where to store the pointers to the input arguments and where to read the result from. An explanation of each of the main blocks that form the XPOE follows.

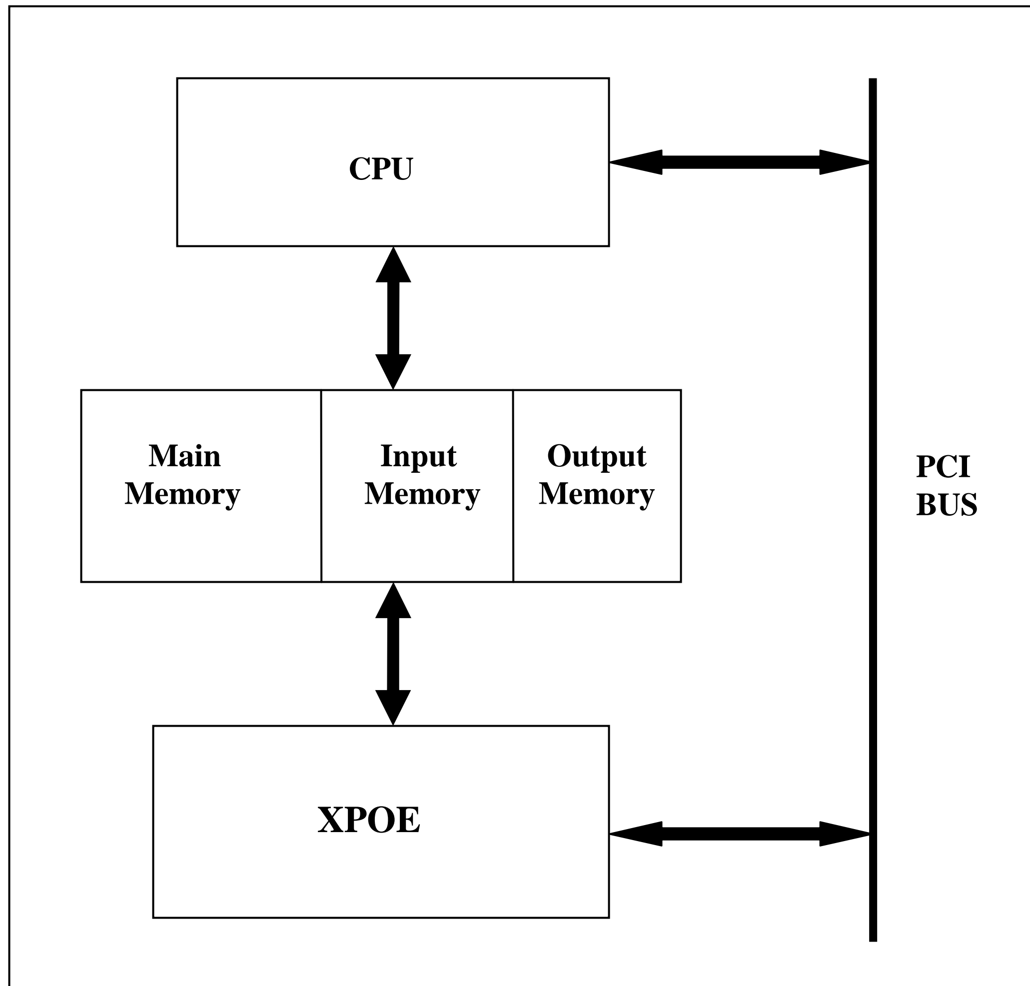

Figure 3.1: High level block diagram showing system level interaction

2. XPOE

The XPOE is made up of 5 main modules. Each one is mentioned below with a brief description. A detailed description of the behavior and functionality of the XPOE and its modules is given in chapters 4 and 5.

- Read Input Memory

This module is the interface between the input section of the memory and the rest of the XPOE. When the XPOE is interrupted by the CPU with a start signal the read input memory module will interpret the opcode received from the CPU

and appropriately read the required number of bytes from the input section of the memory.

- Read Main Memory

This module receives the bytes read from the input memory and based on the opcode interprets the bytes read. It identifies the string pointers and using these pointers it reads the input arguments to the function from the appropriate location in main memory word by word.

- Function Execute

This module implements the execution of the actual XPath function based on the opcode that is passed to the XPOE by the CPU. It operates on the character data passed to it from the main memory. The output of the operation on each input word is then delivered to be written to the output memory.

- Memory Write Module

This module writes the result of the function execution into the output section of the memory. Once that is done a ‘complete’ signal goes high and wakes up the CPU, which will then read the result from the predefined location of the output section of memory.

- Controller

This module implements the Finite State Machine (FSM). When the start interrupt goes high the FSM enables the read input module. Based on signals from the read main memory, function execute and memory write modules the FSM will enable the appropriate modules. When the complete signal goes high indicating that the XPOE has finished execution the CPU will be interrupted.

3.3 Processor savings analysis

3.3.1 Encode-to and Decode-from UTF-8 representation

The analysis starts by showing the CPU cycles savings that is incurred in encoding-to and decoding-from UTF-8 character representation. XPath functions take XML data as arguments. As seen in section 3.1 one of the main reasons that XPath functions are CPU intensive is the UTF representation of XML character data. UTF-8 character representation is assumed for this implementation. A majority of the XPath string functions involve encoding-to or decoding-from UTF-8 representation to some extent or the other. Thus, we can infer that by offloading the encoding and decoding operations onto hardware a significant amount of CPU cycles can be saved by the majority of the XPath string functions. The following tables will elaborate on the encoding and decoding operations and present an analysis on the number of CPU cycles involved with the help of pseudo-code. An example input is used for analysis purposes. The input contains 6 character numbers in the case of encode or a string with a string length of 6 in the case of decode. The number of bytes/octets per character is taken as 3 for this example. The analysis will then compare the CPU utilization involved in the encoding-to and decoding-from UTF-8 representation of this example input for the following two methods:

1) Complete software approach

The software reads the arguments and implements the encoding-to and decoding-from UTF-8 functions.

- Encoding a character number to a UTF-8 character

1. Determine the number of octets required from the char number.
2. Then, prepare the high order bits of the octets.
3. Finally, fill in the appropriate bits from the bits of the character number, expressed in binary.

Table 3.2: Cycles required for an all-software approach for encoding function

TRADITIONAL SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR NUM**
Read the char numbers from memory	L1: READ Reg1, 4 bytes	READ	6	1
Generate mask	MOV Reg2, mask	MOV	30	5
Check how many octets are required per char	COMPARE Reg1, Reg2	COMPARE	30	5
Generate high-order and other bit masks	Mov Reg2, mask	MOV	24	4
Left Shift appropriate no of bits to align num bits	AND Reg1, Reg2	AND	18	3
correctly with the high order bits of the octets	SHIFTL Reg1, 2	SHIFT	12	2
OR to get the num encoded in UTF-8	OR Reg1, bitpattern	OR	24	4
	LOOP L1	INCR	6	1
Write encoded chars back into memory	WRITE Reg1, writeaddress	WRITE	6	1
				SCALE
	TOTAL	**READ**	**6**	1
		COMPARE	**30**	1
		INCR	**6**	1
		MOV	**54**	1
		AND	**18**	1
		SHIFT	**12**	1
		OR	**24**	1
		WRITE	**6**	1
	TOTAL INSTRUCTION CYCLES		**156**	

- Decoding a UTF-8 character to a character number

1. Initialize a number and set its bits to one.
2. Determine how many bits are used to represent the character, that is, which bits encode the character.
3. Finally, distribute the bits from the sequence to the initialized number, form the lower order bits, proceeding to the left. This number will now equal the character number.

Table 3.3: Cycles required for an all-software approach for decoding function

TRADITIONAL SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Read UTF-8 char from memory	L1: READ Reg1, 4 bytes	READ	6	1
Generate mask	MOV Reg2, mask	MOV	6	1
Check number of bytes per char	L2: AND Reg1, Reg2	AND	18	3
Generate value	MOV Reg2, value	MOV	66	11
Mask and compare	COMPARE Reg1, Reg2	COMPARE	66	11
Shift right after each compare to check	SHIFTR Reg1, 8	SHIFT	12	2
subsequent higher bytes	JumpNotEqual L2			
Generate octet bit masks to obtain only those	L3: MOV Reg2, mask	MOV	18	3
bits that make the char number	AND Reg1, Reg2	AND	18	3
Right shift appropriate num of bits for correct	SHIFTR Reg1, 2	SHIFT	12	2
Alignment	OR Reg1, bitpattern	OR	12	2
OR with initialized num to get final char num	LOOP L1	INCR	6	1
Write decoded char nums back into memory	WRITE Reg1, writeaddress	WRITE	6	1
				SCALE
	TOTAL	**READ**	**6**	1
		AND	**36**	1
		COMPARE	**66**	1
		SHIFT	**24**	1
		INCR	**6**	1
		MOV	**90**	1
		OR	**12**	1
		WRITE	**6**	1
	TOTAL INSTRUCTION CYCLES		**246**	

2) Hardware offload approach

The implementation of the encoding and decoding functions are offloaded onto dedicated hardware. The software does trivial address calculations to obtain the result output by the functions.

Table 3.4: Cycles required for a hardware-offloaded approach

PROPRIETARY SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Write the string pointer and strlen into	WRITE strptr, writeaddress	WRITE	2	1
the separate output section of the memory				
Read the result, ie., the encoded or	ADD Offset base address, Index	READ	6	1
decoded value that has been written	READ Reg1, 4 bytes	INCR	6	1
into the output section of the memory by	ADD Value base address, Reg1	ADD	2	1
the accelerator.	L1: READ Reg1, 4 bytes	COMPARE	6	1
	COMPARE Field Length,0			
	JumpNotEqual L1			
				SCALE
	TOTAL	**READ**	**6**	1
		COMPARE	**6**	1
		INCR	**6**	1
		ADD	**2**	1
		WRITE	**2**	1
	TOTAL INSTRUCTION CYCLES		**22**	

From Tables 3.2, 3.3 and 3.4, it can be seen that the cycles required by the hardware-offload method (22 cycles) is significantly lower than those required when using an all-software approach (156 cycles for encoding and 246 cycles for decoding). This translates into a savings of over 85% for encoding and over 89% for decoding functions.

3.3.2 XPath string functions

In section 3.3.1 we saw the significant number of CPU cycles that can be saved by offloading encoding and decoding operations onto hardware. In this section we continue our analysis by examining example XPath string functions that involve encoding and decoding to some extent and hence could lead to similar CPU cycles savings. Each analysis will consider a sample XPath function and corresponding arguments. And similarly the analysis will follow two methods to compare the CPU utilization involved in the execution of the example XPath functions. The complete software approach and the hardware offload approach. A detailed description of all the example functions below is presented in section 2.3.3

- **Function to assemble strings: codepoints-to-string(int, int, …)**
  Returns a string from a sequence of codepoints
  Example: codepoints-to-string(84, 104, 233, 114, 232, 115, 101)
  Result: 'Thérèse'

- Type of input: character numbers / codepoints / integers
- Number of character numbers / integer arguments: 6
- Number of bytes/octets per character: 3

1) Complete software approach: codepoints-to-string function

Table 3.5: Cycles required for an all-software approach: codepoints-to-string function

TRADITIONAL SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR NUM**
Read the char nums from memory	L1: READ Reg1, 4 bytes	READ	6	1
Generate mask	MOV Reg2, mask	MOV	30	5
Check how many octets are required per char	COMPARE Reg1, Reg2	COMPARE	30	5
Generate high-order and other bit masks	Mov Reg2, mask	MOV	24	4
Left Shift appropriate no of bits to align num bits	AND Reg1, Reg2	AND	18	3
correctly with the high order bits of the octets	SHIFTL Reg1, 2	SHIFT	12	2
OR to get the num encoded in UTF-8	OR Reg1, bitpattern	OR	24	4
	LOOP L1	INCR	6	1
Write encoded chars back into memory	WRITE Reg1, writeaddress	WRITE	7	1
				SCALE
	TOTAL	**READ**	**6**	1
		COMPARE	**30**	1
		INCR	**6**	1
		MOV	**54**	1
		AND	**18**	1
		SHIFT	**12**	1
		OR	**24**	1
		WRITE	**7**	1
	TOTAL INSTRUCTION CYCLES		**157**	

2) Hardware offload approach: codepoints-to-string function

Table 3.6: Cycles required for a hardware-offloaded approach: codepoints-to-string function

PROPRIETARY SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Write the string pointer and strlen into	WRITE strptr, writeaddress	WRITE	2	
the separate intput section of the memory				
Read the result, ie., the encoded or	ADD Offset base address, Index	READ	6	1
decoded value that has been written	READ Reg1, 4 bytes	INCR	6	1
into the output section of the memory by	ADD Value base address, Reg1	ADD	2	
The accelerator.	L1: READ Reg1, 4 bytes	COMPARE	6	1
	COMPARE Field Length,0			
	JumpNotEqual L1			
				SCALE
	TOTAL	**READ**	**6**	1
		COMPARE	**6**	1
		INCR	**6**	1
		ADD	**2**	1
		WRITE	**2**	1
	TOTAL INSTRUCTION CYCLES		**22**	

- **Function to disassemble strings: string-to-codepoints(string)**

  Returns a sequence of code points from a string

  Example: string-to-codepoints("Thérèse")

  Result: 84, 104, 233, 114, 232, 115, 101

- Type of input: string
- Number of character numbers / integer arguments: 6
- Number of bytes/octets per character: 3

1) Complete software approach: string-to-codepoints function

Table 3.7: Cycles required for an all-software approach: string-to-codepoints function

TRADITIONAL SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Read UTF-8 chars from memory	L1: READ Reg1, 4 bytes	READ	7	1
Generate mask	MOV Reg2, mask	MOV	6	1
Check number of bytes per char	L2: AND Reg1, Reg2	AND	18	3
Generate value	MOV Reg2, value	MOV	66	11
Mask and compare	COMPARE Reg1, Reg2	COMPARE	66	11
Shift right after each compare to check	SHIFTR Reg1, 8	SHIFT	12	2
subsequent higher bytes	JumpNotEqual L2			
Generate octet bit masks to obtain only those	MOV Reg2, mask	MOV	18	3
bits that make the char number	AND Reg1, Reg2	AND	18	3
Right shift appropriate num of bits for correct	SHIFTR Reg1, 2	SHIFT	12	2
alignment	OR Reg1, bitpattern	OR	12	2
OR with initialized num to get final char num	LOOP L1	INCR	6	1
Write decoded char nums back into memory	WRITE Reg1, writeaddress	WRITE	6	1
				SCALE
	TOTAL	**READ**	**7**	1
		AND	**36**	1
		COMPARE	**66**	1
		SHIFT	**24**	1
		INCR	**6**	1
		MOV	**90**	1
		OR	**12**	1
		WRITE	**6**	1
	TOTAL INSTRUCTION CYCLES		**247**	

2) Hardware offload approach: string-to-codepoints function

Table 3.8: Cycles required for a hardware-offloaded approach: string-to-codepoints function

PROPRIETARY SCHEME CALCULATIONS				
FUNCTION	PSEUDO CODE	OPERATOR	#CYCLES	PER CHAR
Write the string pointer and strlen into	WRITE strptr, writeaddress	WRITE	2	1
the separate intput section of the memory				
Read the result, ie., the encoded or	ADD Offset base address, Index	READ	6	1
decoded value that has been written	READ Reg1, 4 bytes	INCR	6	1
into the output section of the memory by	ADD Value base address, Reg1	ADD	2	1
The accelerator.	L1: READ Reg1, 4 bytes	COMPARE	6	1
	COMPARE Field Length,0			
	JumpNotEqual L1			
				SCALE
	TOTAL	READ	6	1
		COMPARE	6	1
		INCR	6	1
		ADD	2	1
		WRITE	2	1
	TOTAL INSTRUCTION CYCLES		22	

- **Function to test equality and comparison of strings: codepoint-equal(comp1,comp2)**

  Returns true if the value of comp1 is equal to the value of comp2 Example: codepoint-equal('Thérèse', 'Thérèse')

  Result: true

- Type of input: strings
- Number of character numbers / integer arguments: 6
- Number of bytes/octets per character: 3

1) Complete software approach: codepoint-equal

Table 3.9: Cycles required for an all-software approach: codepoint-equal function

TRADITIONAL SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Read string1 from memory	L1: READ Reg1, 4 bytes	READ	7	1
Read string2 from memory	READ Reg2, 4 bytes	READ	7	1
Compare the 2 strings byte per byte until	COMPARE Reg1, Reg2	COMPARE	24	4
not equal… else return true	JumpEqual L1	INCR	6	1
Write result into memory	WRITE Reg1, writeaddress	WRITE	1	
				SCALE
	TOTAL	**READ**	**14**	1
		COMPARE	**24**	1
		INCR	**6**	1
		WRITE	**1**	1
	TOTAL INSTRUCTION CYCLES		**45**	

2) Hardware offload approach: codepoint-equal

Table 3.10: Cycles required for a hardware-offloaded approach: codepoint-equal function

PROPRIETARY SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Write the str pointer and strlen for each string into the separate intput section of the memory	WRITE strptr, writeaddress	WRITE	4	
Read the result	READ Reg1, 4 bytes	READ	1	
				SCALE
	TOTAL	**READ**	**1**	1
		WRITE	**4**	1
	TOTAL INSTRUCTION CYCLES		**5**	

- **Functions on string values: substring(string,start,len), substring(string,start)**

  Returns the substring from the start position to the specified length. Index of the first character is 1. If length is omitted it returns the substring from the start position to the end.

  Example: substring('Beatles',1,4)

  Result: 'Beat'

  Example: substring('Beatles',2)

  Result: 'eatles'

- Type of input: strings
- Number of character numbers / integer arguments: 6
- Number of bytes/octets per character: 3

1) Complete software approach: substring

Table 3.11: Cycles required for an all-software approach: substring function

TRADITIONAL SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Read string from memory	L1: READ Reg1, 4 bytes	READ	7	1
Generate mask	MOV Reg2, mask	MOV	6	1
Check number of bytes per char	L2: AND Reg1, Reg2	AND	24	4
Generate value	MOV Reg2, value	MOV	84	14
Mask and compare	COMPARE Reg1, Reg2	COMPARE	102	17
Shift right after each compare to check	SHIFTR Reg1, 8	SHIFT	18	3
subsequent higher bytes	JumpNotEqual L2	INCR	12	2
Loop back for each char in the input string	LOOP L1	INCR	6	1
Write encoded chars back into memory	WRITE Reg1, writeaddress	WRITE	7	1
				SCALE
	TOTAL	**READ**	**7**	1
		AND	**24**	1
		COMPARE	**102**	1
		SHIFT	**18**	1
		INCR	**18**	1
		MOV	**90**	1
		WRITE	**7**	1
	TOTAL INSTRUCTION CYCLES		**266**	

2) Hardware offload approach: substring

Table 3.12: Cycles required for a hardware-offloaded approach: substring function

PROPRIETARY SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Write the str ptr, strlen, start position and length args	WRITE strptr, writeaddress	WRITE	4	1
into the separate intput section of the memory				
Read the resultant string that has been written	ADD Offset base address, Index	READ	6	1
into the output section of the memory by	READ Reg1, 4 bytes	INCR	6	1
the accelerator.	ADD Value base address, Reg1	ADD	2	1
	L1: READ Reg1, 4 bytes	COMPARE	6	1
	COMPARE Field Length,0			
	JumpNotEqual L1			
				SCALE
	TOTAL	**READ**	**6**	1
		COMPARE	**6**	1
		INCR	**6**	1
		ADD	**2**	1
		WRITE	**4**	1
	TOTAL INSTRUCTION CYCLES		**24**	

- **Functions on string matching: starts-with(string1,string2)**

  Returns true if string1 starts with string2, otherwise it returns false

  Example: starts-with('XML','X')

  Result: true

  ends-with(string1,string2)

  Returns true if string1 ends with string2, otherwise it returns false

  Example: ends-with('XML','X')

  Result: false

- Type of input: strings
- String1 length: 8
- String2 length: 4
- Number of bytes/octets per character: 3

1) Complete software approach: starts-with and ends-with

Table 3.13: Cycles required for an all-software approach: starts-with and ends-with functions

TRADITIONAL SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Read string1 from memory	L1: READ Reg1, 4 bytes	READ	9	1
Read string2 from memory	READ Reg2, 4 bytes	READ	5	1
Compare the 2 strings byte per byte until	COMPARE Reg1, Reg2	COMPARE	16	4
not equal… else return true	JumpEqual L1	INCR	4	1
Write result into memory	WRITE Reg1, writeaddress	WRITE	1	
				SCALE
	TOTAL	**READ**	**14**	1
		COMPARE	**16**	1
		INCR	**4**	1
		WRITE	**1**	1
	TOTAL INSTRUCTION CYCLES		**35**	

2) Hardware offload approach: starts-with and ends-with

Table 3.14: Cycles required for a hardware-offloaded approach: starts-with and ends-with functions

PROPRIETARY SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Write the str pointer and strlen for each string into the separate intput section of the memory	WRITE strptr, writeaddress	WRITE	4	
Read the result	READ Reg1, 4 bytes	READ	1	
				SCALE
	TOTAL	**READ**	**1**	1
		WRITE	**4**	1
	TOTAL INSTRUCTION CYCLES		**5**	

- **Functions on string matching: substring-before(string1,string2)**

  Returns the start of string1 before string2 occurs in it

  Example: substring-before('12/10','/')

  Result: ‘12’

  substring-after(string1,string2)

  Returns the remainder of string1 after string2 occurs in it

  Example: substring-after('12/10','/')

  Result: ‘10’

- Type of input: strings
- String1 length: 8
- String2 length: 4
- Number of bytes/octets per character: 3

1) Complete software approach: substring-before and substring-after

Table 3.15: Cycles required for an all-software approach: substring-before and substring-after functions

TRADITIONAL SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Read string1 from memory	L1: READ Reg1, 4 bytes	READ	9	1
Read string2 from memory	READ Reg2, 4 bytes	READ	5	1
Compare the 2 strings byte per byte until not equal… else return true	COMPARE Reg1, Reg2	COMPARE	32	4
	JumpNotEqual L1	INCR	4	1
Write result into memory	WRITE Reg1, writeaddress	WRITE	4	1
				SCALE
	TOTAL	**READ**	**14**	1
		COMPARE	**32**	1
		INCR	**4**	1
		WRITE	**4**	1
	TOTAL INSTRUCTION CYCLES		**54**	

2) Hardware offload approach: substring-before and substring-after

Table 3.16: Cycles required for a hardware-offloaded approach: substring-before and substring-after functions

PROPRIETARY SCHEME CALCULATIONS				
FUNCTION	**PSEUDO CODE**	**OPERATOR**	**#CYCLES**	**PER CHAR**
Write the str pointer and strlen for each string into the separate intput section of the memory	WRITE strptr, writeaddress	WRITE	4	1
Read the result	READ Reg1, 4 bytes	READ	4	1
				SCALE
	TOTAL	**READ**	**4**	1
		WRITE	**4**	1
	TOTAL INSTRUCTION CYCLES		**8**	

From Tables 3.5 through 3.16, it can be seen that the cycles required by the hardware-offload method is significantly lower than those required when using an all-software approach for each one of the example XPath string functions. Below is a comparison of the savings achieved for each of the functions.

Table 3.17: Comparison of the savings achieved for each of the functions

XPath Function	**S/W Approach (#CYCLES)**	**H/W Offload Approach (#CYCLES)**	**Savings**
codepoints-to-string	157	22	86%
string-to-codepoints	247	22	89%
codepoint-equal	45	5	89%
substring	266	24	91%
starts-with ends-with	35	5	86%
substring-before substring-after	54	8	84%

From table 3.17 we see that the average savings achieved over all functions is around 88%. The feasibility and importance of this number is demonstrated in tables 3.18 and 3.19. Table 3.18 shows the percent savings achieved for varying percentages (taken from 10% to 100%) of the above functions in any XML document or documents. Table 3.19 similarly shows the speedup (from Amdahl's law) that can be achieved for varying percentages of the above string functions.

Table 3.18: Comparison of fraction of string functions to the savings achieved

Maximum achievable savings = 88%

Fraction (or % of string functions)	**Savings (%)**
0.1	9
0.2	18
0.3	26
0.4	35
0.5	44
0.6	53
0.7	62
0.8	70
0.9	79
1.0	88

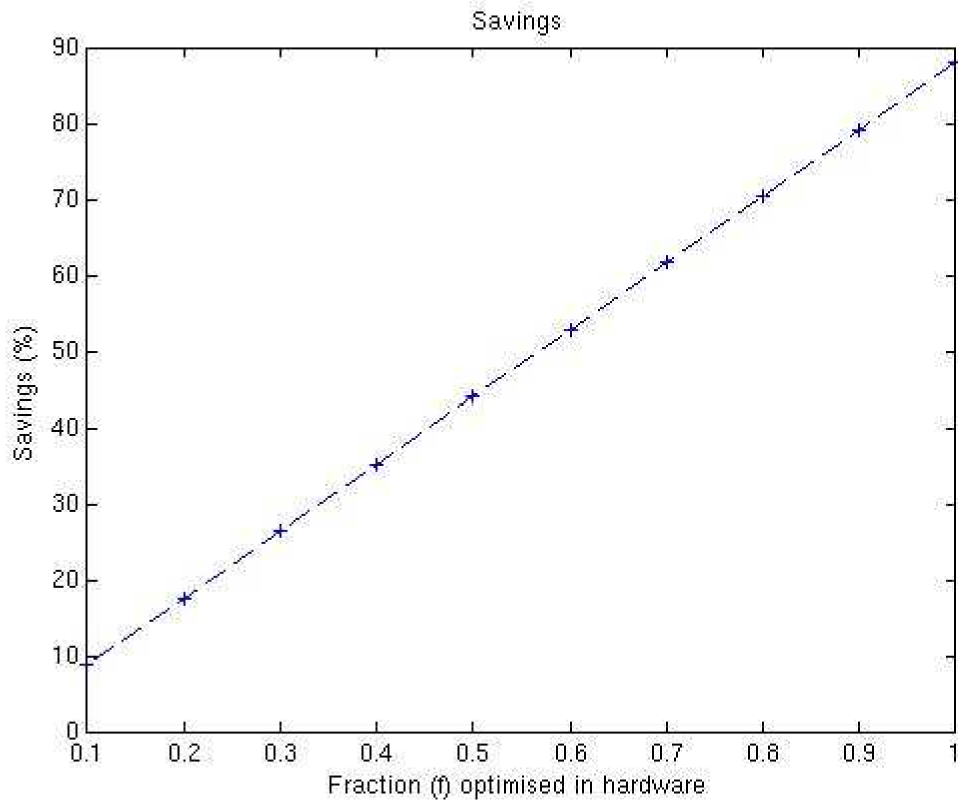

Figure 3.2: Plot of fraction of XPath functions offloaded onto hardware to the savings achieved

Table 3.19: Comparison of fraction of string functions to the speedup achieved

Maximum achievable speedup = s = 8.33

Fraction (f) **(% string functions)**	**Overall Speedup (S)** **(S = 1 / [(1-f) + (f/s)])**
0.1	1.10
0.2	1.21
0.3	1.36
0.4	1.54
0.5	1.76
0.6	2.12

Table 3.19 continued: Comparison of fraction of string functions to the speedup

0.7	2.60
0.8	3.38
0.9	4.81
1.0	8.33

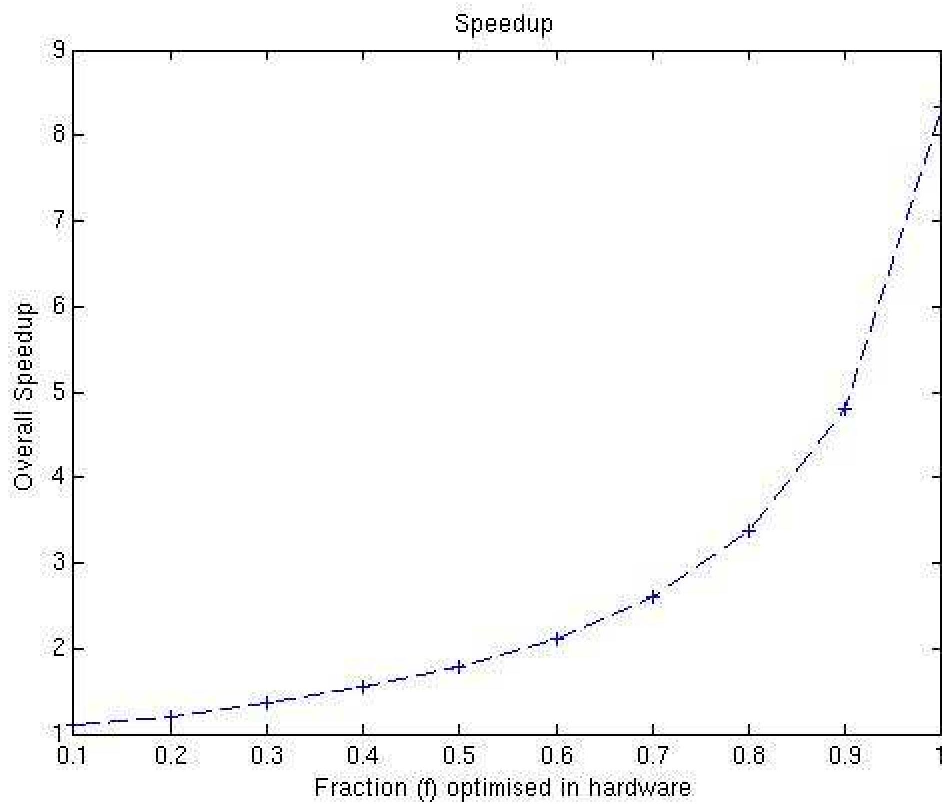

Figure 3.3: Plot of fraction of XPath functions offloaded onto hardware to the speedup achieved

From the above tables and figures we can see that hardware offload results in significant CPU cycles savings and is thus highly favorable.

Chapter 4

Design Architecture

4.1 Introduction

In chapter 3 we saw the potential savings the XPOE could achieve. In this section we take a high level look at the design architecture. The block diagram will be discussed. Functions and implementation details of individual blocks will also be elaborated on.

The design examines the incoming opcode on being interrupted to identify the function to be implemented. The design also simultaneously starts reading the input section of memory to get the required pointers to the input arguments of the function, which are located in main memory. The input arguments are read byte by byte and as per the opcode the appropriate function is executed. The result of the function execution is then written to the output section of the memory byte by byte. Storage occurs only at the final stage and hence no back pressure is exerted on the input. Once the writing of the result into output memory has been

successfully completed, the CPU is interrupted and it is assumed that the PCI device would read this result data, to be examined by the CPU.

The following sections list all the blocks involved in the design. Their functions and a brief idea of their implementation are given.

4.2 Block Diagram and Explanation

The figure 4.1 shows the block level implementation of the XPath Offload Engine (XPOE). We shall discuss the functions and brief implementation in subsequent sections.

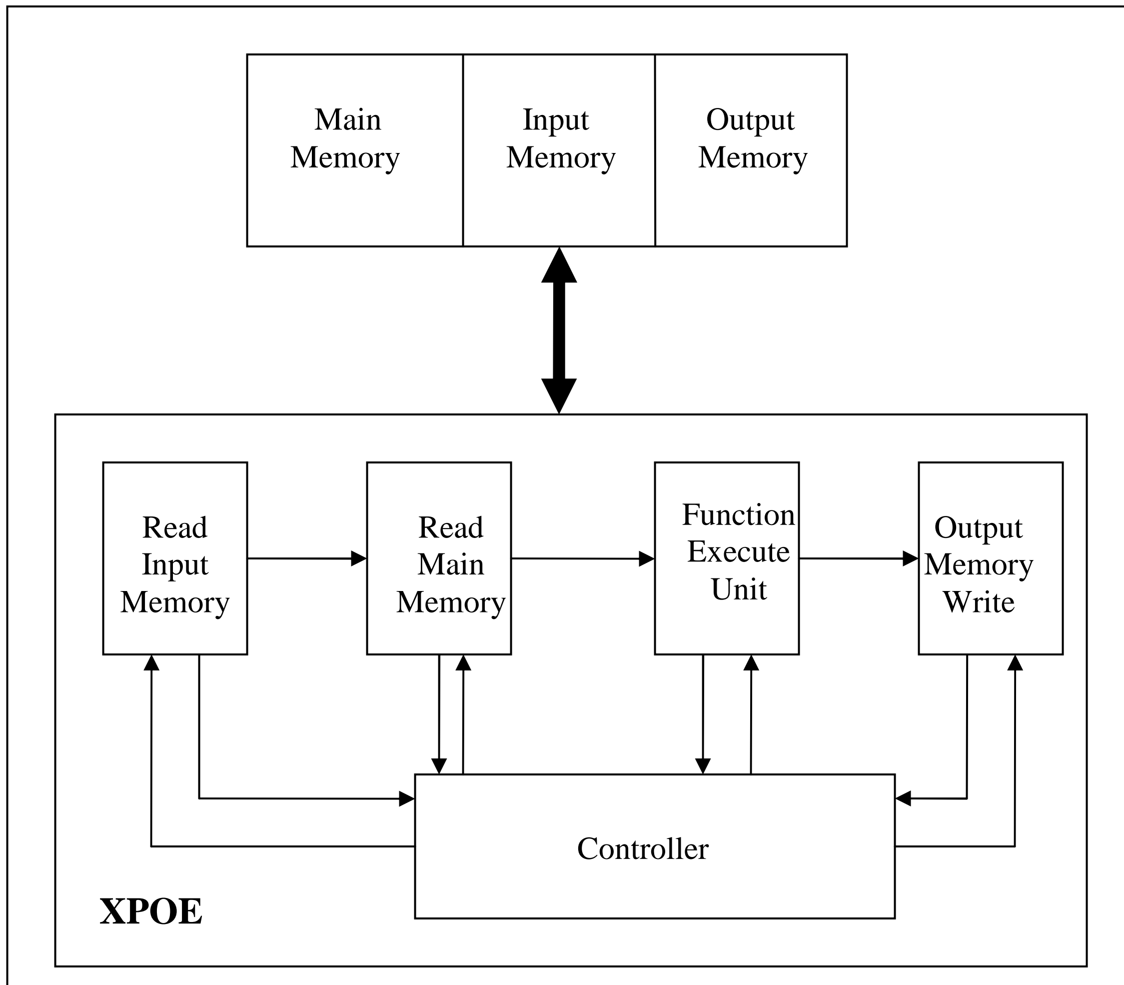

Figure 4.1: Block Diagram of XPOE

4.2.1 Read Input Memory

The following sections list the functions and implementation of the Read Input Memory block.

• Functions

1. Interfaces to the input section of the memory on the input side.
2. Examines the bytes stored at the input section of the memory by the compiler.
3. Reads the bytes from the input section till end of valid byte stream is encountered.
4. Passes the read input bytes to the read main memory block.

• Implementation

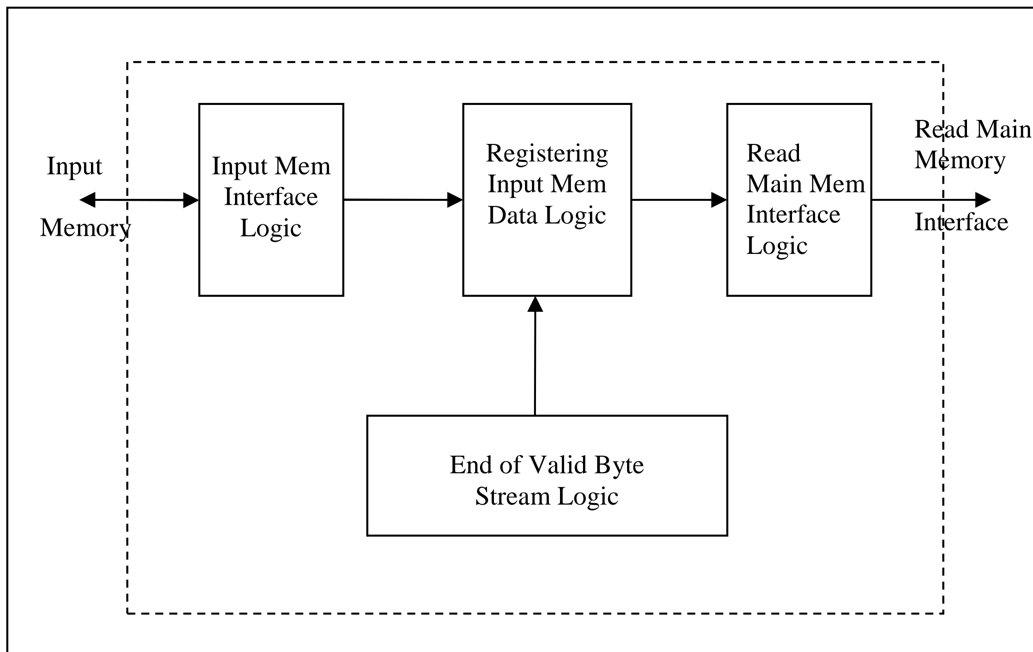

Figure 4.2: Read Input Memory Implementation Diagram

The Read Input Memory module is the interface between the input section of the memory and the rest of the XPOE. When the XPOE is interrupted by the CPU with a start signal the read input memory module will interpret the opcode received from the CPU and appropriately read the required number of bytes from the input section of the memory. This module will continue reading bytes from the input section till an end of stream flag is encountered.

The start interrupt from the CPU will enable this module and when end of valid byte stream is encountered this module would have read all the required bytes from the input section of memory. These bytes are now ready to be passed to the Read Main Memory module for further processing. The end of valid byte stream flag is used because based on the opcode the number of input arguments and hence the number of bytes to be read from the input section of memory will vary.

4.2.2 Read Main Memory

The following sections list the functions and implementation of the Read Main Memory block.

• Functions

1. Interfaces with the Read Input Memory module on the input side and with the Function Execution unit on the output side.
2. Based on the opcode this module identifies which function is to be executed.
3. As per the required function bytes from the input module are used appropriately as input argument pointers and associated lengths.
4. Reads the main memory section with these pointers.
5. Delivers the input argument bytes to the Function Execute module.

• Implementation

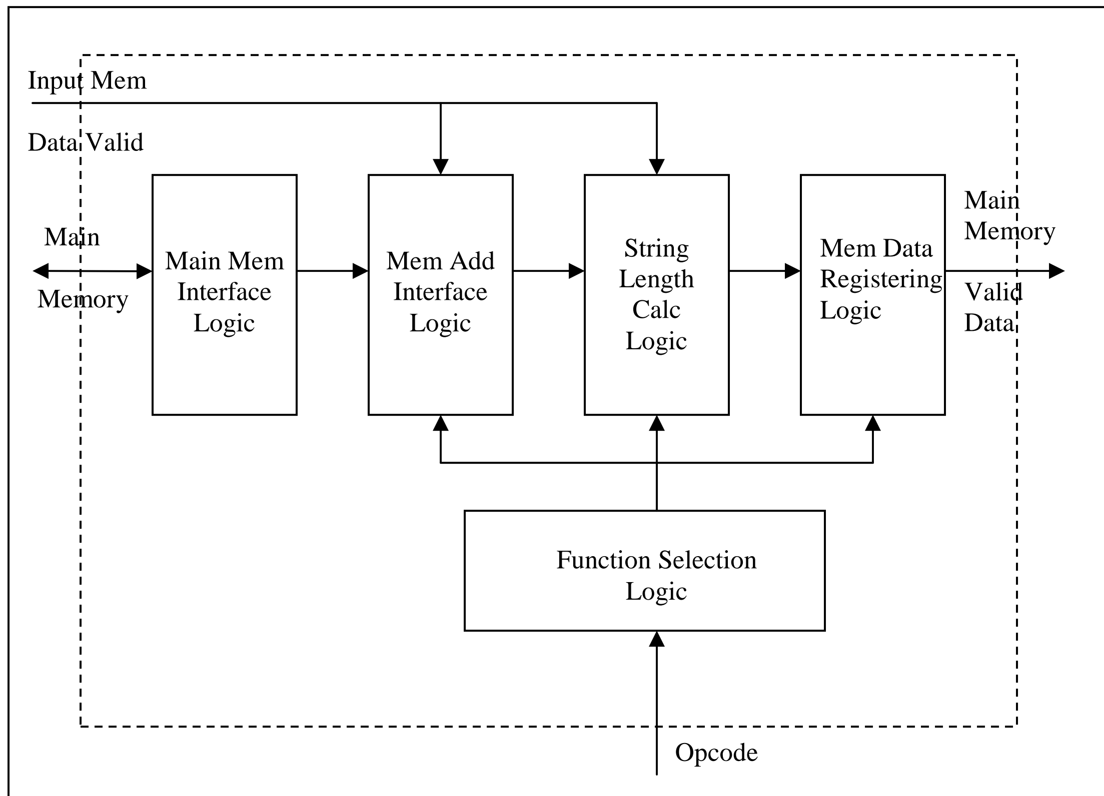

Figure 4.3: Read Main Memory Implementation Diagram

This module receives the bytes read by the Read Input Memory module. Based on the function corresponding to the opcode it interprets the bytes read. It identifies the string pointers and its corresponding lengths and reads from the appropriate location in main memory byte by byte. It then delivers the input arguments to the function byte by byte to the Function Execute module.

4.2.3 Function Execute

The following sections list the functions and implementation of the Function Execute block.

• Functions

1. Interfaces with the Read Main Memory module on the input side and with the Memory Write module on the output side.
2. Based on the opcode this module decides which function to implement.
3. If required it identifies the bytes of the input memory data other than string pointers that might be needed for function execution.
4. It executes the appropriate function block on the input arguments passed to it by the Read Main Memory module.
5. This module will deliver the output of the function execution, which is the data to be written to memory to the Memory Write module.
6. It indicates whether a particular data value that is the output of the execution needs to be written to the output memory or not.

• Implementation

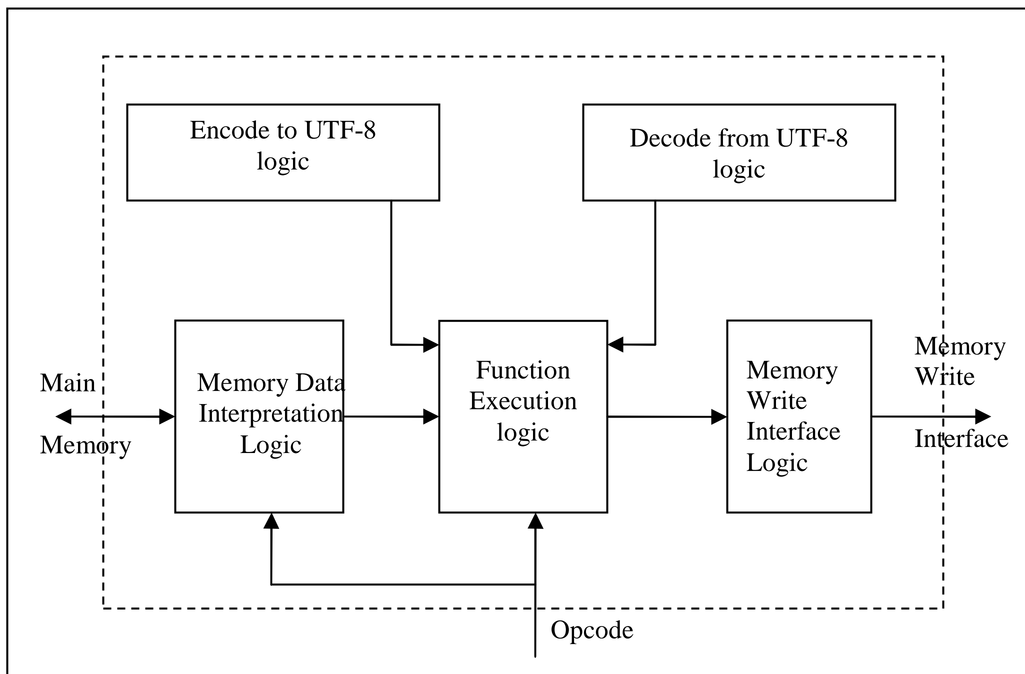

Figure 4.4: Function Execute Implementation Diagram

This module implements the operation of the actual XPath function based on the opcode that is passed to the accelerator by the CPU. The input to this module is from the Read Main Memory module. The input arguments are read from the main memory byte by byte and the Function Execute module operates on these inputs. The output of this operation is character data that may or may not need to be written to the output section of the memory. This data value is passed to the Memory Write module.

A significant percentage of the CPU overhead savings is affected by this module. This module implements the encoding to and decoding from UTF-8, both of which consume the maximum number of CPU cycles.

4.2.4 Memory Write

The following sections list the functions and implementation of the Memory Write block.

• Functions

1. Interfaces with the Function Execute module on the input side and with the output section of the memory on the output side.
2. It receives the character data to be written to memory from the Function Execute module.
3. Writes the input data value to the output section of the memory
4. When the writing of the result has been completed this module will also write the pointer to the result string and its length at the beginning of the output section.
5. Indicates when all writing has completed letting the CPU know that the result is ready to be read.

• Implementation

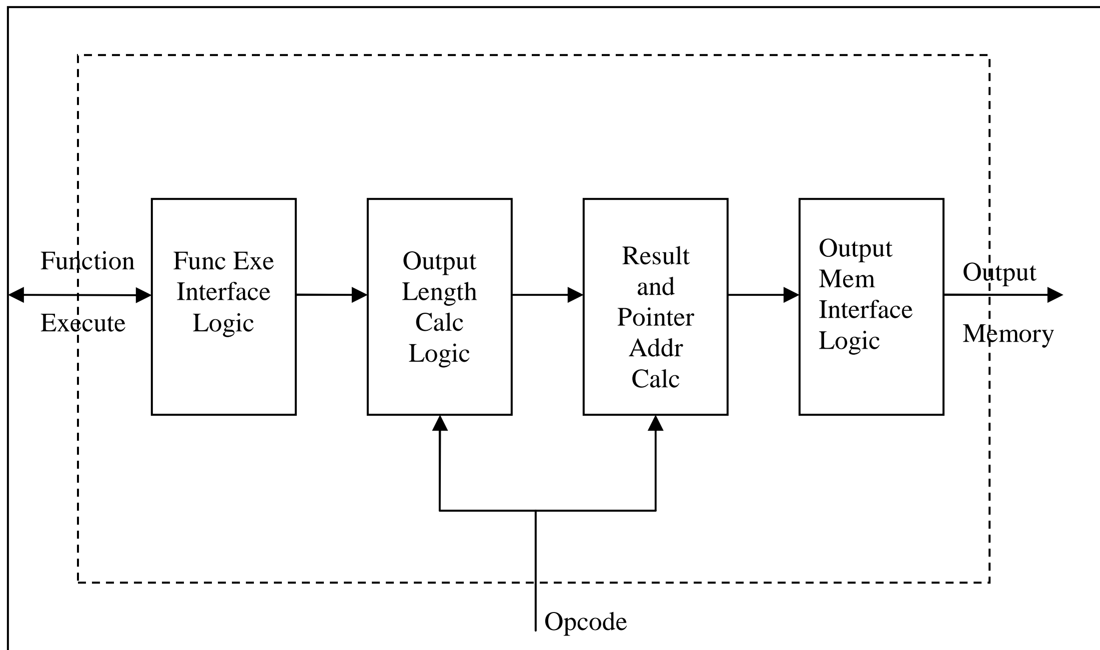

Figure 4.5: Memory Write Implementation Diagram

This module writes the result word by word into the output section of the memory. When the entire result string has been written, a pointer to the start address of the string along with the length of the result string are written into the start of the output section of the memory. Once that is done a complete signal goes high and wakes up the CPU, which will then read the result from the predefined location of the output section of memory. The CPU software can easily use the index and length fields to access the value.

4.3 Design Pipeline

The input can be considered as a continuous flow of input functions / opcodes. One approach to this design could be to first completely buffer the incoming functions, as in all the pointers to the input arguments, then process it, and then

wait for the next group of functions. In this case, there could be a situation where we would need to exert backpressure on the input. To avoid this scenario, we take a pipelined approach. In this approach there is no need to exert any backpressure and we achieve better throughput.

The above sections have discussed the main modules of the XPOE from a block level view by talking about the functionality and basic implementation. Chapter 5 will describe these modules in greater detail, such as, pin interfaces, timing waveforms and detailed architecture.

Chapter 5

Design Module Description

5.1 Introduction

In section 4, we had a top-level look at block-level architecture of the design. We saw the functions of the blocks and had a brief idea of how each block is implemented. The current section aims to provide more details. For each block, the pin interfaces with other blocks and a detailed implementation description are given. This would include the FSM, data structures, arithmetic units and specific logic involved.

5.2 Module Read Input Memory

This section and its subsection sections present a detailed description of the Read Input Memory module. Section 5.2.1 presents the pin interfaces with blocks it interacts with. Section 5.2.2 describes its implementation in detail.

5.2.1 Pin Interface

This section describes the pin interfaces of the Read Input Memory block with the other blocks.

Table 5.1: Interface with the system

No.	**Pin Name**	**Dirn.**	**Width**	**Description**
1	clk	IN	1	Async system reset
2	reset	IN	1	System clock

Table 5.2: Interface with the input section of the memory

No.	**Pin Name**	**Dirn**	**Width**	**Description**
1	enable_inputread	IN	1	Indication to start reading Input memory. This signal is asserted when the start interrupt from CPU is issued.
2	ReadBus1	IN	32	Data read from Input section of the memory.
3	ReadAddress_IPMem	OUT	32	Address generated for reading the Input section of memory.

Table 5.3: Interface with the Read Main Memory module

No.	**Pin Name**	**Dirn**	**Width**	**Description**
1	Inputread_done	OUT	1	Indication that Input memory has been read and data is valid. A signal for Read Main Memory module to use this data to read the Main memory.
2	Inputmem_data	OUT	128	Data read from Input section of the memory.

5.2.2 Architecture

This section starts by discussing the interface timing between the Read Input Memory module and the blocks it interfaces with. It then proceeds to elaborate on the hardware implementation of the blocks in the logic schematic shown in section 4.2.

• Input Memory Interface Timing

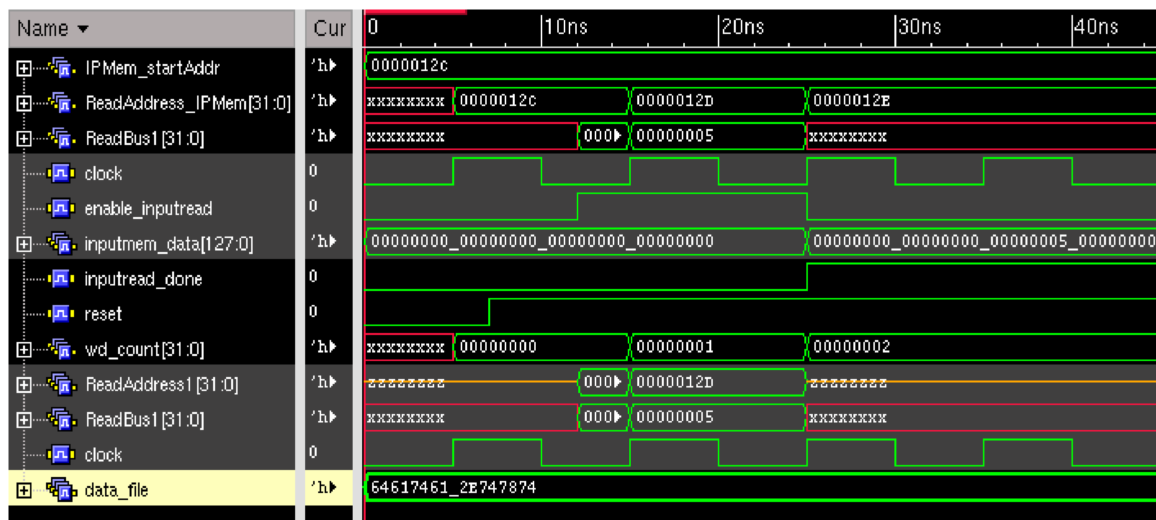

Figure 5.1: Interface waveform for Input Memory

Figure 5.1 shows the behavior between the Read Input Memory module and the input memory. When the start interrupt signal from the CPU is sampled high the data from the input section of the memory is read into an internal register. As each data unit is read from the input memory it is first checked for the end of valid byte stream sequence. As long as the end of byte stream has not been reached, on each positive edge of the clock one word is read from the input section of the memory and is registered. Note that for this data to be valid the start signal should be high and the end of valid stream should not have been reached. Upon reaching the end of valid byte stream inputread_done is asserted

high. By this point all the data that was read from the input memory would have been registered and ready to be delivered to the Read Main Memory module.

• Timing Waveform for internal counters and output interfaces

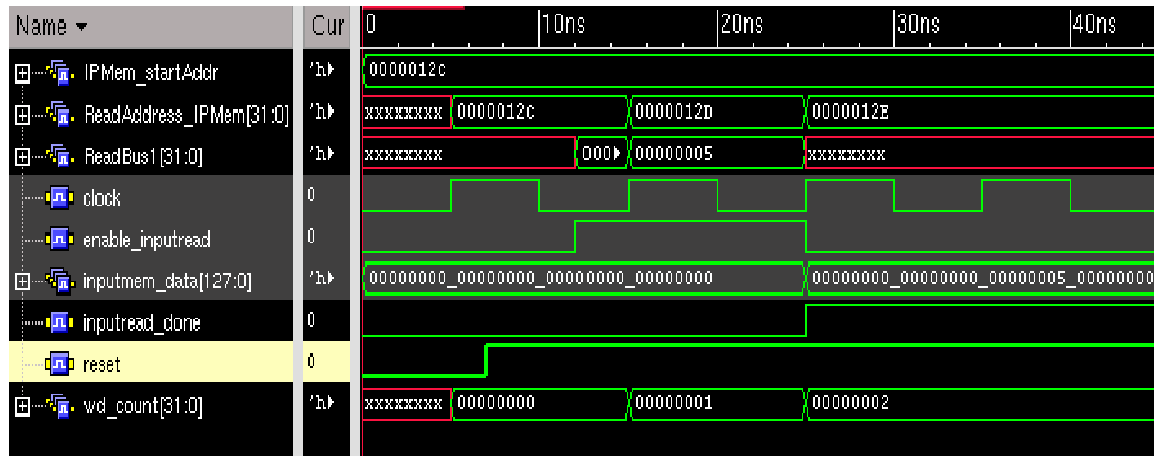

Figure 5.2: Internal counters and Read Main Memory interface timing

Figure 5.2 above shows the timing waveform describing the behavior of the output interfaces and the interface with the Read Main Memory module.

5.3 Module Read Main Memory

This section and its subsection sections present a detailed description of the Read Main Memory module. Section 5.3.1 presents the pin interfaces with blocks it interacts with and section 5.3.2 describes its implementation in detail.

5.3.1 Pin Interface

This section describes the pin interfaces of the Read Main Memory block with the other blocks.

Table 5.4: Interface with the system

No.	Pin Name	Dirn.	Width	Description
1	Clk	IN	1	Async system reset
2	Reset	IN	1	System clock

Table 5.5: Interface with the Read Input Memory module

No.	Pin Name	Dirn	Width	Description
1	enable_readmm	IN	1	Indication to start reading Main memory. This signal is asserted when the Input memory data is valid, i.e. when inputread_done goes high.
2	inputmem_data	IN	128	Data read from Input section of the memory. To be interpreted by the Read Main Memory module as pointers to input arguments in Main memory and their corresponding lengths.

Table 5.6: Interface with the Main Memory module

No.	Pin Name	Dirn	Width	Description
1	ReadBus1	IN	32	Data read from the Main memory section from the first memory read bus.
2	ReadBus2	IN	32	Data read from the Main memory section from the second memory read bus.
3	ReadAddress1_MMem	OUT	32	Address generated from the Input memory data for reading the Main memory
4	ReadAddress2_MMem	OUT	32	Address generated from the Input memory data for reading the Main memory

Table 5.7: Interface with the Function Execute module

No.	Pin Name	Dirn	Width	Description
1	readmm_done	OUT	1	This signal is asserted when the Read Main Memory module has completed reading all input arguments from Main Memory for that particular function.
2	readchar_done	OUT	1	Indication that a word has been read from Main memory and is ready to be operated on by the Function Execute module.
3	mmchardata1	OUT	32	Data read from Main memory. To be processed by the Function Execute module.
4	mmchardata2	OUT	32	Data read from Main memory. To be processed by the Function Execute module.

5.3.2 Architecture

This section starts by discussing the interface timing between the Read Main Memory module and the blocks it interfaces with. We start with the timing diagram between the Read Input Memory and the Main Memory input interface with the Read Main Memory module. This is then followed by the timing diagram with the Function Execute module. It then proceeds to elaborate on the hardware implementation of the blocks in the logic schematic shown in section 4.2.

• Input Interface Timing with the Main Memory and Read Input Memory module

Figures 5.3 and 5.4 shows the timing relation between the Read Main Memory module and the Main Memory and Read Input Memory module. The main memory is read only after 'enable_readmm' goes high. Based on the function specified by the opcode the appropriate word from the byte stream passed from

the Read Input Memory is used as a pointer to the input argument located in main memory. At every successive positive edge of the clock values of the input arguments from the main memory are read until all required entries read as per function requirements. A character counter is maintained. After each character is read from main memory, the count is compared with string length ‘strlen’, obtained from the input memory, as this indicates end of input string has been reached.

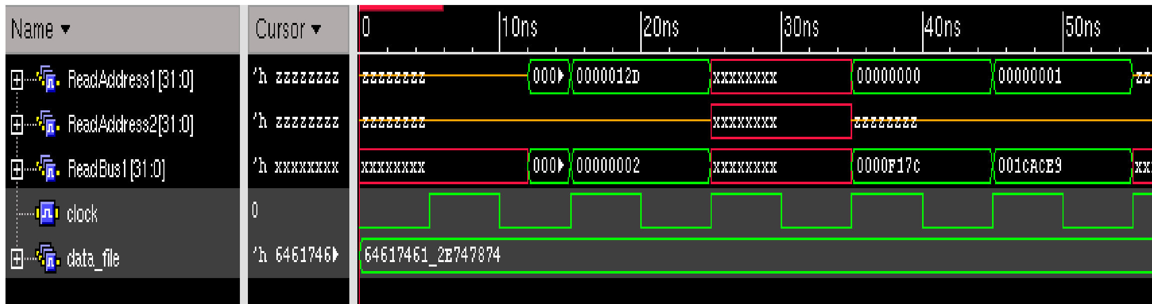

Figure 5.3: Timing relation with Main Memory

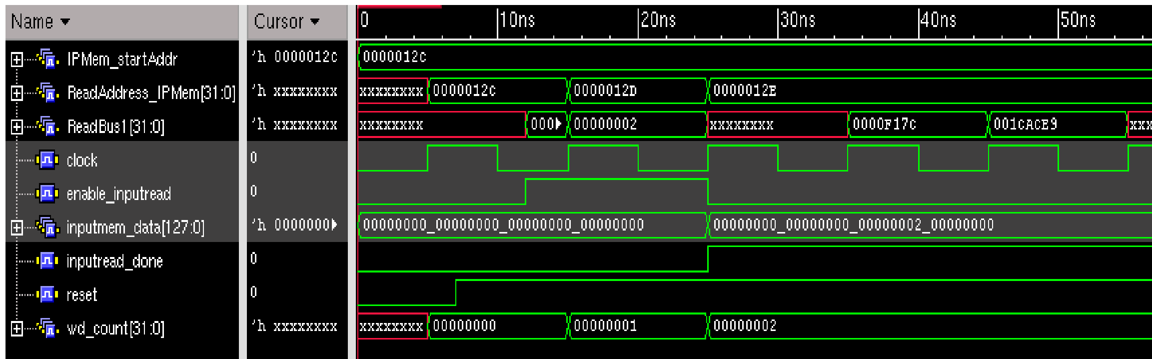

Figure 5.4: Timing relation with Read Input Memory Interface

• Timing Waveform for internal counters and output interfaces

Figure 5.5 shows the timing waveform describing the behavior of the output interfaces and the interface with the Function Execute module. Each word that is read from the main memory is passed over to the Function Execute module.

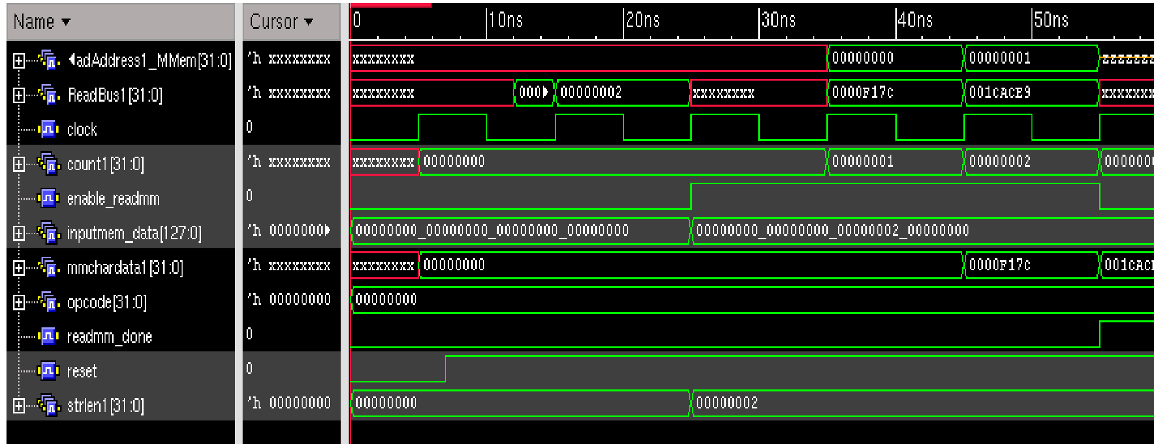

Figure 5.5: Internal counters and Function Execute Interface timing

5.4 Module Function Execute

This section and its subsection sections present a detailed description of the Function Execute module. Section 5.4.1 presents the pin interfaces with blocks it interacts with and section 5.4.2 describes its implementation in detail.

5.4.1 Pin Interface

This section describes the pin interfaces of the Function Execute block with the other blocks.

Table 5.8: Interface with the system

No.	Pin Name	Dirn.	Width	Description
1	Clk	IN	1	Async system reset
2	Reset	IN	1	System clock

Table 5.9: Interface with the Read Main Memory module

No.	Pin Name	Dirn	Width	Description
1	enable_funcexe	IN	1	Indication that a word has been read from Main memory and is ready to be operated on by the Function Execute module. Is asserted when readchar_done goes high.
2	mmchardata1	IN	32	Data read from Main memory, to be processed by the Function Execute module.
3	mmchardata2	IN	32	Data read from Main memory, to be processed by the Function Execute module.

Table 5.10: Interface with the Memory Write module

No.	Pin Name	Dirn	Width	Description
1	funcexe_done	OUT	1	This signal is asserted when the Function Execute module has completed execution for that particular function.
2	mem_write	OUT	1	Indicates whether a particular word needs to be written to memory or not.
3	DataToMem_ready	OUT	1	When asserted this signal indicates that the processed data is ready to be written into the output section of the memory by the Memory Write module.
4	DataToMem	OUT	32	Data ready to be written to Output memory, after being processed by the Function Execute module. Delivered to the Memory Write module.

5.4.2 Architecture

This section starts by discussing the interface timing between the Function Execute module and the blocks it interfaces with. We start with the timing diagram between the Read Main Memory input interface with the Function Execute module. This is then followed by the timing diagram with the Memory Write module. It then proceeds to elaborate on the hardware implementation of the blocks in the logic schematic shown in section 4.2.

• Input Interface Timing with the Read Main Memory module

Figure 5.6 shows the timing relation between the Function Execute module and the Read Main Memory module. The functional unit is enabled to operate on the input character data only after 'enable_funcexe' goes high. Based on the opcode input by the CPU the appropriate function is implemented. The encode to and decode from UTF-8 modules are called as required by the functions.

Figure 5.6: Timing relation with Read Main Memory Interface

• Timing Waveform for internal counters and output interfaces

Figure 5.7 shows the timing waveform describing the details of the behavior of the output interface with the Memory Write module. After each character has

been appropriately operated on, 'DataToMem_ready' will go high to indicate that the result of the operation is ready to be written to the output section of the memory. After the functional unit has finished execution 'funcexe_done' will go high.

Figure 5.7: Internal counters and Memory Write interface timing

5.5 Module Memory Write

This section and its subsection sections present a detailed description of the Memory Write module. Section 5.5.1 presents the pin interfaces with blocks it interacts with and section 5.5.2 describes its implementation in detail.

5.5.1 Pin Interface

This section describes the pin interfaces of the Memory Write block with the other blocks.

Table 5.11: Interface with the system

No.	Pin Name	Dirn.	Width	Description
1	Clk	IN	1	Async system reset
2	Reset	IN	1	System clock

Table 5.12: Interface with the Function Execute module

No.	Pin Name	Dirn	Width	Description
1	funcexe_done	IN	1	This signal is asserted when the Function Execute module has completed execution for that particular function.
2	enable_mem_write	IN	1	This signal is asserted when DataToMem_ready goes high. Indicates that the word delivered by the Function Execute module needs to be written to Output memory.
4	DataToMem	IN	32	Data ready to be written to Output memory, after being processed by the Function Execute module. Delivered to the Memory Write module.

Table 5.13: Output Interface of the Memory Write module

No.	Pin Name	Dirn	Width	Description
1	Complete	OUT	1	This signal is asserted when the XPOE has completed all operations and has finished writing into Output memory. It is an indication to the CPU to take over.
2	WE	OUT	1	This signal is asserted when data to be written to output memory is ready. Enables writing into memory.
3	WriteBus	OUT	32	Contains the word to be written to Output memory.
4	WriteAddress	OUT	32	Output Memory Address generated to write each result word into the output section of the memory.

5.5.2 Architecture

This section starts by discussing the interface timing between the Memory Write module and the blocks it interfaces with. We start with the timing diagram between the Memory Write module and its interfaces. Its interface with the Function Execute module and the Memory Write module's output interface. It then proceeds to elaborate on the hardware implementation of the blocks in the logic schematic shown in section 4.2.

• Input Interface Timing with the Function Execute module

Figure 5.8 shows the timing interface between the Memory Write module and the Function Execute module. The Memory Write module unit is enabled only after 'enable_mem_write' goes high. Based on the counter value the character data passed from the Function Execute unit is written to the appropriate memory location in the output section of the memory.

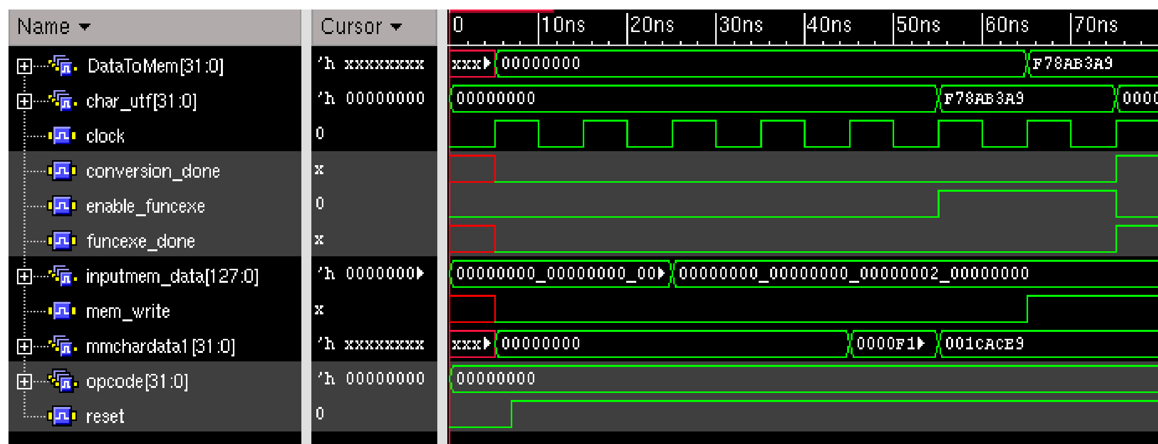

Figure 5.8: Timing relation with Function Execute Interface

• Timing Waveform for internal counters and output interfaces

Figure 5.9 shows the timing waveform describing the details of the Memory Write module's output Interface. When the writing of the result into the output

memory section has been completed, along with writing the pointer to the start of the result and its associated length, the ‘complete’ signal is asserted high to indicate that the accelerator has completed it operation and it’s time for the CPU to take over.

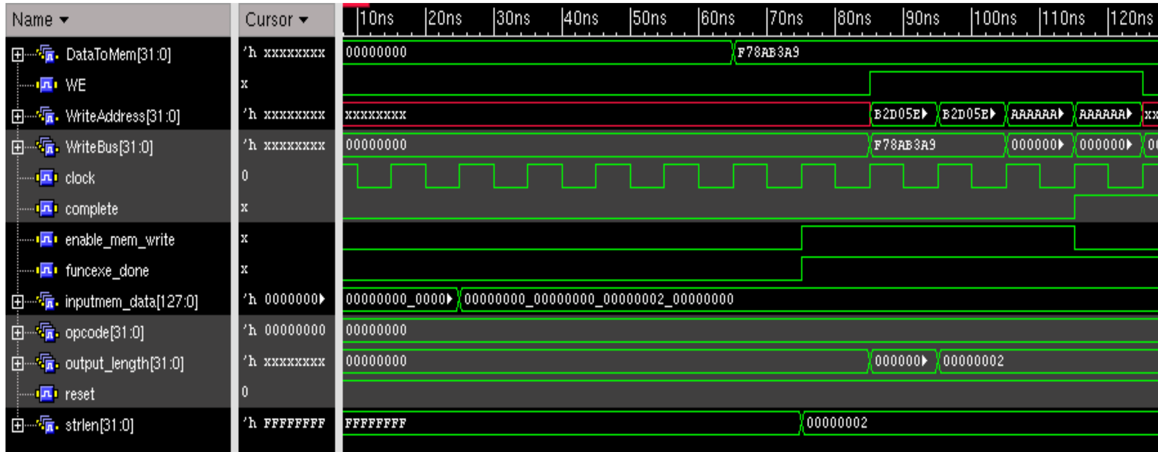

Figure 5.9: Internal counters and output interface

5.6 Controller

This module implements the Finite State Machine (FSM). This section begins with a flowchart, depicting the way the logic proceeds in the design. This is followed by the State Machine Diagram, shown in Figure 5.11 and its transition table.

The overall flowchart for the design is shown in Figure 5.10. Each block in the flowchart can be thought of as a module.

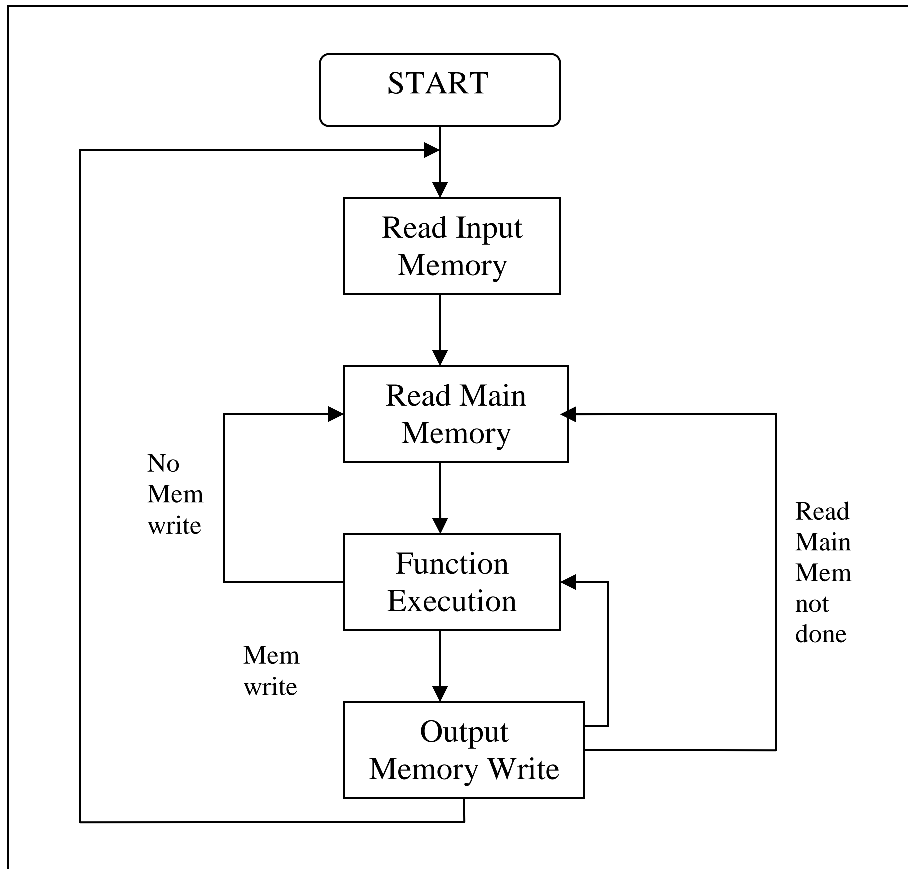

Figure 5.10 Flowchart for XPOE

Figure 5.11 shows the state machine for the XPOE. When the start interrupt goes high the FSM enables the read input module. Based on signals from the read main memory, function execute and memory write modules, the FSM will enable the appropriate modules. When the complete signal goes high indicating that the XPOE has finished execution the CPU will be interrupted. Based on the current state, various counters and interfaces are driven.

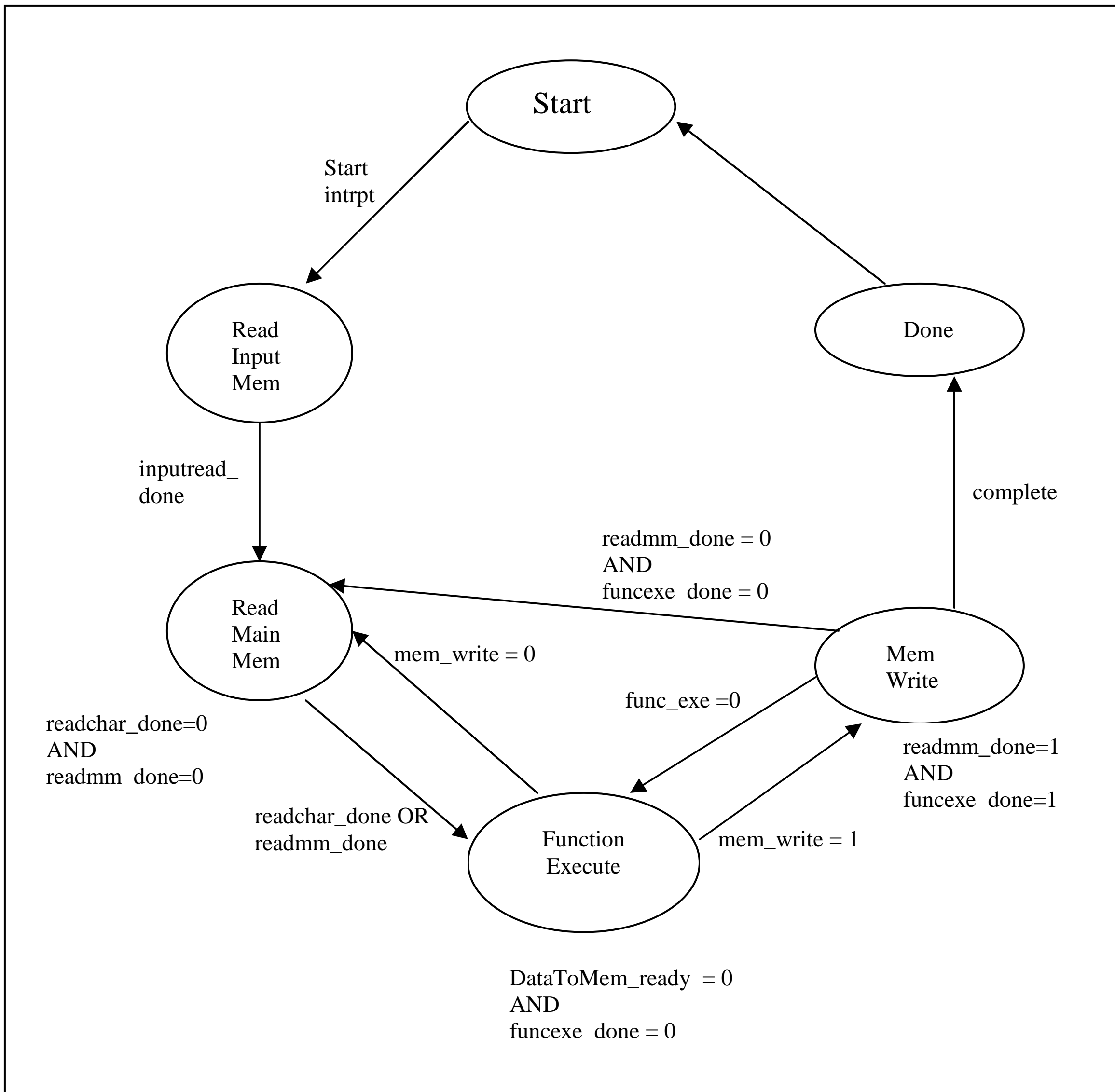

Figure 5.11 State Machine Diagram

The next chapter discusses the verification strategy in detail.

Chapter 6

Software Simulation and Verification

This chapter describes the test environment developed, the testbench components and the verification strategy. It also talks about the additional design features.

6.1 Test Environment Description

• Block Diagram

Figure 6.1 shows the architectural block diagram of the testbench.

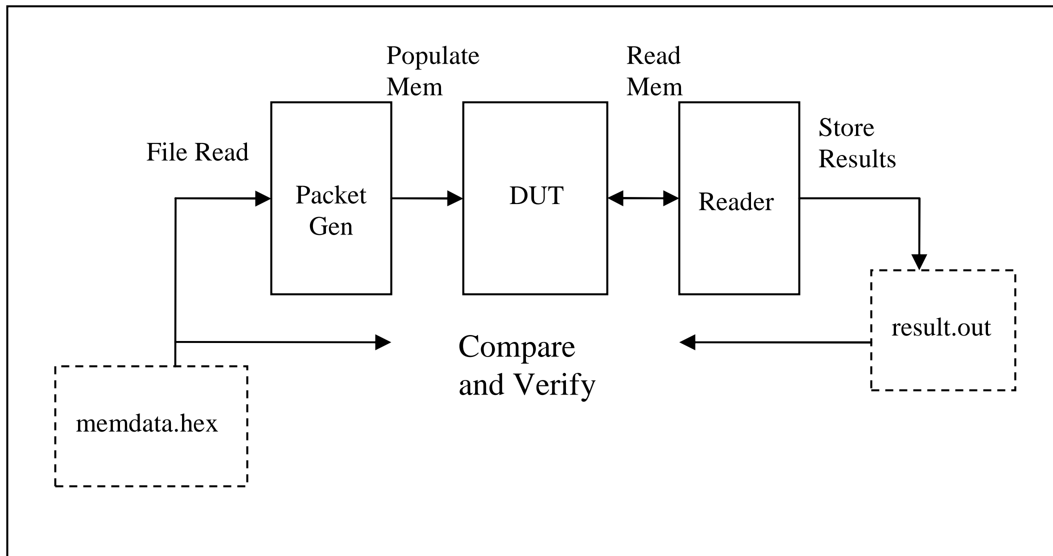

Figure 6.1: Block diagram of testbench

• Inputs

A black-box approach is taken to verify the Device-Under-Test (DUT).

• Outputs

The DUT reads from the main and the input section of memory and after executing the appropriate function as per the opcode it populates the output section of the memory.

After this, it asserts a ’complete’ indication to the external world, indicating that the result of the requested XPath function is ready to be read out from the output section of the memory. The testbench will then read the output memory and outputs a text file result.out. The contents of this result file are formatted and can be examined for correctness of the DUT operation.

• Components

The main components of the testbench are briefly described below:

1. Packet Generator

The Packet Generator loads all the bytes in the input file memdata.hex into the memory.

2. Reader

The Reader reads out all locations of the output section of the memory, formats the data to increase readability and writes to the file result.out.

6.2 Testbench Architecture

This section describes the testbench components in greater detail.

6.2.1 Module Packet Generator (pktgen)

- Pin Interface

Table 6.1: Interface with the system

No.	Pin Name	Dirn.	Width	Description
1	Clk	IN	1	Async system reset
2	Reset	IN	1	System clock

Table 6.2: Interface with the DUT

No.	Pin Name	Dirn.	Width	Description
1	start_int	IN	1	Indicates to the DUT that it can start reading from the Input section of the memory.
2	Opcode	IN	32	Function opcode indicating which specific function implementation has been requested.

The Packet Generator reads in all the data from the input file and populates the memory with this data. The memory width is equal to 4 bytes, as UTF-8 uses a maximum of 4 bytes for character representation.

The pin 'start_int' is then asserted to make data available to the DUT. An opcode value is also fed which will correspond to the particular XPath function under test. When the DUT read all the bytes from the input section of memory and reached the end of the valid byte stream the inputread_done signal is asserted. The DUT will then use the data read from the input memory as pointers to the input arguments located in main memory, and these arguments are read byte by byte and supplied to the remaining blocks of the DUT.

Once all the input arguments are read from main memory or the execute unit has sufficient information to form the output of the function under test the DUT will assert its 'complete' signal.

6.2.2 Module Reader (reader)

- Pin Interface

Table 6.3: Interface with the system

No.	Pin Name	Dirn.	Width	Description
1	Clk	IN	1	Async system reset
2	Reset	IN	1	System clock

Table 6.4: Interface with the DUT

No.	Pin Name	Dirn.	Width	Description
1	Complete	IN	1	Indication that the DUT is done execution and that the result is now ready in the Output Memory.
2	read_enable	OUT	1	Enable/request signal

Table 6.4 continued: Interface with the DUT

No.	Pin Name	Dirn.	Width	Description
3	read_address	OUT	32	The address that is generated to perform the read from the memory.
4	read_data	OUT	32	The data read from the Output Memory.
5	read_valid	OUT	1	Read data is validated when this signal is asserted.

• Architecture

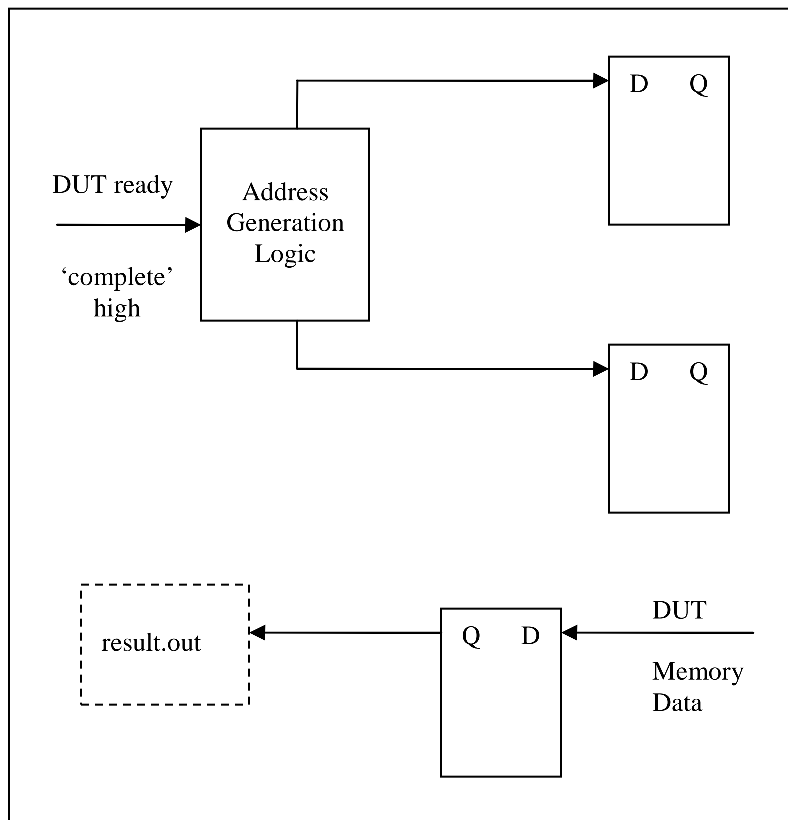

Figure 6.2: Architectural Block Diagram of the Reader Module

The Reader module has to wait for the DUT structure done indication, i.e. the 'complete' signal, before it can start to read the output section of the memory. Once this handshake is asserted, it starts reading every location of the output memory, and formats the data received and writes it to the file result.out. This file is examined to verify DUT functionality.

6.3 Verification Test Plan

The verification test plan is described in this section. The goal is to test the DUT using a set of input script files. Each version of the input file memdata.hex represents a test case/script for a different XPath function that is being tested. The DUT can be tested for a set of features. Typically, one input script would test these features/functions, depending on the contents of the input file. These functions, in no particular order, can be listed as:

1. CPU – ASIC Interaction
   - Start interrupt
   - Complete signal
   - Reset
2. Individual Block Testing
   - Reading Input data from the input memory
   - Reading Input arguments from the main memory
   - Writing the result to the output section of memory
   - Result string length calculation
   - Encoding, decoding and string compares
   - Which Output signals are activated corresponding to changing function/opcode
3. Functional Verification
   - For each of the functions implemented
   - Encoding to UTF-8 from character number

- Decoding from UTF-8 character string
- codepoints-to-string function
- string-to-codepoints function
- codepoint-equal function
- substring function
- starts-with and ends-with functions

4. Controller Operation
    - No illegal states visited
    - The states visited for different functions
    - Number of states visited

6.3.1 Feature Tests

To test some/all features of the DUT, feature tests are employed. The scripts that are run on the DUT test all the features listed. These tests are run with one XPath function contained in one script. The tests are rerun for each and every one of the XPath function that is implemented.

Table 6.5: Feature tests

No.	Script Name	Features covered from the list
1	Memdata1.hex	1,2,3,4 and all subsections
2	Memdata2.hex	1,2,3,4 and all subsections

6.4 Test Results Summary

Table 6.6 summarizes the tests run on a PASS/FAIL basis.

Table 6.6: Summary table of the test results

No.	Category	Script Name	Result/Comments
1	Feature tests	memdata1.hex	PASS
2	Feature tests	memdata2.hex	PASS

6.5 Additional Design Features

- Clock Period: 38.25 ns
- Maximum operating frequency: 26.14 MHz
- Total equivalent area for design: 502,301
- Total data throughput supported: 1.7 Gbps

Chapter 7

Future Work and Conclusion

This chapter describes the future scope of the work presented in terms of feature additions and optimizations to the existing design.

7.1 Feature Additions

This section talks about the features that could be added to the existing design for further improvements.

1. We could look into other XPath functions, other than string functions which might give us similar CPU cycle savings.

2. We could offload more XPath string functions onto hardware and calculate the tradeoff.

3. We could optimize the compiler, thus optimizing XPOE's interactions with the system.

4. We could look into offloading some of the pre-processing also onto hardware.

5. We could extend the encode and decode offload units to include the UTF-16 representation.

6. We could look into incorporating these XPath functions into Xquery expressions and thus offload aspects of Xquery onto hardware.

7. We could extend the implementation to include all collations.

7.2 Verification

Enhanced testing capabilities could be added to the existing test environment.

- Randomization:

The existing test environment can be randomized to generate different combinations of XPath functions that follow. The input arguments to the function could also be randomly generated before storing in the input section of the memory.

- Introduce Error Conditions:

Different error conditions could be created while data is stored in the memory and then the functionality of the DUT could be tested. For example, errors introduced could be on the lines of specifying incorrect number of input arguments, omitting end of valid byte stream, incorrect UTF representation, etc.

- Assertions:

  The existing test environment verifies the correctness of the design by examining the output of the XPOE which is the result of the function after the XPOE has completed execution and the result is ready. However, assertions could be used within the code, to catch bugs earlier, as and when they occur in the testing phase.

7.3 FPGA Implementation and Architecture Optimizations

This section talks about improvements to the architecture such area and timing optimizations of the FPGA implementation.

7.3.1 Area Optimizations

An FPGA consists of an array of configurable logic blocks (CLBs) and routing channels. A typical FPGA logic block consists of a 4-input lookup table (LUT), and a D flip-flop. There is only one output, which can be either the registered or the unregistered LUT output. The logic block has four inputs for the LUT and a clock input. Thus, the LUT can implement a 4-input function. Area optimizations can be achieved when the LUT is used as a shift register or a register array. Advanced place-and-route techniques and FPGA primitives can also be used to reduce area. Techniques such as register packing and register placements can pack more logic together thus optimizing area.

7.3.2 Timing Optimizations

Timing optimizations can be done in several ways. Advanced place-and-route algorithms along with the FPGA Floorplanner can be used for register packing, register retiming and placing communicating modules closer together. Critical

paths are dominated by interconnect delay and are frequently highly circuitous. Such paths can be ‘straightened’ out using advanced techniques. Optimize all short- and long-path timing constraints in an FPGA. These methods are among a few that could lead to better timing.

7.4 Conclusion

The correctness of the design is proved through functional verification, and the analysis of the hardware offload approach demonstrates the CPU cycles saved. Hence, it can be concluded that it is desirable and possible to offload XPath function processing onto hardware.

Bibliography

[1] *Extensible Markup Language (XML) 1.0.* Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, W3C Recommendation 16 August 2006.

[2] *XML Path Language (XPath) 2.0.* Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, Jérôme Siméon, W3C Recommendation 23 January 2007.

[3] *XQuery 1.0: An XML Query Language.* Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, W3C Recommendation 23 January 2007.

[4] *Acceleration Techniques for XML Processors.* Biswadeep Nag, XMLConference 2003.

[5] *FPGA Acceleration of Information Management Services.* Richard Linderman, Mark Linderman, Chun-Shin Lin, Air Force Research Laboratory Information Directorate, 2005

[6] *TCP offload to the rescue.* A. Currid, ACM Queue, 2004.

[7] “Time for toe,” Chelsio Communications, Tech. Rep.

[8] *XML Accelerator Engine.* Jan van Lunteren, Ton Engbersen, Joe Bostian, Bill Carey, Chris Larsson, First International Workshop on High Performance XML Processing, 2004,

[9] *XML Parsing: A Threat to Database Performance.* Matthias Nicola, Jasmi John. CIKM'03, November 3-8, 2003, New Orleans Louisiana 2003.

[10] *How Much Pain for XML's Gain?* Michael Champion, XML 2004 Proceedings by SchemaSoft.

[11] *FPGA Implementation of a SIP Message Processor* Raja Nimmelapelli, Dissertation, NCSU, 2006.

[12] Augmented BNF for Syntax Specifications: ABNF. Request for Comments 2234, 1997.

[13] UTF-8, a transformation format of ISO 10646. Request for Comments 3629, 2003.

[14] UTF-16, an encoding of ISO 10646. Request for Comments 2781, 2000.

[15] Virtex-2 Platform FPGA User Guide, 2005.