Text Preprocessing
in
Programmable Logic

by

Michał Jan Skiba

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

There is a tremendous amount of information being generated and stored every year, and its growth rate is exponential. From 2008 to 2009, the growth rate was estimated to be 62%. In 2010, the amount of generated information is expected to grow by 50% to 1.2 Zettabytes, and by 2020 this rate is expected to grow to 35 Zettabytes[IE10]. By preprocessing text in programmable logic, high data processing rates could be achieved with greater power efficiency than with an equivalent software solution[VAM09], leading to a smaller carbon footprint.

This thesis presents an overview of the fields of Information Retrieval and Natural Language Processing, and the design and implementation of four text preprocessing modules in programmable logic: UTF–8 decoding, stop–word filtering, and stemming with both Lovins'[Lov68] and Porter's[Por80] techniques. These extensively pipelined circuits were implemented in a high performance FPGA and found to sustain maximum operational frequencies of 704 MHz, data throughputs in excess of 5 Gbps and efficiencies in the range of 4.332 – 6.765 mW/Gbps and 34.66 – 108.2 μW/MHz. These circuits can be incorporated into larger systems, such as document classifiers and information extraction engines.

Acknowledgements

I would like to acknowledge the contributions of a number of people at Cisco Systems who have gave me valuable guidance and helped contribute to the development of the ideas in this thesis.

Name	**Contribution**
Jason Marinshaw	Granting approval to pursue this thesis.
Steve Scheid	Supporting Jason's decision.
Gerald Schmidt	Support, insight & suggestions.
Mike Rotzin	Support, insight & suggestions.
Suran de Silva	Running the partnership with the University.
Matthew Robertson	Valuable guidance in linking the work with Cisco.
Sandeep H. Raghavendra	Valuable guidance in linking the work with Cisco.
Vinayak Kamat	Support, insight & suggestions.
Larry Lang	Visionary technology application suggestions.
Sridar Kandaswamy	Facilitating a meeting with Satish Gannu.
Satish Gannu	Ideas for integrating the research with Cisco's vision.
Guido Jouret	Support, insight & suggestions.

Dedication

This work is dedicated to several key people who have been instrumental in guiding me through my endeavors and who have instilled courage within me to have *"dreamed big, and worked hard for my successes"*.

First and foremost to my parents for being exceptional role models of accomplishment, perseverance, dedication, passion and talent. To Paul Garnich for seeding my interest in digital electronics and programmable logic. To Bruno Korst for giving me extremely valuable guidance throughout my undergrad at the University of Toronto and for giving me the freedom to pursue my own project. To Professors Jonathan Rose, Parham Aarabi and Paul Chow for giving me the opportunity to challenge myself with difficult research projects. And finally to my supervisor Professor Gordon Agnew, who has given me the freedom to develop in ways that I would have never imagined.

I am extremely privileged to have been motivated by my exceptionally talented and passionate peers, both in Electrical Engineering and beyond: Iyinoluwa Aboyeji, Seyed Ali Ahmadzadeh, Dr. Navid Azizi, Anna Barycka, Maciej Bator, Dr. Tomasz Czajkowski, Alex Doukas, Szymon Erdzik, Carlo Farruggio, Judyta Frodyma, Grzegosz Fryc, Julius Gawlas, Dr. Monica Gawlik, Kasia Kaminska, Bruno Korst, Dr. Sławomir Koziel, Marta Lefik, Dr. Daniel Mazur, Kamil Mroz, Rinku Negi, Aiden O'Connor, Chuck Odette, Dr. Joseph Paradi, Dr. Artur Placzkiewicz, Jakub Polasik, Mubdi Rahman, Dr. Marek Romanowicz, Dr. George Sandor, Krystian Spodaryk, Paul Sulzycki, Dr. Tamara Trojanowska, Nathan Vexler, Urszula Walkowska, Pawel Waluszko and Matthew Willis.

Contents

Appendix

List of Figures

List of Tables

List of Abbreviations

ALUT	Adaptive Look Up Table
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
BMP	Basic Multilingual Plane
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FTP	File Transmission Protocol
Gbps	Gigabits per second
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IR	Information Retrieval
LSB	Least Significant Bit
LUT	Look Up Table
MSB	Most Significant Bit
NLP	Natural Language Processing
NLTK	Natural Language Toolkit
TCP	Transmission Control Protocol
UCS–2	two byte Universal Character Set
UTF–8	8–bit Unicode Transformation Format

1

Introduction

This thesis investigates how to quickly and efficiently preprocess documents in programmable logic for Information Retrieval (IR) and Natural Language Processing (NLP) applications. It presents designs for UTF–8 decoding, stop word filtering, and stemming with both Lovins'[Lov68] and Porter's[Por80] algorithms, and outlines their respective resource and power efficiencies in a modern Field Programmable Gate Array (FPGA). The text documents could be delivered to the circuits via a computer network, which would require them to process the incoming information as a character stream with a high data transfer rate.

A document is taken to mean a collection of words, which could be embodied in an email, a web page, or other discrete collection of text. By preprocessing in programmable logic, high data processing rates could be achieved with greater power efficiency than with an equivalent software solution[VAM09]. This is important since the amount of digital information in the world is growing at an exponential rate. From 2008 to 2009, the growth rate was 62%. In 2010, the amount of generated information is expected to grow by 50% to 1.2 Zettabytes, and by 2020 this rate is expected to grow to 35 Zettabytes[IE10]. Fast and energy efficient processing will be critical to effectively managing such vast quantities of data with a minimal carbon footprint.

Thesis organization

This thesis is organized as follows. Chapter 2 provides background information on the related fields of Information Retrieval, Natural Language Processing, digital logic and character encoding, and points to several related works. An overview of the guiding principles and constraints that are common to the design and implementation of the four circuits overviewed in this thesis are presented in Chapter 3. The following three chapters focus on each circuit individually. Chapter 4 examines UTF–8 decoding and tokenization. Chapter 5 analyzes an approach to stop word filtering, and Chapter 6 compares the design and implementation of two word stemmers; one based on Lovins' algorithm[Lov68] and the other based on Porter's[Por80].

Chapter 7 concludes the thesis by highlight the resource utilization, performance and power efficiency of the four circuits. It also points to some possibilities for interesting future work, such as Latent Semantic Indexing in programmable logic and the indexing of multimedia, such as video. Supplementary material on text encoding, stop word filtering, Natural Language Processing and stemming can be founded in Appendixies A, B, C and D respectively.

2

Background

Custom hardware implementations are generally significantly faster and more power efficient than their software equivalents. These advantages come at the expense of flexibility and higher development cost. The cost can be justified if the algorithm does not undergo frequent adjustments and the economics of scale are leveraged, with the resulting product being either sold to a very large market, as in the case of Intel's CPUs and Nvidia's GPUs, or its costs aggregated over a large number of users, as in the case of Cisco's IP packet routers. In both scenarios, fully custom devices, known as Application-Specific Integrated Circuits (ASICs), are entirely justified. However, a mid–market also exists and it can be economically catered to with implementations in programmable logic, with the most popular family of devices being Field–Programmable Gate Arrays (FPGAs). Furthermore, by being programmable, they facilitate prototyping, require shorter development cycles and allow for quick upgrades when in use. FPGAs are the target devices for all of the circuits presented in this thesis.

Although FPGAs are not as well optimized for maximum operating frequency and minimal power consumption as ASICs are, they have been shown to accommodate significantly accelerated implementations of inherently parallel algorithms in the domains of signal processing and finance. There are relatively few publications on the topic of accelerated text processing in FPGAs. However, given the exponential growth in the amount of digital information in the world[IE10], it is likely that publications on Information Retrieval (IR) and Natural Language Processing (NLP) applications will become more common. This chapter will present an overview of these two fields, as well as outline some of the related implementations in digital logic. It will also present an overview of several common text encoding standards.

2.1 Information Retrieval

IR is a domain that concerns itself with efficient access to large amounts of stored text. It has a well–established history and has reached an initial level of maturity sufficient for deployment in industry and business. Its strongest overlap is with the field of Natural Language Processing (NLP). The definitive commonality for both is the textual material on which they operate. The tools implemented at various levels of abstraction are different since NLP has the diverging concerns of text analysis, representation and generation. Nevertheless, linguistic techniques are being borrowed from NLP to enhance the performance of IR systems.

> The fundamental technique of information retrieval is measuring similarity. A query is examined and transformed into a vector of values to be compared with the measurements taken over the stored documents.[WIZD05]

IR can be broken down into three major sequential components: *(1)* document preprocessing, *(2)* building a document index, and *(3)* query processing and matching to documents. Classical models in IR represent a document by a set of indexed keywords. The governing hypothesis is that the words themselves provide a good indication of the document's contents, the topic, theme, emphasis and so on. Preprocessing involves extracting words from the document and the first two components of this process are *(i)* stripping document formatting and *(ii)* segmenting the document's character stream into *tokens*, which are the fundamental data structures on which subsequent stages operation on. Tokens are often individual words. Section 4.4 provides a more detailed presentation of tokenization, the process of segmenting a stream of characters into tokens.

The preprocessing stage can also attempt to filter out the document's most salient keywords. Content words such as nouns, verbs, and adjectives convey most of the document's semantics. On the other hand, function words such as prepositions, pronouns, and determiners are universal across all documents and have little impact on determining a document's contents. Function words can be filtered out with a *stop–word* list, a concept explored in Chapter 5. Furthermore, some words represent the same underlying concept (eg. *walk*, *walks*, *walking*). They can be *conflated* to their morphological root form with a technique called *stemming*, which removes their affix either based on a set of heuristic rules or with the aide of linguistic analyzers. Two approaches to stemming are explored in Chapter 6. There are inconsistent results in the literature regarding the effectiveness and benefit of both techniques to IR systems in general[Hul96, KMM00]. Nevertheless, they are standard features in search engines such as Google's or IR libraries, such as the Apache Foundation's Lucene[The09b].

The major component in IR is the construction of a document index. The index can be represented as an inverted file, which generally takes the form of a two–column table in which the document's tokens are associated with the location of their occurrence in the document (starting character position). Such a structure can facilitate efficient search and retrieval. The tables on the following page illustrate how the text fragment *"Video technology will bring patients in rural clinics closer to the hospital services they need. It will soon"* would be converted to an inverted index file, both with and without stop–word elimination & stemming. The difference in size between the two tables underlines an important benefit of stop–word filtering and word conflation: index compression, which leads to more efficient memory use. The inverted index can be simplified to only note the number of occurrences of a given word, disregarding character position.

Inverted index file without stemming or stop–word elimination

Vocabulary	Occurrences
soon	107
will	18, 102
it	99
need	92
they	87
services	78
hospital	69
the	65
to	62
closer	55
clinics	47
rural	41
in	38
patients	29
bring	23
technology	7
video	1

Inverted index file with stemming and stop–word elimination

Vocabulary	Occurrences
need	92
service	78
hospital	69
close	55
clinic	47
rural	41
patient	29
bring	23
technology	7
video	1

The major component of IR is processing the search query and matching it with occurrences in the indexed documents. Three statistical matching techniques that are used in the bulk of IR systems are Boolean, vector space and probabilistic models. Boolean is the simplest and most efficient. It requires that the user provide search queries with strict logical operators, such as *and*, *or*, *not*, and it returns documents whose contents satisfy the intersection of these constraints. IEEE Xplore implements this matching technique[Ins10]. Simplicity is also a drawback: the system is not capable of ranking the documents based on the keywords themselves, and more importantly, searches on the basis of word pres-ence rather than concept, the way humans do.

Vector–space models associate non–binary weights with indexed terms, and use them to compute the degree of similarity between documents and queries. This yields a list of ranked results, which can be sorted from strongest match (most relevant) to weakest (least relevant). Important variables for this technique are *term frequency*, which provides a measure of how well the term describes the document contents, and *inverse document frequency*, a measure of how unique the term is to a certain document. An important alternative is Latent Semantic Indexing, which facilitates search based on concept rather than individual index terms. This robust technique has been successfully implemented in a number of commercial applications.

The third statistical matching technique is based on probabilistic models that treat the query as a way to specify properties of an ideal answer set, with the properties being characterized by the semantics of the index terms. The model ranks documents by relevance, but has drawbacks related to search initialization. A more technical look at indexing in programmable logic is presented in Section 7.1.

Performance Measures

Table 2.1 outlines four important text processing performance measures: *precision*, *recall*, *accuracy* and *error rate*. For a system that always assigns all candidates, its precision and recall are the same and performance is measured either in terms of *accuracy*, which is calculated the same way as precision, or in terms of *error rate*[Mik03].

Measure	Calculation
precision	number of correct answers / number of answers produced
recall	number of correct answers / total number of expected answers
accuracy	number of correct answers / number of answers produced
error rate	$(\lvert 1 - accuracy \rvert) \times 100\%$

Table 2.1: Text Processing Performance Measures

For the English language, the two standard corpora typically used for evaluation of text processing tasks are the *Brown corpus*[KFC67] and *Wall Street Journal (WSJ)* corpus. Both contain over one million words and both are included in the Penn Treebank, which contains correct tokenization and part–of–speech information for a total of 4.5 million words[MMS93, MTM$^+$99]. There are a number of large corpora that are not fully annotated and these include the 2 billion word Oxford English Corpus, 400 million word Corpus of Contemporary American English[Dav09] and 100 million word British National Corpus[Bri09].

2.2 Natural Language Processing

Natural Language Processing (NLP) is a field in Computer Science that deals with the generation, manipulation and understanding of natural (human) languages, such as English. The field has substantial overlap with computer language processing. However, the major distinction is that computer languages typically have strict, *formal*, rules, whereas those in human languages are much more relaxed with poetry being a good example of this. More importantly, natural languages, rather than computer languages, are used for the purpose of communication between humans, even when the information is conveyed between networked computers.

Overview

Table 2.2 overviews six layers of abstraction in Natural Language Processing, from most mechanical (parsing) to most abstract (pragmatic analysis). Parsing can be understood as preparing a document for analysis. This generally involves stripping out formating with regular expression matching in order to yield a sequence of characters. The next layer is lexical analysis, which focuses on individual words. Word stemming (Chapter 6) and speech recognition based on Hidden Markov Models (HMMs) falls in this category. Above that is syntactical analysis, where the structure of a sequence of words, generally a sen-

Pragmatic Analysis	**purposeful use of sentences in situations** • requires (world) knowledge that extends beyond the text • Cyc project at the University of Austin is an attempt to utilize world knowledge in NLP[Wik09]
Discourse Analysis	**interpret the structure and meaning of paragraphs** • requires resolution of anaphoric references & identification of discourse structure • requires discourse knowledge: how the meaning of a sentence is determined by the preceding one • requires knowledge of how a sentence functions in text
Semantic Analysis	**creating meaningful representation from linguistic inputs** • grammatically valid sentences can be meaningless: *"Colorless green ideas sleep furiously"*[Cho57] • lexical semantics (meaning of words) is key, and WordNet[Fel98] can be used for this
Syntactic Analysis	**analyzes a sentence to find its structure** • identifies how words in a sentence relate to each other • checks grammatically with contraints: word order, number and case agreement • requires syntactic knowledge and grammer rules • Part–of–speech tagging is an example
Lexical Analysis	**analysis of individual words** • requires morphological knowledge: structure and formation of words from basic units (morphemes) • rules for forming words from morphemes are language specific • speech recognition concentrates on this
Parsing	**preparing a text file for analysis** • removes document formatting to yield a sequence of characters for analysis • Apache Lucene has parsers for PDF, HTML, Microsoft Word, and OpenDocument documents[The09b]

Table 2.2: Levels of text processing, adapted from[ST08]

tence, is determined and interpreted. Part–of–speech tagging is an important component of this phase and its product is the association of a word category (noun, verb, adjective, etc.) with each individual word. Part–of–speech tagging is statistical in nature and is typically implemented with HMMs and Viterbi decoders. The logical meaning of a sentence is extracted during semantic analysis. A sentence, like Chomsky's *"Colorless green ideas sleep furiously"*, can be grammatically valid but contain no meaning. Above this is discourse analysis, which interprets the meaning and structure of paragraphs. It needs to determine the meaning of a sentence given the prior preceding sentences' meaning. Resolution of anaphoric references such as *it* or *they* is key. At the highest level of abstraction in Natural Language Processing is pragmatic analysis. Here, the entire document is analyzed in the context of external knowledge. This is analogous to interpreting findings presented in a conference paper with the aide of background knowledge and experiences.

Regular Expression Matching

Regular expression matching is a means of matching patterns, or even patterns of patterns. In text, this can be done on a bit or character level. Regular expression matching generally performs fairly shallow processing of text since it does not require a linguistic understanding of the underlying text. The context scope is focused on the (usually narrow) window of characters or bits in which the pattern is being matched.

One important application of a bit–level pattern recognition is data compression, in which a repeating long pattern can be swapped for a shorter code. In virus detection, bit patterns represent the signature of a particular type of attack. One of the most often cited pieces of software to used to identify the patterns is Snort[Sou09]. A complete listing of character–level regular expression syntax is listed in Table C.2. Regular expression matching is used extensively, both on a bit and character level, in the circuits outlined in Chapters 4, 5 and 6.

Part–of–Speech Tagging

Part–of–speech (POS) tagging is the process of associating words in a sentence with their grammatical categories, such as those outlined in the Penn Treebank POS Tag Table(C.1). For example, the sentence:

`The lead paint is unsafe.`

Can be tagged as:

`The`/Det `lead`/N `paint`/N `is`/V `unsafe`/Adj.

In which `/Det` denotes a determinant, `/N` a noun, `/V` a verb and `/Adj` an adjective. Relationships can be extracted with the aide of tags: the `lead paint` has the attribute of being `unsafe`. The same can be done with more complex sentences or sequences of sentences in order to build the rich relationships that humans derive when reading and interpreting text. These relationships can be stored in a relational or graphical database to enable searching and data mining. Most POS tagging is performed with the aide of HMMs.

2.3 Digital Logic

In the overwhelming majority of cases, IR and NLP algorithms are implemented in software, largely because open source software libraries exist in a number of programming languages, including Python[BLK09, BKL09, Liu09], Java[LLC09, The09d, Ai09, McC09, Gro09b, RG09, Liu09, Mor08, Gro09a, The09a, EAD09] and C++[LLC09, Ult09, Pet09, fLTA, The09a]. Lucene, a popular IR software library, is sponsored by the Apache Foundation and has a large community of developers and users[The09b]. Many NLP libraries are predominately maintained by university researchers and their use can significantly accelerate development, and the exploration of new ideas. Because of the longer development cycles and greater testing and implementation costs, there are no equivalent libraries for implementation in digital logic[1].

The closest published work to the topics explored in this thesis (UTF–8 transcoding, stop word filtering and stemming), is on accelerated and energy efficient document filtering [VAM09, FJ06, LE$^+$06]. The design proposed by Vanderbauwhede *et. al.* performs document filtering at 12.8 Gbps and matches documents against 20 topic profiles. Other publications have explored matching character patterns, either for the purpose of determining the language of the document's content[JG07], high–speed XML parsing[MLC08], packet inspection and routing [BSMV06, LPB06, NP08, JP09] or specifically virus and intrusion detection [HL09, DL06, SGBN06]. With respect to NLP in FPGAs, parsers for context–free grammars have been developed and published [Cir01, CML06, CML08].

Applications requiring a large number of parallel character string comparisons have made use of Bloom filters, which are space–efficient probabilistic data structures that test the membership of a set[Blo70]. These data structures are not exact: queries can yield false positives, but not false negatives. Extensions proposed by Song *et. al.* can facilitate exact matching and guarantee a boundary on worst–case lookup time[SDTL05]. By being space efficient, they can be placed in memory close to the indexing circuitry (typically on chip) or in high–bandwidth and low latency memory adjacent to the FPGA, minimizing memory access delay. Recently, Ho & Lemieux have shown that 80,282 character sequences containing a total of 8,224,848 characters, with an average length of 102 characters, can be fit within 4MB of memory and support virus detection at a throughput rate of approximately 1.6Gbps[HL09].

[1] However, OpenCores does maintain a large collection of ASIC & FPGA designs for other applications: http://opencores.org/projects

2.4 Character Encoding

There is a large number of character encoding standards though many of these have been, or are being, replaced by Unicode. This section presents three of the most common encodings standards: ASCII, Unicode and UTF–8.

ASCII

The American Standard Code for Information Interchange (ASCII) is based on the alphabetical ordering in the English language and was developed between 1960–1963. Its proliferation during the early history of modern computing systems led it to become the most widely used character encoding system in the world. Amongst all modern encoding, it is the simplest and often a starting point for improvements or regional variants. Extensions, such as ISO/IEC 8859, added support for other European languages. In the past decade, Unicode has overtaken ASCII as the preferred character encoding standard. Table A.1 shows all of the printable ASCII codes and their corresponding binary codes. Most of the 33 control characters in the ASCII table are now obsolete.

Unicode

Unicode is a standard for consistently encoding the vast majority of the world's writing systems. Included in the standard is data about how characters function[1]. The latest standard at the time of writing is 5.2, which encodes 90 scripts and a total of 107,361 characters[The09c]. By supporting such a large number of languages, it has led to internationalization and localization in computing. Unicode is logically divided into 17 *planes*, each of which consists of 65,536 code points (2^{16}). Plane 0 is referred to as the *Basic Multilingual Plane* (BMP). It contains complete character and symbol encodings for the modern writing systems in use by the vast majority of people in the world, and consequently, it is rare that characters from the remaining planes are ever used. Plane 1 contains mostly historical scripts. Plane 2 contains seldom used Unified Han Ideographs, which were included in the standard for completeness. Planes 3–13 are unused, 14 contains some non–graphical characters and 15–16 are for private use by proprietary systems. Encoding all 1,114,112 possibilities would require 21–bit characters. This is particularity space inefficient given that the majority of writing in electronic form is encoded in a small number of scripts whose characters are found at the beginning of the BMP.

UTF–8

UTF–8 is a variable–length encoding that reduces the average length of a character sequence encoded in Unicode. Encoding lengths can vary from one to four bytes. One byte encoding is effectively ASCII, which supports basic Latin characters. Two byte encoding adds complete support for Latin character extensions, Greek, Cyrillic, Armenian, Hebrew, Arabic and others. Three byte encoding covers the entire BMP, while four byte encoding covers all of Unicode. Officially endorsed by the Internet Mail Consortium, it is the most popular encoding format on the Internet.

UTF–8 is the most popular Unicode Transformation format. The others include UTF–1, UTF–7, UTF–16/UCS–2 and UTF–32/UCS–4. UCS–2 is a fixed 16–bit encoding with a one–to–one correspondence to Unicode's BMP. It is used as the internal character encoding standard in this project for two reasons: *(i)* the vast majority of digital text is encoded in languages whose characters are completely encoded by the BMP (optimum data width & flexibility) and *(ii)* a fixed character data width and propagation rate helps reduce data flow irregularities and design complexity. The implementation of a UTF–8 to UCS–2 transcoder is presented in Chapter 4.

Other Formats

Unicode was designed to overcome limitations in size and scope that affected other character encodings and as a result, various other encodings have become obsolete. This includes ISO/IEC 8859, which focused on European languages and whose first 256 characters have identical encoding in Unicode, Windows code pages (amongst which Windows-1252 is one of them), and EBCDIC used by IBM mainframes. GB18030 is closely related to Unicode. Since 2000, the government of the People's Republic of China has mandated that all software sold in its country support the encoding format and that CJK Unified Ideograph characters outside of the BMP must be supported.

Notes

[1] Taken directly from the Foreword to Unicode Version 5.2[The09c]: *The assignment of characters is only a small fraction of what the Unicode Standard and its associated specifications provide. They give programmers extensive descriptions and a vast amount of data about how characters function: how to form words and break lines; how to sort text in different languages; how to format numbers, dates, times, and other elements appropriate to different languages; how to display languages whose written form flows from right to left, such as Arabic and Hebrew, or whose written form splits, combines, and reorders, such as languages of South Asia; and how to deal with security concerns regarding the many "look–alike" characters from alphabets around the world. Without the properties, algorithms, and other specifications in the Unicode Standard and its associated specifications, interoperability between different implementations would be impossible.*

3

Project Overview

The focus of this thesis is on circuits that preprocess text in programmable logic. These circuits perform *(i)* character format transcoding, *(ii)* tokenization, *(iii)* stop–word filtering and *(iv)* stemming. As shown in Figure 3.1, the text could be streamed into this chain of modules from a computer network utilizing the TCP/IP protocol stack. An overview of the guiding principles and constraints that are common to the design of these components is presented in this chapter, while the following chapters focus on each component individually.

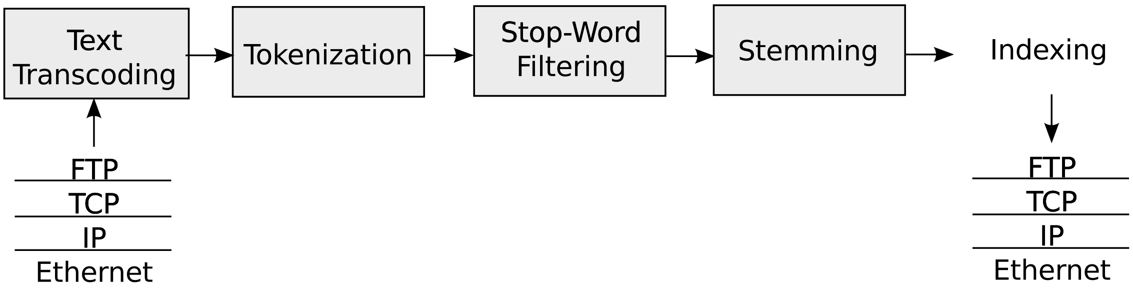

Figure 3.1: Project Overview

These four components can be used to detect and filter email spam[MIFR+06] or be integrated into the input stage of a larger document indexing system, where the hardware datapath could be interfaced with network–accessible storage (perhaps in the form of a Storage Area Network (SAN) or Network Attached Storage (NAS)). As observed by [VAM09], such a configuration could be significantly more power efficient than a software–based solution. However, the greatest performance and power advantages would be gained if the preprocessing circuit connected directly to the data bus of the disk array.

3.1 Network Attachment

With the proliferation of cloud computing, a network–centric computing model has begun to dominate large–scale data management and computing. In this model, information and processing services are provided for users on demand, and this places pressure on computing systems to support access by multiple users and necessarily over a network connection. Central to the cloud computation model is the notion of virtualization: providing data or a service through an interface while abstracting the underlying mechanics. The need for network access is recognized in the design of the datapath in the following two ways. First, the entire internal text processing architecture is designed according to a stream–processing model with the goal of operating at *line–rate*, that is, at the transfer rate supported by the network connection. Control signals are unidirectional (forward) in order to eliminate back-pressure on components located earlier on in the datapath (*congestion* from a network point–of–view). Furthermore, each component has a deterministic processing throughput guarantee, rather than a statistical one. Such a design approach could make the implementation of these circuits in computer networking equipment more attractive. Secondly, the circuit is designed to support UTF–8, the most common character encoding on the Internet (see Section 2.4 and Chapter 4).

Although the lack of a CPU can make it expensive to design and implement a document format decoder — a module that removes all formatting and file–format specific information from a document, yielding plain text — the decoding can be shifted to the client or an intermediary in order to overcome this disadvantage.

3.2 Modularity & Signaling

A key advantage of modularity is the ability to abstract interface and behavior from the underlying implementation. With a standard interface (Figure 3.2) and timing (Figure 3.3), design and testing can be simplified, while modules can be seamlessly swapped to quickly implement improved functionality or different algorithms.

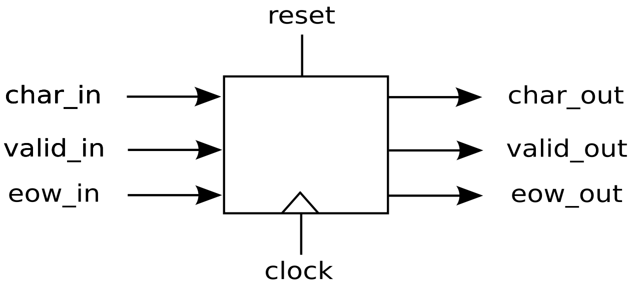

Figure 3.2: Standard module interfaces

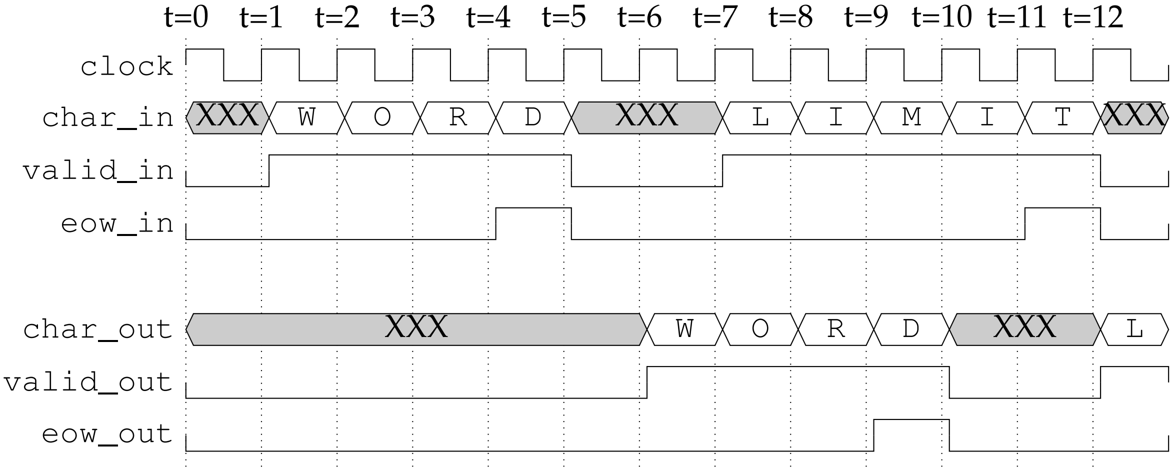

Figure 3.3: Standard module interface timing diagram (with delay)

Figure 3.3 details the signaling involved in transferring characters between modules. Signals with the suffixes `_in` denote those routed into a module, while `_out` suffixes denote signals routed out. char is a 16–bit bus that supports the encoding of a single character found in Unicode's Basic Multilingual Plane (BMP). This truncated and fixed–width 16–bit encoding is referred to as UCS–2 (see Section 2.4). There are two reasons behind this approach: *(i)* the vast majority of digital text is encoded in languages whose characters are completely encoded by the BMP (optimum data width & flexibility) and *(ii)* a fixed character data width and propagation rate helps reduce dataflow irregularities and design complexity. This could lead to more regular power consumption.

Token Boundaries

All characters are aligned to the positive edge of the master clock. Word boundaries are denoted by the edges of the valid signal, which is used by the module both to capture the incoming character and to *enable* the module's datapath. An implicit assumption affecting all modules is that when the valid flag is set, the character stream is sequential with no repetition, starting with a first character position in the source document. Since the valid flag functions as a clock enable for downstream modules, there is no strict requirement for the characters in a word to be adjacent to each other.

The eow (*end–of–word*) flag denotes that the entire word has just been transfered. It is particularly useful for stemming operations, which operate on suffixes. However, the eow flag has little meaning for non–segmented languages, such as Oriental languages, where words do not have explicit boundaries and a sequence of one–characters words can be joined to form multi–character words[Mik03]. For these languages it may be more advantages to associate a sequence number with each character and allow a tokenizer based on HMMs to determine how the characters should be segmented into words[ZLC+03]. Once segmented, the eow flag may become useful again.

Adhering to the standard interface described in Figure 3.3 on page 18, the valid & eow flags are appropriately set and synchronized with the output when they detect a word boundary. This *out–of–band* control signaling scheme leads to three clear benefits:

1. avoids modifying the original character stream, which allows for reverting changes in later stages,
2. natively handles tokens with white–space and punctuation and as a result,
3. does not require a mark–up language based on SGML or XML to maintain flexibility at the expense of transmission overhead and coding complexity[Mik03].

An additional benefit of the out–of–bound signaling is that it allows the tokenization circuitry to focus on one character at a time and identify token boundaries rather than token patterns. This facilitates the design of single–pass circuitry with at most *O(n)* complexity, a critical perquisite for supporting *real time* processing of a line–rate stream. This topic is explored further in Section 4.4.

Synchronous Signaling

The signaling described above does not support asynchronous data transmission for the following reasons: *(i)* all modules are designed to process one character per clock cycle, which leads to a uniform transmission rate and avoids the need for handshaking protocols and *(ii)* all of the modules outlined in this thesis are relatively small in terms of area footprint and when implemented in an ASIC, would not create clock skew large enough that clock domain boundaries would have to be accounted for.

Managing Data Flow Irregularities

The character stream that will arrive at the final stage in the module chain (indexing), shown in Figure 3.1, will have an irregular data flow rate. Characters will arrive at a rate of one per clock cycle, but the whitespace between words can be of arbitrary length. This is a consequence of the design principles used to support line–rate processing at a deterministic rate and with synchronous signaling between modules (see Section 3.1). Table 3.1 lists cases for each module in which whitespace would be injected into the character stream. The flow rate irregularities associated with the network interface have been included for completeness and it is feasible that they can be eliminated if sufficient memory is available for buffering. The lower effective data rate at the end of the processing chain relative to the input to the module chain could relax the indexing stage's throughput requirements.

Module	**Source of Data Flow Irregularity**
Network Interface	transmission errors, windowing, throttling, datagram unwrapping
UTF–8 Decoding	malformed sequences, variable encoding length
Tokenizaton	token scope (word, sentence, etc.)
Stop–Word Filtering	word filtering by whitespace substitution
Stemming	modifying word length

Table 3.1: Sources of Dataflow Irregularity

3.3 Regular Expression Matching with Fully–Decoded Delay Lines

The stop word filter and two stemmers decode the incoming character stream prior to performing regular expression matching. Moscola has shown that such an approach can achieve a throughput in excess of 10 Gbps, and that the performance can be linearly scaled with operating frequency and circuit size[MLC08]. This design technique, presented in Figure 3.4, is key to demonstrating that text processing can be performed faster in hardware than in software, with efficient area and resource utilization, and with significantly less power consumption.

With this approach, the incoming character stream is *one-hot* encoded; the character is decoded such that only the single wire on the output bus that corresponding to the character will have the logical value `1` at any given time. Registers in the bus allow multiple character positions to be compared. In this particular example, the suffix *ily* is being matched and a minimum 2–letter stem length rule is being applied simultaneously (the asterisk denotes *'any letter'*). With character decoding being performed only once, on

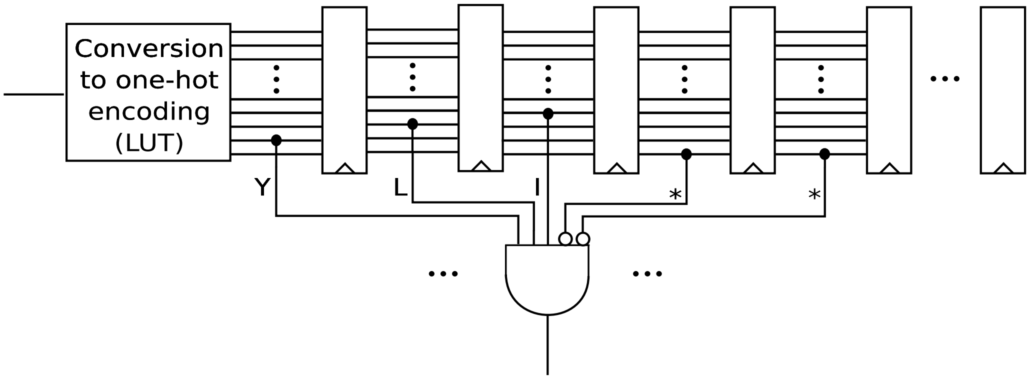

Figure 3.4: Regular Expression Matching with fully–decoded delay lines

input, complex logic expressions can be build with simple logic expressions for which a single wire is sufficient to represent a character and its position. With the latest generation FPGAs having 6–input Look Up Tables[1], the throughput is often only limited by physical device constraints on maximum operating frequency, such as wire delay and routing efficiency within the FPGA. Modern synthesis tools will optimize the register count of each delay line to only what is required by the logic expressions.

3.4 Supporting Multiple Languages

Multilingual support would require an additional language detection stage, and it would need to buffer a sample of the incoming character stream in order to determine the language. Following this, the character stream would be routed to the language–specific chain of text processing modules. The language detection stage could be implemented by searching for n–character sequences that are characteristic to that language[CT94]. An implementation of such an approach has achieved a throughput of 11.2 Gbps and an average accuracy of 99.45% across several languages[JG07]. The fixed–width UCS–2 is capable of supporting all common human languages in use.

[1] At the time of writing (May 2010), the newest FPGA architectures available on the market were Altera Stratix V and Xilinx Virtex–6.

3.5 Technology used for Benchmarking

All circuits detailed in thesis were simulated and implemented on an Altera Stratix IV EP4SGX230KF40C2 FPGA, a device that is part of the manufacturer's most advanced product line in production at the time of writing. It was programmed and tested with the manufacturer's *Stratix IV Development Kit*. This 40 nm device was released in the fourth quarter of 2008 and contains 182,400 registers, 228,000 programmable Adaptive Look Up Tables (ALUTs) and 888 pins for general interfacing, some of which connect to 3.1875 Gbps transceivers. The unit price is significant: $9,020.01. For cost sensitive applications, a lower–end product such as the 60 nm Altera Cyclone IV GX EP4CGX150DF31C7 ($537.46 unit price) should be considered though it will not be capable of achieving as high of a clock rate[2].

FPGA design tools incorporate statistical optimization techniques, such as simulated annealing and Tabu search. Random algorithm seeding can yield slightly different placement and timing results. Similarly, the performance of the underlying silicon fabric varies between devices, leading to different *speed grades*. Thus, designs always need to allow for a performance margin in order to guarantee performance across a large range of device speeds, operating temperature ranges and noise interference levels, and such margins can be determined through Monte Carlo simulation. All power figures were measured on the development board via on–board current monitors.

[2] Prices sourced from *Altera Buy Online* on May 28, 2010 `http://www.altera.com/buy/buy-index.html`

4

Text Encoding

UTF–8 is the most popular text encoding format for communication via the Internet. Its encoding length varies from 1–4 bytes and the module described in this chapter decodes it to UCS–2, the project's internal 16–bit character encoding. The conversion to a fixed character data width reduces dataflow irregularities and design complexity in subsequent modules. Table 4.1 outlines how various UTF–8 sequence lengths are translated. Table A.1 displays all of the printable ASCII characters, which are encoded by a single UTF–8 byte.

	UTF–8					
Unicode	Byte 1	Byte 2	Byte 3	Byte 4	**Internal 16–bit**	
`U+0000–007F`	`0xxxxxxx`				`00000000`	`0xxxxxxx`
`U+0080–07FF`	`110yyyxx`	`10xxxxxx`			`00000yyy`	`xxxxxxxx`
`U+0800–FFFF`	`1110yyyy`	`10yyyyxx`	`10xxxxxx`		`yyyyyyyy`	`xxxxxxxx`
`U+10000–10FFFF`	`11110zzz`	`10zzyyyy`	`10yyyyxx`	`10xxxxxx`	`11111111`	`11111101`

Table 4.1: UTF–8 Decoding

4.1 UTF–8 Decoding Errors

The UTF–8 decoder handles the following three types of errors, which it treats as out–of–bound encodings, and replaces each erroneous sequence with the Unicode BMP replacement character (`11111111 11111101`):

1. incorrect sequence start code
2. incorrect continuation byte encoding
3. invalid number of continuation bytes

The Unicode replacement character allows for better error handling by subsequent modules since it does not ambiguate whitespace and preserves the original word length. The decoder does not handle *overlong* sequences, which are those encoding that can be represented by a shorter sequence:

```
1100000x  (10xxxxxx)
11100000   100xxxxx   (10xxxxxx)
11110000   1000xxxx    (10xxxxxx   10xxxxxx)
```

Although such encoding are known to have been used to exploit string operators in web servers (as with the *Code Red Worm*), such a vulnerability does not exist for this project.

4.2 Design and Implementation

The decoder handles one UTF–8 byte every clock cycle and is controlled by a 13–state FSM, which is binary encoded and shown in Figures 4.1 and 4.2. Because of the small number of FSM states and the FPGA's 6–input ALUTs, the maximum clock rate is not sensitive to the complexity of the FSM's binary decoding nor state transition logic. As a result, utilizing one–hot encoding, a technique applied in high performance ASIC design, does not improve maximum clock frequency, which at 704 MHz is already at the physical limit for this device. Instead, one–hot encoding would increase register usage. Resource utilization and timing results are shown in Table 4.2.

The circuit's maximum data throughput rate is 5.632 Gbps. The incoming network data rate that it can effectively support will be greater in proportion to the number of packet transmission errors and amount of datagram overhead in the network interface. Assuming error–free transmission over Ethernet (1500 btye MTU) with IPv6 (320 bit header), TCP (160+ bit header) and no application–layer overhead, the data rate entering the network interface could be at least 4.17% larger (5.867 Gbps). However, a more meaningful metric would be *character throughput rate*, the average number of characters processed per second, which depends on the average UTF–8 character encoding length for a representative piece of text. Consequently, English text will likely lead to a character transfer rate just under twice that of Cyrillic text.

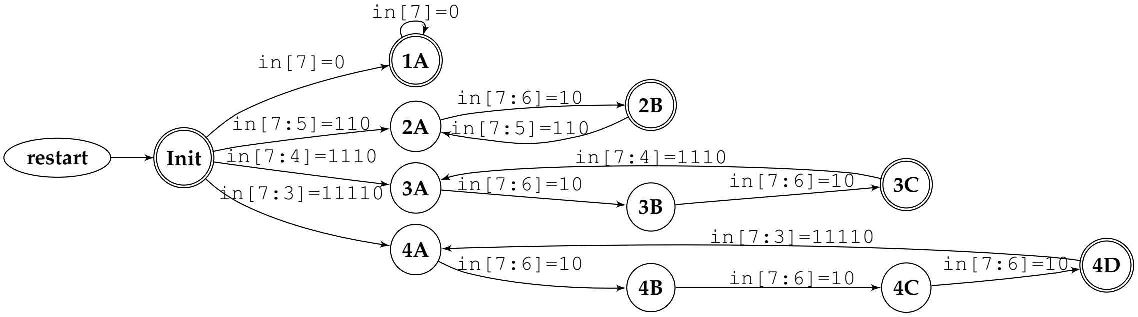

Figure 4.1: UTF–8 Decoder FSM

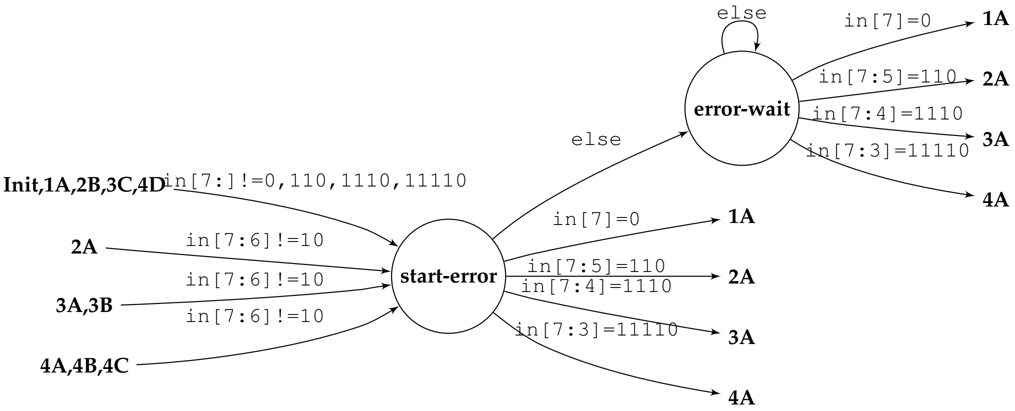

Figure 4.2: UTF–8 Decoder FSM Error States

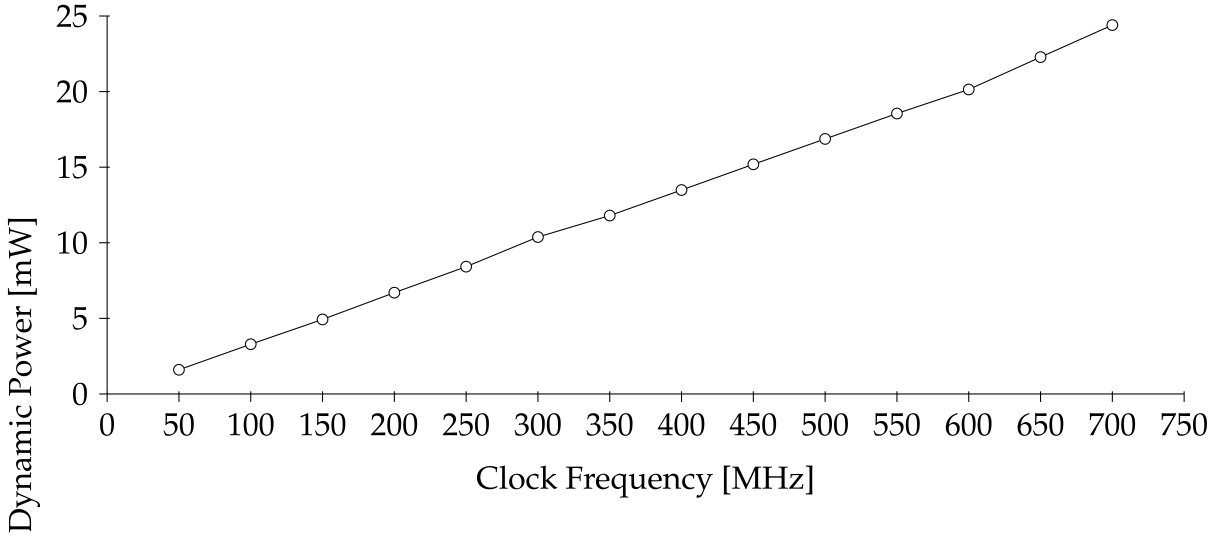

Figure 4.3: UTF–8 Decoder Dynamic Power Consumption

	Area		Speed	
Device	LUTs	Registers	Fmax (MHz)	Critical Path
Stratix IV GX EP4SGX230KF40C2	37	46	704	Master clock

Table 4.2: Resourse usage and performance of a UTF–8 decoder

Figure 4.3 shows the circuit's dynamic power consumption at different clock frequencies, which scales linearly with circuit toggle rate and clock frequency. At 704 MHz, the maximum operating frequency, the circuit is consuming 24.4 mW while processing a UTF–8 data stream at a rate of 5.632 Gbps. This translates to a power efficiency of 4.332 mW/Gbps or 34.66 μW/MHz. However, at this operating frequency, the FPGA's total power consumption is 967 mW. 127.74 mW is consumed by the design's 30 input & output wires, which are routed to the FPGA's input & output pins that drive relatively large load and parasitic capacitances. This figure would be zero in a complete design where the input is chained to a network interface and the output to the next stage (tokenizer). The remaining 814.77 mW is dissipated by leakage currents in the inactive portions of the FPGA.

The input data stream to the UTF–8 decoder was James Joyce's *Ulysses*, which was obtained from Project Gutenberg: http://www.gutenberg.org The power consumption of the circuit with text in 9 other languages was also measured. While the activity factors of the three most significant bits in the UTF–8 encoded data stream showed significant variability, the difference in circuit power consumption was negligible.

4.3 Avoiding Whitespace Gaps in Tokens

The module's input and output bandwidths are not equal. The input is 8 bits wide and can have variable byte length, while the output is fixed at 16 bits. In order for the output to be read out at one character per cycle with no whitespace inserted in between characters in a word, buffering by means of a circular buffer can be implemented. Figure 4.4 overviews such a design. Whenever an incoming character is fully decoded, it is written to the next available position in the circular buffer, and this address is incremented in the *write counter*. A whitespace character is taken to signal the end of a word or token, at which point an internal `eow` flag is generated and the current value of the *write counter* is passed to the *read counter*. The `eow` flag value for each character is stored a 17^{th} bit position associated with each decoded character, revealing the output stage from implementing duplicate end–of–word detection logic.

On the output stage, an `eow` value of `1` triggers the *read counter* to increment a memory address pointer from the last read position (end of previous word/token), to the position passed in (end of the latest word/token), and wait there for the next token to be fully buffered. After the last character is read from the circular buffer, whitespace is injected into the output stream until the next word/token has been completely buffered and is ready to be read back.

The output is read faster than the input is written in proportion to the average number of byte sequences required to encode the input. Thus, the buffer's memory requirements are bounded. The actual size of the buffer needs to be at least as large as the character length of the longest token. The circular buffer also needs to be implemented in dual ported memory in order to permit the writing of decoded input characters while the output stage is reading. Furthermore, the *read counter*'s terminal address needs to allow for updating while the counter is incrementing in order to allow a short token to be added to the read queue while the output stage is reading a long token. Based on an initial analysis of full length books in nine segmented languages available in the Gutenberg Project, the need for a reasonable margin, and the base–2 regularity of memory size, a circular buffer with 32 positions was found to be sufficient.

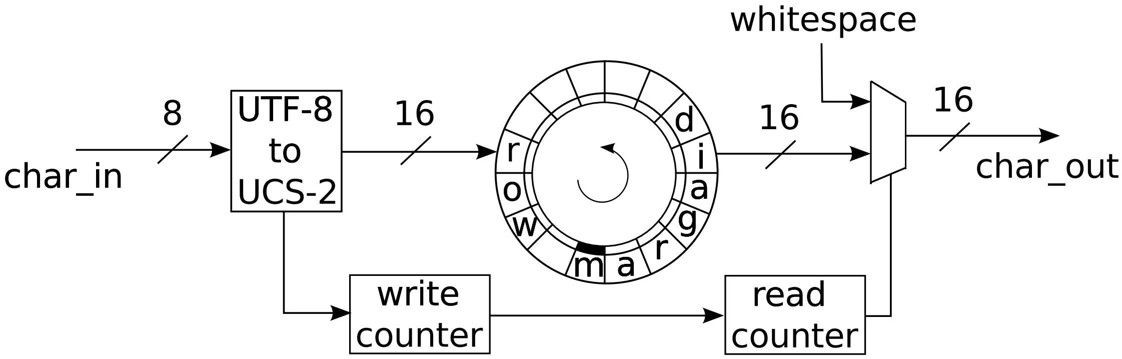

Figure 4.4: UTF–8 Decoder with Dual Ported Memory. The state of the `eow` flag associated with each character is shown in the box beside each character, with a filled box denoting a value of `1`.

4.4 Tokenization

Tokenization is the process of segmenting a character stream into discrete linguistic units (tokens) corresponding to some linguistic abstraction, and is an important preprocessing stage for Natural Language Processing applications such as part–of–speech tagging (see Table C.1). Examples of tokens include single words, such as *computer*, hyphenated words such as *low–budget*, or a sequence of words such as the named entity *United States of America*. More broadly, tokens can also be paragraphs, sentences, syllables, or phonemes. Tokenization is generally considered to be relatively straight forward for *segmented languages*, which use Latin-, Cyrillic- or Greek-based characters and in which explicit separators such as blank spaces and punctuation strongly predict segmentation boundaries. However, ambiguous punctuation, hyphenation, clitics, apostrophes and language–specific rules make an accurate tokenizer more difficult to design[Mik03].

Non–segmented languages, such as Oriental languages, increase design complexity since tokens do not have explicit boundaries and a sequence of one–characters words can be joined to form multi–character words[Mik03]. Tokenization for these languages can be achieved with hidden Markov models[ZLC+03], n–gram methods or other statistical techniques. Higher–level text segmentation would involve segmenting noun and verb groups, splitting sentences into clauses, and so on. As with other text processing modules, *precision*, *recall*, *accuracy* and *error rate* are important performance measures (see Table 2.1).

In the previous section, the transition from a printable character to whitespace signaled a token boundary and triggered the update of the read pointer's terminal address. This is effectively whitespace tokenization and is the simplest and least memory intensive tokenization strategy. The specific whitespace characters in these scheme included `space`, `tab`, `new line` and `form feed`.

A tokenizer's required precision is often dictated by the needs of the target application. However, the design should always aim for the highest precision since tokenization errors propagate to later linguistic processing stages. A more complete tokenizer is included in the Natural Language Toolkit[BLK09]. It handles abbreviations, words with optional internal hyphens, currency, percentages and treats ellipses and punctuation as separate tokens[BKL09]. Table 4.3 contains a complete listing of the regular expressions used by this tokenizer.

Neither the whitespace tokenizer nor the NTLK tokenizer handles hyphenation. *End–of–line hyphens* are used for text justification and are inserted during typesetting. It is

Regular Expression	Comment
([A-Z]\.)	abbreviations, e.g. U.S.A.
\w+(-\w+)*	words with optional internal hyphens
\$?\d+(\.\d+)?	currency and percentages, e.g. $12.40, 82%
\.\.\.	ellipsis
[][.,;"'?():-_`]	punctuations are separate tokens

Table 4.3: Regular expressions used by the default NLTK tokenizer, reproduced from[BKL09]

assumed that they are resolved by a previous stage that handles document format stripping. *True hyphens* are left in place and the decision of removing the hyphenation – and how – is left to the word indexing stage. True hyphenations can be grouped into two general categories: *lexical hyphens* (co–, pre–, multi–, etc.) and *semantically determined hyphenation* (case–based, three–to–five–year, etc.)[Mik03]. Email addresses, URLs, dates, citations, numbering schemes, etc. are not considered in this project, since incorporating capabilities to recognize them would lead towards designing a preprocessor for information extraction[Moe06].

5

Stop–Word Filtering

The largest text corpus for the modern English language is the Oxford English Corpus, which contains over 2 billion words. 80% of the source material is written in British and US English and all has been created no earlier than the year 2000. 30.5% is sourced from news and weblogs; material that provides an accurate snapshot of the most common words in daily use. Table B.1 lists the 100 most common word in the corpus, which account for about 50% of all word occurrences in the corpus. Of these, *the* occurs almost 100 million times (approx. 5%), while the ten most common words together occur 25% of the time. A vocabulary of 7000 words is sufficient to cover 90% of all words in use, while an additional 43,000 words are required to cover the next 5%. The distribution has an extremely long tail of rare words; a vocabulary in excess of 1 million words is needed to cover 99% of all words in use. Table B.2 lists the 25 most common nouns, verbs and adjectives. Other contemporary corpora include the Corpus of Contemporary American English (over 400 million words, [Dav09]) and the British National Corpus (over 100 million words, [Bri09]).

Such an uneven distribution implies that *(i)* natural language text has high dimensionality: of the hundreds of thousands of words in an language, only a small percentage is used in a typical document, and that *(ii)* the most salient words in a given document are those which fall between these two extremes [ST08]. The words which occur most frequently across all documents in a given collection, and thus add no differentiating, unique meaning to the individual document, can be filtered out by means of a stop–word list.

The design of a stop–word list leads to trade–off: recall versus precision. The inversely proportional relationship between the two is shown in Figure 5.1. High precision filtering will miss many useful documents while at the other extreme, a search with high recall will return most useful documents but not be effective at filtering documents with low relevance. The optimal stop word list is dependent on the application. In this chapter, Porter's stop word list is used in order to determine how the number of characters and words affects circuit performance. The complete word listing can be found in Table B.3.

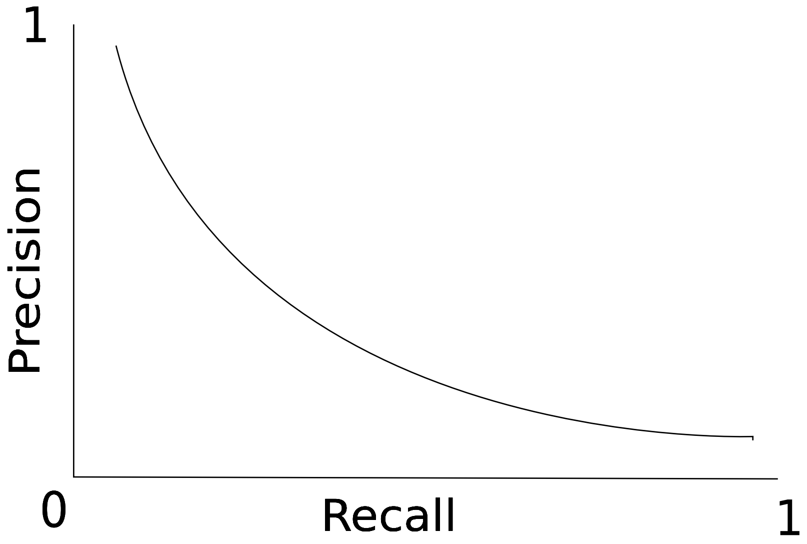

Figure 5.1: Tradeoff between recall and precision

Design & Implementation

This circuit implements regular expression matching with the technique presented in Section 3.3, and the conceptual schematic is shown in Figure 5.2. A token is *filtered* out from the character stream by swapping its characters with whitespace characters. Within the matching stage, the stop words are sorted by character length and the match with the longest length will determine how many characters will be swapped. `eow_in` denotes the end of the token. It allows the correct character length to be determined and *enables* the signals sent to the multiplexers. The token matching circuitry has a propagation delay that spans several cycles, and the incoming character stream is synchronized to the output of the token matching circuit by a multi–cycle delay line (`n` registers in series, as shown on the input of Figure 5.2).

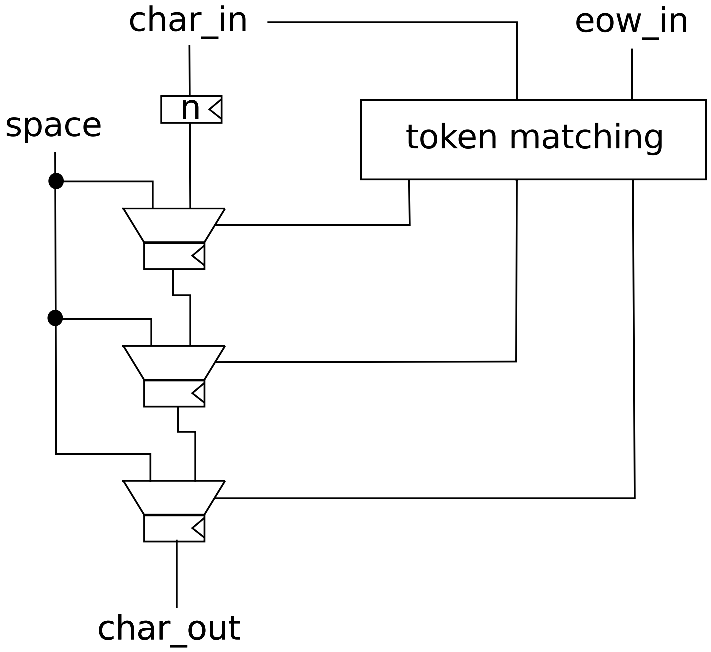

Figure 5.2: A stop word filter

Figure 5.3 shows the ALUT and Register requirement for different cumulative stop word character counts. The ALUT requirement grows approximately linearly since ALUTs are used to match words and determine character length precedence. The register count on the other hand does not exhibit linear growth all the way through. At low cumulative character counts, a disproportionately large number of registers is required to create the delay line and multiplexer infrastructure. As more words are encoded, these resources are reused, slowing growth. The steep tail in the 700–800 character range is the effect of a few 7–9 letter words being added and little register reuse amongst the 7^{th}, 8^{th} and 9^{th} characters.

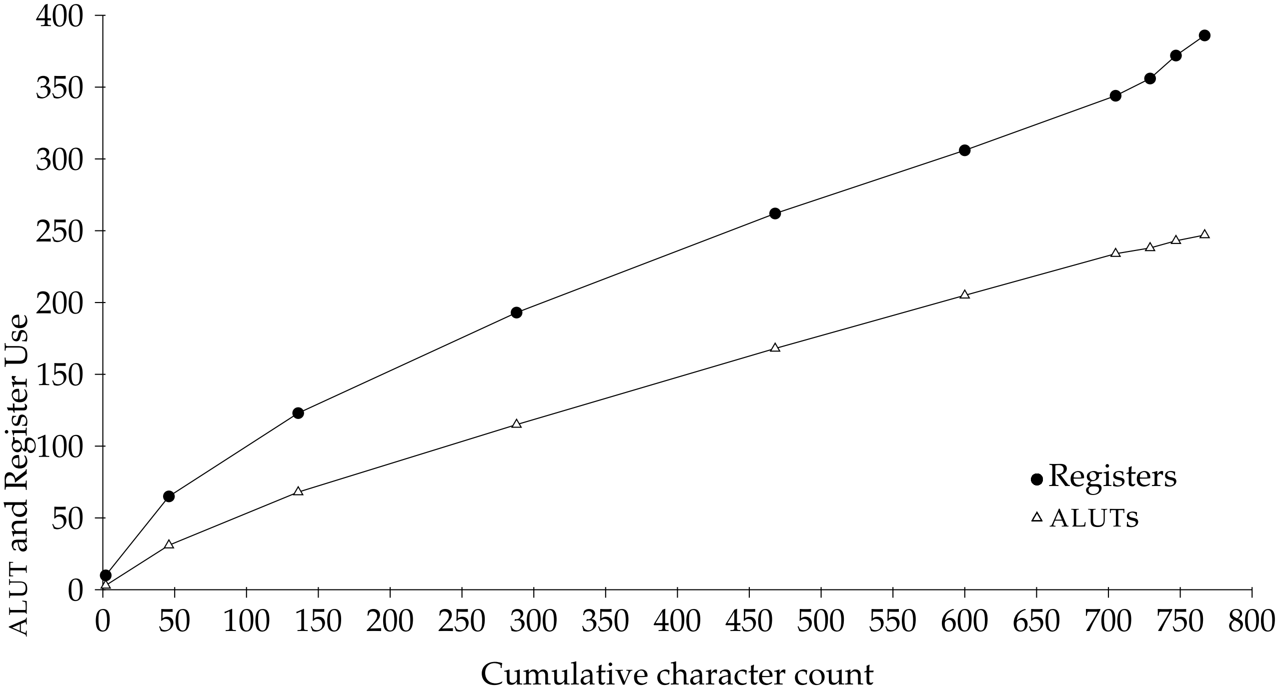

Figure 5.3: Resource usage as a function of vocabulary size

Figure 5.4 shows the circuit's dynamic power consumption at different clock frequencies, which scales linearly with circuit toggle rate (an effect of the clock frequency). At 704 MHz, the maximum operating frequency, the circuit is consuming 45.36 mW while processing a UCS–2 data stream at a rate of 11.264 Gbps. This translates to a power efficiency of 4.05 mW/Gbps or 64.8 μW/MHz. However, at this operating frequency, the FPGA's total power consumption is 986.64 mW, with 125.14 mW begin consumed by the design's input & output pins and the remaining 816.14 mW dissipated by the remainder of the FPGA fabric, which is largely inactive.

	Area		Speed	
Device	ALUTs	Registers	Fmax	Critical Path
Stratix IV GX EP4SGX230KF40C2	314	664	704	Master Clock

Table 5.1: Resourse usage and performance of a stop–word filter

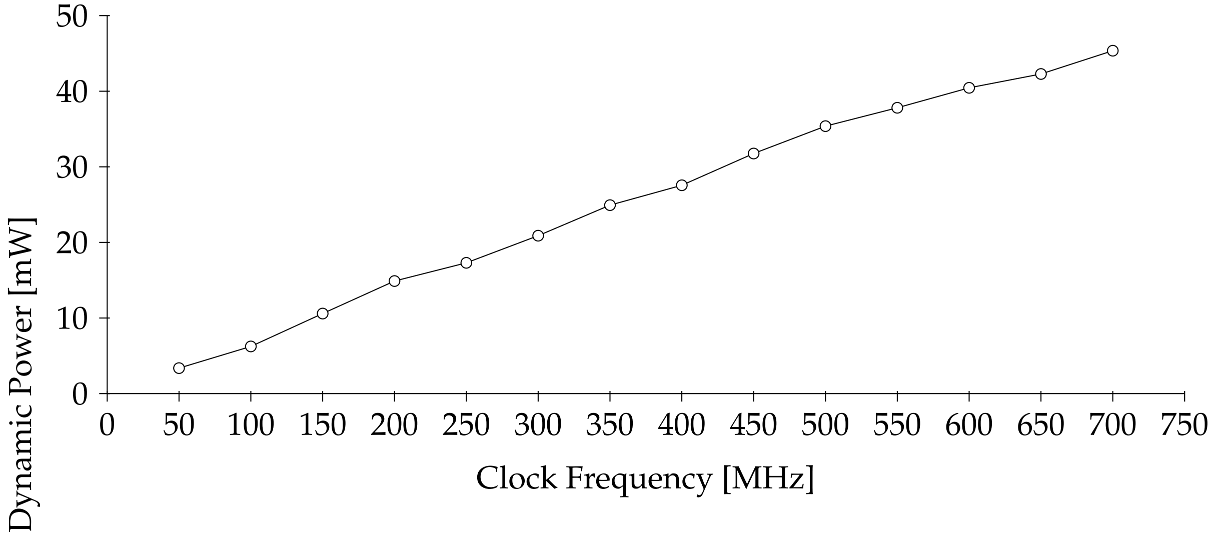

Figure 5.4: Stop–Word Filter Dynamic Power Consumption

6

Stemming

Stemming reduces a word to its root form, disregarding morphological information such as pluralization, gender conjugation, and so on. For example, *fishing*, *fished* and *fishes* can be reduced to the stem *fish*. Stemming is useful for compressing a word index and increasing the recall of related documents. For this reason, the root form does need to have valid spelling. It is the mapping between words that is important. Web search engines, such as Google's, also implement stemming during their query expansion phase.

Broadly speaking, there are two ways of performing stemming: either solely with a set of regular expression rules (lexicon–free), or with the help of a lexicon. There are also several metrics that gauge the *strength* of a given stemming algorithm[FF03]. These include:

- mean number of words per conflation class
- index compression factor
- number of words and stems that differ
- mean number of characters removed in forming stems
- median and mean modified Hamming between words and their stems

This chapters focuses on the implementation of the Lovins and Porter stemmers in programmable logic, providing comparative metrics to gauge the design complexity, performance and resource utilization. Both are lexicon–free stemmers. Of these, Porter's stemmer has been recognized to consistently yield good performance in larger IR and NLP applications. However, the choice of design and stemming strength is dependent on the application.

Stemmers which were not considered in this chapter include designs by Paice & Husk[Pai90], Dawson[Daw74] and Krovetz[Kro93]. The design by Paice and Husk applies suffix removal rules iteratively. Without a deterministic guarantee on processing time, it is not suitable for real time applications. Dawson's stemmer is an extension of Lovin's design which implements approximately four times as many suffix matchings, and a more reliable partial matching procedure to correct spelling. Krovetz's stemmer is an accurate but relatively weak stemmer which in practical applications requiring more compressed indexes needs to be complemented with a second stemmer. This stemmer also requires a dictionary (memory).

6.1 Lovins

This stemmer was published by Julie Beth Lovins in 1968[Lov68] and was the first of its kind. The single pass algorithm leads itself well to an implementation in digital logic. It can be divided into two stages. In the first stage (*"Trimming"* in Figure 6.1), 294 suffixes are compared in order to find the longest match. These endings are 1 to 11 characters long and are associated with a condition code, which specifies how much of the stem can be removed and how the trimmed word should be transformed in the second stage. There is a total of 29 conditions, many of which require that a particular letter or pair of letters be identified in the stem. The second stage (*"Transforming"* in Figure 6.1) corrects spelling by applying one of 35 rules, each of which typically involves swapping letters. Table D.1 lists the frequency of Lovins stem lengths, Table D.2 contains a complete listing of the 294 suffixes, and Table D.3 lists the context–sensitive rules and the frequency of their occurrence.

Hardware Design

These two stages can be viewed as string splicing and concatenation. Their schematic is shown in Figure 6.2. The input, a stream of characters entering the circuit at a rate of one character per clock cycle, is shown on the lefthand side. The two stages perform their processing in parallel with the input flow.

In order to achieve a high throughput and operational clock rate, all 294 suffixes are matched in parallel and with the smallest area (ALUT & register) utilization possible since increasing area can negatively influence processing speed. Moscola's technique of fully decoding the incoming character stream into a pipelined one–hot encoded bus, where the width of the bus corresponds to the size of the alphabet and the pipeline stage to the character position[MCL07], is applied in order to achieve high area density and high throughput.

As shown in Figure 6.1, suffix identification simplifies to tapping appropriate *letter lines* in the bus. The character length of the ending is directly proportional to the number of logic gate inputs. The gate output is a logical `1` if the ending exists. Rules, such as a minimal stem length, can also be implemented with complex logic gates, and merged with the suffix identification function. While 26 letter lines are sufficient for the English alphabet, an additional one is used to denote non–letters such as whitespace and punctuation. If this line is equal to zero, then a valid letter resides in the associate character position, allowing the design to avoid counters when evaluating length-based stem rules. An additional apostrophe character line is also instantiated. The `eow` flag acts as an enable that validates stem identification output.

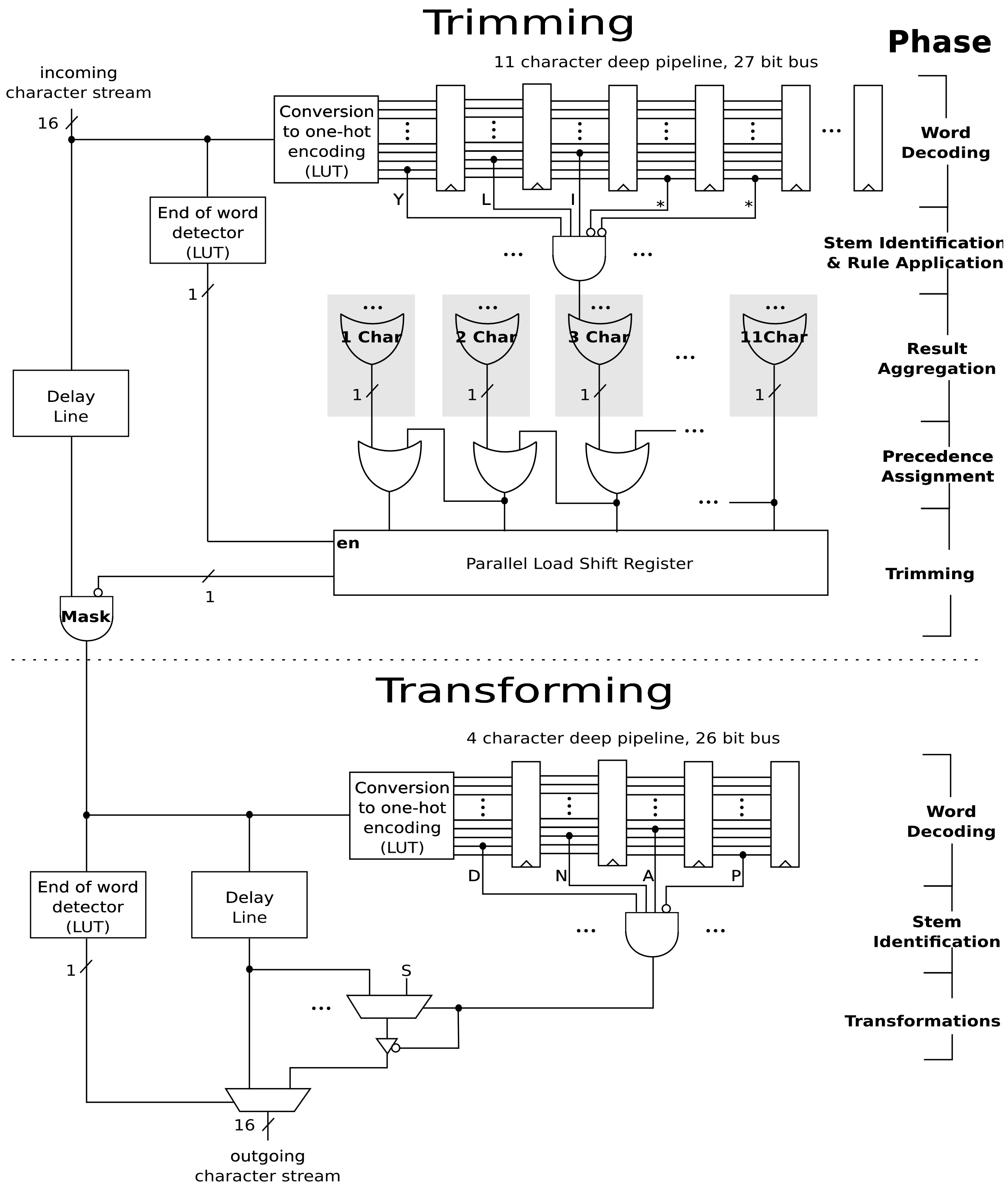

Figure 6.1: A design of the Lovins Stemmer in programmable logic

Input	Output
1XXXXXXXXXX	11111111111
01XXXXXXXXX	01111111111
001XXXXXXXX	00111111111
0001XXXXXXX	00011111111
00001XXXXXX	00001111111
000001XXXXX	00000111111
0000001XXXX	00000011111
00000001XXX	00000001111
000000001XX	00000000111
0000000001X	00000000011
00000000001	00000000001
00000000000	00000000000

Table 6.1: Precedence assigner truth table

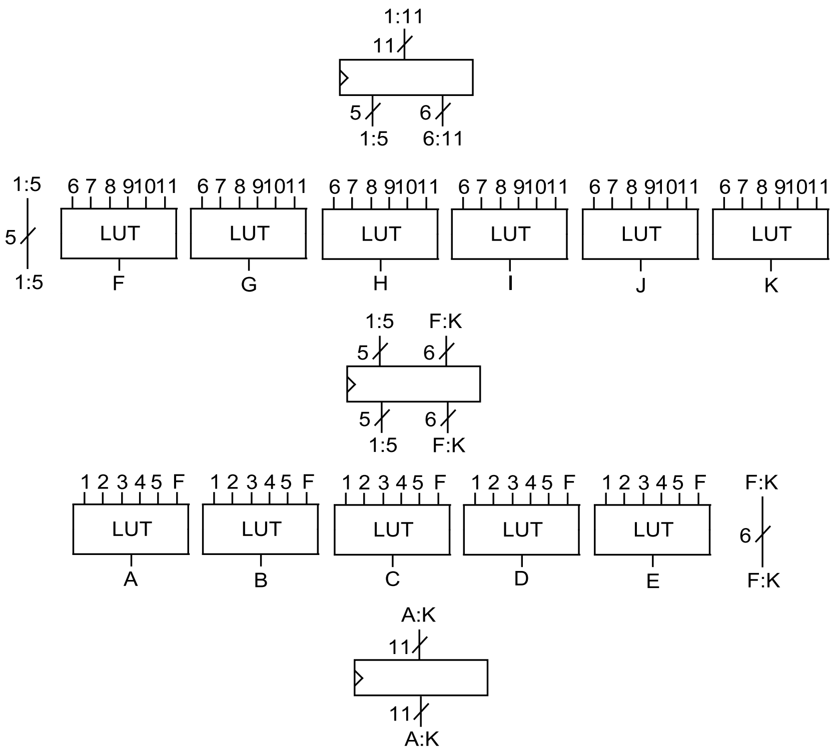

Figure 6.2: Mapping the precedence assigner into lookup tables to overcome an input-wide critical path

The 294 suffix identification circuits are grouped by suffix length. Within each group, the outputs of the stem matching circuits are aggregated with a tree of OR gates in order to determine if any suffix of a given length had been identified. There is overlap in suffix spelling, which could lead to suffixes of several different lengths being matched. To identify the longest, the OR–ed suffix length bits are fed into a *precedence assigner* (Table 6.1 and Figure 6.2). The precedence assigner has 11 inputs – one for each OR tree output – and ordered from longest suffix character length (at the MSB) to the lowest. Match precedence is given to the longest stem match and as such, the circuit identifies the logical 1 closest to the MSB and masks all bits between it and the LSB, inclusive, with 1s. The function's truthtable is shown in Table 6.1. By mapping the precedence assigner into ALUTS, which are effectively small programmable memories, a critical path spanning from the MSB to the LSB is avoided, allowing for a higher operational clock frequency. The precedence assigner's output is passed into a parallel load shift register, which is synchronized to a delayed copy of the original word. The shift register feeds a series of multiplexers that overwrite the ending with whitespace (space characters), effectively trimming it.

The second stage is a simplified implementation of the first. The character stream is again decoded in order to efficiently identify the suffixes needing spelling correction. The spelling is corrections are made by swapping appropriate letters (bit patterns) into the correct positions in the incoming character stream by means of multiplexers. Of the 35 possible corrections, at most one is identified and made, simplifying arbitration to the output bus to a tristate buffer.

This design has two important drawbacks. The first is that the fanout of *letter lines* in the decoded character bus is *(i)* irregular, since it depends on the suffix character distribution and *(ii)* could be quite large for frequently occurring letters such as the vowel *e*. The second problem is that the associated suffix identification and rule evaluating logic is fixed. A programmable, memory–based implementation could allow for rule updates and support for multiple languages. These two points may be important considerations

	Area		Speed	
Device	ALUTs	Registers	Fmax	Critcal path
Stratix IV GX EP4SGX230KF40C2	605	1040	704	Master Clock

Table 6.2: Lovins Stemmer resource utilization and speed

for an ASIC implementation, though they have not been found to be performance drawbacks for the target FPGA.

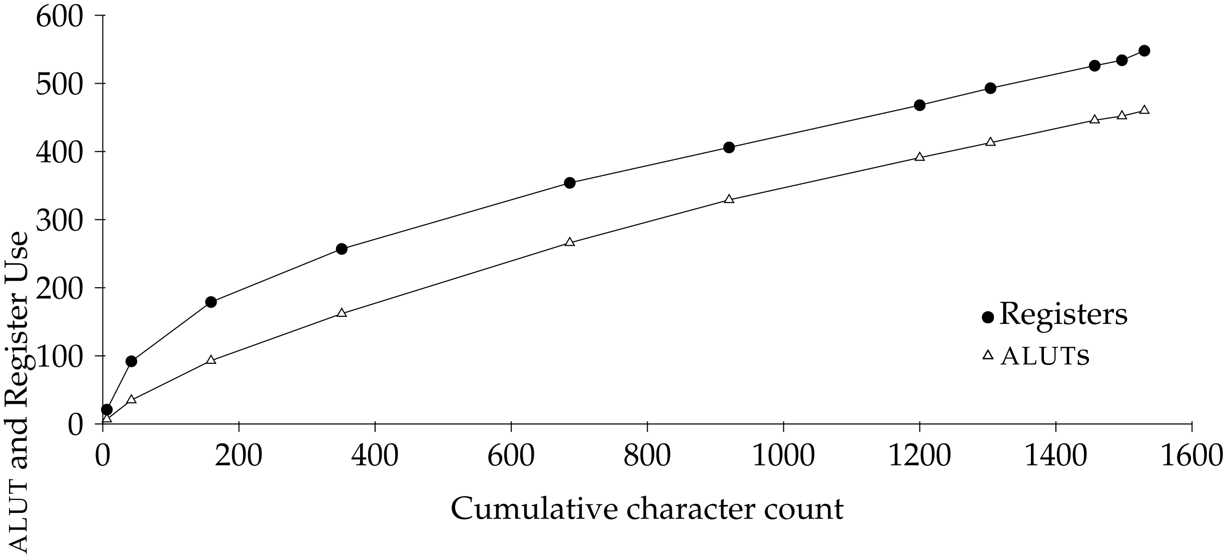

Figure 6.3: Resource usage as a function of vocabulary size

6.2 Porter

Another popular stemmer is the one developed by Martin Porter[Por80]. It is a five step, linear algorithm based on the idea that suffixes in the English languages are generally combinations of smaller and simpler suffixes. Similarly to the Lovins stemmer, each step matches suffixes and evaluates the suffix removal with a condition, such as the minimum number of vowels required to remain in the resulting stem. This stemmer has been implemented in a number of document and Natural Language Processing applications, including NLTK, Drupal and Lucene[The09b].

Algorithm Overview

Porter[Por80] conjectures that any word is a composition of consonants and vowels, which are defined as:

vowel	a, e, i, o and y when preceded by a consonant
consonant	all letters not vowels and y when preceded by a vowel

Using the following nomenclature:

v	a vowel	V	one or more consecutive vowels
c	a consonant	C	one or more consecutive consonants

Any word can be denoted in the following form:

$$[C](VC)^m[V]$$

Where m denotes the *measure* of a word. The measure can be roughly interpreted as the number of syllables in the word. The square brackets, [], denote that the preceding consonants or trailing vowels are optional. The following table shows how a word can be decomposed to adhere to this notation.

m	form	examples
0	$[C][V]$	tree → $[\text{TR}][\text{EE}]$
1	$[C](VC)[V]$	trees → $[\text{TR}](\text{ees})$
2	$[C](VC)(VC)[V]$	treaty → $[\text{TR}](\text{eat})(\text{ty})$

In addition to checking the word measure, the condition may also be a compound form of one or more of the following rules (taken from the original paper[Por80]):

rule	explanation
$*S$	stem ends with S (and similarly for the other letters)
$*v*$	the stem contains a vowel.
$*d$	the stem ends with a double consonant
$*o$	the stem ends cvc, where the second c is not W, X or Y

The algorithm proceeds by first removing pluralizations, then simplifying stems, and finally fixing spelling. The transformations and rules associated with each of the five steps are detailed in Appendix D.2.

Hardware Design

In hardware, the Porter stemmer is segmented into the five steps detailed in Appendix D.2. As in the original algorithm proposed by Martin, the first step is further partitioned into three steps. The flow between these steps is shown in Figure 6.4. In this design, m is used extensively in suffix transformation conditions. It is computed in the `m counter` module for each character of the incoming token. This means that for any given character in a token, the associated value of m reflects the running total as if that character has the last in the token. Within the Porter stemmer, the standard signaling of Figure 3.3 is appended with `m_in` and `m_out`. Since the conditions against which m is tested are > 0, $= 1$, > 1, m is implemented as a two bit signal and generated by a saturating two bit up counter. This reduces comparison complexity and consequently, overall circuit size.

The shaded stages (2, 3 & 4) denote those implementing fully decoded delay lines (Section 3.3). Those which are not shaded compare 16–bit characters directly since the number of characters that needs to be matched is small and the net ALUT and register use is smaller than with the character decoding circuitry. As with stop word filtering and the Lovins stemmer design, multiplexers are used to substitute characters or whitespace directly into the outgoing character stream (see Figure 5.2). The multiplexers are controlled by suffix matching logic and enabled with the `eow` flag.

6.3 Implementation & Comparison

Figure 6.5 shows the power consumption of the two designs. While the Porter stemmer utilizes fewer ALUTs (27.1%) and fewer registers (13.5%) than the design for the Lovins Stemmer, the Porter stemmer engages more of its resources into the 16–bit text data flow.

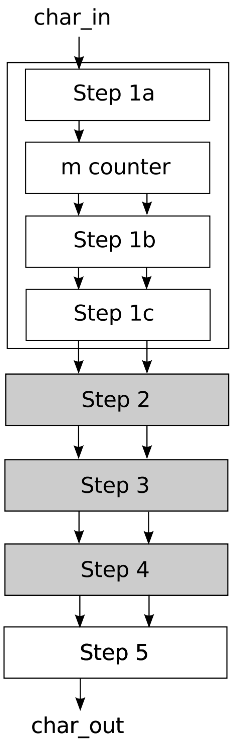

Figure 6.4: Porter stemmer design

	Area		**Speed**	
Device	ALUTs	Registers	Fmax	Critical path
Stratix IV GX EP4SGX230KF40C2	441	900	704	Master Clock

Table 6.3: Porter Stemmer resource utilization and speed

This leads to a higher average activity factor for each gate with respect to the Lovins design, whose resources are more selective with respect to the data they operate on. The result is greater power use. At 704 MHz, the Lovins stemmer consumes 63.04 mW, which translates to efficiencies of 5.597 mW/Gbps and 89.55 μW/MHz. The Porter stemmer consumes 20.9% more: 76.2 mW, translating to efficiencies of 6.765 mW/Gbps and 108.2 μW/MHz.

The power consumption and resource usage of this implemenation of the Porter stemmer is more sensitive to the width of the character bus than the Lovins design. At at clock

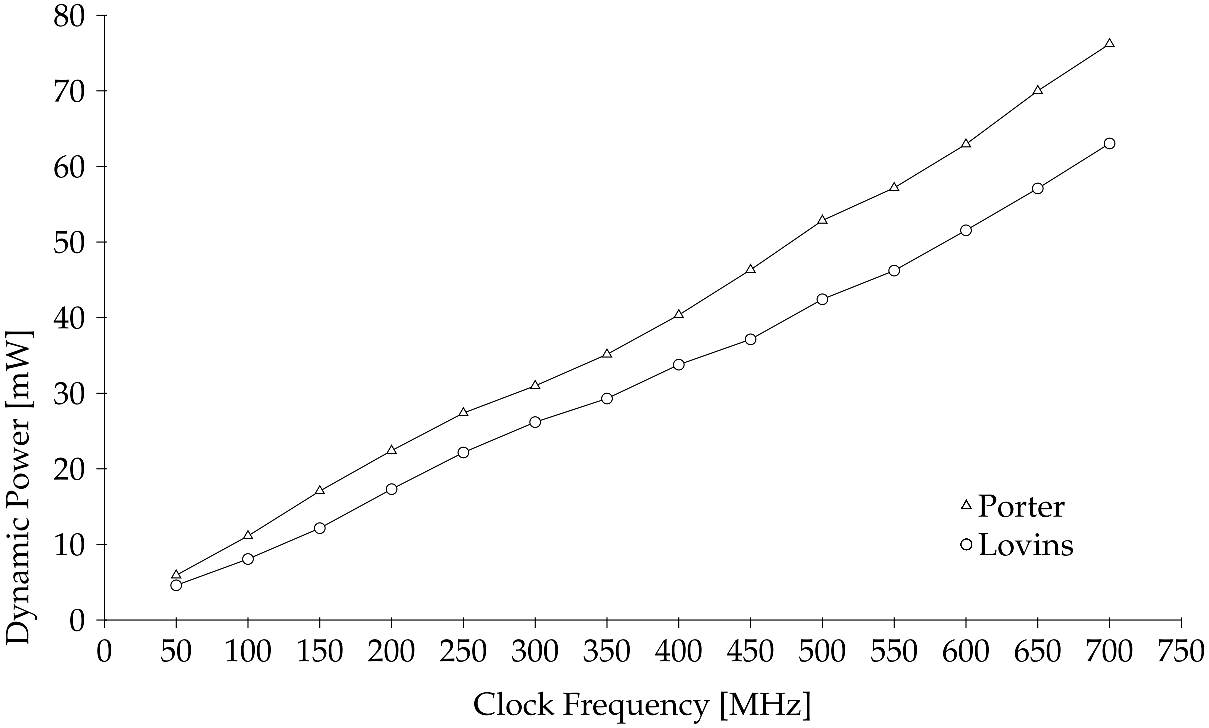

Figure 6.5: Lovins and Porter Stemmer Dynamic Power Consumption

frequency of 250 MHz, the 16–bit Porter datapath consumes 27.38 mW, while the 7–bit equivalent (which consequently only supports ASCII encoding), consumes 19.9 mW — 27.3% less. The difference in resource utilization is equally significant. The 7–bit datapath requires 353 ALUTs (20.0% fewer) and 564 Registers (37.3% fewer). The difference in power consumed by the FPGA's interfacing pins is minimal (2.6%) since English–language utilizes characters from beyond the ASCII more seldomly than other languages (bits above position 6 toggle very infrequently). The difference between the 16–bit and 7–bit datapaths for the Lovins Stemmer are less significant: the design with the 7–bit wide datapath consumes 15.1% less power, and requires 6.6% fewer ALUTs and 20.1% fewer registers.

7

Conclusions & Future Work

This thesis presented an overview of Information Retrieval and Natural Language Processing, as well as the design and implementation of four circuits for text preprocessing: *(i)* UTF–8 decoding, *(ii)* stop word filtering, and stemming with both *(iii)* Lovins' and *(iv)* Porter's algorithms. Table 7.1 summarizes each module's resource utilization, maximum speed and power efficiency. It was found that the Lovins stemmer design was more power efficient than the Porter stemmer since power consumption was not as tightly dependent on the width of the character bus. With respect to the stop word filter and Lovins stemmer, it was also found that there is a good correlation between power consumption and resource usage, and the number of characters being matched. This confirms Moscola's finding that the performance of this architecture does scale linearly with operating frequency and circuit size[MLC08].

	Area		**Speed**		**Power**	
Circuit	ALUTs	Registers	Fmax	P_{Fmax}	mw/Gbps	μW/MHz
UTF–8 Decoder	37	46	704	24.40	4.332	34.66
Stop Word Filter	314	664	704	45.36	4.05	64.80
Lovins Stemmer	609	1040	704	63.04	5.597	89.55
Porter Stemmer	441	900	704	76.20	6.765	108.2

Table 7.1: Circuit implementation summary

All circuits were able to achieve the maximum clock frequency supported by the target FPGA. There are two reasons for this. The first is extensive pipelining in the designs, which maintained no more than two levels of logic between registers. Design synthesis was able to map this logic into a single level of ALUTs, leaving wiring delay and other physical device constraints to limit the maximum clock frequency. The extent of the pipelining is evident in each circuit implementation by the greater number of registers over ALUTs. The second reason is that the use of large memory circuits directly in the datapath was avoided since memory read and write operations tend to have a higher latency than ALUTs. Given these two factors, it is likely that an equivalent ASIC design would be able to achieve an operational frequency far in excess of 1 GHz. Fast register design will be important for achieving the maximum possible operating speed.

7.1 Future Work

There are a number of interesting directions for future work, and they include: *(i)* interfacing with a computer network, *(ii)* comparing the power efficiency of the designs to their software equivalents, *(iii)* addressing the issue of text indexing in hardware *(iv)* as well as video indexing. Efficient memory and storage management on large networked platforms could also be investigated.

Interfacing with a computer network

Supporting access to a computer network is key to increasing the utility and access of the project to a larger group of users, and in doing so, aggregating the cost of the system across a larger pool of users. One future direction would be to support document transfer over FTP, TCP and IPv6 and integrate support for managing sessions and multiple users, perhaps through a soft processor on the FPGA such as Altera's NicheStack TCP/IP Stack[Alt09] or a custom solution that can sustain throughput in the multi-Gigabit range[SL04].

Comparison with software

A thorough comparison with an equivalent software implementation could be made to compare the energy efficiency of the designs detailed in this thesis with their software equivalents. A reasonable software equivalent to compare the performance of the programmable logic to is an information retrieval framework that facilitates full text indexing and searching, such as Lucene[The09b].

Comparison with GPU

A cost effective alternative to FPGAs is a Graphics Processing Unit (GPU). The price difference can be attributed to a significantly larger market, as well as stronger competition within that market. In recent years, Nvidia has developed a product line called Tesla, which enables general–purpose computing on graphics processing units (GPGPU)[Nvi10, LNOM08]. At the time of writing, they have been shown to significantly accelerate applications in protein interactions[SH10, JBC10], neuroscience[Sco10], computer vision[PQ10] and communications[AKBN09].

Indexing Text

The fifth and final module in the project overview presented in Figure 3.1 is a circuit that generates and maintains an index to all words would in all documents that are passed into the system. Such a module would facilitate document searching through statistical matching techniques based on vector space models or Latent Semantic Indexing. It could be tuned to different applications, such as information discovery, automated document classification, text summarization, relationship discovery, automatic generation of link charts of individuals and organizations, matching technical papers and grants with reviewers, online customer support, determining document authorship, automatic keyword annotation of images, understanding software source code, filtering spam, information visualization, essay scoring or literature-based discovery, amongst others.

In terms of approaches to hardware implementation, Bloom filters and their derivatives have been shown to yield good results in programmable logic applications requiring character sequence matching and indexing, such as in virus detection [HL09, DL06, SGBN06, DSTL06, DL05, HGSD09] and IP address look–ups for packet routing [NP08, SHKL09]. These space–efficient probabilistic data structures test the membership of a set[Blo70], but are not exact: queries can yield false positives, but not false negatives. By being space efficient, they can be placed in memory close to the indexing circuitry (typically on chip) or in high–bandwidth and low latency memory adjacent to the FPGA, minimizing memory access delay.

Nevertheless, the Bloom filters in the forms presented in the cited literature are suitable for the construction of indexes which implement boolean search (since testing set membership is largely sufficient for operators such as conjunction (*and*), disjunction (*or*), negation (*not*)). However, these approaches are not suitable for probabilistic searches that can rank results, since probabilistic search methods require additional information to be associated with the query, such as the frequency of word occurrence or the character positions at which the word is located. The second important issue is memory capacity: on–chip memory is limited, force the index to be restricted in size or be moved off chip. Likely some form of *counting* Bloom filters[SMV08] would need to be investigated.

Indexing Video

Video is a popular method for transferring information and is perhaps the most significant contribution to bandwidth usage and storage space in applications connected to the Internet. It is also computationally expensive to process and index in software. A hardware design may be able to both accelerate processing for network applications with real time processing constraints, as well as enable more processing on the actual content in addition to its metadata (eliminating redundancy, filtering, etc.).

Bibliography

[Ai09] Alias-i. Lingpipe. http://alias-i.com/lingpipe/index.html, December 2009.

[AKBN09] A.F. Abdelrazek, M. Kaschub, C. Blankenhorn, and M.C. Necker. A novel architecture using nvidia cuda to speed up simulation of multi-path fast fading channels. In *IEEE 69th Vehicular Technology Conference, 2009.*, pages 1–5, 2009.

[Alt09] Altera Corporation. Ethernet and the nichestack tcp/ip stack – nios ii edition. www.altera.com/literature/hb/nios2/n2sw_nii52013.pdf, November 2009.

[BKL09] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, 2009.

[BLK09] Steven Bird, Edward Loper, and Ewan Klein. Natural language toolkit. http://www.nltk.org, July 2009.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[Bri09] British National Corpus. British national corpus. http://www.natcorp.ox.ac.uk/, January 2009.

[BSMV06] Joao Bispo, Ioannis Sourdis, Joao M.P.Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 119–126, December 2006.

[Cho57] Noam Chomsky. *Syntactic Structures*. Mouton, 1957.

[Cir01] Cristian Raul Ciressan. *An FPGA-based Syntactic Parser for Large Size Real-Life Context-Free Grammars*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2001.

[CML06] Young H. Cho, James Moscola, and John W. Lockwood. Context-free-grammar based token tagger in reconfigurable devices. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, 2006.

[CML08] Young H. Cho, James Moscola, and John W. Lockwood. Reconfigurable content-based router using hardware-accelerated language parser. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(2):1–25, April 2008.

[CT94] William Cavnar and John M. Trenkle. N–gram–based text categorization. In *In Proceedings of SDAIR–94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[Dav09] Mark Davies. Corpus of contemporary american english. http://www.americancorpus.org/, 2009.

[Daw74] J.L. Dawson. Suffix removal and word conflation. *ALLC Bulletin*, 2:33–46, 1974.

[DL05] Sarang Dharmapurikar and John Lockwood. Fast and scalable pattern matching for content filtering. In *ANCS '05: Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pages 183–192, 2005.

[DL06] Sarang Dharmapurikar and John Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *Selected Areas in Communications, IEEE Journal on*, 24(10):1781–1792, October 2006.

[DSTL06] Sarang Dharmapurikar, Haoyu Song, Jonathan Turner, and John Lockwood. Fast packet classification using bloom filters. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 61–70, 2006.

[EAD09] EADS Defence and Security. Weblab: Open platform for processing multimedia documents. http://weblab-project.org/, December 2009.

[Fel98] Christiane Fellbaum. *WordNet: an electronic lexical database*. MIT Press, 1998.

[FF03] William B. Frakes and Christopher J. Fox. Strength and similarity of affix removal stemming algorithms. *SIGIR Forum*, 37(1):26–30, 2003.

[FJ06] Michael Freeman and Thimal Jayasooriya. A hardware ip-core for information retrieval. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 115–122, 2006.

[fLTA] Center for Language, Speech Technologies, and Applications. Freeling. http://www.lsi.upc.edu/~nlp/freeling/, December.

[Gro09a] The Stanford Natural Language Processing Group. Stanford nlp toolkit. http://nlp.stanford.edu/software/index.shtml, December 2009.

[Gro09b] UCLA Medical Imaging Informatics Group. Mii nlp toolkit. http://www.mii.ucla.edu/nlp/, December 2009.

[HGSD09] J. Harwayne-Gidansky, D. Stefan, and I. Dalal. Fpga-based soc for real-time network intrusion detection using counting bloom filters. In *Southeastcon, 2009. SOUTHEASTCON '09. IEEE*, pages 452–458, 5–8 2009.

[HL09] Johnny Tsung Lin Ho and Guy G.F. Lemieux. Perg-rx: A hardware pattern-matching engine supporting limited regular expressions. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 257–260. ACM, 2009.

[Hul96] David A. Hull. Stemming algorithms - a case study for detailed evaluation. *Journal of the American Society for Information Science*, 47:70–84, 1996.

[IE10] IDC and EMC. The digital universe decade - are you ready? http://www.emc.com/collateral/demos/microsites/idc-digital-universe/iview.htm, May 2010.

[Ins10] Institute of Electrical and Electronics Engineers. Ieee xplore. http://ieeexplore.ieee.org/Xplore/, March 2010.

[JBC10] A.C. Jacob, J.D. Buhler, and R.D. Chamberlain. Rapid rna folding: Analysis and acceleration of the zuker recurrence. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) 2010*, pages 87–94, 2010.

[JG07] Arpith Jacob and Maya Gokhale. Language classification using n–grams accelerated by fpga–based bloom filters. In *HPRCTA '07: Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications*, pages 31–37, 2007.

[JM08] Daniel Jurafsky and James Martin. *Speech and Language Processing*. Prentice Hall Series in Artificial Intelligence. Pearson Prentice–Hall, 2 edition, 2008.

[JP09] Weirong Jiang and Viktor K. Prasanna. Large-scale wire-speed packet classification on fpgas. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 219–228, 2009.

[KFC67] Henry Kucera, W. Nelson Francis, and John B. Carroll. *Computational Analysis of Present-Day American English*. Brown University Press, 1967.

[KMM00] Mark Kantrowitz, Behrang Mohit, and Vibhu Mittal. Stemming and its effects on tfidf ranking. In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 357–359, 2000.

[Kro93] Robert Krovetz. Viewing morphology as an interface process. pages 191–202. ACMPress, 1993.

[LE+06] John W. Lockwood, , Stephen G. Eick, Justin Mauger, John Byrnes, Ron Loui, Andrew Levine, Doyle J. Weishar, and Alan Ratner. Hardware accelerated algorithm for semantic processing of document streams. In *IEEE Aerospace Conference (Aero 06)*, 2006.

[Liu09] Hugo Liu. Montylingua. http://web.media.mit.edu/~hugo/montylingua/, December 2009.

[LLC09] Orchestr8 LLC. Alchemyapi - transforming text into knowledge. http://www.alchemyapi.com/, November 2009.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. In *IEEE Micro*, volume 28, pages 39–55, 2008.

[Lov68] Julie Beth Lovins. Development of a stemming algorithm. In *Mechanical Translation and Computational Linguistics*, volume 11, pages 22–31, June 1968.

[LPB06] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *Information Theory, 2006 IEEE International Symposium on*, pages 2774–2778, July 2006.

[McC09] Andrew Kachites McCallum. Machine learning for language toolkit. http://mallet.cs.umass.edu/, December 2009.

[MCL07] James Moscola, Young H. Cho, and John W. Lockwood. Hardware-accelerated parser for extraction of metadata in semantic network content. In *Aerospace Conference, 2007 IEEE*, pages 1–8, March 2007.

[MIFR+06] J.R. Méndez, E.L. Iglesias, F. Fdez-Riverola, F. D'iaz, and J.M. Corchado. *Current Topics in Artificial Intelligence*, volume 4177/2006, chapter Tokenising, Stemming and Stopword Removal on Anti-spam Filtering Domain, pages 449–458. Springer–Verlafg Berlin Heidelberg, 2006.

[Mik03] Andrei Mikheev. *The Oxford Handbook of Computational Linguistics*, chapter Text Segmentation, pages 201–218. Oxford University Press, 2003.

[MLC08] James Moscola, John Lockwood, and Young Cho. Reconfigurable content-based router using hardware-accelerated language parser. *ACM Transactions on Design Automation of Electronic Systems*, 13(2):1–25, 2008.

[MMS93] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: the penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[Moe06] Marie-Francine Moens. *Information Extraction Algorithms and Prospects in a Retrieval Context*. The Information Retrieval Series. Springer, 2006.

[Mor08] Tom Morton. Opennlp. http://opennlp.sourceforge.net/, November 2008.

[MTM$^+$99] Mitchell Marcus, Ann Taylor, Robert MacIntyre, Ann Bies, Constance Cooper, Mark Ferguson, and Alyson Littman. Penn treebank project. http://www.cis.upenn.edu/~treebank/, February 1999.

[NP08] A. Nikitakis and L. Papaefstathiou. A memory-efficient fpga-based classification engine. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 53–62, 14–15 2008.

[Nvi10] Nvidia Corporation. High performance computing (hpc) - supercomputing with nvidia tesla. http://www.nvidia.com/object/tesla_computing_solutions.html, July 2010.

[Pai90] Chris D. Paice. Another stemmer. *SIGIR Forum*, 24(3):56–61, 1990.

[Pet09] Georgios P. Petasis. The ellogon language engineering platform. http://www.ellogon.org, December 2009.

[Por80] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, July 1980.

[PQ10] S.P Ponce and F. Quek. Low-cost, high-speed computer vision using nvidia's cuda architecture. In *37th IEEE Applied Imagery Pattern Recognition Workshop, 2008.*, pages 1–7, 2010.

[RG09] The MARF Research and Development Group. The modular audio recognition framework. http://marf.sourceforge.net/, December 2009.

[Sco10] R. Scorcioni. Gpgpu implementation of a synaptically optimized, anatomically accurate spiking network simulator. In *2010 Biomedical Sciences and Engineering Conference (BSEC)*, pages 1–3, 2010.

[SDTL05] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 181–192, 2005.

[SGBN06] Dinesh C Suresh, Zhi Guo, Betul Buyukkurt, and Walid A. Najjar. Automatic compilation framework for bloom filter based intrusion detection. In *International Workshop On Applied Reconfigurable Computing – Lecture Notes in Computer Science*, pages 413–418, 2006.

[SH10] B. Sukhwani and M.C. Herbordt. Fast binding site mapping using gpus and cuda. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[SHKL09] Haoyu Song, Fang Hao, M. Kodialam, and T.V. Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *INFOCOM 2009, IEEE*, pages 2518–2526, 19–25 2009.

[SL04] David V. Schuehler and John W. Lockwood. A modular system for fpga-based tcp flow processing in high-speed networks. In *14th International Conference on Field Programmable Logic and Applications (FPL)*, pages 301–310, 2004.

[SMV08] E. Safi, A. Moshovos, and A. Veneris. L-cbf: A low-power, fast counting bloom filter architecture. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16:628–638, June 2008.

[Sou09] Sourcefire Incorporated. Snort. http://www.snort.org/, July 2009.

[ST08] Tanveer Siddiqui and U.S. Tiwary. *Natural Language Processing and Information Retrieval*. Oxford University Press, 2008.

[The09a] The Apache Software Foundation. Apache unstructured information management applications. http://incubator.apache.org/uima/index.html, December 2009.

[The09b] The Apache Software Foundation. Lucene. http://lucene.apache.org/, November 2009.

[The09c] The Unicode Consortium. *The Unicode Standard Version 5.2*. Unicode Consortium, 2009.

[The09d] The University of Sheffield. General architecture for text engineering. http://gate.ac.uk/, December 2009.

[Ult09] Ultralingua. Semantic search. http://ultralingua.com/en/semantic-search.htm, December 2009.

[VAM09] W. Vanderbauwhede, L. Azzopardi, and M. Moadeli. Fpga–accelerated information retrieval: High–efficiency document filtering. In *Proceedings of the 19th International Conference on Field Programmable Logic and Applications*, 2009.

[Wik09] Wikipedia. Cyc. http://en.wikipedia.org/wiki/Cyc, December 2009.

[Wik10] Wikipedia. Ascii. http://en.wikipedia.org/wiki/Ascii, March 2010.

[WIZD05] Sholom M. Weiss, Nitin Indurkhya, Tong Zhang, and Fred J. Damerau. *Text Mining: Predictive Methods for Analyzing Unstructured Information*. Springer, 2005.

[ZLC+03] Hua-Ping Zhang, Qun Liu, Xue-Qi Cheng, Hao Zhang, and Hong-Kui Yu. Chinese lexical analysis using hierarchical hidden markov model. In *Proceedings of the second SIGHAN workshop on Chinese language processing*, pages 63–70, 2003.

Appendicies

A

Text Encoding

Binary	Glyph	Binary	Glyph	Binary	Glyph
010 0000		100 0000	@	110 0000	‘
010 0001	!	100 0001	A	110 0001	a
010 0010	"	100 0010	B	110 0010	b
010 0011	#	100 0011	C	110 0011	c
010 0100	$	100 0100	D	110 0100	d
010 0101	%	100 0101	E	110 0101	e
010 0110	&	100 0110	F	110 0110	f
010 0111	’	100 0111	G	110 0111	g
010 1000	(	100 1000	H	110 1000	h
010 1001	)	100 1001	I	110 1001	i
010 1010	*	100 1010	J	110 1010	j
010 1011	+	100 1011	K	110 1011	k
010 1100	,	100 1100	L	110 1100	l
010 1101	-	100 1101	M	110 1101	m
010 1110	.	100 1110	N	110 1110	n
010 1111	/	100 1111	O	110 1111	o
011 0000	0	101 0000	P	111 0000	p
011 0001	1	101 0001	Q	111 0001	q
011 0010	2	101 0010	R	111 0010	r
011 0011	3	101 0011	S	111 0011	s
011 0100	4	101 0100	T	111 0100	t
011 0101	5	101 0101	U	111 0101	u
011 0110	6	101 0110	V	111 0110	v
011 0111	7	101 0111	W	111 0111	w
011 1000	8	101 1000	X	111 1000	x
011 1001	9	101 1001	Y	111 1001	y
011 1010	:	101 1010	Z	111 1010	z
011 1011	;	101 1011	[	111 1011	{
011 1100	<	101 1100	\	111 1100	|
011 1101	=	101 1101	]	111 1101	}
011 1110	>	101 1110	∧	111 1110	∼
011 1111	?	101 1111	_	111 1111	

Table A.1: Table of printable ASCII characters, reprinted from[Wik10]

B

Stop–Word Filtering

1 – 20	21 – 40	41 – 60	61 – 80	81 – 100
the	this	so	people	back
be	but	up	into	after
to	his	out	year	use
of	by	if	your	two
and	from	good	how	
a	they	who	some	our
in	we	get	could	work
that	say	which	them	first
have	her	go	see	well
I	she	me	other	way
it	or	when	than	even
for	an	make	then	new
not	will	can	now	want
on	my	like	look	because
with	one	time	only	any
he	all	no	come	these
as	would	just	its	give
you	there	him	over	day
do	their	know	think	most
at	what	take	also	us

Table B.1: The 100 most common words in the Oxford English Corpus

Nouns	Verbs	Adjectives
time	be	good
person	have	new
year	do	first
way	say	last
day	get	long
thing	make	great
man	go	little
world	know	own
life	take	other
hand	see	old
part	come	right
child	think	big
eye	look	high
woman	want	different
place	give	small
work	use	large
week	find	next
case	tell	early
point	ask	young
government	work	important
company	seem	few
number	feel	public
group	try	bad
problem	leave	same
fact	call	able

Table B.2: The 25 most common nouns, verbs and adjectives in the Oxford English Corpus

1st Person	**Verb Forms**	**Compound**	**Other**	**Other**
Singular	**& Auxillaries**	**Forms**	**Words**	**Words**
I	am	I’m	let’s	again
	is	you’re	that’s	further
me	are	he’s	who’s	then
my	was	she’s	what’s	once
the	were	it’s	here’s	
myself	be	we’re	there’s	here
	been	they’re	when’s	there
us	being	I’ve	where’s	when
our		you’ve	why’s	where
ours	have	we’ve	how’s	why
ourselves	has	they’ve		how
	had	I’d	a	
you	having	you’d	an	all
your		he’d		any
yours	do	she’d	and	both
yourself	does	we’d	but	each
yourselves	did	they’d	if	few
	doing	I’ll	or	more
he		you’ll	because	most
him	would	he’ll	as	other
his	should	she’ll	until	some
himself	could	we’ll	while	such
	ought	they’ll	of	
she			at	no
her		isn’t	by	nor
hers		aren’t	for	not
herself		wasn’t	with	only
		weren’t	about	own
it		hasn’t	against	same
its		haven’t	between	so
itself		hadn’t	into	than
		doesn’t	through	too
they		don’t	during	very
them		didn’t	before	
their			after	
theirs		won’t	above	
themselves		wouldn’t	below	
		shan’t	to	
what		shouldn’t	from	
which		can’t	up	
who		cannot	down	
whom		couldn’t	in	
this		mustn’t	out	
that			on	
these			off	
those			over	
			under	

Table B.3: Pronouns on Porter’s Stop–Word List

C

Natural Language Processing

Tag	Description	Example	Tag	Description	Example
CC	coordinating conjunction	*and, but, or*	**SYM**	symbol	+,%,&
CD	cardinal number	*one, two, three*	**TO**	“to”	*to*
DF	determiner	*a, the*	**UH**	interjection	*ah, oops*
EX	existential “there”	*there*	**VB**	verb, base form	*eat*
FW	foreign word	*mea culpa*	**VBD**	verb, past tense	*ate*
IN	preposition / sub. conj.	*of, in, by*	**VBG**	verb, gerund	*eating*
JJ	adjective	*yellow*	**VBN**	verb, past participle	*eaten*
JJR	adj., comparative	*bigger*	**VBP**	verb, non-3sg pres	*eat*
JJS	adj., superlative	*wildest*	**VBZ**	verb, 3sg pres	*eat*
LS	list item marker	*1, 2, One*	**WDT**	wh-determiner	*which, that*
MD	modal	*can, should*	**WP**	wh-pronoun	*what, who*
NN	noun, sing. or mass	*llama, snow*	**WP$**	possessive wh-	*whose*
NNS	noun, plural	*llamas*	**WRB**	wh-adverb	*how, where*
NNP	proper noun, singular	*IBM*	**$**	dollar sign	*$*
NNPS	proper noun, plural	*Carolinas*	**#**	pound sign	#
PDT	predeterminer	*all, both*	**“**	left quote	*‘ or “*
POS	possessive ending	*’s*	**”**	right quote	*’ or ”*
PRP	personal pronoun	*I, you, he*	**(**	left parenthesis	*[,({,<*
PRP$	possessive pronoun	*your, one’s*	**)**	right parenthesis	*],),},>*
RB	adverb	*quickly, never*	**,**	comma	*,*
RBR	adverb, comparative	*faster*	**.**	sentence-final punc.	*. ! ?*
RBS	adverb, superlative	*fastest*	**:**	mid-sentence punc.	*: ; . . . – -*
RP	particle	*up, off*			

Table C.1: Penn Treebank Part–of–Speech tags, reproduced from[JM08]

Expression	Description	Examples & Expansions
	Single character expressions	
.	any single character	spi.e matches “spice”, “spike”, etc.
\char	matches a nonalphanumeric *char* literally	* matches “*”
\n	newline character	
\r	carriage return character	
\t	tab character	
[...]	any single character listed in the brackets	[abc] matches “a”, “b”, or “c”
[...-...]	any single character in the range	[0-9] matches “0” or “1” ... or “9”
[^...]	any single character not listed	[^sS] matches one character that is neither “s” or “S”
[^...-...]	any single character not in the range	[^A-Z] matches one character that is not an uppercase letter
	Anchors/Expressions with match positions	
\^	beginning of line	
\$	end of line	
\b	word boundary	nt\b matches “nt” in “paint” but not “pants”
\B	word non-boundary	all\B matches “all” in “ally” but not in “wall”
	Counters/Expressions which quantify previous expressions	
*	zero or more of the previous r.e.	a* matches “”, “a”, “aa”, ...
+	one or more of the previous r.e.	a+ matches “a”, “aa”, “aaa”, ...
?	exactly one or zero of thre previous r.e.	colou?r matches “color” or “colour”
{n}	*n* of the previous r.e.	a{4} matches “aaaa”
{n,m}	from *n* to *m* of previous r.e.	
n,	at least *n* of previous r.e.	
.*	any string of characters	
(...)	grouping for precedence	
...|...	matches either of neighbour r.e.s	(dog)|(cat) matches “dog” or “cat”
	Shortcuts	
\d	any digit	[0-9]
\D	any non-digit	[^0-9]
\w	any alphanumeric/underscore	[a-zA-Z0-9_]
\W	any non-alphanumeric	[^a-zA-Z0-9_]
\s	whitespace	[␣\r\t\n\f]
\S	non-whitespace	[^␣\r\t\n\f]

Table C.2: Regular Expression syntax, reproduced from[JM08]

D

Supplimentary Material on Stemming

The following is a collection of supplimentary material on word stemming.

D.1 Lovins Algorithm

Length:	1	2	3	4	5	6	7	8	9	10	11
Frequency:	6	18	39	48	67	39	40	13	17	4	3

Table D.1: Frequency of Lovins stem lengths (in characters)

1	remove one of double *b,d,g,l,m,n,p,r,s,t*
2	iev → ief
3	uct → uc
4	umpt → um
5	rpt → rb
6	urs → ur
7	istr → ister
7a	metr → meter
8	olv → olut
9	ul → l; except following *a,o,i*
10	bex → bic
11	dex → dic
12	pex → pic
13	tex → tic
14	ax → ac
15	ex → ec
16	ix → ic
17	lux → luc
18	uad → uas
19	vad → vas
20	cid → cis
21	lid → lis
22	erid → eris
23	pand → pans
24	end → ens; except following *s*
25	ond → ons
26	lud → lus
27	rud → rus
28	her → hes; except following *p,t*
29	mit → mis
30	ent → ens; except following *m*
31	ert → ers
32	et → es; except following *n*
33	yt → ys
34	yz → ys

Table D.2: Listing of transformation rules on stem terminations.

A	199	Minimum stem length = 2
B	30	Minimum stem length = 3
C	6	Minimum stem length = 4
D	1	Minimum stem length = 5
E	15	Do not remove ending after _e_
F	8	Minimum stem length = 3 and do not remove ending after _e_
G	2	Minimum stem length = 3 and remove ending only after _f_
H	1	Remove ending only after _t_ or _ll_
I	3	Do not remove ending after _o_ or _e_
J	1	Do not remove ending after _a_ or _e_
K	1	Minimum stem length = 3 and remove ending only after _l_, _i_ or _u.e_
L	2	Do not remove ending after _u_, _x_ or _s_, unless _s_ follows _o_
M	2	Do not remove ending after _a_, _c_, _e_ or _m_
N	2	Minimum stem length = 4 after _s**_, elsewhere = 3
O	1	Remove ending only after _l_ or _i_
P	1	Do not remove ending after _c_
Q	1	Minimum stem length = 3 & do not remove ending after _l_ or _n_
R	2	Remove ending only after _n_ or _r_
S	1	Remove ending only after _dr_ or _t_, unless _t_ follows _t_
T	1	Remove ending only after _s_ or _t_, unless _t_ follows _o_
U	1	Remove ending only after _l_, _m_, _n_ or _r_
V	1	Remove ending only after _c_
W	1	Do not remove ending after _s_ or _u_
X	1	Remove ending only after _l_, _i_ or _u*e_
Y	4	Remove ending only after _in_
Z	1	Do not remove ending after _f_
AA	1	Remove ending only after _d_, _f_, _ph_, _th_, _l_, _er_, _or_, _es_ or _t_
BB	3	Minimum stem length = 3 and do not remove ending after _met_ or _ryst_
CC	1	Remove ending only after _l_

Table D.3: Context–sensitive rules and their frequency. The implicit assumption in each condition is that the minimum stem length is 2.

11	alistically	B	arizability	A	izationally	B						
10	antialness	A	arisations	A	arizations	A	entialness	A				
9	allically	C	antaneous	A	antiality	A	arisation	A	arization	A	ationally	B
	ativeness	A	eableness	E	entations	A	entiality	A	entialize	A	entiation	A
	ionalness	A	istically	A	itousness	A	izability	A	izational	A		
8	ableness	A	arizable	A	entation	A	entially	A	eousness	A	ibleness	A
	icalness	A	ionalism	A	ionality	A	ionalize	A	iousness	A	izations	A
	lessness	A										
7	ability	A	aically	A	alistic	B	alities	A	ariness	E	aristic	A
	arizing	A	ateness	A	atingly	A	ational	B	atively	A	ativism	A
	elihood	E	encible	A	entally	A	entials	A	entiate	A	entness	A
	fulness	A	ibility	A	icalism	A	icalist	A	icality	A	icalize	A
	ication	G	icianry	A	ination	A	ingness	A	ionally	A	isation	A
	ishness	A	istical	A	iteness	A	iveness	A	ivistic	A	ivities	A
	ization	F	izement	A	oidally	A	ousness	A				
6	aceous	A	acious	B	action	G	alness	A	ancial	A	ancies	A
	ancing	B	ariser	A	arized	A	arizer	A	atable	A	ations	B
	atives	A	eature	Z	efully	A	encies	A	encing	A	ential	A
	enting	C	entist	A	eously	A	ialist	A	iality	A	ialize	A
	ically	A	icance	A	icians	A	icists	A	ifully	A	ionals	A
	ionate	D	ioning	A	ionist	A	iously	A	istics	A	izable	E
	lessly	A	nesses	A	oidism	A						
5	acies	A	acity	A	aging	B	aical	A	alist	A	alism	B
	ality	A	alize	A	allic	BB	anced	B	ances	B	antic	C
	arial	A	aries	A	arily	A	arity	B	arize	A	aroid	A
	ately	A	ating	I	ation	B	ative	A	ators	A	atory	A
	ature	E	early	Y	ehood	A	eless	A	elity	A	ement	A
	enced	A	ences	A	eness	E	ening	E	ental	A	ented	C
	ently	A	fully	A	ially	A	icant	A	ician	A	icide	A
	icism	A	icist	A	icity	A	idine	I	iedly	A	ihood	A
	inate	A	iness	A	ingly	B	inism	J	inity	CC	ional	A
	ioned	A	ished	A	istic	A	ities	A	itous	A	ively	A
	ivity	A	izers	F	izing	F	oidal	A	oides	A	otide	A
	ously	A										
4	able	A	ably	A	ages	B	ally	B	ance	B	ancy	B
	ants	B	aric	A	arly	K	ated	I	ates	A	atic	B
	ator	A	ealy	Y	edly	E	eful	A	eity	A	ence	A
	ency	A	ened	E	enly	E	eous	A	hood	A	ials	A
	ians	A	ible	A	ibly	A	ical	A	ides	L	iers	A
	iful	A	ines	M	ings	N	ions	B	ious	A	isms	B
	ists	A	itic	H	ized	F	izer	F	less	A	lily	A
	ness	A	ogen	A	ward	A	wise	A	ying	B	yish	A
3	acy	A	age	B	aic	A	als	BB	ant	B	ars	O
	ary	F	ata	A	ate	A	eal	Y	ear	Y	ely	E
	ene	E	ent	C	ery	E	ese	A	ful	A	ial	A
	ian	A	ics	A	ide	L	ied	A	ier	A	ies	P
	ily	A	ine	M	ing	N	ion	Q	ish	C	ism	B
	ist	A	ite	AA	ity	A	ium	A	ive	A	ize	F
	oid	A	one	R	ous	A						
2	ae	A	al	BB	ar	X	as	B	ed	E	en	F
	es	E	ia	A	ic	A	is	A	ly	B	on	S
	or	T	um	U	us	V	yl	R	s’	A	’s	A
1	a	A	e	A	i	A	o	A	s	W	y	B

Table D.4: Listing of all 294 stems and their associated conditions, sorted by character length.

D.2 Porter Algorithm

Step 1: plurals and past participles

Step 1a			Step 1b		
sses	→	*ss*	$(m > 0)$ & *eed*	→	*ee*
ies	→	*i*	$(\ast v \ast)$ & *ed*	→	`null`
ss	→	*ss*	$(\ast v \ast)$ & *ing*	→	`null`
s	→	`null`			

If either the second or third rule in Step 1b is executed, the following is done:

at	→	*ate*
bl	→	*ble*
iz	→	*ize*
$\ast d$ and not ($\ast L$ or $\ast S$ or $\ast Z$)	→	single letter
$m = 1$ & $\ast o$	→	*e*

Step 1c		
$(\ast v \ast)$ *y*	→	*i*

Step 2

$(m > 0)$ *ational*	→	*ate*	$(m > 0)$ *tional*	→	*tion*
$(m > 0)$ *enci*	→	*ence*	$(m > 0)$ *anci*	→	*ance*
$(m > 0)$ *izer*	→	*ize*	$(m > 0)$ *abli*	→	*able*
$(m > 0)$ *alli*	→	*al*	$(m > 0)$ *entli*	→	*ent*
$(m > 0)$ *eli*	→	*e*	$(m > 0)$ *ousli*	→	*ous*
$(m > 0)$ *ization*	→	*ize*	$(m > 0)$ *ation*	→	*ate*
$(m > 0)$ *ator*	→	*ate*	$(m > 0)$ *alism*	→	*al*
$(m > 0)$ *iveness*	→	*ive*	$(m > 0)$ *fulness*	→	*ful*
$(m > 0)$ *ousness*	→	*ous*	$(m > 0)$ *aliti*	→	*al*
$(m > 0)$ *iviti*	→	*ive*	$(m > 0)$ *biliti*	→	*ble*

Step 3

$(m > 0)$ *icate*	→	*ic*	$(m > 0)$ *ative*	→	`null`
$(m > 0)$ *alize*	→	*al*	$(m > 0)$ *iciti*	→	*ic*
$(m > 0)$ *ical*	→	*ic*	$(m > 0)$ *ful*	→	`null`
$(m > 0)$ *ness*	→	*ic*			

Step 4

$(m > 1)$ *al*	→	`null`	$(m > 1)$ *ance*	→	`null`
$(m > 1)$ *ence*	→	`null`	$(m > 1)$ *er*	→	`null`
$(m > 1)$ *ic*	→	`null`	$(m > 1)$ *able*	→	`null`
$(m > 1)$ *ible*	→	`null`	$(m > 1)$ *ant*	→	`null`
$(m > 1)$ *ement*	→	`null`	$(m > 1)$ *ment*	→	`null`
$(m > 1)$ *ent*	→	`null`	$(m > 1)$ and $(*S$ or $*T)$ *ion*	→	`null`
$(m > 1)$ *ou*	→	`null`	$(m > 1)$ *ism*	→	`null`
$(m > 1)$ *ate*	→	`null`	$(m > 1)$ *iti*	→	`null`
$(m > 1)$ *ous*	→	`null`	$(m > 1)$ *ive*	→	`null`
$(m > 1)$ *ize*	→	`null`			

Step 5: correcting the endings

Step 5a			Step 5b		
$(m > 1)$ *e*	→	`null`	$(m > 1$ and $*d$ and $*L)$	→	single letter
$(m = 1$ and not $*o)$ *e*	→	`null`			