Department of Electrical and Computer Engineering

Hardware-Based Text-to-Braille Translation

Xuan Zhang

This thesis is presented for the Degree of

Master of Engineering

of

Curtin University of Technology

July 2007

Declaration

To the best of my knowledge and belief this thesis contains no material previously published by any other person except where due acknowledgment has been made.

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university.

Signature:

Date:

Abstract

Braille, as a special written method of communication for the blind, has been globally accepted for years. It gives blind people another chance to learn and communicate more efficiently with the rest of the world. It also makes possible the translation of printed languages into a written language which is recognisable for blind people. Recently, Braille is experiencing a decreasing popularity due to the use of alternative technologies, like speech synthesis. However, as a form of literacy, Braille is still playing a significant role in the education of people with visual impairments. With the development of electronic technology, Braille turned out to be well suited to computer-aided production because of its coded forms. Software based text-to-Braille translation has been proved to be a successful solution in Assistive Technology (AT). However, the feasibility and advantages of the algorithm reconfiguration based on hardware implementation have rarely been substantially discussed. A hardware-based translation system with algorithm reconfiguration is able to supply greater throughput than a software-based system. Further, it is also expected as a single component integrated in a multi-functional Braille system on a chip. Therefore, this thesis presents the development of a system for text-to-Braille translation implemented in hardware. Differing from most commercial methods, this translator is able to carry out the translation in hardware instead of using software. To find a particular translation algorithm which is suitable for a hardware-based solution, the history of, and previous contributions to Braille translation are introduced and discussed. It is concluded that Markov systems, a formal language theory, were highly suitable for application to hardware based Braille translation. Furthermore, the text-to-Braille algorithm is reconfigured to achieve parallel processing to accelerate the translation speed. Characteristics and advantages of Field

Programmable Gate Arrays (FPGAs), and application of Very High Speed Integrated Circuit Hardware Description Language (VHDL) are introduced to explain how the translating algorithm can be transformed to hardware. Using a Xilinx hardware development platform, the algorithm for text-to-Braille translation is implemented and the structure of the translator is described hierarchically.

Key words

Braille translation, Assistive technology, Markov theory, Algorithm reconfiguration, FPGA, VHDL

Acknowledgements

I would like to thank my supervisor, Cesar Ortega-Sanchez, and co-supervisor, Iain Murray, for their continued help and direction. Without them, it would have been difficult for me to achieve all that I have done. I also want to say thanks to Dr. Jonathen K. Millen, a friend from America. The technical report on DOTSYS II he supplied was very helpful and important for this research project. Thanks to Leigh Harrison, for his support in designing the Braille keyboard.

Nomenclature

AT	Assistive Technology
FPGA	Field Programmable Gate Array
VHDL	Very high speed integrated circuit Hardware Description Language
ICEB	International Council on English Braille
UEB	Unified English Braille
ASCII	American Standard Code for Information Interchange
FSM	Finite State Machine
WHO	World Health Organization
DFA	Deterministic Finite Acceptor
PLD	Programmable Logic Device
RAM	Random Access Memory
PROM	Programmable Read-Only Memory
EPROM	Erasable Programmable Read Only Memory
PLA	Programmable Logic Array
FPLA	Field-Programmable Logic Array
PAL	Programmable Array Logic
SPLD	Simple Programmable Logic Device
CPLD	Complex Programmable Logic Device
FPD	Field-Programmable Devices
MPGA	Mask-Programmable Gate Arrays
NRE	Non-Recurring Engineering
IC	Integrated Circuits
ASIC	Application Specific Integrated Circuits
LUT	Look-up Table
CLB	Configurable Logic Block
MUX	Multiplexer
ALU	Arithmetic Logic Unit
EMI	Electromagnetic Interference
RC	Reconfigurable Computing
IEEE	Institute of Electrical and Electronics Engineers
RTL	Register Transfer Level
UDM-PD	Universal Design Methodology for Programmable Devices
UART	Universal Asynchronous Receiver/Transmitter
SOC	System On Chip

Table of Contents

Table of Figures

Tables

1. Introduction

According to the latest statistical data on the magnitute of blindness and visual impairment from World Health Organization (WHO), more than 161 million people around the world were visually impaired, of whom 124 million people had low vision and 37 million were blind [1]. Many of them rely on Braille as a tool of learning and communication. Although Braille dots still do not resemble print letters, Braille has been adapted to almost every language in the world and remains the major medium of literacy for blind people everywhere [2]. Since Louis Braille published his first embossed Braille book in 1829, millions of books have been published in Braille for people with visual impairment [3].

Recently, Braille is suffering descreasing popularity due to the use of alternative technologies, like speech synthesis [4]. However, as a form of literacy, Braille is still playing a significant role in the education of people with visual impairments. On the other hand, reading straight from the text can avoid potential errors or problems like indecipherable meanings or misspelling caused by speech synthesis. Therefore, Braille should still be a critical part of blind education and culture [5].

Automatic Braille Translation is a popular topic in the AT of visual impairment and has been wildly discussed and analysed since 1960s [6-8]. Currently, there are some commercially available programs and other computer-assisted applications which specialise in Braille translation. Some of these use personal computers to achieve translation and other functions, such as Duxbury, the most popular multi-language Braille translation software [8]. In this case, the speed of translation tends to be strongly related to particular computers utilised in the process. This kind of software is mainly designed for users with nomal eyesight, so it has low accessibility for visually impaired users and are used by transcribers translating existing print texts. They have to read the computer screen by using some AT tools, such as screen-reading software or Braille displays. There are also some portable devices specially designed for blind users which can perform text-to-Braille translation, such as Mountbatten Brailler [9]. These devices are based on a microcontroller running a translating program. As a small computer on a chip, a microcontroller is also able to perform multi-functional tasks, however, because a microcontroller is designed for general purposes, and the operations are based on sequential executions of instructions [10], therefore it may not be fast enough to perform mass translations of text documents. However, the advent of FPGAs made it possible to build a faster and stable hardware-based translation system which can also be integrated into a portable device, at a more affordable cost when compared to personal computers. To do this, the translating algorithm needs to be reconfigured so that the design can be applied to a parallel architecture in FPGAs. A hardware based translation system implemented in a FPGA is able to work as a single component, supplying greater throughput, and it also can be used as a module which is integrated in a universal Braille embedded system on a chip (SOC), supplying multi-functions. All of these components of the SOC will be integrated in one FPGA.

In Chapter 2, some history about Braille will be presented and the main difficulties for text-to-Braille translation are discussed. In Chapter 3, the text-Braille translation problem is discussed using formal language. Furthermore, several translation systems are given and comparisons are made to find out an applicable method of implementing hardware-based Braille translation. In Chapter 4 some essential knowledge of FPGAs including structures and some important characteristics will be presented. Also, another powerful tool, VHDL is introduced. Based on the information presented in Chapter 2, 3 and 4, a hardware based solution for Braille translation is discussed in Chapter 5. An implementation of fast translation based on FPGAs is explained, and some test results are included as well in this section. This section also includes how a text-to-Braille translator is integrated. Some conclusions and possibilities for future work are given in Chapter 6.

2. Background of Braille

2.1 Overall Description

The history of Braille can be traced back to the 1800s. The following text illustrates the origins of Braille, and it was published on the website of the Duxbury Systems Company [8]. “It was a French army captain, Charles Barbier de la Serre, who invented the basic technique of using raised dots for tactile writing and reading. His original objective was to allow soldiers to compose and read messages at night without illumination. Barbier later adapted the system and presented it to the Institution for Blind Youth, hoping that it would be officially adopted there. He called the system Sonography, because it represented words according to sound rather than spelling. However, Barbier's system was too complex for soldiers to learn. Based on Barbier’s system, in 1821, the young Frenchman, Louis Braille developed the system known as Braille that is widely used by blind people to read and write”.

Each Braille character or "cell" is made up of 6 dot positions, shown in Figure 1, arranged in a rectangle comprising 2 columns of 3 dots each [11-13]. A dot may be raised at any of the 6 positions, so, counting the space, in which no dot is raised, there are 64 such combinations in total (that is, 2 to the power of 6). “There is no intentional relation between the arrangement of dots in a cell and the shape of the corresponding ink-print character” [11]. “For reference purposes, a particular combination may be described by naming the positions where dots are raised, the positions being universally numbered 1 through 3 from top to bottom on the left, and 4 through 6 from top to bottom on the right.” For example, dots 1-3-4 would describe a cell with three dots raised, at the top and bottom in the left column and on top of the right column [8].

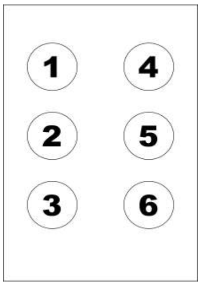


Figure 1. Braille Cell

The original Braille is a quite straightforward dots system which includes characters for each letter of the alphabet, punctuation marks, and numerals. This corresponds to Grade 1 Braille, where there is nearly a one-to-one correspondence between letters and Braille cells. Figure 2 [14] shows several single-cell letter codes.

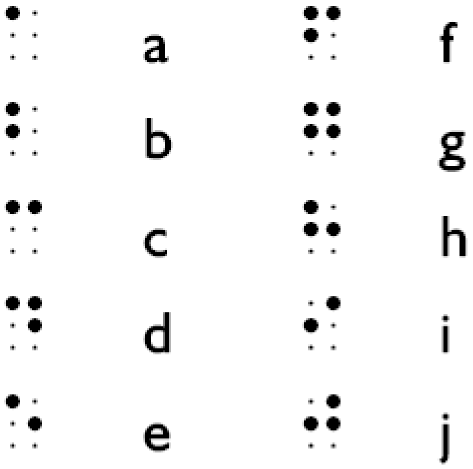


Figure 2. Single-Cell Letter Codes in Braille

Although it is clearly easy to transcribe Braille by simply substituting the equivalent Braille character for its printed equivalent using Grade 1 Braille, such a character-by-character transcription is used only by beginners [5] [12] [15] and the process is significantly time-consuming. On the other hand, “the size of the Braille cell is such that only about 25 lines of about 40 cells each, that is 1000 characters, can fit on a page of the usual size, which is the size of A4 page. This contrasts with the 3500 or so characters that will fit on a standard, smaller, typed page. Moreover, Braille paper must be much heavier to hold the dots, and the dots themselves considerably increase the effective thickness of a page [13]. The result is that paper Braille is very bulky. To mitigate this problem somewhat, most larger Braille books are published in "interpoint", that is with the embossing done on both sides of each sheet, with a slight diagonal offset to prevent the dots on the two sides from interfering with each other. But even in interpoint, a standard desk dictionary is likely to occupy a whole bookcase in Braille" [8].

Partly because of the bulk problem, and partly to improve the speed of writing and reading, the literary Braille codes for English and many other languages employ "contractions" that substitute shorter sequences for the full spelling of commonly-occurring letter groups. When contractions are used, the Braille is usually called "Grade 2," in contrast to "Grade 1" transcriptions where all words are spelled out letter-for-letter. In English-language Braille, for which 189 contractions have been developed, almost all Braille is written in Grade 2 [17]. Therefore, many Braille cells have multiple meanings. For example, the same cell which stands for the word "but" also means the single letter 'b'. Some examples of Grade 2 Braille cells are shown in Figure 3. Grade 2 Braille can contain more information, and therefore it can be read and produced much faster than Grade 1 Braille [14].

	Grade 1	Grade 2
⠃	B	BUT
⠉	C	CAN
⠙	D	DO
⠟	P	PEOPLE
⠏	Q	QUITE

Figure 3. Some contracted codes in Grade 2 Braille

Grade 2 Braille, although an effective way for the blind to learn and communicate, had not been accepted and utilised widely. One reason is Grade 2 Braille has complex rules about how to use contractions, and also because translation between Braille and text was very time-consuming and expensive [16]. But with the development of digital technology and innovations in Braille education, Grade 2 Braille has been accepted as one of most important ways for the blind to learn and communicate.

2.2 Coding Difficulties in Grade 2 Braille

Controversies continued for years among Braille rule makers which concentrated on the strings of letters to be contracted and those particular rules for specifying contractions [11]. Some of them insisted that particular strings of letters should always be contracted, regardless of the context or syllable boundaries; while the other group, argued that constraints of syllable boundaries must be considered. For instance, the string “dis” should only be used when these letters form the first syllable of a word, so this string will be contracted in the word “distrust”, but will be separated in “dish”. Another example is that the rules for "syllabification" would not allow the "ea" letter string in "react" to be contracted, whereas the "sequence" proponents would allow the contraction.

However, researchers realised that the necessity for a unified English Braille coding system could not be ignored, because the unified Braille is able to harmonise literary and technical codes into a systematic representation of print characters by unique and unambiguous Braille symbols [18][21]. Therefore, the International Council on English Braille (ICEB) was established in 1991 to develop the Unified English Braille (UEB). “In April 2004, the ICEB General Assembly declared Unified English Braille to be substantially complete and that it could be recognized as an international standard and considered for adoption by individual countries” [18]. If sequence rules are used, and syllable constraints are ignored, Braille translation becomes straightforward and easily implemented. However, in both British and American Braille syllabification was put into consideration [18] [19]. It resulted in a more complex system and a big increase in translation rules. Therefore context sensitivity becomes a very important characteristics of Grade 2 Braille [5] [20].

The main reason for using syllable constraints is that these linguistic units can easily be recognised as function units, which are helpful for the blind in comprehending Braille codes. Generally, these syllable units are of two types. Some are morphs, or basic lexical units of the language [11]. These include prefixes (com-, be-, mini-); roots, both free and bound (snow, boat, house, -turb, -ceive); derivational suffixes which affect the meaning of a word (-dom, -ness, -ship, -al) and inflectional suffixes, which affect the grammatical role of a word (-s, -ed, -ing). The second type of linguistic unit which is contracted is the cluster. There are both consonant clusters (st, sh, ch, chr, fth) and vowel clusters (ea, ou, ai). Many of these units are also contracted in Braille [11].

Because of the complexities of defining Grade 2 Braille rules, the translation between text and Braille becomes very difficult to implement. Generally, the rules are based on the position of a letter sequence in a word, pronunciation of the letter string, and syllabification of the word.

According to the British Grade 2 Braille rules, letters of the alphabet are also used in Braille to represent whole words; these are referred to as "wordsigns" [14]. As shown in Figure 3, the Braille cell for the letter 'b' is also a wordsign standing for "but". In this particular case, the introduction of a wordsign will not increase any difficulties in translation. However, some cells representing contractions also can be used as wordsigns. For instance, the contraction "ch" is also a wordsign for "child". The contraction "th" also means "this". In this case, priorities between contraction strings and wordsigns have to be differentiated. Another example of priority difficulties is with the contraction "th" and wordsign "the". Here, capital letters are used indicating strings to be contracted. When "th" appears in the word "THEn", instead of using "th", "the" has to be used. Another example is that the contraction "ou" is also used as a wordsign: it stands for "out", but it may only be used where it represents the whole word and where no other letters are added to it. Therefore, "out" can be treated as wordsign but "outside" cannot.

In British Grade 2 Braille, most wordsigns have a higher priority than letters of the alphabet [14]. However, there are some particular cases where wordsigns may not be used. Wordsigns “to”, “into”, and “by”, for instance, may only be expressed by their respective wordsigns when they can be written adjoining a word that follows; where no word immediately follows, or where a word does follow but the sense does not permit this joining up, they may not be expressed by wordsigns. Therefore, in the following instances “to” and “by” must be written out, and “into” written “INto”:

- where any one of these words occurs at the end of a Braille line, or immediately before a punctuation mark;
- where any one of these words is followed by a conjunction, such as “and”, “or”, etc. For example “to and for”, “by and by”, or “by and large”;
- where the sense indicates that a slight pause is made after any one of these words, even though there is no comma to mark it;
- where any one of these words is joined to another word by a hyphen to form a compound word.

The main difficulty of Grade 2 Braille is in relation to the usage of string contractions. Some string contractions are related to their positions in a word. For instance, contractions “er”, “ed”, and “ou” can be used in any part of a word. Yet, “ing”, and “ble” may be used in any part of a word except at the beginning of the word. Contractions “be”, “con”, “dis”, and “com” may only be used when they form the first syllable at the beginning of a word. Syllable boundaries are another important elements which are considered in determining string contractions. The same example can be used: "be", "con", and "dis" can be contracted only if they relate to syllables at the beginning of a word. Thus “CONcept”, but not “cone”; “DISturb”, but not “disc”, “Berate”, but not “bell”. Another example is the contraction “ea”: “lEAd” can be contracted, but “react” can not.

There are cases where two contractions overlap, a particular contraction has to be carefully chosen . For example, in words containing the letters “THEd” and “THEr”, the contraction “the” is used in preference to the contraction “th” and “ed” or “er”. The contraction “ea” is always to be used in preference to “ar”, except when “ea” occurs at the beginning of a word. Thus, not fEAr, but feAR, not lEArn, but leARn.

Some general contraction rules are appended as a guide to their use. In the examples given below, the letters that may not be contracted are italicised, the contractions are written as capitals:

Contractions may not be used:

- to bridge the components of a compound word, as, cAR*th*orse.
- to bridge a prefix to an English root word, as, r*ea*dmit, or pr*eA*Range.
- which would upset the usual pronunciation of syllables, as, a*sT*Hma.

There are other aspects of written language that create complexity in Braille as well. It is difficult for blind people to detect numbers and capitalised letters, because each single Braille code normally has several representations. Therefore, compound punctuation signs are used to denote particular punctuation, or special characters such as numbers or single letters. For instance, in Braille, letters used for special purposes, such as to denote Roman Numbers, or to designate persons or objects, must be immediately preceded by the Letter Sign, in order to show that they are being so used, and not as wordsign. Arabic figures 1-9 and 0 are represented in Braille by the letters A-I and J respectively, when they are immediately preceded by the numeral signs.

3. Translation Algorithm

The coding difficulties in Grade 2 Braille make it very difficult to procduce elaborate conversion programs, because it is a difficult and time consuming job to describe contractions by using rules. Therefore, the production of rules for the Grade 2 Braille becomes the main issue in text-Braille translations.

In this section, the text-Braille translation algorithm is going to be discussed by using essential concepts of formal language. Furthermore, several translation systems will be described, and comparisons are also going to be made, to find the most suitable method of implementing hardware-based Braille translation.

3.1 Basic Concepts

When translation between text and Braille is considered, there are two kinds of alphabets involved. One is the alphabet of Latin letters, Arabic numerals, punctuation marks, and some special symbols. The other includes 64 Braille codes. However, instead of using dot representations, there is a subset of the American Standard Code for Information Interchange (ASCII) character set which uses 64 of the printable ASCII characters to represent all possible six-dot combinations of Braille. This subset is called the Braille ASCII set, or computer Braille set [15].

Braille ASCII uses the 64 ASCII characters between 32 and 95 inclusive. All capital letters in ASCII correspond to their equivalent values in Braille. Unlike standard print, all letters in Braille are lower-case by default, unless otherwise specified by preceding them with a capitalisation symbol. The Braille ASCII table is shown in Appendix I.

Therefore, the translation process can be regarded as a symbol transformation from one kind of ASCII alphabet set into the other.

The two alphabets can be defined as follows:

- A finite, nonempty set $\sum_1$ of Latin letters, Arabic numerals, punctuation marks, and special marks is called Alphabet 1:

$\sum_1$= {A, B, C, … , X, Y, Z, a, b, c, … , x, y, z, 0, 1, 2 , … , 9, ., ,, !, #, $, space …}

- A finite, nonempty set $\sum_2$ of 64 Braille ASCII characters is called Alphabet 2:

$\sum_2$ = {space, !, ", #, $, %, ^, … , _ }

For example, $w_1=x_0\ x_1 \ldots x_m$ is a string, where $x_0\ x_1 \ldots x_m$ are elements of $\sum_1$. To translate string $w_{1,}$ a particular group of rules have to be applied. The translation results should be another string $w_2=y_0\ y_1 \ldots y_n$ ($n<=m$), where $y_0\ y_1 \ldots y_n$ are elements of $\sum_2$. 'm' will be equal to or greater than 'n' because Braille ASCII codes must be reduced after applying contractions rules.

The text to Grade 2 Braille translating process can be described as follows:

$$x_0\, x_1 \ldots x_m \xrightarrow{\text{Rules}} y_0\, y_1 \ldots y_n$$

Likewise, the Braille to text translation is given below:

$$y_0\, y_1 \ldots y_n \xrightarrow{\text{Rules}} x_0\, x_1 \ldots x_m$$

As discussed in the previous sections, the translation between Grade 2 Braille and text is a context-sensitive process. This means that translation for a particular string might depend on the previous translated string. In this case, the basic principle for

Braille translation has been decided: to translate a string w_1 into w_2, the translation process must be in order, and start with the leftmost element x_0 and end with the rightmost element x_m; the same for translating w_2 into w_1.

3.2 Finite State Machine (FSM)

A finite state machine [22-24] is a model of behaviour composed of states, transitions and actions. FSMs are a formalism growing from the theory of finite automata in computer science. An FSM has a set of states and a set of transitions between states; the transitions are triggered by input vectors and produce output vectors. The states can be seen as recording the past input sequence, so that when the next input is seen, a transition can be made based on the previous information.

A deterministic finite accepter (DFA) is used to explain how a particular string can be recognised [23-27]. To recognise strings is the first step in text-Braille translation. A DFA gives a binary output, saying either “yes” or “no” to answer whether the input is accepted by the machine or not. All states of the FSM are said to be either accepting or not accepting. If when an input is processed the current state is an accepting state, the input is accepted; otherwise it is rejected. Accepters can be used to recognise language by generating one-bit binary output, but they are not able to output strings of ASCII codes [26].

A DFA is defined by the quintuple

$$M = (Q, \sum, \delta, q_0, F)$$

where

Q is a finite set of internal states;

$\sum$ is a finite set of symbols called the input alphabet, which is a set of printable ASCII codes;

δ: Q × $\sum$ → Q is a function called the transition function; For example, a transition function, δ (q_0, b)= q_1, where b $\in\sum$ and q_0,$q_1$$\in$Q, means when a character 'b' is received by this DFA, the state for the DFA will transit from q_0 to q_1.

$q_0 \in Q$ is the initial state, and;

$F \subseteq Q$ is a set of final states.

A DFA operates in the following manner. At initiation, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the state machine, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the machine is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ [27].

Transition graphs are used to visualise and represent finite automata. In transition graphs, vertices represent states and edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are current values of the input symbol. The initial state will be identified by an incoming, unlabeled arrow, not originating at any vertex. Finite states are drawn with a double circle.

An example is given below to show how a DFA is going to achieve the recognition of the string "but".

The transition graph in figure 4 represents the DFA

$M = (\{q_0, q_1, q_1, q_2, q_3, q_b, q_{bu}, q_{but}\}, \{\text{printable ASCII codes}\}, \delta, q_0, \{q_{but}\})$,

where δ is given by

δ (q_0, !b) = q_1, δ (q_0, b)= q_b, δ (q_1, *)= q_1,

δ (q_b, !u) = q_2, δ (q_b, u)= q_{bu}, δ (q_2, *)= q_2,

δ (q_{bu}, !t) = q_3, δ (q_{bu}, t)= q_{but}, δ (q_3, *)= q_3 (* means any input characters).

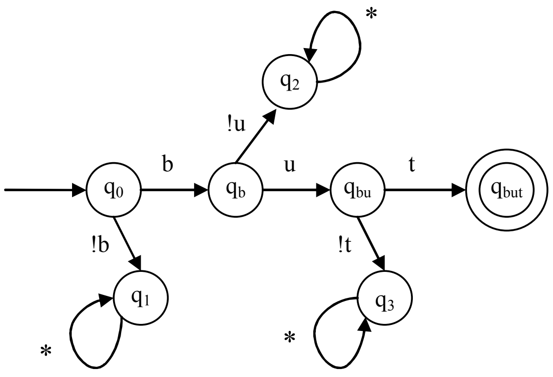


Figure 4. Transition Graph

This DFA is able to recognise the string “but”. Starting in state q_0, the first symbol which the DFA can recognise is the letter ‘b’. After receiving ‘b’, the DFA goes into state q_b. However, if the DFA receives other input symbols in state q_0, the next state is going to be q_1 which is called a trap state. The reason we call q_1 a trap state is that once the DFA goes into q_1, it will never escape, no matter what the next input is. Likewise, states q_2 and q_3 are trap states. Therefore, the only string which this DFA is able to recognise is “but”.

The example discussed above shows how to use a DFA to recognise one string. However, we can also use one DFA to recognise more than one string, and the set of those strings accepted by this DFA is called a language. Based on Grade 2 Braille rules, it is possible to build a DFA which is able to find particular strings to be contracted. For example, Figure 5 [28] shows a DFA to recognise strings, “herence”, “herer”, “hered”, and “here”. According to British Grade 2 Braille, the word “here” can be contracted to “h” in Braille, but when a letter ‘n’, ‘d’, or ‘r’ follows after it, the string “here” has to be separated into three parts: ‘h’, “er”, and ‘e’, where “er” is another contraction. Therefore, those cases need to be differentiated in a DFA.

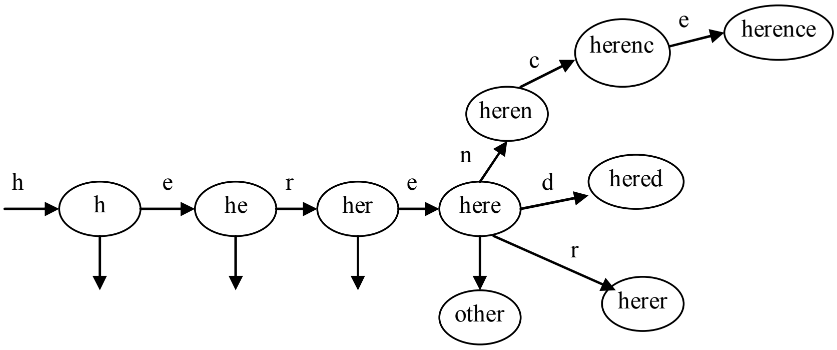


Figure 5. A DFA of recognising several strings

Here, we did not give a strict definition for the DFA given in Figure 5, because transition functions for this DFA are extensive and multiple. Starting with the recognition of the letter ‘h’, the DFA goes through four states to find the string “here”. But in every state, another transition function happens when receiving another letter. So a complete DFA beginning with ‘h’ can be used to describe all the rules originating from ‘h’. Evidently the DFA is very big and contains huge sums of transition functions. The advantage of using a DFA to recognise strings is that superfluous comparisons can be prevented. For example, the DFA goes through four states to find “here”, and if it receives a space or a punctuation character, the recognition of a word “here” is completed. The DFA will never go back to find “herer”, “hered”, and “herence” which are exceptions with higher priorities. Therefore, DFAs are a fast and efficient method of using Grade 2 Braille rules to recognise strings.

However, since DFAs are characterised by having no temporary storage, they are severely limited in their capacity to ‘remember’ things during computation [27]. The output generated by a DFA can only be one-bit binary code. Especially, a DFA has significant difficulties in recognising strings when right contexts are introduced. Therefore, a more powerful automaton might be needed to perform text-Braille translation.

As this discussion has indicated, using a finite state machine to perform Braille translation is not a wise option. This is because no matter how powerful the automaton is, the machine itself is a description of Braille translation rules. In this case, the state machine has a great number of states that make it very difficult to implement. Futhermore, when translation rules need to be revised, or new rules need to be added, the finite state machine is not able to remain the valid function. Therefore, a better method for translation will be discussed in the next section.

3.3 Markov System

3.3.1 Definition

A Markov algorithm is a string rewriting system that uses grammar-like rules to operate on strings of symbols [15] [26] [27]. This algorithm is named after the Russian mathematician, Andrey Markov. Although he is best known for his work on the theory of stochastic processes, which later were called Markov chains, Markov algorithms have proved to be one of his most important contributions. His algorithms have been shown to be Turing-complete, which means that they are suitable as a general model of computation, and can represent any mathematical expression from their simple notation [29].

A string rewriting system is a substitution system used to transform a given string according to specified rewriting rules. It includes an alphabet $\sum$ and a set of transformation rules. What distinguishes one rewriting system from another is the nature of $\sum$ and restrictions for the application of the production rules [30].

A Markov algorithm is a rewriting system whose production rules

$$x \rightarrow y$$

are considered ordered. In a derivation, the first applicable production rule must be used. The leftmost occurrence of the substring ‘x’ must be replaced by ‘y’. Some of the production rules may be singled out as terminal production rules; they will be shown as

$$x \rightarrow .y.$$

A derivation starts with some strings $w \in \sum$ and continues until either a terminal production rule is used or until there are no applicable rules.

The principles of using the Markov system are as follows [27]:

- Check the rules in order from top to bottom to see whether any of the strings to the left of the arrow can be found in the symbol string.
- If none are found, stop executing the algorithm.
- If one or more is found, replace the leftmost matching text in the symbol string with the text to the right of the arrow in the first corresponding rule.
- If the applied rule was a terminating one, stop executing the algorithm.
- Return to step 1 and carry on.

The following example shows the basic operation of a Markov algorithm. The Markov system of this example has a group of production rules listed as follows:

1. “A” → “paper”
2. “B” → “Braille translator”
3. “X” → “text”

4. “S” → “software”

5. “T” → “the”

6. “the software” → “hardware”

7. “a never used” → .“terminating rule”.

A symbol string, “This A describes a X to B implemented in T S.”, is going to be transformed by applying those production rules. Based on the algorithm, production rules need to be checked in order, and if a particular rule is applicable, the left hand side string must be replaced by the right hand side.

If the algorithm is applied to the above example, the Symbol string will change in the following manner:

1. “This paper describes a X to B implemented in T S.”

2. “This paper describes a X to Braille translator implemented in T S.”

3. “This paper describes a text to Braille translator implemented in T S.”

4. “This paper describes a text to Braille translator implemmented in the S.”

5. “This paper describes a text to Braille translator implemmented in the software.”

6. “This paper describes a text to Braille translator implemented in hardware.”

And then, the system will terminate.

In Markov algorithms, the rewriting process is proved to be irreducible, because the rewriting process must be ended when the terminating rule is applied [31]. However, if a new Markov rewriting system is built properly, then a string obtained by applying one Markov rewriting system can be transformed back to the original string. In the example given above, the original string can be generated by applying new production rules to the final string. The new rule set is shown as follows:

1. “paper” → “A”
2. “text” → “X”
3. “Braille translator” → “B”
4. “hardware” → “the software”
5. “the” → “T”
6. “software” → “S”
7. “a never used” → .“terminating rule”.

This characteristic of the Markov algorithm indicates its potential capability for performing both text-to-Braille and Braille-to-text translations. However, when the Markov system is applied to text-Braille translation, the key issue is to find an effective method of generating production rules. As has been discussed, it is difficult to describe every kind of translation case using production rules, because of the complexities of the English language, and the language and its vocabulary are continually evolving.

There are several software systems specially developed for text-Braille translation. The translating algorithms of these systems are based on the Markov system, although the name of this theory didn’t appear in the authors’ reports. Three types of text-to-Braille translation systems are given below to explain different methods of generating production rules; the systems and their characteristics will be discussed in the following sections.

3.3.2 DOTSYS II: Finite-State Syntax-Directed Braille Translation

The renowned Braille translation software, Duxbury, is widely used by virtually all of the world's leading Braille publishers. It is a truly multilanguage-Braille translation software [8]. The original translation algorithm of Duxbury was based on the DOTSYS II system developed by Jonathen Millen in 1970 [28].

As a product of the 1970s, the system was described using COBOL language. Few concepts about formal language were used in Millen’s report, therefore, even people who are not familiar with automata theory can still understand the algorithm. Even though the Markov system was not mentioned in related reports or papers, the DOTSYS II system performs text-to-Braille translation in the same manner as the Markov system works. The main issues in Millen’s report on DOTSYS II are about a method of translating rule production and how the rules are organised. In this section, how the system works will be described in detail, since as the first successful automatic Braille translator, DOTSYS II built the foundation of automatic Braille translation, and was the first system to introduce methods and concepts which were later borrowed by other translating systems.

3.3.2.1 Translation Procedures

First, the procedures of how the system works are given. A mnemonic summary of the chronological steps in the algorithm is given in Table 1 [28]. The definitions in Table 1 will be discussed in the following paragraphs.

Table 1. Chronological form of the translation algorithm as loop of five steps

	Procedures	Tables needed in each step
1	Index	Alphabet table
2	Search	Contraction table, decision table, right-context table
3	Output	Contraction table or alphabet table, sign table
4	Shift	Contraction table
5	Change	Transition table

DOTSYS II followed the same principle of Braille translation mentioned previously, which is to translate input text from left to right. For a specific application, the system uses a 10-character buffer as a sliding window to store the string to be translated.

If ten or fewer characters beginning at the left end of the buffer are to be translated as a group in Braille, then the system puts out the Braille sign or signs and shifts the contents of the buffer left to move the contracted characters out of the buffer. The same number of new characters are read in the vacancies at the right end of the buffer.

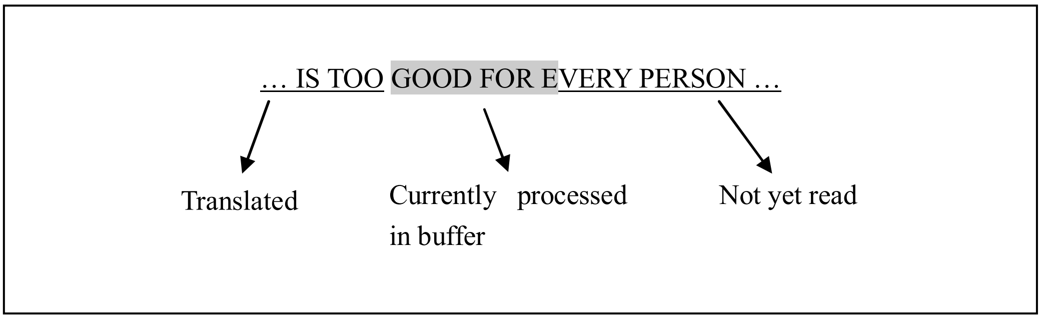
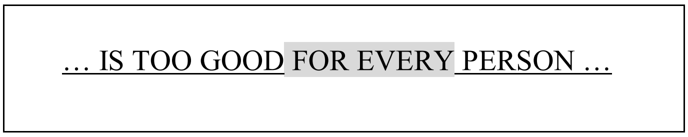

… IS TOO GOOD FOR EVERY PERSON …
Translated
Currently processed in buffer
Not yet read

Figure 6. The ten-character buffer

Figure 6 [28]shows an example of a ten-character buffer. The word GOOD occurs at the left end of the buffer. Since GOOD can be contracted to the string GD, the system puts out the two Braille signs standing for the string GD, which are its Braille translation and then shifts the buffer contents left by four characters. As shown in

Figure 7 [28], a space becomes the left most character in the buffer after the word GOOD.



… IS TOO GOOD FOR EVERY PERSON …

Figure 7. Contents in the buffer after translating GOOD

3.3.2.2 Method of Generating Production Rules

To achieve text-to-Braille translation, an important issue is how to determine whether or not there is a character string beginning at the left end of the buffer which can be contracted. DOTSYS II uses a table containing all the character strings for which there are standard Braille contractions, i.e. the contraction table.

The contraction table not only contains the 189 English Braille contractions, but also additional entries to implement some rules and indicate exceptions to others. Because it is based on Braille rules, some special cases have to be considered. For example, when the contracted letter groups EA and AR overlap, as in the word NEAR, the AR contraction is preferred. So a rule is created by putting the letter group EAR in the contraction table and specifying its translation to be the two Braille signs for E and AR.

The alphabet table is conceptually part of the contraction table. It contains the Braille signs used for individual characters when they are not translated as part of a contraction. It also contains indexing entries for the efficient search of the contraction table. The sign table, which is used for output, sets up a correspondence between the numbers from 0 to 63 and the 64 Braille signs. This allows the numerical equivalent of the signs to be used in the program for indexing and for storage in tables.

The format for contraction table entries is shown in Figure 8 [28]. Each contraction table entry has five fields, containing: all but the first character of a character string to be represented, a right-context character, the input class number, the number of characters to shift out of the buffer, and the numerical codes of the Braille signs to be put out.

String	Right context	Input class	Shift	Signs

Figure 8. Format for contraction table entries

The first field of the contraction table entry, the string field, comprises nine characters, and represents a character string recognised as a whole when encountered in the buffer. The first character in the string is encoded by the position of the entry in the table, and is not included in the string field. The string field consists of the second through to the last characters in the character string; if there are fewer than nine, they are followed by a dollar sign, and enough trailing blanks to make a total of nine characters. Figure 9 [28] is an example of the table entry for contraction EA.

String	Right	Input class	Shift	Signs
A$	L	4	2	2 99 99 99

Figure 9. Contraction table entry for EA

The second field of the contraction table entry, the right-context field, contains one character: either a blank or a character, such as ‘L’ or ‘P’. In Figure 9, the ‘L’ indicates that a letter must be found in the buffer immediately following the character group to which the entry would apply, in order for this contraction table entry to be applicable. A ‘P’ indicates that any character other than a letter must be found there, instead. A blank means the absence of a right-context condition. Right context, is an important contribution of Millen’s DOTSYS II, and is also used in both Slaby’s and

Paul Blenkhorn's systems which will be discussed in following sections. But in comparison with Millen's DOTSYS II, Blenkhorn's right context contains more information.

The third field of the contraction table entry is input class. Before discussing this field, it is necessary to mention the introduction of finite state memory in DOTSYS II.

Sometimes left contexts also need to be considered in contraction rules. To solve this problem, DOTSYS II uses a finite memory to implement rules involving characters to the left of the characters currently under inspection. Unlike the right context mechanism, which tests only an immediately adjacent character, the finite state memory allows a contraction to be affected by characters which may have occurred previously. The finite state memory appears in DOTSYS II in the form of a state vector. The state vector is a number of consecutive storage locations called state variables. There are five variables in DOTSYS II which are shown in Table 2 [28]. Each variable contains the character Y for yes and N for no. The occurrence of some characters in the input text can change specific state variables from N to Y, or vice versa, as they are shifted out of the buffer.

Table 2. State variables

1	After the start of a number
2	After the start of a word
3	Grade 1 translation
4	In a quotation
5	In italicised text

Another concept, input class, is also introduced. Because many different characters or contractions may have the same effect on the state vector, they are grouped into numbered input classes. This way, the new value of each state variable depends on its old value plus the input class of the alphabet or contraction table entry. The definitions for input classes are shown in Table 3 [28].

Table 3. Input class

1	Contractions always used in grade 2
2	Digits
3	Most punctuations
4	Contractions used after the start of a word
5	$G (grade switch)
6	Contractions used only at the start of a word
7	Isolated full-word contractions
8	$P” (start paragraph in quotation)
9	$P (start paragraph in italics)
10	“ (left quote)
11	” (right quote)
12	__ (begin italics)
13	_ (last word of italics)
14	(space)
15	A to J occurring in a number

Based on the state variables and input classes, the transition table and decision table are built into DOTSYS II. The transition table specifies the re-computation of the state vector. When a particular entry is applied, the value of a specific state variable will be evaluated depending on the input class.

Figure 10 [28] shows how the transition table is organised. NIC and NSV are, respectively, the number of input classes and the number of state variables. Each row is associated with a state variable, and each column is associated with an input class. Each entry, which can be R, S, T, or -, specifies the effect of an input class on a state variable, as show in Table 4 [28].

	Input Class				
State Variable		1	2	…	(NIC)
	1	R	S	…	R
	2	-	T		-
	…				
	(NSV)	-	R	…	R

Figure 10. Transition table

Table 4. Effect of input classes on state variables, as specified by transition table entries								
Transition table entry	R		S		T		-	
Old value of state variable	Y	N	Y	N	Y	N	Y	N
New value of state variable	N	N	Y	Y	N	Y	Y	N

The decision table has the final say on whether a contraction table entry is to be used to translate the initial part of the contents of the buffer. Its decision depends on the input class and on the state variables. The format of the decision table is shown in Table 5 [28].

Table 5. Decision table format					
Column		1	2	…	(NDTC)
Input Class	1	Y	-	…	-
	2	G	G	…	G
	…				
State Variables	1	-	-	…	-
	2	-	N	…	Y
	…				
	(NSV)	N	N	…	N

Each column gives the decision whether to use a contraction having a particular input class, conditional upon the values of some state variables. Given an input class, the leftmost column is found such that the corresponding decision entry is not a dash but Y, N, or G. Letter G, standing for go, means that a contraction with that input class is made, regardless of the values of the state variables. An entry Y means yes, and N means no, but these apply only if the conditions on the state variables, stated in the lower part of the column, have been satisfied. A condition Y means that the corresponding state variable must have a value of Y; a condition N means that the corresponding state variable must have a value of N; a condition '-' means that the value may be either Y or N.

If the conditions on the state variables are not satisfied, or if the decision entry is '-', this column does not give the decision, and the next column to the right is tested. If no column gives a decision, then the decision is taken to be no; the contraction is not made, and the contraction table search must continue.

According to the entry format discussed above, we can conclude that a character string beginning at the left end of the buffer is contracted if three conditions are satisfied:

- It matches an entry in the contraction table;
- It has an acceptable right-context character;
- A valid value can be found in the decision table.

The fourth field of the contraction table entry, the shift field, is a positive integer giving the number of characters to be shifted out of the buffer after this entry is applied. This field might appear redundant, because the length of the character string just translated is implied in the string field, but it is sometimes convenient to translate and shift out of the buffer only a part of the character string to which the entry applies.

The last field is the signs field which contains four numbers representing the Braille translation of that part of the character strings which will be shifted out of the buffer. The correspondence between the numbers and Braille signs is given in the sign table.

If the translation is fewer than four Braille signs, the extra numbers are specified as 99.

3.3.2.3 Further Development

After the introduction of DOTSYS II, the system was further revised and improved. Based on DOTSYS II, an American, Joseph Sullivan, developed DOTSYS III, which was a portable Braille translator, by refining the table sets and modifying the translating program [32] [33]. Although the system was able to achieve the text-to-Braille translation, as Sullivan claimed, DOTSYS III represented a compromise between the perfect and the possible, and mistakes in its translations could only be identified and corrected by human intervention [33].

Fortunately, what Jonathan Millen and Joseph Sullivan contributed became the foundation of the multi-language Braille translation software, Duxbury [8]. This software is able to perform bidirectional translation (text-to-Braille and Braille-to-text) covering dozens of major languages including English, French, Spanish, German, Danish, Italian, and Polish. It has the ability to include tactile graphics files for mixed text-and-graphic documents, and can handle diverse file formats. However, this software is powerful, but relatively expensive (595 US dollars with single license) [8]. Also, since the translation algorithm and data details are treated as confidential, commercial documents, it is difficult to determine how the system exactly works.

Furthermore, DOTSYS has its own drawbacks. Because the system is driven by a set of tables, it becomes very difficult for non-experts to update or modify them when new rules need to be evaluated [37]. Therefore, there is a need for new systems to be developed which solve these problems.

3.3.3 Slaby's System

3.3.3.1 A Universal Braille Translator Based on the Markov System

In 1975, a German, Wolfgang Slaby, made another attempt to apply the Markov system to text-to-Braille translation [34]. He realised that diverse languages lead to very different problems in automating the process of translation, because each of the translation algorithms possesses a lot of language-dependent components. Therefore, he tried to formalise an algorithm which is applicable to multi-language Braille translation, in other words, a universal system. In comparison with the finite-state syntax-directed Braille translation, this system provides a more formalised Braille translation process.

In this system, a quadruple is defined, to explain the Markov system of production rules as follows:

A quadruple m = ($\sum$, Δ, Γ, R) is called a Markov system of production rules if, and only if, the following is valid:

1. $\sum$, Δ, and Γ are alphabets with $\sum \subseteq \Gamma$ and $\Delta \subseteq \Gamma$.
   - $\sum$ is the input alphabet including Latin capital letters, Arabic numerals, punctuation marks and some other special characters.
   - Δ is an alphabet which concludes 64 Braille cells.
   - Γ is the working alphabet, which is the union of $\sum$ and Δ.
2. (Γ, R) is a system of production rules and R is an ordered set.

The empty word, i.e., the word over $\sum$ consisting of zero symbols, is denoted by ζ.

Let $\sum^*$ be a set of all words over the alphabet $\sum$, including ζ.

Let w be any word over $\sum^*$. If m is applicable to w, the following conditions have to be met:

There exists $u \rightarrow v \in R$, such that $u \rightarrow v$ is applicable to w.

Let $\sum$ be applicable to w and let $u_0 \rightarrow v_0$ be the first applicable production rule according to the order defined for R. Then since u_0 is a sub-word of w there exists x, $y \in \sum^*$ such that $w = x\ u_0\ y$.

Then we define

$m\ (w) = x\ v_0\ y$,

$m^n\ (w) = m\ (m^{n-1}\ (w))$ for $n \in N$, provided that m is applicable to $m^{n-1}\ (w)$.

Thus, for each w symbol $\sum^*$, there is one of the following two cases which is satisfied:

Case 1: there exists an integer $r_w \in N$ such that m is applicable to $m^{r}{}_{w}{}^{-1}\ (w)$ and m is not applicable to $m^{r}{}_{w}\ (w)$. ($m^{r}{}_{w}{}^{-1}\ (w)$ and $m^{r}{}_{w}\ (w) \in \Delta^*$)

Case 2: for each $n \in N$, m is applicable to $m^n\ (w)$. ($m^n\ (w)$ symbol Δ^*)

It is easy to see that $m^{r}{}_{w}{}^{-1}\ (w)$ or $m^n\ (w)$ is the result of the application of m to w.

If the system focuses on translating any specific language into the corresponding Grade 2 Braille, the appropriate Markov system of production rules $m_{spec.lang}$ has the following form:

$m_{spec.lang} = (\sum_{1,} \sum_2, \Gamma, R_{spec.lang})$.

But when a complete Markov system of production rules is going to be built, it becomes a very difficult linguistic problem, because of language specific ambiguities. These ambiguities encountered by Slaby are the main difficulties of translating Grade 2 Braille, as discussed in Chapter 2. Slaby's system is able to overcome this problem, but the solution of these ambiguities results in a rapid increase in the number of production rules. For example, based on this model, a set of more than 6000 replacement rules for German-contracted Braille was developed [35].

A severe disadvantage created by the large size of the rule set is that translating a word by iteratively scanning a large set of replacement rules sequentially will significantly slow down the translation process. Therefore in 1974 a new model - a segment translation system - was developed by Slaby.

3.3.3.2 Segment Translation System

A segment translation system [35] is an ordered list of translation rules of type u → v[x, y] where:

> If the word w is to be translated, and if u is the segment of w that has to be translated next, then u is translated by the segment v, provided that u has x as a left neighbour and u as a right neighbour in w ([x, y] is called a context condition).

This system also uses some basic terms and definitions of formal language theory like alphabet $\sum$, finite string over $\sum$, nullstring ε, concatenation vw of two strings v and w, length l (v) of a string v, segment or substring u of a string w, prefix h (w) and suffix t (w) of length n of a string w, formal language over $\sum$, and concatenation of formal languages.

The definition of a segment translation system is described as follows:

A quadruple $S = (\sum, \Delta, \Diamond, R)$ is called a segment translation system, if

- $\sum$ and Δ are alphabets.
- $\Diamond \in \sum \cup \Delta$ ($\Diamond$ is the frontier symbol).
- R symbol $\{u \rightarrow v[x, y] \mid u \in \sum^*, u \neq \varepsilon; v \in \Delta^*; x \in \{\Diamond, \varepsilon\}\cdot\sum^*; y \in \sum^*\cdot\{\Diamond, \varepsilon\}\}$, R is a finite, non-empty set. Each $u \rightarrow v[x, y] \in R$ is called a segment translation rule.
- For any $u \rightarrow v[x_1, y_1], u \rightarrow v[x_2, y_2] \in R$: ($x_1 = x_2$ and $y_1 = y_2 \Rightarrow v_1 = v_2$).
- Let ls (R) = $\{u \mid u \in \sum^*$ and there exists $u \rightarrow v[x, y] \in R\}$. Then for each u $\in$ ls (R) there exists $v \in \Delta^*$, such that $u \rightarrow v[\varepsilon, \varepsilon] \in R$ (the base rule).

In order to produce a translation z of a word w a concrete segment translation system is applied according to the following principles:

1. Look for a suitable prefix u of the remainder of w not yet translated, for which there exists a segment translation rule u → v[x, y].
2. If there are several such prefixes u1 and u2 in ls (R) with l (u1) > l (u2), then u1 takes priority over u2.
3. If u ∈ ls (R) is the longest prefix of the remainder of w not yet translated and if there exist segment translation rules u → v1[x1, y1] and u → v2[x2, y2] the context conditions of which are matched in w, then the rule with the longer context condition will be applied.

Based on these principles, the translation algorithm can be specified as shown in Figure 11:

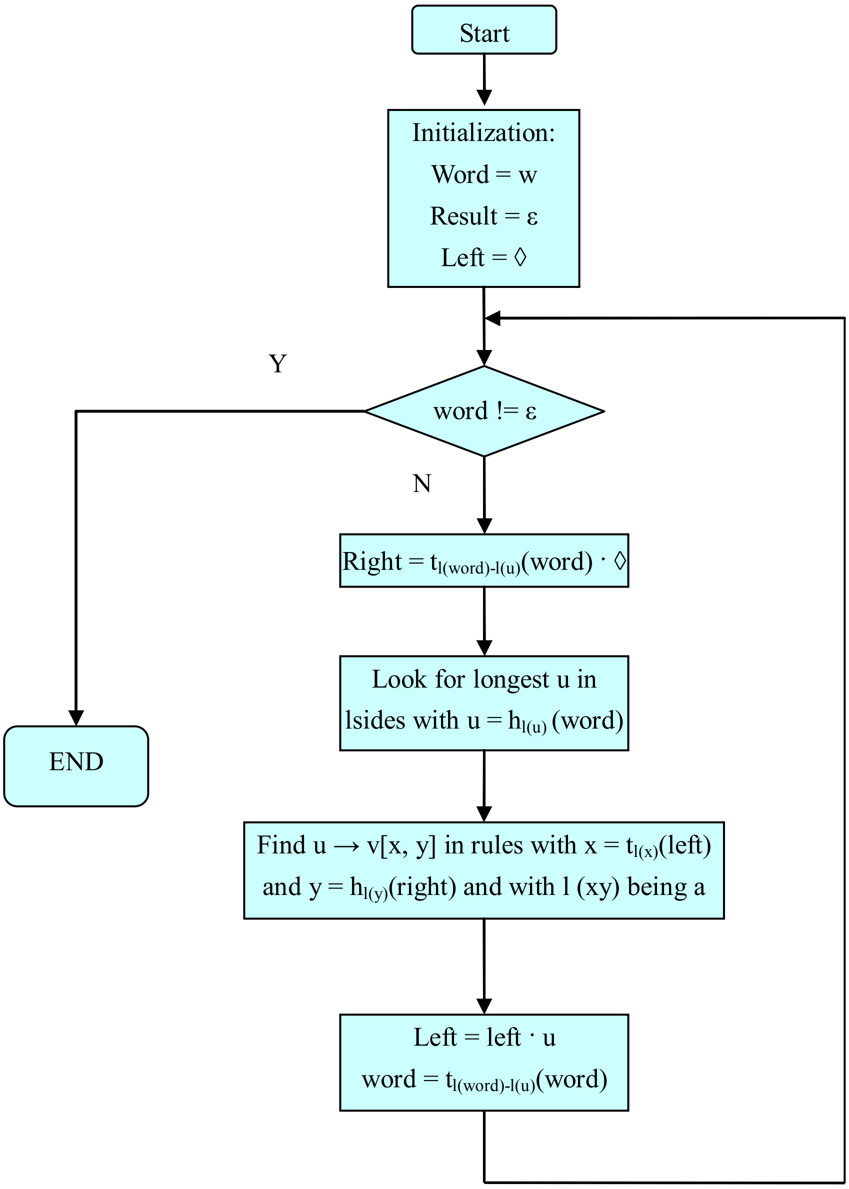

Start
Initialization:
Word = w
Result = ε
Left = ◊
word != ε
Y
N
Right = $t_{l(word)-l(u)}$(word) · ◊
Look for longest u in lsides with u = $h_{l(u)}$ (word)
END
Find u → v[x, y] in rules with x = $t_{l(x)}$(left) and y = $h_{l(y)}$(right) and with l (xy) being a
Left = left · u
word = $t_{l(word)-l(u)}$(word)

Figure 11. Translation algorithm of segment translation system

The next issue was to build a complete rule table for the segment translation system. Slaby presented a general idea of generating production rules for a text-to-German contracted-Braille translator. The method is to examine the preceding and following letters of a particular string to be contracted so that a rule can be decided which describes the environment where a certain sequence of the string must be translated different from the normal translation rule. This method is very similar to the one proposed by Hermann Kamp [36]. Using this method, a system with 4510 different rules has been produced. With this system a correctness rate of one translation error per 20000 characters was achieved.

Slaby's method is also suitable for translating other Latin languages to Braille. However, since constructing a complete rule table is related to complicated linguistic problems, it is a highly complex and time-consuming task.

The procedures of generating production rules for Slaby's system are described as follows:

1. Selecting suitable segment u to become a left side of a segment translation rule u → v[x, y]; a good beginning for this step is to take all those segments u for which a contraction is defined in the Braille system.
2. Laying down the base rule u → v [ε, ε] for each such segment.
3. Detecting words containing the segment u for which this segment is not translated correctly by the rules generated so far; using these words for generating additional rules.

It is evident that the format of production rules for the segment translation system is much simpler and more straightforward than the Finite-State Syntax-Directed Braille Translation system. The production rules in a segment translation system consist of only four parts, which are respectively left context, a string to be translated, right context, and Braille codes. Therefore, the system performs translation in reference to only one table, whereas DOTSYS II employs several tables to do the same job. The advantage of using this simple format is that the table can very easily be updated when mistakes are found or new rules are needed, and even a person who is not familiar with Braille is able to modify the rules.

3.3.4 Paul Blenkhorn's System

Based on Slaby's system, a British man, Paul Blenkhorn developed his own text-to-Braille translation system [35]. This system has the same characteristics as Slaby's, in that the system can be readily updated and modified by people who are not experts in computer algorithms.

This system also borrowed some concepts from DOTSYS, such as input class, transition state and the decision table, so that it is able to operate with a finite number of states which can hold the current context, as well as having capabilities for both left and right-context matching. The system has been designed so that a wide range of options and data can be introduced using a set of tables, including Braille rules, which are presented in a clear manner.

In the application of Slaby's system to the conversion of Braille into print, the approach taken was predominantly to use the state machine and right-context matching capabilities of this system to achieve the translation. However, as noted above, the updates of tables for a state machine requires a good deal of care and a detailed understanding of the system's operation. The use of context-matching rules is much more straightforward and easier to understand. Consequently, the print-to-Braille application has been constructed so that the bulk of the translation is achieved by using context-specific rules.

The state engine is only used for switching between grades of Braille and for handling letter signs. The decision table used by the state engine can be found in Table 6.

The decision table has 5 states and 6 input classes.

The States are:

1. Grade 2 Braille.
2. Grade 1 Braille.
3. After Letter Sign (Grade 2).
4. After Letter Sign (Grade 1).
5. Computer Braille.

The Input Classes are:

1. Any Braille except computer Braille.
2. Grade 2 rule.
3. Valid after Letter sign (Grade 2). (Used to switch back to Grade 2 after end of word.)
4. Valid after Letter sign (Grade 1). (Used to switch back to Grade 1 after end of word.)
5. Computer Braille.
6. Always allowed.

Table 6. State, Input Class, and Decision Table I

state	6 input classes					
1	1	2	0	0	0	0
2	1	0	0	0	0	0
3	1	0	3	0	0	0
4	1	0	0	4	0	0
5	0	0	0	0	5	0

In Blenkhorn’s system, there are a total of 1228 production rules. The complete rule table can be seen in Appendix II.

The format of each row in the table is:

Input class <TAB> Rule <TAB> New state

If the new state is ‘-‘, then no change occurs in the current state. The input class is set for each rule and is used in conjunction with the decision table to set the level of

Braille (i.e., Grade 2, Grade 1, or Computer Braille), and for letter sign placement in words that mix letters and numbers.

The rule is in the following format:

Left context [focus] Right context = input text

Several wildcards can be used in the left context and the right context. These are as follows:

“!” a letter;

“#” a number;

“!” a space or punctuation (include apostrophe);

“space” only a space character;

“|” zero or more capital signs;

“‘” one or more characters that are potentially roman numerals;

“;” zero or more letters;

“+” one or more digits.

An example is given here to explain how Blenkhorn’s system works. A rule table shown in Table 7 is used for translation.

Table 7. Fragment of Rule Table

	Input Class	Left Context	Focus	Right Context	Output Text	New State
1	2	~	G	;#	;G	3
2	2	#	G		;G	3
3	1	~	G	;#	;G	4
4	1	#	G		;G	4
5	2	!	GHAI		GHAI	-
6	2	!	GHEAD		GH1D	-
7	2	!	GHEAP		GH1P	-
8	2	!	GHIL		GHIL	-
9	2	!	GHOL	E	GHOL	-
10	2	!	GHOR	N	GHOR	-
11	2	!	GHOUS	E	GH\S	-
12	2		GHUN	T	GHUN	-
13	2		GH		<	-
14	2		GOOD		GD	-
15	2		GOVERN	ESS	GOV]N	-
16	2	~	GO	~	G	-
17	2	!	GG	!	7	-
18	2	.	GREAT		GRT	-
19	1		G	.!	G	-
20	1		G	~	G	-
21	1		G	~	G	-
22	1		G		G	-

Assume that we want to translate the word "GO". If the word is between two spaces, then we can use the spaces as the left and right contexts. For the first step, the system will find the table entry according to the first letter of this word. Obviously, the entry is letter 'G'. Then, the system will go through the rules of letter 'G', and check the rules including focus, input class, present state, left and right context, one by one, until finding the rule "2~[GO]~=G-". Because all the information of this rule matches the input, the translated result is "G". The hyphen mark for the new state means that the new state remains unchanged. The original algorithm of Blenkhorn's system is displayed in Appendix III.

Blenkhorn's algorithm has been implemented in a procedural program using C, proving that the algorithm works well. The reason for using C programming language is that C is a general-purpose structured programming language and it is portable, which means C was designed to give access to any level of the computer down to raw machine language [38]. However, modifications are necessary for its implementation in hardware.

In the system presented in this thesis, input class and states are not used, because when the system performs the Grade 1 and Grade 2 Braille translation, all the rules, except those for letter signs where the index input class is 1 or 2, have present states of either 1 or 2. Therefore, those rules always have a value of 1 according to the decision table.

On the other hand, rules which have presented states 3 and 4 in the rule table are always valid and once the next space character is found, the system will change the state to 1 or 2. In summary, if Computer Braille is not going to be considered, the decision table is not necessary.

3.3.5 Braille-to-Text Translation

The early contributions of computerised Braille translation were basically concerned with text-to-Braille translation [39]. However, it has been found that there are many similarities between text-to-Braille and Braille-to-text translation processes. The translation of Braille to text is very helpful for sighted persons who are not Braille literate to understand documents printed in Braille [41]. The translation from Braille-to-text is a reverse process corresponding to text-to-Braille translation, and it is also considered to be a context-sensitive conversion. In fact, the algorithm used by text-to-Braille translation is perfectly suitable for the Braille-to-text translation. Therefore, Paul Blenkhorn developed both systems using the same method [40] [42]. In contrast to the text-to-Braille converting system, when Blenkhorn constructed the rule table of the Braille-to-text translation, the left context was excluded from each rule. Instead, he used a method similar to DOTSYS II, which was to build a decision table containing left-context information. The decision table contains input classes and finite states, shown in Table 8.

This decision includes 6 states and 7 input classes.

The states are:

1. At the start of the word.
2. In punctuation at the start of the word.
3. After the start of the word.
4. Within a number.
5. Within members of the group “&!(A)”.
6. Within the scope of a letter sign.

The input classes are:

1. Don’t care
2. Valid at the start of the word.
3. Valid in punctuation, or at the start of the word.
4. Only valid after the start of the word.
5. Valid for members of the group “&!(A)”.
6. Valid within the scope of a letter sign.
7. Valid within a number.

Table 8. State, Input Class, and Decision Table II

state	7 input classes						
1	1	2	3	0	0	0	0
2	1	0	3	0	0	0	0
3	1	0	0	4	0	0	0
4	1	0	0	0	0	0	7
5	0	0	0	0	5	0	0
6	0	0	0	0	0	6	0

The format of the rules is:

Input class <TAB> rule <TAB> new state

The input class is set for each rule and is used in conjunction with the decision table to determine if a rule fires.

The rule is in the following format:

[focus] right context = output text

Several wildcards can be used in the right context. They are:

“!” --- one or more of the set “&!(A)”;

“ ” --- any white space character;

“~” --- one or more potential punctuation characters, and;

“_” --- actual space character.

In Blenkhorn’s Braille-to-text translation system, there are a total of 498 rules, and the rule table is shown in Appendix IV.

4. FPGA and VHDL Information

4.1 About FPGAs

Field Programmable Gate Arrays (FPGAs) are digital integrated circuits (ICs) that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks [43-50]. The “field programmable” portion of the FPGAs name refers to the fact that its programming takes place in the field. This means that FPGAs can be configured in the laboratory. With their introduction in 1985, FPGAs have been an alternative for implementing digital logic in systems [43]. The applications of FPGAs conver many fields of digital electronics, including custom IC designs, digital signal processing (DSP) and development of embedded systems, etc. Especially, FPGAs have created a new area in their own right: reconfigurable computing (RC), which refers to exploiting the inherent parallelism and reconfigurability provided by FPGAs to hardware accelerate software algorithms [43].

4.1.1 Evolution of Programmable Logic

FPGAs are one type of Programmable Logic Devices (PLDs). PLDs were invented in the late seventies [44]. Their growing popularity has seen them become one of the largest growing sectors in the semiconductor industry. PLDs provide designers ultimate flexibility and design integration, are easy to design with and can be reprogrammed time and time again even in the field to upgrade system functionality.

The first type of user-programmable chip [45] that could implement logic circuits was the Programmable Read-Only Memory (PROM), in which address lines can be used as logic circuit inputs and data lines as outputs. Logic functions, however, rarely require more than a few product terms, and a PROM contains a full decoder for its address inputs. PROMS are thus an inefficient architecture for realising logic circuits, and so are rarely used in practice for that purpose.

The first device developed later specifically for implementing logic circuits was the Field-Programmable Logic Array (FPLA), or simply PLA for short. A PLA consists of two levels of logic gates: a programmable "wired" AND-plane followed by a programmable "wired" OR-plane. A PLA is structured so that any of its inputs (or their complements) can be AND'ed together in the AND-plane; each AND-plane output can thus correspond to any product term of the inputs. Similarly, each OR-plane output can be configured to produce the logical sum of any of the AND-plane outputs. With this structure, PLAs are well-suited for implementing logic functions in sum-of-products form. They are also quite versatile, since both the AND terms and OR terms can have many inputs (this feature is often referred to as wide AND and OR gates).

When PLAs were introduced in the early 1970s, by Philips, their main drawbacks were that they were expensive to manufacture and offered somewhat poor speed performance. Both disadvantages were due to the two levels of configurable logic, because programmable logic planes were difficult to manufacture and introduced significant propagation delays. To overcome these weaknesses, Programmable Array Logic (PAL) devices were developed. PALs feature only a single level of programmability, consisting of a programmable "wired" AND plane that feeds fixed OR-gates. To compensate for lack of generality incurred because the OR-plane is fixed, several variants of PALs are produced, with different numbers of inputs and outputs, and various sizes of OR-gates. PALs usually contain flip-flops connected to the OR-gate outputs so that sequential circuits can be realised. PAL devices are important because when introduced they had a profound effect on digital hardware design, and also they are the basis for some of the newer, more sophisticated architectures that will be described shortly. Variants of the basic PAL architecture are featured in several other products known by different acronyms. All small PLDs, including PLAs, PALs, and PAL-like devices are grouped into a single category called Simple PLDs (SPLDs), who's most important characteristics are low cost and very high pin-to-pin speed-performance.

As technology has advanced, it has become possible to produce devices with higher capacity than SPLDs. The difficulty with increasing capacity of a strict SPLD architecture is that the structure of the programmable logic-planes grows too quickly in size as the number of inputs is increased. The only feasible way to provide large capacity devices based on SPLD architectures is then to integrate multiple SPLDs onto a single chip and provide interconnect to programmably connect the SPLD blocks together. Many commercial field-programmable devices (FPD) products exist on the market today with this basic structure, and are collectively referred to as Complex PLDs (CPLDs).

The highest capacity general purpose logic chips available today are the traditional gate arrays sometimes referred to as Mask-Programmable Gate Arrays (MPGAs). MPGAs consist of an array of pre-fabricated transistors that can be customized into the user's logic circuit by connecting the transistors with custom wires. Customization is performed during chip fabrication by specifying the metal interconnect, and this means that in order for a user to employ an MPGA a large setup cost is involved and manufacturing time is long. Although MPGAs are clearly not FPDs, they are mentioned here because they motivated the design of the user-programmable equivalent: FPGAs.

In 1985, a company called Xilinx introduced a completely new idea. The concept was to combine the user control and time to market of PLDs with densities and cost benefits of gate arrays. Like MPGAs, FPGAs comprise an array of uncommitted circuit elements, called logic blocks, and interconnect resources, but FPGA configuration is performed through programming by the end user. An illustration of a typical FPGA architecture appears in Figure 12 [45]. As the only type of FPD that supports very high logic capacity, FPGAs have been responsible for a major shift in the way digital circuits are designed.

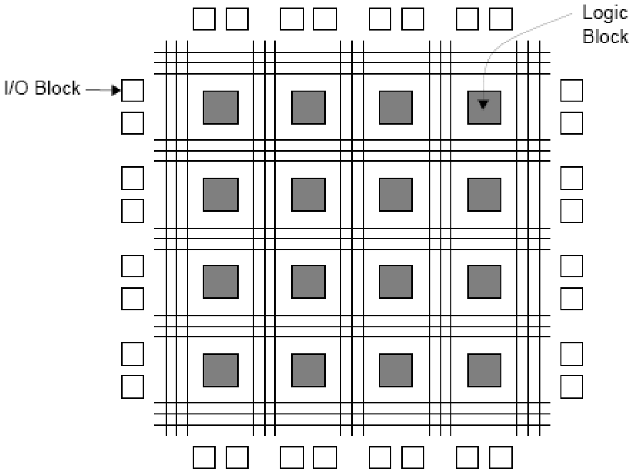


Figure 12.Structure of a FPGA

4.1.2 Architecture of Xilinx FPGAs

There are three main types of FPGAs according to programmable elements to be used. One is based on static RAM (SRAM), the second type based on antifuses, and the third is FLASH-based FPGAs. [48] [49].

The first, SRAM programming, involves static RAM bits as the programming elements. There bits can be combined in a single memory and used as Look-up Table (LUT) to implement any kind of combinational logic. An example of usage of SRAM-controlled switches is illustrated in Figure 13 [45], showing two applications of SRAM cells: for controlling the gate nodes of pass-transistor switches and to control the select lines of multiplexers that drive logic block inputs. The figures gives an example of the connection of one logic block (represented by the AND-gate in the upper left corner) to another through two pass-transistor switches, and then a multiplexer, all controlled by SRAM cells. Whether an FPGA uses pass-transistors or multiplexers or both depends on the particular product.

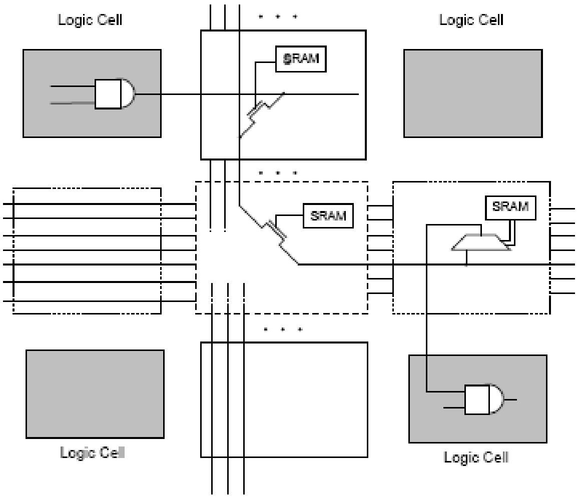


Figure 13.SRAM-controlled Programmable Switches

The other type of programmable switch used in FPGAs is the antifuse. Antifuses are originally open-circuits and take on low resistance only when programmed. Antifuses are suitable for FPGAs because they can be built using modified CMOS technology. Antifuse FPGAs have the advantage of lower power over SRAM-based FPGAs, but an antifuse FPGA can only be programmable once.

A new type of FPGAs using flash technology is called flash-based FPGAs. These devices are essentially the same as SRAM-based devices, except that they use flash EPROM bits for programming. Flash EPROM bits tend to be small and fast. They are non-volatile like antifuse, but reprogrammable like SRAM.

Each FPGA vendor has its own FPGA architecture, but in general terms they all have a basic structure shown in Figure 12. The architecture consists of configurable logic blocks (CLBs), configurable I/O blocks, and programmable interconnect to route signals between CLBs and I/O blocks. Also, there is clock circuitry for driving the clock signals to each flip-flop in each logic block. Additional logic resources such as arithmetic logic units (ALUs), memory, and decoders may also be available [50].

CLBs contain the programmable logic for the FPGA. The diagram in Figure 14 shows a typical CLB, containing RAM for creating arbitrary combinational logic functions [51]. It also contains flip-flops for clocked storage elements and multiplexers (MUXes) in order to route the logic within the block and to route the logic to and from external resources. These MUXes also allow polarity selection, reset input, and clear input selection.

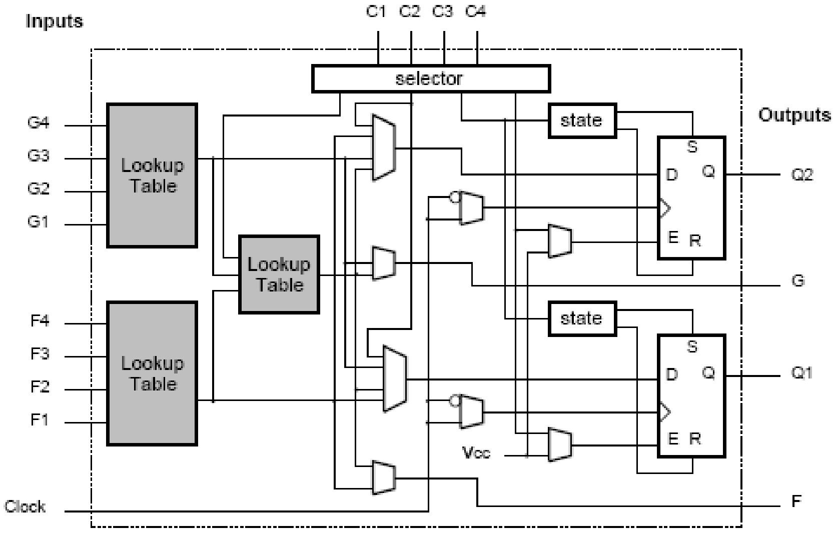


Figure 14.FPGA configurable logic block (CLB)

On the left of the CLB are two 4-input memories, also known as 4-input lookup tables or 4-LUTs. 4-input memories can produce any possible 4-input Boolean equation. Feeding the output of the two 4-LUTs into a 3-input LUT, produces a wide variety of outputs. Four signals labelled C1 through C4 enter at the top of the CLB. These are inputs from other CLBs or I/O blocks on the chip, allowing outputs form other CLBs to be input to this particular CLB. These interconnect inputs allow designers to partition large logic functions among several CLBs. They also are the basis for connecting CLBs in order to create a large, functioning design.

The MUXes throughout the CLB are programmed statically. In other words, when the FPGA is programmed, the select lines are set high or low and remain in that state. Some MUXes allow signal paths through the chip to be programmed. For example, MUX M1 is programmed so that the top right flip-flop data is either input C2, or the output of one of the two 4-LUTs or the output of the 3-LUT.

Some MUXes are programmed to affect the operation of the CLB flip-flops. MUX M2 is programmed to allow the top flip-flop to transition on the rising or falling edge of the clock signal. MUX M3 is programmed to always enable the top flip-flop, or to enable only when input signal C4 is asserted to enable it.

The I/O pins on a chip connect it to the outside world. The I/O pins on any chip perform some basic functions:

- Input pins provide electrostatic discharge protection.
- Output pins provide buffers with sufficient drive to produce adequate signals on the pins.
- Three-state pins include logic to switch between input and output modes.

The pins on one FPGA must be programmable to accommodate the requirements of the configured logic. A standard GPGA pin can be configured as an input, output, or three-state pin.

Pins may also provide other features. Registers are typically provided at the pads so that input or output values may be held. The slew rate of outputs may be programmable to reduce electromagnetic interference; lower slew rates on output signals generate less energetic high frequency harmonics that show up as electromagnetic interference (EMI).

An FPGA has several kinds of interconnect: short wires, general-purpose wires, global interconnect, and specialized clock distribution networks. The reason that FPGAs need different types of wires is that wires can introduce a lot of delay, and wiring networks of different length and connectivity need different circuit designs.

4.1.3 FPGAs vs. ASICs

In comparison with PLDs, there is another kind of un-programmable digital devices called application specific integrated circuits (ASICs) [46]. An ASIC is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. By carefully tuning each ASIC to a given job, the computer designer can produce a smaller, cheaper, faster chip that consumes less power than a programmable processor. A custom graphics chip for a PC, for instance, can draw lines or paint pictures on the screen 10 or 100 times as quickly as a general-purpose central processing unit can [47].

An ASIC is not a programmable device, but it is important precursor to the developments leading up to CPLDs and FPGAs. Nowadays, the ASIC vendor has created a library of cells and functions that the designer can use without needing to know precisely how these functions are implemented in silicon. The ASIC vendor also typically supports software tools that automate such processes as circuit synthesis and circuit layout.

The design of an ASIC goes down to the masks used to fabricate an IC. The ASIC must be fabricated on a manufacturing line, a process that takes several months, before it can be used or even tested. ASICs have some significant advantages because they are designed for a particular purpose: they are very fast and the circuitry is dense, allowing lots of functionalities on a die; and the cost is low for high volume production. The disadvantage is that it takes time for the ASIC vendor to manufacture and test the parts. Also, the customer incurs a large charge up front, called a non-recurring engineering (NRE) expense, which the ASIC vendor charges to begin the entire ASIC process. And if there is a mistake, it is a long, expensive process to fix it and manufacture new ASICs.

Hardware designers always wanted something that gave them the advantages of an ASIC, such as circuit density and speed, but with the shorter turnaround time of a programmable device. A new development in integrated circuits offers an ideal solution: large and fast FPGAs—highly tuned hardware circuits that can be modified at almost any point during use. FPGAs consist of arrays of configurable logic blocks that implement the logical functions of gates. Logic gates are like switches with multiple inputs and a single output. They are used in digital circuits to perform basic binary operations such as AND, NAND, OR, NOR and XOR. In most hardware that is used in computing today, the logical functions of gates are fixed and cannot be modified. In FPGAs, however, both the logic functions performed within the logic blocks and the connections between the blocks can be altered by sending signals to the chip. These blocks are structurally similar to the gate arrays used in some ASICs, but whereas standard gate arrays are configured during manufacture, the configurable logic blocks in FPGAs can be rewired and reprogrammed repeatedly, long after the integrated circuit has left the factory. Compared to ASICs, the design time of FPGAs is much shorter. FPGAs, because they are standard parts, have several advantages in design time. They can be used as prototypes, they can be programmed quickly, and they can be used as parts in the final design. Moreover, FPGAs are more affordable and less risky than ASICs, because using FPGAs, we do not need to worry about high NRE cost, long delay in design and testing. Meanwhile, FPGAs have the same advantages as ASICs. They have high complexity and reliability, the circuits are dense, and the physical size is small. FPGAs consist of huge sum of standard programmable logic blocks which can be applied to massively parallel operation.

4.1.4 Parallel Processes on FPGAs

The use of programmable logic to accelerate computation, also called Reconfigurable Computing (RC) arose in the late 1980's with the widespread commercial availability of FPGAs. RC researchers found that the speed of direct hardware execution on a FPGA is 10 times to 100 times faster than the equivalent software algorithm. FPGAs offer significant advantages over microprocessors for high performance, low volume applications, particularly for applications that can exploit customized bit widths and parallel processing [52].

The speed advantage of FPGAs derives from the fact that the programmable hardware is customized to a particular algorithm. An FPGA can be configured to perform arbitrary fixed precision arithmetic, with the number of arithmetic units, their type, and their interconnection uniquely defined by the algorithm. In contrast, the design of a fixed instruction set processor must accommodate all possible operations that an algorithm might require for all possible data types. A comparison of a multiply-accumulator between reconfigured architecture in a FPGA and fixed processors are shown in Figure 15 [52]. In this example, the FPGA is configured to hold an array of application-specific processing units. Each processing unit contains four 8-bit adders and 16-bit multiply-accumulate unit which are operating in parallel. Hardware address generators are used to access off-chip data. The microprocessor in this example has a Harvard architecture. It accesses the sequential instruction stream through the Instruction Cache. The data memory hierarchy includes external memory, cache and integer and floating point register files, which supply operands to the arithmetic units.

The example illustrates the major differences between reconfigurable and processor-based computing. The FPGA is configured into a customized hardware implementation of the application. The hardware is usually data path driven, with minimal control flow, and is able to process data in parallel; processor-based computing depends on a linear instruction stream including loops and branches.

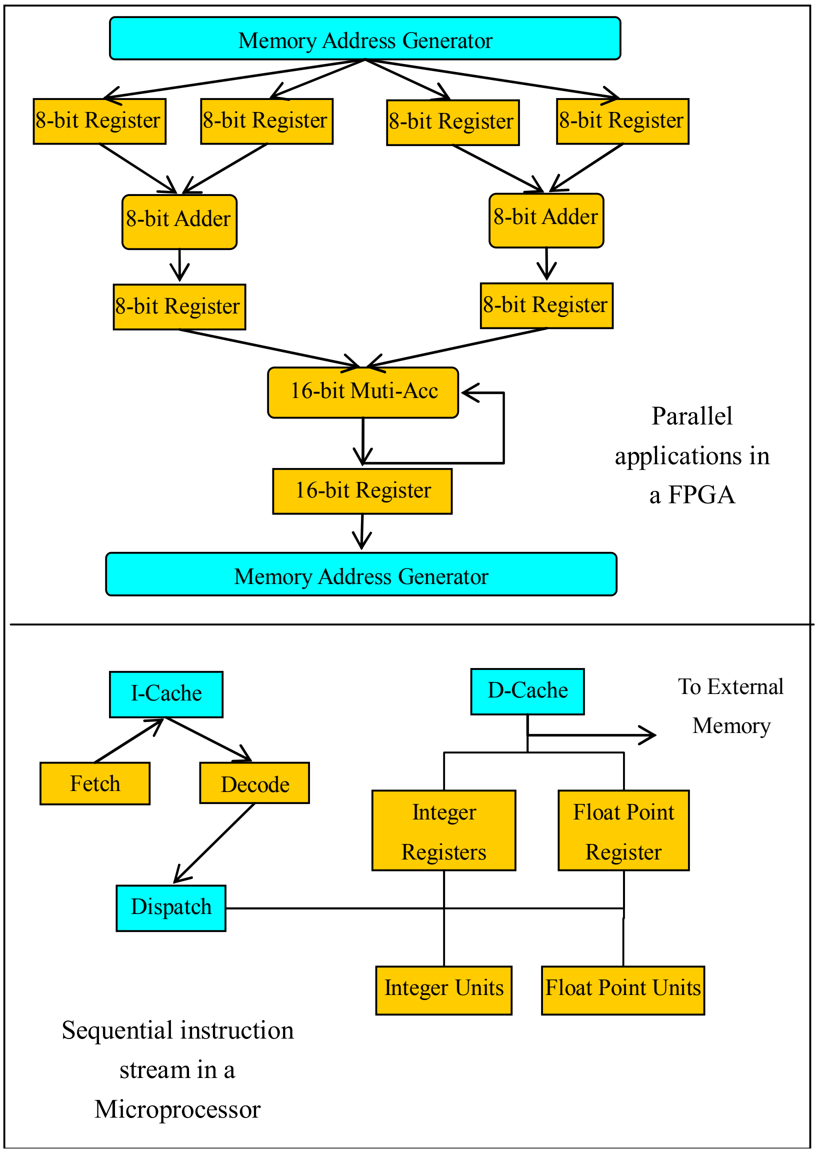


Figure 15. Reconfigurable architecture in a FPGA and sequential architecture in Microprocessor

4.2 VHDL

When Computer Aided Design (CAD) technology was developed, Hardware Description Language (HDL) became one of the main methods that designers used to implement hardware designs in CAD systems [53]. Although the method of describing hardware using Higher level programming language has been developed [54], HDL is still accepted as a standard and most popular tool for all kinds of digital designs. In this section, we will talk about the characteristics of VHDL, and the advantages of using VHDL in FPGA applications.

There are two prevailing versions of hardware description languages: one is verilog HDL, and the other one is VHDL. The coding style of verilog HDL is more similar to some high-level software programming languages, such as C [55]. Therefore, it has more flexibility and can be accepted more easily by software programmers. However, beginners tend to make mistakes because of its flexibility. In this thesis, VHDL with the stricter coding style is preferred by the author.

VHDL originated from the American Department of Defence, which recognised that they had a problem in their hardware procurement programmes. They suffered low compatibility which means it was very difficult to transfer design data to other companies for secondary sourcing, and there was no guarantee that these languages would survive for the life expectancy of the hardware they described.

The solution was to have a single, standard hardware description language, and to achieve this the earliest version of VHDL was developed. Later on, the importance of the language development, and especially the importance of standardisation of the language, was recognised and so the formative language was passed into the public domain by its inclusion in the Institute of Electrical and Electronics Engineers (IEEE) standard in 1986 [56] [57].

The combination of computing and electronics is the key to the success of VHDL [56]. Making use of advantages of software programming language, VHDL can be used to design hardware with complex functions and algorithms, and it helps designers significantly reduce design cycles in contrast with traditional design methods.

VHDL is intended to cover every level of the design cycle from system specification to netlist, making it a tool capable of supporting all levels of the hardware design cycle [58-60].

The system level focuses on the description of the functionalities of the system [58-60]. A constant concern at this level is to forget useless details, which would imply architectural choices too early in design methodology. A system description with too much detail is a drawback since it restricts further architectural choices or implies a given technology. Therefore, hiding the information structure is desirable and the notion of concurrency may not be necessary at this phase. But this level of description is not suitable to be synthesised.

The synthesisable level, also named Register Transfer Level (RTL), focuses on logic synthesis for the design [58-60]. Logic synthesis offers an automated route from an RTL design to a gate-level design. This level is the potential input for synthesis tools. RTL is a high-level design methodology which can be used for any digital system. In RTL design, a circuit is described as a set of registers and a set of transfer functions indicating the flow of data between registers. The registers are implemented directly as flip-flops, while the transfer functions are implemented as blocks of combinational logic. The language must allow a description of the model at this level with a sufficient level of abstraction towards the physical level.

The netlist level is the potential output of synthesis tools [58-60]. It is a structural view appearing as a collection of model instantiations. This kind of description involves the existence of model libraries. The notion of time is often present in the description of these models, from the notion of propagation delay through a gate to very sophisticated delays. At this step, the language has to offer an optimal flexibility in terms of timing configuration or technology.

Hierarchy is another outstanding character of VHDL [61]. There are a number of reasons for using hierarchy. First, using hierarchical methods, designers are able to divide a system into several parts. Third party components can be incorporated into a design more easily, and this leads to a higher degree of confidence in the integrity of the design. Each subcomponent can be designed and tested in isolation before being incorporated into the higher levels of the design. This testing of intermediate levels is much simpler than testing the whole system. Subcomponents can be used concurrently and also can be reused somewhere else.

5. Text-to-Braille Translation Based on Hardware

5.1 Universal Design Methodology for Programmable Devices (UDM-PD)

The universal design methodology is used in the design of hardware Braille translation systems. Using universal design methodology, designers aim to design devices that are free from manufacturing defects, work reliably over their lifetime and function correctly in the whole system [59] [62]. UDM-PD is also able to help designers design a device with good time efficiency and with fewer resources used.

UDM-PD outlines a specific design flow for creating a programmable device which, when followed, allows a good design to be reached. The design flow consists of the steps shown in Figure 16 [62]. Each particular design will require slight variations in the specifics of each step, but essentially the steps will be the same.

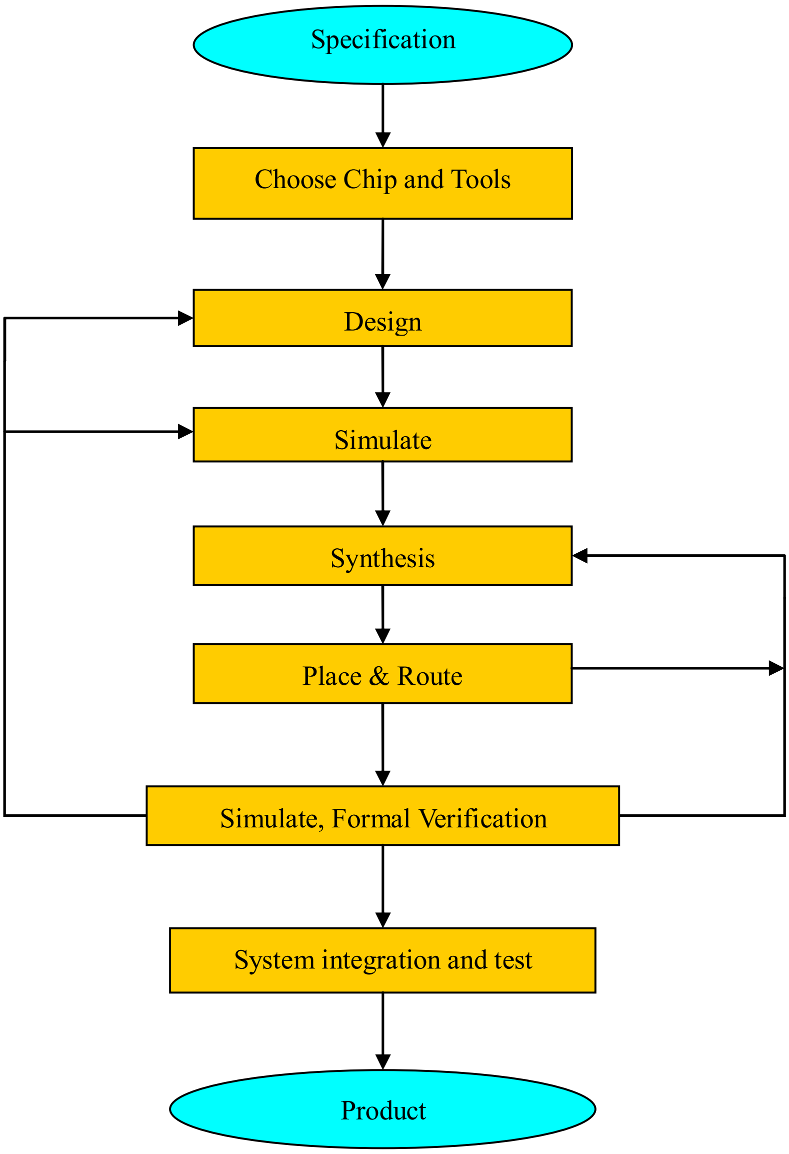


Figure 16. Design Flow of UDM

A Top-Down design methodology is used in system implementations where high-level functions are defined first, and the lower-level implementation details are filled in later [63].

5.2 Text-to-Braille Translation

5.2.1 System Specification and Tools Selection

A specification is an absolute necessity for a digital design. A specification allows each engineer to understand the entire design and how the part for which he/she is responsible connects to the whole design. It allows the engineer to design the correct interface to the rest of the chip. It also saves time and thus cost, and helps avoid misunderstanding between co-designers.

The system specification for the text-to-Braille translator includes the following information:

- External block diagram showing how the chip fits into the system
- Internal block diagram showing each major functional section
- Description of the I/O pins
- Gate count estimate
- Test procedures

A specification also should have descriptions about package type, power consumption target, price target, and timing estimate including setup and hold times for input pins, clock cycle time etc. However, in this system, the issues of price and packaging are not considered, since this system is not related to the development of a commercial product. Power consumption and timing constraints similarly are not discussed, as once development tools and chip are selected, it is very easy to obtain these data.

The external block diagram, shown in Figure 17, indicates how the device fits into the testing system [65]. As well as a text-to-Braille translator, a simple universal asynchronous receiver/transmitter (UART) is integrated in FPGA for communicating with computers [64]. The serial receiver sends two output signals to the translator. The one-bit data-ready signal is indicated using a thin arrow, and the other signal is 8-bits data indicated by a thick arrow. A handshake communication was built between the serial transmitter and the translator. A one-bit signal is sent to the translator from the transmitter to indicate if the transmitter is ready to receive data. The thick arrow from the translator represents 8-bits data, while the black arrow is for data ready. For further development, the text-to-Braille translator is going to be used as a standalone component in a universal Braille device which is able to perform multi-tasks including Braille note taking, translation, speaking and interface to printers and embossers. Therefore, parallel communication is preferred in this particular application.

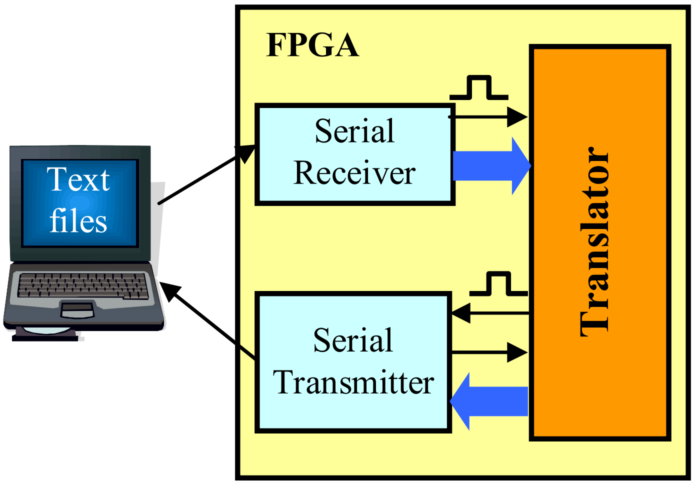


Figure 17. External Block Diagram

The internal block diagram is shown in Figure 18. The translating process is quite straightforward. Before translation starts, the rule table has to be stored in a look-up table first. The input block takes charge of this task as well as receiving characters to be translated. The translation block receives the string from the input block, fetches rules from the look-up table according to the entry character, and then translates it from leftmost character to the rightmost. This block implements text-to-Braille translation including left context, focus and right context checking functions. A feedback signal will be sent back to the input block indicating how many characters have been translated. The translation results are sent to the output block which has a handshake communication with the serial transmitter.

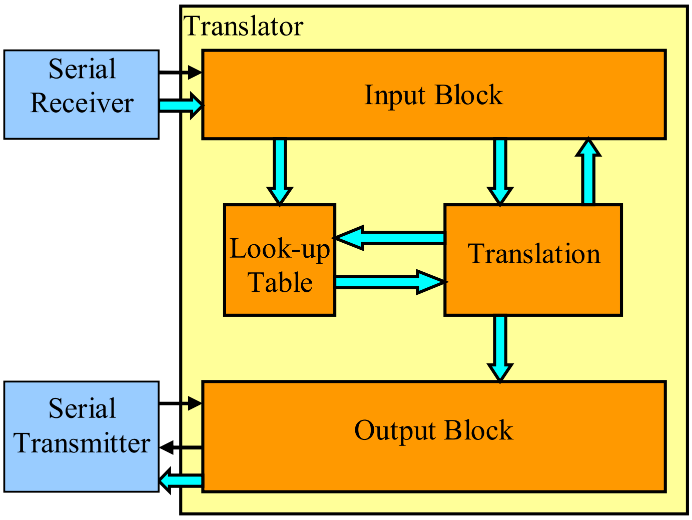


Figure 18. Internal Block Diagram of the translator

Xilinx FPGA products were chosen as a development platform for implementation. As the inventor of FPGAs, the Xilinx Company is able to supply consumers with several series of high quality FPGAs and powerful development tools.

Because a big rule table is needed in text-to-Braille translation, the FPGA has to include enough memory to store the table. The size of the original table is 20 kilo-bytes, but the format of the table has been changed to fit in the translator. As a result, the table has 33 kilo-bytes of data. Therefore, The Memec Virtex-4 FX12 LC Development Board was selected for the translation system, and correspondingly, a Xilinx Virtex family FPGA, XC4VFX12, is integrated on this board [66]. The Virtex XC4VFX12 FPGA includes 86 kilo-bits of distributed RAM and 648 kilo-bits of block RAMs, so it supplies abundant memory resources so that two rule tables for both text-to-Braille and Braille-to-text translation can be stored in the same chip.

There is a hard PowerPC processor core is integrated in this FPGA, or a alternative soft Intelectual Property of a microcontroller Microblaze can be configured in this

FPGA. Consequently, this FPGA can be used to build a universal Braille system on a chip (SOC). The Memec Virtex-4 FX12 LC Development board, shown in Figure 19, includes 64MB of DDR SDRAM, 4MB of Flash, USB-RS232 Bridge, a 10/100/1000 Ethernet PHY, 100 MHz clock source, RS-232 port, and additional user support circuitry to develop a complete system. So the board is able to satisfy the requirements of the translation system for testing purposes.

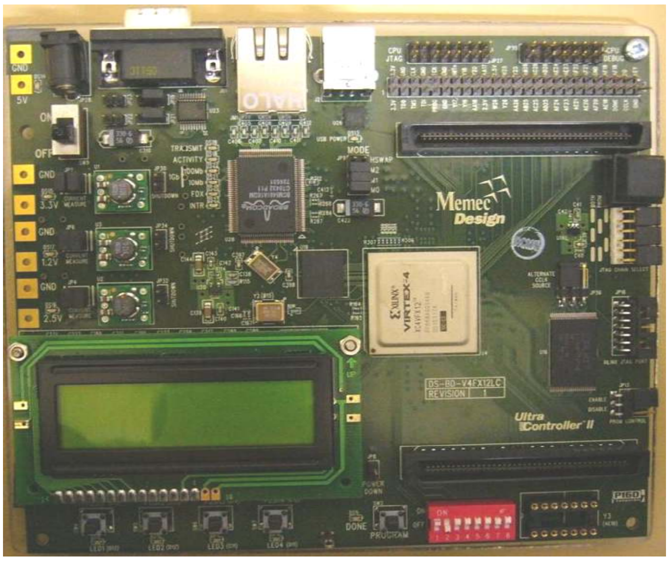


Figure 19. Memec Virtex-4 FX12 LC FPGA Development board

5.2.2 Design

Figure 20 shows the block diagram of the text-to-Braille translator consisting of ten blocks. Before translation starts, the rule table needs to be sent to the look-up table. To do this, a block, called a data-controller was built in this system. While the translator is initialised, particular signals generated from the data-controller enable the look-up-table block and disable the translating block, and translating rules will be sent one by one to the look-up table. Every address in the look-up table corresponds to a single translating rule. To fit the rule table to the look-up table, the format of the rules has been changed. The redundant information such as square brackets, tabs and equal marks has been removed. The length of each part of each rule is the same, and the short one has zeros following to indicate the end.

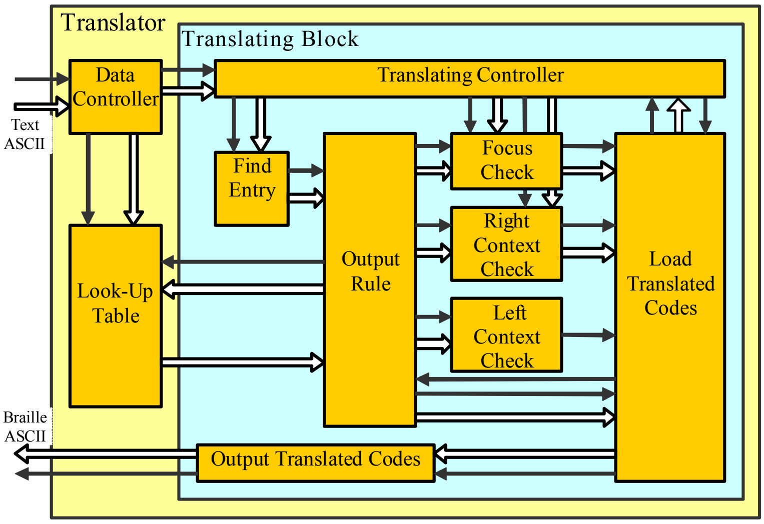


Figure 20. Block Diagram of Text to Braille Translator

To explain how the translation is processed, behavioural simulation results for translating a string “SHOULD ” are given in Figure 21. Behavioural simulation is the first check which is to verify RTL code and to confirm that the design is functioning as intended [66].

In this simulation, the translator receives a untranslated string “SHOULD ” and a one-bit signal “ready_rx” from the serial receiver, and sends translated codes to the serial transmitter through the output signal “out_char”. The “ready_rx” signal is to indicate that data are ready on the output “datain” of the serial receiver, which is the string “SHOULD ” in this example. The translation startes by detecting the input character “space”. Another input named “ready_tx” is a output signal from the serial transmitter. A handshaking scheme is used between the serial transmitter and translator. Both of these two signals are initialised to logic ‘1’. A logic ‘1’ on the signal “ready_tx” means the serial transmitter is ready to receive data. In this case, a logic ‘0’ will be generated on “wr” when the translator has translated codes to send.

The serial transmitter will store the translated data by detecting the falling adge of “wr”, and then generate a logic ‘0’ on “ready_tx” showing its busy with sending data. Conrrespondingly, the translator will set “wr” to logic ‘1’ again to wait for another high of “ready_tx”.

The translating-controller block, shown in Figure 20, is to control the translating process. This block activate the translation process by sending a control signal “convert” and untranslated data “d1” to “d12” obtained from the serial receiver to other blocks. Meanwhile, it also gets feedback from the load-translated-codes block. In Figure 21, a intenal signal “empty” generated by the translating-controller block is used as a indicator to notice other blocks the state of the translator. When “empty” is in logic ‘1’, it represents the translator is receiving data from the serial transmitter, or sending data to the serial transmitter. On the contrary, a logic ‘0’ on “empty” shows the occurrence of translating processes. In this system, translation is carried out word by word, with spaces used as stop signs. In one translation period, the translating controller receives one word and the following space, translates it, and waits for the next word.

The load-translated-codes block feeds back the number of translated characters and the translating controller will skip over those characters and find a new entry. In the simulation given in Figure 21, a internal signal “num_translated” is used as the feedback number. The original text contained in signal “d1” to “d12” will be sent to the focus-check block and right-context-check block, and the entry character “d1” to the find-entry block.

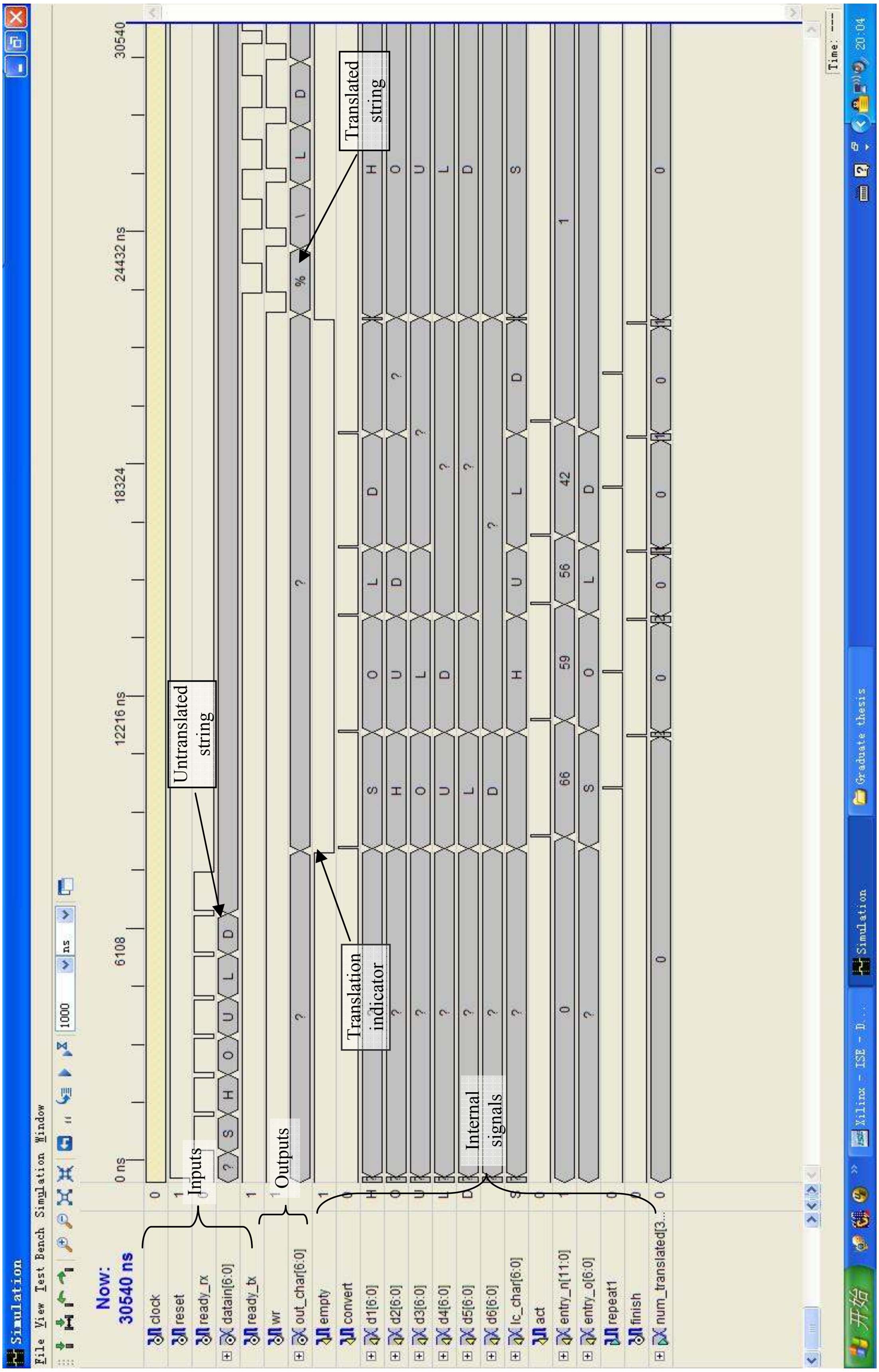


Figure 21. Behavioural simulation for translating a string “SHOULD ”

The find-entry block receives the entry character from the translating controller and outputs a particular address to the output-rule block. In this block, there is a look-up table which stores all the entry addresses. If an address corresponds to a particular entry character, this address signal "entry_n" and an address ready signal "act" will be sent to the output-rule block. However, if no entry address can be found corresponding to a particular character, this character and a find-address fail-signal will be sent to the output-translated-codes block, and these two signals will pass through the next four blocks.

Two operations keep running in the output-rule block. One is reading rules from the look-up-table block, and the other one is sending every single rule to focus-check, right-context-check, left-context-check, and load-translated-codes blocks. Input signals for the output-rule block are all the outputs from the find-entry block, the look-up table block and feedback signals from load-translated-codes blocks which indicate if the output rule is used correctly or not. However, because no enough space is available, the internal signals for the the output-rule, focus-check, right-context-check, and left-context-check are not included in the simulation.

The look-up-table block consists of 18 block RAMs, which are configured to a 1200*256 bits memory, and can accommodate the whole rule table with total 1031 entries. The look-up-table block remains in read mode after writing the rule table. Therefore, once receiving the address from the find-entry block, the output-rule block will send the address to the look-up-table block to read one rule and send it separately to focus-check, right-context-check, left-context-check and out-rule block. If the rule cannot be used, a feedback signal will be generated and the output-rule block will get the next rule and send it until the focus is successfully translated.

The “focus” is one or more characters in the original text which should be translated. According to Paul Blenkhorn’s method, the same focus can have different left and right contexts. Therefore, checking left and right context is necessary. If a string is identical to the focus in a particular rule, the right and left contexts need to be checked as well. If all three parts match, the rule fires, and the string can be translated.

Focus-check and right-context-check blocks receive not only the rule from output-rule block, but also the whole group of words to be translated from the translating controller, because more than one letter of focus and right context might be checked. The structures of these three blocks are similar because they perform similar functions. Therefore, here only the focus-check block will be described.

Because Blenkhorn’s algorithm is based on software procedural programming, a sequential processing is used to achieve focus, right and left context checking, where these three functions are included in different blocks, and invoked one by one [37] [40]. Apparently, a sequential design can make implementation easier, but meanwhile the processing speed could be slown down. However, the hardware-based translator, as shown in Figure 20, provides a parallelism to make focus-check, right-context-check and left-context-check blocks work concurrently, providing better performance than sequential implementations. Each block generates signals for the load-translated-codes block indicating if the focus, the right context or the left context were successfully matched. If one of the three fails, then a signal is sent back to the output-rule block requesting the next rule. If the focus, right context and left context match one of the rules, then the load-translated-codes block sends the translated codes to the output-translated-codes block, and informs to the translating-controller block how many characters were translated.

As mentioned, the output-translated-codes block has a handshake connection with a serial transmitter. After one group of characters has been translated, this block will send the characters one by one when the transmitter is ready. Then a new cycle starts.

For instance, Figure 22 shows a post-place-and-route simulation for translating a string “SHOULD AND ”. Post-place-and-route simulation is to verify that the actual gate-level implementation matches the functional behavior simulated earlier [67]. It is a critical step, because a circuit which simulates correctly with time-unit delay may not work properly when actual routed delays are added to the design. In the simulation, the translator is synchronised by a 16.6 MHz clock, that is 60 nano seconds per clock cycle. When the string “SHOULD ” is sent to the translator through the serial receiver, it takes 13 microseconds to for the translator to do the conversion and another 10 microseconds to send the untranslated string to the serial transmitter. Then, the translator waits for the new feed-in “AND ”.

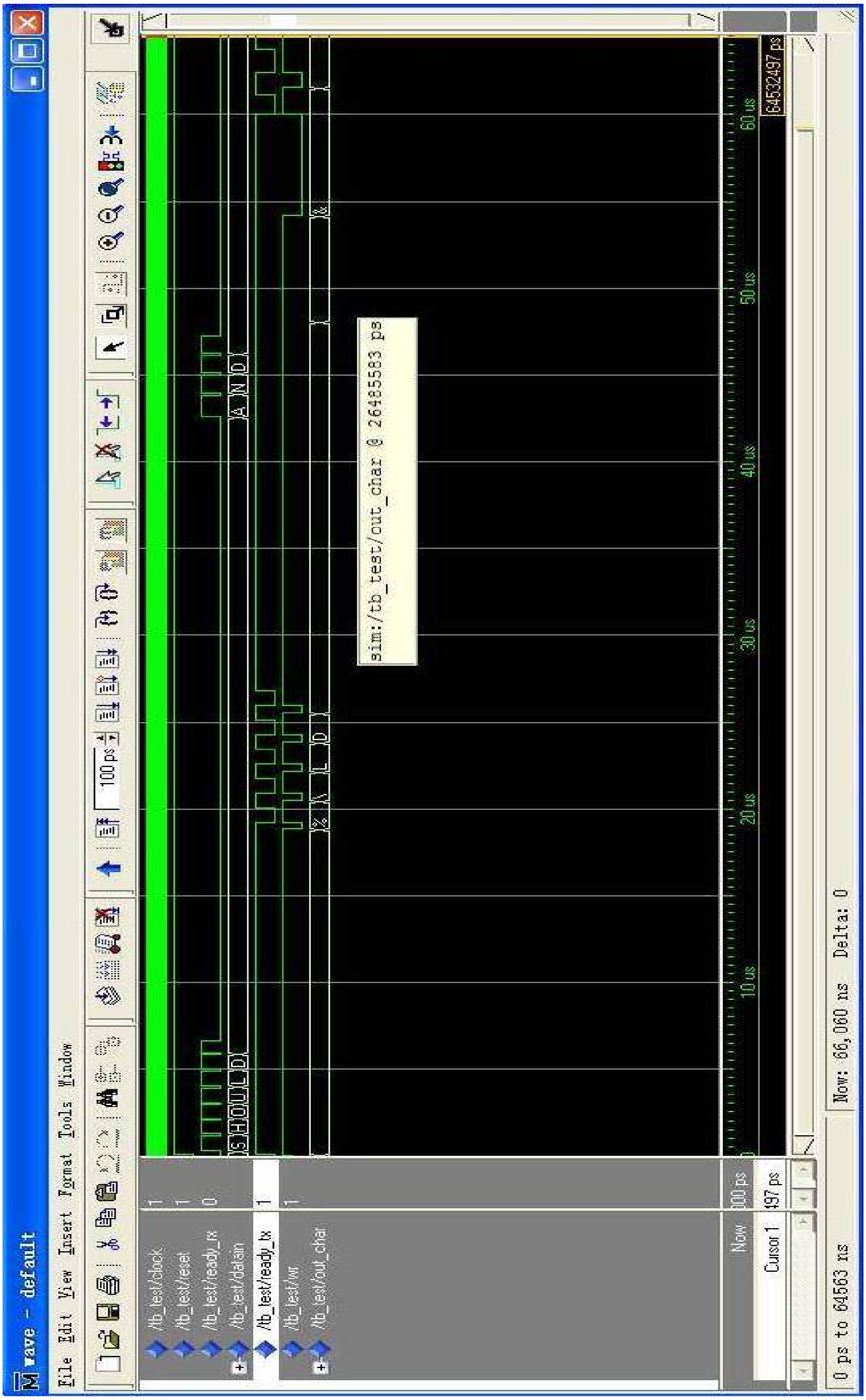


Figure 22. Post place and route simulation for translation a string “SHOULD AND ”

5.2.3 Implementation and Test

In this particular implementation, the translator uses a 100 MHz clock source. The serial channel receives from a PC the text file to be translated at 4800 baud and sends the translated text back to the PC at 57600 baud. In this setting, the translator runs much faster than the serial communication channel. The translation process can be finished in the time required to transmit one bit of information back to the PC. However, because this system is only able to translate groups of characters, after translation is done, the serial transmitter has to send all the translated codes to the computer before the next group of text is received. That is why the baud rate for transmitting is 16 times faster than for receiving. However, in embedded applications, parallel communications can be used to increase the throughput.

All the blocks of the serial communication and the translator have been built hierarchically using VHDL, and the coding program can be found in Appendix V. Xilinx's ISE FPGA-development suite was used for system implementation, synthesis, simulation and FPGA configuration. The device utilisation summary of this implementation generated by the software is shown in Table 9. A Xinlix Virtex-4 XC4FX12 FPGA is used for the system implementation. To test the system, text files were sent and received from a PC using Hyper Terminal, Windows's terminal emulation tool.

Table 9. Device Utilization Summary (Virtex-4 XC4FX12)

Logic Utilization	Used	Available	Utilization
Number of Slices	2836	5472	51%
Number of Slice Flip Flops	2619	10944	23%
Number of 4 input LUTs	4621	10944	42%
Number of bonded IOBs	5	320	1%
Number of FIFO16/RAMB16s	19	36	52%
Number of GCLKs	8	32	25

In some commercial Braille translation programs, there is an option to use only capital letters for the translation results. In this system, all the text is converted to capitals before translation. This is partly because the rule table in Blenkhorn's system only includes capital letters, and also because this method can decrease the complexity of implementation.

For comparison purposes the output of Duxbury, a commercial Braille translation program mentioned in Chapter 3, has been used. The text used is a fragment of Paul Blenkhorn's paper [37].

In Table 10, the results show that the hardware translator is able to complete the translation successfully.

The main difference between the two translations is how quotation marks are translated. Duxbury translates them as @, while the hardware-based system translates them as 8 and 0. This is because Duxbury can not recognise 8-bit ASCII codes such as quotation marks and commas, but the hardware system can.

The single most important issue that anyone interested in Braille translation must appreciate is that results must be essentially error-free. Standards for Braille translation are much higher than for print. This level of accuracy is necessary because Braille uses the same cells for different purposes in different contexts. As a consequence, even slight errors can cause extreme difficulties in interpretation. Therefore, all translations must be very precise. The results show that this hardware translating system is able to complete the translation successfully and precisely.

Table 10. Original text and translations generated by software and by hardware

Original text	Translation by Duxbury commercial software	Translation by hardware
THE PRODUCTION OF BRAILLE USING COMPUTERS IS NOW WELL ESTABLISHED, AND THERE HAVE BEEN A NUMBER OF METHODS EMPLOYED TO ACHIEVE THIS, PARTICULARLY FOR AMERICAN ENGLISH BRAILLE. HOWEVER, IT HAS BEEN NOTED THAT THERE IS A NEED FOR THE “DEVELOPMENT OF COMPUTER SOFTWARE WHICH IS EASILY ADAPTED FOR TRANSLATING TEXT TO CONTRACTED BRAILLE FOR LANGUAGES SUCH AS HINDI AND PORTUGUESE” [1, P. 30], AND ONE OF THE MAJOR GOALS OF THE WORK REPORTED HERE HAS BEEN TO ADDRESS THIS NEED. A FURTHER GOAL HAS BEEN TO DEVISE A SYSTEM THAT CAN BE READILY UPDATED AND MODIFIED, BY PEOPLE WHO ARE NOT EXPERTS IN COMPUTER ALGORITHMS, IN ORDER TO REFLECT CHANGES/-ENHANCEMENTS TO THE BRAILLE RULES OF A GIVEN LANGUAGE. MANY EARLIER SYSTEMS, ALTHOUGH EFFECTIVE TRANSLATORS, HAVE PROVED DIFFICULT TO MODIFY FOR EITHER SUCH MINOR CHANGES OR FOR NEW LANGUAGES.	! PRODUC;N (BRL US+ -PUT]S IS N[WELL E/ABLI%$1 & "! H BE5 A NUMB] (ME?ODS EMPLOY$ 6A*IEVE ?1 "PICUL>LY = AM]ICAN 5GLI% BRL4 H["E1 X HAS BE5 NOT$ T "! IS A NE$ =! @IDEVELOP;T (-PUT] S(TW>E : IS EASILY ADAPT$ = TRANSLAT+ TEXT 63TRACT$ BRL = LANGUAGES S* Z H9DI & PORTUGUESE@I ,7#A1 P4 #CJ7'1 & "O (! MAJOR GOALS (! "W REPORT$ "H HAS BE5 6A4RESS ? NE$4 A FUR!R GOAL HAS BE5 6DEVISE A SY/EM T C 2 R1DILY UPDAT$ & MODIFI$1 0P :O >E N EXP]TS 9 -PUT] ALGORI?MS1 9 ORD] 6REFLECT *ANGES/-5H.E;TS 6! BRL RULES (A GIV5 LANGUAGE4 _M E>LI] SY/EMS1 AL? E6ECTIVE TRANSLATORS1 H PROV$ DI6ICULT 6MODIFY = EI S* M9OR *ANGES OR = NEW LANGUAGES4 AL? ? SY/EM HAS BE5 DESIGN$ 6COPE)A L>GE NUMB] (DI6]5T LANGUAGES1	! PRODUC;N (BRL US+ -PUT]S IS N[WELL E/ABLI%$1 & "! H BE5 A NUMB] (ME?ODS EMPLOY$ 6A*IEVE ?1 "PICUL>LY = AM]ICAN 5GLI% BRL4 H["E1 X HAS BE5 NOT$ T "! IS A NE$ =! 8DEVELOP;T (-PUT] S(TW>E : IS EASILY ADAPT$ = TRANSLAT+ TEXT 63TRACT$ BRL = LANGUAGES S* Z H9DI & PORTUGUESE0 ,7#A1 P4 #CJ7'1 & "O (! MAJOR GOALS (! "W REPORT$ "H HAS BE5 6A4RESS ? NE$4 A FUR!R GOAL HAS BE5 6DEVISE A SY/EM T C BE R1DILY UPDAT$ & MODIFI$1 0 P :O >E N EXP]TS 9 -PUT] ALGORI?MS1 9 ORD] 6REFLECT *ANGES/-5H.E;TS 6! BRL RULES (A GIV5 LANGUAGE4 _M E>LI] SY/EMS1 AL? E6ECTIVE TRANSLATORS1 H PROV$ DI6ICULT 6MODIFY = EI S* M9OR *ANGES OR = NEW LANGUAGES4 AL? ? SY/EM HAS BE5 DESIGN$ 6COPE)A L>GE NUMB] (DI6]5T LANGUAGES1

Another significant issue to discuss is speed. The timing results show that under the same working frequency, the FPGA is able to perform translation much faster than a microcontroller. In Table 11, a Mitsubishi microcontroller M16c/62 is compared with the FPGA used in this design. In the microcontroller, the translation software achieves the same results as the FPGA does. Both of them have the same working frequency supplied by clock. However, when they translate the first rule of the A table, the FPGA is much faster. Especially, as the last rule of the A table is translated, it is easy to see that the FPGA shows a much higher efficiency than the microcontroller.

Table 11. Timing Comparison between FPGA and Microcontroller

Rule	FPGA (16mHZ)	MC-M16c/62 (16mHZ)
[AND] → &	12 μs	300 μs
[A] → A	46 μs	1200 μs

5.3 Fast Text-to-Braille Translation

5.3.1 Algorithm

Since the translator follows the Markov algorithm, the rules have to be checked from top to bottom sequentially [71]. For instance, in Blenkhorn’s text-to-Braille translation system, all rules are listed in ASCII alphabetical order. For rules whose focuses start with the same character(s), the order in which they appear in the table is related to their priority. The first rule which is found has to be used.

Take the rule table with the letter ‘A’ at the beginning as an example to explain the translation process. There are 50 rules in this group, and the terminating rule is “1 [A] = [A] -”. If a contraction “AR” needs to be translated, the system has to check the 5 rules before the rule “2 [AR] = & -”. In this case, the string “AR” can be translated quickly. But, if the word “ALTOGETHER” needs to be translated, the system has to check 36 rules before the rule “2 ~ [ALTOGETHER] = ALT -”. Especially, when only the terminating rule has to be used, the translation speed will be slowed down significantly. However, the translation process can be accelerated if the ‘A’ group is separated into small subgroups which can be used in parallel.

The results show that based on Blenkhorn’s algorithm, the system is able to perform the translation precisely. However, when mass text documents need to be translated, a faster method for text-to-Braille translation is obviously preferred. Therefore, in the following paragraphs, a parallel translating method is discussed.

To achieve faster translation, independent translating cells have been built. In each cell, there is an alphabetically ordered sub-table. During the translation process, those translating cells which are activated perform translation concurrently.

The principles for generating subgroups can be described as follows:

- Keep the original order of the rule table unchanged.
- For letter rules, use original terminating rules as one single subgroup, called the terminating subgroup. The cell which stores the terminating rules is called the terminating cell. Therefore, when translation is performed, the terminating subgroup never fails to be used.
- Rules have to be separated into groups properly, so that only one translating cell except the terminating cell is able to apply a particular rule successfully during the translating process. Therefore, if one rule’s focus is part of another rule’s, and there is no left and right context to distinguish between these two rules, they can not be separated.

Take the ‘A’ rules as an example to explain the principle of generating subgroups. In the ‘A’ table, those rules with focus “AND ” and “AND” are used as a subgroup. In this case, the contraction “AND” will never be translated by two cells. The rules beginning with the string “AFTER” need to be used in one subgroup. Using this method, the ‘A’ rules can be separated into 7 subgroups, while the biggest table, ‘B’ table with 122 rules, can be separated into 9 subgroups. For those tables with a small number of rules, such as the ‘J’ table which only has 10 rules, it is not necessary to separate these rules into subgroups.

5.3.2 Architecture

Figure 23 shows a block diagram of the text-to-Braille translator implemented in an FPGA. Before the translation starts, the data-controller receives the rule tables and distributes them to particular block RAMs located in translating cells. Then the data-controller is ready to receive text.

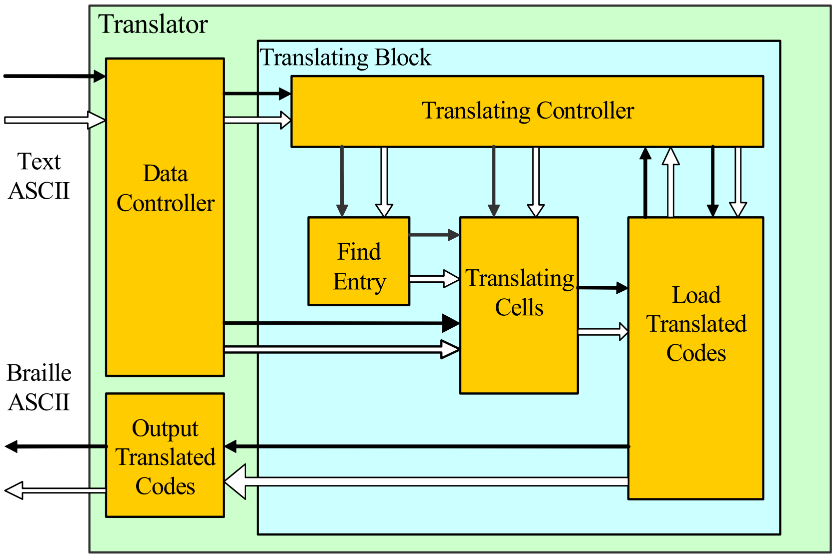


Figure 23. Block Diagram of Text-to-Braille Translator

The translating controller block gets feedback from the load-translated-codes block and also receives and stores the text data in registers. The load-translated-codes block feeds back the number of translated characters so that the translating controller can skip over those characters and find a new entry. The entry character is sent to the find-entry block. The original text is sent to translating cells. In this particular implementation, the translator carries out the conversion word by word and five words at a time.

The find-entry block receives one entry character from the translating-controller and outputs addresses for the corresponding translating-cells. The entry character is the first un-translated character in the input text string. In the find-entry block, there is an address decoder that translates the entry characters into addresses. If no entry address can be found for a particular character, then the un-translated character and a fail signal are sent to the output-translated-codes block.

The translating cells receive un-translated codes from the translating-controller as well as addresses from the find-entry block. The parallel translating processes is shown in Figure 24. Those cells which received addresses will carry out the translation.

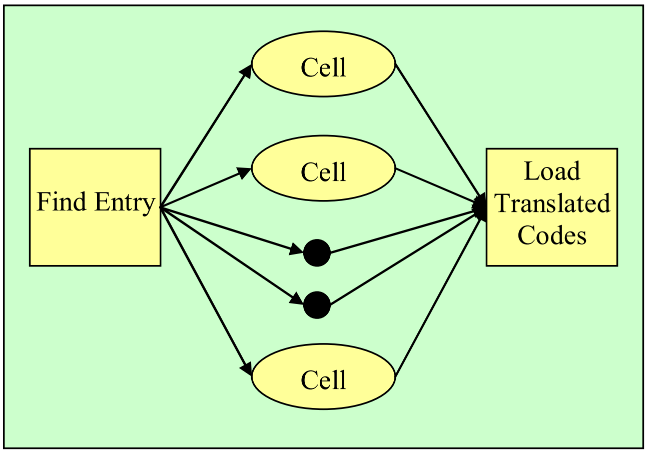


Figure 24. Translating in Parallel

Before the architecture of translating cells is given, it is necessary to discuss the scheme for constructing a rule look-up table with multiple access. The rule look-up table performs as a Read Only Memory (ROM), because it just needs to be written once, and then only read operations are allowed. To let each translation cell fetch particular rules from different addresses, an architecture given in Figure 25 would be the best solution. In this diagram, only a group of four 32-bit registers are used in the look-up table. To write a particular register, an address should be sent to the 2-bit input address, and meanwhile the input "write" must be set to logic high. The outputs of registers are connected to two 4-to-1 multiplexers Mux0 and Mux1. Hence, this look-up table supplies two output ports where any value from each of the registers can be read by selecting a particular address at inputs "Address A" and "Address B". However, when this architecture is applied to the parallel text-Braille translation, it becomes very difficult to implement due to the size of the look-up table. For instance, to do the translation by using nine translation cells, a 1031*256 bits RAM with a group of nine 256-bit multipexers is needed, but there is no sufficient distributed RAM available in the FPGA currently used. Therefore, another solution has been used in this implementation, that is to use block RAMs integrated in FPGAs.

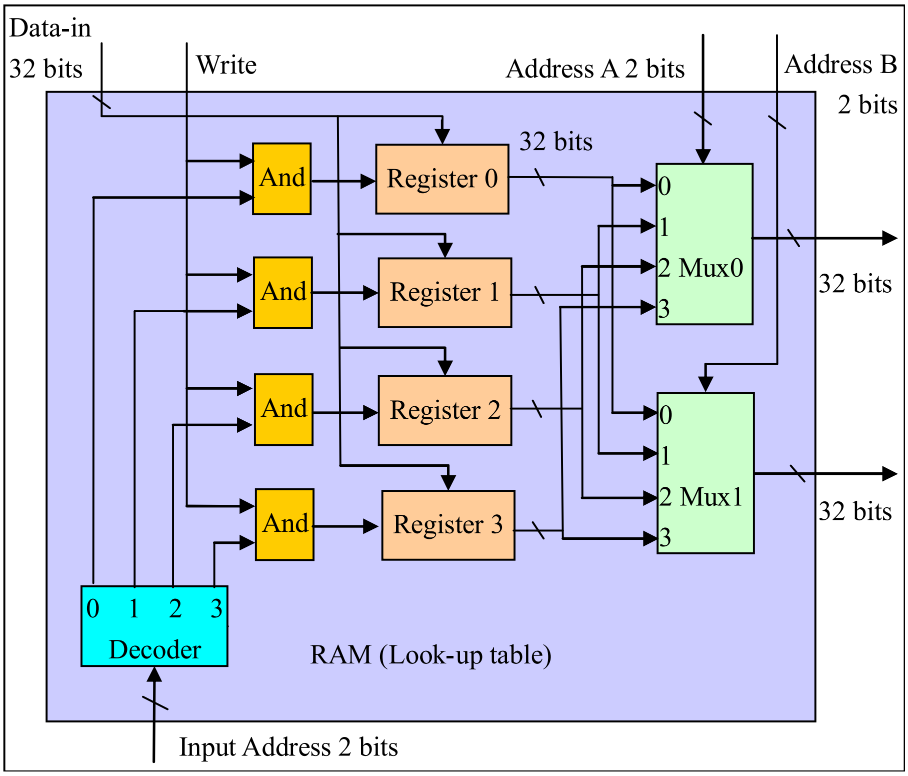


Figure 25. A look-up table to 2 output ports

In each translation cell, there is a look-up table constructed using block RAMs. Figure 26 shows the block diagram of a translating cell. Every cell has a block RAM where a subset rule table is stored in alphabetical order. Before the translation process starts, the un-translated codes from the translating-controller are sent to the focus and right-context check blocks by the output-rule block. Then the output-rule receives an address and gets a particular rule from the rule table. The rule will be separately sent to the three following blocks. The focus, right-context and left-context check blocks are built using finite state machines which are able to check if the rule can be applied.

As shown in Figure 26, the three blocks work concurrently, providing better performance than sequential implementations [68].

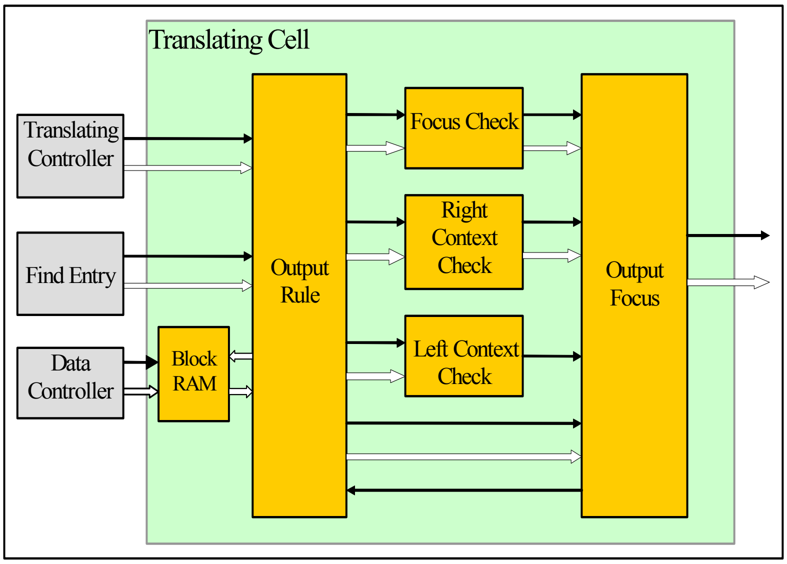


Figure 26. Block Diagram of a Translating Cell

Each block generates signals for the output-focus block indicating if the focus, the right context or the left context were successfully matched. The translation output will be sent to the load-translated-codes block. If one of the three fails, then a signal is sent back to the output-rule block requesting the next rule. If no rule can be used, a signal will be generated and sent to the load-translated-codes block indicating that the translating cell cannot find a match for translation.

The load-translated-codes block will receive translation results from the terminating cell or one of the other cells. The terminating never will fail to be applied. However, compared with other cells, the terminating cell has lower priority. Therefore, if the load-translated-codes block receives translated codes from two cells respectively, the codes from the terminating cell will be discarded.

Therefore, the load-translated-codes block will output the translation according to set priorities. Meanwhile, it will send signals to the translating-controller block to indicate how many characters were translated.

After one group of characters has been translated, the output-translated-codes block transmits the corresponding Braille ASCII characters one by one. Then the translation of a new set of characters can begin.

5.3.3 Implementation and Results

The testing system follows the same method as the hardware based translation system, as show in Figure 18. The program for fast translation in VHDL can be found in Appendix VI. To show how the translating process is accelerated, sample translation results have been presented.

The texts to be translated, as well as the results of the translation were stored in a PC as text files and transmitted using an RS-232 serial connection.

The testing system works as follows:

1. The text to be translated is sent to the FPGA through a serial link using Hyper Terminal.

2. Part of the FPGA implements a receiver that converts serial data into bytes that are loaded into the translator.

3. The translator takes the new character and stores it in a buffer. Characters are stored until a space is detected. At this point the translation process described in section 2 takes place.
4. The results of the translation are sent to a serial transmitter so that they can be received and stored in a text file by the computer.

In this implementation, the FPGA receives the text file to be translated at 4,800 baud and sends the translated text back to the PC at 57,600 baud. The reason for using different baud rates for receiving and transmitting has been explained in Section 5.2.

To simplify the implementation, all rules were modified to be the same length. ASCII code 0 was used as the end-sign for every part of the rule. As a consequence, a rule table with 1031 rules occupies 31Kilo Bytes memory. However, Virtex-4 FPGAs have dedicated memory blocks that can contain the complete table.

For testing, outputs of the hardware translator were compared with the outputs of the previous work which uses the sequential translating method [65]. Using the simulation tool, ModelSim, the numbers of clock cycles using sequential and parallel methods can be accurately calculated. To show how the translation process goes inside the translator, only three translting cells are built for simulation purposes, because it is impossible to display all internal signals in one simulation with more than three cells.

In this particular implementation of the parallel translator, the ‘A’ rule set is separated into three groups which are stored in three cells respectively. By comparison, a sequential translator is used to translate the same string as the parallel one does, and both translators use a 25 MHz clock source which is a clock with 40 nano seconds per cycle. The behavioural simulation, shown in Figure 27, indicates that translation processes are being operated concurrently in three translation cells. It takes 4110 nano seconds or 103 clock cycles to translate the string “ACROSS ”. Correspondingly, in Figure 28, a long delay, 28560 nano seconds or 741 clock cycles, is generated when the rules are checked sequentially. This is because the rule for translating a string “ACROSS” is stored as the first rule of ‘A’ rule set in the third translation cell. Therefore, the parallel is able to fectch this rule very fast. A further comparison is given in Table 12, by listing the sample test results which are based on six translation cells. The results show that the parallel method is able to perform translations with superior speed.

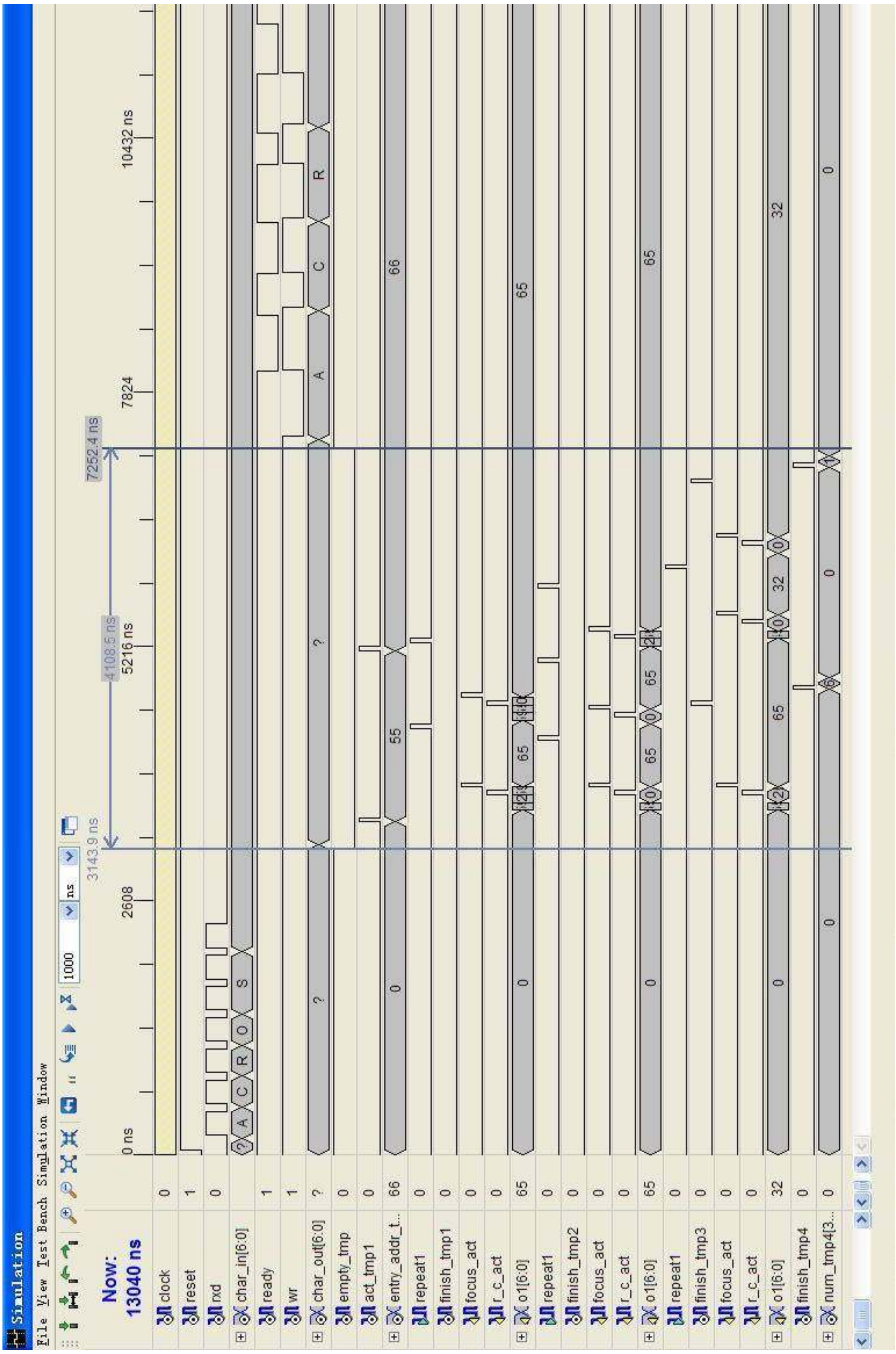


Figure 27. Behavioural simulation for translating a string “ACROSS ” in parallel

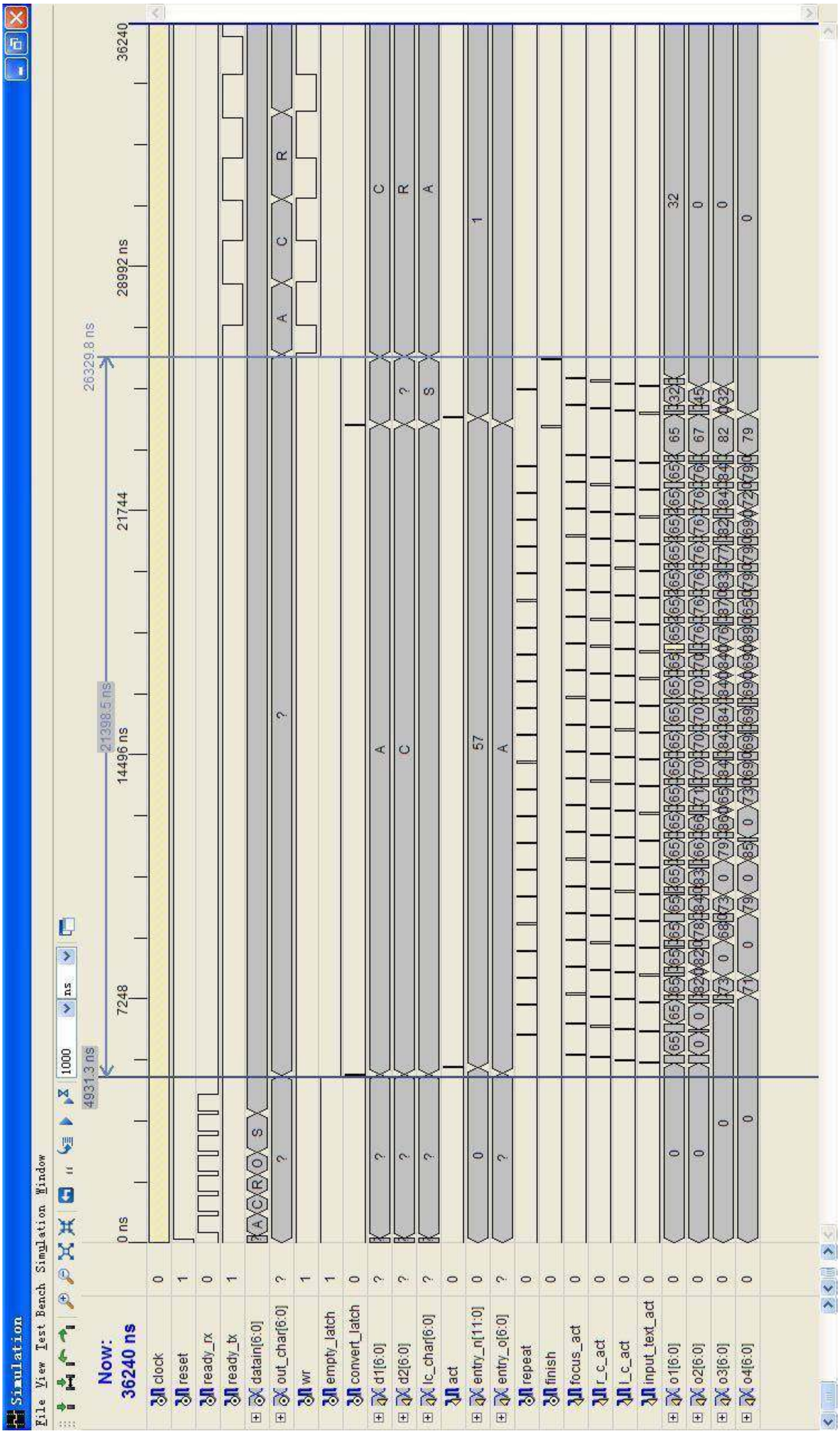


Figure 28.Behavioural simulation for translating a string “ACROSS ” in sequential

Table 12. Timing comparison between sequential and parallel methods

Un-translated focuses	Translation results	Time by sequential method (clock cycles)	Time by parallel method (clock cycles)
ARIGHT	A”R	132	136
AND	&	137	142
AS	Z	392	105
ABOUT	AB	410	129
ABOVE	ABV	434	154
AFTERNOON	AFN	477	104
AFTERWARD	AFW	499	127
AFTER	AF	571	190
ALWAYS	ALW	595	104
ALREADY	ALR	660	133
ALSO	AL	614	157
ACROSS	ACR	714	103
ACCORDING	AC	739	134
AUND	AUND	804	103
AINES	A9NES	843	128
A	A	926	207

Although the parallel processing algorithm is able to supply a much greater throughput than the original translation algorithm, this is achieved at the price of more programmable logic needed and more complex structure developed. Compared to the device introduced in Section 5.2, a bigger Xilinx Virtex-4 xc4vsx35 FPGA is used in this design. The Virtex-4 xc4vsx35 FPGA contains 15360 programmable slices in total, and there are 4 configurable logic blocks (CLB) in one slice [70]. Table 13 shows how the occupancy of programmable slices of the translator grows with the number of translation cells.

Table 13.Resource occupation of Fast Braille Translation

Number of translation cell	Number of slices occupied	
	Number	Percentage
1	2918	19%
2	3379	22%
3	3994	26%
4	4608	30%
5	5069	33%
6	5530	36%
7	6144	40%
8	6758	44%
9	7219	47%

5.4 Braille-to-Text Translation

5.4.1 Architecture

As discussed in Chapter 3, Braille-to-text translation is referred to as a opposite string rewriting process compared to text-to-Braille translation. However, because both translations are context sensitive, so the algorithm of Braille-to-text translation is very similar to the one used in the text-to-Braille translation, and this decides that both the systems have similar architectures.

Parallel architectures are not applied to the Braille-to-text translation because of two aspects to be considered. First, in contrast to a rule table with 1031 rules used in text-to-Braille translator, the table for Braille-to-text translation only includes 498 translating rules. Furthermore, the alphabetical rule subsets also have a much smaller size. For example, The ‘B’ table of text-to-Braille translator includes 122 rules, while the Braille-to-text translator only contains 10 rules in its ‘B’ table.

On the other hand, Braille-to-text translation is mostly used as a functional module in Braille notetakers. Braille notetaker is a small, portable device that supplies a Braille entry for the the blind to take notes. The input mechanism of a Braille notetaker is a keyboard with six keys and a space bar which is used to enter either Grade I or Grade

II Braille [73]. When the blind use a notetaker to take notes, they also can select this function to translate their Braille codes into natural languages. In this circumstance, the translating speed is not a critical issue but translating accuracy.

Figure 29 shows the architecture of the Braille-to-text translator. Instead of using left-context checking in text-to-Braille translation, the Braille-to-text translator uses a finite-state machine to achieve a decision table check. The decision check function is as described in Section 3.3.5. Since functions of other blocks are the same as those in text-to-Braille translator which have been described in detail, therefore, they won't be repeated here.

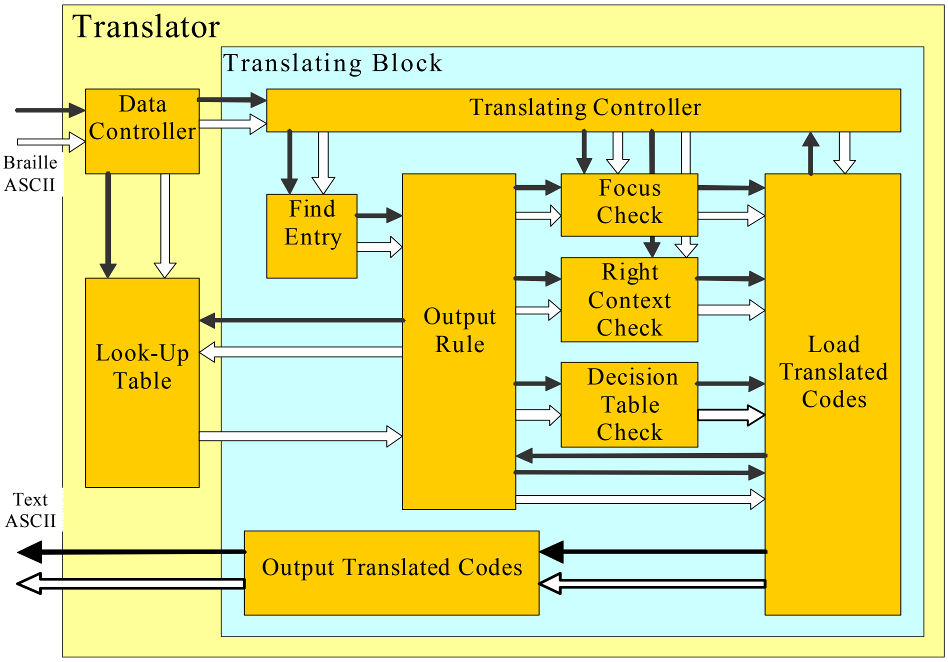


Figure 29. Block Diagram of Braille-to-Text Translator

Similarly, the testing system also follows the same method discussed in previous sections. Figure 30 shows a post-place-and-route simulation for translating a string of Braille "ABV ". The translation process is activated when the last character, a space sign is received. Then, rules will be checked sequentially until the rule "3 [ABV]=above 3" is found. The first number 3 is the input class indicating the rule can be used when the first letter of a string is at the start of a word. In this case, because the letter ‘A’ is the first letter of the string “ABV”, so this rule fires. A translated string “ABOVE ” is sent to the serial transmitter through the output “out_char”.

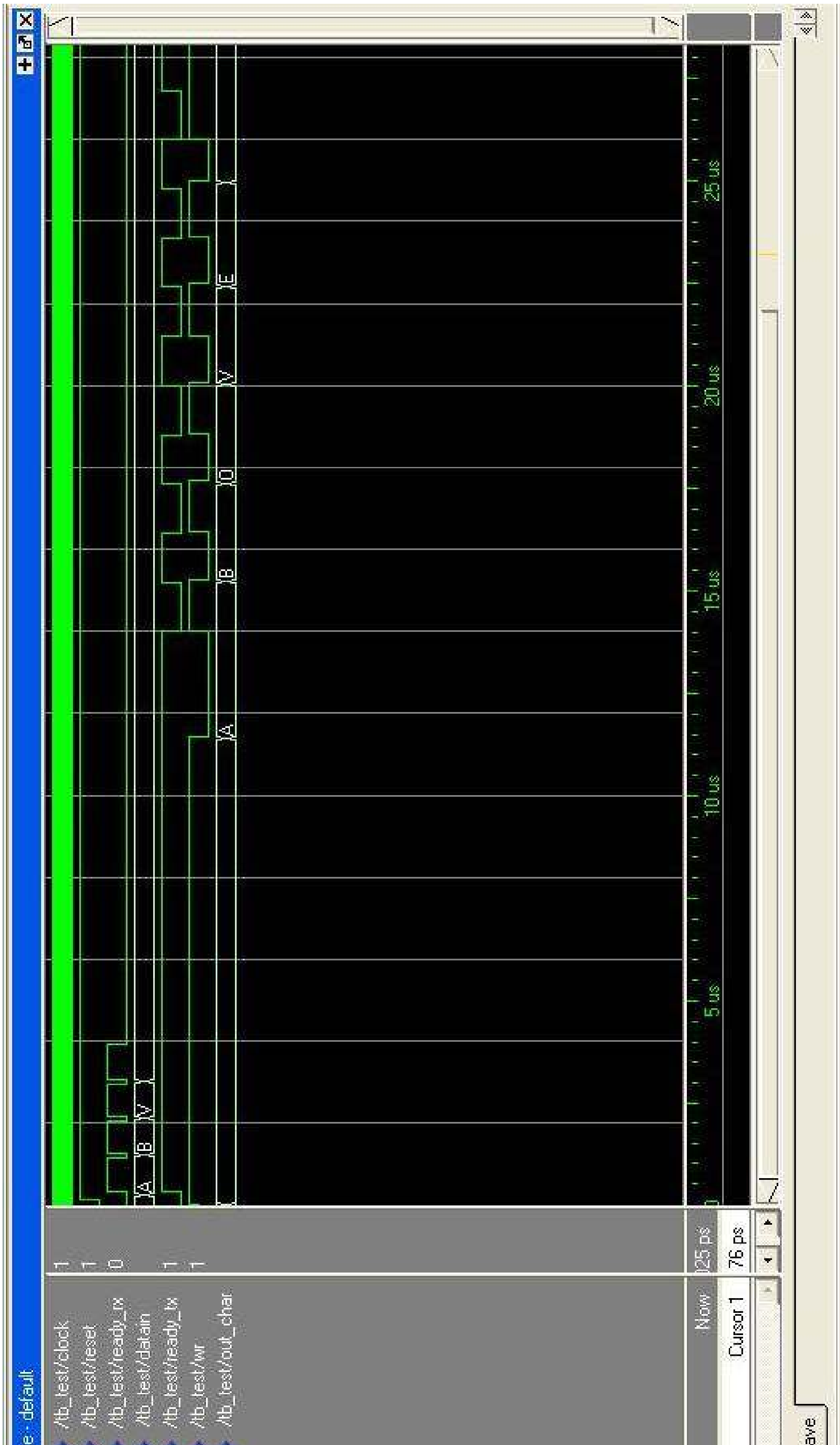


Figure 30. Post-place-and-route simulation for translating a string “ABV ”

5.4.2 Braille Keyboard

The testing system for the Braille-to-text translation is quite different from the version of text-to-Braille translation. Since Braille note takers are being widely used for the blind to take notes in Braille, and Braille-to-text translation is a typical application in these note takers, a Braille keyboard [72] [73] is integrated into the system and is used as an input device.

Most commercial Braille note takers utilise microcontrollers with software running in them to perform multiple functions including note taking, translation, and real time speaking.

When using a Braille keyboard, up to six buttons need to be pressed simultaneously. Because a simple matrix-like keyboard, shown in Figure 31, has deficiencies to deal with multiple key detections [74], so it is normally not referred to as an ideal solution for Braille keyboard. Instead, a popular one is to use optical detectors and a microcontroller to detect when buttons are depressed [75]. One example of this approach reported in D. G. Evans's paper is to use infrared light source/sensor pairs and microcontrollers [76]. When using optical detectors, if a key is depressed, it breaks the light beam between the source and sensor. Thus, the sensor generates a pulse that can be received by a microcontroller.

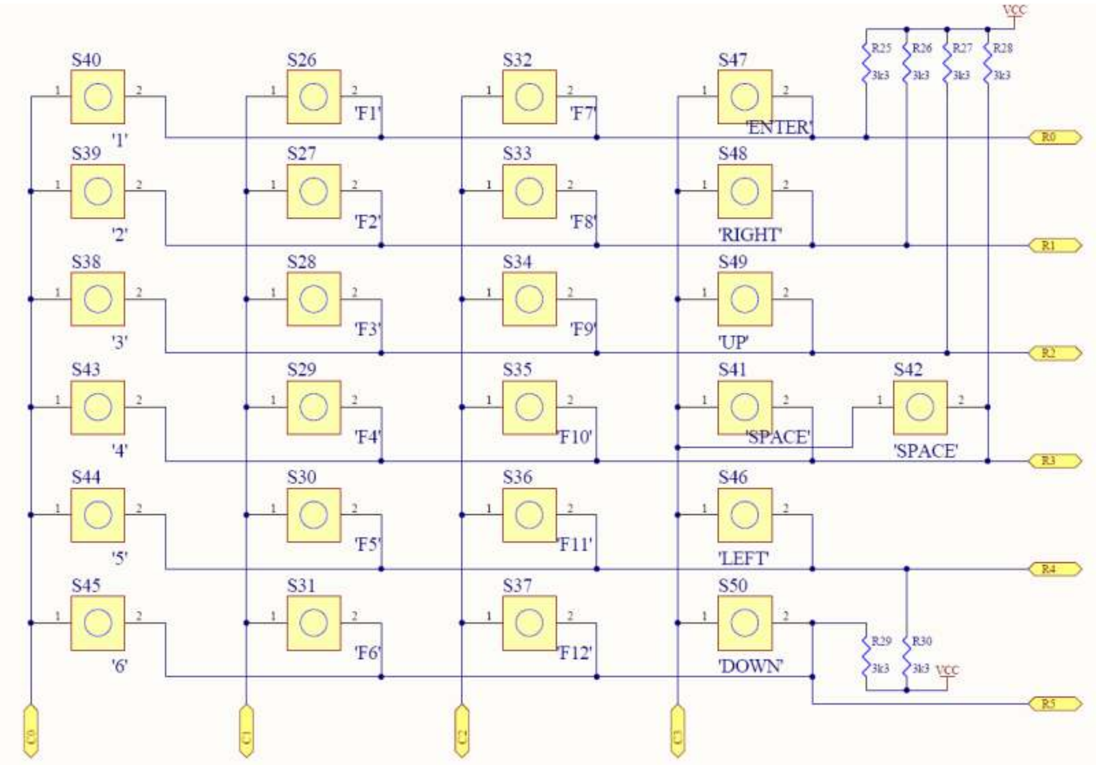


Figure 31. Schematic of Braille Keyboard

A new method was developed to implement a Braille keyboard using a simple 4 x 6 push-button matrix [77]. The buttons have been organised and positioned so as to achieve multiple key detections. A keyboard controller described in VHDL was implemented to send and receive scan codes.

Figure 32 shows the layout of the proposed Braille keyboard. In line with current designs, the keyboard has function and direction keys. The arrangment of six Braille keys and space follows Perkins-style keyboard. Perkins Brailler, invented by David Abrahams, is a Braille typewriter which has been widely used by blind people [72]. The layout of Perkins' keyboard is adopted by most of commercial Braille note taking products due to its success. To build a machine that is easy to use for the visually-impaired, the function keys can be used for selecting functions including speaking, printing, translating, embossing. Thus the direction keys can be used as "shift", "control", "capital lock" and "backspace".

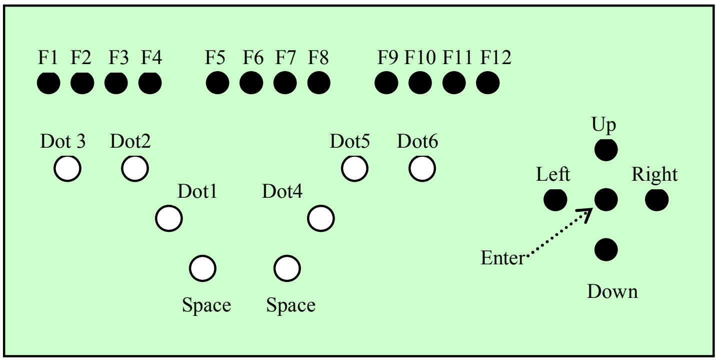


Figure 32. Layout of Braille Keyboard

Figure 33 shows the schematic diagram of the 4 x 6 matrix keyboard. The layout of the schematic shows that the six Braille keys are located in the first column, twelve functions keys in the second and third columns, and enter, space and direction keys in the fourth.

The keyboard controller keeps sending 4-bit scan codes to the keyboard inputs from C0 to C3 and scans one column at a time. To do this, a circular shift register containing a intial binary value “0111” is used in the controller. For example, Figure 32 indicates a process of scanning the first column of keys for the Braille keyboard. To scan the first column, the keyboard controller sends binary code “0111” to the inputs of the keyboard C0 to C3. If key S39, S43 and S44 are pressed down, the most significant bit ‘0’ will pass through these three keys to the output ports R1, R3 and R4. Therefore, a six-bit binary output signal “101001” has been generated and sent back to the keyboard controller. Likewise, to scan the second column, the shift register will shift right the code “0111” for one bit to generate the second scan code “1011”.

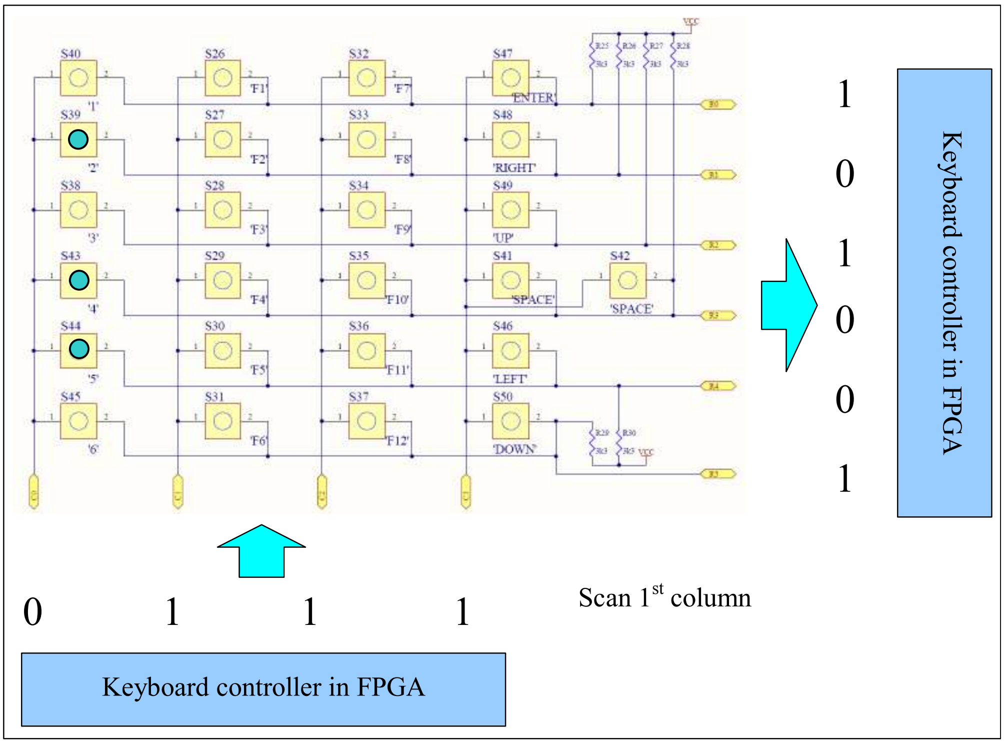


Figure 33. A process of scanning the first column of keys

When a pushbutton is pressed or released, it does not generate a clear pulse. Instead, the output may toggle as illustrated in Figure 34. This is called bouncing problem [78]. The bouncing time is very short (usually less than 10ms). To overcome this problem, the controller reads the 6-bit output of the keyboard 13 milliseconds after sending the scan codes so that the bouncing process can be avoided. If no button is pressed, the outputs from the keyboard remain high, or logic '1'. Once one button is pressed, the output at the row where the button is located goes to logic '0'.

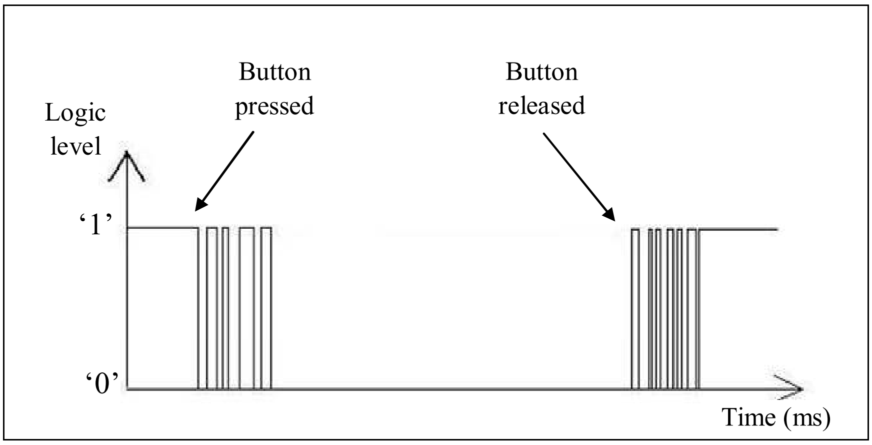


Figure 34. Bouncing process for a pushbutton

A big disadvantage of matrix keyboards is the problem of overlap. When one or two keys are pressed this type of keyboard works well. However, if more than two keys are to be pressed simultaneously, overlaps may happen.

To explain what the overlap is, a example is given. In Figure 33, if keys s26, s39, and s40 are pushed down, the wire which is connected to C0, R0 and R1 is conductive. Therefore, when the controller sends a low signal to scan the second column, this low signal passes through three buttons and goes to the output R1. The result is that instead of getting three keys pressed, the controller gets four which are S26, s27, s39, and s40. To avoid this, the six Braille keys are kept in the same column.

Figure 35 shows the block diagram of the Braille keyboard controller.

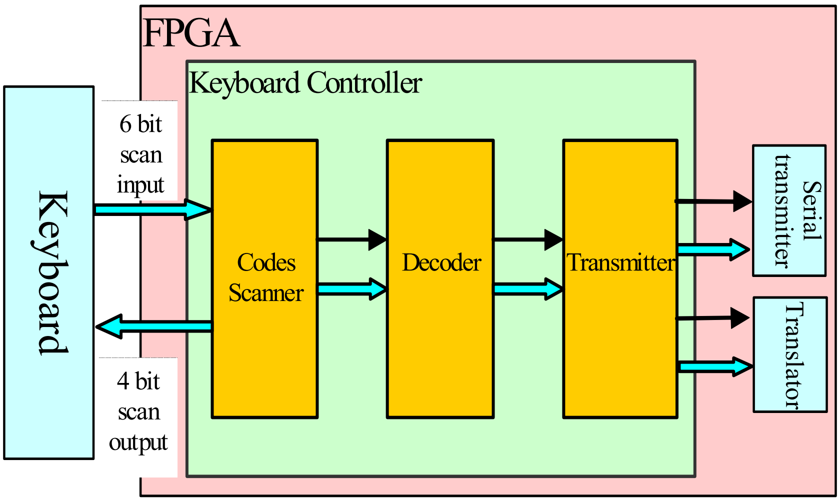


Figure 35. Block Diagram of Keyboard Controller

The codes scanner block generates 4-bit outputs with only one of the bits set to zero; and gets 6-bit signals coming back from the keyboard. There is a state machine working in the FPGA to generate scan codes for the keys that are pressed. The state machine, given in Figure 36 works as follows:

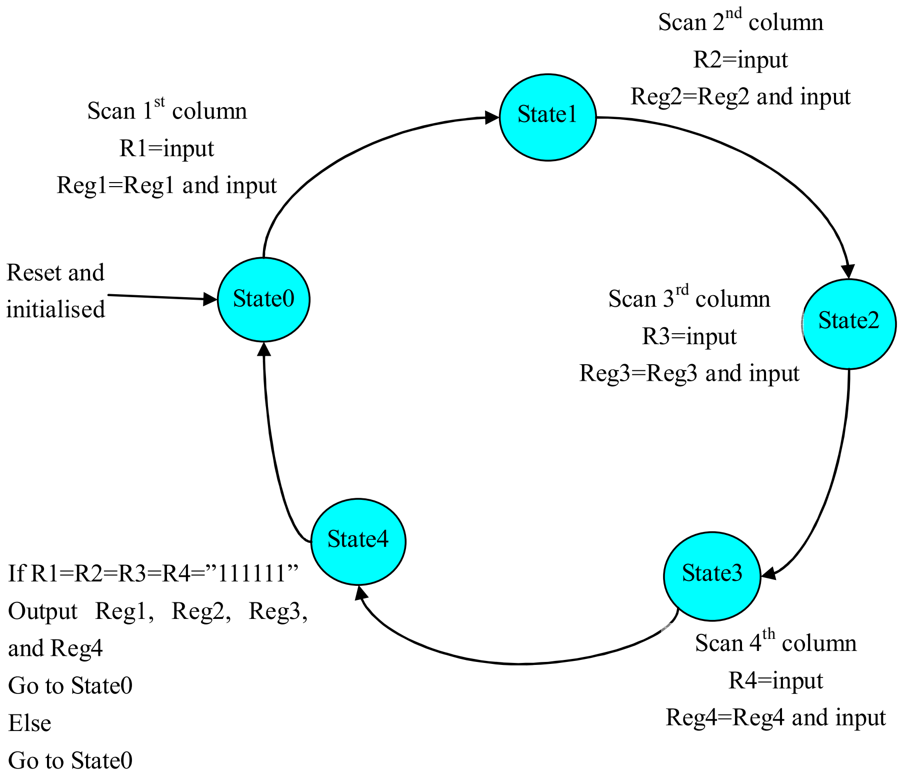


Figure 36. State diagram for the keyboard controller

- In the first state, the scanner sends 4 bits codes "0111", scanning the first column. Two registers 6-bit called R1 and Reg1 are used in this state to store input signals and both are initialised to all ones. After the debouncing process which is about 13 milliseconds, the six bits input codes will be saved into register R1. Meanwhile, a result of a logic AND operation between the input and the current value of Reg2 will be stored in Reg2.

- The same process is repeated in the next three states to scan the next three columns. But instead of "0111", codes "1011", "1101", and "1110" are sent respectively in the three states. Likewise, in each state, two registers are used to store input and the result of the AND operation. There are totally two groups of 4 registers (R1, R2, R3, R4) and (Reg1, Reg2, Reg3, Reg4).

- In the fifth state, the registers R1, R2, R3 and R4 are checked to see if the value for each register is all ones showing buttons have been released. If they have not, the state machine will go back to the first state to repeat the operation. If all buttons are released, the data stored in registers Reg1, Reg2, Reg3 and Reg4 are sent to the decoder, and then these registers will be set to all ones.

The decoder block receives the data from the codes scanner and decodes the 6-bit input to Braille ASCII codes or particular control codes corresponding to the particular buttons that were pressed. The transmitter block outputs the data to the serial transmitter or Braille-to-Text translator depending on the particular function to be selected.

5.4.3 Implementation and Test

Figure 37 shows a top level schematic diagram for the Braille notetaker. The current version of the notetaker only has three functions, which are note taking and Braille-to-text translation using a standard keyboard, and Braille-to-text translation using the Braille keyboard. Therefore, only three function keys have been defined: F1 for translations using a standard keyboard, F2 for translations using the Braille keyboard, and F3 for Braille notetaking. Figure 38 shows the system that was implemented for testing purposes.

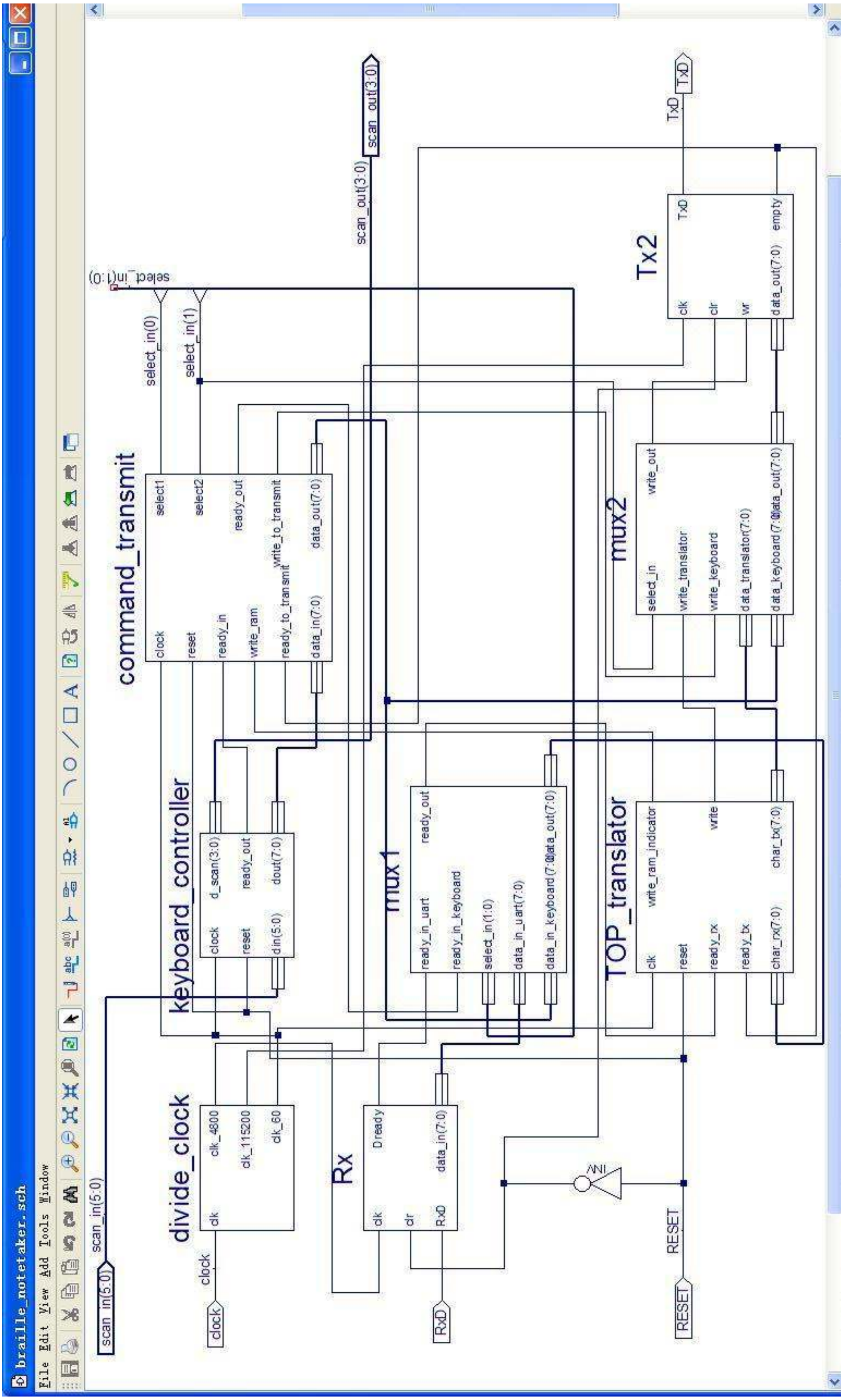


Figure 37. Schematic diagram for the Braille notetaker

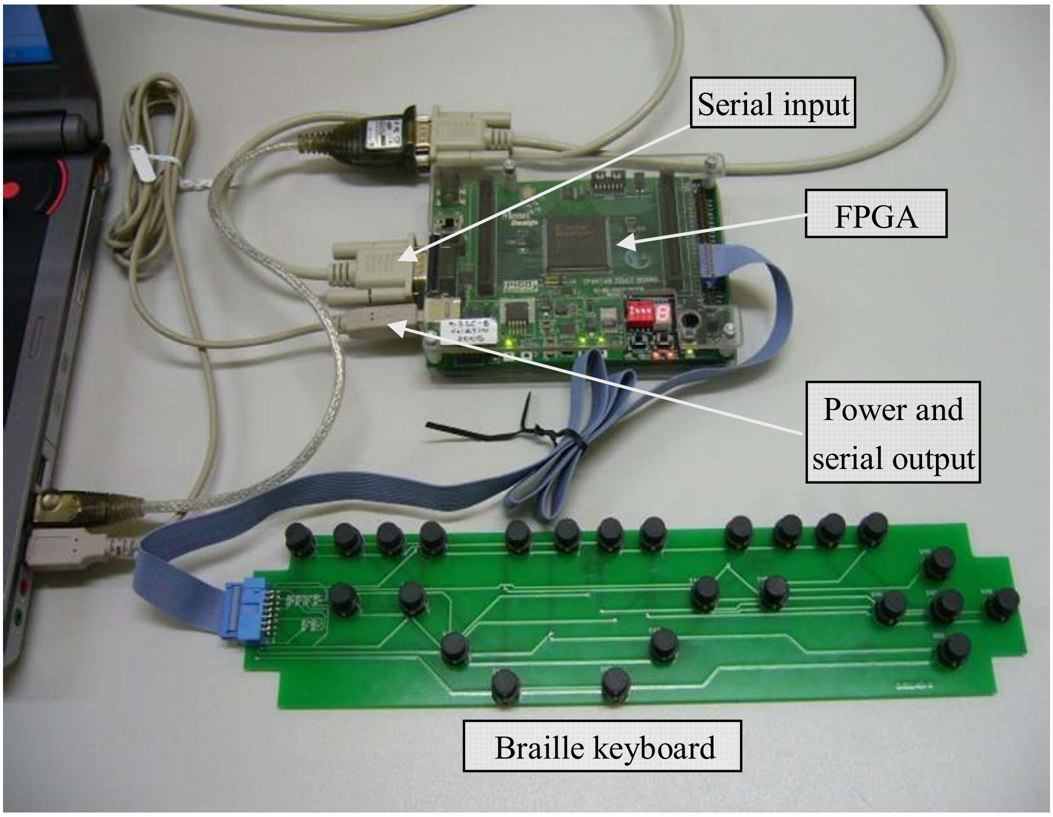


Figure 38. Testing System of Braille Notetaker

A Spartan-3 XC3S400 FPGA development board [79] is used in this design. The RS-232 serial connection is used to send the rule table to the look-up table and save the Braille codes or translation results in a text file. The system works as follows:

- The rule table is sent to the translator in the FPGA through the serial reciever Rx using Hyper Terminal before the scan process starts, and then the function will be intialised as translations using a standard keyboard.
- Then users can select a particular function to be performed using keys F1, F2 or F3 predifined on the Braille keyboard. If the note-taking function is selected, all Braille codes typed will be sent to a serial transmitter directly. A multiplexer MUX2 is used to select data from the translator or the keyboard controller to be outputs.

- If the translation function using a standard keyboard is selected, the untranslated Braille ASCII codes will be sent through the multiplexer MUX1 to the translator. The translator takes the Braille ASCIIs and stores them in a buffer. Characters are stored until a space or carriage return is detected. At this point the Braille to text translation process takes place. The results of the translation are sent to a serial transmitter Tx2 through MUX2 so that they can be received and stored in a text file by the computer.
- If translation using the Braille is selected, the MUX1 will select data from the keyboard controller to be the untranslated codes, not those from Rx. Then the translation process happens as described before.

The device utilisation summary for the implementation is shown in Table 14. The programs for the Braille notetaker has been coded in VHDL, which can be found in Appendix VII. As a stand-alone component, this system could be improved by exploring new functions. For instance, a real-time speaking function would supply auditory feedback to allow the detection and correction of typing errors. Interfaces to control a Braille embosser and a printing machine could also be incorporated.

Table 14. Device Utilization Summary (Spartan-3 XC3S400)

Logic Utilization	Used	Available	Utilization
Number of Slices	1927	3584	53%
Number of Slice Flip Flops	2275	7168	31%
Number of 4 input LUTs	3311	7168	46%
Number of bonded IOBs	19	141	13%
Number of BRAMs	6	16	37%
Number of GCLKs	8	8	100%

5.5 System Integration: A system for Text-to-Braille Translation

This section has not been finished because of time limitation. However, it is necessary to discuss the feasibility and method, because the system integration is able to supply users a complete design. It means it allows users to operate this translator without intervening the inside. For example, they do not need to program the FPGA and load the whole table to the FPGA for each time. What they need to know is just the particular function for each button on the Braille keyboard. Every time when the FPGA is powered on, it would perform as a Braille translator in a second.

To do this, the configuration of the design must be saved somewhere on the development board. When the board is powered up, the configuration will be mapped to the FPGA [80]. From this ponit, each Xilinx FPGA development board supplies at least one Xilinx Platform Flash which allows designers to store an FPGA design in nonvolatile memory [79].

Another problem to be solved is how to pre-store the rule table before using it. In the testing systems discussed in previous sections, rule tables must be loaded to translators manually every time before further operations. However, this is not what users expect. Fortunately, a Virtex-4 FX12 LC development board supplies a 4 mega-bit Flash memory [69]. Therefore, the rule table including Braille-to-text and text-to-Braille translation can be saved in this Flash memory beforehand. Every time when the design stored in Xilinx Platform Flash is configured as a translator to the FPGA, the translator will retrieve rules from the Flash memory one by one and save them in the block RAMs. The whole process can be finished when the board is powered up.

Because the same algorithm is used by Braille-to-text and text-to-Braille translators, the architectures of these two translators' are very similar. This fact makes possible the development of a translator which is capable of doing both jobs. In ths case, the parallel translating scheme will be used in the text-to-Braille translation. However, creating a multi-functional translator is helpful to simplify the design.

Figure 39 shows the block diagram of a bi-directional Braille translator. To perform both translations, a look-up table with a sufficient memory is needed to store both of the rule tables. Since two tables occupy 47 kilobytes of information, a Virtex-4 FPGA XC4VF12 with 81 kilobytes of block RAM is capable of accommodating the rule table. A mode select input is used to switch between the two translation modes. The decision-table-check block is only for Braille-to-text translation, while left-context-check is for text-to-Braille. The right-context-check block includes context information for both types of translation.

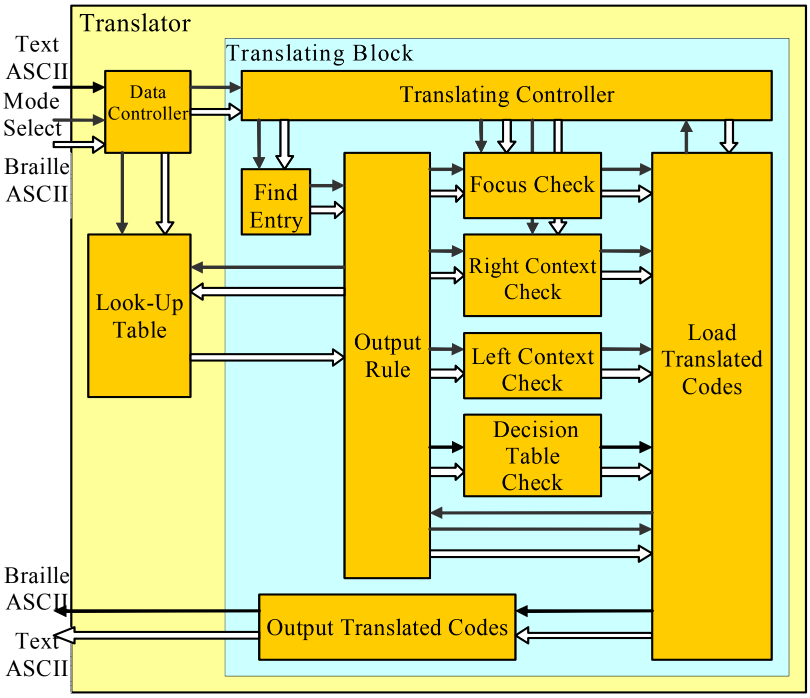


Figure 39. Block Diagram of Bi-directional Translator

Figure 40 shows a general block diagram for the whole system. There is a on-board Flash memory containing the rule table for text-Braille translations. When the board is powered up, the write-LUT block will load the table to the look-up-table block from the Flash. Then, users can select particular operations using function keys on the Braille keyboard. The functions are defined as text-to-Braille translation, Braille-to-text translation using a standard keyboard or Braille keyboard, and Braille notaking. The multiplexer Mux1 is to select data from the Hyper terminal or Braille keyboard to the input for the translator, while Mux2 is to select data from the translator or Braille keyboard to be the input for the serial transmitter. Other blocks perform the functions as described in previous sections.

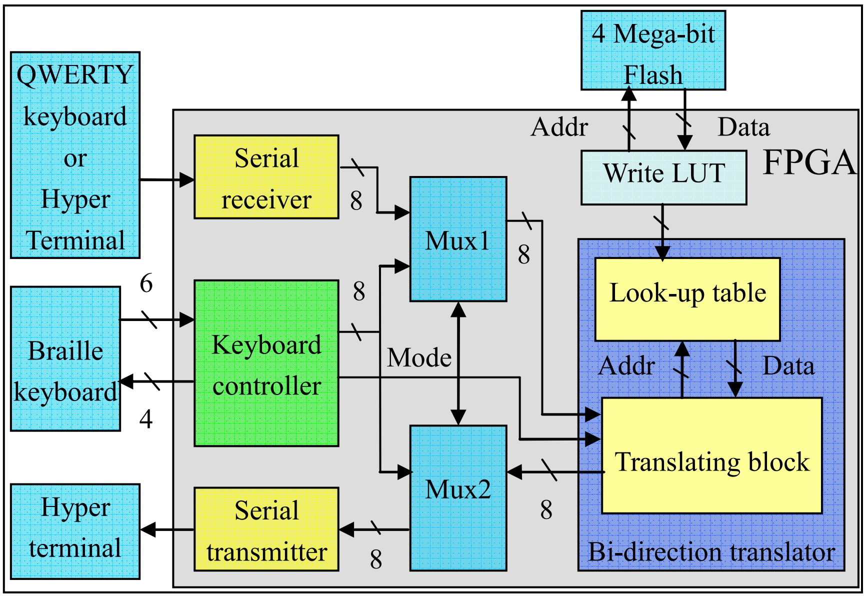


Figure 40. Block diagram of the translating and notetaking system

6. Conclusions

The original motivation of this thesis is to build a sophisticated hardward-based text-to-Braille translation system which can perform as a single module in a multi-funcitonal Braille system on a chip. Since lots of functions need to be processed for the microcontroller or processor, having a hardware implementation of the translation system is very helpful and necessary to liberate the microcontroller from the heavy load of the translation.

To find a suitable algorithm for the hardware-based text-Braille translation system, some background information is given in this thesis to explain how rules of Grade 2 Braille are used and the main difficulties for the text-Braille translation. Further, several referenced translating systems have been presented and discussed. It has been found that these translating systems appear to be software-based and tend to use look-up tables containing rules to do string substitutions [28] [33-35] [37] [40].

This look-up-table mechanism is referred to as a string rewriting system called the Markov system. What distinguishes one Markov system from another is the nature of the alphabet and the way to generate production rules. In this thesis, the rule formats of these translating systems have been analysed. In considering hardware implementation, a rule format containing context information was shown to be the most suitable because of its simplicity.

A main issue in this thesis is the algorithm reconfiguration. Lots of contents are given to explain how the software-based algorithm is reconfigured to accelerate the translation process using FPGAs. Because the parallelism is well known as a popular method to accelerate the processing speed in FPGAs, this thesis tends to pursue a parallel architecture for the text-to-Braille translation. Several parallel architectures have been used in the implementation. For example, make focus-check, right and left context-check work concurrently, and build several cells to do the translation. Moreover, in order to show that a parallelism is able to supply bigger throughput than a sequential structure, some substantial data were demonstrated. The testing results presented in this thesis prove that the parallel translation processing can be 7 times or even more faster than the sequential, and it depends on how many translation cells are used. To achieve accurate text-to-Braille translations, the rule tables proposed by Paul Blenkhorn are used in this hardware implementation [37] [40]. This parallel scheme can also be used in other language-based Braille translations. For example, the German Braille has a similar Braille translation system which includes a table with 4510 rules. In this case, the use of parallel translation cells appears to be well suited to the system and is able to accelerate the translation speed effectively.

However, a method of developing a very fast Braille embosser has not yet been found, and this may make fast text-to-Braille translation redundant. Yet, with the development of new technology, faster embossing could soon be possible. The present embossing machines are slow, mostly because they are based on mechanical parts. In the future, however, it is likely that the introduction of the laser technology and special paper sheets will make embossing speed hundreds or thousands of times faster. In that case, fast Braille translation hardware will find its own position in the marketplace. On the other hand, the design of the translation system presented a solution of using large look-up tables. The parallelisms used in the translator could be suitable for other applications. For example, the design given in Figure 25 demonstrates a register file which can be accessed to any address from different output ports. This could be used to build a fast look-up-table-based multiplier.

Another component, a Braille keyboard interface has been described in the testing system for Braille-to-text translation. As explained in the thesis, the Braille keyboard employs a matrix-like structure which is quite different from most of Braille keyboards available in commercial markets. This supplies another solution of Braille keyboards, which proves to be stable, but more affordable. Using this Braille keyboard, the system is able to achieve the basic functions of a notetaker, including note taking and Braille-to-text translation. A method of how both text-Braille and Braille-text translations are integrated in one FPGA has also been discussed in the last section. The system integration is able to supply a complete Braille system with a bi-directional translator for users.

At the present stage, the text-to-Braille translation chip is a stand-alone component, but it can be integrated in a bigger system. Therefore, future work will be to build a system on chip for multi-functional Braille system including translations. For instance, a real-time speaking function would supply auditory feedback to allow the detection and correction of typing errors. Interfaces for a Braille embosser and a printing machine could also be needed. Therefore, the system should consist of a microcontroller for interface and control, normal and Braille keyboard, the text-Braille translator and a speaker or double talk device, shown in Figure 41. The on-chip hardware-based translator is able to perform Braille translation very fast so that it can release the microcontroller from heavy burden of translations. Meanwhile, the Braille system is also able to access to internet resource, supplying on-line information for the blind. For further improvement, a multi-language-Braille translator should be considered. Look-up tables could be stored in flash memory so that when a particular translation is performed, the microcontroller will load the rule table from the flash memory, and send data to the translator.

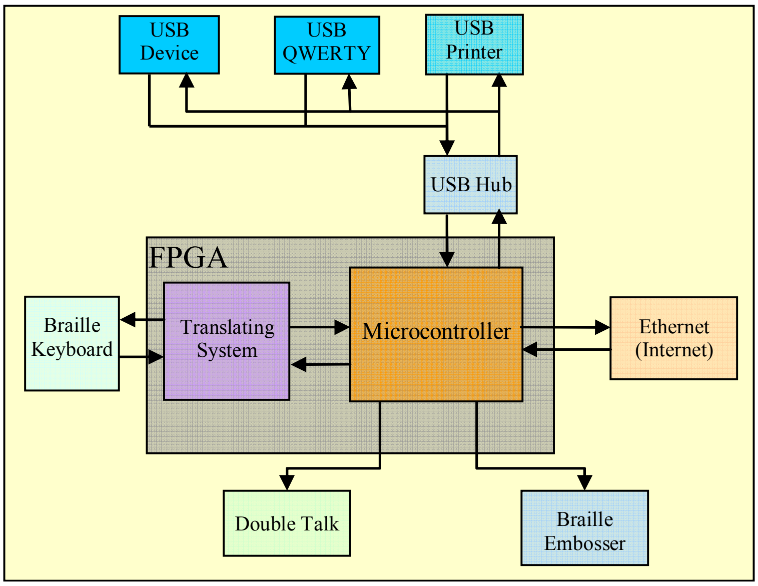


Figure 41. Multi-functional Braille systemsystem

7. References

1. www.who.int
2. www.rnzfb.org.nz
3. www.brailler.com
4. I. Bruce, A. McKennell and E. Walker, 1991 “Blind and Partially-Sighted Adults in Britain: the RNIB Survey”, London, Volume 1, H.M.S.O
5. A. King, “Text and Braille Computer Translation”, 2001, Project Report, Department of Computation, Institute of Sci & Tech, University of Manchester, http://alasdairking.me.uk/brailletrans
6. H. Werner, 1975, “The Historical Development of Automatic Braille Production in Germany”, ACM SIGCAPH Computers and the Physically Handicapped, issue 15, pp. 41-43
7. P. Coleman, 1975, “Some Reflections on the Current State of Automatic Braille Translation”, ACM SIGCAPH Computers and the Physically Handicapped, issue 15, pp. 8-11
8. www.duxburysystems.com
9. www.mountbattenbrailler.com
10. P. Spasov, Microcontroller Technology, the 68HC11 and 68HC12, 5th edition, New Jersey, Pearson/Prentice Hall, 2004, ISBN: 0131247913
11. A. Jonathen, 1972, “Recent Improvement in Braille Transcription”, in Proceedings of the ACM annual Conference, vol. 1, Boston, pp. 208-218

12. M. Truquet, 1976, “Braille Grade II Translation Program”, ACM SIGCAPH Computers and the Physically Handicapped, issue 20, pp. 25-33

13. Personal Report, 1975, “150 Years of Braille”, Applied Ergonomics, Volume 6, issue 4, pp. 237-238

14. British Braille, A Restatement of Standard English Braille Compiled and Authorised by the Braille Authority of the United Kingdom, 2004, ISBN: 0909797901

15. en.wikipedia.org

16. www.dotlessbraille.org

17. J. Viding, 1975, “Computerised Braille Production”, ACM SIGCAPH Computers and the Physically Handicapped, issue 15, pp. 35-40

18. www.iceb.org

19. English Braille American Edition, 2002, Braille Authority of North America, www. Brailleauthority.org

20. P. K. Das, R. Das and A. Chaudhuri, 1995, “A Computerised Braille Transcriptor For the Visually Handicapped”, 14th Conference of the Biomedical Engineering, New Delhi, PP. 3/7-3/8

21. W. Jolley, 2006, “Unified English Braille: A Literacy Bedrock in the Digital Age”, twelfth ICEVI world coference, Kuala Lumpur

22. T. Kam, Synthesis of finite state machines: functional optimization, Boston: Mass Kluwer Acadmic Publishers, 1997, ISBN: 0792398424

23. J. E. Hopcroft, R. Motwani and J. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd edition, Boston, Mass: Pearson Addison-Wesley, 2006, ISBN: 0321455363

24. M. Lawson, Finite Automata, Boca Raton: Chapman & Hall/CRC, 2004, ISBN: 1584882557

25. J. Carroll and L. Darrell, Theory of Automata: With An Introduction of Formal Language, N.J. : Prentice Hall, 1989, ISBN: 0139137084

26. R. Floyd and R. Beigel, The Language of Machines: An Introduction to Computability and Foamal Languages, New York: Computer Science Press, 1994, ISBN: 0716782669

27. L. Peter, Introduction to Formal Languages and Automata, Boston: Jones and Bartlett, 2001, ISBN: 0763714224

28. J. K. Millen, 1970, "Finite-State Syntax-Directed Braille Translation", Technical Report MTR-1829, MITRE Corporation, Bedford, Massachusetts

29. Caracciolo di Forino, 1968, "String processing languages and generalized Markov algorithms: In Symbol manipulation languages and techniques", D. G. Bobrow (Ed.), North-Holland Publ. Co., Amsterdam, The Netherlands, pp. 191-206.

30. K. H. Blasius and Hans-Jurgen Burckert, Deduction System in Artificial Intelligence, Chichester, England: Ellis Horwood, 1989, ISBN: 0745804098

31. H. B. Enderton, A Mathematical Introduction to Logic, New York, Academic Press, 1972, ISBN: 0122384504

32. R. Gildea, 1975, “Automatic Braille Translation in the United States of America”, ACM SIGCAPH Computers and the Physically Handicapped, issue 15, pp.12-13

33. J. E. Sullivan, 1975, “DOTSYS III: A Portable Braille Translator”, in proceedings of the ACM annual conference, issue 15, New York, pp. 14-19

34. W. A. Slaby, 1975, “The Markov System of Production Rules: A Universal Braille Translator”, ACM SIGCAPH computers and the physically handicapped, issue 15, pp. 53-59

35. W. A. Slaby, 1990, “Computerized Braille Translation”, journal of microcomputer appl., vol. 13, issue n2, pp. 107-113

36. H. Kamp, 1975, “Gaining Production Rules For A Markov Braille Translation Algorithm”, ACM SIGCAPH computers and the Physically Handicapped, issue 15, pp. 60-61

37. P. Blenkhorn, 1997, “A System For Converting Print into Braille”, IEEE transactions on rehabilitation engineering, vol. 5, no. 2, pp. 121-129

38. B. A. Forouzan, and R. F. Gilberg, Computer Science: A Structured Programming Approach Using C, Pacific Grove, Canada: Brookes/Cole 2001, ISBN: 0534374824

39. D. W. Croisdale, H. Camp, Computerized Braille Production: Today and Tomorrow, Springer, 1983, ISBN: 0387120572

40. P. Blenkhorn, 1995, “A System For Converting Braille into Print”, IEEE transactions on rehabilitation engineering, vol. 3, no. 2, pp. 215-221

41. I. Murray and A. Pasquale, 2006, “A portable device for the translation of braille to text”, Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility, pp. 231 - 232

42. A. Andras, V. Theresa, E. Gareth and P. Blenkhorn, 2002, “Braille to Text Translation for Hungarian”, in Proceedings of Computer Helping People with Special Needs 8th International Conference, ICCHP 2002, Linz, Austria, pp. 610-617

43. Maxfield Clive, The Design Warrior’s Guide to FPGAs: devices, tools, and flows, Amsterdam, London: Newnes, 2004, ISBN: 9780750676045

44. W. Wolf, FPGA-Based System Design, Upper Saddle River, N. J.: Prentice Hall PTR, 2004, ISBN: 0131424610

45. S. Brown and J. Rose, Architecture of FPGAs and CPLDs: A Tutorial, Department of Electrical and Computer Engineering, University of Toronto

46. M. Richard, ASIC & FPGA verification: a guide to component modelling, Amsterdam: Elsevier/Morgan Kaufmann, 2005, ISBN: 0125105819

47. J. Villasenor and W. H Mangione-Smith, “Configurable Computing”, Scientific American, June, 1997, pp. 66-71

48. J. V. Oldfield and R. C. Dorf, Field-Programmable Gate Arrays: Reconfigurable Logic For Rapid Prototyping and Implementation of Digital Systems, New York: John Wiley & Sons, Inc., 1995, ISBN: 0471556653

49. Q. Lu, F. Yi, H. Yu, and X. Xu, Embedded System Design Based on FPGA, Jiang, S and Wang, J (Eds), China Machine Press, 2005, ISBN: 7111153375

50. www.fpga.com.cn

51. Xilinx Company, Spartan-3 User Guide, electronic documentation, version 1.4, 2005

52. M. B. Gokhale and P. S. Graham, Reconfigurable Computing: Accelerating Computation With Field-Programmable Gate Arrays, Dordrecht, The Netherlands : Springer, 2005, ISBN: 0387261052

53. P. J. Ashenden, The Designer's Guide to VHDL, 2nd edition, San Francisco, Calif.: Morgan Kaufmann, 2002, ISBN: 1558606742

54. D. Pellerin and S. Thibault, Pratical FPGA Programming in C, NJ: Prentice Hall Professional Technical Reference, 2005, ISBN: 0131543180

55. S. Palnitkar, Verilog HDL: A Guide to Digital Design and Synthesis, 2nd edition, NJ: SunSoft Press, 2003, ISBN: 0130449113

56. B. Steven, Fundamentals of Digital Logic With VHDL Design, Boston: McGraw-Hill, 2000, ISBN: 0070125910

57. Z. Mark, Digital System Design with VHDL, New York: Prentice Hall, 2000, ISBN: 0201360632

58. K. C. Chang, Digital Systems Design With VHDL and Synthesis, Los Alamitos, Calif.: IEEE Computer Society, 1999, ISBN: 0769500234

59. B. Cohen, VHDL Coding Style and Methodologies, 2nd edition, Boston: Kluwer Academic Publishers, 1999, ISBN: 0792384741

60. R. Andrew, VHDL for Logic Synthesis, 2nd edition, New York: Wiley Sons, 1998, ISBN: 047198325X

61. M, Morris, Logic and Computer Design Fundamentals, 2nd edition, New Jersey: Prentice Hall, 2000, ISBN: 0130124680

62. B. Zeidman, Designing with FPGAs and CPLDs, CMP books, 2002, ISBN: 1-57820-112-8

63. A. R. Hambley, Electronics: A Top-Down Approach to Computer-Aided Circuit Design, New York: Maxwell Macmillan International, 1994, ISBN: 0023493356

64. W. Walter and A. Singh, The 8088 and 8086 microprocessors: programming, interfacing, software, hardware, and applications, New Jersey: Prentice Hall, 1991, ISBN: 0132483378

65. X. Zhang, C. Ortega-Sanchez, and I. Murray, "Text-to-Braille Translator in a Chip", International Conference on Electrical and Computer Engineering, 2006, Dhaka, Bangladesh, pp. 530-533

66. Xilinx ISE 8.2 Quick Start Tutorial

67. A. Mignotte, E. Villar, and L. Horobin, System On Chip Design Languages, Springer 2002, ISBN: 1402070462

68. X. Zhang, C. Ortega-Sanchez, and I. Murray, "Hardware-Based Text-to-Braille Translator", the 8th International ACM SIGACCESS Conference on Computers & Accessibility, Portland, Oregon, USA, 2006, pp. 229-230

69. Memec Inc. Virtex-4(TM) FX12 LC Development Board User's Guide, electronic documentation, version 1.0, 2005

70. Xilinx Company, Virtex-4 Family Overview, electronic documentation, version 1.5, 2006

71. X. Zhang, C. Ortega-Sanchez, and I. Murray, "A System For Fast Text to Braille Translation Based on FPGAs", 3rd Southern Conference on Programmable Logic (SPL), 2007, Mar del Plata, Argentina, pp. 125-130

72. www.perkins.org

73. Technical Considerations in Production Selection, codi.buffalo.edu

74. B. R. Sanchez and G. G. Garcia, 2006, "Keyboard of Automatic Teller in Braille Code", 16th International Conference of Electronics, Communication and Computers, pp. 39-46

75. J. Spragg, 1984, "Interfacing a Perkins Brailler to a BBC micro," Microprocessors Microsyst, vol. 8, pp. 524-527

76. D.G. Evans, S. Pettitt, P. Blenkhorn, 2002, "A Modified Perkins Brailler for Text Entry into Windows Applications", Neural Systems and Rehabilitation Engineering, IEEE Transactions on Rehabilitation Engineering, Volume: 10, Issue: 3, pp. 204-206

77. X. Zhang, C. Ortega-Sanchez, and I. Murray, 2007, "A Hardware Based Braille Note Taker", 3rd Southern Conference on Programmable Logic (SPL), 2007, Mar del Plata, Argentina, pp. 131-136

78. M. Jones, A Practical Introduction to Electronic Circuits, 3rd edition, Cambridge; New York: Cambridge University Press, 1995, ISBN: 0521472865

79. Memec Inc. Spartan-3 LC Development Board User's Guide, electronic documentation, version 2.0, 2004

80. Virtex-4 Configuration Guide, Xilinx Company, www.xilinx.com

"Every reasonable effort has been made to acknowledge the owners of copyright material. I would be pleased to hear from any copyright owner who has been omitted or incorrectly acknowledged."

8. Appendix

Appendix I: Braille ASCII Set

Table Origination: en.wikipedia.org

Binary	Dec	Hex	Glyph	Braille Dots
0010 0000	32	20	(space)	
0010 0001	33	21	!	2-3-4-6
0010 0010	34	22	"	5
0010 0011	35	23	#	3-4-5-6
0010 0100	36	24	$	1-2-4-6
0010 0101	37	25	%	1-4-6
0010 0110	38	26	&	1-2-3-4-6
0010 0111	39	27	'	3
0010 1000	40	28	(	1-2-3-5-6
0010 1001	41	29	)	2-3-4-5-6
0010 1010	42	2A	*	1-6
0010 1011	43	2B	+	3-4-6
0010 1100	44	2C	,	6
0010 1101	45	2D	-	3-6
0010 1110	46	2E	.	4-6
0010 1111	47	2F	/	3-4
0011 0000	48	30	0	3-5-6
0011 0001	49	31	1	2
0011 0010	50	32	2	2-3
0011 0011	51	33	3	2-5
0011 0100	52	34	4	2-5-6
0011 0101	53	35	5	2-6
0011 0110	54	36	6	2-3-5
0011 0111	55	37	7	2-3-5-6
0011 1000	56	38	8	2-3-6
0011 1001	57	39	9	3-5
0011 1010	58	3A	:	1-5-6
0011 1011	59	3B	;	5-6
0011 1100	60	3C	<	1-2-6
0011 1101	61	3D	=	1-2-3-4-5-6
0011 1110	62	3E	>	3-4-5
0011 1111	63	3F	?	1-4-5-6
0100 0000	64	40	@	4
0100 0001	65	41	A	1
0100 0010	66	42	B	1-2
0100 0011	67	43	C	1-4

0100 0100	68	44	D	1-4-5
0100 0101	69	45	E	1-5
0100 0110	70	46	F	1-2-4
0100 0111	71	47	G	1-2-4-5
0100 1000	72	48	H	1-2-5
0100 1001	73	49	I	2-4
0100 1010	74	4A	J	2-4-5
0100 1011	75	4B	K	1-3
0100 1100	76	4C	L	1-2-3
0100 1101	77	4D	M	1-3-4
0100 1110	78	4E	N	1-3-4-5
0100 1111	79	4F	O	1-3-5
0101 0000	80	50	P	1-2-3-4
0101 0001	81	51	Q	1-2-3-4-5
0101 0010	82	52	R	1-2-3-5
0101 0011	83	53	S	2-3-4
0101 0100	84	54	T	2-3-4-5
0101 0101	85	55	U	1-3-6
0101 0110	86	56	V	1-2-3-6
0101 0111	87	57	W	2-4-5-6
0101 1000	88	58	X	1-3-4-6
0101 1001	89	59	Y	1-3-4-5-6
0101 1010	90	5A	Z	1-3-5-6
0101 1011	91	5B	[	2-4-6
0101 1100	92	5C	\	1-2-5-6
0101 1101	93	5D	]	1-2-4-5-6
0101 1110	94	5E	^	4-5
0101 1111	95	5F	_	4-5-6

Appendix II: Rule Table for Text-to-Braille Translation [37]

Rule format: input class <tab>left context [focus] right context =output<tab> next state

2	['EN] = 'EN	-	2	[CANNOT]=_C	-	1	[O].~=O	-
3	[ ]=	1	2	~[CAN]~=C	-	1	.[O]=O	-
4	[ ]=	2	2	~[CATI]ON=CATI	-	1	[O].!=O	-
1	[--]=--	-	2	~[CENT]=C5T	-	1	~[O]~=;O	-
1	[-]=--	-	2	[CCH]=C*	-	1	[O]=O	-
1	[ ]=	-	2	![CC]!=3	-	5	[O]=O	-
5	[ ]=	-	1	[C].~=C	-	2	~[P];#=;P	3
1	[!]=6	-	1	.[C]=C	-	2	#[P]=;P	3
5	[!]=!	-	1	[C].!=C	-	1	~[P];#=;P	4
1	["] ="1	-	1	~[C]~=;C	-	1	#[P]=;P	4
1	["...]=8 "'	-	1	[C]=C	-	2	~[PH]ONEY=PH	-
1	["=]=8;	-	5	[C]=C	-	2	[PHONES]S=PH"O	-
1	["]^=8	-	2	~[D];#=;D	3	2	[PHONETI]=PHONETI	-
1	["]^^=8	-	2	#[D]=;D	3	2	[PHONE]~=PH"O	-
1	["] (=8	-	1	~[D];#=;D	4	2	~[PAR]TH=P>	-
1	["]~=0	-	1	#[D]=;D	4	2	[PART]="P	-
1	["]=8	-	2	[D'YOU]=D'Y\	-	2	[PAID]=PD	-
5	["]="	-	2	~[DAFT]ER=DAFT	-	2	[PAINS]TAK=PA9S	-
1	[#]#=#	-	2	[DAY]="D	-	2	[PAGODA]=PAGODA	-
1	[#]=#	-	2	~[DO']=DO@	-	2	~[PEOPLE]~=P	-
5	[#]=#	-	2	~[DO]~=D	-	2	~[PERHAPS]=P]H	-
1	[$G1]=	2	2	~[DIS]HEA=4	-	2	[PERCEIVE]=P]CV	-
1	[$G2]=	1	2	~[DIS]HA=4	-	2	[PERCEIVIN]G=P]CV	-
1	[$#]=	-	2	~[DIS]HO=4	-	2	[PERSE]VER=P]SE	-
1	[$+]=;6	-	2	~[DISH]=DI%	-	2	[PEACH]=PR1*	-
1	[$-]=;-	-	2	~[DISK]S=DISK	-	2	[PRED]AC=PR$A	-
1	[$X]=;8	-	2	~[DIS]~=DISK	-	2	[PREDA]TOR=PR$A	-
1	[$D]=;4	-	2	~[DISC]S~=DISC	-	2	[PRED]ECES=PR$	-
1	[$=]=;7	-	2	~[DISC]~=DISC	-	2	[PREDI]L=PR$I	-
1	[$/]=	-	2	[DISPIRIT]=DI_S	-	2	[PREDI]C=PR$I	-
1	[$$]=$$	-	2	~[DI]SULPH=DI	-	2	[PRENT]ICE=PR5T	-
1	[$]#=4	-	2	~[DIS]!=4	-	2	[PRERO]G=PR]O	-
5	[$]#=$	-	2	~[DINGH]=D9<	-	2	~[PRE]=PRE	-
1	[$]!=@4	-	2	![DDAU]GHTER=DDAU	-	2	[POST]H=PO/	-
1	[%]=3P	-	2	[DDAY]=D"D	-	1	[P].~=P	-
5	[%]=%	-	2	![DD]!=4	-	1	.[P]=P	-
2	[&ING]=&+	-	2	[DECEIVE]=DCV	-	1	[P].!=P	-
1	[&]=&	-	2	[DECEIVING]=DCVG	-	1	~[P]~=;P	-
5	[&]=&	-	2	[DECLARING]=DCLG	-	1	[P]=P	-
2	['CAUSE]='CAUSE	-	2	[DECLARE]=DCL	-	5	[P]=P	-
2	['DO]='DO	-	2	~[DE]NAT=DE	-	2	~[Q];#=;Q	3
1	!['D]~='D	-	2	~[DESH]ABILLE=DESH	-	2	#[Q]=;Q	3
2	['FLU]='FLU	-	2	[DEAW]=DEAW	-	1	~[Q];#=;Q	4
2	['ER]=']	-	2	~[DEAC]T=DEAC	-	1	#[Q]=;Q	4
2	['EN]='5	-	1	[D].~=D	-	1	[Q].!=Q	-
2	~['IN]=,8IN	-	1	.[D]=D	-	1	~[Q]~=;Q	-
2	['IN]='IN	-	1	[D].!=D	-	1	[Q]=Q	-
2	['YOU]='Y\	-	1	~[D]~=;D	-	5	[Q]=Q	-
1	!['C]~='C	-	1	[D]=D	-	1	~'[RD]~=4RD	-
1	!['M]~='M	-	5	[D]=D	-	1	#[RD]~=RD	-
2	['NEATH]='N1	-	2	~[E];#=;E	3	1	.[RD]~=RD	-
1	!['N]-='N	-	2	#[E]=;E	3	2	~[R];#=;R	3
2	['TIS]='TIS	-	1	~[E];#=;E	4	2	#[R]=;R	3
2	['TWAS]='TWAS	-	1	#[E]=;E	4	1	~[R];#=;R	4
1	!['T]~='T	-	2	|[ENOUGH] =5	-	1	#[R]=;R	4
1	!['S]='S	-	2	![EDISH]=EDI%	-	2	[RIGHT]="R	-
1	#['S]='S	-	2	![ED]OOM=ED	-	2	~[RATHER]~=R	-
1	!["]=0'	-	2	![ED]OM=ED	-	2	[RAFT]ER=RAFT	-
1	~['0]=#'J	-	2	![ED]OVE=ED	-	2	[RARED]~=RAR$	-
1	~['1]=#'A	-	2	![ED]OWN=ED	-	2	~[RANS]OME=RANS	-
1	~['2]=#'B	-	2	![ED]EEP=ED	-	2	[RAR]ENAL=RAR	-
1	~['3]=#'C	-	2	![ED]REAM=ED	-	2	~[REA]B=REA	-
1	~['4]=#'D	-	2	![ED]ROP=ED	-	2	[REACHING]=R1*+	-
1	~['5]=#'E	-	2	![ED]RUM=ED	-	2	~[REACH]I=REA*	-
1	~['6]=#'F	-	2	![EDD]FO=E4	-	2	~[REACH]=R1*	-
1	~['7]=#'G	-	2	![EDAL]E=EDAL	-	2	~[RE]AC=RE	-
1	~['8]=#'H	-	2	[ED]=$	-	2	~[READ]AP=READ	-
1	~['9]=#'I	-	2	[EDREAG]H=ER1<	-	2	~[REA]DD=REA	-
1	~[']~='	-	2	[EROO]M=EROO	-	2	~[READ]J=READ	-
1	![']~='	-	2	[ER]=]	-	2	~[READ]M=READ	-
1	~[']=,8	-	2	[ELECTRO]=ELECTRO	-	2	~[READ]O=READ	-
1	[']='	-	2	[E]NAME=E	-	2	~[READ]V=READ	-
5	[']='	-	2	[ENCED]=5C$	-	2	~[REA]F=REA	-
1	[(]=7	-	2	[ENCEA]=5C1	-	2	~[REA]G=REA	-
5	[(]=(	-	2	[ENCER]=5C]	-	2	~[REAL]IG=REAL	-
1	[)]=7	-	2	![ENCE]=;E	-	2	~[REAL]IN=REAL	-
5	[)]=)	-	2	[ENESS]=E;S	-	2	~[RE]ALL=RE	-
1	[*]#=;8	-	2	![ENOO]K=ENOO	-	2	~[REAN]=REAN	-
1	[*]=99	-	2	~[ENOUGH'S]=5'S	-	2	~[REAP]P=REAP	-
5	[*]=*	-	2	~[EN]~=EN	-	2	~[REAS]C=REAS	-
1	[+]#=;6	-	2	[EN]=5	-	2	~[REAS]S=REAS	-
1	[+]="	-	2	![EAR]=E>	-	2	~[REAT]T=REAT	-
5	[+]=+	-	2	![EALLY]=E,Y	-	2	[REAW]AKE=REAW	-
1	#[,0]='J	-	2	![EALO]GY=EALO	-	2	~[REDEE]M=R$EE	-

1	#[,1]='A	-
1	#[,2]='B	-
1	#[,3]='c	-
1	#[,4]='D	-
1	#[,5]='E	-
1	#[,6]='F	-
1	#[,7]='G	-
1	#[,8]='H	-
1	#[,9]='I	-
1	[,]#='	-
1	[,]=1	-
5	[,]=,	-
2	[-T]0~=-T	-
2	[-ING]~=-+	-
2	[-IN]~=-9	-
2	[-C]OM=-C	-
2	[-BY]=-BY	-
2	[--INTO|]=-}96	-
2	[--IN]=--9	-
2	[--C]OM=--C	-
1	![----]=----	-
1	[--]~=--	-
1	[--]=-	-
1	#[-0]=-J	-
1	#[-1]=-A	-
1	#[-2]=-B	-
1	#[-3]=-C	-
1	#[-4]=-D	-
1	#[-5]=-E	-
1	#[-6]=-F	-
1	#[-7]=-G	-
1	#[-8]=-H	-
1	#[-9]=-I	-
1	[-]=-	-
5	[-]=-	-
1	[.] =4	-
1	[....]="'4	-
1	[...']="'0'	-
1	[..."]="'0	-
1	[...]="'	-
1	[.]+.=	?
1	.+[.]=	-
1	#[.]## A.M.=	-
1	#[.]## P.M.=	-
1	#[.0]=1J	-
1	#[.1]=1A	-
1	#[.2]=1B	-
1	#[.3]=1C	-
1	#[.4]=1D	-
1	#[.5]=1E	-
1	#[.6]=1F	-
1	#[.7]=1G	-
1	#[.7]=1H	-
1	#[.8]=1I	-
1	[.0]=#1J	-
1	[.1]=#1A	-
1	[.2]=#1B	-
1	[.3]=#1C	-
1	[.4]=#1D	-
1	[.5]=#1E	-
1	[.6]=#1F	-
1	[.7]=#1G	-
1	[.8]=#1H	-
1	[.9]=#1I	-
1	[.]#=1	-
1	[.]~=4	-
1	[.]=4	-
5	[.]=.	-
2	[/SUB]=*	-
2	[/SUP]=+	-
1	[/]+/=	-
1	/+[/]=	-
1	~[/]#=;4	-
1	[/]=/	-
5	[/]=/	-
1	#[0]=J	-
1	[0]=#J	-
5	[0]=0	-
2	#[1ST]=A/	-
2	[1ST]=#A/	-
1	#[1]=A	-
1	[1]=#A	-
5	[1]=1	-
1	#[2]=B	-
1	[2]=#B	-
5	[2]=2	-
1	#[3]=C	-
1	[3]=#C	-
5	[3]=3	-
1	#[4]=D	-
1	[4]=#D	-
5	[4]=4	-
1	#[5]=E	-
2	![EADE]~=EADE	-
2	![EADD]=1DD	-
2	![EAX]=EAX	-
2	![EAPP]=EAPP	-
2	![EANCE]=E.E	-
2	![EAND]=E&	-
2	![EATION]=E,N	-
2	![E]AWAY=E	-
2	![EA]BLE=AE	-
2	![EA]!=1	-
2	[EEVER]=EEV]	-
2	~[EVERY]~=E	-
2	~[EVERTO]N="ETO	-
2	~[EVERT]=EV]T	-
2	[EVERD]I~=EV]D	-
2	[EVER]="E	-
2	[EITHER]=EI	-
2	[ETHER]E=E!R	-
1	[E].~=E	-
1	.[E]=E	-
1	[E].!=E	-
1	~[E]~=;E	-
1	[E]=E	-
5	[E]=E	-
2	~[F];#=;F	3
2	#[F]=;F	3
1	~[F];#=;F	4
1	#[F]=;F	4
2	~[FOR]THE~==	-
2	~[FOR]A~==	-
2	~[FOREVER]=="E	-
2	~[FOR]ENS==	-
2	~[FORE]==E	-
2	[FOR]==	-
2	[FRUI]T=FRUI	-
2	[FRIEN]DE=FRI5	-
2	[FRIEN]DI=FRI5	-
2	[FRIEND]=FR	-
2	~[FROM]~=F	-
2	[FIRST]=F/	-
2	~[FIAN]C!=FIAN	-
2	[FLEAR]IDD=FL1R	-
2	![FULLE]=;LLE	-
2	![FULLY]=;LLY	-
2	![FULL]=FULL	-
2	![FUL]=;L	-
2	[FFOR]=F=	-
2	![FF]!=6	-
2	[FATHER]="F	-
2	~[FAERY]=FA]Y	-
1	[F].~=F	-
1	.[F]=F	-
1	[F].!=F	-
1	~[F]~=;F	-
1	[F]=F	-
5	[F]=F	-
2	~[G];#=;G	3
2	#[G]=;G	3
1	~[G];#=;G	4
1	#[G]=;G	4
2	![GHAI]=GHAI	-
2	![GHEAD]=GH1D	-
2	![GHEAP]=GH1P	-
2	![GHIL]=GHIL	-
2	![GHOL]E=GHOL	-
2	![GHOR]N=GHOR	-
2	![GHOUS]E=GH\S	-
2	![GHUN]T=GHUN	-
2	[GH]=<	-
2	[GOOD]=GD	-
2	[GOVERN]ESS=GOV]N	-
2	~[GO]~=G	-
2	![GG]!=7	-
2	[GREAT]=GRT	-
1	[G].~=G	-
1	.[G]=G	-
1	[G].!=G	-
1	~[G]~=;G	-
1	[G]=G	-
5	[G]=G	-
2	~[H];#=;H	3
2	#[H]=;H	3
1	~[H];#=;H	4
1	#[H]=;H	4
2	|[HIS] =8	-
2	[HADD]!=HA4	-
2	~[HADE]=HADE	-
2	~[HADR]IAN]=HADR	-
2	~[HAD]=_H	-
2	~[HAVE]~=H	-
2	~[HIMSELF]=HMF	-
2	~[HIM]~=HM	-
2	[HEDGE]ROW=H$GE	-
2	~[RED]EMPT=R$	-
2	~[RED]E=RED	-
2	~[RED]I=RED	-
2	~[REDOUB]T=R$\B	-
2	~[REDOUND]=R$.D	-
2	~[RE]DO=RE	-
2	~[REDR]AW=REDR	-
2	~[REDU]C=R$U	-
2	~[REDU]ND=R$U	-
2	~[REDU]=REDU	-
2	~[RE]NAM=RE	-
2	~[RENA]V=RENA	-
2	~[RENO]M=RENO	-
2	~[RENU]M=RENU	-
2	~[REREDO]S=R]$O	-
2	~[RE]R=RE	-
2	[REVER]EN=R"E	-
2	[REVER]IE=R"E	-
2	~[REVER]=REV]	-
2	[REJOICE]=RJC	-
2	[REJOICING]=RJCG	-
2	[RECEIVE]=RCV	-
2	[RECEIVING]=RCVG	-
1	[R].~=R	-
1	.[R]=R	-
1	[R].!=R	-
1	~[R]~=;R	-
1	[R]=R	-
5	[R]=R	-
1	#[S]~='S	-
2	~[S];#=;S	3
2	#[S]=;S	3
1	~[S];#=;S	4
1	#[S]=;S	4
2	~[STILL]~=/	-
2	![STID]E=STID	-
2	![STION]=S;N	-
2	![STIME]=S"T	-
2	![STHEAD]=/H1D	-
2	![ST]HOOD=/	-
2	![S]TH=S	-
2	![ST]OWN=ST	-
1	~'[ST]~=4/	-
2	~[ST].~=ST	-
2	[ST]=/	-
2	~[SHALL]~=%	-
2	![SHART]=SH>T	-
2	![SHAW]K=SHAW	-
2	[SHOULD]ER=%\LD	-
2	[SHOULD]=%D	-
2	![SHOUS]E=SH\S	-
2	![SHOO]D=SHOO	-
2	![SHOR]N=SHOR	-
2	![SHOR]SE=SHOR	-
2	![SHOUND]=SH.D	-
2	![SHIL]L=SHIL	-
2	![SHEAR]T=SHE>	-
2	![SHEAD]=SH1D	-
2	![SHUN]D~=SHUN	-
2	~[SH]'=%	-
2	~[SH]~=SH	-
2	[SH]=%	-
2	![SION]=.N	-
2	![SINGH]=S9<	-
2	[SAID]=SD	-
2	![SOFAR]=SOF>	-
2	![SOMED]~=SOM$	-
2	![SOME]TRY=SOME	-
2	![SOME]TRIC=SOME	-
2	![SOME]TER=SOME	-
2	[SOMER]!=SOM]	-
2	[SOME]="S	-
2	~[SO]~=S	-
2	[SEVERED]=S"E$	-
2	[SEVER]E=SEV]	-
2	[SEVER]ITY=SEV]	-
2	[SED]ATIV=S$	-
2	[SPHER]=SPH]	-
2	[SPIRIT]=_S	-
2	~[SUB]=SUB	-
2	[SUCH]=S*	-
2	~[SSH]~=S%	-
2	[SS]H=SS	-
2	[SWED]ISH=SW$	-
2	~[SWOR]D=SWOR	-
2	[SQUA]LLY=SQUA	-
1	[S"]~=SO'	-
1	[S']~=S'	-
1	[S].~=S	-
1	.[S]=S	-
1	[S].!=S	-
1	~[S]~=;S	-
1	[S]=S	-

1	[5]=#E	-
5	[5]=5	-
1	#[6]=F	-
1	[6]=#F	-
5	[6]=6	-
1	#[7]=G	-
1	[7]=#G	-
5	[7]=7	-
1	#[8]=H	-
1	[8]=#H	-
5	[8]=8	-
1	#[9]=I	-
1	[9]=#I	-
5	[9]=9	-
1	[:]#=	-
1	[:]=3	-
5	[:]=:	-
1	[;]=2	-
5	[;]=5	-
1	[<]=8	-
5	[<]=<	-
2	~[=T-SHI]IRT=;T-SHI	-
2	[=]!=;	3
1	[=]!=;	4
1	[=]=;8	-
5	[=]==	-
1	[>]=0	-
1	[>]=>	-
1	[?]=8	-
5	[?]=?	-
2	[@EN]=@EN	-
2	[@ER]=@ER	-
2	[@ED]=@ED	-
2	[@O]NG=@O	-
2	[@AR]=@AR	-
1	[@]=@	-
5	[@]=@	-
2	~[A];#=;A	-
2	#[A]=;A	-
1	~[A];#=;A	-
1	#[A]=;A	-
2	[ARIGHT]=A"R	-
2	[AR]=>	-
2	~[AND]THE~=&	-
2	~[AND]A~=&	-
2	~[AND]OF~=&	-
2	~[AND]WITH~=&	-
2	~[AND]FOR~=&	-
2	[AND]=&	-
2	~[ANTEA]TER=ANT1	-
2	[ANTEN]NA=ANT5	-
2	[ANTER]IOR=ANT]	-
2	~[ANTE]=ANTE	-
2	[ANTIN]OM=ANT9	-
2	~[ANTI]=ANTI	-
2	![ANCE]=.E	-
2	[ANEMONE]=ANEMONE	-
2	![ATION]=,N	-
2	~[AS]~=Z	-
2	[ABOUT]=AB	-
2	[ABOVE]=ABV	-
2	[AGAIN]=AG	-
2	[AFTERNOON]=AFN	-
2	[AFTERWARD]=AFW	-
2	~[AFTER]E=AFT	-
2	~[AFTER]I=AFT	-
2	[AFTER]=AF	-
2	![ALLY]=,Y	-
2	~[ALWAYS]~=ALW	-
2	~[ALSO]~=AL	-
2	~[ALMOST]~=ALM	-
2	~[ALREADY]~=ALR	-
2	~[ALTHOUGH]~=AL?	-
2	~[ALTOGETHER]=ALT	-
2	~[ACROSS]~=ACR	-
2	~[ACCORDING]=AC	-
2	[AUND]ER=AUND	-
2	[AINES]S=A9ES	-
2	![AED]~=A$	-
2	[AE]D=AE	-
2	[AE]A=AE	-
2	[AERO]=A]O	-
2	![AER]=AER	-
2	~[AENE]AS=AENE	-
2	[AE]N=AE	-
1	[A]=A	-
5	[A]=A	-
2	~[B];#=;B	3
2	#[B]=;B	3
1	~[B];#=;B	4
1	#[B]=;B	4
2	[BRO']=BRO'	-
2	[HER]ESY=H]	-
2	[HERI]SI=H]E	-
2	[HERE]TI=H]E	-
2	[HERE]R=H]]	-
2	[HER]EN=H]	-
2	[HER]ED=H]	-
2	[HER]EF=H]	-
2	[HERE]="H	-
2	~[HERSELF]=H]F	-
2	[HYDRO]=HYDRO	-
2	~[HM]~=H'M	-
1	[H].~=H	-
1	.[H]=H	-
1	[H].!=H	-
1	~[H]~=;H	-
1	[H]=H	-
5	[H]=H	-
2	~[I];#=;I	3
2	#[I]=;I	3
1	~[I];#=;I	4
1	#[I]=;I	4
2	|[IN] =9	-
2	~[INTO] AND =9TO	-
2	~[INTO] AT =9TO	-
2	~[INTO] BUT =9TO	-
2	~[INTO] IF =9TO	-
2	~[INTO] IN =9TO	-
2	~[INTO] IS =9TO	-
2	~[INTO] WAS =9TO	-
2	~[INTO] WHEN =9TO	-
2	~[INTO] FOR =9TO	-
2	~[INTO] OF =9TO	-
2	~[INTO] OR =9TO	-
2	~[INTO] TO =9TO	-
2	~[INTO HI]S =96HI	-
2	~[INTO ENOU]GH=965\	-
2	~[INTO]_=96	-
2	~[INTO]!=96	-
2	~[INTO]#=96	-
2	[INDIA]RUB=9DIA	-
2	![INGRA]=9GRA	-
2	![ING]=+	-
2	![INESS]=I;S	-
2	~[IN]-=9	-
2	~[IN] =IN	-
2	~[IN]~=IN	-
2	![IN]=9	-
2	[IN]!=9	-
2	![ITY]=;Y	-
2	~[ITSELF]~=XF	-
2	~[ITS]~=XS	-
2	~[IT]~=X	-
2	[IRRE]VERS=IRRE	-
2	[IEVER]=IEV]	-
2	[IETN]AMESE=IETN	-
2	~[IMMEDIATE]=IMM	-
2	[IO]NE=IO	-
2	~[ISOM]ER=ISOM	-
1	[IV]~=;IV	-
1	[II]~=;II	-
1	[III]~=;III	-
1	[I]=I	-
5	[I]=I	-
2	~[J];#=;J	3
2	#[J]=;J	3
1	~[J];#=;J	4
1	#[J]=;J	4
2	~[JUST]~=J	-
1	[J].~=J	-
1	.[J]=J	-
1	[J].!=J	-
1	~[J]~=;J	-
1	[J]=J	-
5	[J]=J	-
2	~[K];#=;K	3
2	#[K]=;K	3
1	~[K];#=;K	4
1	#[K]=;K	4
2	~[KNOWLEDGE]~=K	-
2	[KNOW]="K	-
2	~[KILO]=KILO	-
1	[K].~=K	-
1	.[K]=K	-
1	[K].!=K	-
1	~[K]~=;K	-
1	[K]=K	-
5	[K]=K	-
2	~[L];#=;L	3
2	#[L]=;L	3
1	~[L];#=;L	4
1	#[L]=;L	4
2	~[LATI]MER=LATE	-
2	[LAERT]ES=LA]T	-
5	[S]=S	-
1	~'[TH]~=4?	-
1	#[TH]~=?	-
1	.[TH]~=?	-
2	~[T];#=;T	3
2	#[T]=;T	3
1	~[T];#=;T	4
1	#[T]=;T	4
2	![THAND]=TH&	-
2	![THART]=TH>T	-
2	~[THAT]~=T	-
2	[THERER]=!R]	-
2	[THERED]=!R$	-
2	[THERE]SA=!RE	-
2	[THERE]TT=!RE	-
2	[THEREEN]=!RE5	-
2	[THERE]="!	-
2	![THERD]=TH]D	-
2	~[THEIR]=_!	-
2	[THESE]~=^!	-
2	~[THEMSELVES]=!MVS	-
2	[THENCE]=?;E	-
2	[THEND]=?5D	-
2	[THEAST]=?1/	-
2	![THEAD]=TH1D	-
2	![THEART]T=THE>	-
2	[THE]=!	-
2	~[THIS]~=?	-
2	![THIL]L=THIL	-
2	[THRO']=?RO'	-
2	[THROUGH]="?	-
2	~[THOSE]=^?	-
2	![THOO]D=THOO	-
2	![THOO]K=THOO	-
2	![THOR]SE=THOR	-
2	![THOUS]E=TH\S	-
2	![THOL]E=THOL	-
2	![THOL]D=THOL	-
2	~[THYSELF]=?YF	-
2	[TH]=?	-
2	~[TO] AND~=TO	-
2	~[TO] AT =TO	-
2	~[TO BE]~=6BE	-
2	~[TO] BUT=TO	-
2	~[TO BY]!=TO 0	-
2	~[TO] IF =TO	-
2	~[TO] IN =TO	-
2	~[TO] IS =TO	-
2	~[TO] WAS =TO	-
2	~[TO] WERE =TO	-
2	~[TO] WHERE =TO	-
2	~[TO] WITH =TO	-
2	~[TO] FOR =TO	-
2	~[TO] OF =TO	-
2	~[TO] OR =TO	-
2	~[TO] TO =TO	-
2	~[TO HIS]=6HIS	-
2	~[TO ENOUGH] =65\<	-
2	~[TO _BE]=6.BE	-
2	~[TO _]=6.	-
2	~[TO =]=6	-
2	~[TO]!=6	-
2	~[TO]#=6	-
2	~[TOGETHER]=TGR	-
2	~[TODAY]=TD	-
2	~[TOMORROW]=TM	-
2	~[TONIGHT]=TN	-
2	~[TO-DAY]=TD	-
2	~[TO-MORROW]=TM	-
2	~[TO-NIGHT]=TN	-
2	~[TORE]ADOR=TORE	-
2	![TION]=;N	-
2	![TI]MEN=TI	-
2	![TIME]TER=TIME	-
2	[TIME]="T	-
2	[TEAROOM]=T1ROOM	-
2	[TWOULD]=TWD	-
2	~[TWO]=TWO	-
2	[TLDE]DG=TLED	-
2	![TLE]D!=TLE	-
2	![TTLE]N=TTLE	-
1	[T].~=T	-
1	.[T]=T	-
1	[T].!=T	-
1	~[T]~=;T	-
1	[T]=T	-
5	[T]=T	-
2	~[U];#=;U	3
2	#[U]=;U	3
1	~[U];#=;U	4
1	#[U]=;U	4
2	~[UNDER]I=UND]	-
2	~[UNDER]O=UND]	-

2	~[BUT]~=B	-	2	![LESS]=.S	-	2	~[UNFUL]F=UNFUL	-
2	[BBLE]=B#	-	2	[LETTER]=LR	-	2	[UNDER]="U	-
2	![BB]!=2	-	2	~[LIKE]~=L	-	2	~[UNEAS]=UN1S	-
2	|[BE] =2	-	2	[LITTLE]=LL	-	2	~[UNEAR]=UNE>	-
2	~[BEATI]F=2ATI	-	2	[LORD]="L	-	2	~[UNLESS]~=UN.S	-
2	~[BEATI]T=2ATI	-	2	[LAHAD]=LA_H	-	2	~[UNITY]=UN;Y	-
2	~[B]EA=B	-	2	~[LLAN]D=LLAN	-	2	~[UN]=UN	-
2	~[BECAUSE]=2C	-	1	[L].~=L	-	2	[USEA]GE=USEA	-
2	~[BECK]=BECK	-	1	.[L]=L	-	2	~[US]~=U	-
2	~[BEC]=2C	-	1	[L].!=L	-	2	[UPON]=^U	-
2	~[BED]A=2D	-	1	~[L]~=;L	-	1	[U].~=U	-
2	~[BED]E=2D	-	1	[L]=L	-	1	.[U]=U	-
2	~[BED]I=2D	-	5	[L]=L	-	1	[U].!=U	-
2	~[BEDRA]G=2DRA	-	1	~[M]C`~=;M	-	1	~[U]~=;U	-
2	~[BED]=B$	-	2	~[M];#=;M	3	1	[U]=U	-
2	~[BEET]HOVEN=BEET	-	2	#[M]=;M	3	5	[U]=U	-
2	~[BE]E=BE	-	1	~[M];#=;M	4	2	~[V];#=;V	3
2	~[BEFORE]=2F	-	1	#[M]=;M	4	2	#[V]=;V	3
2	~[BE]F=2	-	2	![MENT]=;T	-	1	~[V];#=;V	4
2	~[BEG]A=2G	-	2	~[MAHA]=MAHA	-	1	#[V]=;V	4
2	~[BEG]E=2G	-	2	[MANY]=_M	-	2	~[VERY]~=V	-
2	~[BEG]I=2G	-	2	[MONTRE]AL=MONTRE	-	2	~[VICEN]=VIC5	-
2	~[BEG]O=2G	-	2	~[MORE]'N=MORE	-	2	~[VICE]=VICE	-
2	~[BEG]R=2G	-	2	~[MORE]~=M	-	1	[V].~=V	-
2	~[BEG]U=2G	-	2	~[MORT]IMER=MORT	-	1	[V].TH=;V	-
2	~[BE]G=BE	-	2	[MOTHEA]TEN=MO?1	-	1	.[V]=V	-
2	~[BEHIND]=2H	-	2	[MOTHER]APY=MO!R	-	1	[V].!=V	-
2	~[BEH]=2H	-	2	[MOTHER]="M	-	1	~[V]~=;V	-
2	~[BEING]=2+	-	2	~[MIS]TI=MIS	-	1	~[VI]~=;VI	-
2	~[BEIN']=2IN	-	2	~[MIST]RIAL=MIST	-	1	~[VII]~=;VI	-
2	~[BE]I=BE	-	2	~[MIST]REA=MIST	-	1	~[VIII]~=;VI	-
2	~[BEJ]=2J	-	2	~[MIST]RU=MIST	-	1	[V]=V	-
2	~[BEL]A=2L	-	2	~[MIST]RANS=MIST	-	5	[V]=V	-
2	~[BEL]E=2L	-	2	~[MIS]TH=MIS	-	2	~[W];#=;W	3
2	~[BE]LE=2	-	2	[MIST]=MI/	-	2	#[W]=;W	3
2	~[BELOW]=2L	-	2	~[MIS]=MIS	-	1	~[W];#=;W	4
2	~[BEL]O=2L	-	2	[MICRO]=MICRO	-	1	#[W]=;W	4
2	~[BEL]Y=2L	-	2	[MUCH]=M*	-	2	|[WAS]-=WAS	-
2	~[BE]L=BE	-	2	[MUST]AFA=MU/	-	2	|[WAS] =0	-
2	~[BE]M=2	-	2	~[MUSTA]NG=MU/A	-	2	|[WERE]-=WERE	-
2	~[BENEATH]=2N	-	2	~[MUSTAR]D=MU/>	-	2	|[WERE] =7	-
2	~[BEN]IGN=B5	-	2	~[MUSTER]=MU/]	-	2	[WA]F=WA	-
2	~[BEN]I=2N	-	2	[MUST]=M/	-	2	~[WITH]THE~=)	-
2	~[BEN]U=2N	-	2	~[MYSELF]~=MYF	-	2	~[WITH]A~=)	-
2	~[BEN]=B5	-	2	~[MC]=MC	-	2	[WITH]=)	-
2	~[BEQU]=2QU	-	1	[M].~=M	-	2	~[WIIL]~=W	-
2	~[BERET]=B]ET	-	1	.[M]=M	-	2	~[WHICH']=:I*'	-
2	~[BERG]=B]G	-	1	[M].!=M	-	2	~[WHICH]~=:	-
2	~[BERK]=B]K	-	1	~[M]~=;M	-	2	![WHID]E=WHID	-
2	~[BERL]=B]L	-	1	[M]=M	-	2	![WHERE]D=WH]D	-
2	~[BERM]=B]M	-	5	[M]=M	-	2	[WHEREVER]=:]"E	-
2	~[BERN]=B]N	-	1	~`[ND]~=4ND	-	2	[WHERE']ER=:]E'	-
2	~[BERR]=B]R	-	1	#[ND]~=ND	-	2	[WHERE]=":	-
2	~[BERS]=B]S	-	1	.[ND]~=ND	-	2	~[WHOSE]~=^:	-
2	~[BER]T=B]	-	2	~[N];#=;N	3	2	![WHOUS]E=WH\S	-
2	~[BER]W=B]	-	2	#[N]=;N	3	2	[WH]=:	-
2	~[BERYL]=B]YL	-	1	~[N];#=;N	4	2	[WOULD]=WD	-
2	~[BER]BECK=B]	-	1	#[N]=;N	4	2	[WORK]="W	-
2	~[BE]R=2	-	2	~[NIGHT]=NI<T	-	2	[WORD]=^W	-
2	~[BESIDE]=2S	-	2	~[NOBLES]SE=NO#S	-	2	[WORLD]=_W	-
2	~[BESS]=BESS	-	2	~[NOT]~=N	-	1	[W].~=W	-
2	~[BESTEA]D=BE/1	-	2	~[NONE]~=N"O	-	1	.[W]=W	-
2	~[BESTI]A=BE/I	-	2	~[NONES]~=N"OS	-	1	[W].!=W	-
2	~[BESTING]=BE/+	-	2	~[NON]ESS=NON	-	1	~[W]~=;W	-
2	~[BESTI]=2/I	-	2	~[N]ONES=N	-	1	[W]=W	-
2	~[BEST]O=2/	-	2	~[NONE]THE=N"O	-	5	[W]=W	-
2	~[BESTR]=2/R	-	2	~[NON]=NON	-	2	~[X];#=;X	3
2	~[BEST]=BE/	-	2	[NOWI]SE=NOWI	-	2	#[X]=;X	3
2	~[BE]S=2	-	2	[NOWA]Y=NOWA	-	1	~[X];#=;X	4
2	~[BET]A=2T	-	2	[NO]WHERE=NO	-	1	#[X]=;X	4
2	~[BETEL]=2TEL	-	2	[NA]MENT=NA	-	1	[X]:~=;X	-
2	~[BETH]I=2?	-	2	[NAME]="N	-	1	[X].~=X	-
2	~[BETH]O=2?	-	2	![NESS]=;S	-	1	.[X]=X	-
2	~[BET]I=2T	-	2	[NECESSARY]=NEC	-	1	[X].!=X	-
2	~[BET]O=2T	-	2	[NCRE]A=NCRE	-	1	~[X]~=;X	-
2	~[BETR]=2TR	-	1	[N].~=N	-	1	[X]=X	-
2	~[BETWEEN]=2T	-	1	.[N]=N	-	5	[X]=X	-
2	~[BETW]=2TW	-	1	[N].!=N	-	2	~[Y];#=;Y	3
2	~[BEW]=2W	-	1	~[N]~=;N	-	2	#[Y]=;Y	3
2	~[BEYOND]=2Y	-	1	[N]=N	-	1	~[Y];#=;Y	4
2	~[BEZ]=2Z	-	5	[N]=N	-	1	#[Y]=;Y	4
2	~[BE']=2'	-	2	~[O];#=;O	3	2	[YOUNG]="Y	-
2	[BLESS]=B.S	-	2	#[O]=;O	3	2	~[YOURSELF]=YRF	-
2	[BLEN]D=BL5	-	1	~[O];#=;O	4	2	~[YOURSELVES]=YRVS	-
2	[BLEED]=BLE$	-	1	#[O]=;O	4	2	[YOUR]=YR	-
2	![BLEU]=BL1U	-	2	~[OF]THE~=(	-	2	~[YOU']M=Y\'	-
2	![BLE]=#	-	2	~[OF]A~=(	-	2	~[YOU]~=Y	-
2	[BLIN]DE=BL9	-	2	[OFOR]=O=	-	1	[Y].~=Y	-
2	[BLIN]DI=BL9	-	2	[OF]=(	-	1	.[Y]=Y	-
2	[BLIND]=BL	-	2	~[OUT]HELD=\T	-	1	[Y].!=Y	-
2	~[BLUE]=BLUE	-	2	~[OUT]~=\	-	1	~[Y]~=;Y	-
2	~[BY AND BY]~=BY & BY		2	![OUND]=.D	-	1	[Y]=Y	-
	-		2	![OUNT]=.T	-	5	[Y]=Y	-

2	~[BY AND] =BY &	-	2	[OUGHT]="\	-	2	~[Z];#=;Z	3
2	~[BY] AT =BY	-	2	~[OURSELVES]~=\RVS	-	2	#[Z]=;Z	3
2	~[BY BUT] =BY B	-	2	[OU]=\	-	1	~[Z];#=;Z	4
2	~[BY IN] =BY 9	-	2	[OWORK]=O"W	-	1	#[Z]=;Z	4
2	~[BY]ON =BY	-	2	[OW]=[	-	1	[Z].~=Z	-
2	~[BY THE B]Y=0! B	-	2	![ONG]=;G	-	1	.[Z]=Z	-
2	~[BY] TO =BY	-	2	[ONEA]=ON1	-	1	[Z].!=Z	-
2	~[BY] WAS =BY	-	2	[ONEC]K=ONEC	-	1	~[Z]~=;Z	-
2	~[BY] WITH~=BY	-	2	[ONENESS]="O;S	-	1	[Z]=Z	-
2	~[BY] WITHOUT=BY	-	2	[ON]EN=ON	-	5	[Z]=Z	-
2	~[BY HIS] =0HIS	-	2	[ONER]=ON]	-	1	[[]=,7	-
2	~[BY ENOUGH]=05\<	-	2	[ONED]=ON$	-	5	[[]=[	-
2	~[BY =]==0	-	2	[ONES]IA=ONES	-	1	[\]=/	-
2	~[BY]!=0	-	2	[ONES]IM=ONES	-	5	[\]=\	-
2	~[BY]#=0	-	2	[ONES]S~=ONES	-	1	[]]=7'	-
2	~[BRAILLE]=BRL	-	2	~[ONESELF]~="OF	-	5	[]]=]	-
1	[B].~=B	-	2	[ONES]E~=ONES	-	1	[^]=,	-
1	.[B]=B	-	2	[ONEST]="O/	-	5	[^]=^	-
1	[B].!=B	-	2	[ONE]E=ONE	-	1	[__]#=--#	-
1	~[B]~=;B	-	2	[ONEOU]S=ONE\	-	1	[__]=..	-
1	[B]=B	-	2	[ONEO]=ONEO	-	2	~[_EN]~=.EN	-
5	[B]=B	-	2	[ONEY]~="OY	-	2	[_ENOUGH]=.5	-
2	~[C];#=;C	3	2	[ONEU]M=ONEU	-	2	[_TO]_=.6	-
2	#[C]=;C	3	2	[ONE]UR=ONE	-	2	[_IN] =.9	-
1	~[C];#=;C	4	2	[ONET]ED=ONET	-	2	[_INTO]_=.96	-
1	#[C]=;C	4	2	[ONET]S=ONET	-	2	[_WAS] =.0	-
2	[CHILDREN]=*N	-	2	[ONET]CY=ONET	-	2	[_WERE] =.7	-
2	~[CHILD]~=*	-	2	[ONET]~=ONET	-	2	[_HIS] =.8	-
2	[CHARACTER]="*	-	2	[ONET]TE=ONET	-	2	[_BE] =.2	-
2	[CH]RISTO=*	-	2	[ONEL]S=ONEL	-	2	[_BY]_=.0	-
2	[CHRIST]="C	-	2	![O]NEL~=O	-	1	[_/]=	-
2	[CH]=*	-	2	[ONE]="O	-	5	[_]=_	-
2	~[COMIN']=-IN'	-	2	[O]IN=O	-	1	[_]=	-
2	~[COMMON]EST=-MON	-	2	[OI]=OI	-	1	[`]=^	-
2	~[COM]!=-	-	2	[OEN]=OEN	-	5	[`]=`	-
2	~[C]ONE=C	-	2	[OED]!=OED	-	1	[{]=,7	-
2	~[CONO]=CONO	-	2	~[OVEREA]T=OV]1	-	5	[{]={	-
2	~[CON]NED=CON	-	2	[QUICK]=QK	-	1	[|]=^	-
2	~[CO]NA=CO	-	2	~[QUITE]~=Q	-	5	[|]=|	-
2	[CONY]=CONY	-	2	[QUESTION]="Q	-	1	[}]=7'	-
2	~[CONKER]=3K]	-	1	[Q].~=Q	-	5	[}]=}	-
2	[CONK]=CONK	-	1	.[Q]=Q	-	1	[~]=^	-
2	~[CONCEIVING]=3CVG	-	2	~[OVER]=OV]	-	5	[~]=~	-
2	~[CONCEIVE]=3CV	-	2	~[O'CLOCK]~=O'C	-			
2	~[CONCH]~=CON*	-	2	[ORSE]RADISH=ORSE	-			
2	~[CONS]~=CONS	-	2	[OON]E=OON	-			
2	~[CON]!=3	-	2	~[OLE]A=OLE	-			
2	[COULD]=CD	-	2	~[O]~MY=O	-			
2	[COEN]ZYME=CO5	-	1	~[O]~DEAR=O	-			

Appendix III: Translation Algorithm Using Structured English [37] [40]

```
program convert
begin
    do
        read_word
        convert word into normal form      // use table to convert lower to upper case.
                                           // tidy up graphics characters etc.
        convert_print_into_braille
        TABLE II (Continued.)
        RULE TABLE FOR TEXT TO STANDARD ENGLISH BRAILLE
    while not end_of_input
end // of main program

procedure convert_print_into_braille
    begin // turn print word into braille
        set current_state to 1
        set current_character to first character in word
            while still converting do // do the whole word
                begin
                    set match to FALSE // initialize for the loop
                    start search in rule table at rule defined by
                current_character
                repeat
                if focus_matches and state_ok and right_context_ok
                    and left_context_ok then
                    begin
                        output right hand side of rule      // i.e. the text
                                                            after the equals
                                                            sign
                        set current_state to new_state      // get new state
                                                            from end of the
                                                            rule
                        e
                        move along word by size of current rule focus
                    set match to TRUE
                    end
                else go to next rule
                if not match
                and new rule does not start with same letter as current_character
            then
                    begin // no more rules for that character
                        output current_character // so use default option
                        set current_state to 1
                        set match to TRUE // and output braille
                    character
                    end
            until match // keep going round until done current character
                set current_character to first character in word
            end // while still converting – keep going until done whole word
    end // of convert print into braille

function focus_matches
    begin
        set match to TRUE
        set input_index to index into input_buffer position for current_character
        set rule_index to index start of focus for rule
            TABLE II (Continued.)
            do
                if input_buffer[input_index] != rule[rule_index] then // not got a
            match
                    set match to FALSE
                    increment rule_index // move along rule
                    increment input_index // move along input
            while match and (rule[rule_index] != ’]’) // Note: ’]’ terminates focus
            return match
    end // of focus_matches

function state_ok
    begin // nonzero entry fires state
        if decision_table[input_class of current rule, current_state] > 0 then
            return FALSE
        else
            return TRUE
    end; // of state_ok

function left_context_ok // similar to right_context_ok below

function right_context_ok
    begin
        set match to TRUE
        increment input_index // step over ’]’
        do
            if rule[rule_index] is a wildcard then // ‘!’, ‘#’, ‘_’, ‘ ’, ‘|’, ‘‘’, ‘;’ or ‘+’
        begin
            if not valid_wildcard_match then //see wildcard definitions-Appendix 3
                    // Note: this will move along input buffer
                set match to FALSE // and increment input_index appropriately
            else do wildcard match // see wildcard definitions-Appendix 3
        end
            else
        begin
            if input_buffer[input_index] != rule[rule_index] then // not got a match
                set match to FALSE
                increment input_index // move along rule
        end
        increment rule_index // move along input
        while match and (rule[rule_index] != TAB) // Note: TAB terminates
                    // right hand context of rule
        return match
    end // of right_context_ok
```

Appendix IV: Rule Table for Braille-to-Text Translation [40]:

Rule format: input class <tab>[focus] right context =output<tab> next state

1	[ ]=	1	6	[6]=!	1	1	[I]=i	3
6	[ ]=	1	1	[6]=!	1	7	[J]=0	4
1	[!MVS]=themselves	3	1	[7']=]	3	6	[J]=j	6
3	[!]!: =the	5	6	[7']=]	3	3	[J]: =just	3
5	[!]= the	5	2	[7]_=were	3	1	[J]=j	3
1	[!]=the	3	3	[7]=(	2	6	[K]=k	6
3	["1] ="	1	1	[7]: =)	3	3	[KC/S#]=kilocycles/s	4
1	["D]=day	3	6	[7]: =)	3	7	[KC/S]: = kilocycles/s	3
1	["E]=ever	3	6	[7]=(	2	3	[KC/S]: =kilocycles/s	3
1	["F]=father	3	4	[7]=gg	3	1	[KC#]=kilocycles	4
1	["HAF]=hereafter	3	1	[7]=(	1	7	[KC]: = kilocycles	3
1	["H]=here	3	1	[8"']="...	2	3	[KC]: =kilocycles	3
1	["K]=know	3	6	[8']='	3	3	[KW#]=kilowatts	4
1	["L]=lord	3	1	[8']: ='	3	7	[KW]: = kilowatts	3
1	["M]=mother	3	2	[8] =his	3	3	[KW]: =kilowatts	3
1	["N]=name	3	3	[8]="	2	3	[K]: =knowledge	3
1	["OF]=oneself	3	6	[8]: =?	31	1	[K]=k	3
1	["O]=one	3		[8]: =?	3	6	[L]=l	63
1	["P]=part	3	6	[8]="	1		[LR]=letter	3
1	["Q]=question	3	1	[8]="	1	3	[LL]A=ll	3
1	["R]=right	3	6	[99]=*	1	3	[LL]E=ll	3
1	["S]=some	3	1	[99]=*	1	3	[LL]I=ll	3
1	["T]=time	3	3	[96]=into	1	3	[LL]O=ll	3
1	["U]=under	3	6	[9]=in	3	3	[LL]U=ll	3
1	["W]=work	3	1	[9]=in	3	3	[LL]=little	3
1	["Y]=young	3	3	[:]: =which	3	3	[LB#]=pounds	4
1	["!]=there	3	6	[:]=wh	3	7	[LB]: = pounds	3
1	["*]=character	3	1	[:]=wh	3	3	[LB]: =pounds	3
1	["?]=through	3	1	[;6]=+	1	3	[L]: =like	3
1	[":]=where	3	1	[;_]="	1	1	[L]=l	3
1	["\]=ought	3	1	[;8]=*	1	6	[M]=m	6
1	["]="	1	1	[;4]=/	1	1	[M*]=much	3
7	[#]=:	4	1	[;7]==	1	1	[M/]=must	3
6	[#]=	4	4	[;E]=ence	3	1	[MYF]=myself	3
4	[#]=ble	3	4	[;G]=ong	3	3	[MN#]=minutes	4
1	[#']='	4	4	[;L]=ful	3	3	[MN]: =minutes	3
1	[#]=	4	4	[;N]=tion	3	3	[MC/S#]=megacycles/s	4
6	[$]=ed	3	4	[;S]=ness	3	7	[MC/S]: = megacycles/s	3
1	[$]=ed	3	4	[;T]=ment	3	3	[MC/S]: =megacycles/s	3
1	[%D]=should	3	4	[;Y]=ity	3	3	[MC#]=megacycles	3
3	[%]: =shall	3	6	[;]=	6	7	[MC]: = megacycles	3
6	[%]=sh	3	1	[;]=	6	3	[MC]: =megacycles	3
1	[%]=sh	3	6	[<]=gh	3	3	[M#]=miles	4
1	[&/OR]=and/or	3	1	[<]=gh	3	3	[M]: =more	3
3	[&]!: =and	5	6	[>]=ar	3	1	[M]=m	3
5	[&]= and	5	1	[>]=ar	3	6	[N]=n	6
6	[&]=and	3	1	[?YF]=thyself	3	1	[NEC]: =necessary	3
1	[&]=and	3	3	[?]: =this	3	3	[NEI]: =neither	3
1	["'0]: =..."	3	6	[?]=th	3	1	[NEWSLR]=newsletter	3
1	["'8]: =...?	3	1	[?]=th	3	3	[N]: =not	3
1	["']=...	2	1	[@4]!=$	3	1	[N]=n	3
6	['S]='s	2	6	[@]='	1	6	[O]=o	6
3	[']='	2	1	[@]='	1	3	[OZ#]=ounces	4
7	['S]='s	2	7	[A]=1	4	7	[OZ]: =ounces	3
7	[']: ='	4	6	[A]=a	6	3	[OZ]: = ounces	3
7	[']=,	4	3	[ABV]=above	3	3	[O'C]=o'clock	3
6	[']='	2	3	[AB]: =about	3	1	[O]=o	3
4	[']='	3	1	[ACLY]=accordingly	3	6	[P]=p	6
3	[']='	2	3	[AC]: =according	3	1	[PD]: =paid	3
1	[']='	2	3	[ACR]: =across	3	1	[P}CVG]=perceiving	3
1	[`]=`	2	3	[AF]B=after	3	1	[P}CV]=perceive	3
3	[(]!: =of	5	3	[AF]G=after	3	1	[P}H]=perhaps	3
5	[(]= of	5	3	[AF-]=lafter-	1	3	[PT#]=pt	4
1	[(]=of	3	3	[AFN]=afternoon	3	3	[PT]: =pt	3
3	[)]!: =with	5	3	[AFW]=afterward	3	3	[P>#]=paragraph	4
5	[)]= with	5	3	[AF]?=after	3	7	[P>]: = paragraph	3
1	[)]=with	3	3	[AF]M=after	3	3	[P>]: =paragraph	3
3	[*N]=children	3	3	[AF]D=after	3	1	[P4M4]=p.m.	3
3	[*]: =child	3	1	[AF]: =after	3	3	[P#]=p.	4
1	[*]=ch	3	3	[AG/]=against	3	3	[P]: =people	3
1	[+]=ing	3	3	[AG]: =again	3	1	[P]=p	3
7	[,]=	1	3	[ALM]: =almost	3	6	[Q]=q	6
6	[,,]=<SHIFT_WORD>	6	3	[ALR]: =already	3	1	[QT#]=quarts	4
6	[,]=<SHIFT_CHAR>	6	3	[AL]: =also	3	7	[QT]: = quarts	3
3	[,,]=<SHIFT_WORD>	1	3	[AL?]: =although	3	3	[QT]: =quarts	3
3	[,]=<SHIFT_CHAR>	1	3	[ALT]: =altogether	3	1	[QR#]=quaters	4
4	[,N]=ation	3	3	[ALW]: =always	3	7	[QR]: = quaters	3
4	[,Y]=ally	3	1	[A4M4]=a.m.	3	3	[QR]: =quaters	3
6	[,8]='	2	3	[A]!: =a	5	1	[QK]=quick	3
3	[,8]='	2	5	[A]= a	5	3	[Q]: =quite	3
3	[,7]=[	1	1	[A]=a	3	1	[Q]=q	3
7	[,G]: = grammes	3	7	[B]=2	4	6	[R]=r	6
3	[,G]: =grammes	3	6	[B]=b	6	1	[R4I4P4]=r.i.p	3
3	[,G]#=grammes	3	3	[BLLY]: =blindly	3	1	[RCVG]=receiving	3
7	[,M]: =metres	3	3	[BL]F=blind	3	1	[RCV]=receive	3
3	[,M]: =metres	3	3	[BL;S]: =blindness	3	1	[RJCG]=rejoicing	3

3	[,M]#=metres	3
7	[,L]: =litres	3
3	[,L]: =litres	3
3	[,L]#=litres	3
1	[,,]=^^	1
1	[,]=^	1
7	[--]=-	2
6	[--]=-	2
1	[----]=----	3
1	[--]= --	2
7	[-]=-	4
6	[-]~: =-	6
4	[-]=-	2
3	[-]=com	3
1	[-]=-	2
4	[.D]=ound	3
4	[.E]=ance	3
4	[.N]=sion	3
4	[.S]=less	3
4	[.T]=ount	3
1	[.1]=>	3
1	[.]=_	1
7	[/]#=/	4
7	[/]: =st	3
3	[/]: =still	3
6	[/;]=/	6
6	[/]=st	3
1	[/]=st	3
4	[0']='	3
2	[0]_=was	3
3	[0]=by	1
6	[0]="	3
1	[0]: ="	3
1	[0]="	1
7	[1] =,	4
7	[1]=.	4
1	[1]: =,	3
6	[1]=,	3
4	[1]=ea	3
1	[1]=,	1
3	[2C]: =because	3
3	[2F]H=before	3
3	[2F]: =before	3
3	[2H]H=behind	3
3	[2H]: =behind	3
3	[2LL]: =belittle	3
3	[2L]: =below	3
3	[2N]: =beneath	3
3	[2SS]: =besides	3
3	[2S]: =beside	3
3	[2T]: =between	3
3	[2Y]: =beyond	3
3	[2]=be	3
6	[2]=;	3
1	[2]: =;	3
4	[2]=bb	3
1	[2]=;	1
1	[3CVG]=conceiving	3
1	[3CV]=conceive	3
1	[3P#]=per cent	4
7	[3P]= per cent	4
1	[3P]=per cent	3
1	[3#]=:	4
6	[3]=:	3
1	[3]: =:	3
4	[3]=cc	3
3	[3]=con	3
1	[3]=:	1
3	[4#]=dollars	4
7	[4] =.	3
6	[4]=.	6
1	[4]: =.	3
4	[4]=dd	3
3	[4]=dis	3
1	[4]=.	1
3	[5]: =enough	3
6	[5]=en	3
1	[5]=en	3
3	[6]=to	1
1	[6]: =!	3
4	[6]=ff	3
1	[BL]M=blind	3
3	[BL]: =blind	3
1	[BRL]=braille	3
3	[B]: =but	3
1	[B]=b	3
7	[C]=3	4
6	[C]=c	6
1	[C/O]=c/o	3
3	[CW#]=hundredweight	4
7	[CW]: = hundredweight	3
3	[CW]: =hundredweight	3
3	[CD]=could	3
3	[C#]=cents	4
3	[C]: =can	3
1	[C]=c	3
7	[D]=4	4
6	[D]=d	6
1	[DCVG]=deceiving	3
1	[DCV]=deceive	3
1	[DCLG]=declaring	3
3	[DCL]=declare	3
3	[DM#]=dm	4
3	[DM]: =dm	3
7	[DM]: = dm	3
3	[DG#]=degrees	4
7	[DG]: = degrees	3
3	[DG]: =degrees	3
3	[D#]=pence	4
3	[D]: =do	3
1	[D]=d	3
7	[E]=5	4
6	[E]=e	6
3	[EI]: =either	3
3	[EX#]=ex	4
7	[EX]: = example	3
3	[EX]-=ex	3
3	[EX]: =example	3
7	[EXS]: = examples	3
3	[EXS]: =examples	3
1	[E4G4]=e.g.	3
3	[E]: =every	3
1	[E]=e	3
7	[F]=6	4
6	[F]=f	6
3	[F/]=first	3
1	[FRS]=friends	3
1	[FR]L=friend	3
1	[FR]: =friend	3
3	[FT#]=feet	4
7	[FT]: = feet	3
3	[FT]: =feet	3
3	[F#]=francs	4
3	[F]: =from	3
1	[F]=f	3
7	[G]=7	4
6	[G]=g	6
3	[GD]=good	3
1	[GRT]=great	3
3	[GL#]=gallons	4
7	[GL]: = gallons	3
3	[GL]: =gallons	3
3	[G#]=guineas	4
3	[G]: =go	3
1	[G]=g	3
7	[H]=8	4
6	[H]=h	6
3	[H}F]=herself	3
3	[HMF]=himself	3
3	[HMM]=hmm	3
3	[HM]=him	3
3	[HR#]=hours	4
3	[HR]: =hours	3
3	[H]: =have	3
1	[H]=h	3
7	[I]=9	4
6	[I]=i	6
1	[IMM;S]=immediateness	3
1	[IMMLY]=immediately	3
3	[IMM] =immeiate	3
1	[I4E4]=i.e.	3
3	[I#]=inches	4
1	[RJC]=rejoice	3
3	[R#]=rupees	4
3	[R]: =rather	3
1	[R]=r	3
6	[S]=s	6
1	[SD]: =said	3
1	[S*]: =such	3
3	[ST#]=stones	4
7	[ST]: = stones	3
3	[SE#]=seconds	4
3	[SE]: =seconds	3
3	[S#]=shillings	4
3	[S'#]=section	4
3	[S]: =so	3
1	[S]=s	3
6	[T]=t	6
3	[TD]=today	3
3	[TGR]=together	3
3	[TM]=tomorrow	3
3	[TN]=tonight	3
3	[T#]=tons	4
3	[T]:=that	3
1	[T]=t	3
3	[U4K4]=U.K.	6
6	[U]=u	6
3	[U]: =us	3
1	[U]=u	3
6	[V]=v	6
3	[V]: =very	3
1	[V]=v	3
6	[W]=w	6
3	[WD]=would	3
3	[W]: =will	3
1	[W]=w	3
6	[X]=x	6
3	[XS]: =its	3
1	[XF]=itself	3
3	[X]:=it	3
1	[X]=x	3
6	[Y]=y	6
1	[YRF]=yourself	3
1	[YRVS]=yourselves	3
3	[YR]=your	3
3	[YD#]=yards	4
7	[YD]: = yards	3
3	[YD]: =yards	3
3	[Y]:=you	3
1	[Y]=y	3
6	[Z]=z	6
3	[Z]: =as	3
1	[Z]=z	3
1	[\RVS]=ourselves	3
3	[\]: =out	3
6	[\]=ou	3
1	[\]=ou	3
1	[^U]=upon	3
1	[^W]=word	3
1	[^!]=these	3
1	[^?]=those	3
1	[^:]=whose	3
1	[^]=`	1
1	[_C]=cannot	3
1	[_H]=had	3
1	[_M]=many	3
1	[_S]=spirit	3
1	[_W]=world	3
1	[_!]=their	3
6	[_]=_	1
1	[_]=_	1
1	[{O]=.	3
6	[[]=ow	3
1	[[]=ow	3
3	[=]!: =for	5
5	[=]= for	5
6	[=]=for	3
1	[=]=for	3
6	[]]=er	3
1	[]]=er	3

Appendix V: VHDL Coding For Text-to-Braille Translation

Included in the companion CD

Appendix VI: VHDL Coding For Fast Text-to-Braille Translation

Included in the companion CD

Appendix VII: VHDL Coding For Braille Notetaker

Included in the companion CD