DESIGN OF ADVANCED MOTION COMMAND GENERATORS UTILIZING FPGA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ULAŞ YAMAN

IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MECHANICAL ENGINEERING

JUNE 2010

Approval of the thesis:

DESIGN OF ADVANCED MOTION COMMAND GENERATORS UTILIZING FPGA

submitted by **ULAŞ YAMAN** in partial fulfillment of the requirements for the degree of **Master of Science in Mechanical Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Suha Oral _____
Head of Department, **Mechanical Engineering**

Assist. Prof. Dr. Melik Dölen _____
Supervisor, **Mechanical Engineering Dept., METU**

Assist. Prof. Dr. A. Buğra Koku _____
Co-Supervisor, **Mechanical Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Mehmet Çalışkan _____
Mechanical Engineering Dept., METU

Assist. Prof. Dr. Melik Dölen _____
Mechanical Engineering Dept., METU

Assist. Prof. Dr. A. Buğra Koku _____
Mechanical Engineering Dept., METU

Assist. Prof. Dr. E. İlhan Konukseven _____
Mechanical Engineering Dept., METU

Assoc. Prof. Dr. Veysel Gazi _____
Electrical and Electronics Engineering Dept., TOBB-ETU

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as require by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name : Ulaş YAMAN

Signature :

ABSTRACT

DESIGN OF ADVANCED MOTION COMMAND GENERATORS UTILIZING FPGA

Yaman, Ulaş

M.Sc., Department of Mechanical Engineering

Supervisor : Assist. Prof. Dr. Melik Dölen

Co-Supervisor : Assist. Prof. Dr. A. Buğra Koku

June 2010, 143 pages

In this study, universal motion command generator systems utilizing a Field Programmable Gate Array (FPGA) and an interface board for Robotics and Computer Numerical Control (CNC) applications have been developed. These command generation systems can be classified into two main groups as polynomial approximation and data compression based methods. In the former type of command generation methods, the command trajectory is firstly divided into segments according to the inflection points. Then, the segments are approximated using various polynomial techniques. The sequence originating from modeling error can be further included to the generated series. In the second type, higher-order differences of a given trajectory (i.e. position) are computed and the resulting data are compressed via lossless data compression techniques. Besides conventional approaches, a novel compression algorithm is also introduced in the study. This group of methods is capable of generating trajectory data at variable rates in forward and reverse directions. The generation of the commands is carried out according to the feed-rate (i.e. the speed along the trajectory) set by the external logic dynamically. These command generation techniques are implemented in MATLAB and then the best ones from each

group are realized using FPGAs and their performances are assessed according to the resources used in the FPGA chip, the speed of command generation, and the memory size in Static Random Access Memory (SRAM) chip located on the development board.

Keywords: Command Generation, Data Compression, FPGA, Polynomial Approximation, Adjustable Feed-rate, Linear Interpolation

ÖZ

FPGA KULLANARAK İLERİ HAREKET KOMUT ÜRETEÇLERİ TASARIMI

Yaman, Ulaş
Yüksek Lisans, Makina Mühendisliği Bölümü
Tez Yöneticisi : Yrd. Doç. Dr. Melik Dölen
Ortak Tez Yöneticisi : Yrd. Doç. Dr. A. Buğra Koku

Haziran 2010, 143 sayfa

Bu çalışmada, Alan Programlanabilir Kapı Dizini (Field Programmable Gate Array - FPGA) kullanan evrensel hareket komutu üreteçleri ve Robotik / Bilgisayarlı Sayısal Denetim ("Computer Numerical Control" - CNC) uygulamaları için bir arayüz kartı geliştirilmiştir. Geliştirilen komut üreteç sistemleri, fonksiyon yaklaşıklama ve veri sıkışırtırma tabanlı olmak üzere iki sınıfa ayrılabilir. Fonksiyon yaklaşıklama tabanlı komut üreteç uygulamalarında, komut dizini öncelikli olarak bükülme noktalarından bölümlere ayrılmıştır. Daha sonrasında bu bölümler farklı fonksiyon yaklaşıklama yöntemleri kullanılarak ifade edilmiştir. Yaklaşıklamadan kaynaklanan hata dizini kodlama sırasında saklanarak üretilen komutlara beslenebilir. Diğer komut üretme yöntemlerinde ise, verilen hareket dizininin yüksek dereceden farkı alındıktan sonra kayıpsız veri sıkıştırma teknikleri kullanılarak sıkıştırılır. Bu çalışmada, geleneksel sıkıştırma tekniklerinin yanı sıra yeni bir veri sıkıştırma yöntemi de sunulmuştur. Bu grupta önerilen yöntemler, komutları ileri ve geri yönlerinde farklı hızlarda üretebilme yetilerine sahiptirler. Komut üretim hızı sisteme dışarıdan dinamik olarak beslenmektedir.

Geliştirilen komut üretme teknikleri MATLAB kullanılarak bilgisayar ortamında gerçekleştirilmiş ve her grupta en iyi sonucu veren yöntemler FPGA kullanılarak gerçekleştirilmiştir. Bu yöntemler FPGA kırmığı üzerinde kullandıkları kaynaklar, komut üretim hızı ve geliştirme kartında bulunan Durağan Rastgele Erişimli Bellek'te (Static Random Access Memory - SRAM) depolanan verinin büyüklüğüne göre değerlendirilmişlerdir.

Anahtar kelimeler: Komut Üretimi, Veri Sıkıştırma, Alan Programlanabilir Kapı Dizini, Ayarlanabilir Üretim Hızı, Doğrusal Enterpolasyon

Alevlerin ucunda sönen hayatlara,

"Yaşamayı ciddiye alacaksın,
yani o derecede, öylesine ki,
mesela, kolların bağlı arkadan, sırtın duvarda,
yahut kocaman gözlüklerin,
beyaz gömleğinle bir laboratuvarda,
insanlar için ölebileceksin,
hem de yüzünü bile görmediğin insanlar için,
hem de hiç kimse seni buna zorlamamışken,
hem de en güzel en gerçek şeyin
yaşamak olduğunu bildiğin halde."

Nazım Hikmet RAN

ACKNOWLEDGEMENTS

First of all I offer my heartily thanks to my supervisor Assist. Prof. Dr. Melik Dölen and co-supervisor Assist. Prof. Dr. A. Buğra Koku, who have supported me throughout the thesis with their unyielding patience and knowledge.

I would like to thank to the member of the SPARC research group for their technical support throughout the thesis period. This period might have been so painful without their support and advices.

Additionally, I would like to show gratitude to my colleagues at the department for their sincere friendships especially during the writing process of the thesis and the meal times.

I would like to thank TÜBİTAK for the scholarship (BİDEB 2228) and the conference travel grant during my graduate study. My study was also a part of TÜBİTAK project under the contract number 108E048.

Lastly, I am deeply in debt to my parents Şahhanım and Azimet Yaman for their never-ending love and spiritual support at critical and opportune times.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

LIST OF TABLES

TABLES

LIST OF SYMBOLS

SYMBOLS

B	Bernstein Polynomials
c_{max}	maximum value of coefficients
c_{min}	minimum value of coefficients
d_{max}	maximum value of original data sequence (counts)
d_{min}	minimum value of original data sequence (counts)
e_{max}	maximum value of error sequence (counts)
e_{min}	minimum value of error sequence (counts)
f	feed-rate (mm/s)
f_{max}	maximum feed-rate (mm/s)
L	Legendre Polynomials
l	binary length of encoded data (bits)
m	group size of Golomb coding
N	length of original data sequence
n	order of finite difference
∇	finite difference

n_0	number of zero magnitude data
N_c	number of coefficients
q	motion-state sequence
r	compression ratio
r_e	compression of error sequence
T	Chebyshev Polynomials
u	decoded command

ABBREVIATIONS

ADC	Analog-to-Digital Converter
ASM	Algorithmic State Machine
BJT	Bipolar Junction Transistor
BP	Bernstein Polynomials
CAM	Computer Aided Manufacturing
CD	Clock Divisor
CLDATA	Cutter Location Data
CMOS	Complementary Metal Oxide Semiconductor
CNC	Computer Numerical Control
CP	Chebyshev Polynomials

CTM	Command Transmit Module
DAC	Digital-to-Analog Converter
DDFS	Direct Digital Frequency Synthesizers
DM	Driver Module
DSP	Digital Signal Processor
DU	Decoding Unit
FPGA	Field Programmable Gate Array
FPOM	Floating Point Operation Module
FSM	Finite State Machine
GTL	Gunning Transceiver Logic
GTLP	Gunning Transceiver Logic Plus
HILS	Hardware-in-the-Loop Simulator
I/O	Input / Output
IP	Intellectual Property
JSD	Joint State Data
LED	Ligth Emitting Diode
LP	Legendre Polynomials
LVDS	Low Voltage Differential Signaling
LVPECL	Low Voltage Positive Emitter Coupled Logic

LZ	Lempel Ziv
MMU	Memory Management Unit
NC	Numerical Control
PWM	Pulse Width Modulation
PWMG	Pulse Width Modulation Generator
RMS	Root Mean Square
SDRAM	Synchronous Dynamic Random Access Memory
SISO	Serial Input Serial Output
SIPO	Serial Input Serial Output
SM	Splitter Module
SR	Shift Register
SRAM	Static Random Access Memory
SW	Sine Wave
TTL	Transistor-to-Transistor Logic
VHDL	Very High Speed Integrated Circuit Hardware Description Language

CHAPTER 1

INTRODUCTION

1.1 Motivation

Modern servo-drive systems employ digital motion controllers (DSPs, micro-controllers) to regulate precisely not only motor currents (electromagnetic torque) but also motor's angular velocity along with the position. If the drive system is configured for (digital) motion control, the relevant reference signals (velocity or position) must be generated by a central controller unit (host) depending on the trajectory to be followed. These signals are eventually transferred to each motor driver via a serial communication protocol (SERCOS, CAN, Profibus, TCP/IP, RS-232, RS-485, etc.). This approach frequently pushes the communication interface to its limits for high-end applications.

Industrial motion controller units utilize vector data tables to represent the trajectory in terms of linear patches. These cards can then perform a linear interpolation between the two consecutive entries in real-time to produce the relevant reference signals for the position servo-control loop. For complex trajectories the size of the vector table may exceed the available resources on the system. The conventional machining approach does not meet the requirements of high speed and high accurate machining in cases where the trajectories are complicated.

In today's technology, memory devices with large capacity as well as multi-core processors running at high clock frequencies are widely available in the market at relatively low cost. Consequently, there is a potential for devising simple yet very effective command generators for computer numerically controlled machinery that benefit fully from the properties of these advanced devices [1].

The fundamental motivation of this study is to develop a direct command generator system with variable feed-rate in forward and reverse directions for servo motor drives where the commands could be produced directly in the drive system without the need for intermittent data transfer from a host controller. This FPGA based system, which could be directly embedded into a motor drive system, is expected to generate the relevant commands by utilizing not only the (dynamically adjusted) speed along the traced trajectory but also decompressed data being produced in advance to represent trajectory to the desired accuracy.

1.2 Scope of the Thesis

The difficulties mentioned in the previous section are overcome by the proposed method in this study. It is implemented into the conventional manufacturing process as illustrated in Figure 1-1. The figure indicates that the method is applicable for both Robotics and CNC Applications. Input to the system is Tool Location Data (TLD) for CNC Applications and Joint State Data (JSD) for Robotics Applications. After the encoder commands are formed according to the incoming data from the previous operations, the encoding algorithm is applied onto the commands structuring the encoded JSD or TLD. Encoding methods applied within the context of this thesis are

- Lossless Data Compression of Higher Order Finite Differences of JSD / TLD
    - Huffman Compression Algorithm
    - Arithmetic Coding

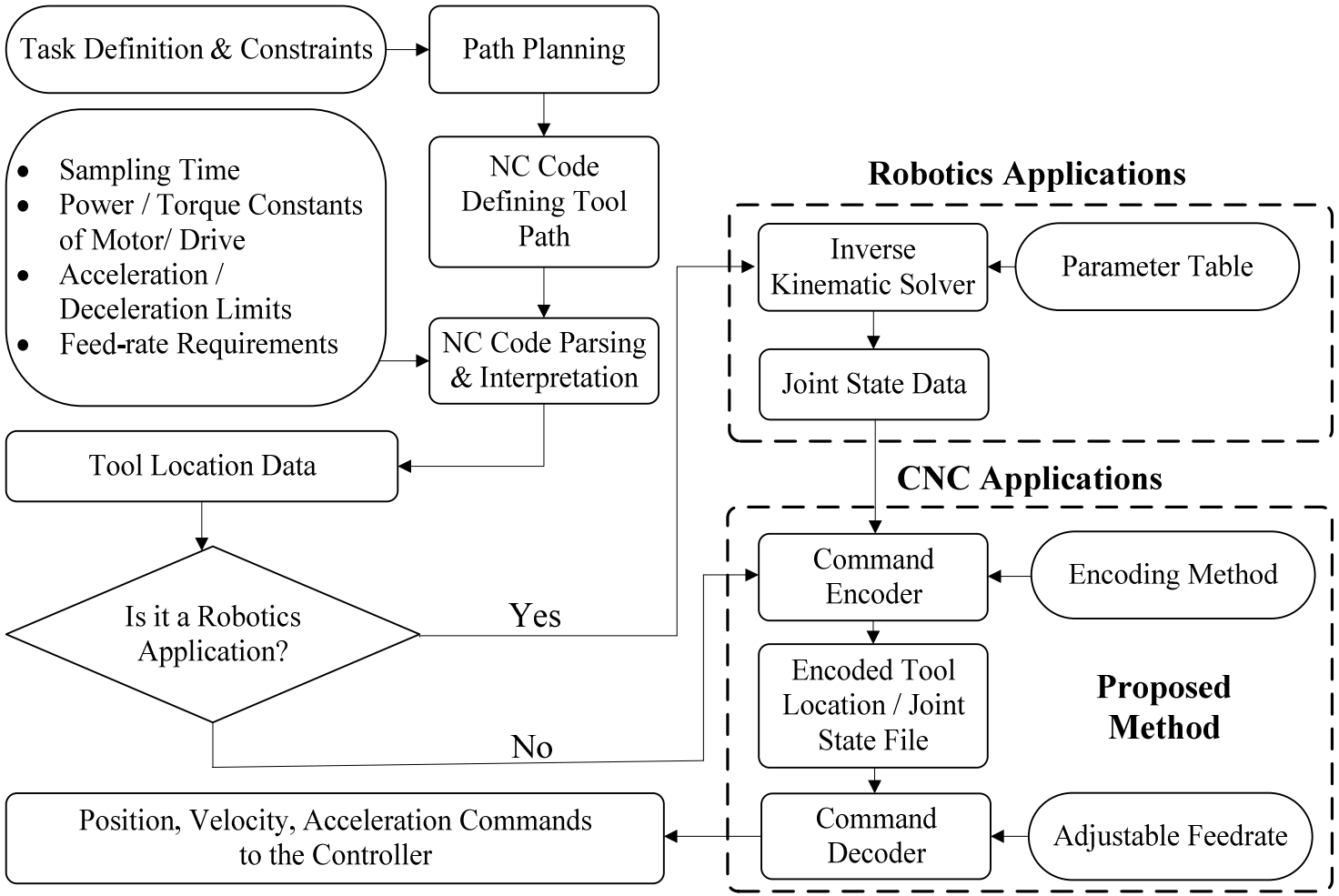

Figure 1-1 Flow Chart of the Proposed Method

- Proposed Compression (ΔY) Algorithm

- Polynomial Representation of Segmented JSD / TLD
  - Chebyshev Polynomials
  - Legendre Polynomials
  - Bernstein Polynomials

The performances of these aforementioned encoding methods are comparatively evaluated according to approximation error, memory requirement, computational complexity, ease of decoding, etc. in MATLAB environment. The best methods for each group are realized on the FPGA Development Board. Once the encoded JSD / TLD is stored into the SRAM of the board, the decoding circuit in the FPGA chip is ready to decode the data in real-time. Consequently, decoded position, velocity, and acceleration commands can be fed to the (centralized or distributed) joint-axis motion controller as the reference signals.

The arrangement of the proposed command generation method in the thesis with the other units of the FPGA based motion control system is illustrated in Figure 1-2. This general motion control system is composed of three main units: FPGA Based Command Generation System, FPGA Based Motion Controller System, and Hardware-in-the-Loop Simulator (HILS). The last two units are not within the scope of the thesis. Only the shaded area in Figure 1-2 is elaborated throughout the thesis.

The generated reference commands (position, velocity, and acceleration) are transferred to the second FPGA board in the system which is responsible for motion control. The velocity and acceleration commands are also sent to the motion control unit, since the control algorithm employed within the FPGA may require these commands for precise control. Then the generated control inputs are fed to the HILS unit through the developed FPGA Input / Output (I/O) Interface Board. This board is also discussed in the thesis. After the manipulated input vector is transferred to the HILS, the control process is simulated and the resulted output vector is fed to the controller unit with the help sensors utilization.

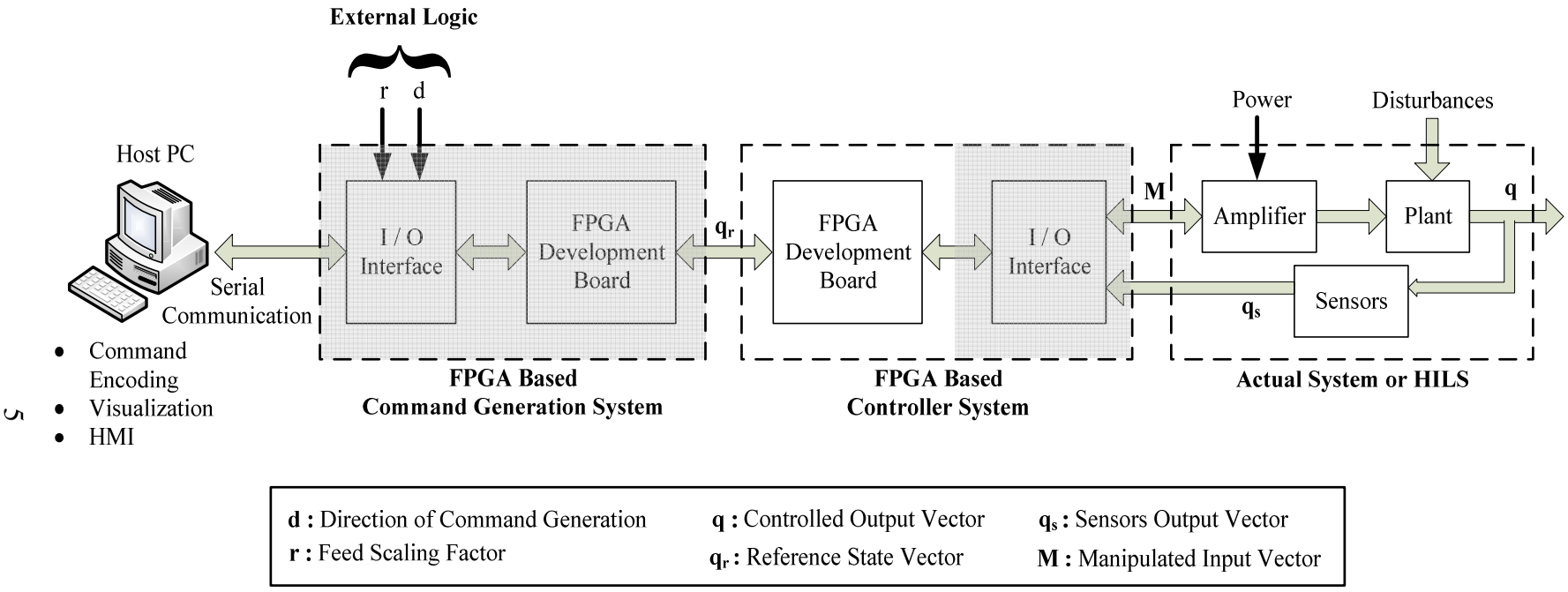

Figure 1-2 FPGA Based Motion Control System

1.3 Organization

The thesis is divided into seven chapters. The second chapter discusses the literature on the relevant topics of the thesis such as differencing, data compression algorithms, polynomial approximation techniques, FPGA implementation of these methods, FPGA-based command generation systems, and command generators with variable feed-rates. In the following chapter, a board developed for FPGA interfacing with CNC and Robotics Applications is investigated. This interface board consists of five modules namely mother board, analog input card, analog output card, digital input card, and digital output card. The fourth chapter elaborates the first group of command generation methods utilizing segmentation and polynomial approximation in MATLAB. The approximation method with Chebyshev Polynomials, which has the smallest compression ratio, is employed on the FPGA Development Board using two different approaches. In the first approach, the algorithm is implemented with a softcore embedded processor in the FPGA chip and in the second approach the algorithm is directly realized in the FPGA chip utilizing Very High Speed Integrated Circuit Hardware Description Language (VHDL). In the fifth chapter the second group of command generation methods utilizing differencing and compression are explained and their performances are evaluated in MATLAB. The most successful one of these compression algorithms, namely the proposed compression method, is realized on the FPGA Development Board in two different ways as done in the previous chapter. In this chapter, the concept of command generation in forward and reverse direction with variable feed-rate is also introduced and evaluated in detail. The method utilizes the command compression algorithm proposed in this chapter. During implementation on the FPGA chip some variations are also considered and compared with each other. In the sixth chapter, the most successful command generation method is employed on a test case and its performance is evaluated. The thesis is concluded by summarizing the key results of this research. Possible future works are also presented in the last chapter.

CHAPTER 2

LITERATURE SURVEY

In this chapter of the thesis, the relevant literature topics are discussed in detailed manner and open research areas are also highlighted for possible further study.

2.1 Differencing

The study on literature starts with a detailed investigation of differencing and its application areas. With the help of this study, the necessity of differencing before compression in command generation is elaborated.

Differencing is commonly used in different fields such as data (video, image, speech, index, etc.) compression, and movement detection in tracking. The major requirement in differencing is that the samples in the data must be coherent so that each sample can easily be predicted according to the difference value and the previous sample. Due to the ease of implementation, differencing without compression schemes may be preferable in video literature [3].

It is very common to use differencing during the storage of files that do not have big variations between the entries. For instance, if the average temperature of the days in a year is necessary for graphical purposes, it is wise to store the difference of temperatures between consecutive entries (i.e., d_1, $d_2 - d_1$, …, $d_{365} - d_{364}$) rather than storing the absolute temperature value of each day (i.e., d_1, d_2, …, d_{365}) [4].

This approach decreases the memory usage since the difference between the two consecutive data points is less than the absolute value of the former entry. There may also be many zeros in the differenced format, which will further decrease the memory size.

Differencing is also used in detection of movement of objects. Balch et al. [5] developed a novel machine vision system that can easily follow hundreds of small insects. In order to demonstrate the usability of their system, they analyzed the behaviours of ant colonies in their study. In detection, they simply subtracted the current image from the back ground image to find the movement.

2.2 Data Compression Techniques

In this part of the chapter, different data compression methods are explained and their main application areas are explained. Although the digital communication system performances and the mass storage density are improving rapidly, data compression algorithms still continue to be an important part of many engineering fields since it can eliminate the disadvantages of data storage and overcome the limitations of transmission bandwidths via enabling devices to send the same amount of data in fewer bits [6].

Two types of data compression techniques exist in the literature: lossless and lossy. Lossless data compression, where the original data is extracted without any loss after decompression, applications have increased over the past years due to the need to improve the storage capacity and data transfer rate [7]. On the other hand in lossy compression, original data can only be approximated after decompression. Lossy compression is usually used in situations when the data do not need to be stored perfectly. For instance, pictures can be restored using lossy compression paradigms without much difference from the original picture. In cases where data loss cannot be afforded lossless compression techniques must be used. This is valid especially in text files, since loss of a character can lead to much different situations [8].

The most commonly used data compression techniques in the literature are Huffman, Arithmetic, and Golomb Coding methods. These three paradigms are discussed in a detailed manner in the following sub-sections.

2.2.1 Huffman Coding

Huffman Coding [9] algorithm is one of the most commonly used lossless data compression method in the literature in various areas. The main concept of the algorithm is that it maps an alphabet (of the same size) to a totally different form composed of strings with variable size. The characteristic properties of these symbols are that the ones having high probability have a smaller representation than those occurring less in the file.

Huffman coding belongs to the group of variable length character encoding methods, since in the resulting code the most common characters would be short and the infrequent ones would be long. For the illustration, assume that it is necessary to encode the characters U, L, A, S, Y, M, and N. If conventional approach were used, only three bits per character were necessary for encoding. Suppose that the relative frequency of these characters is as given in Figure 2-1. In the figure, the characters and their frequencies in the sequence are displayed on the left side. On the right side of the figure Huffman codes are displayed for each character. Huffman tree is formed in between of the frequencies and the Huffman codes. During constructing the tree, after the frequency data of the sequence is determined, two elements with the lowest frequencies are selected as the leaf nodes of the Huffman tree. Then the frequencies of these two elements are added together and the resulting value becomes the frequency for the new node. This approach continues until the Huffman tree is completed or until the last node having a frequency of 100%. Huffman code of each character is found by leading from the top of the tree to the corresponding character.

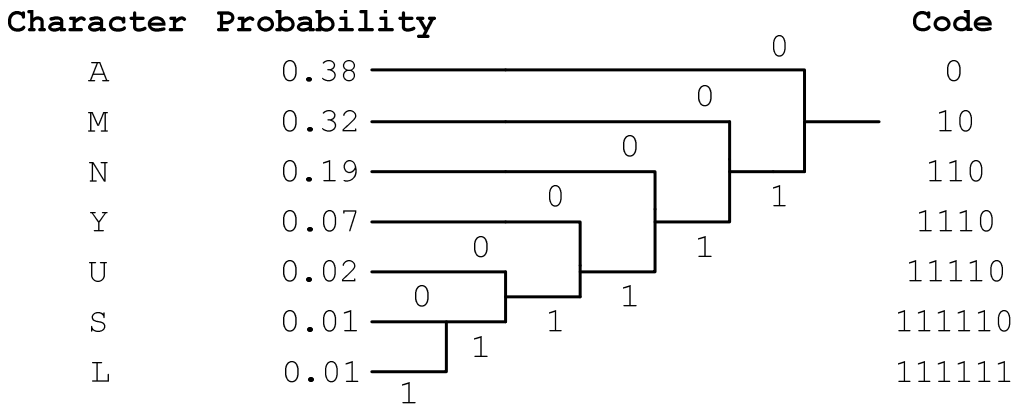

Figure 2-1 Huffman Code for the Given Set of Characters

Conventional Huffman encoding is a non-deterministic one, since a data set can be represented by more than one possible Huffman tree. While constructing the Huffman tree in Figure 2-1, two additional rules described in [8] are applied in order have a unique Huffman tree representation. The first extra rule is that the characters with shorter Huffman codes are placed to the upper branches of the nodes. For instance, in Figure 2-1 S and L are at the bottom part of the tree since they have the longest Huffman codes. Secondly, characters with Huffman codes of same length are placed according to their appearance order. In the figure S and L have the same length of codes. Since S is assumed to be encountered first, it is placed on the upper branch of the node. If these two extra rules are also applied during encoding, it is guaranteed that the Huffman tree is unique for the given set of characters. Due to the word-based memory units, it is difficult to implement Huffman encoding schemes which have variable length of codes. Additional rules described above should be used for fast and lossless recovery in FPGA implementations [8].

A sample encoded data in the Huffman code of Figure 2-1 is decoded in Figure 2-2. It is important to start at the beginning of the data stream in order to decode without any error. When the Huffman codes in Figure 2-1 are examined, it can be

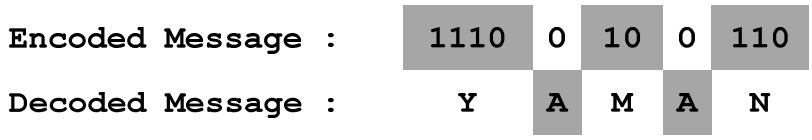

Figure 2-2 Huffman Decoding of a Data Stream Encoded in the Huffman Code of Figure 2-1

observed that all of the codes end with a value of '0' except the last Huffman code which is comprised of only 1's. During decoding every bit of the data stream is examined and it is stored to a register until the value in the register matches with a value in the dictionary. After comparing the value in the register with the Huffman codes in the dictionary, original data is generated.

Due to the word-based memory units, it is difficult to implement Huffman encoding schemes which have variable length of codes. Additional codes should be used for fast and lossless recovery in FPGA implementations [8].

2.2.2 Arithmetic Coding

Arithmetic Coding [10] is another commonly used lossless data compression method. According to Witten et al. [11], arithmetic coding is much better than Huffman coding in many cases. Messages are represented by an interval of real numbers between 0 and 1in arithmetic coding. If a long message is going to be encoded, the interval in which the message lies becomes very small and the number of bits representing that interval becomes increases. For the illustration, assume that the letters used in a text are U, L, A, S, Y, M, and N. Their probabilities are given in Figure 2-3. The range, which is [0, 1), is known by both encoder and decoder of arithmetic coding algorithm. It is distributed according to the probabilities of characters as given in Figure 2-3. There is no priority order of range for the characters. It is randomly distributed over the total range. Assume

Character	Probability	Range
A	0.2	[0,0.2)
L	0.3	[0.2,0.5)
M	0.1	[0.5,0.6)
N	0.2	[0.6, 0.8)
U	0.05	[0.8, 0.85)
S	0.05	[0.85, 0.9)
Y	0.1	[0.9, 1)

Figure 2-3 Probabilities and Ranges of Sample Characters for Arithmetic Coding

that the message "Yaman" is going to be transmitted from the encoder. This encoding process is shown in Figure 2-4. Since the first letter of the message is Y, the encoder starts with narrowing the range to [0.9, 1). The next letter, A, narrows down this range to first one-fifth of it, because the region [0, 0.2) is reserved for A. Proceeding in this way, encoder arrives into the final range, [0.91024, 0.91032).

During decoding, all known about the message is the range [0.91024, 0.91032). Decoder immediately understands that the message starts with the character Y, since it lies in the range [0.9, 1) allocated for Y in Figure 2-3. This clears that the second character is A, since it produces the range [0.9, 0.92), which encloses the range of the original message, [0.91024, 0.91032). The decoder continues in this way to identify the message. The decoder does not need to know the ends of the range set by the encoder. Any number in the range [0.91024, 0.91032) can be used. The main problem decoder faces is that it cannot detect the end of the message. To overcome this drawback a special character may be used which is known to encoder and decoder [11].

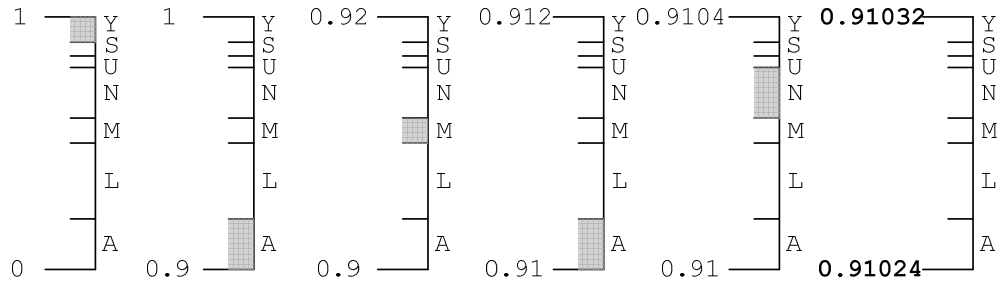

Figure 2-4 Encoding of the message "Yaman" in Arithmetic Coding

2.2.3Golomb Coding

Golomb coding [12] is a lossless data compression paradigm found in the literature. It is not as complex as other compression algorithms but its performance is a bit lower than the previously described data compression methods. The applications of Golomb coding are generally focused on video compression systems such as H.264 video standard [13].

The term group size (*m*) is a very important parameter in Golomb coding. It has a direct effect on the compression ratio. Selecting *m* as the power of two increases the efficiency of coding [12]. After the value of *m* is determined, data set which is to be compressed is divided into subsets of having zeros at the beginning and one at the end. Run lengths of a sample data set are given in Figure 2-5.

Data Set: 00100010100000010010000000010010001111001

Subsets: *001 0001 01 0000001 001 000000001 001 0001 1 1 001*

Run Lengths: 2 3 1 6 2 8 2 3 0 0 2

Figure 2-5 Run Length Determination

Before the encoding process, the codes should be formed. An example of Golomb coding for a group size (m) of 4 is given in Figure 2-6. Codes are created according to the run lengths which are grouped of size m and these groups are named as A_k. Codes appearing on the right of Figure 2-6 have prefixes determined according to their group and tails. Each prefix has (k - *1*) number of ones followed by a zero. Tails are the binary representation of integers starting from zero value to (m - *1*). It is important to note that the widths of the tails must be the same. For instance, if the group size is selected as eight, the widths of the tails must be three which is $log_2(m)$. The code for each run length is formed by adding the prefix and the tail.

Group	Run Length	Prefix	Tail	Code
A_1	0	0	00	000
	1		01	001
	2		10	010
	3		11	011
A_2	4	10	00	1000
	5		01	1001
	6		10	1010
	7		11	1011
A_3	8	110	00	11000
	9		01	11001
	10		10	11010
	11		11	11011

Figure 2-6 Golomb Coding with a Group Size of 4

Encoded version of the data set given in Figure 2-5 can be seen in Figure 2-7. When the resulting code in Figure 2-7 is examined, it can easily be observed that the compression ratio is not that much when compared to other compression algorithms.

```
      Subsets: 001 0001 01 0000001 001 000000001 001 0001   1   1  001
Encoded Data :010  011 001  1010   010   11000   010  011   000 000 010
```

Figure 2-7 Golomb Encoding

2.3 FPGA Implementations of Data Compression Techniques

Lossless data compression, where the original data is extracted without any loss after decompression, applications have increased over the past years due to the need to improve the storage capacity and data transfer rate [7]. There are many examples for the hardware implementations of conventional data compression techniques in the literature. Among these techniques, Huffman [14 - 15], Lempel-Ziv (LZ) [16 - 17], and Golomb [18] compression algorithms are the most popular ones for FPGA implementations.

Rigler et al. [14] implemented Huffman and LZ encoders on an FPGA development board using VHDL in order to serve a basis for hardware implementations of the popular compression program GZIP. Since this implementation is planned to be work with GZIP compression program, while forming the Huffman trees two additional rules are used to make sure that the trees are deterministic. According to the results presented modified Huffman algorithm uses less hardware resources than the LZ algorithm. De Araújo et al. [15] employed a different approach for Huffman Algorithm. They implemented a

microprogrammable controller on the FPGA to perform lossless data compression utilizing Huffman method. They claimed that with this flexible architecture, other compression algorithms such as Arithmetic- and Golomb coding algorithm can easily be implemented.

Abd El Ghany et al. [16] realized the LZ encoding and decoding algorithm on the FPGA. In order to increase the efficiency, they used systolic array approach which resulted in a 40% decrease in the compression rate and 30% decrease in resource usage. Cui [17] also implemented LZ compression algorithm on the FPGA by not using the conventional dictionary approach. The dictionary is divided into smaller units. By this approach, the look-up time is decreased and parallel operations become performable.

Among conventional data compression techniques, hardware implementations of different algorithms for compressing specific data structures are also present in the literature. For instance, Yongming et al. [19] have realized the Linear Approximation Distance Threshold algorithm on FPGA to compress the Electrocardiograph signals. Similarly, Valencia and Plaza [20] developed an FPGA-based data compression technique based on the concept of spectral unmixing to compress hyperspectral data. The novel compression method described in the study can be regarded as task-specific since it is developed to compress the signed integer position command sequences. It may not yield better results for text or image compression.

2.4 FPGA Implementations of Polynomial Approximation Methods

Polynomial approximation is not commonly used in command generation systems due to the inevitable errors in approximations. In order to minimize these types of errors, function approximation algorithms are applied to model the segmented trajectory in a piecewise fashion. The aim of this segmentation is to decrease the error resulting from approximation. In the literature, there exist various

segmentation approaches. The most common one is uniform segmentation, in which the widths of the segments are equal and the number of segments is limited to powers of two. Since the segment widths cannot be customized according to local function characteristics, a huge amount of segments are necessary to fulfill the error requirement [21]. To overcome this problem, dynamic segmentation depending on the inflection points of the trajectory is proposed in this study.

Selecting an appropriate function approximation technique is very important especially in hardware implementations since errors resulting from the approximation should be stored for lossless reconstruction [22]. Another important aspect in approximation is the degree of polynomials used. When memory resources are limited, higher degree polynomials are commonly applied at the expense of increased computational complexity [23].

Hardware implementations of function approximation techniques are frequently used in developing Direct Digital Frequency Synthesizers (DDFS) in the literature. Ashrafi et al. [24] proposed an FPGA-based method that utilizes Chebyshev polynomial series interpolation. The developed technique unifies the results of ROM-less polynomial approximation methods for sinusoidal DDFS implemented on FPGAs [25] [26].Among various approximation techniques, Chebyshev polynomials are usually preferred for hardware implementation. This is due to the fact that Chebyshev polynomials give better results for non-periodic signals that are limited in range. In the study it is also turned out that the performance of Chebyshev polynomials are superior to Legendre, and Bernstein – Bezier polynomials.

2.5 FPGA-Based Command Generation Systems

Implementations of command generation methods on FPGA chips are not very common in the literature due to high computational complexity involved in the methods. Therefore, the techniques employed on FPGA have simplifications

and/or include error compensation modules into the systems. For instance, Su et al. [27] developed a motion command generation chip utilizing FPGA for point-to-point motion applications. They implemented trapezoidal and S-curve motion planning adopting the digital convolution method rather than the complex polynomial type method. With this approach, the computational complexity is significantly decreased. Furthermore, they developed a real-time output pulse compensation algorithm to eliminate the error in the number of output pulses and the results are found to be satisfactory.

Jeon and Kim [28] also used the digital convolution method and designed an FPGA-based acceleration and deceleration circuit for industrial robots and CNC machine tools. Likewise, the method developed by Su et al. [27], they did not use the complex polynomial technique to generate velocity profiles of various acceleration and deceleration characteristics that require extensive computations. Since the current techniques are not satisfactory for generating velocity profiles for industrial robots and CNC machine tools [29], they developed a new method to compensate this deficiency. According to the experimental results given in the paper, they were able to generate unsymmetrical velocity profiles that cannot be generated by digital convolution techniques. Comparing the two works, former one is superior over the latter method. The error in the output pulses is not compensated in Jeon and Kim's study [28] so that the errors are inevitable between the command and response signals. On the other hand, the method proposed in the paper generates commands without any error. Furthermore, it generates position, velocity, and acceleration profiles at the same time.

2.6 Feed-rate Control of CNC Machine Tools

The precision in manufacturing continually improves. In the manufacturing process, the quality of the product is dependent on the functions of CNC machine. Feed-rate control of the machine tool is very important factor for a high-precision CNC machine.

There are various algorithms proposed on feed-rate control in order to increase the surface quality of the product. For instance Cheng et al. [30] employed a predictor-corrector algorithm to estimate the servo command at the next sampling time. In another study of Cheng et al. [31] developed a new interpolator to produce servo commands for real-time control of CNC machining. The main advantage of the proposed interpolator is being capable of generating motion commands for servo controllers at variable feed-rates. In a similar study of Xu et al. [32], they presented variable interpolation schemes for planar implicit curves. They were also able to interpolate in real-time to improve machining efficiency. In the proposed method, the feed-rate is set by the operator according to geometrical state of the surface. In other words, it is decreased when the tool is machining curved parts and increased on planar surfaces.

2.7 Open Research Areas

During the literature survey, not only the current research efforts, but also unexplored areas in the field have been surveyed. With the help of this study, the scope of the thesis has been determined.

First of all, the tool location data are not directly generated in conventional command generation systems (CAD/CAM + CNC processors). Instead, the Numerical Control (NC) code, which represents the trajectory via an ensemble of geometric entities (such as line, arc, helix, NURBS etc.), is parsed and interpreted in real-time to produce the position of the tool accordingly. The reason for this rather meandering approach is due to the fact that data storage and retrieval was extremely costly in the past. With this limitation is greatly circumvented in today's technology, the tool location data may directly be processed to increase the performance of real-time command generation systems. With this approach, the use of post processors, (machine-depended) NC codes, and complex hardware for real-time data interpolation may be eliminated.

Secondly, the real-time data decompression techniques, which may further alleviate the efficiency of this direct command data generation method, are not fully explored in the literature. For instance, there is no FPGA implementation of Arithmetic coding (regardless of the application area) since it is very difficult to synthesize a digital circuitry that is realizing the algorithm via hardware description languages such as Verilog and VHDL. According to the results obtained in this thesis, the performance of Arithmetic coding is much better than any other compression algorithms when applied to the context of command generation. Thus, the FPGA implementation of Arithmetic coding in command generation methods might be a remarkable contribution.

CHAPTER 3

FPGA INTERFACE

3.1 Introduction

The FPGA Interface to be devised within the scope of the thesis is a board whose port connections can be defined through software and is capable of connecting various sensors and actuators (in any order) to the FPGA based motion control system. The current version of the interface can be regarded as a prototype and it is planned to finalize the interface as a product in the future.

When the devices and their outputs signal used in motion control applications are considered, it is concluded that the interface (as in the data acquisition cards) should have four main communication channels:

1) **Analog Input:** This card of the interface is responsible for the connection of analog devices (sensors, drivers, converters, etc.) to the system. The voltage ranges of these type of devices are

    a. Unipolar 5V $\in$ [0, 5] V

    b. Bipolar 5V $\in$ [-5, 5] V

    c. Unipolar 10V $\in$ [0, 5] V

    d. Bipolar 10V $\in$ [-10, 10] V

2) **Analog Output:** This developed unit is to be used for electromechanical systems and controlling motor drives. The output voltage ranges of this card are the same with the Analog Input card.

3) **Digital Input:** Devices (switches, sensors, etc.) generating digital signals are connected to the motion control system via this card. For generality, the developed card should be compatible with the below digital logic families:

   a. TTL, LS-TTL

   b. CMOS (ACT, HCT, 74C)

4) **Digital Output:** This interface is basically used to generate digital signals in order to sustain digital communication with various devices. The outputs of this card should be compatible with the logic families given above.

The microcontrollers and Digital Signal Processors (DSPs) employed in the motion control industry have internal Analog to Digital Converters (ADCs) and Digital to Analog Converters (DACs) but the FPGA chips have neither of them. Another issue that should be noted is that the most of the FPGA chips operate with 3.3V CMOS/ACT logic families. Thus, in order to evaluate the aforementioned signals there should be signal-converter circuits that can be connected to the digital ports of the FPGA chip on the interface.

The interface shown in Figure 3-2 basically consists of a main board having eight identical slots reserved for four types of daughter cards. The daughter cards are Analog Input, Analog Output, Digital Input, and Digital Output Cards. These boards are to be evaluated in the following subsections in a detailed manner.

When the literature is investigated, there exists no full device that can be utilized with analog and digital peripheral devices having various voltage ranges. The

most related works include the programmability of FPGA pins according to the needs of the users and the voltage level converters.

Menon et al. [41] developed a programmable input/output unit composed of three input and one output circuits. These four circuits are coupled to one of the FPGA pins and enabling of these circuits is carried by the programmable bits. With this unit a selected pin of the FPGA can accept TTL, GTL, GTLP, LVPECL, or LVDS voltage levels as input and generate TTL, GTL, or GTLP compatible signals. Goetting et al. [42] designed a similar system, but they implemented the structure within the FPGA chip.

Chang [43] introduced the Application Specific FPGA phrase to the literature in his study. These integrated circuits include at least two functional units, which can be a digital to analog converter, a compressed image decoder, a random access memory, etc. The main purpose of the FPGA chip used in this design is to maintain the communication between the functional units and to connect them.

3.2 Mother Board

The main responsibility of the mother board is to host different types of daughter cards and provide required voltages and signals to these boards. Schematic design of the mother board shown in Figure 3-1 is drawn in Proteus 7 Professional [44]. As can be seen from the figure, the main board has three main functions: **i)** to generate bipolar (reference) triangular waveform, **ii)** to produce reference voltages, **iii)** to supply different voltage ranges to the daughter cards.

Bipolar triangular waveform generator output is used by the Analog Input Cards (AIC) to compare the analog signals with the generated waveform. Triangular waveform generated by the module can be seen in Figure 3-7. The circuit implements the general triangular waveform generator described in [45]. The generated signal should be exactly in between 0V and 3.3V for proper analog to digital conversion. To eliminate the small deviations from the desired voltage

levels the resistances (R5 and R6) used in the corresponding circuitry should perfectly be matched. The frequency of the waveform is set by the resistance R6 and the capacitance C2 used in the design, which is about 40 kHz in this case.

On the other hand, the reference voltage generator outputs a constant voltage of 1.65V. This voltage is obtained from the resistance R2 which is serially connected the resistance R1. The capacitance C1 is used to eliminate the noise. The operational amplifier U1 is a voltage follower used not to let the board draw current from the voltage divider. If the voltage follower is not used, during the operation of the board reference voltage can oscillate.

Electrical modules on the upper of the figure are the indicators of available voltage levels supplied by the power source. They also distribute these voltages to the daughter cards. J1, J2, J3, and J4 are connected to 3.3V, 5V, V_{DD}, and 15V voltage supplies, respectively. The capacitances between C3 and C11 are placed to eliminate the noise on the voltage resources. Light Emitting Diodes (LEDs) are also used to indicate the availability of the voltage source on the board.

As can be seen from Figure 3-2, the main board is capable of holding only eight daughter boards. The connectors placed on the left side of the board are used to supply 3.3, 5, 12, and -12 voltages. On the other hand, the connectors on the right side are used to connect peripheral devices to the system. FPGA pins are connected to the headers on the middle of the connectors on the right side.

In Figure 3-3, Altera DE1 FPGA Development Board, designed interface board, and the power supply are shown with their connections. According to the number of pins used for one mother board, FPGA Development Board can support up to four mother boards. This means that thirty-two daughter cards can be placed on the main boards, so that with this set-up an eight axis system can easily be controlled.

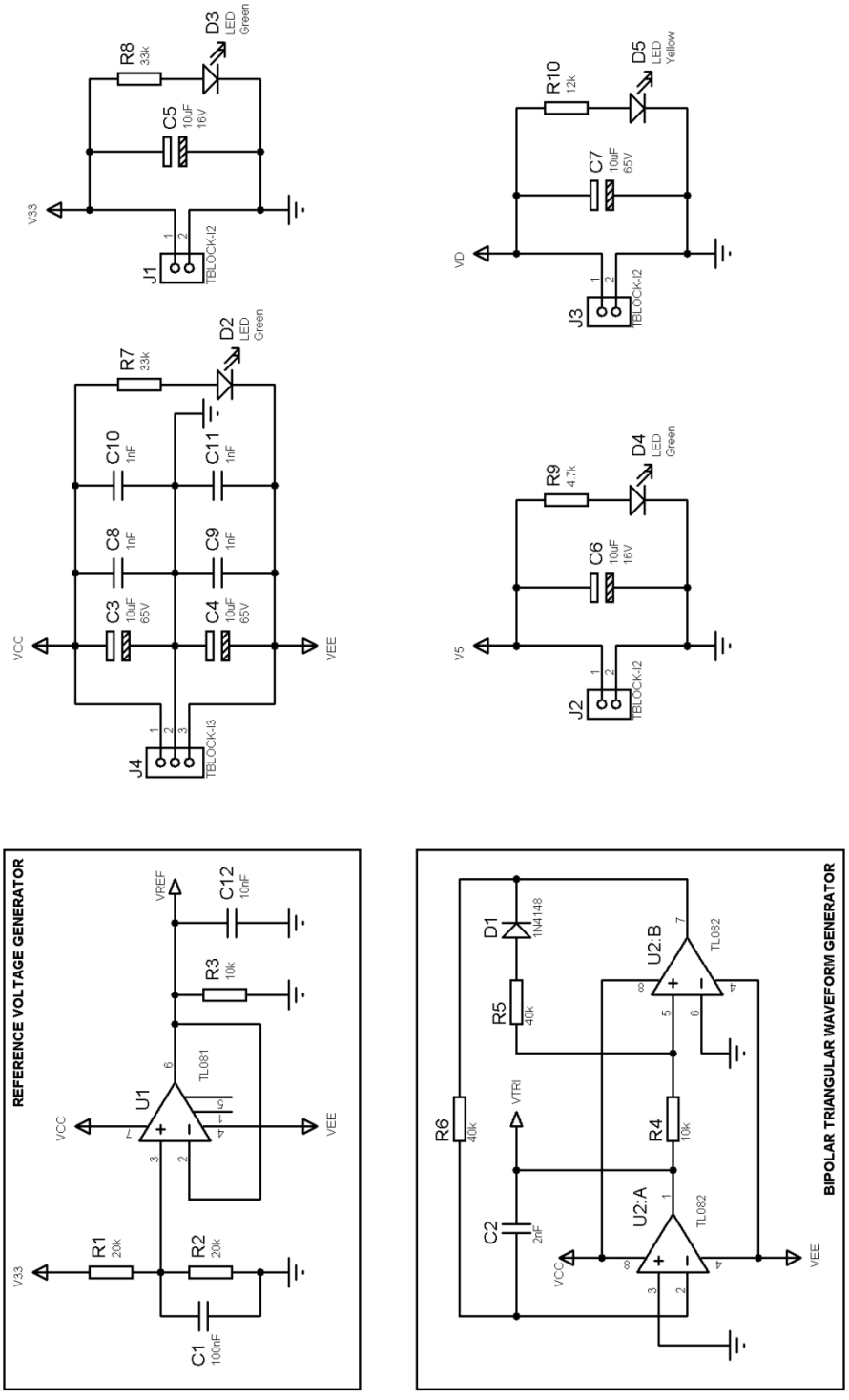

Figure 3-1 Schematic Design of the Mother Board

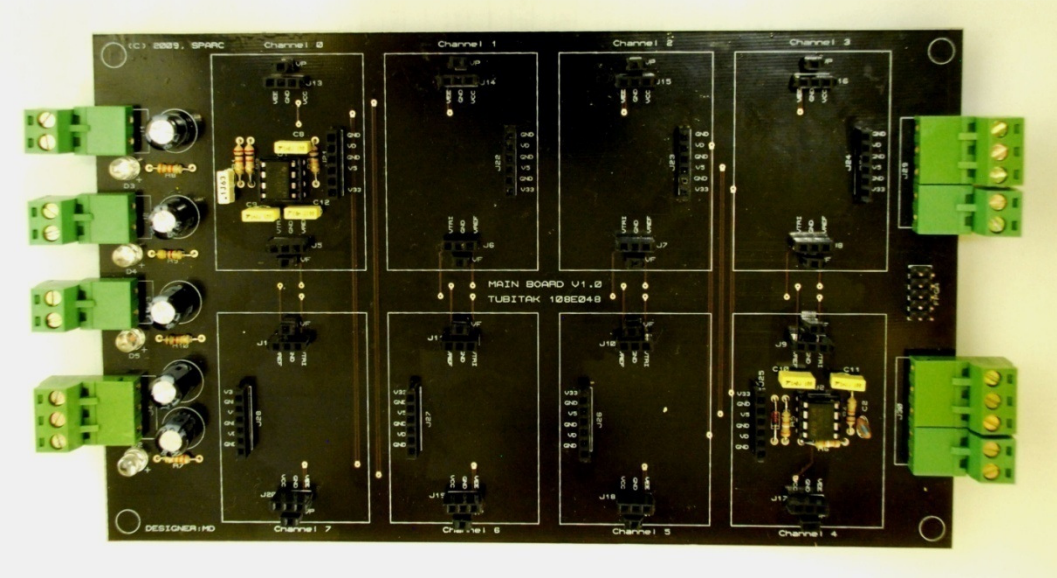

Figure 3-2 Main Board

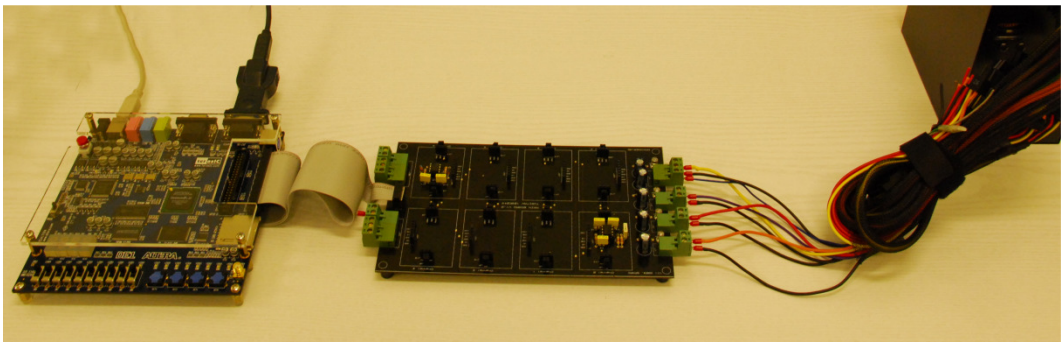

Figure 3-3 FPGA Development Board and Its Interface

3.3 Analog Input Card

The main task of this card is to convert the signals of analog sensors used in the motion control industry to digital signals utilizing Pulse Width Modulation (PWM). According to the block diagram given in Figure 3-4, the input analog signal is first amplified and biased according to the range selection done by the user.

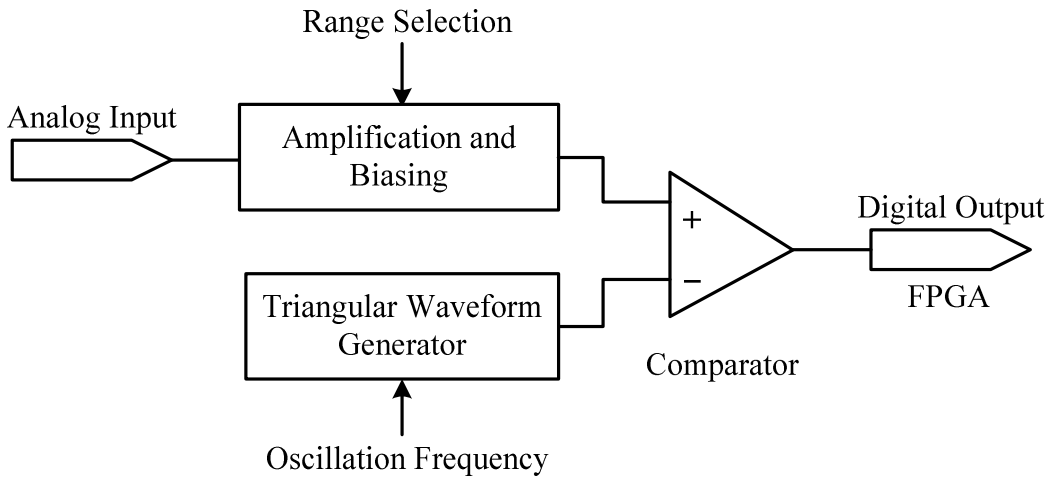

Figure 3-4 Block Diagram of Analog Input Card

The modified input signal is then compared with the triangular waveform on the mother board and the digital output is fed to the corresponding pin of the FPGA chip for further computations.

The analog input card, whose circuit schematic is illustrated in Figure 3-5, is compatible with various voltage ranges as discussed in Section 1. The conversion of the voltage levels are carried out with a dipswitch and a two way jumper placed on the card. With this approach the voltage range of the sensor is to be determined. Voltage ranges are obtained according to the position of the switches and the jumper are given in Table 3-1.

The most important part of the design is placed on the upper part of Figure 3-5. The operational amplifier U1:A is used to modify the range of input signal. The output voltage of U1: A can be defined as

$$V_1 = \left(-\frac{R_1}{R_4/2}\right) V_p \qquad (0 \le V_p \le 5V) \tag{3.1}$$

$$V_1 = \left(-\frac{R_1}{R_4}\right) V_p \qquad (0 \le V_p \le 10V) \tag{3.2}$$

$$V_1 = \left(-\frac{R_1}{R_4}\right) V_p \qquad (-5 \le V_p \le 5V) \tag{3.3}$$

$$V_1 = \left(-\frac{R_1/2}{R_4}\right) V_p \qquad (-10 \le V_p \le 10V) \tag{3.4}$$

Then the output signal of U1:A is shifted according to the polarity of the input port signal by U1:B. For the unipolar cases, the resistance R3 is connected parallel to the resistance and for bipolar cases it is not connected to the circuit.

After the modification of the voltage range is completed, the modified input signal is compared with the triangular waveform generated from the mother board utilizing a comparator chip (U2:A) and its output is fed to the FPGA chip for further analysis of the signal. The vacant pins of the chip U2 are connected to ground not to cause any problems.

The manufactured card in Figure 3-6 is tested by placing it on one of the slots available on the mother board as shown in Figure 3-3. After all connections are done, by using a function generator, analog signals having different frequencies and magnitudes are fed to the designed analog input card to evaluate the performance of it. In Figure 3-7, signals obtained using the analog input card are shown. As described above, firstly the input analog signal (green) is scaled to the[0, 3.3] voltage range (blue). Then this signal is compared with the triangular waveform and a pulse sequence changing according to the amplitude of the input signal is generated by the card. Hence, it will be possible for the FGPA to determine the value of input signal by measuring the widths of the pulses.

Table 3-1 Voltage Levels

First Switch	Second Switch	Jumper	Voltage Range
1	0	Unipolar	[0, 5] V
1	1	Unipolar	[0, 10] V
0	0	Bipolar	[-5, 5] V
0	1	Bipolar	[-10, 10] V

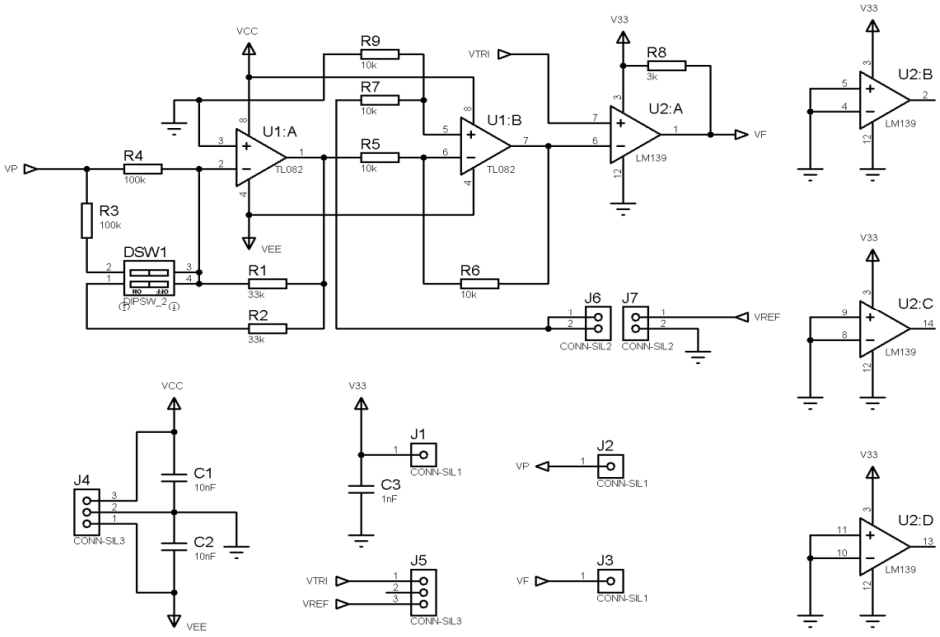

Figure 3-5 Schematic Design of the Analog Input Card

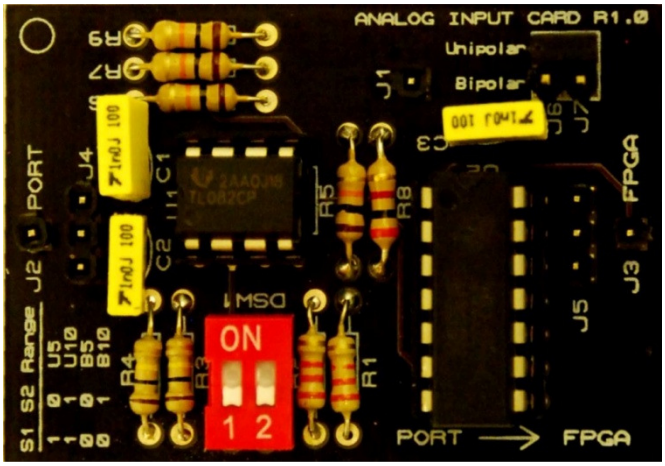

Figure 3-6 Analog Input Card

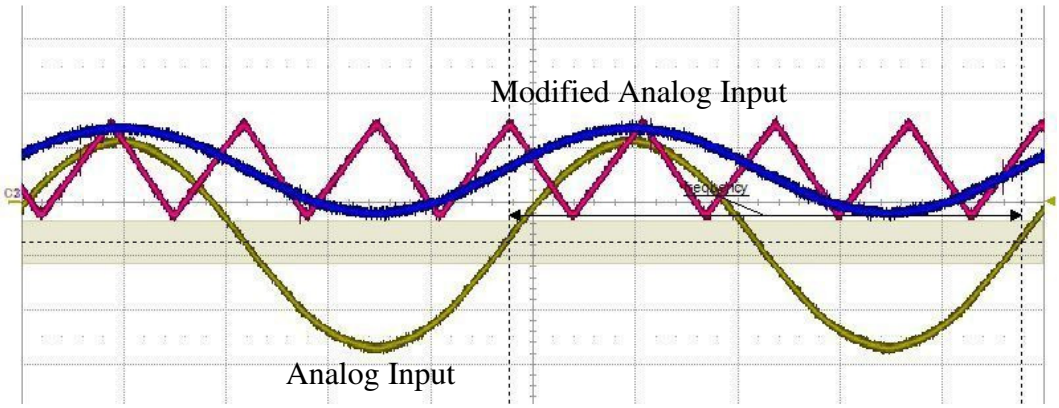

Figure 3-7 Signals of the Analog Input Card

Analog signals having different frequencies (100 Hz, 1 kHz, 10 kHz) are fed to the card and the resulting signals are shown in Figure 3-8, Figure 3-9, and Figure 3-10. It can be inferred from the results that at 40 kHz frequency pulse width modulation is successfully performed. It should also be noted that the resolution of this analog to digital converter is about 10 bits due to the 50 Mhz clock used on the FPGA board.

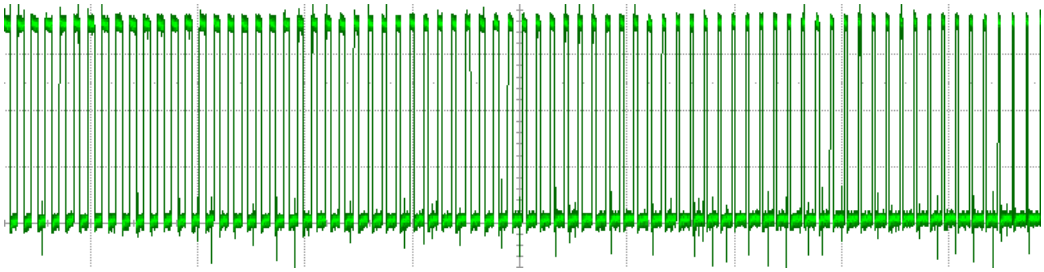

Figure 3-8 Output Signal of the Analog Input Card at 100 Hz

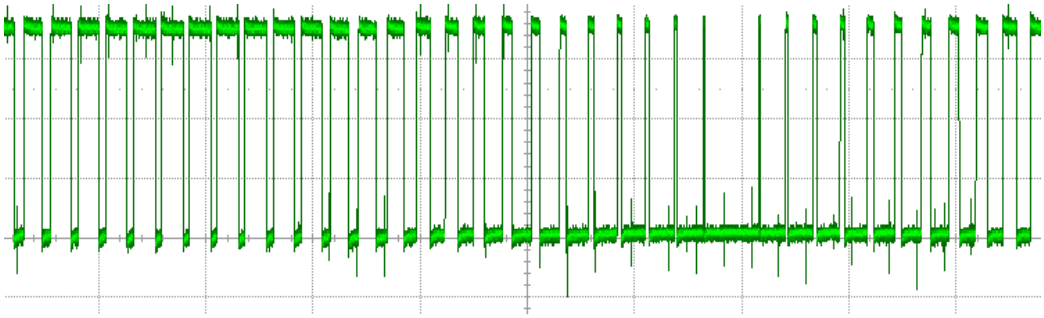

Figure 3-9 Output Signal of the Analog Input Card at 1 kHz

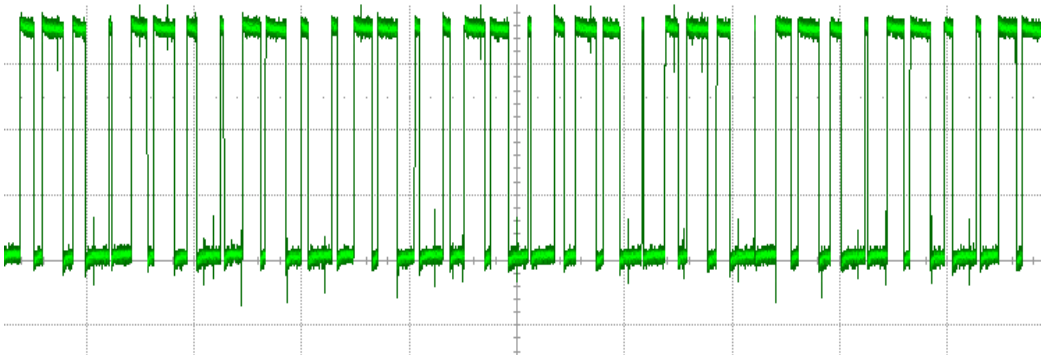

Figure 3-10 Output Signal of the Analog Input Card at 10 kHz

3.4 Analog Output Card

The main function of this card, whose schematic shown in Figure 3-12, is to generate analog signals that are necessary to control motor drives and electromagnetic devices. The voltage ranges of the output signals are modified in a similar fashion described in the previous section. The positions of the switches and the jumper are the same to obtain voltage ranges as given in Table 3-1.

The block diagram of the analog output card is given in Figure 3-11. Firstly, the generated PWM signals from the FPGA development board are transferred to the low-pass filter. According to the cut-off frequency of the filter, digital signals are converted to their analog counterparts. Then these analog signals are amplified and biased according to the voltage range selections of the user.

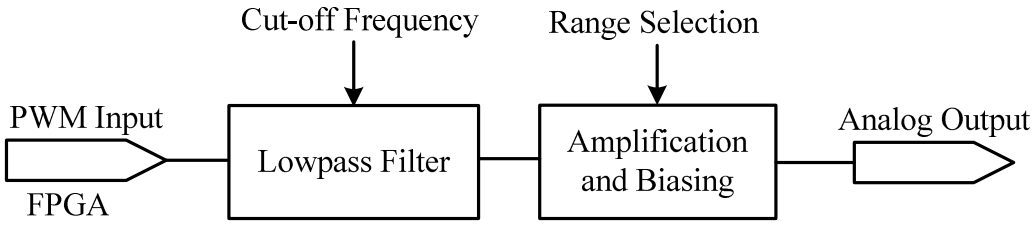

Figure 3-11 Block Diagram of Analog Output Card

As can be seen from Figure 3-12, the incoming PWM signal from the FPGA pin is fed to an active low pass filter (R1 and C1) having a cut-off frequency of 5 kHz. Then the signal is shifted according to the polarity selection by the jumper which is placed on J5 and J6 by the help of U1:B. In the final part of the design another operational amplifier (U1:C) is used to scale the output signal according to the following expression:

$$V_P = \left(-\frac{R_6/2}{R_7}\right) V_9 \qquad (0 \le V_p \le 5V) \tag{3.5}$$

$$V_P = \left(-\frac{R_6}{R_7}\right) V_9 \qquad (0 \le V_p \le 10V) \tag{3.6}$$

$$V_P = \left(-\frac{R_6}{R_7}\right) V_9 \qquad (-5 \le V_p \le 5V) \tag{3.7}$$

$$V_P = \left(-\frac{R_6}{R_7/2}\right) V_9 \qquad (-10 \le V_p \le 10V) \tag{3.8}$$

In order to evaluate the performance of the manufactured card in Figure 3-13, a universal sinusoidal signal generator is designed in Quartus II 9 Web Edition software. Its schematic design is shown in Figure 3-14. There are three different modules in the design, namely Clock Divisor (CD), Sine Wave (SW), and PWM Generator (PWMG). SW module can be regarded as the core of this design, since the other two modules are in communication with the SW module. It reads discrete values of a quarter sine wave from a look-up table and sends these values in an alternating manner to complete a full sine wave to the PWMG module. The PWMG module simply generates the output PWM signal according to the incoming value from the SW module. The main task of the CD is to divide the global clock according to the user inputs and pass it to the SW and PWMG modules. With the switches available on the Altera DE1 Development Board, it is possible to generate sinusoidal signals ranging between 90 Hz and 12 MHz. By using this design, sinusoidal signals with frequencies of 100 Hz, 1 kHz, and 4 kHz are generated and fed to the analog output card to observe the performance of it.

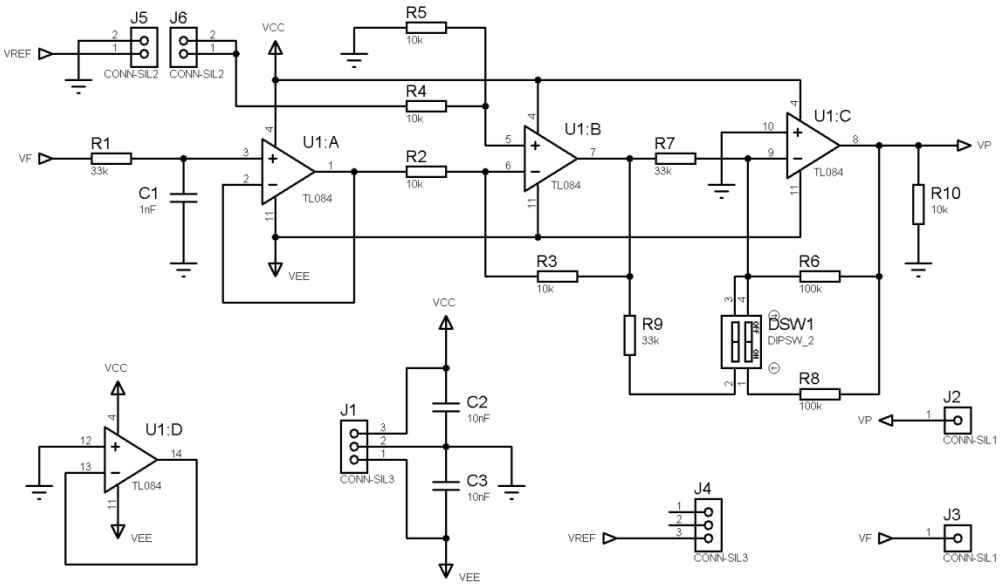

Figure 3-12 Schematic Design of the Analog Output Card

When the results shown in Figure 3-15, Figure 3-16, and Figure 3-17 are considered, it can be concluded that the card works properly at low and intermediate frequencies. On the other hand, it is observed during the tests that at frequencies higher than 4 kHz the magnitude and the frequency of the output signal are deteriorated. This is due to the fact that there is a low pass filter whose cut-off frequency is 5 kHz in the design. Thus, the cut-off frequency of the filter should be determined according to the needs of the application.

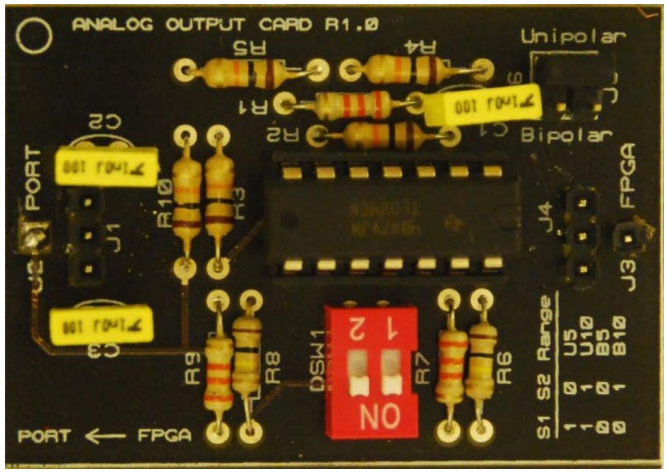

Figure 3-13 Analog Output Card

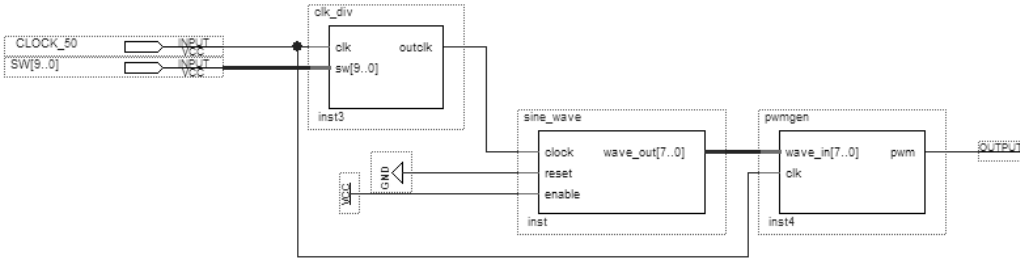

Figure 3-14 Hardwired FPGA Implementation of the Sinusoidal Signal Generator

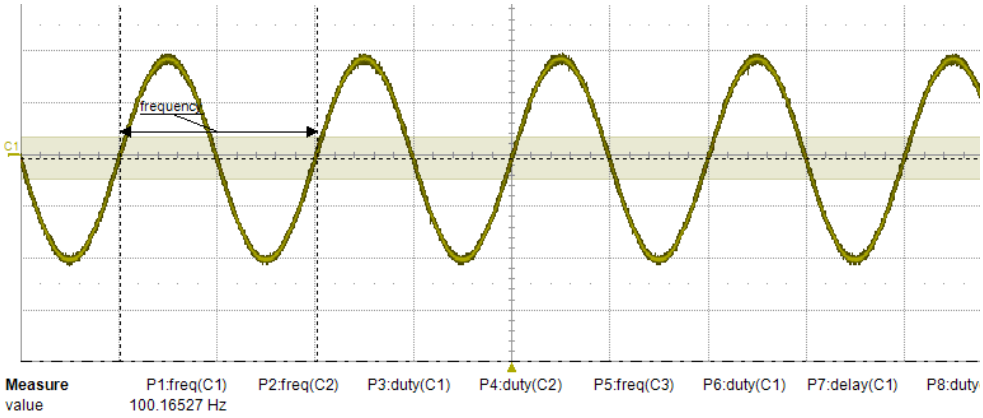

Figure 3-15 Output Signal of the Analog Output Card at 100 Hz

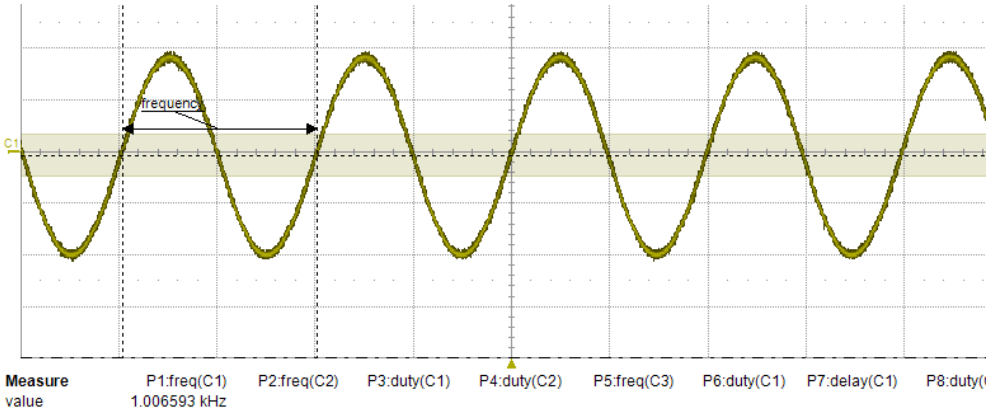

Figure 3-16 Output Signal of the Analog Output Card at 1 kHz

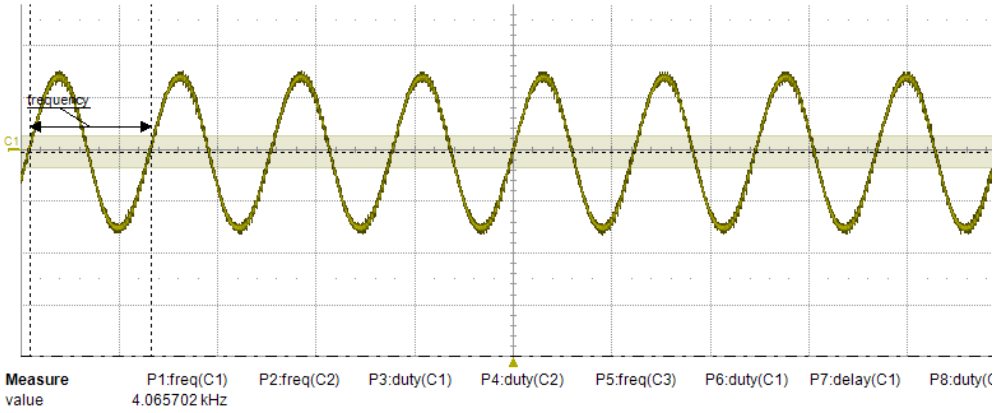

Figure 3-17 Output Signal of the Analog Output Card at 4 kHz

3.5 Digital Input Card

The digital input card, whose schematic drawing is provided in Figure 3-18, is basically used to convert digital signals belonging to various logic families such as TTL, LS-TTL, and CMOS to 3.3V CMOS based compatible inputs for FPGA chips. The main components of the design are the two Bipolar Junction Transistors (BJTs). Voltage level shifting is carried out with the help of these high speed (600 MHz) transistors coded as 2N2369. Firstly the digital input is fed to the base of the first transistor and the collector of this transistor is connected to the

base of the second transistor. The emitters of the transistors are directly connected to ground and collectors are fed from the 3.3V supply. With this design digital inputs at various logic levels are properly converted to FPGA compatible range. In the literature the circuit is known as the totem pole output circuit. In order to evaluate the performance of the digital input card, square signals (TTL) at different frequencies (1 kHz, 10 kHz, and 100 kHz) are fed to the manufactured card shown in Figure 3-19 via function generator. According to the results given in Figure 3-20, Figure 3-21, and Figure 3-22, at low and intermediate frequencies there is no remarkable change in the form of output signal whose range is [0, 3.3] V. On the other hand, as the frequency of the input digital signal increases, the rise time of the output signal tends to increase. The fundamental reason of this problem is that there exist undesirable capacitances at connection points and between electrical routes. Although it seems that the performance of the card is sufficient for many industrial applications, some improvement should be done on the card to let it operate properly also at high frequencies.

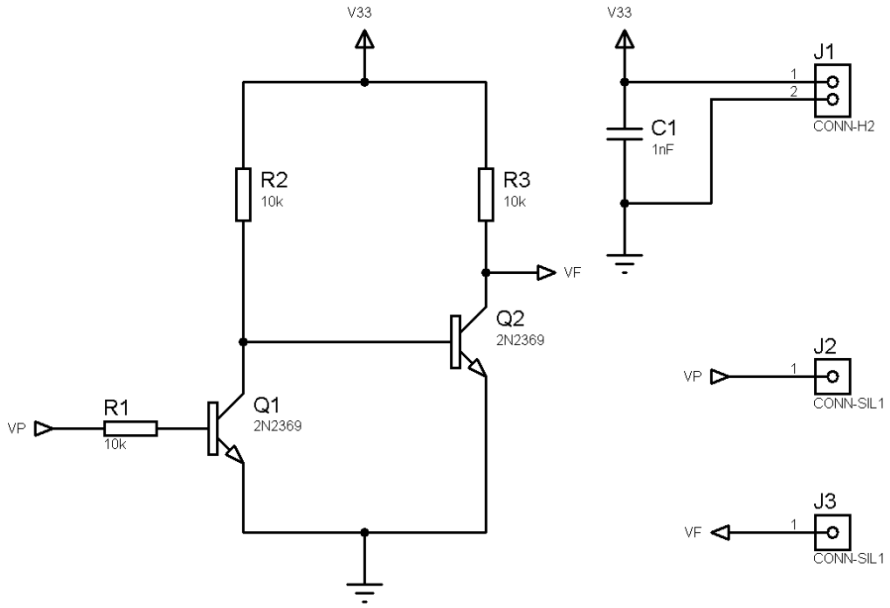

Figure 3-18 Schematic Design of the Digital Input Card

Figure 3-19 Digital Input Card

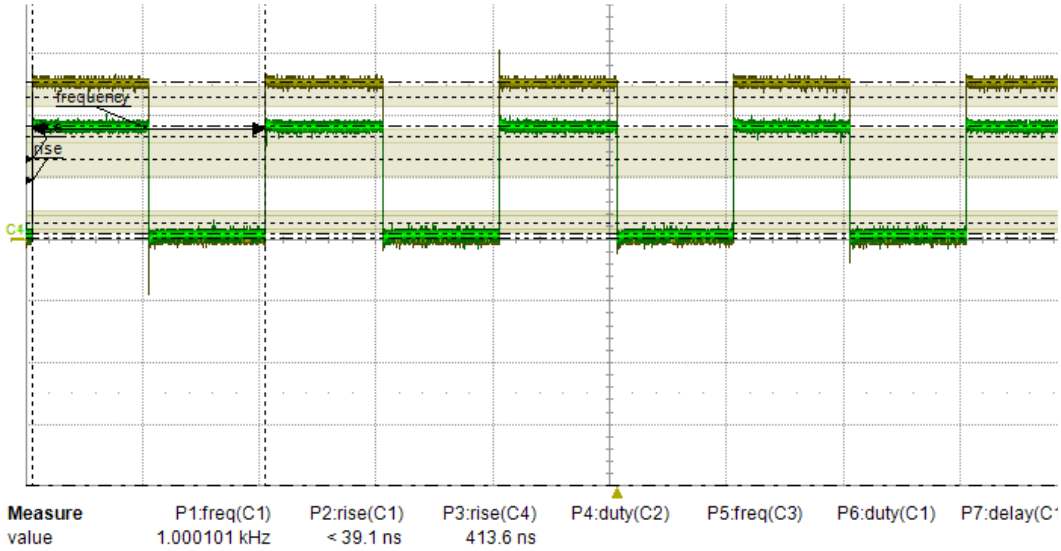

Figure 3-20 Output Signal of the Digital Input Card at 1 kHz

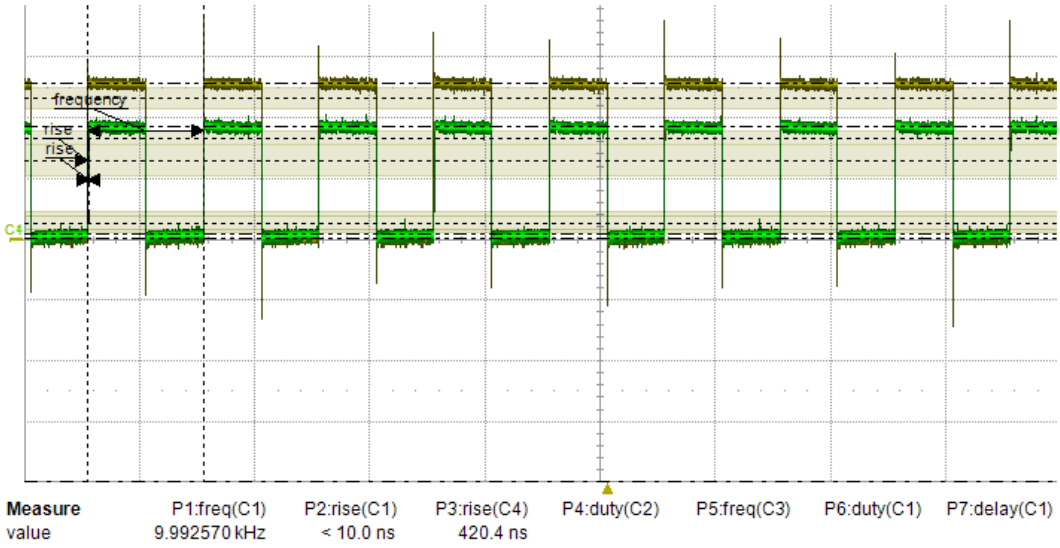

Figure 3-21 Output Signal of the Digital Input Card at 10 kHz

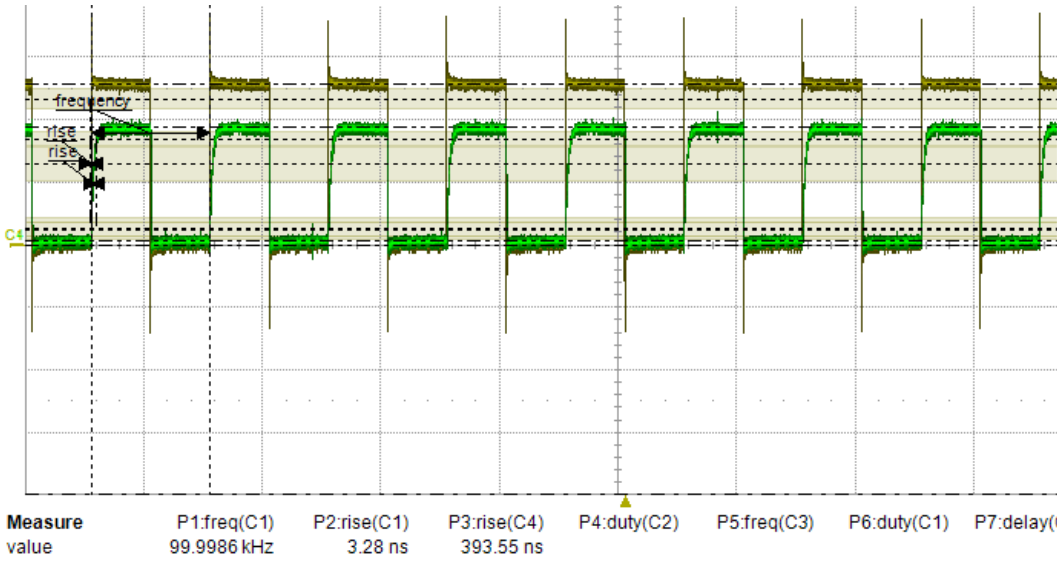

Figure 3-22 Output Signal of the Digital Input Card at 100 kHz

3.6 Digital Output Card

The digital output card, whose circuit schematic is shown in Figure 3-23, is basically used to generate digital signals to maintain the communication between the FPGA chip and other devices. The output signals of this card should be compatible with the logic families mentioned in the previous section. The output voltage range of this card is also changed with a jumper as in the other cards. When the schematic design is considered, the applied method is similar to the one in the digital input card. Two BJTs are used and this time the output voltage level is connected to the collector legs of the transistors.

In order to test the digital output card in Figure 3-24, square signals at high frequencies are generated utilizing the FPGA board and fed to the card. The response of the digital output card to the square signal with a frequency of 300 kHz is shown in Figure 3-25. As the case in the evaluation of the digital input card, the rise time of the output signal is much higher than the rise time of the input signal. There is almost no difference between the fall times. For the illustration of the effect of duty cycle on the rise time, signals at different duty cycles are fed to the card and the results are shown in Figure 3-26. As can be inferred from the figure, there occurs delays in the output signals due to the capacitive effects. Although these delays are acceptable for many applications, improvements should be made to decrease these types of delays.

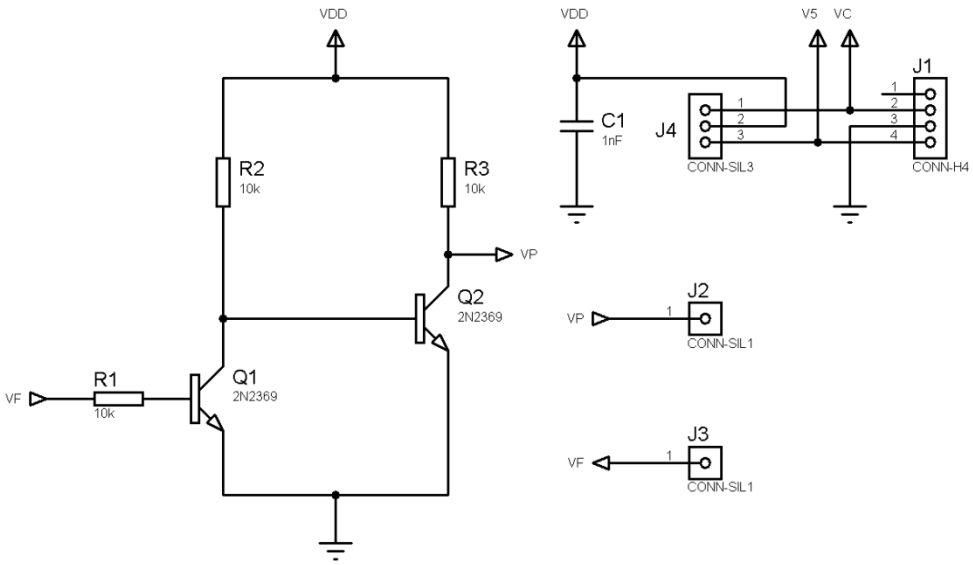

Figure 3-23 Schematic Design of the Digital Output Card

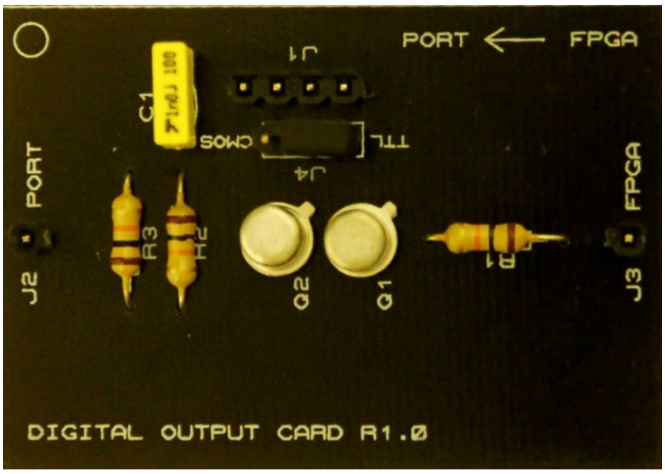

Figure 3-24 Digital Output Card

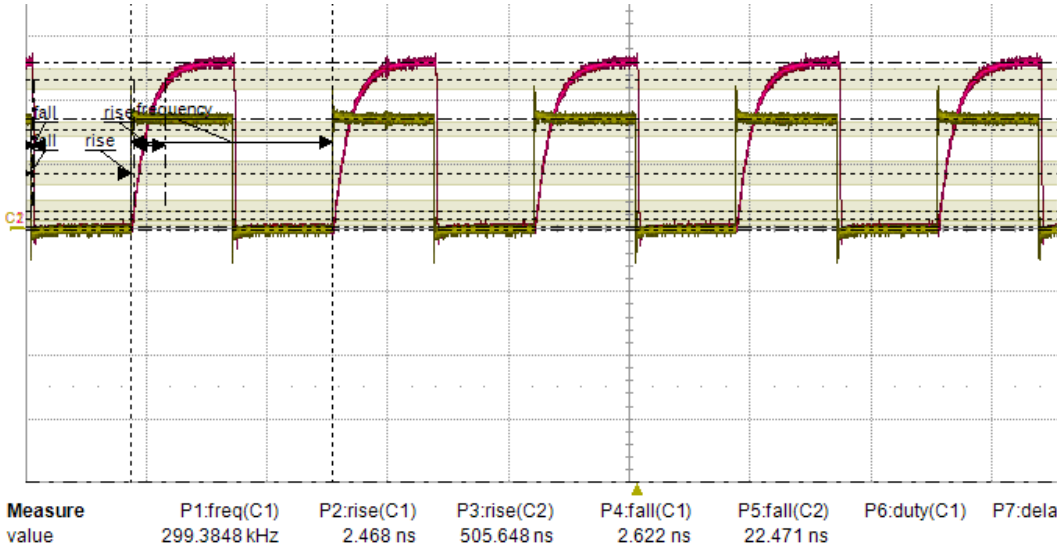

Figure 3-25 Output Signal of the Digital Output Card at 300 kHz

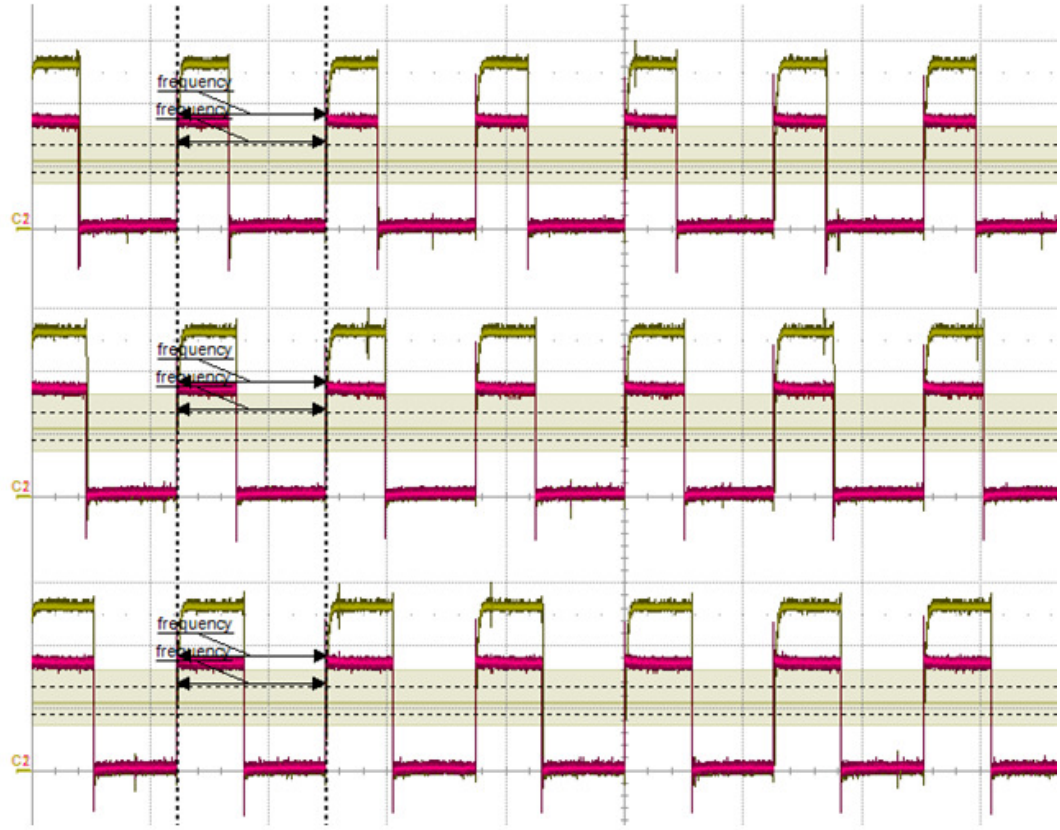

Figure 3-26 Output Signal of the Digital Output Card for Various Duty Cycles

3.7 Closure

In this chapter of the thesis, the developed FPGA interface is elaborated in a detail manner. This interface can be used to connect various peripheral devices to FPGA without any electronic concerns. When the overall design is considered, the modifications on the interface are carried out by hand utilizing the auxiliary switches and jumpers on the daughter cards. In the further designs of the interface, it is planned to replace switches with fast analog switches that can transfer the current in two ways. These analog switches may also be used to configure the channels electronically.

CHAPTER 4

COMMAND GENERATION METHOD UTILIZING SEGMENTATION AND POLYNOMIAL APPROXIMATION

The first developed command generation method utilizes segmentation with respect to the inflection points of the trajectory and then employs function approximation paradigms [33] such as, Chebyshev, Legendre, Bernstein – Bezier, etc. to represent the continuous trajectory efficiently. In this chapter, after the importance of segmentation is discussed, some background information on polynomials is given. According to the performance evaluation of polynomial types in MATLAB, the most successful one is realized on the FPGA Development Board via two different approaches, hardwired and embedded softcore processor.

4.1 Segmentation

Segmentation is preferable when a complex trajectory is to be approximated since a single polynomial (with extremely high order) might not be sufficient to approximate the whole trajectory to the desired accuracy. When the trajectories are segmented before approximation, the magnitude of errors decreases remarkably. On the other hand, segmentation also brings additional computational loads to the approximation methods. Hence, during the evaluation

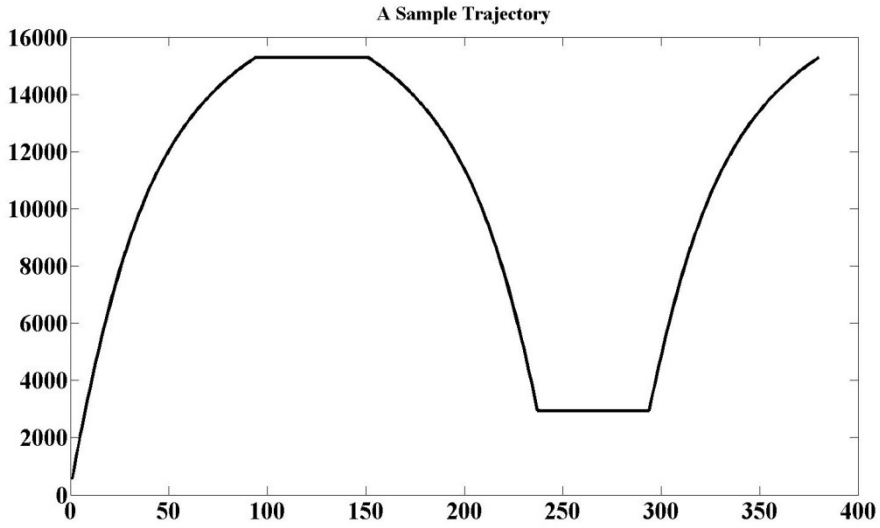

Figure 4-1 A Sample Trajectory

of methods, this extra effort should be also assessed. For illustration, a sample trajectory given in Figure 4-1 is to be approximated with Chebyshev Polynomials (CPs) with two different approaches. In the first approach, the trajectory is directly approximated by ten CPs. In the second approach, it is first divided into sections considering to the inflection points and thus each section is approximated with the same number of CPs.

In MATLAB, the given trajectory is approximated and the resulting sequence in Figure 4-2 is obtained. As can be seen from the figure, the approximated trajectory has significant deviations if compared to the original one. Similarly, the approximation error is demonstrated in Figure 4-3. When it is compared with the original trajectory in Figure 4-1, one can easily perceive that at the inflection points of the original trajectory, the magnitudes of the errors are much greater.

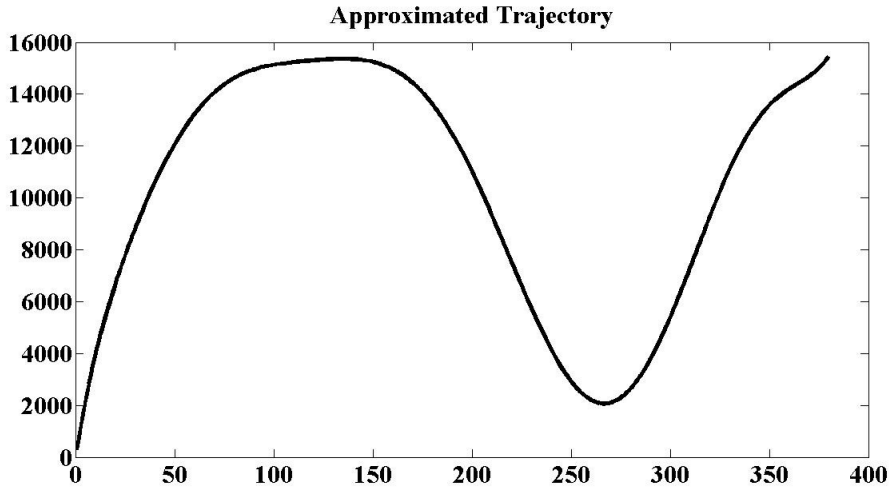

Figure 4-2 Approximation without Segmentation

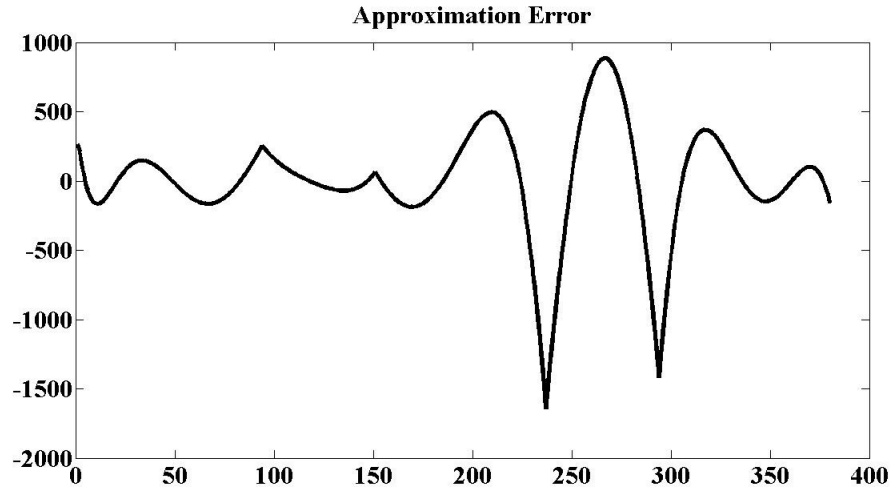

Figure 4-3 Approximation Error without Segmentation

There are two possible ways to decrease the magnitude of errors at these points: **i)** number of polynomials used for approximation can be increased, **ii)** the trajectory can be divided into sections. When the hardware implementations are considered, the former solution is not preferred due to the resulting computational burden. Thus, the sample trajectory is divided into five segments according to the inflection points shown in Figure 4-4. Each segment is approximated utilizing ten CPs. As can be seen from Figure 4-5, the segmentation before approximation gives much better results than the previous method. The error profile is plotted in Figure 4-6 and it is compared to that of its counterpart. Note that the maximum error in Figure 4-6 is about one-sixth of the maximum error in Figure 4-3. Hence, the segmentation before approximation is most feasible approach to approximate curves with C^0 continuity.

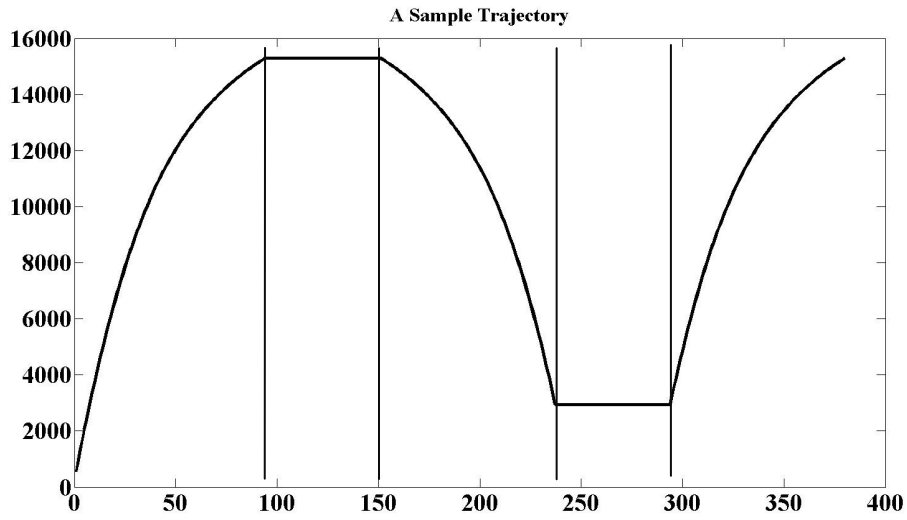

Figure 4-4 Segmented Sample Trajectory

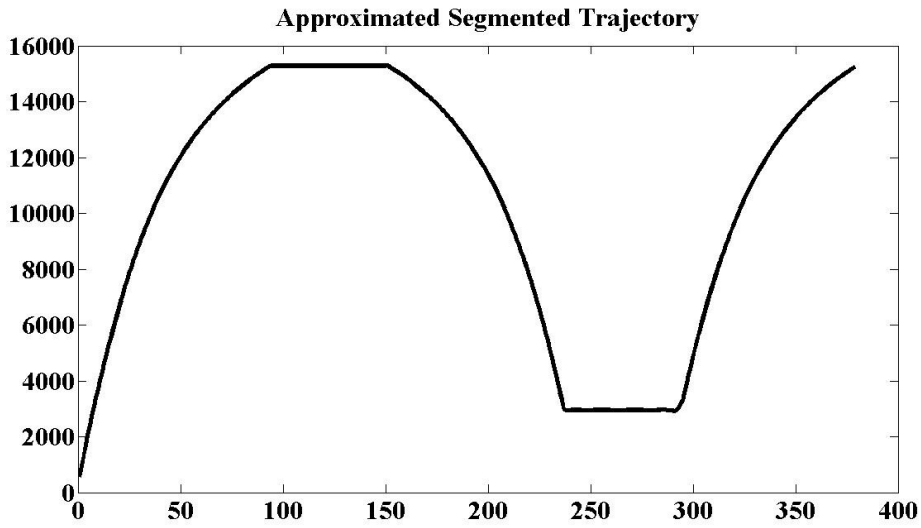

Figure 4-5 Approximated Segmented Trajectory

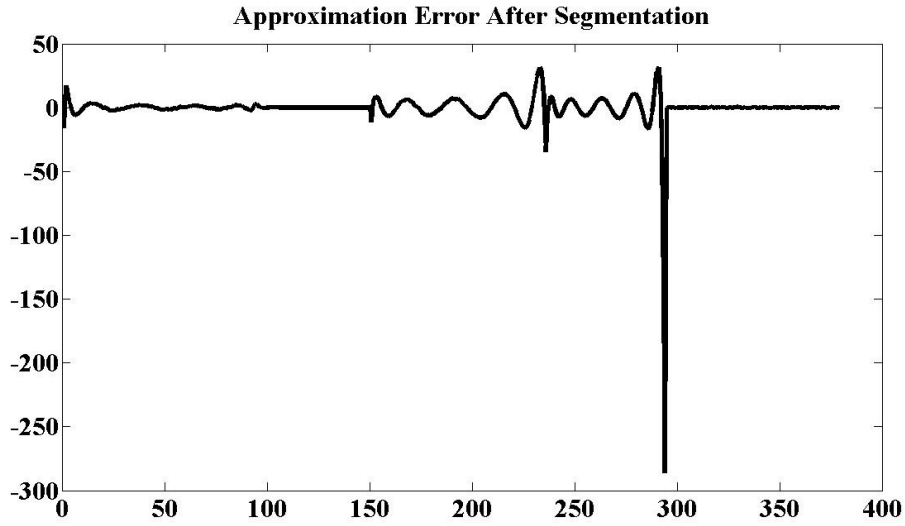

Figure 4-6 Approximation Error After Segmentation

4.2 Polynomial Techniques

In this proposed command generation method, various polynomial approximation techniques are applied onto the segmented trajectory. Before evaluating the different types of polynomials, some background information on such methods will be given to maintain self-containment of this thesis.

Suppose that a command trajectory is to be approximated with an n^{th} order polynomial in the interval $[x_{min}, x_{max}]$:

$$y(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n \tag{4.1}$$

If there exists sufficient ($m \geq n$) number of data points $\{(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$ on the trajectory, the unknown polynomial coefficients $\{a_0, a_1, \ldots, a_n\}$ in (4.1) can be calculated to represent the trajectory. In cases where the number of data points is equal or greater than the number of polynomial coefficients, the remaining coefficients can be determined using the least squares. That is, with these coefficients, (m+1) equations can be written:

$$Y = X \cdot A \tag{4.2}$$

$$A = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \tag{4.3}$$

$$X = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix} \tag{4.4}$$

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{4.5}$$

Polynomial coefficients in (4.2) are determined using the pseudo-inverse method described in [34]:

$$A = (X^T X\ )^{-1} X^T Y \tag{4.2}$$

In polynomial approximation methods, the basis functions are the key factors to approximate the functions more accurately. When (4.1) is considered, the exponential functions $\{x^1, x^2, \ldots, x^{n-1}, x^n\}$ can be treated as the basis functions. This natural selection of basis function does not result in a good representation due to the fact that the employed basis functions are not mutually orthogonal. That is,

$$\int_{x_{min}}^{x_{max}} x^i x^j dt \neq 0 \tag{4.7}$$

where $i, j \in \{0, 1, \ldots, n\}$ and $i \neq j$. In approximations, it is a wise choice to use orthogonal functional forms as the basis functions according to the characteristics of the function to be approximated. On the other hand, these selected basis functions should easily be calculated also converge to the solution with less error [35].

In this polynomial approximation based command generation method, Chebyshev, Legendre, and Bernstein polynomials are elaborated and most suitable one is used in FPGA implementations. When the basis functions of these polynomials are compared, Legendre and Chebyshev polynomials use cosine functions and Bernstein polynomials use binomials.

4.2.1 Chebyshev Polynomials

Chebyshev polynomials (CPs) are usually preferred for hardware implementation, since the CPs approximate non-periodic signals within a limited range better than other polynomial types. This feature is well suited for the commanded trajectories encountered in motion control applications.

CPs, which are strictly defined over an interval x ∈ [-1, 1], are formed recursively to yield a set of orthogonal polynomials. When a different interval is considered, a change of variables is employed to be able to utilize the CPs. In approximation theory, the CPs are regarded as important polynomials due to the fact that roots of first-kind CPs are used as the nodes in polynomial interpolation. As a result, CPs decreases the problem of Runge's phenomenon and also makes an approximation which is very close to the polynomials of the best approximation for a continuous trajectory under the maximum norm. There is another reason why CPs are good for approximation: When the series is truncated at some term, the error resulted from this cut-off is very close to the first term after the cut-off. This makes the computation of error easy. First five CPs (T_0 – T_4) are given in Figure 4-7, which are formed according to the recurrence formula:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x), \qquad T_0 = 1, \quad T_1(x) = x, \qquad n \geq 1 \tag{4.8}$$

Any trajectory can be approximated by

$$y(x) = \sum_{n=0}^{\infty} a_n T_n(x). \tag{4.9}$$

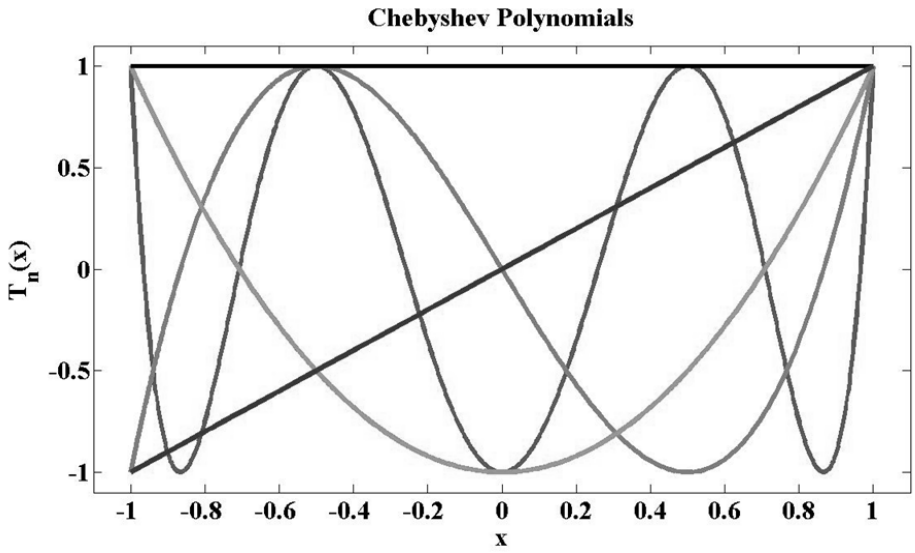

Figure 4-7 First Five Chebyshev Polynomials

When the basis functions of CPs are compared to the basis functions of Fourier, it can be inferred that there is a similarity between them. After employing change of variables, the different basis functions are formed from the trigonometric functions of Fourier via mapping $z = cos(\theta)$.

$$T_n(z) = cos(n\theta) \tag{4.10}$$

Then the following two series become equivalent:

$$f(z) = \sum_{n=0}^{\infty} a_n T_n(z) \tag{4.11}$$

$$f\big(cos(\theta)\big) = \sum_{n=0}^{\infty} a_n cos(n\theta) \tag{4.12}$$

Although the series $f(z)$ is not periodic, $f\big(cos(\theta)\big)$ is periodic in θ with a period of 2π. Due to the equivalence of series, the exponential convergence of Fourier series guarantees the convergence of Chebyshev series.

4.2.2 Legendre Polynomials

For non-periodic trajectories, Legendre Polynomials (LPs) can be used instead of CPs in the interval [-1, 1]. If the computational domain is divided into sub domains, the formulation of Legendre basis functions becomes simpler than that of Chebyshev basis functions. The convergence characteristics of these two different polynomial types are the same, but the maximum error of Legendre series is worse than the maximum error of Chebyshev series. Another difference between these polynomials is that the CPs oscillate uniformly over the interval but the LPs are non-uniform and their magnitudes are small when compared to their counterparts [35]. Just like CPs, LPs can be defined as

$$y(x) = \sum_{n=0}^{\infty} a_n L_n(x). \tag{4.13}$$

These polynomials (L_n) are calculated utilizing the recurrence formula and shown in Figure 4-8.

$$(n+1)L_{n+1}(x) = (2n+1)xL_n(x) - nL_{n-1}(x)\,, L_0 = 1, L_1(x) = x, n \geq 1 \tag{4.14}$$

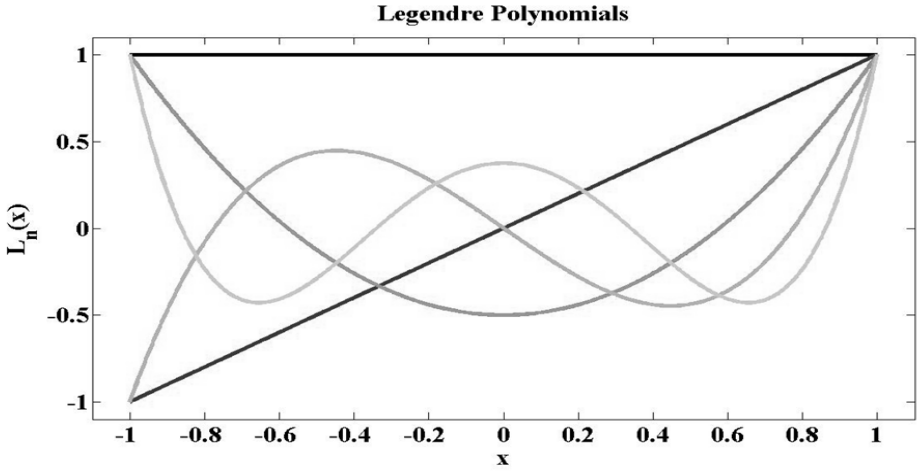

Figure 4-8 First Five Legendre Polynomials

4.2.3 Bernstein Polynomials

The main difference of Bernstein Polynomials (BPs) from Chebyshev and Legendre polynomials is that the BPs are defined in the interval [0, 1] rather than [-1, 1] and always positive. One of the popular application areas of BPs is the generation of Bezier curves in computer graphics. Bernstein basis polynomials are formed using the expression:

$$B_{i,n}(x) = \binom{n}{i} x^i (1-x)^{n-i} \quad (i = 0,1,\cdots,n) \tag{4.15}$$

The Bernstein polynomials are plotted in Figure 4-9 and defined as the linear combinations of Bernstein basis functions given in the equation:

$$y(x) = \sum_{k=0}^{n} a_n B_{k,n}(x). \tag{4.16}$$

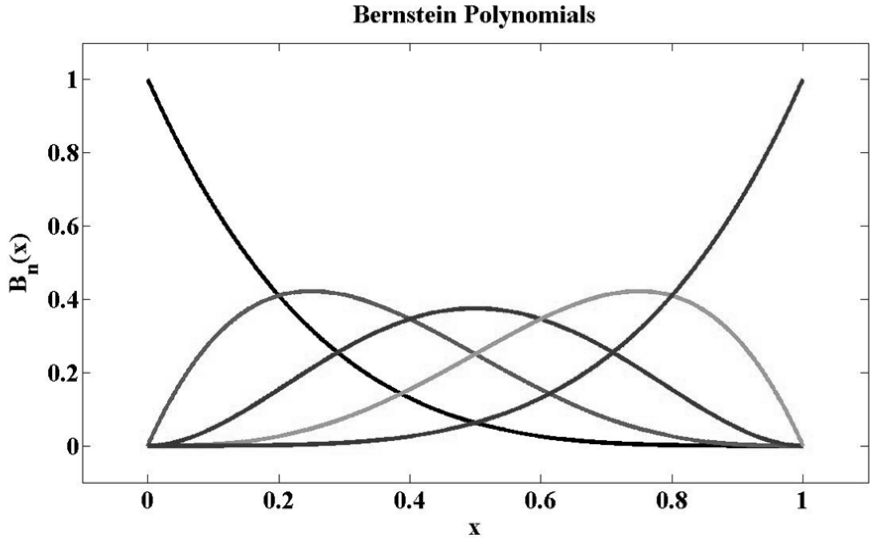

Figure 4-9 First Five Bernstein Polynomials

In order to decrease the computational complexity of Bernstein polynomials, (just like other polynomial techniques) Bernstein polynomials can also be defined in a recursive fashion:

$$B_{k,n}(x) = (1-x)B_{k,n-1}(x) + xB_{k-1,n-1}(x) \tag{4.17}$$

4.3 Performance Evaluation

Before implementing the proposed command generation method on FPGA, polynomial approximation techniques are compared and the most suitable one for command generation is selected for FPGA implementation. This evaluation is carried out in MATLAB, which includes special functions that cannot be easily realizable on the FPGA.

The proposed command generation method based on segmentation and different polynomial approximation methods are applied on the command sequences generated for all six joints of a PUMA 560 manipulator. Figure 4-10 shows these

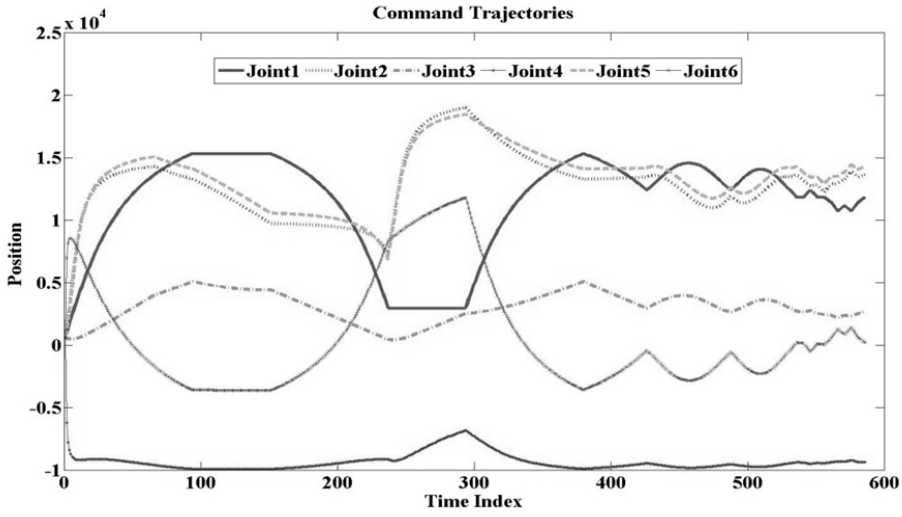

Figure 4-10 Command Trajectories of a PUMA Manipulator

trajectories in encoder counts. It is assumed that the quadrature encoder at each joint has the ability to generate 40000 (= 4 × 10000 counts/rev) counts in one revolution.

The evaluation is carried out for three different variations of the method: **i)** approximation, **ii)** segmentation and approximation, **iii)** segmentation, approximation and error compression.

4.3.1 Single Segment Approximation with Error Sequence Storage

In this version of the command generation method, the trajectories are approximated without any segmentation but with error sequence storage. Chebyshev, Legendre, and Bernstein – Bezier polynomials based approximation method is applied on the command trajectory of the first joint of PUMA 560 manipulator. Plot showing the effect of error root mean square (RMS) value on the compression ratio is provided in Figure 4-11. The number of polynomial

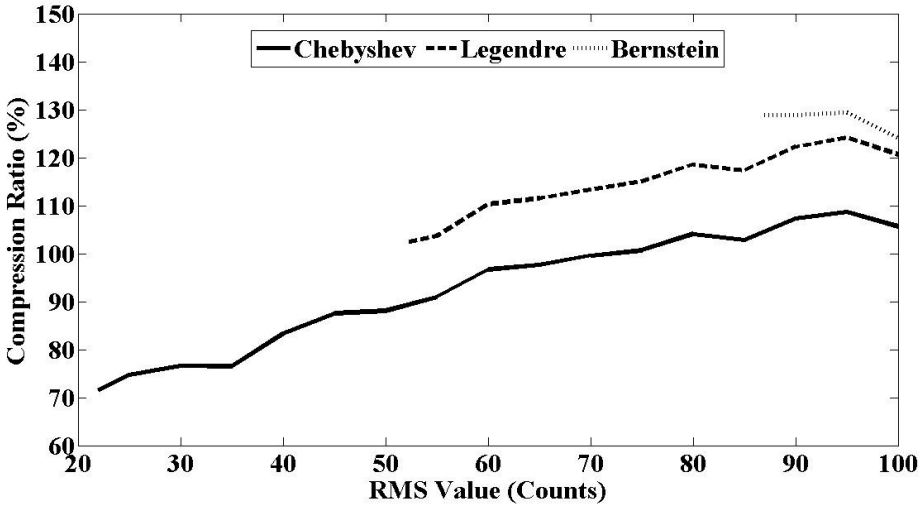

Figure 4-11 Performance of Polynomial Approximation Methods without Segmentation

coefficients used for approximations are also illustrated in Figure 4-12. These methods cannot approximate the trajectory below certain error values, since singularity problems (i.e. the determinant of inverted matrix in (4.6) approaches to zero) occur in the pseudo-inverse technique. Even if the solution of the problem were not ill-conditioned with increasing order, the performance of the approximation methods could not be regarded as acceptable in terms of the large number of coefficients involved to reconstruct the original trajectory. Since the approximation errors mostly lie outside the tolerable range, trajectories must be divided into sections from their inflection points.

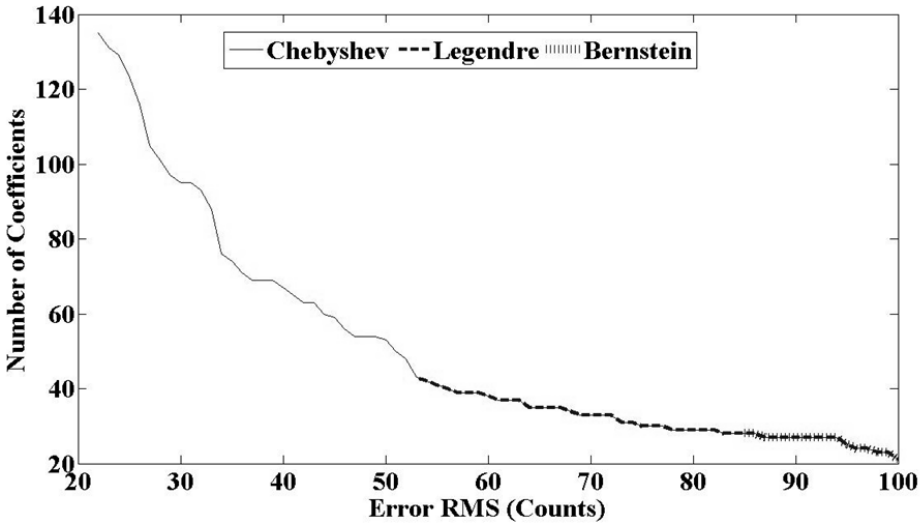

Figure 4-12 Number of Polynomial Coefficients for Different Error RMS Values

During the storage of polynomial coefficients, it is aimed to decrease the computational complexity of the implementation on the FPGA chip. Thus, the coefficients are first rounded to integers and error sequences are formed according to these integer coefficients.

It is critical to notice that while calculating the compression ratios for the above-mentioned techniques, all necessary parameters and error sequence are taken into

account. The following expression is used to calculate the compression ratio of pure approximation methods:

$$r = \frac{\left\lceil \frac{N_c}{8} fix\left[\frac{log(c_{max} - c_{min})}{log(2)} + 1\right]\right\rceil + \left\lceil \frac{N}{8} fix\left[\frac{log(e_{max} - e_{min})}{log(2)} + 1\right]\right\rceil}{\left\lceil \frac{N}{8} fix\left[\frac{log(d_{max} - d_{min})}{log(2)} + 1\right]\right\rceil} \tag{4.18}$$

where N is the length of the original data sequence; N_c is the number of coefficients used in the approximation; c_{max}, d_{max}, e_{max}, and c_{min}, d_{min}, e_{min} represents the maximum- and the minimum values of the coefficients, the original data sequence, and the error sequence respectively.

4.3.2 Segmentation and Approximation with Error Sequence Storage

In this sub-section of the evaluation, the trajectories in Figure 4-10 are firstly segmented according to the inflection points and then polynomial approximation methods are employed on these segments. The results are given in Figure 4-13. In the method, the most important parameter is the approximation error RMS value. If it is selected to be small, the error in the approximation decreases but the number of polynomials and coefficients increase tremendously. Therefore, a proper RMS value should be selected minimum compression ratio. The formula required to calculate the compression ratio is the same with the one given in Section 4.3.1. As it is inferred from Figure 4-13 that the best performance is achieved for all approximation methods when the error RMS value is four. Relatively high compression ratios at the beginning of the plot are due to the number of polynomials used to achieve corresponding RMS value. After the value of four, compression ratio tends to rise for all polynomial types since the memory required for storing the errors increases. Lastly, when the three approximation

methods are considered, Chebyshev Polynomials outperform the Legendre and Bernstein – Bezier polynomials for approximation.

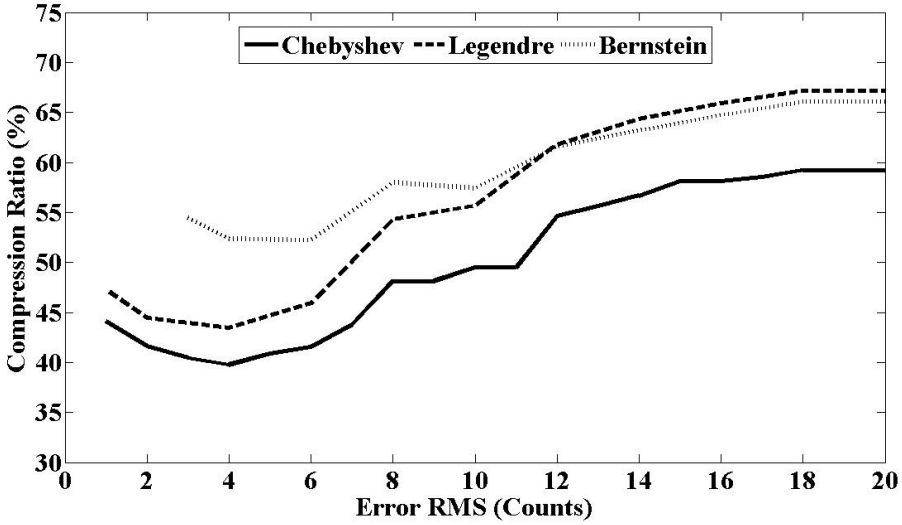

Figure 4-13 Performance of Polynomial Approximation Methods with Segmentation

4.3.3Segmentation and Approximation with Error Sequence Compression

As can be understood from the name of this sub-section, the main difference of this variation of the method is that the errors resulting from the approximation are compressed before storing. For the error compression, the data compression technique proposed in Chapter 5 is used. The performance of this variation is illustrated in Figure 4-14. As in the other variations, CPs outperform the other polynomial types and it can easily be inferred from the figure that this variation is the best of all variations of the command generation method based on polynomial approximation paradigms. By employing CPs, the original trajectory can be compressed to 23% of its original size. It should be noted that during the compression of error sequence, its first order of difference is taken and compressed.

The compression ratio calculation is carried out with the same equation given in Section 4.3.1 with an additional compression ratio parameter (r_e) for the error sequence as

$$r = \frac{\left\lceil \frac{N_c}{8} fix\left[\frac{log(c_{max} - c_{min})}{log(2)} + 1\right]\right\rceil + r_e\left\lceil \frac{N}{8} fix\left[\frac{log(e_{max} - e_{min})}{log(2)} + 1\right]\right\rceil}{\left\lceil \frac{N}{8} fix\left[\frac{log(d_{max} - d_{min})}{log(2)} + 1\right]\right\rceil} \tag{4.19}$$

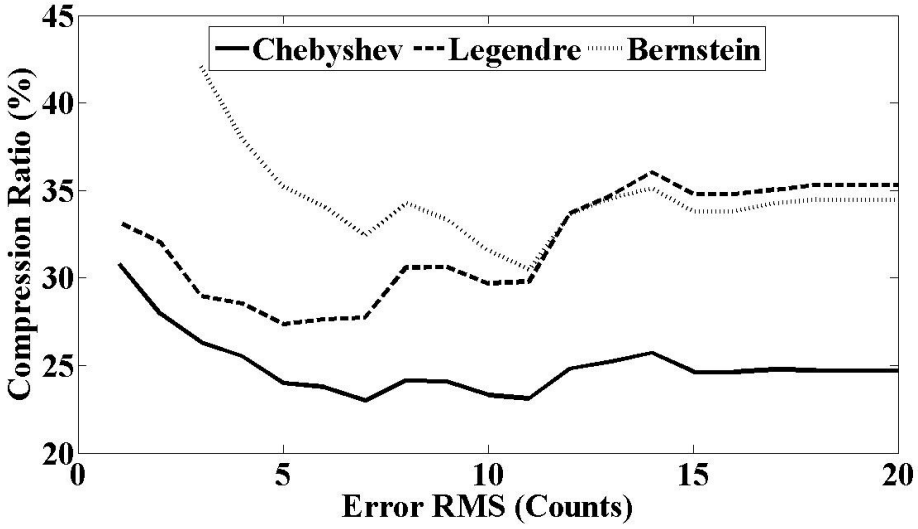

Figure 4-14 Performance of Polynomial Approximation Methods with Segmentation and Error Compression

4.4 FPGA Implementation

After the elaboration of different polynomial techniques employed on the trajectories of the PUMA manipulator (see Figure 4-10), it is turned out that the Chebyshev polynomials give much better results than the Legendre and the

Bernstein polynomials. Thus, the Chebyshev polynomials based command generation method is realized utilizing the Altera DE 1 FPGA Development Board (with Cyclone II FPGA) [36] via two different approaches. In the first approach, the command generation method is directly written in VHDL utilizing the schematic design property of software Quartus II 9.0 Web Edition. This technique will be referred to as "Hardwired" approach. In the second technique, (rather than writing directly in hardware description language) architecture is implemented in NIOS II Embedded Development Environment [37], where a softcore processor IP serving as an embedded microcontroller is deployed on the FPGA. In the following two sub-sections, the differences between these approaches will be discussed and their performances shall be evaluated using the joint-state trajectories generated for the PUMA manipulator.

4.4.1 Hardwired Approach

In the first approach of FPGA implementations, the proposed command generation method is realized by directly writing the algorithm in VHDL. During writing, schematic design property of Quartus II 9.0 Web Edition is used. With this property of the software, it is much easier to sustain and track the communication between different modules performing specific tasks. Schematic design generating the commands for the first joint of PUMA manipulator, whose trajectories are given in Figure 4-10, is provided in Figure 4-15. In this design, there are mainly four different modules and two memory units. The modules are Driver Module (DM), Splitter Module (SM), RS-232 Module, Floating Point Operation Module (FPOM). The first memory unit is used to store Chebyshev polynomials. In this memory unit only hundred discrete values of first eight Chebyshev polynomials are stocked. Thus, in this hardwired implementation there is a restriction on the maximum width of the segments, which is hundred. If the length of the segment is less than hundred, then a proper change of variables is applied to the segment. On the other hand, in the second memory unit polynomial coefficients and the widths of the segments are stored. These coefficients are

passed to the FPOMs in a proper order for multiplication with the Chebyshev polynomials. After the multiplications, the results are summed in other FPOMs and sent to the RS-232 module in order to transfer the commands to the computer. Table 4-1 represents the allocated resources in FPGA for the implementation of the method with the hardwired approach. It can be inferred from the table that the almost half of the logic elements are used for this method, which can be regarded as high. The number of pins used is very low, since overall design does not have any communications with peripheral devices. It just internally generates the commands. It uses 37% of the multipliers due to the FPOMs embedded in the design. For a better illustration of resource usage, Figure 4-16 can be viewed. It can also be understood from the figure that about half of the resources are consumed by the architecture. The most important thing of this implementation is that it takes only 750 μs to generate 586 commands, which is fast for command generation systems. In the following sub-sections, the four main modules used in the design are explained in detail.

Table 4-1 FPGA Resources used in Hardwired Approach

Total Logic Elements	11098 (45%)
Total Combinational Functions	10885 (44%)
Dedicated Logic Registers	2644 (11%)
Total Registers	2644
Total Pins	4 (2%)
Total Memory Bits	26880 (4%)
Embedded Multipliers 9-bit Elements	49 (37%)
Total PLLs	0

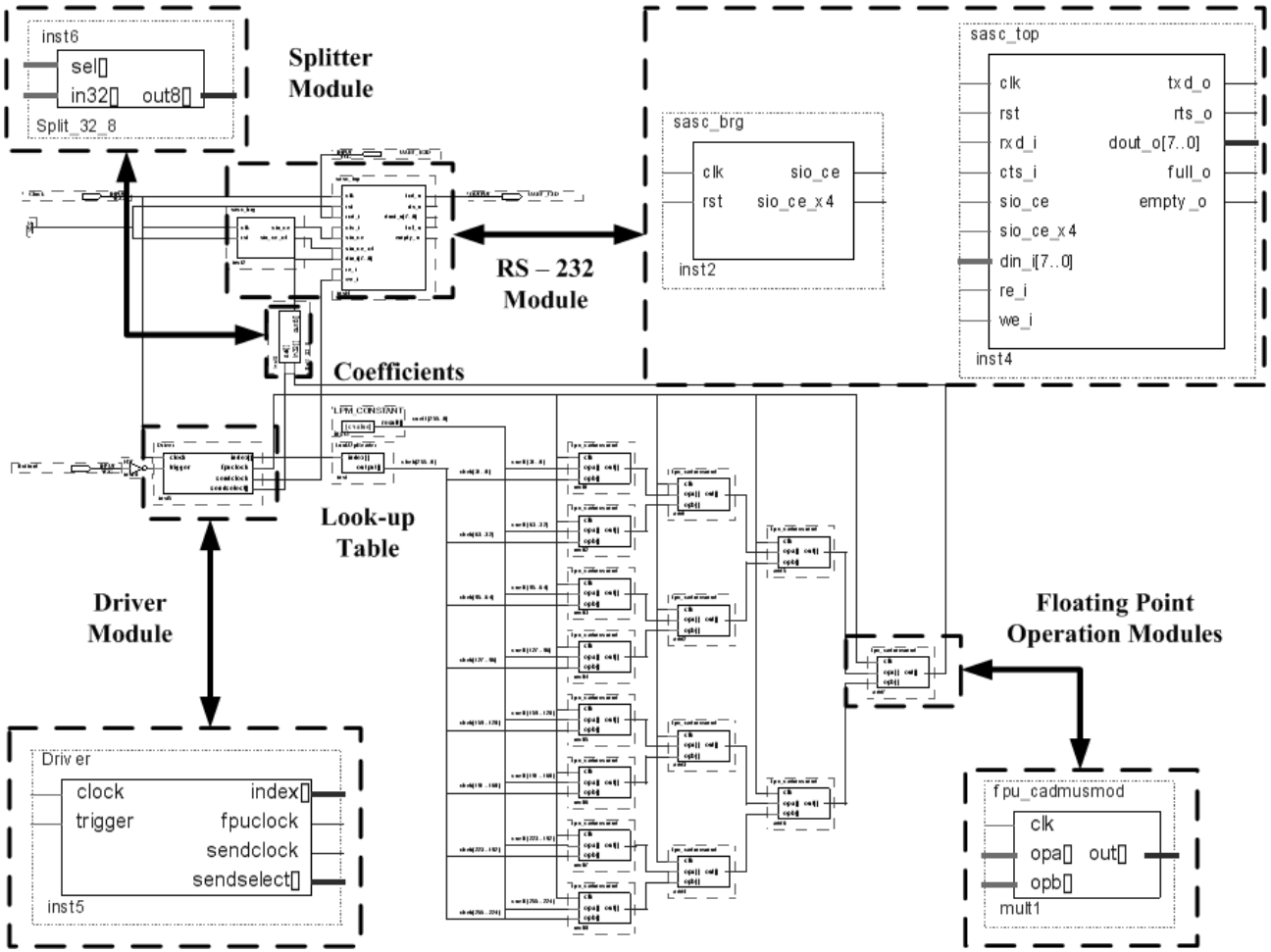

Figure 4-15 Hardwired FPGA Implementation of the Method

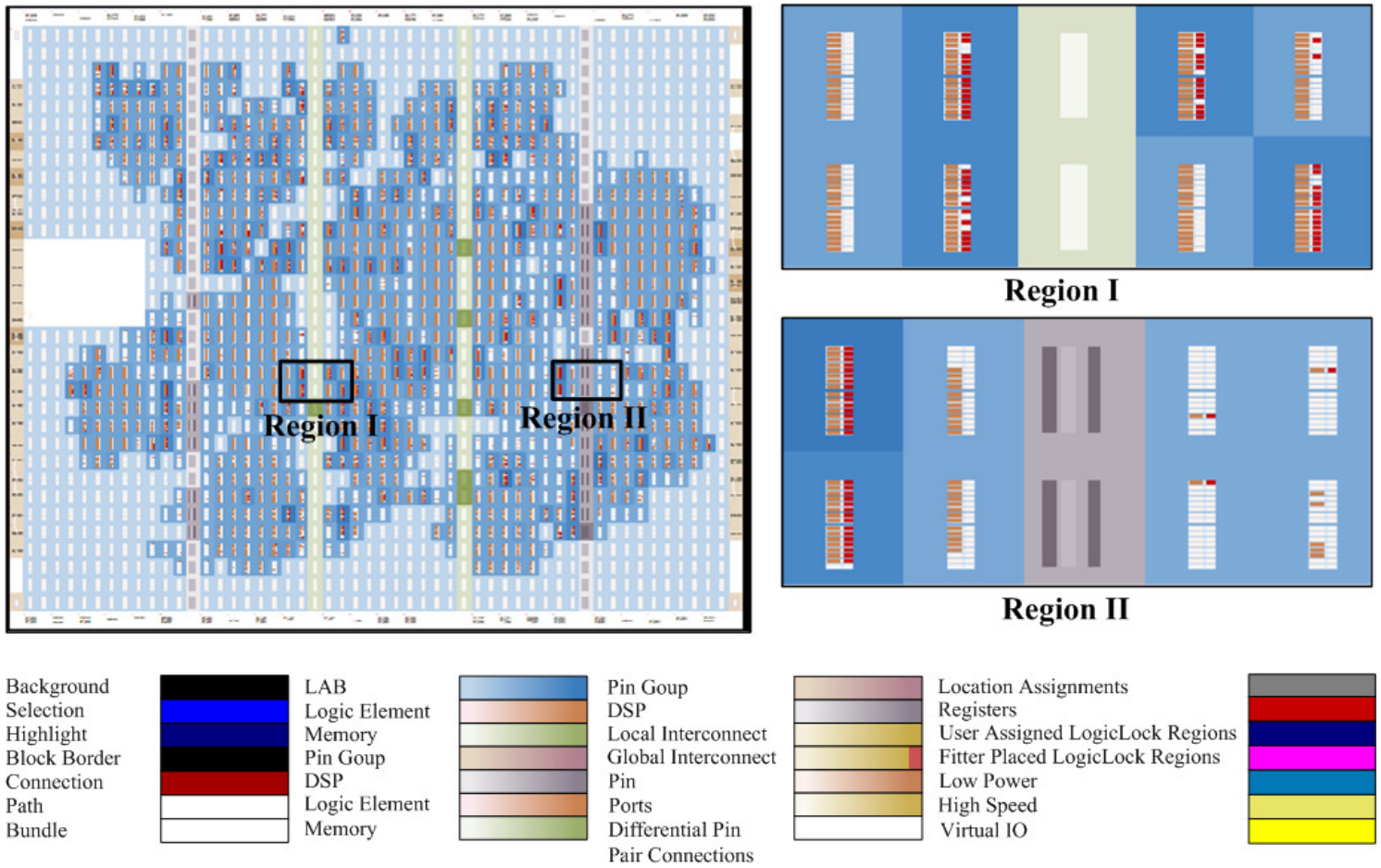

Figure 4-16 Floor Plan of the Synthesized Digital Circuitry for the Hardwired Implementation

4.4.1.1 Driver Module

Driver Module can be regarded as the manager module of the design. It communicates with all the modules and memory units to make the command generator operate properly. As can be seen from Figure 4-15, the DM has two inputs and four outputs. The first input is the global clock available on the FPGA board, which is 50 Mhz in this design, and the second input is the trigger input given by an external logic to the system to generate commands. Among outputs of the DM, `index` is sent to the Look-up Table. With the index value, Look-up Table outputs the values of CPs to the first set of FPOMs. `fpuclock` is generated and transferred to the FPOMs to perform mathematical operations in the desired order of the DM. Communication with the RS-232 Module is sustained with the output `sendclock`. Since the DM knows exactly how many cycles are necessary for mathematical operations, it outputs this clock accordingly. `sendselect` output is given to the SM to split the 32-bit command values into 8-bit values. With all these inputs and outputs the DM operates as the main module of the design.

4.4.1.2 Floating Point Operation Module

There are fifteen FPOMs in the design. Eight of them perform multiplications, and the rest sums the results of these multiplications. The original of this module was developed by Usselmann [38]. This current version is simplified to respond only to the needs of the design. The module, whose schematic is provided in Figure 4-15, has three inputs and an output. The first input `clk` is received from the DM module and the other inputs are the 32-bit floating point data to be used in mathematical operations. The only output `out` is the result of multiplication or addition in this architecture. It is sent to the other FPOMs for further operations or sent to the RS-232 module.

4.4.1.3 Splitter Module

Splitter module, shown in Figure 4-15, is the simplest module used in the design. It just takes the input `sel` as the clock and splits the incoming 32-bit wide floating point data `in32` into four 8-bit wide values. Finally sends these data to the RS-232 module directly. The reason why there is need for such segmentation is that the available RS-232 module can operate with 8-bit data values.

4.4.1.4 RS-232 Module

The RS-232 Module, developed by Usselmann [40], actually consists of two sub-modules; `sasc_brg` and `sasc_top` as named in Figure 4-15. The module on the left side is the baud rate generator. The baud rate is adjustable using the divisor registers in the module according to the global input given as input to the module. The module on the right side is responsible for the communication with the PC and the incoming data.

4.4.2 Embedded Softcore Processor Approach

In the second approach, the command generation algorithm is written in C programming language and the resulting are cross-compiled to run on a softcore processor deployed on the FPGA. The "machine code" is then downloaded to this processor. Note that the embedded softcore is designed in the NIOS II Embedded Development Environment. Schematic design of this *algorithmic state machine* (ASM) is shown in Figure 4-17. The softcore Intellectual Property (IP) has a Synchronous Dynamic Random Access Memory (SDRAM) Unit additional to the basic micro-processor units. SDRAM is used to store Chebyshev polynomials, coefficients, and widths of the segments. Due to the usage of an external memory unit, hardware resources used in the FPGA is less than the ones used in the first

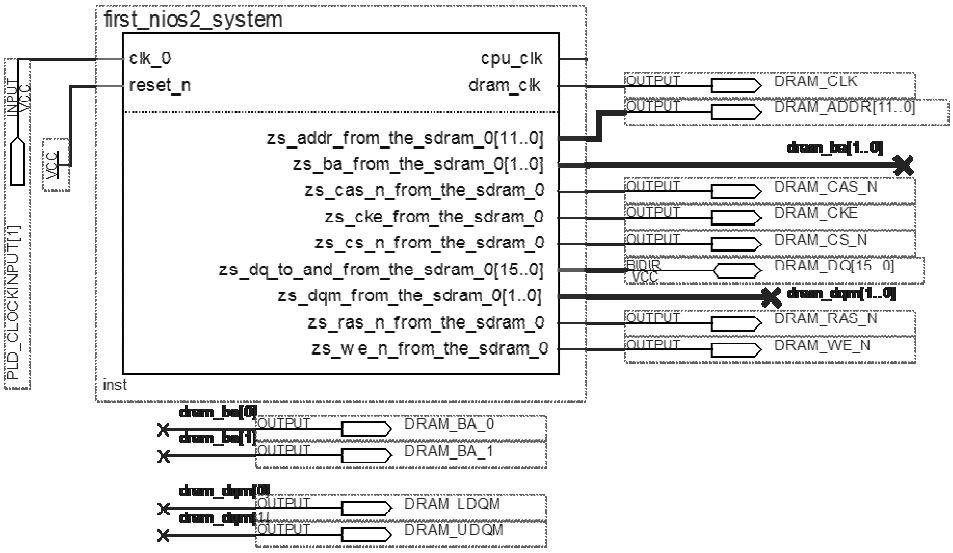

Figure 4-17 Softcore FPGA Implementation of the Method

approach as can be seen in Table 4-2. The processor occupies only about one-fifth of the resources. This can also be verified from the chip plan given in Figure 4-18. For the given trajectory, the implementation only uses 61 kB (0.007% of SDRAM) of memory. The time necessary for generating a command sequence of 586 points is around 247 ms, which is much higher than the first approach. It is critical to note that the basis functions (i.e. Chebyshev polynomials) can be generated in advance and it takes roughly 213 ms.

Table 4-2 FPGA Resources used in Softcore Approach

Total Logic Elements	3444 (18%)
Total Combinational Functions	3085 (16%)
Dedicated Logic Registers	1914 (10%)
Total Registers	1966
Total Pins	39 (12%)
Total Memory Bits	28992 (12%)
Embedded Multipliers 9-bit Elements	0
Total PLLs	1 (25%)

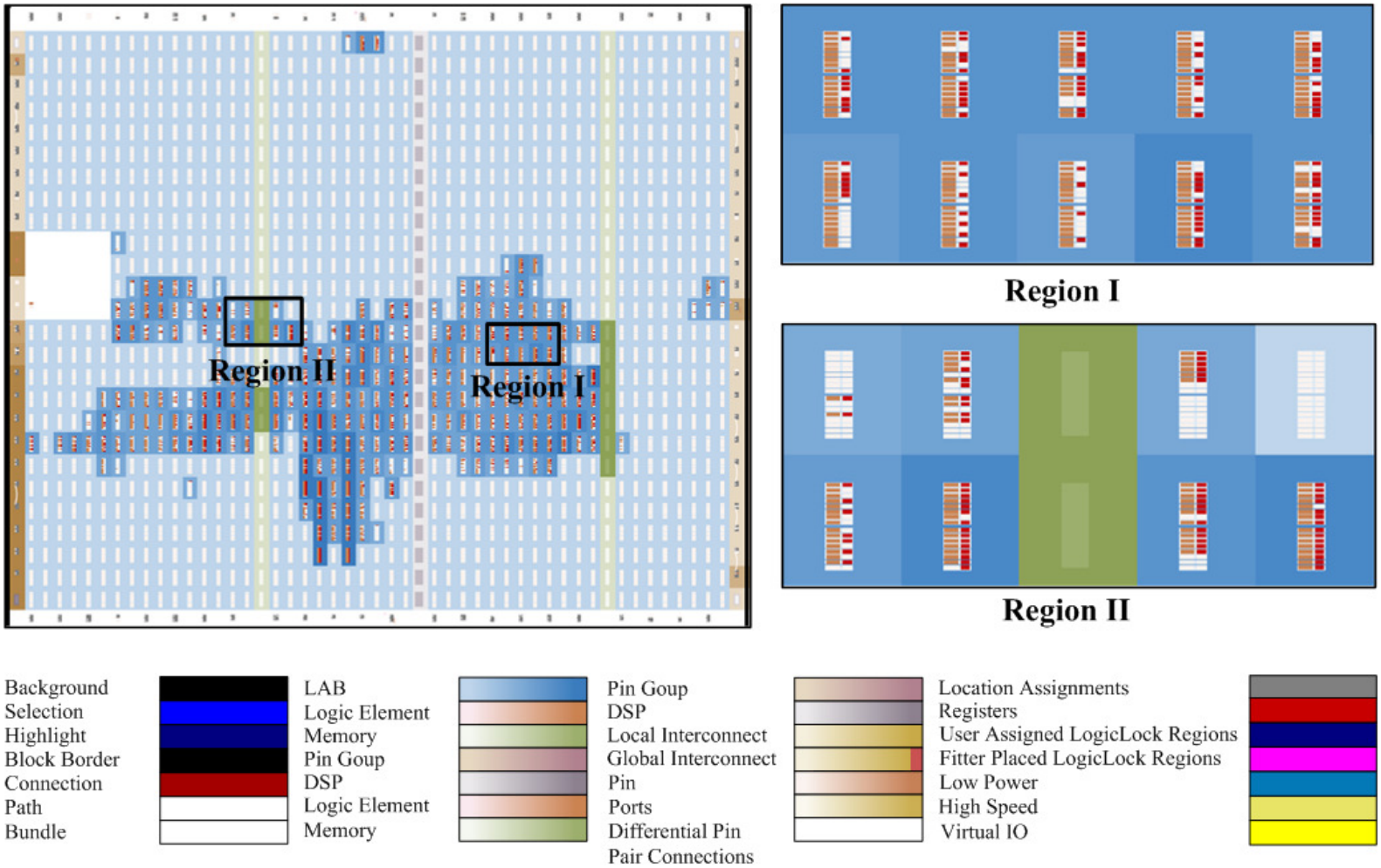

Figure 4-18 Floor Plan of the Synthesized Digital Circuitry for the Softcore Implementation

4.5 Closure

In this chapter, command generation method based on segmentation and polynomial (Chebyshev, Legendre, and Bernstein) approximation was proposed. The method was then implemented on the FPGA development board using two different approaches. Pure polynomial approximation was not used for the implementations, since during the elaboration of the method in MATLAB environment it turned out that the segmentation of the trajectory was inevitable for small magnitudes of errors. The two implementations have their own advantages over the other. Hardwired approach is much faster than the softcore counterpart. On the other hand, the hardware resources used by the embedded softcore is about the half of resources occupied by the hardwired approach. To sum up, if there is no restriction on the usage of hardware resources, it is better to implement hardwired approach which is much faster than the other.

CHAPTER 5

COMMAND GENERATION METHOD UTILIZING DIFFERENCING AND COMPRESSION WITH VARIABLE FEED-RATE

The second developed command generation method consists of two phases: differencing and compression. The main difference of this method from the first one is that there is no approximation during command generation and as a result there are no representation errors. In this chapter, after the need for differencing (before compression) is explained, the proposed data compression technique will be elaborated. In the following section, the performances of various data compression methods are comparatively elaborated. According to the obtained results, variable feed-rate input (i.e. time/velocity scalar) is incorporated to the most successful method. Finally, the command generation algorithm is implemented on the FPGA board using two different approaches: hardwired and embedded softcore processor.

5.1 Differencing

Methods involving higher-order differences of time-sequences are applied to decrease the memory required for storage [4]. Note that in the literature this technique is referred to as differencing or relative encoding [3]. Rather than storing directly the whole command trajectory, it is beneficial to store the

differentiated data along with the necessary initial values. Higher-order difference of a sequence can be represented as

$$\nabla q = q(k) - q(k-1), \tag{5.1}$$

$$\nabla^2 q = \nabla q(k) - \nabla q(k-1), \tag{5.2}$$

$$\nabla^{\mathrm{n}} q = \nabla^{\mathrm{n}-1} q(k) - \nabla^{\mathrm{n}-1} q(k-1). \tag{5.3}$$

In these equations, $\nabla^{\mathrm{n}} q$ represents the n$^{\mathrm{th}}$ order difference and k is the index. As the order of difference increases, the memory needed for the storage of the sequence decreases. In order to represent the relationship between the memory requirement and the order of difference, several trajectories are formed and their differences are taken up to 7$^{\mathrm{th}}$ order [1]. This result is shown in Figure 5-1. As can be seen from the figure, after the third-order difference, there is an increase in the memory usage since the sign of each data point frequently changes in an alternating fashion after the 4$^{\mathrm{th}}$ order difference. Hence, the range of data broadens considerably. Therefore, the best solution for data storage is achieved when the order of difference is three or four for most of the motion control applications.

Figure 5-1 Effect of Order of Difference on Memory

Note that the original data can be extracted using the differenced data along with the initial values. For the first order differences, this integration (accumulation) operation can be expressed as

$$q(k) = q(k-1) + \nabla q(k) \tag{5.4}$$

When $k = 1$, an initial value $q(0)$ along with $\nabla q(1)$ is needed to calculate $q(1)$. If the order of difference is greater than one, number of necessary initial values increases. While calculating the compression ratios of the methods utilizing differencing, memory required for the initial values should also be considered. Differencing without compression can also be used as an alternative command generation method. For the illustration of this method, the second joint trajectory of the PUMA manipulator (shown in Figure 4-10) is reconsidered. When the area underneath the trajectory and its differences are plotted in Figure 5-2, it can be observed that the areas decrease remarkably. On the other hand, the

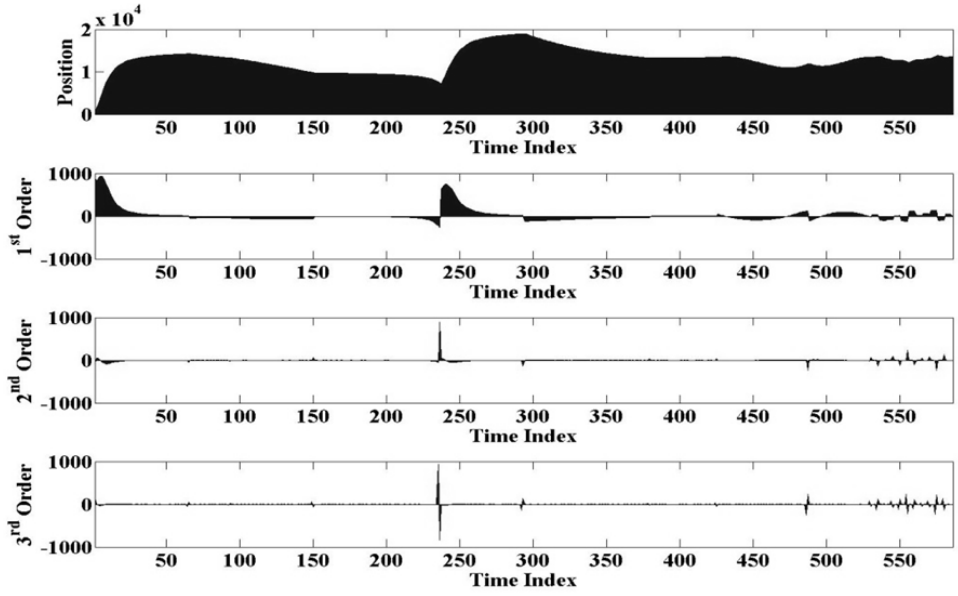

Figure 5-2 Second Joint Trajectory of PUMA Manipulator and Its Differences up to Third Order

compression ratios of differences do not change as remarkably as the areas change. This is due to the reason that there must be constant bit widths for the trajectory and the maximum value of the sequence determines this width. In Table 5-1, compression ratios are given up to sixth order of difference for all the trajectories of the manipulator. It should be noted that in the calculation of compression ratios, the necessary initial values for decoding are also considered.

Table 5-1 Compression Ratios vs Order Difference [%]

	Order of Difference					
Joint Number	**1**	**2**	**3**	**4**	**5**	**6**
1	73.25	66.61	66.52	73.07	79.62	86.17
2	73.25	73.25	79.80	79.71	86.26	92.81
3	61.55	69.22	69.12	76.79	76.68	84.24
4	99.90	92.69	92.59	92.50	92.40	92.30
5	73.25	73.25	79.80	79.71	86.26	92.81
6	93.27	86.53	86.44	86.35	86.26	86.17

5.2 Proposed Data Compression Algorithm

Data compression algorithms described in the second chapter are universal methods. That is, they can be utilized to compress any data type with diverse statistical attributes including text, audio, image, video, etc. While this generality feature can be regarded as an advantage, the implementation of such compression paradigms on FPGA chips can be often times quite complex and could drain considerable resources on a particular FPGA. On the other hand, the compression

algorithm proposed in this thesis is specifically developed to deal with optical position encoder commonly encountered in motion control applications. Since the output of these sensors, which satisfy C^0 (and frequently C^1) continuity, can be conveniently represented as (signed or unsigned) integers, one can exploit such (temporal) sequences to come up with an efficient compression technique that is easier to implement on a FPGA chip with modest resources. In the following subsections, encoding and decoding algorithms of this method are to be explained in detail.

5.2.1 Encoding Process

The basic idea behind this technique is that when the higher-order differences of a reference trajectory (i.e position/location sequence) in a typical motion control application is computed, the (integer) values in the resulting sets do decrease considerably. Furthermore, since most motion control applications require constant velocity along the traced trajectory, the majority of the differentiated data is likely to be null (0) while the rest is composed of small integers in which the probability of occurrence is inversely correlated with the magnitude. Considering that a small integer number would require fewer bits, the difference data would take up significantly less memory if compared to the original data set. Unlike entropy-based (general) compression techniques (like Huffman coding), one can directly encode the difference data in this technique without calculating the probability density of the processed data owing to the fact that the special requirements associated with the motion control applications (due to operational concerns) tightly dictate the statistical distribution data beforehand.

Consequently, the proposed compression algorithm (to be referred to as the **ΔY Method** hereafter) is employed on the higher-order differences of the command trajectory (usually position). Once the differenced data of the command sequence is calculated according to the specified order, the resulting data is compressed (a.k.a. "compacted") utilizing the ΔY algorithm. In Figure 5-3, a sample encoding

process for the third-order difference is illustrated. Note that the compressed code consists of three fields (Sign, Amplitude, and Length Fields) and initial value set. The sign field includes sign bits: 0 and 1 represent positive- and negative numbers respectively. Note that if the magnitude of data is zero, no sign bit is assigned for this special case. Hence, the length of the sign field equals to the difference between the number of data points (in the differenced set) and the number of zeros in the data. Similarly, the amplitude field encodes the absolute values of the data sequentially as binary numbers with variable length. To extract the differenced data, another field (a.k.a "length field"), which yields the length of each value in the amplitude field, needs to be formed. As can be seen from Figure 5-3, this field contains sequences of 1's and 0's in an alternating manner. Once can detect the length of a particular number in the amplitude field by simply counting the bits in between two consecutive transitions (0-to-1 or 1-to-0) detected in the length field. Lastly, the order of differencing, and the initial values are needed for lossless decompression. The initial values, which are used to initialize integrator (or accumulator) states, depend on the order of the difference. That is, the number of integrators used in decoding is equal to the order of difference.

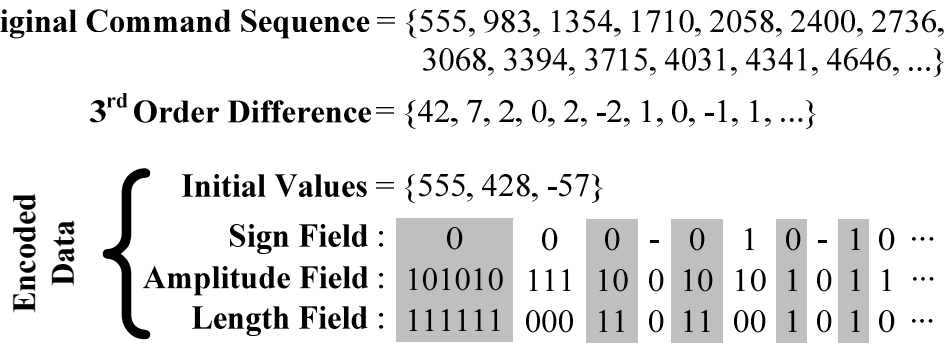

Figure 5-3 Sample Encoding Process for ΔY Method

5.2.2 Decoding Process

Architecture implementing the decoding process of the ΔY technique is illustrated in Figure 5-4. Decoding starts out with the comparison of the consecutive bits in the length field to detect the transitions in this field. After the length of a particular binary data residing in the amplitude field (l_i) is determined through the bit comparison module and the counter, the information is passed to the left-shift register. Considering the length, the shift register extracts the amplitude of the data from the corresponding field and transfers the data to the differenced data module. After taking the sign value (utilizing another left shift register) from the sign field, the differenced data module completes its task and sends the data to the integration module. In the mean time, the initial values are transferred to the integration module and the original data by accumulating the differenced data in a sequential manner.

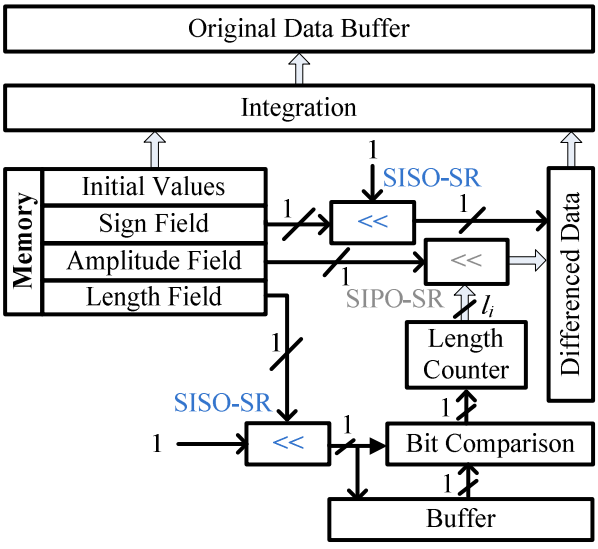

Figure 5-4 Decoding Process of ΔY Decompression Algorithm

It is critical to notice that as indicated in the first section of this chapter, the presented technique generates the original data (i.e. position) by accumulating the finite differences of that sequence in a successive fashion. For instance, when n equals to 3, the third-order difference (i.e. jerk) is accumulated to obtain second-order difference (i.e. acceleration/deceleration). The corresponding results are iteratively accumulated until the position at a particular instant in time is calculated. Hence, the discrete-time derivatives of the command sequence, which are frequently required by modern motion controller topologies, are inherently computed in this technique.

5.3 Performance Evaluation

In this section, Huffman, Arithmetic Coding, and the ΔY compression algorithms are applied on the trajectories of the PUMA 560 manipulator in MATLAB (see Figure 4-10). After the finite differences of the command trajectory for the first joint are computed for various orders, the data compression algorithms are employed to compress the differentiated trajectory data. Resulting compression ratios (in percent) are presented in Table 5-2. In the table, *n* represents the order of finite difference. Note that while calculating the compression ratios for the methods utilizing differencing, the memory required for the initial values is also taken into consideration. As observed from Table 5-2, if $n > 1$, the compression algorithms yield much better results owing to the fact that the increments of encoder counts from one sampling step to another (i.e. angular velocity) are still quite high as the robot performs a jerky motion throughout the followed trajectory in this particular example. Another conclusion to be drawn from the table is that the performance of the ΔY algorithm is superior to those of the others. With $n = 3$, the command sequence can be compressed to about one-fourth of its original size. Notice that the results given in Table 5-2 are also in good agreement with Figure 5-1. Up until the third order, the compression ratio decreases and after that there is an increasing trend. This situation may be

explained by the decline in the frequencies of the numbers in the sequence and the expansion in the range of the data values.

Table 5-2 Compression Ratios for Various Orders

	Compression Ratio (%)		
n	**Huffman**	**Arithmetic Coding**	**ΔY Method**
0	197.1	181.4	182.9
1	131.0	118.6	77.9
2	34.1	32.7	30.3
3	34.8	31.9	23.6
4	46.4	42.3	30.8
5	60.6	54.1	39.3
6	81.7	70.9	49.6

It is critical to notice that while calculating the compression ratios for the above-mentioned techniques, all necessary parameters to extract the original command sequence (including compressed code, initial values, dictionary tables, etc.) are taken into account. The following expression is used to calculate the compression ratio of the ΔY technique:

$$r = \frac{\left\lceil \frac{1}{8}\left(\sum_{i=1}^{N-n} 2l_i + N - n - n_0\right)\right\rceil + \left\lceil \frac{n}{8}\left\lceil \frac{log(d_{max} - d_{min})}{log(2)} + 1\right\rceil\right\rceil}{\left\lceil \frac{N}{8} fix\left[\frac{log(d_{max} - d_{min})}{log(2)} + 1\right]\right\rceil} \tag{5.5}$$

where N is the length of the original data sequence; n is the order of finite difference; l is the binary length of each data; n_0 is the number of zero magnitude data; d_{max} and d_{min} represents the maximum- and the minimum value of the original data sequence respectively.

To be able to determine which compression algorithm is suitable for command sequences, the aforementioned methods are applied to all trajectories generated for the PUMA 560 manipulator after taking third-order finite differences of the angular position data (in encoder counts). The results are shown in Table 5-3. It is clearly seen from the table that the proposed method leads better results than the contending techniques.

Table 5-3 Compression Ratios for Third Order Differences

		Joint Number					
		1	**2**	**3**	**4**	**5**	**6**
Method	**Huffman**	34.8	43.9	36.0	37.0	46.0	41.3
	Arithmetic	31.9	40.2	33.8	34.8	42.0	37.9
	ΔY	23.6	24.2	24.1	22.2	24.7	25.3

5.4 Command Generation with Variable Feed-rate Input

A novel command generation scheme, where the programmed velocity along the traversed trajectory can be changed dynamically, is elaborated in preceding section. In CNC applications, the speed (i.e. feed-rate) through the course of motion is generally modified by external input (like feed-rate override knob). Under some extreme cases (such as the control scheme of an electro-discharge

machine), it might be desirable to reverse the direction of motion as dictated by an external source. Therefore, the proposed command generation method is to be augmented to accommodate a variable feed-rate input.

With this property, the users will be able to change the rate of command generation in both forward and reverse directions. During generation, when there is a need for the intermediate command values, a linear interpolator should be incorporated to the design. That is, this unit is to interpolate between the two decoded command values based on the following expressions:

$$a_k = a_{k-1} + f_k \ (mod\ f_{max}) \tag{5.6a}$$

$$m := \begin{cases} m-1, & a_{k-1} + f_k < 0 \\ m+1, & a_{k-1} + f_k > f_{max} \end{cases} \tag{5.6b}$$

$$u_k = u_{m-1} + \frac{(u_m - u_{m-1})a_k}{f_{max}} \tag{5.7}$$

where u represents the decoded commands at the interval $m \in \{0, 1, \ldots, N\}$; k is the time index. Similarly, $f_k \in \{-f_{max}, \ldots -1, 0, 1, \ldots f_{max}\}$ indicates the current value of the feed-rate input to the system while $f_{max} \in Z^+$ denotes the maximum feed-rate at which commands could be generated. Note that the variable (a_k) in (4.7) essentially serves as a time scaling factor.

In Figure 5-5, a sample interpolation is carried out with a feed-rate of (3/8)f_{max}. That is, if the sampling time is selected as 0.008 s, then with the specified feed-rate the new sampling time becomes 0.003 s. As can be seen from the figure that before the 4th interpolated command is generated, the difference value is updated and the next three commands are generated according to the new difference value.

The shaded area underneath the 2nd original command can be regarded as the error of the interpolation algorithm. After each original data point, extrapolation can also be used rather than interpolation. In the extrapolation case, it is guaranteed to

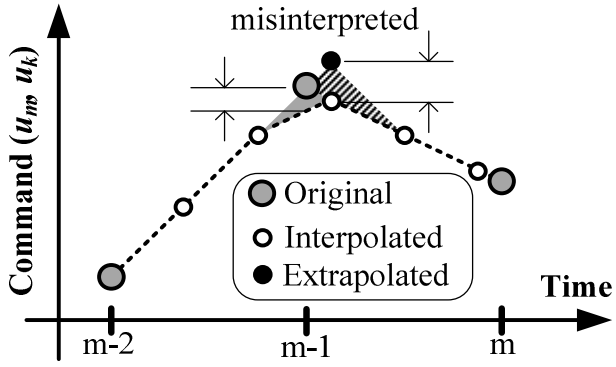

Figure 5-5 Interpolated Data

generate the original command. On the other hand, duration of the representation error of the extrapolation is always larger than the one of interpolation. Another approach to eliminate the representation errors at least at the original command points, the original data can be generated regardless of the sampling time.

The proposed command generation method with variable feed-rate is evaluated in MATLAB before implementing it on an FPGA development board. The feed-rate profile in Figure 5-6 is applied to the original command trajectory illustrated in Figure 5-7. The feed-rate profile is formed such that all the commands are first generated in the forward direction and then in the reverse direction with continuously changing feed-rate. There occur some command representation errors (as shown in Figure 5-8) at each original data points since the interpolation algorithm (described in the previously) is not capable of generating commands at these points. When Figure 5-7 and Figure 5-8 are considered together, it can easily be inferred that the larger errors occur at the inflection points of the trajectory.

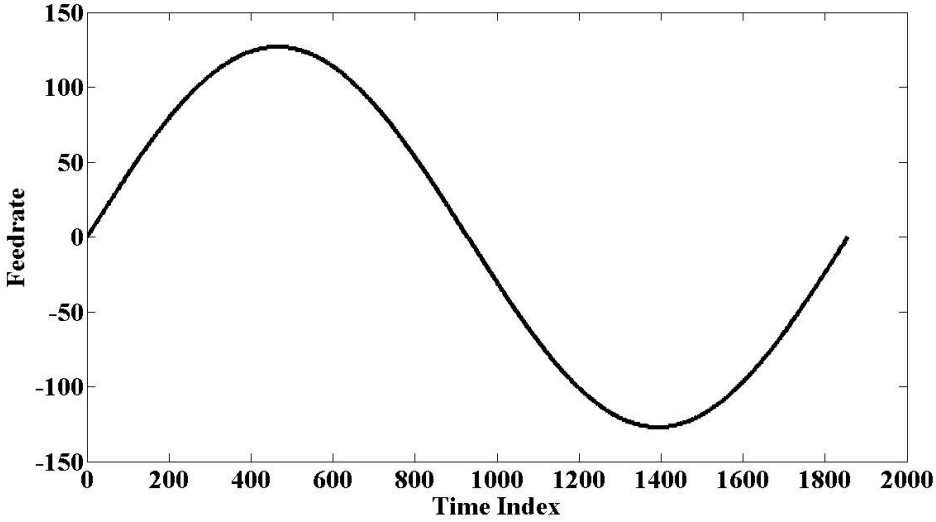

Figure 5-6 Feed-rate Profile

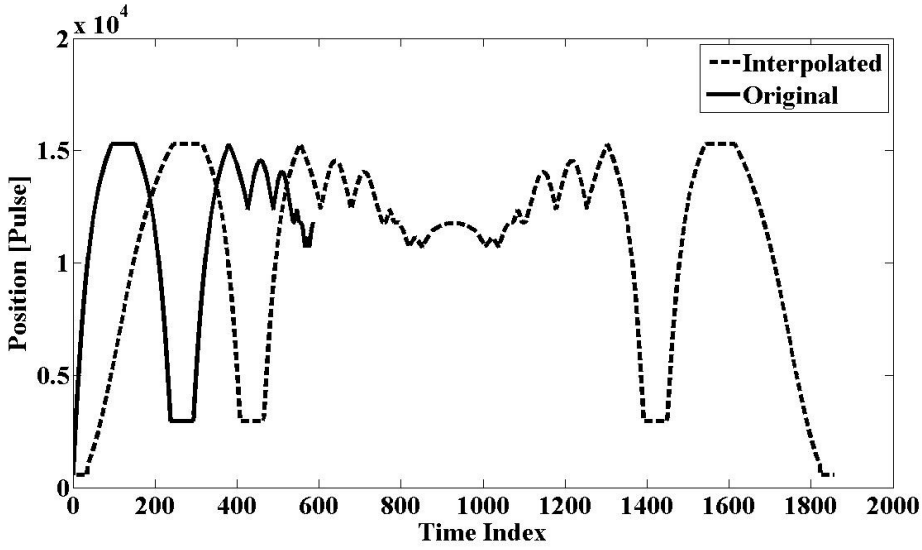

Figure 5-7 Interpolated and Original Command Sequences

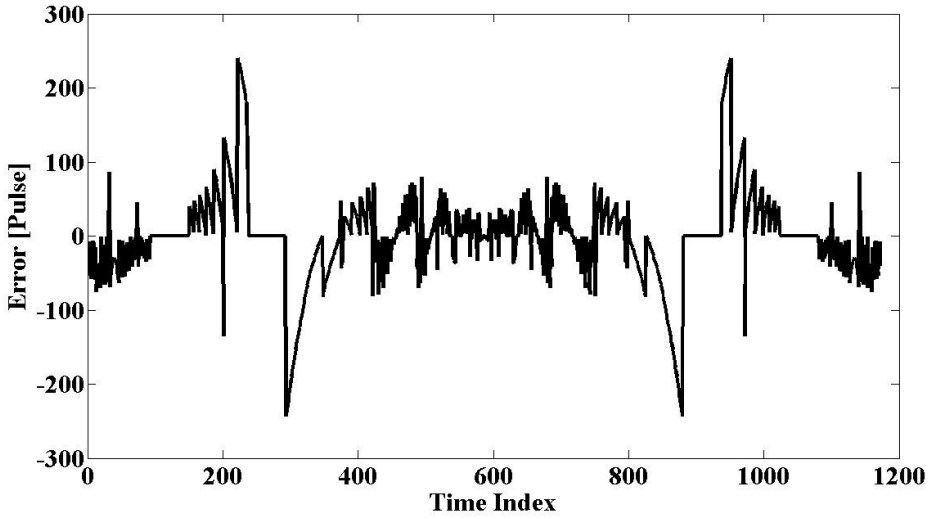

Figure 5-8 Command Representation Errors

5.5 FPGA Implementations

After the elaboration of different data compression techniques employed on the trajectories of the PUMA manipulator, it is turned out that the ΔY compression method exhibits superior performance over the Huffman and the Arithmetic compression techniques. Thus, the ΔY method based command generation paradigm is realized utilizing the Altera DE1 FPGA Development Board (with Cyclone II FPGA) [36] via two different approaches. In the first approach, the command generation method is directly written in VHDL utilizing the schematic design property of software Quartus II 9.0 Web Edition. This technique will be referred to as "hardwired" approach. In the second technique, (rather than writing directly in hardware description language) architecture is implemented in NIOS II Embedded Development Environment [37], where a softcore processor IP serving as an embedded microcontroller is deployed on the FPGA. In the following two sub-sections, the differences between these approaches will be discussed and their

performances shall be evaluated using the joint-state trajectories (see Figure 4-10) generated for the PUMA manipulator.

5.5.1 Hardwired Approach

In this approach, the proposed command generation algorithm is realized by a finite state machine (FSM) which is directly designed through the use of VHDL. During this phase, the schematic design property of Quartus II 9.0 Web Edition is used. With this property of the software, it is much easier to sustain and keep track of the communications among different modules performing specific tasks. The schematic of the design is illustrated in Figure 5-9. In this design, there are mainly six modules: SRAM Controller, Memory Management Unit (MMU), Decoding Unit (DU), Accumulators, Interpolator, and RS-232 Module. Before explaining each module, the allocated resources on FPGA while implementing the method are represented in Table 5-4. As can be seen, only 9% of the total logic

Table 5-4 FPGA Resources used in Hardwired Approach

Total Logic Elements	1731 (9%)
Total Combinational Functions	1491 (8%)
Dedicated Logic Registers	911 (5%)
Total Registers	911
Total Pins	50 (16%)
Total Memory Bits	0
Embedded Multipliers 9-bit Elements	2 (4%)
Total PLLs	0

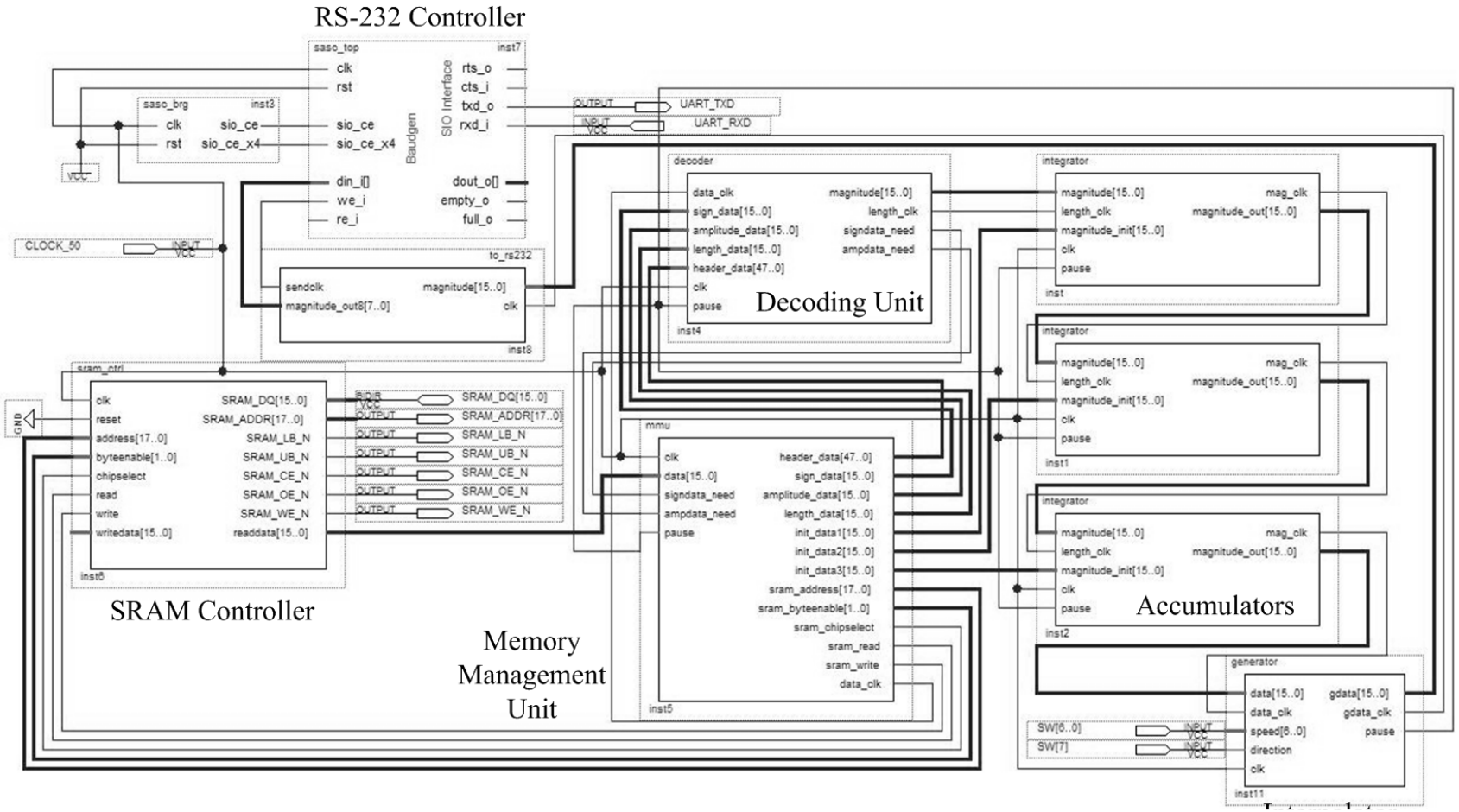

Figure 5-9 Hardwired FPGA Implementation of the ΔY Method

elements available on the FPGA chip are utilized in the design. Note that the FSM is implemented to generate commands for a single manipulator joint controller. Hence, one can implement 6 parallel FSMs to produce the commands for all joints of the PUMA manipulator without exhausting the resources of the chip. The number of pins used is a bit higher than ones in the implementation of the polynomial approximation methods. The reason of this increase is that the FPGA chip needs to be connected to the SRAM chip on the development board to store compressed (joint state) commands. For a better illustration of resource allocation, the floor plan of the "synthesized" digital circuitry on the chip is illustrated in Figure 5-10. As can be seen, only a small portion at the center of the chip is deployed to realize the corresponding architecture [38].

In order to evaluate the performance of the implementation in a detailed manner, the method is applied for all the trajectories of the manipulator (Figure 4-10). The results are given in Table 5-5. In the following sub-sections, the six main modules used in the design are investigated in detail.

Table 5-5 Time for the Generation of Command Sequences

Joint Number	Clock Cycles	Time (ms)
1	14569	0.29
2	15609	0.31
3	14039	0.28
4	13865	0.27
5	16045	0.32
6	15434	0.30

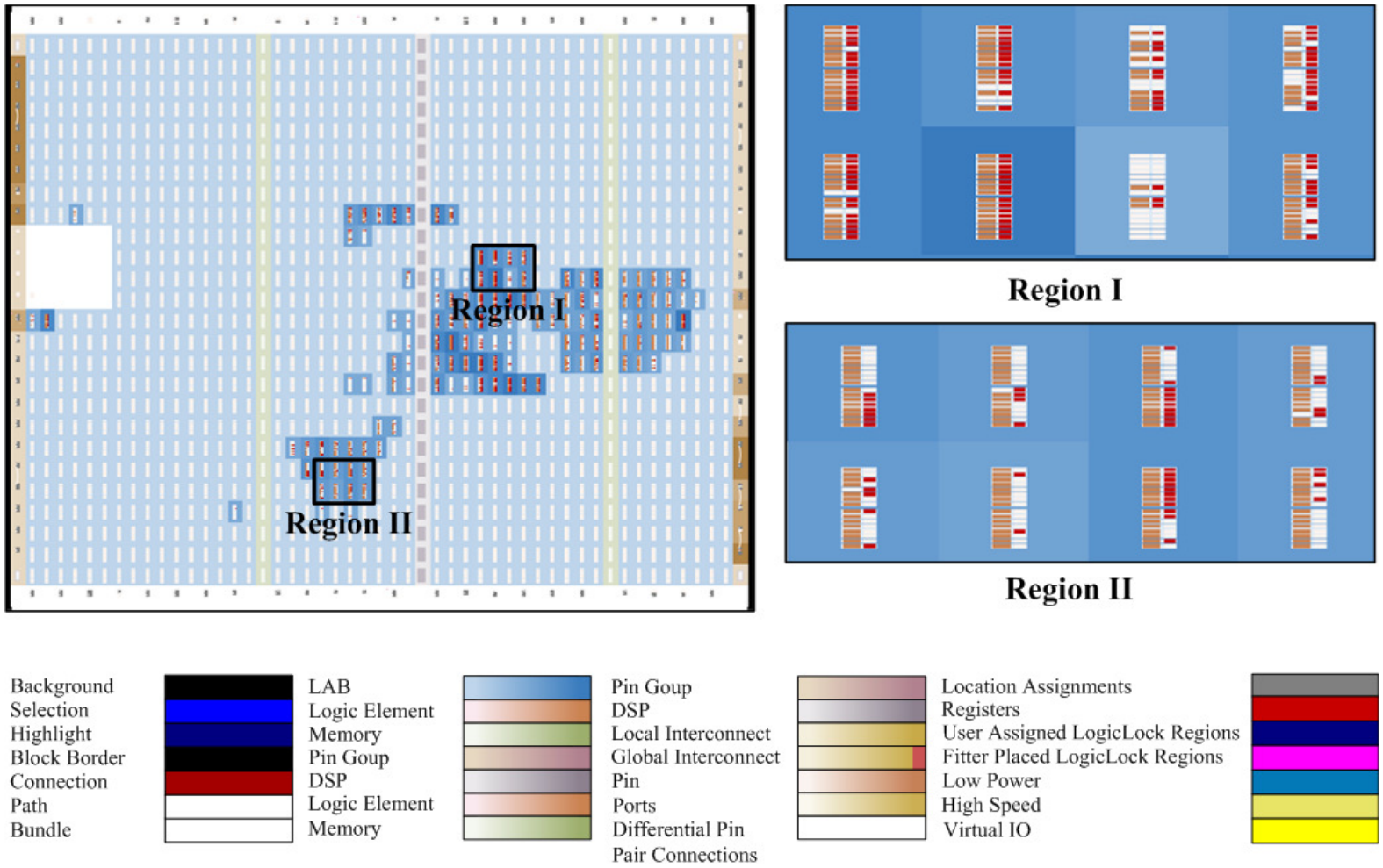

Figure 5-10 Floor Plan of the Synthesized Digital Circuitry for the 1st Architecture

5.5.1.1 SRAM Controller

The main task of the SRAM controller is to maintain the communication between the MMU and the SRAM located on the FPGA Development Board. It sends out the compressed data to the MMU (one by one in this case) according to the address information emanating from the MMU. Schematic version of the module is shown in Figure 5-11. All of the outputs of the module except `readdata` are connected to the SRAM chip on the development board, which is organized as 256K words by 16 bits. The output `readdata` is directly connected to the MMU. This output is responsible for sending the data on the specified address of the memory. The inputs `clk` and `reset` are the global clock and reset pins fed to the

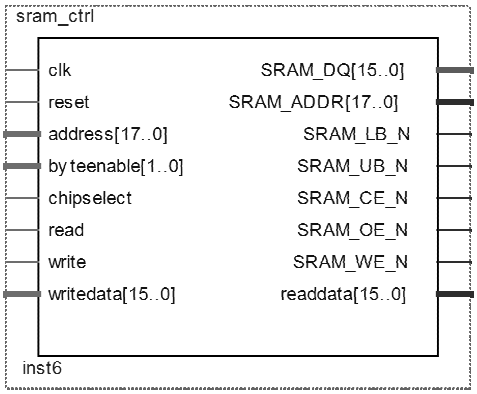

Figure 5-11 SRAM Controller

module. The rest of the inputs are connected to the outputs of the MMU. The `address` input to the module determines the value of output data to the MMU. `byteenable`, `chipselect`, and `read` inputs are set to high during to operation to be able to use the module. `write` input is set to low since there is no writing operation during the decoding process of the compressed data. In order to use the memory efficiently, the compressed code is structured as shown in Figure 5-12 for a generic command sequence. The first three words of

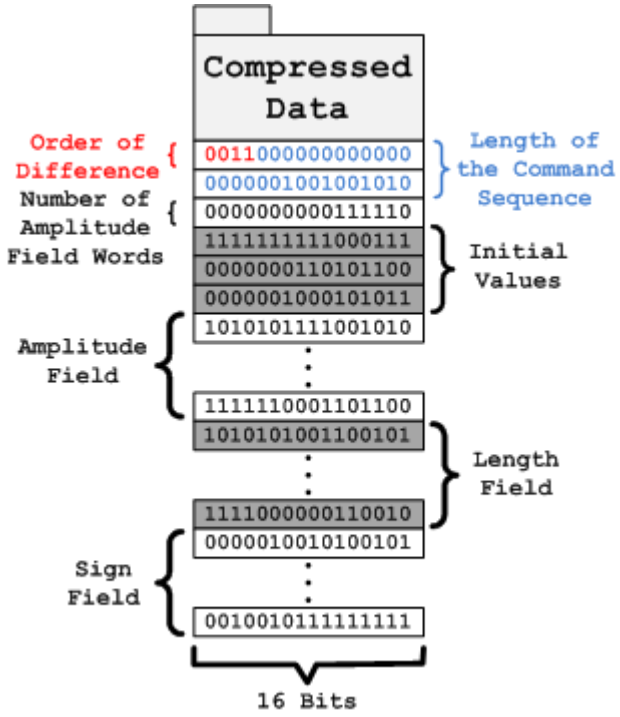

Figure 5-12 Compressed File Format

the compressed data can be regarded as the header. Initial 4 bits of the first word indicate the order of finite difference (where a maximum of 15^{th} order for the differences can be represented). The rest of the first word and the second word (28 bits) are reserved for expressing the length of the command sequence. Finally, the last word of the header is used to specify the number of words reserved for the magnitude field, which indirectly determines the starting address of the sign field. After the header part, the initial values section is located. They are stored in the form of signed binary integers. The number of initial values necessary for integration is set by the order of finite difference which is represented with the first 4 bits of the data. After the information about the compressed data and initial values are given, the amplitude field is then stored in the proceeding words. Since the length of the header part and the number of initial values are known, the starting address of the amplitude field is easily determined during decoding. Note that the length and sign fields are located after the amplitude field. The starting addresses of these two fields are calculated via the number of amplitude field

words stored in the third word of the compressed data. With the described data format, the compressed sequences are generated without any error.

5.5.1.2 Memory Management Unit

The MMU, whose schematic is shown in Figure 5-13, can be regarded as the core of the design since it communicates with all modules except the RS-232 Controller. Input signals to this module are limited when the number of output signals is considered. Input signals are only the data sent from the SRAM Controller (`data`), acknowledgment signals (`signdata_need` and `ampdata_need`) coming from the DU indicating that the unit is out of data, and the `pause` input set by the interpolator. Output signals are header data for the compressed file, three fields transferred to the DU, the initial values sent to the accumulators, and the necessary outputs connected to the SRAM Controller.

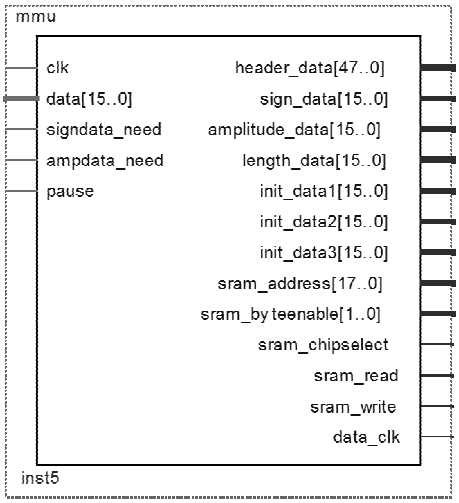

Figure 5-13 Memory Management Unit

The basic operating principles of the MMU are described in Figure 5-14. As can be seen, there are four states of this unit: i) *Get Header*, ii) *Get Initial Conditions*, iii) *Fetch First Set*, iv) *Send & Wait*. After the system is reset, the unit starts

acquiring header data (words) from the SRAM and sending them to the DU. In the next state, the initial values are conveyed to the accumulators in a proper order. In the following state, the first set of words from amplitude-, length-, and sign fields are fetched from the SRAM and are sent to the DU to initiate the decoding process promptly. In *Send & Wait* state, the words from each field are sent to the DU. This state is only initiated when the incoming signals `signdata_need` and `ampdata_need` are set. It should be noted that there is no signal indicating the necessity for a data point from the length field. When a word from the amplitude field is needed, the corresponding word from the length field is sent automatically to the DU. This state lasts until all the commands are generated.

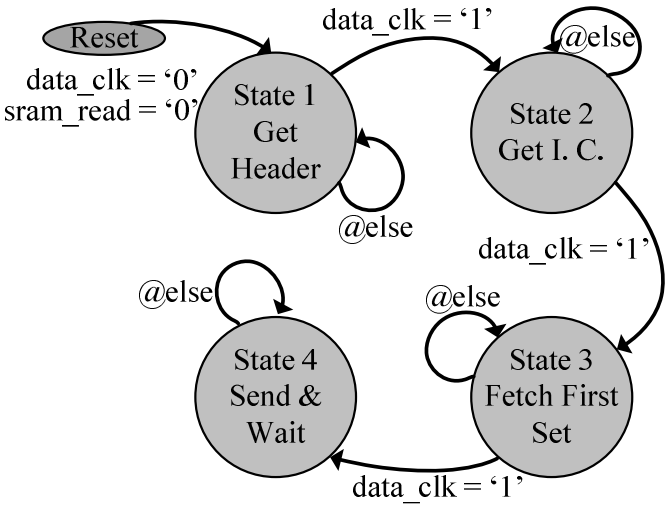

Figure 5-14 State Diagram of Memory Management Unit

5.5.1.3 Decoding Unit

DU is the module where the decoding algorithm is implemented. It communicates with the MMU, the first accumulator module, and the integrator module. The schematic design of this module is provided in Figure 5-15. All the input signals to this unit except the `clk` and `pause` signals are fed from the MMU. Two of the output signals are the acknowledgement signals (`signdata_need` and `ampdata_need`) which are described in the previous sub-section. The remaining two output signals are directly connected to the first accumulator. Thus, the decoded command is transferred to the accumulator in signed integer format at an additional clock indicating that a new command is being submitted.

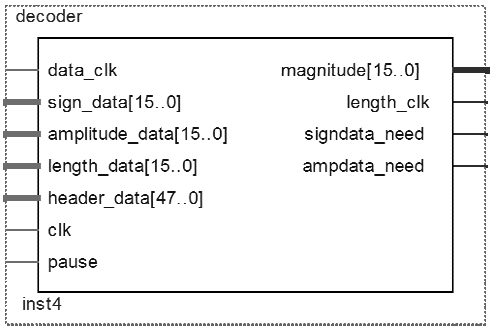

Figure 5-15 Decoding Unit

The basic operating principles of the DU are depicted in Figure 5-16. Decoding that constitutes nine states starts when the header data from the MMU are acquired. Then, the header data (constituting the order of difference, length of the command sequence, and the number of amplitude field words) are divided and stored for further use. In the second state, the first set of words from three different fields is saved. Second set received from the MMU is

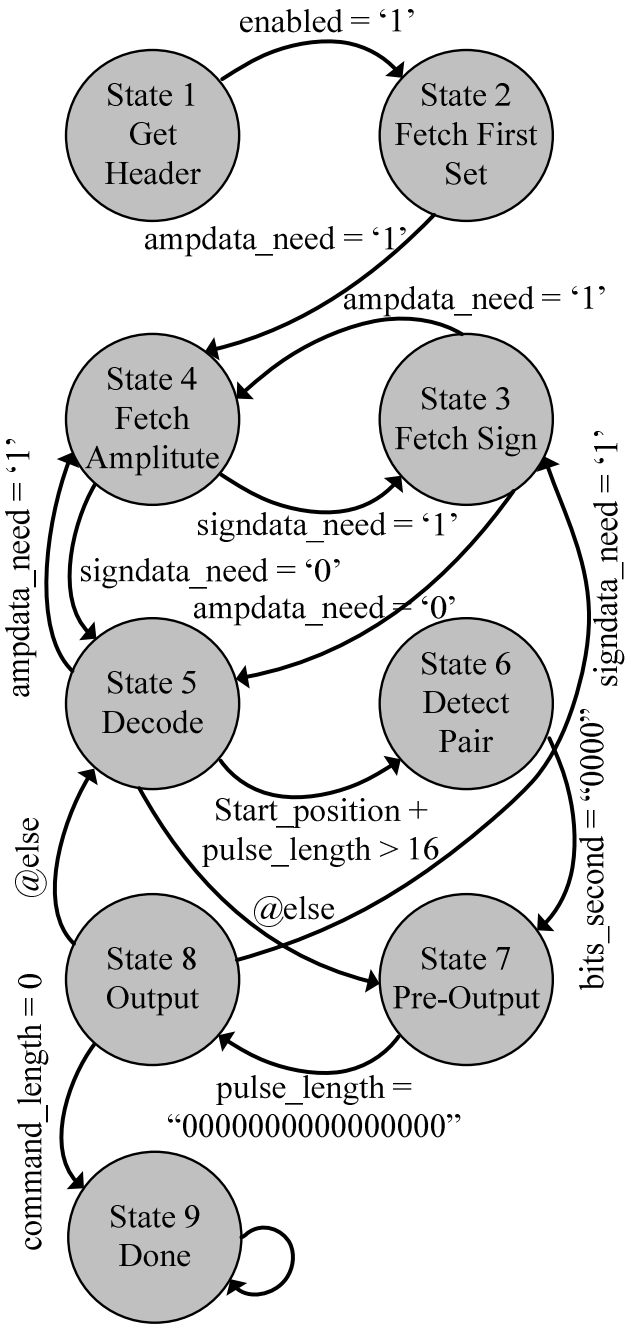

Figure 5-16 State Diagram of Memory Unit

stored in the third and fourth states. While decoding, the second set is necessary since it may turn out that the corresponding command is distributed between two consecutive words. The main task of this unit is executed in the *Decode* state of which is associated with five other states. When there is a lack of data during decoding, the finite-state "decoding" machine moves either on to *Fetch Amplitude* or *Fetch Sign* states to obtain the required data. If the decoding is complete for a given command, the data (in unsigned integer format) are processed in the *Pre-Output* state. In case the corresponding command is stored in two different words, *Detect Pair* state takes over for proper decomposition. Note that in *Pre-Output* state, the decoded command is rolled into a single word and passed onto the *Output* state. The conversion of unsigned to signed integer format is performed in the *Output* state. For this purpose, the data from the sign field must be ready. When sign data run out, the DU moves onto the third state and gets the necessary data. After the decoded command is formed as signed integer, it is sent to the first accumulator instance.

5.5.1.4 Accumulators

Accumulator (integrator) modules (Figure 5-17) are the simplest elements of this design. It gets the input data, sums it with the previous value of the accumulator and outputs the resulting value to the next accumulator. The number of accumulators in the design depends on the order of difference. Note that the given design in Figure 5-9 is hardwired and can decompress data differentiated up to the third order. However, the general design should have 15 accumulator instances (in compliance with the format specified in Section 5.5.1.1). A de-multiplexer unit must be incorporated to the design to deselect the unused accumulator instances. Notice that in the proposed design, the three accumulators yield the acceleration, velocity, and position profiles of the commanded trajectory. This attribute is one of the advantages of the proposed method. Since when a state-space controller is embedded into the system, the velocity and acceleration profiles must be ready for use.

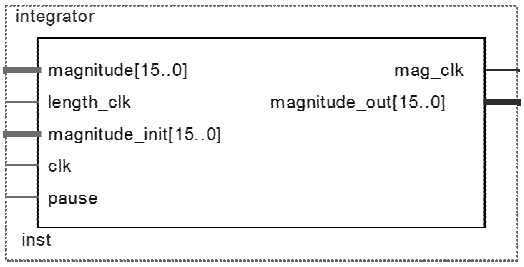

Figure 5-17 Accumulator Module

Instead of using a series of accumulators in the design, only one integrator can be used alternatively, whose schematic design is shown in Figure 5-18. In this design all the initial values are fed to this module at the beginning of decoding process. Integration formula for the third order difference is employed within the module and its result is sent to the interpolator module.

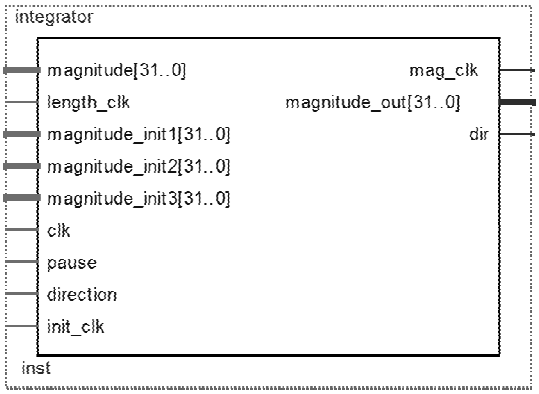

Figure 5-18 Integration Module

5.5.1.5 Interpolator

The interpolator used in the design, Figure 5-19, simply performs the computations described with the equations given in Section 4.4. While generating the commands, it sends `pause` signals to the DU, MMU, and accumulators to stop their operations. When there is need for a new original command, it sets the `pause` signal to low. Internal inputs to this module are data and its clock coming from the last accumulator module, and the global clock used in the system. External inputs are the ones given by the user and these are the feed-rate of generation and its direction. In order to overcome the delays between the generated commands, it also employs a buffer inside.

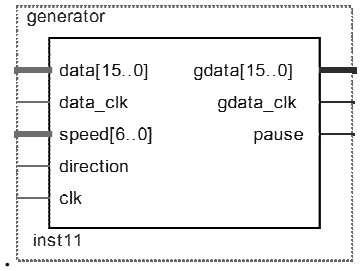

Figure 5-19 Interpolator Module

5.5.1.6 RS-232 Controller

The RS-232 Controller used in this architecture is the same with the one used in the implementation of the method based on Chebyshev polynomials approximations in the previous chapter.

5.5.2 Embedded Softcore Processor Approach

In the second approach of the FPGA implementations, the command generation algorithm is written in C programming language and the resulting are cross-compiled to run on a softcore processor deployed on the FPGA as done in the previous chapter. Then the resulted code is downloaded to the designed processor. Schematic design of this *algorithmic state machine* (ASM) is shown in Figure 5-20. The main difference of this design from the one designed in Chapter 4 is that there is a parallel input-output port in the softcore processor. With this property, a variable feed-rate input can be supplied to the system externally. As can be seen from Table 5-6, the hardware resources of this architecture is twice those of the hardwired architecture. It is critical to note that a sequential ASM is essentially implemented in this approach; there will not be a considerable increase in the resources when other trajectories are also generated. The memory required on the SDRAM will increase. The floor plan of the synthesized logic circuitry is illustrated in Figure 5-21. With the help of the performance counter module of the softcore processor, the time needed for decoding a sequence of 586 data points is roughly 25 ms.

Table 5-6 FPGA Resources used in Softcore Approach

Total Logic Elements	3549 (19%)
Total Combinational Functions	3146 (17%)
Dedicated Logic Registers	1984 (11%)
Total Registers	2036
Total Pins	46 (15%)
Total Memory Bits	28992 (12%)
Embedded Multipliers 9-bit Elements	0
Total PLLs	1 (25%)

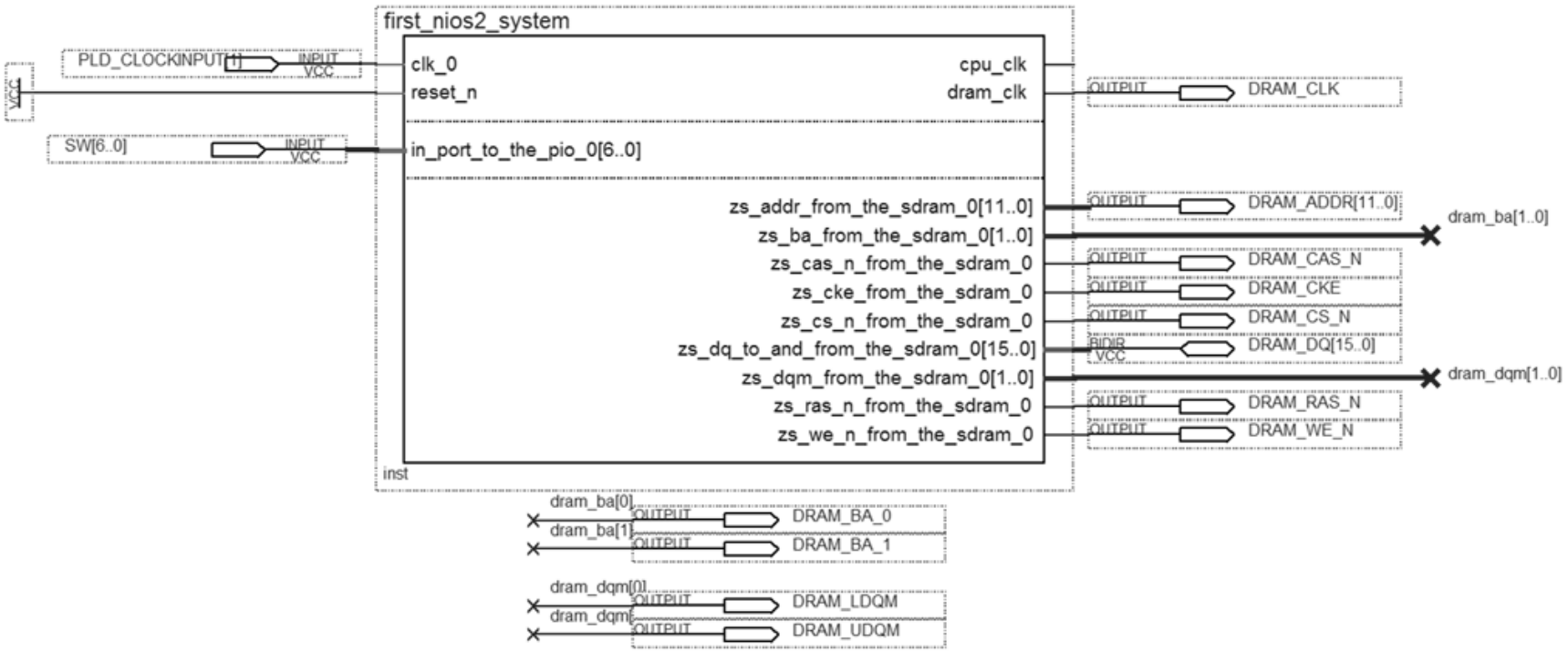

Figure 5-20 Implementation of the Method using Softcore Processor IP

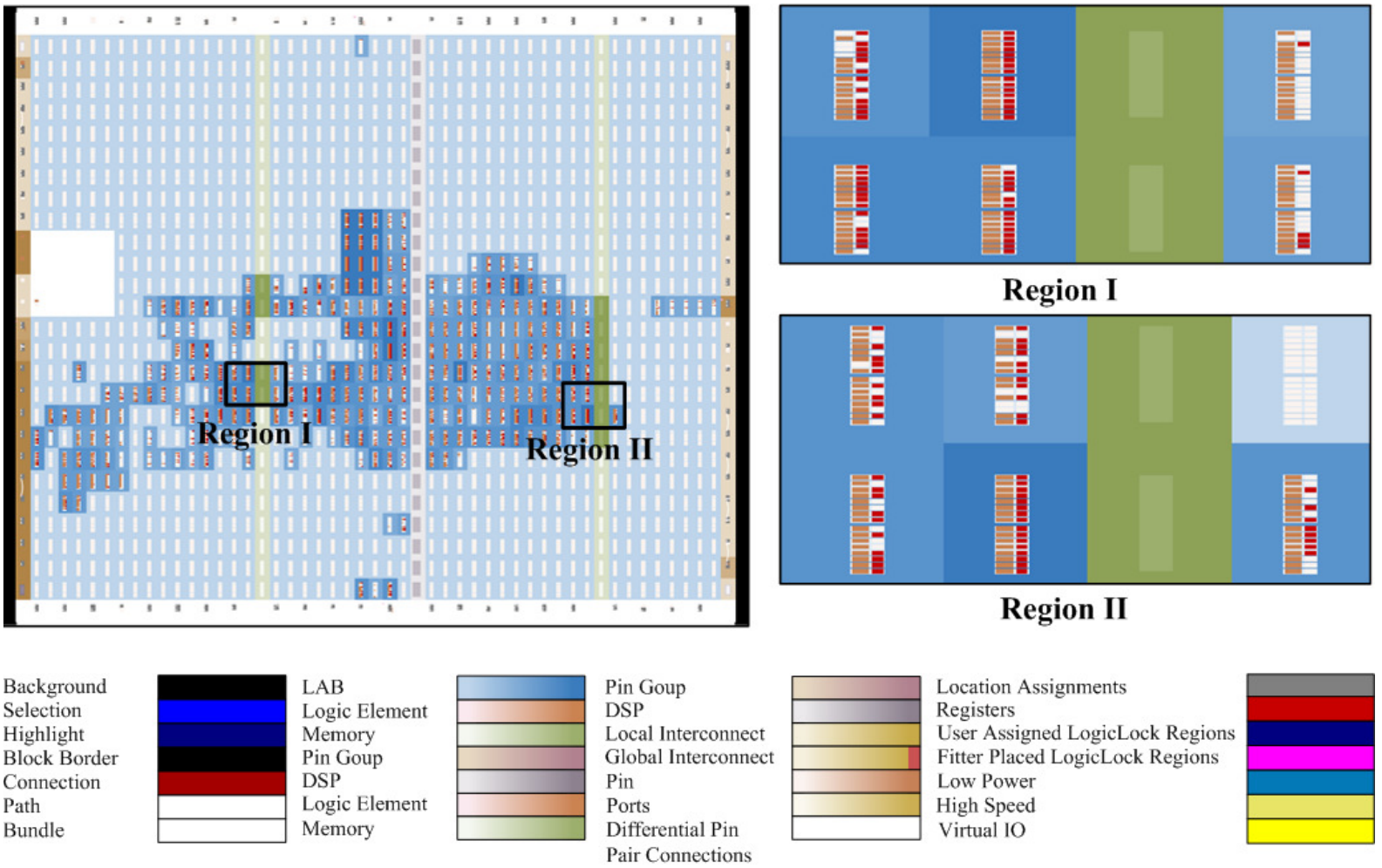

Figure 5-21 Floor Plan of the Synthesized Digital Circuitry for the 2nd Architecture

5.6 Closure

In this chapter, command generation method (a.k.a. ΔY) based on differencing and data compression techniques is proposed and implemented on the FPGA development board using two different approaches. During the performance evaluations, it is turned out that there is no sense in compressing the encoder pulses without taking higher order differences of them and proposed data compression technique is always more successful than the Huffman and Arithmetic coding methods. Taking higher order differences of the trajectory before compression is necessary. After differencing, the frequency of numbers in the sequence increases, (entropy based) data compression makes sense. Note that the novel compression method suggested in paper is not a universal. Its advantages reveal when the command sequence consists of integers showing acceleration and deceleration characteristics.

The hardwired FPGA implementation of the method outperforms the softcore embedded processor approach. Time need to generate same amount of commands for the softcore is about 100 times greater than the time needed by the hardwired architecture.

When the command generation method proposed in this chapter compared with the one proposed in the previous chapter, the former one is the most suitable one. Since in this case, less hardware resources are used and time necessary to generate commands is much lower than the method with polynomial approximation.

CHAPTER 6

CASE STUDY ON COMMAND GENERATION

6.1 Introduction

Up to this chapter, two different command generation methods are proposed and realized utilizing an FPGA development board (Altera DE1 with Cyclone II FPGA): **i)** Command Generation via Segmentation and Polynomial Approximation, **ii)** Command Generation via Differencing and Data Compression. When these two methods are compared, it can be concluded that the command generation method based on differencing and compression circumvents the other technique in terms of speed, resource utilization, compression ratio, and ease of implementation. Thus, during the case study, the performance of differencing and compression based method will be investigated through a detailed case study.

The chapter is organized as follows: the command sequences for a three-axis CNC vertical machining center are introduced. These sequences represent the desired cutting tool position when machining the injection mold of a bottle. After that, the trajectories are compressed with three different compression algorithms (Huffman, Arithmetic Coding, ΔY) after taking higher order differences. Once the compressed commands are generated, the FPGA implementation is carried out after some modifications on previously described schematic design. The results of the methods are compared and discussed at the final section of this chapter.

6.2 Sample Command Trajectory

To test the efficiency of the command generation methods on a realistic case, the manufacturing of a plastic injection mold for a bottle is taken into account. Despite the fact that the mold consists of two complementary parts (male/female), only the machining of the female (or negative) mold is considered in this work.

Before generating the commands for this specific machining task, the NC code, which is given in the Appendix D, are obtained from the sample codes of CncSimulator software [46]. It is critical to notice that an NC program is an industry-standard means of defining the trajectory of a programmable (CNC) machine system. Most CAD/CAM packages (after post-processing for a particular machine tool) do generate the NC code directly to carry out a particular machining task. From the functional point of view, the program describes the trajectory in terms of continuous linear- and circular segments using special (G) functions (such as G0, G1, G2/G2). That is, the code includes only the relevant parameters to define the trajectories in a piecewise fashion. For instance, to define a linear patch, only the destination (end-point) coordinates of this path needs to be specified along with the speed on the trajectory. Similarly, the destination/target coordinates as well as the radius of the curve may be sufficient to define an arc on a specific plane. On the other hand, the reference commands (position, velocity, acceleration) are to be supplied to the motion controller at each sampling period. Hence, the NC code must be processed (or interpolated) to generate the intermediate position data of the tool at equidistant time interval.

Utilizing the MATLAB script (`trajectory_generation.m`) developed by Akıncı [1], the command trajectories for the three axes {x, y, z} of the CNC machining centre are formed. These trajectories are shown in Figure 6-1 as a 3D plot. The trajectories along fundamental axes are illustrated in Figure 6-2, Figure 6-3, and Figure 6-4. Despite the fact that a NC code (by design) guarantees the C^0 continuity of the path, one needs to take into consideration not only the physical limitations of the power generating systems (electrical motors, drivers) but also

the requirements of the tasks. The velocity and acceleration/deceleration profiles along the given trajectory must be modified to account for these machine (or task) related issues. Note that, while generating the trajectory data, no attempt is made to maintain the C^1 continuity of the resulting trajectory for this test case. The velocity profiles can be seen in Figure 6-5, Figure 6-6, and Figure 6-7. Hence, some sharp changes in the velocity profile might be observed. As can be seen from the figures, the simulated manufacturing process lasts for 926 seconds. Since the sampling time is selected as 1 ms, 926000 commands are generated for each axis. During the generation of motor commands, it is assumed that one revolution of an axis-motor corresponds to 10 mm of translation along a particular axis.

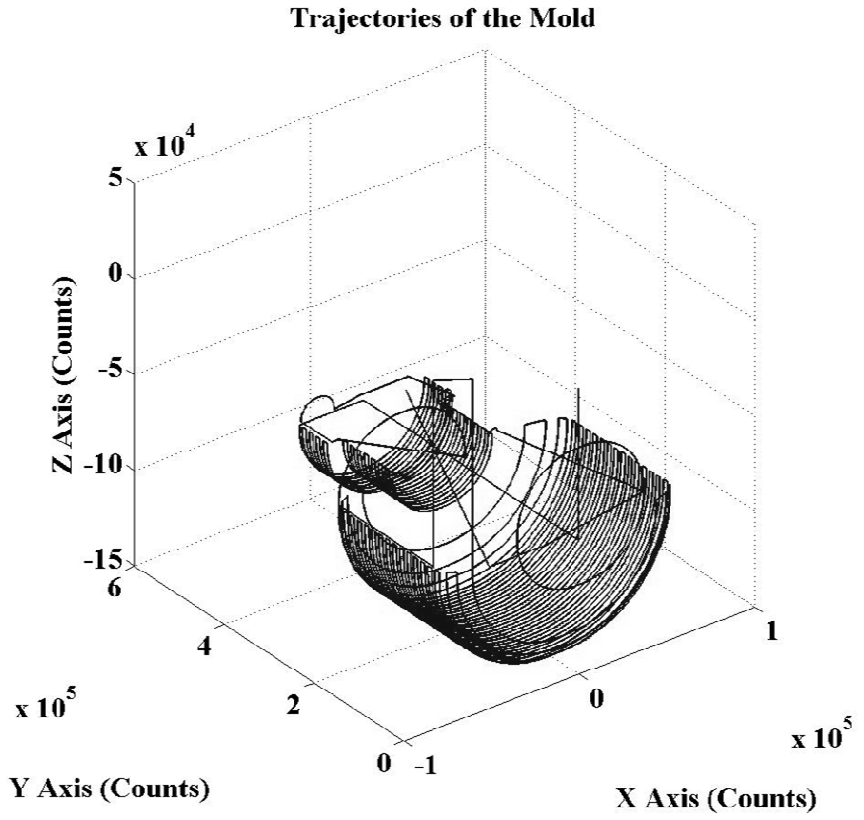

Figure 6-1 Trajectories of the Mold

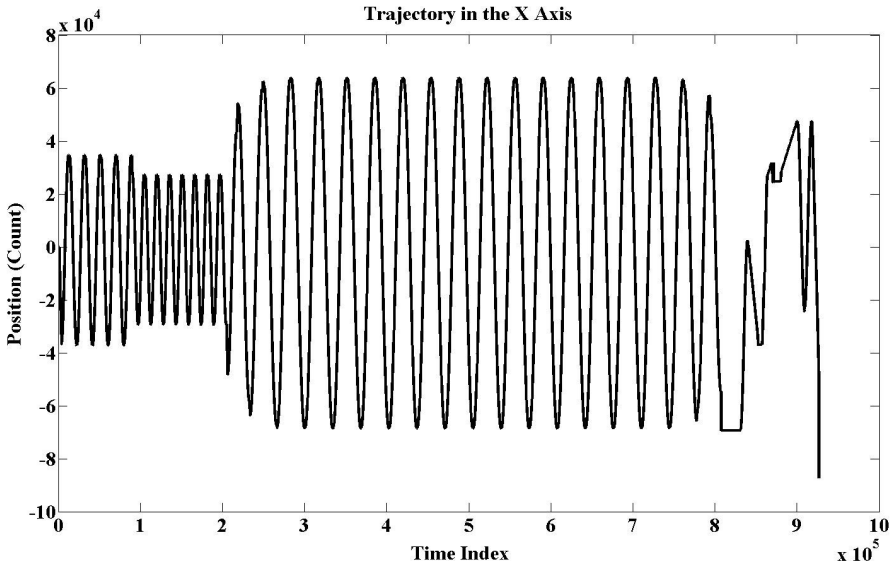

Figure 6-2 Trajectory in the X Axis

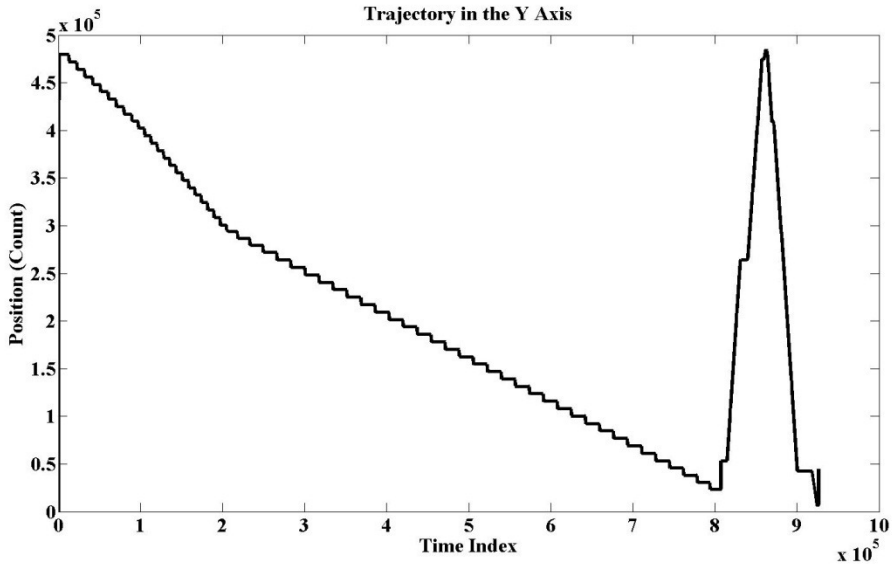

Figure 6-3 Trajectory in the Y Axis

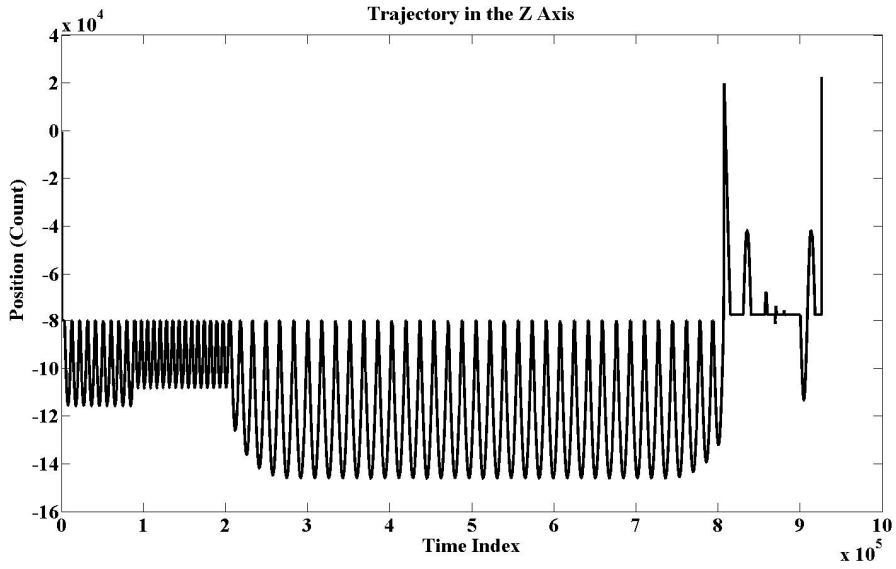

Figure 6-4 Trajectory in the Z Axis

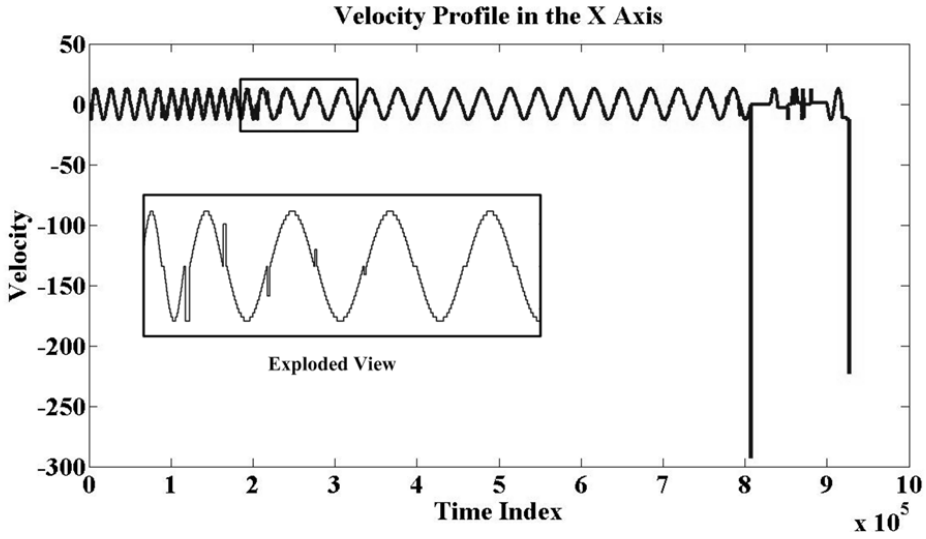

Figure 6-5 Velocity Profile in the X Axis

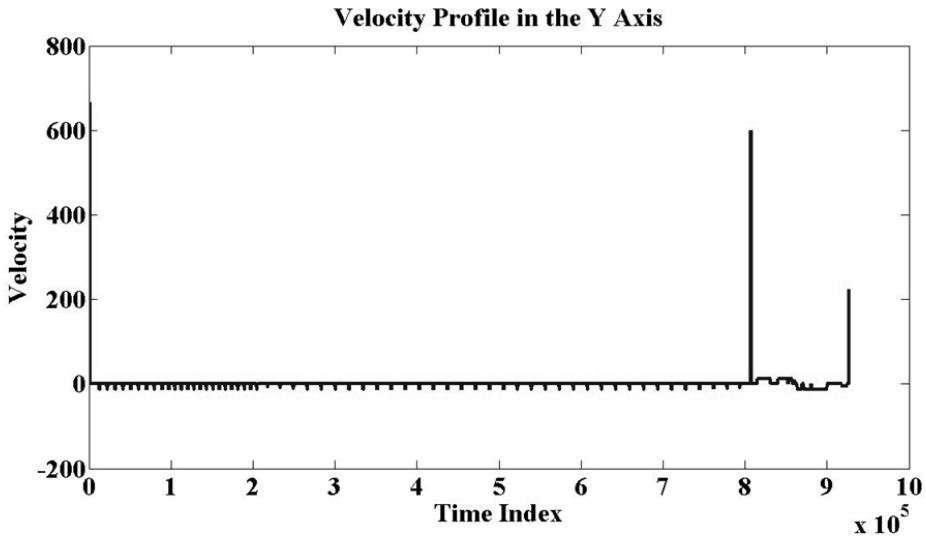

Figure 6-6 Velocity Profile in the Y Axis

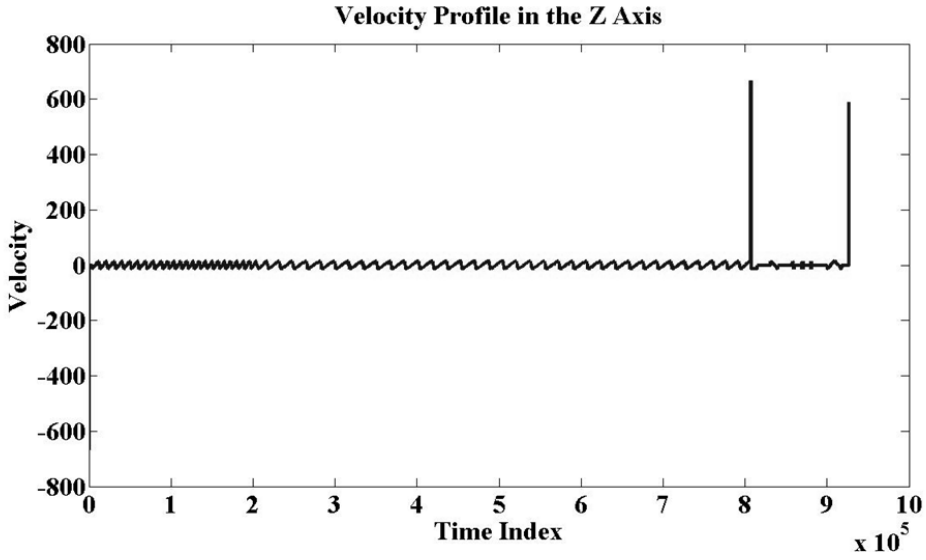

Figure 6-7 Velocity Profile in the Z Axis

6.3 Evaluation of Methods

In the scope of the thesis, two different command generation algorithms are introduced and implemented on the FPGA. To compare the methods (in terms of utilization of resources, compression of command data, and the generation time), the first trajectory in Figure 4-10 is reconsidered and the results are presented in Table 6-1. It can be inferred from the table that the first method uses nine times more FPGA resources than the second one. As stated in previous chapters, the reason of high consumption of the resources is the FPOMs used in the design – especially for multiplication of coefficients with polynomials. When the compression ratios are considered, the first method is also worse than the second approach. The first method cannot compress data as much; since for a reasonable approximation error, the number of polynomials as well as their corresponding coefficients should be kept high. This also causes the implemented design to generate the same amount of commands in longer time duration.

Table 6-1 Implementation Comparison of Proposed Command Generation Methods

	FPGA Resources [%]	**Compression Ratios [%]**	**Duration [ms]**
Segmentation and Approximation [I]	45	39.76	0.75
Differencing and Compression [II]	9	23.56	0.29

The first method generates 586 commands in 750 µs whereas the second method completes the generation process in 290 µs. Furthermore, there occur no representation errors in the commands for the second approach.

It is proven that the second method outperforms the first method in different aspects. Therefore, the trajectories presented in the previous article are encoded according to the second method. Before realizing the decoding on the FPGA, the trajectories are encoded with slight variations on the second method in MATLAB. First, the method is evaluated with only differencing. No further compression is employed on the differenced data. Table 6-2 represents the resulting compression ratios for differences up to sixth-order for all three axes of the given trajectory. The best performances are achieved for the first-order of difference regardless of the selected axis. These results are not totally in agreement with the data presented in Table 5-1 and Figure 5-1. Thus, a careful study on order of difference should be carried out when evaluating the complete method with various compression algorithms.

Table 6-2 Compression Ratios [%] vs Order of Differences for the Test Case

	Order of Difference					
	1	**2**	**3**	**4**	**5**	**6**
X Axis	50.00	55.56	55.56	61.11	66.67	72.22
Y Axis	50.00	55.00	55.00	60.00	65.00	70.00
Z Axis	61.11	61.11	61.11	66.67	72.22	77.78

In order to decide on the order of difference of the method for FPGA implementation, a further study on the trajectories is carried out. The Huffman, Arithmetic Coding, and the proposed compression (ΔY) algorithms are applied on the trajectories of the three axes for an order up to the sixth. The results of

compression algorithms for various orders are provided in Table 6-3, Table 6-4, and Table 6-5. Surprisingly, the arithmetic coding seems to be far superior to other compression algorithms regardless of the order and the trajectory.

Table 6-3 Results of Huffman Compression Algorithm for Various Orders of Difference [%]

Axis	**Order of Difference**					
	1	**2**	**3**	**4**	**5**	**6**
X	25.62	5.58	5.59	5.64	5.56	5.72
Y	6.29	5.01	5.01	5.02	5.02	5.03
Z	25.04	5.57	5.60	5.62	5.67	5.70

Table 6-4 Results of Arithmetic Coding Algorithm for Various Orders of Difference [%]

Axis	**Order of Difference**					
	1	**2**	**3**	**4**	**5**	**6**
X	25.36	0.13	0.23	0.36	0.46	0.59
Y	3.96	0.02	0.03	0.05	0.06	0.08
Z	24.72	0.13	0.25	0.36	0.49	0.08

Table 6-5 Results of the ΔY Compression Algorithm for Various Orders of Difference [%]

Axis	Order of Difference					
	1	**2**	**3**	**4**	**5**	**6**
X	32.83	10.53	10.53	10.55	10.58	10.66
Y	13.69	10.01	10.01	10.02	10.03	10.04
Z	32.21	10.53	10.54	10.57	10.60	10.69

This is mainly due to the high sampling frequency (1 kHz). As the sampling frequency decreases, the compression ratios of the algorithms approach to each other. The reason behind this fact is that the number of different values increases. Another conclusion to be drawn from the tables is that after the second order of difference, there are not any remarkable changes in the compression ratios.

Since compressing the original data to one-tenth of its original size seems to be adequate, the ΔY compression algorithm is selected for the implementation on the FPGA. Ease of implementation of the algorithm has a strong effect on this selection.

6.4 FPGA Implementation

In the previous section of the chapter, it is concluded that the second or third-order of differences before employing the proposed compression algorithm on the trajectories are the optimum orders for the implementation on the FPGA. In Chapter 5 implementation of the ΔY compression algorithm based command generation method is carried out via two different approaches, namely softcore (processor) and hardwired approaches. When the results of the techniques are

considered, it can be concluded that the hardwired approach is faster and expends less resources than the softcore approach.

Before realizing the generator for all axes, a decoder for the trajectory of x-axis is designed in Quartus II 9.0 (Web Edition) in order to compare the utilization of resources with the design given in Chapter 5. In this new design shown in Figure 6-8, several modifications on the modules are done. First of all, the widths of the registers in the modules are increased from 16 to 32 bits since it is not possible to represent the magnitudes of the commands with 16 bits. This conversion of registers affects the resource utilization in the FPGA. When the results given in Table 6-6 and Table 5-4 are compared, it can be inferred that with the doubled register sizes, the new design consumes more than twice the resources of the prior

Table 6-6 FPGA Resources used in Hardwired Approach of the Case Study for the First Axis

Total Logic Elements	4475 (24%)
Total Combinational Functions	3997 (21%)
Dedicated Logic Registers	1549 (8%)
Total Registers	1549
Total Pins	99 (31%)
Total Memory Bits	0
Embedded Multipliers 9-bit Elements	4 (8%)
Total PLLs	0

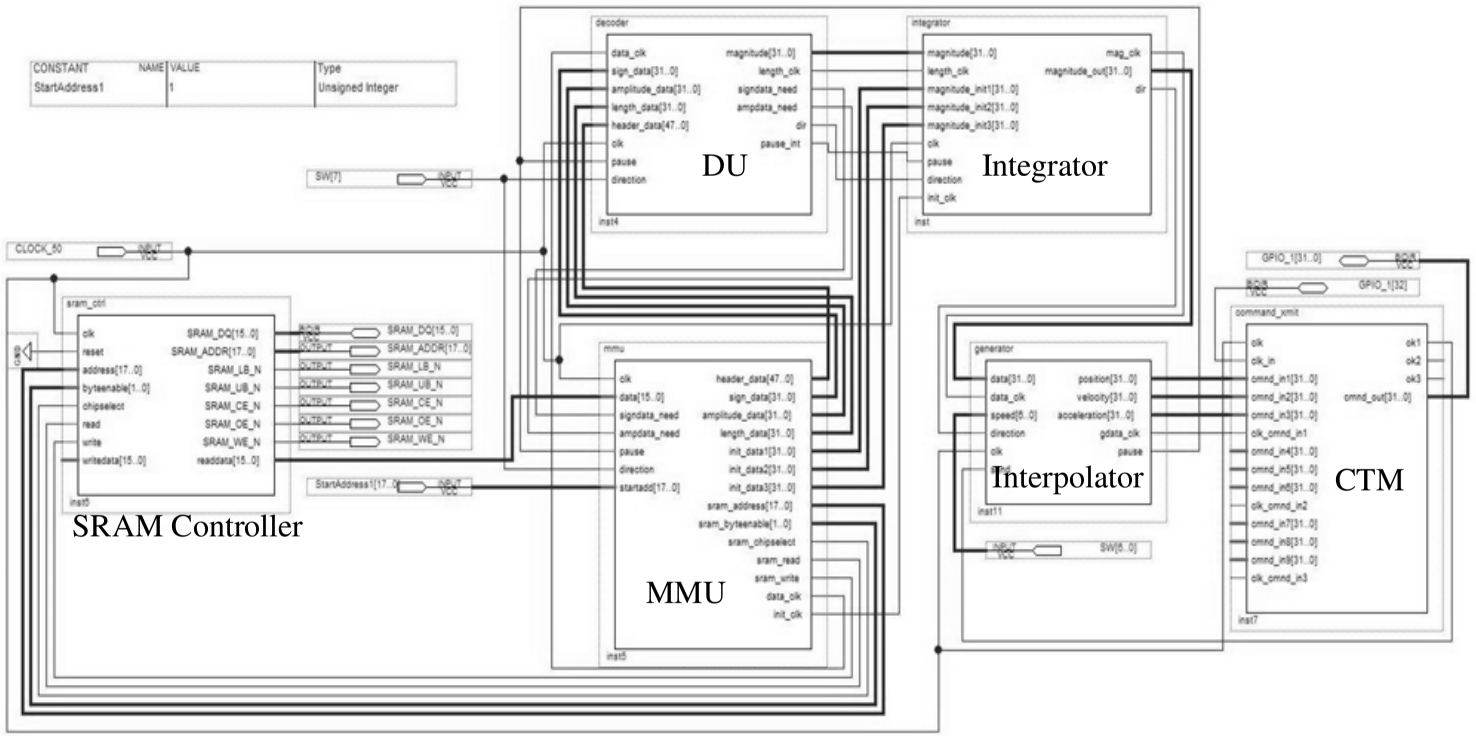

Figure 6-8 Hardwired FPGA Implementation of Command Generator for the X-Axis

design. Another modification in the design with respect to the one in Figure 5-9 is that instead of accumulator modules, an Integrator Module (IM) is embedded to the design. The main difference in this module (which shown in Figure 6-9) is that all of the initial values are directly transferred to this module at the beginning of the decoding process. Position, velocity, and acceleration profiles are fed with a data clock to the Command Transmit Module (CTM) shown in Figure 6-10 according to the acknowlegment signal `ok1` coming from the CTM connected to the `pause` input. The transferred commands are also depended on the current direction of decoding.

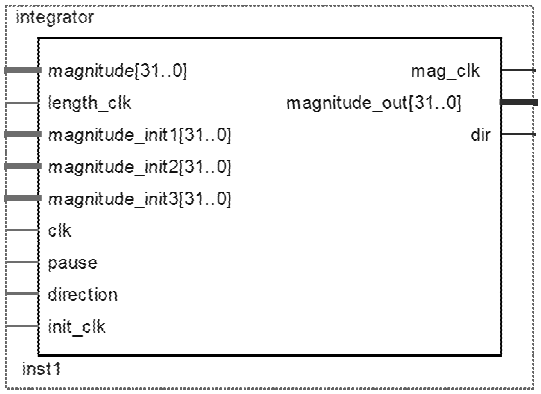

Figure 6-9 Integrator Module

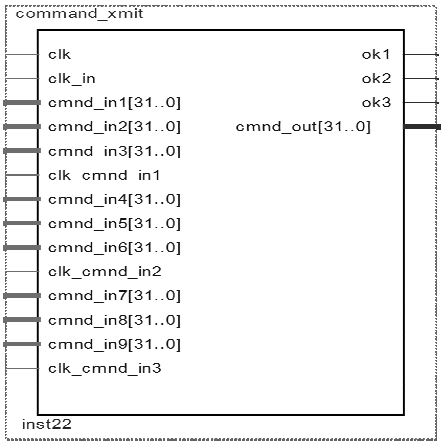

Figure 6-10 Command Transmit Module

Note that this module accepts all the commands from the three axes and sends these signals according to the incoming signal from the controller unit, `clk_in`. The incoming signal shown in Figure 6-11 is comprised of 9 consecutive clocks (at 10 MHz). These pulse sets are periodically generated at the sampling frequency of 1 kHz. At the rising edge of these clocks, the module sends commands to the controller and at the falling edge, the controller receives them.

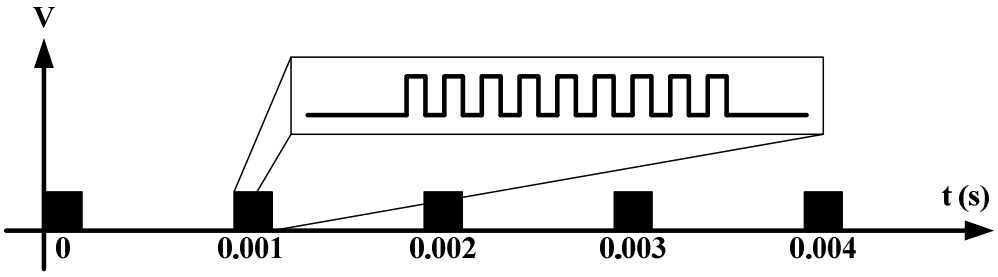

Figure 6-11 Incoming Clock Signal from the Controller

In the full design of the command generator for the case study, there should three separate command generators as illustrated in Figure 6-12. Note that there are three SRAM controllers in this design. However, the additional SRAM controllers do not increase the resource usage significantly since they are connected to the same pins of the FPGA chip. Another issue that should be noted is that the CTM is also responsible for prevents the conflicts that may arise during the communication of SRAM with the MMUs. The output of the CTM is connected to the General Purpose Input Output (GPIO) pins of the FPGA development board. Any controller can be connected to these GPIO pins to receive reference commands for the trajectories. The hardware resources utilized in this design is given in Table 6-7. When the table is compared in terms of the resources used for only x-axis, it can easily be inferred that the complete design

almost utilizes three times more resources than the previous one as expected. It is critical to notice that it was stated in Chapter 5 that all six trajectories of the manipulator can be generated using the FPGA development board. However, after changing the size of the registers to 32 bits, it is now possible to generate only four state trajectories.

Table 6-7 FPGA Resources Used in Hardwired Approach of the Case Study for all the Axes

Total Logic Elements	13366(71%)
Total Combinational Functions	12062 (64%)
Dedicated Logic Registers	4568 (24%)
Total Registers	4568
Total Pins	231 (73%)
Total Memory Bits	0
Embedded Multipliers 9-bit Elements	12 (23%)
Total PLLs	0

Finally, the chip floor plan of the synthesized circuit design is given in Figure 6-13. It can be seen that most of the resources are occupied as shown in Table 6-7. For a better illustration of the chip plan, two regions are zoomed. When the colors in the legend are considered, it can be concluded that the most of the chip is reserved for logic elements, connection of elements, and registers.

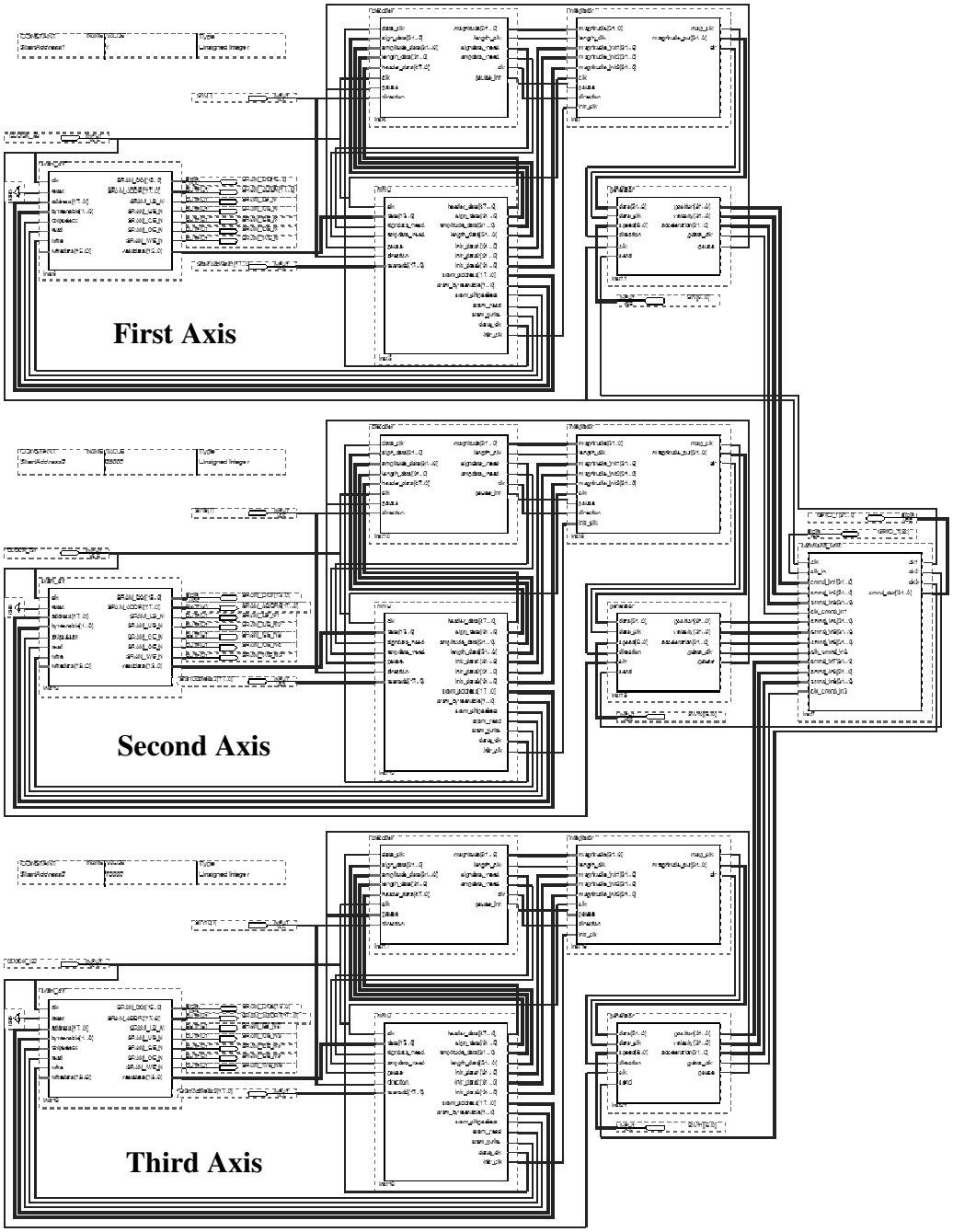

Figure 6-12 Hardwired FPGA Implementation of the Command Generator

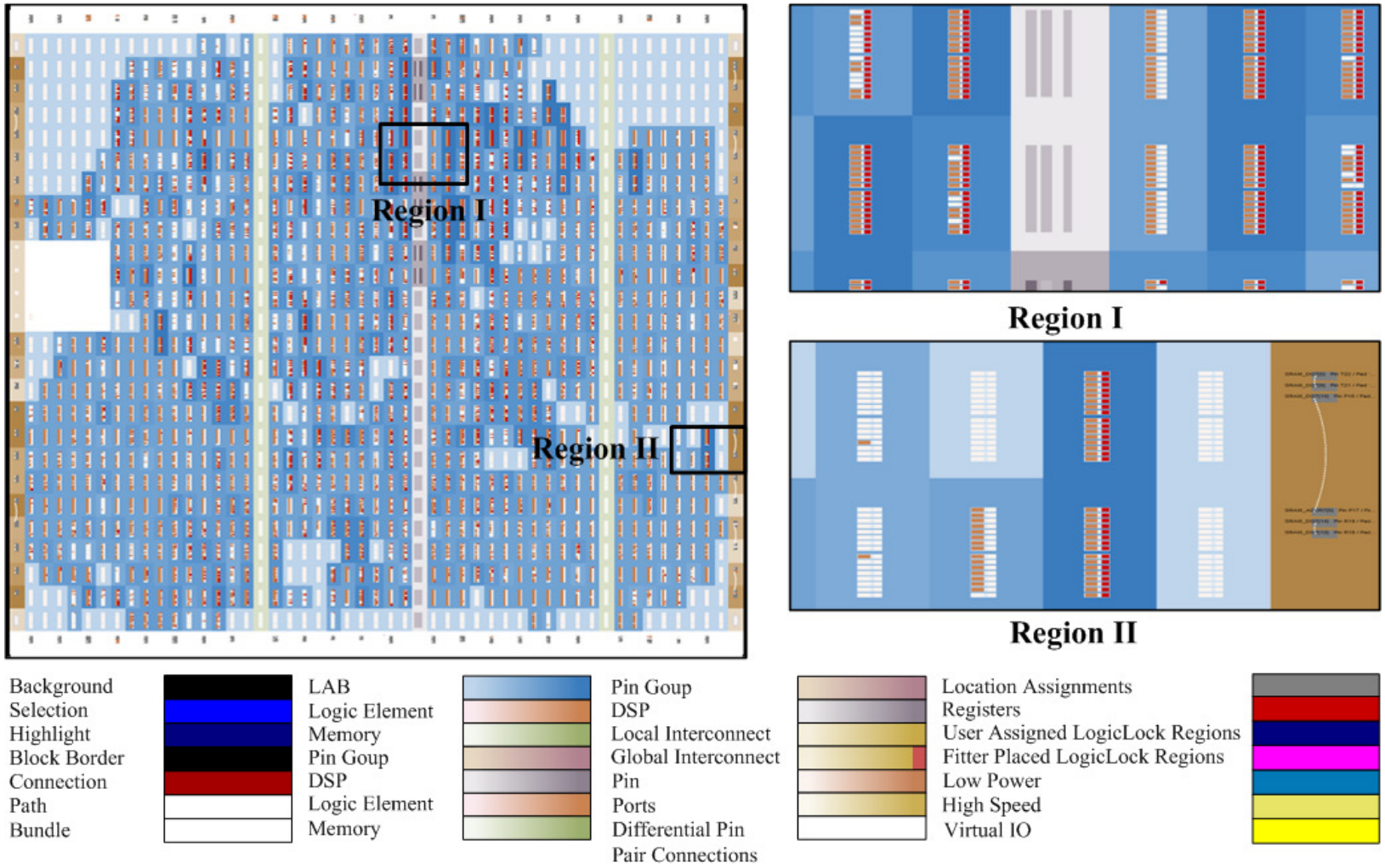

Figure 6-13 The Chip Floor Plan of the Synthesized Circuit Design

6.5 Closure

In this chapter, the most successful command generation method in the thesis, which consists of differencing and the ΔY data compression algorithm, is employed to generate the trajectories of a CNC vertical machining center while machining a plastic injection mold of a bottle. The selected method for FPGA implementation is further investigated in MATLAB and the results are discussed.

During the performance comparison of data compression algorithms, it is turned out that the Arithmetic coding algorithm outperforms its counterparts. The Arithmetic coding is not selected since the algorithm is not very suitable for parallel decoding. On the other hand, the ΔY is specifically developed for parallel decoding and its implementation on the FPGA is advantageous if compared to other compression methods.

The complete design built for the generation of all trajectories of the mold used 71% of the resources on the FPGA chip. If the number of axis increases in the system, the FPGA chip (Cyclone II) should be upgraded to (an advanced one like Cyclone III) generate all command trajectories.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this study, two different advanced command generators utilizing FPGA for computer controlled mechanisms have been developed. In both of the proposed command generators after the encoding of the trajectories are completed, the size of the original data is compressed to at least one-fourth of it.

The FPGA interface developed in the scope of the thesis can be regarded as the most important unit of the command generator system, since without the interface the system cannot be connected to the controller unit and the PC performing encoding operations. Beside providing communication with various devices, the interface is also used to convert digital signals to analog signals and vice versa. During these conversions, the voltage ranges are scaled and shifted according to the ranges set by the user.

Once the FPGA interface is introduced in the third chapter, the first command generation method is elaborated in the following chapter. This first method is basically an polynomial approximation algorithm. Since pure approximation does not give good results (compression ratios) for the complex trajectories. The method is modified such that the polynomial approximation is employed after segmenting the trajectory according to its inflection points and the representation

errors are stored after compressing them via proposed compression algorithm (ΔY). Then after the FPGA implementations, it is turned out that the hardwired approach uses more resources than the embedded softcore processor approach, but the command generation time of the hardwired approach is much better than the embedded softcore counterpart. Thus, the hardwired approach can be preferable for the systems where a high profile FPGA chip (Cyclone III, Stratix IV, etc.) is implemented.

The second FPGA based command generation method is composed of two parts: differencing and compression. During the encoding process, the trajectories are not directly compressed. This is due to the fact that the data values on the trajectory are completely different and compressing them is meaningless. Thus, taking higher order differences of the trajectories are inevitable if remarkable compression ratios are required to be achieved. The optimum order of difference is found to be three for almost all trajectories elaborated. For the second part of the corresponding command generation method, Huffman, Arithmetic Coding, and the ΔY compression algorithms are employed and evaluated. The FPGA implementations are carried out for the ΔY method, since it is much easier to implement and its performance is better than the other compression algorithms. The hardwired implementation of this command generation method outperforms the embedded softcore processor approach in utilization of resources and command generation time.

Comparing the proposed command generation methods in the thesis, it has been observed that differencing and the ΔY compression algorithm based method has presented the best compression ratio and the FPGA resource utilization. Hence, the case study is carried out with this method. In the case study, the manufacturing of a plastic injection mold for a bottle is taken into account. Although the mold has two complementary parts, only the female part is considered. For a detailed elaboration, the three compression algorithms are employed within the method to the trajectories of manufacturing process. It is turned out that the Arithmetic coding algorithm is much better than the other two algorithms, in contrast to the results obtained in the previous chapter of the thesis. Despite the superior

performance of Arithmetic coding, in FPGA implementations of the case study the ΔY compression algorithm is applied. The reason of this selection is that the implementation of ΔY method is much easier and its compression ratios are acceptable when the available resources on the FPGA board are considered. The ΔY method is also suitable for parallel decoding which increases the speed of command generation.

To summarize, the proposed FPGA based command generation system is faster than its counterparts and can be implemented to various control systems along with the interface developed in the thesis.

7.2 Future Work

In addition to the scope of the thesis there exist still some contributions that can be made on the topic. These can be classified as the improvements on the FPGA interface, encoded data transfer to the FPGA development board, and implementation of the Arithmetic coding algorithm on the FPGA.

For the FPGA interface part; instead of jumpers and switches used in daughter cards, fast analog switches that can transfer current in two ways are planned to be used. Furthermore, the configuration of channels (the selection of cards) may be completely done electronically via analog switches, and multiplexers. By utilizing the multilayer circuit printing technology and surface mountable electronical devices the final version of the interface can be designed and manufactured.

Another topic on which contributions can be made is the encoded data transfer to the decoder embedded on the FPGA development board. In the scope of the thesis, this topic is not elaborated much. For the data transfer, serial port of the computer is used. Design of a robust data transfer protocol is inevitable when the limited memory resources of the FPGA board are considered. During the command generation, the designed software should overwrite the old data according to the current status of generation. If the direction of command

generation is suddenly changed, then the designed software should restore the previous data.

The last further contribution can be done by elaborating the Arithmetic coding with various trajectories and sampling times, since in the case study it is turned out that the Arithmetic coding can compress the trajectories of the plastic injection mold for a bottle up to one-thousandth of their original size. In order to make a strong decision on the validity of the performance of Arithmetic coding, various trajectories should be studied. Implementation of the algorithm may also be a big contribution, since there exists no full decoder implemented on an FPGA chip for the Arithmetic coding algorithm.

REFERENCES

[1] Akıncı, A. "Universal Command Generator for Robotics and CNC Machinery," Middle East Technical University Graduate School of Natural and Applied Sciences, Master of Science Thesis, 2009.

[2] Yaman, U., Mutlu, B. R., Dolen, M., Koku, A. B., "Direct command generation for electrical servo motor drives," Proceedings of the 12th International Conference on Electrical Machines and Systems, IEEJ Industry Applications Society, Tokyo, pp. 1-6, November 2009.

[3] Sayood, K. "Introduction to Data Compression," Elsevier Inc, 2006.

[4] Reghbati, H. K. "Special Feature An Overview of Data Compression Techniques," Computer, vol. 14, no. 4, pp. 71-75, 1981.

[5] Balch, T., Khan Z., Veloso, M., "Automatically Tracking and Analyzing the Behavior of Live Insect Colonies," Proceedings of AGENTS'01, Montreal, pp. 521-528, 2001.

[6] Stearns, S. D. "Arithmetic Coding in Lossless Waveform Compression," IEEE Transactions on Information Theory, IT – 21, pp. 228-230, 1975.

[7] Dickson, K. "Cisco IOS Data Compression," San Jose, CA, USA, 2000.

[8] Rigler, S., Bishop, W., Kennings, A. "FPGA-Based lossless data compression using Huffman and LZ77 algorithms," Canadian Conference on Electrical and Computer Engineering, pp. 1235-1238, 2007.

[9] Huffman, D. "A Method for the Construction of Minimum-Redundancy Codes," Proceedings of the Institute of Radio Engineers, vol. 40, no. 9, pp. 1098-1101, September 1952.

[10] Rissanen, J. J., Langdon G. G., "Arithmetic Coding," IBM J. Res. Develop. vol. 23, pp. 149-162, 1979.

[11] Witten, I. H., Neal, R. M., Cleary, J. G., "Arithmetic coding for data compression," Communications of the ACM, vol. 30, pp. 520-540, 1987.

[12] S. W. Golomb, S. W., "Run Length Encodings," IEEE Transactions on Information Theory, vol. 12, pp. 399-401, 1966.

[13] Ostermann, J., Bormans, J, List, P., Marpe, D., Narroschke, M., Pereirra, F., Stockhammer, T., and Wedi, T., "Video Coding with H.264/AVC: Tools, Performance, and Complexity," IEEE Circuits and System Magazine, vol. 4, no. 1, pp. 7-28, 2004.

[14] Rigler, S., Bishop, W., and Kennings, A., "FPGA-Based lossless data compression using Huffman and LZ77 algorithms," Canadian Conference on Electrical and Computer Engineering, pp. 1235-1238, 2007.

[15] De Araujo, T. M. U., Pinto, E. R., De Lima, J. A. G., and Batista, L. V., "An FPGA implementation of a microprogrammable controller to perform lossless data compression based on the Huffman algorithm," 13th IBERCHIP Workshop, 2007.

[16] Abd El ghany, M. A., Salama, A. E., and Khalil, A. H., "Design and implementation of FPGA-based systolic array for LZ data compression," IEEE International Symposium on Circuits and Systems, pp.3691-3695, 2007.

[17] Cui, W., "New LZW data compression algorithm and its FPGA implementation," Picture Coding Symposium 2007, Lisbon (Portugal), Nov. 2007.

[18] H'ng, G. H., Salleh, M. F. M., and Halim, Z. A., "Golomb coding implementation in FPGA," Elektrika Journal of Electrical Engineering, pp. 36-40, 2008.

[19] Yongming, Y., Jungang, L., and Jianmin, W., "LADT arithmetic improved and hardware implemented for FPGA - Based ECG data compression," Proceedings of 2nd IEEE Conference on Industrial Electronics and Applications, pp.2230-2234, 2007.

[20] Valencia, D., and Plaza, A., "FPGA-Based hyperspectral data compression using spectral unmixing and the pixel purity index algorithm," Computational Science, pp.881- 891, 2006.

[21] Lee, D., Luk, W., Villasenor, J., and Cheung, P., "Hierarchical Segmentation Schemes for Function Evaluation," Proceedings of IEEE International Conference on Field-Programmable Technology, pp. 92-99, 2003.

[22] Lee, D., Cheung, C. C., Luk, W., and Villasenor, J. D., "Hardware Implementation Trade-Offs of Polynomial Approximations and Interpolations," IEEE Transactions on Computers, vol.57, no.5, pp.686-701, May 2008.

[23] Michard, R., Tisserand, A., and Veyrat-Charvillon, N., "Small FPGA Polynomial Approximations with 3-bit Coefficients and Low-Precision

Estimations of the Powers of X," Proc. 16th IEEE Int. Conf. On Application-Specific Systems,Architecture and Processors, pp.334-339, 2005.

[24] Ashrafi, A., Adhami, R., Joiner, L., and Kaveh, P., "Arbitrary Waveform DDFS utilizing Chebyshev Polynomials Interpolation," IEEE Transactions on Circuits and Systems I: Regular Paper, vol. 51, no. 8, pp. 1468-1475, 2004.

[25] Sodagar, A. M., and Lahiji, G. R., "A Pipeline ROM-Less Architecture for Sine-Output Direct Digital Frequency Synthesizers Using the Second-Order Parabolic Approximation," IEEE Transactions on Circuits and Systems. II, vol. 48, no. 9, pp. 850-857, 2001.

[26] Ashrafi, A., Pan, Z., Adhami, R., and Wells, B. E., "A Novel ROM-less Direct Digital Frequency Synthesizer Based on Chebyshev Polynomial Interpolation," Proc. of the 36th Symposium on System Theory, pp. 393-397, 2004.

[27] Su, K. H., Hu, C. K., and Cheng, M. Y., "Design and Implementation of an FPGA-based Motion Command Generation Chip," Proceedings of IEEE International Conference on Systems, Man, and Cybernetics, 2006.

[28] Jeon, J. W., and Kim, Y. K., "FPGA based acceleration and deceleration circuit for industrial robots and CNC machine tools," *Mechatronics*, vol. 12, pp. 635-642, 2002.

[29] Jeon, J. W., "An efficient acceleration for fast motion of industrial robots," Proceedings of IEEE 21st IECON, pp. 1336–41, 1995.

[30] Cheng, C. W., Tsai, M. C., and Maciejowski, J., "Feed-rate control for non-uniform rational B-spline motion command generation," Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture, vol. 220, pp. 1855-1861, 2006.

[31] Cheng, C. W., Tsai, M. C., and Cheng, M. Y., "Real-time variable feed-rate parametric interpolator for CNC machining," 15th IFAC World Congress, Barcelona, Spain, 2002.

[32] Xu, H. Y., Tam, H. Y., Zhou, Z., and Tse, P. W., "Variable feed-rate CNC interpolation for planar implicit curves," Advanced Manufacturing Technology, vol. 18, pp. 794 - 800, 2001.

[33] Cody, W. J., "A survey of practical rational and polynomial approximation of functions," SIAM Rev., vol. 12, no. 3, pp. 400–423, 1970.

[34] Vanicek P., "Approximate Spectral Analysis by Least-squares Fit," Astrophysics and Space Science, pp.387–391, Volume 4, 1969.

[35] Boyd, J. P., "Chebyshev and Fourier Spectral Methods," Dover Publications, Inc., 2000.

[36] Altera DE1 FPGA Development and Education Board User Manual, Altera Co., v1.1, 2006.

[37] NIOS II Processor Reference Handbook, Altera Co., 2009.

[38] U. Yaman, M. Dolen, and A. B. Koku, "A Novel Command Generation Method with Variable Feedrate utilizing FGPA for Motor Drives," Proceedings of 8th IEEE Workshop on Intelligent Solutions in Embedded Systems, Crete, Greece, 8-9 July 2010.

[39] R. Usselmann, "Open Floating Point Unit Manual," http://www.opencores.org, last accessed date: 06/06/2010.

[40] R. Usselmann, "Simple Asynchronous Serial Comm. Device," http://www.opencores.org/cores/sasc/, last accessed date: 06/06/2010.

[41] Menon, S. M., Bobra, Y. K., Ghia, A. V., and Zaliznyak, "Programmable Input/Output Circuit for FPGA for use in TTL, GTL, GTLP, LVPECL, and LVDS Circuits," US-Patent 6218858, 2001.

[42] Goetting, F. E., Frake, S. O., Kondapalli, V. M., and Young, S. P., "FPGA with a Plurality of I/O Voltage Levels," US-Patent 5877632, 1999.

[43] Chang, W. "Application Specific Field Programmable Gate Array," US-Patent 5687325, 1997.

[44] Proteus PCB Design Software, http://www.labcenter.co.uk/index.cfm, last accessed date: 07/06/2010.

[45] Driscoll, F. F., Coughlin, R. F., "Operational Amplifiers and Linear Integrated Circuits," 5th Edition, Prentice Hall College Div, 1997.

[46] CncSimulator, http://www.cncsimulator.com, last accessed date: 13/06/2010.

APPENDIX A

LIST OF VERILOG / VHDL MODULES

In this appendix, the Verilog / VHDL modules used in the study are listed. Brief descriptions of each module along with the related sections of the thesis are also presented.

Table A-1 List of Verilog / VHDL Modules Utilized in the Thesis

Name	Mentioned in	Description
`clk_div`	Chapter 3.4	Modifies the global clock used in the design according to the external input. If this module is paused, then the circuit also stops operating.
`pwmgen`	Chapter 3.4	Generates PWM signals according to the input vector.
`sine_wave`	Chapter 3.4	Generates sine waves by reading the values of quarter sine wave from the look-up table. The frequency of the sine wave is set by changing the global clock of the system.

sine_package	Chapter 3.4	It is look-up table holding the values of quarter sine wave according to the pre-defined resolution.
Driver	Chapter 4.4	Coordinates the modules used in the approximation architecture design with each other for proper operation.
Split_32_8	Chapter 4.4	Splits the 32-bit input to four bytes and sends these bytes to the serial communication module.
LookUpReader	Chapter 4.4	Reads the value of Chebyshev coefficients from the previously formed look-up table.
chebyshev_coef	Chapter 4.4	Holds the values of Chebyshev coefficients in binary form.
fpu_cadmusmod	Chapter 4.4	Performs mathematical floating point operations [38].
sasc_brg	Chapter 4.4	It is the baud-rate generator used for serial communication [40].
sasc_top	Chapter 4.4	It is the main controller of the serial communication [40].
sram_ctrl	Chapter 5.5	It is the SRAM Controller of Altera DE 1 Development Board [36].
decoder	Chapter 5.5	Decodes the encoded commands according to the ΔY compression algorithm and transmits the decoded commands to the first accumulator.

mmu	Chapter 5.5	This is the Memory Management Unit of ΔY decompression architecture. It receives the necessary data from the SRAM Controller and passes to other modules in the architecture.
accumulator	Chapter 5.5	Sums the incoming value with the previous one and sends to the corresponding module.
interpolator	Chapter 5.5	Interpolates between the two consecutive command values according to the feed-rate input.
integrator	Chapter 6.4	Includes the accumulators according to the order of difference in encoding.
command_xmit	Chapter 6.4	Receives the commands for position, velocity, and acceleration for all the axes and transmits these commands to the control unit of the system.

APPENDIX B

NIOS II EDS 9.0 C CODES

In this appendix, the C files used in NIOS II Integrated Development Environment are presented in Table B-1. The parts of the thesis they are used are also mentioned. In Table B-2, NIOS II C file for the ΔY decompression algorithm is presented.

Table B-1 List of NIOS II C Files Utilized in the Thesis

Name	Mentioned in	Description
`DYdecompression.c`	Chapter 5.5.2	Algorithmic state machine of the ΔY decompression algorithm
`ChebyshevApp.c`	Chapter 4.4.2	Algorithmic state machine of the Chebyshev approximation algorithm

Table B-2 NIOS II C file for the ΔY Decompression Algorithm

```
#include <stdio.h>
#define uint8 unsigned char
#define uint16 unsigned short
#define uint32 unsigned long
#define int8 char
#define int16 short
#define int32 long
#define SECTION1 1
int main(void)
{
    const int twos[16] = { 1, 2, 4, 8, ..., 16384, 32768};
    /*Field Declarations */
    uint8 sign[74] = {143, 128, 0, ..., 166, 1};
    uint8 amp[98] = {232, 67, 147, ..., 237, 0};
    uint8 term[98] = {240, 123, 28, ..., 238, 0};
    uint8 sign_ = 0;
    uint8 amp_ = 0;
    uint8 term_ = 0;
    uint32 original[586];
    float original_we[586];
    int32 l1 = 586; /*Length of the Sign Field */
    int32 l2 = 777; /*Length of Amplitude and Termination Fields*/
    int32 count = 0; /*Counts the length of the amplitude value.*/
    int32 i = 0;    /*For Loop Counter */
    int32 i1 = 0;
    int32 j = 0;   /*For Loop Counter */
    int32 k = 0;
    int32 k1 = 0;
    int32 r = 0;  /*Remainder*/
    int32 r1 = 0;
    int32 l = 0;  /*Original Data Counter*/
    int32 a = 0;  /*Original Data Counter*/
```

```
for(i=0; i<586; i++)
{
    original[i]=0;
}
i = l2 / 8;
r = l2 % 8;
term[i] = term[i] << (8-r);
amp[i] = amp[i] << (8-r);
i = l1 / 8;
r = l1 % 8;
sign[i] = sign[i] << (8-r);


while (k < l2)
{
    i = k / 8;
    r = k % 8;
    k1 = k;
    i1 = i;
    r1 = r;

    if (k == (l2-1))
    {
        if ((((term[i]<<r) & 128)==128))
        {
            term[i]=term[i] & (254<<(7-(r+1)));
        }
        if ((((term[i]<<r) & 128)==0))
        {
            term[i]=term[i] | (1<<(7-(r+1)));
        }
    }

    term_ = term[i] << r;

    if ((term_ & 128) == 128)
    {
```

```
            count ++;
            if (((r != 7) & (((term_ <<1) & 128) == 0)) |
((r==7)&((term[i+1] & 128) == 0)))
            {
                for (j=0; j<count; j++)
                {
                    amp_ = amp[i1] << r1;
                    if ((amp_ &  128) == 128)
                    {
                        original[l] += twos[j];
                    }
                    k1--;
                    i1 = k1 / 8;
                    r1 = k1 % 8;
                }
                count = 0;
                l++;
            }
        }
        else
        {
            count ++;
            if (((r != 7) & (((term_ <<1) & 128) == 128)) |
((r==7)&((term[i+1] & 128) == 128)))
            {
                for (j=0; j<count; j++)
                {
                    amp_ = amp[i1] << r1;
                    if ((amp_ &  128) == 128)
                    {
                        original[l] += twos[j];
                    }
                    k1--;
                    i1 = k1 / 8;
                    r1 = k1 % 8;
                }
                count = 0;
                l++;
```

```
            }
        }
        k++;
    }

    for (k=0; k<l1; k++)
    {
        i = k / 8;
        r = k % 8;

        sign_ = sign[i] << r;

        if ((sign_ & 128) == 0)
        {
            original[k]= original[k] ;
        }
        else
        {
            original[k]= - original[k] ;
        }
    }
    return 0;
}
```

APPENDIX C

MATLAB M-FILES

In this appendix, the MATLAB scripts (m-files) used in the study are explained briefly in Table C-1. Related sections of these files in the thesis are also noted. In Table C-2, MATLAB M-file for the ΔY compression algorithm is presented.

Table C-1 List of MATLAB M-Files Utilized in the Thesis

Name	Mentioned in	Description
`above_err.m`	Chapter 4	Counts the number of data values above the acceptable error margin.
`acc.m`	Chapter 5	Accumulates the sequence according to the initial values supplied to the function.
`bernstein.m`	Chapter 4	Generates Bernstein polynomials with various orders and lengths.

bp_approx.m	Chapter 4	Performs Bernstein approximation on the given trajectory.
bp_approx_errcomp.m	Chapter 4	First performs Bernstein approximation and then compresses the approximation errors.
chebypol.m	Chapter 4	Generates Chebyshev polynomials with various orders and lengths.
compviadiff.m	Chapter 5	Compares the compression ratios of differencing the input sequence for various orders.
cp_approx.m	Chapter 4	Performs Chebyshev approximation on the given trajectory.
cp_approx_errcomp.m	Chapter 4	First performs Chebyshev approximation and then compresses the approximation errors.
dacomp.m	Chapter 5	Performs Arithmetic coding algorithm on the given sequence for the specified order of difference and outputs the compressed code and memory requirements.

dhcomp.m	Chapter 5	Performs Huffman compression algorithm on the given sequence for the specified order of difference and outputs the compressed code and memory requirements.
dycomp.m	Chapter 5	Performs ΔY compression algorithm on the given sequence for the specified order of difference and outputs the compressed.
dydcomp.m	Chapter 5	Decompresses the encoded data with ΔY compression algorithm and outputs the original data sequence.
dydcompreverse.m	Chapter 5	Decompresses the encoded data with ΔY compression algorithm in reverse order and outputs the original data sequence.
gen_enc_pulses	Chapter 6	Generates sequences of encoder pulses from the tool location data sequences.
legendre.m	Chapter 4	Generates Legendre polynomials with various orders and lengths.

`lookupgen.m`	Chapter 4 & 5	Creates look-up file for Chebyshev coefficients or encoded data.
`lp_approx.m`	Chapter 4	Performs Legendre approximation on the given trajectory.
`lp_approx_errcomp.m`	Chapter 4	First performs Legendre approximation and then compresses the approximation errors.
`to16.m`	Chapter 5 & 6	Converts 8-bit data to 16-bit data for SRAM compatibility.
`trajectory_generation.m`	Chapter 5 & 6	Generates command trajectories from the NC code according to the sampling time.
`vsint.m`	Chapter 5 & 6	Performs variable speed interpolation on the given trajectory.

Table C-2 M-file for the ΔY Compression Algorithm

```
%
% This function compresses a given time sequence
% using ΔY compression technique.
% ($ REV 1.4, UY & MD, FEB-2010 $)
%
% Input arguments:
%    q - time sequence (integer)
%    n - order of differences {1,2,3, ...}
%
% Output argument:
%   cdat - compressed data structure with following fields:
%     amp: amplitude (bytes)
%     len: length (bytes)
%     sgn: sign (bytes)
%      ic: initial conditions
%       n: order of difference
%       m: length of original sequence
%
function cdat = dycomp(q,n)
  if (nargin==1), n = 3; end
  m = length(q); q = q(:); y = diff(q,n);
%
% Calculate ICs
%
  ic = zeros(n,1,'int32'); ic(1) = q(1); q = q(1:n+1);
  if(n>1)
    for i = 2:n
      t = diff(q,i-1); ic(i) = t(1);
    end
  end
%
% Sign field
%
```

```
  ns = 8*ceil((m-n)/8); s = [(y>=0); zeros(ns-m+n,1)];
  sf = zeros(ns/8,1,'uint8');
  for k = 1:ns/8
    i = 1 + 8*(k-1); j = i + 7;
    sf(k) = bin2dec(num2str(s(i:j)'));
  end
%
% Amplitude- and length fields
%
  as = []; ts = as; toggle = true; y = abs(y);
  for k = 1:(m-n)
    str = dec2bin(y(k)); L = length(str);
    if (toggle)
      ts = [ts num2str(ones(1,L),'%d')];
    else
      ts = [ts num2str(zeros(1,L),'%d')];
    end
    toggle = not(toggle); as = [as str];
  end
%
% Now, some padding...
%
  na = ceil(length(as)/8); L = 8*na-length(as);
  as = [as num2str(zeros(1,L),'%d')];
  if (toggle)
    ts = [ts num2str(ones(1,L),'%d')];
  else
    ts = [ts num2str(zeros(1,L),'%d')];
  end

  tf = zeros(na,1,'uint8'); af = zeros(na,1,'uint8');
  for k = 1:na
    i = 1 + 8*(k-1); j = i + 7;
    af(k) = bin2dec(as(i:j)); tf(k) = bin2dec(ts(i:j));
  end
  cdat = struct('amp',af,'len',tf,'sgn',sf,'ic',ic,'n',n,'m',m);
end
```

APPENDIX D

NC CODE OF PLASTIC INJECTION MOLD FOR A BOTTLE – CASE STUDY

```
G00 Z-20.01
G00 Y120
G00 X-9.449 Y120.5 T4 (dia 4mm)
G01 Z-20 F200 S150
G18 G03 X9.449 Z-20 I9.449 K0
G01 X9.5 Y120
G01 Y118.5
G18 G02 X-9.5 I-9.5 K0
G01 Y116.5
G18 G03 X9.5 Z-20 I9.5 K0
G01 Y114.5
G18 G02 X-9.5 I-9.5 K0
G01 Y112.5
G18 G03 X9.5 Z-20 I9.5 K0
G01 Y110.5
G18 G02 X-9.5 I-9.5 K0
G01 Y108.5
G18 G03 X9.5 Z-20 I9.5 K0
G01 Y106.5
G18 G02 X-9.5 I-9.5 K0
G01 Y104.5
G18 G03 X9.5 Z-20 I9.5 K0
G01 Y103.5
G01 X7.5 Y102.5
G18 G02 X-7.5 I-7.5 K0
G01 Y100.5
G18 G03 X7.5 Z-20 I7.5 K0
G01 Y98.5
G18 G02 X-7.5 I-7.5 K0
G01 Y96.5
G18 G03 X7.5 Z-20 I7.5 K0
G01 Y94.5
G18 G02 X-7.5 I-7.5 K0
G01 Y92.5
G18 G03 X7.5 Z-20 I7.5 K0
G01 Y90.5
G18 G02 X-7.5 I-7.5 K0
G01 Y88.5
G18 G03 X7.5 Z-20 I7.5 K0
G01 Y86.5
G18 G02 X-7.5 I-7.5 K0
G01 Y84.5
G18 G03 X7.5 Z-20 I7.5 K0
G01 Y82.5
G18 G02 X-7.5 I-7.5 K0
G01 Y80.5
G18 G03 X7.5 Z-20 I7.5 K0
G01 Y78.5
G18 G02 X-7.5 I-7.5 K0
G01 Y76.5
G18 G03 X7.5 Z-20 I7.5 K0
G01 Y74.5
G18 G02 X-7.5 I-7.5 K0
G01 Y73.487
G01 X-12.243 Y72.5
G18 G03 X12.243 Z-20 I12.243 K0
G01 X14.928 Y70.5
G18 G02 X-14.928 I-14.928 K0
G01 X-16.367 Y68.5
G18 G03 X16.367 Z-20 I16.367 K0
G01 X17.165 Y66.5
G18 G02 X-17.165 I-17.165 K0
G01 X-17.487 Y64.5
G18 G03 X17.487 Z-20 I17.487 K0
G01 X17.5 Y64
G01 Y62.5
G18 G02 X-17.5 I-17.5 K0
G01 Y60.5
```

```
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y58.5
G18 G02 X-17.5 I-17.5 K0
G01 Y56.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y54.5
G18 G02 X-17.5 I-17.5 K0
G01 Y52.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y50.5
G18 G02 X-17.5 I-17.5 K0
G01 Y48.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y46.5
G18 G02 X-17.5 I-17.5 K0
G01 Y44.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y42.5
G18 G02 X-17.5 I-17.5 K0
G01 Y40.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y38.5
G18 G02 X-17.5 I-17.5 K0
G01 Y36.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y34.5
G18 G02 X-17.5 I-17.5 K0
G01 Y32.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y30.5
G18 G02 X-17.5 I-17.5 K0
G01 Y28.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y26.5
G18 G02 X-17.5 I-17.5 K0
G01 Y24.5
G18 G03 X17.5 Z-20 I17.5 K0
G01 Y22.5
G18 G02 X-17.5 I-17.5 K0
G01 Y20.5
G18 G03 X17.5 I17.5 K0
G01 Y18.5
G18 G02 X-17.5 I-17.5 K0
G01 Y16.5
G18 G03 X17.5 I17.5 K0
G01 Y14.5
G18 G02 X-17.5 I-17.5 K0
G01 Y12.5
G18 G03 X17.5 I17.5 K0
G01 Y10.5
G18 G02 X-17.5 I-17.5 K0
G01 Y10
G01 X-17.381 Y8.5
G18 G03 X17.381 I17.381 K0
G01 X16.832 Y6.5
G18 G02 X-16.832 I-16.832 K0
G01 X-15.746 Y4.5
G18 G03 X15.746 I15.746 K0
G01 X13.831 Y2.5
G18 G02 X-13.831 I-13.831 K0
G00 Z5
G00 X-17.5 Y10
G01 Z-20
G01 Y64
G18 G02 X-7.5 Y73.487 I9.5 J0
G01 Y102.5
G01 X-8.5
G18 G02 X-9.5 Y103.5 I0 J1
G01 Y120
G18 G02 X-7 Y122.5 I2.5 J0
G01 X0
G01 X7
G18 G02 X9.5 Y120 I0 J-2.5
G01 Y103.5
G18 G02 X8.5 Y102.5 I-1 J0
G01 X7.5
G01 Y73.487
G18 G02 X17.5 Y64 I0.5 J-9.487
G01 Y10
G18 G02 X8 Y0.5 I-9.5 J0
G01 X0
G01 X-8
G18 G02 X-17.5 Y10 I0 J9.5
G00 Z5
M30
```