REAL-TIME MOTION CONTROL USING FIELD PROGRAMMABLE GATE ARRAYS

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

BY

BARIŞ RAGIP MUTLU

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

JUNE 2010

Approval of the thesis:

REAL-TIME MOTION CONTROL USING FIELD PROGRAMMABLE GATE ARRAYS

submitted by **BARIŞ RAGIP MUTLU** 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: 22.06.2010

ABSTRACT

REAL-TIME MOTION CONTROL USING FIELD PROGRAMMABLE GATE ARRAYS

Mutlu, Barış Ragıp
M.Sc., Department of Mechanical Engineering
Supervisor: Assist. Prof. Dr. Melik Dölen
Co-Supervisor: Assist. Prof. Dr. A. Buğra Koku

June 2010, 160 pages

In this thesis, novel implementation methods for FPGA based real-time motion control systems are investigated. These methods are examined for conventional and modern controller topologies as well as peripheral device interfaces which are mutually essential pieces of a motion controller. The developed methods are initially tested one by one to assess the performance of the individual design; and finally an assembled solution is developed to test the overall design. Tests of the overall design are realized via hardware-in-the-loop simulation of a real-world control problem, selected as a CNC machining center. The developed methods are discussed in terms of their success, resource consumptions and attainable sampling rates.

Keywords: FPGA, Motion control

ÖZ

ALAN PROGRAMLANABİLİR KAPI DİZİNİ KULLANILARAK
GERÇEK ZAMANLI HAREKET DENETİMİ

Mutlu, Barış Ragıp
Yüksek Lisans. Makina Mühendisliği Bölümü
Tez Yöneticisi: Yard. Doç. Dr. Melik Dölen
Ortak Tez Yöneticisi: Yard. Doç. Dr. A. Buğra Koku

Haziran 2010, 160 sayfa

Bu çalışmada, alan programlanabilir mantık kapısı dizini (Field programmable gate array - FPGA) tabanlı gerçek zamanlı hareket denetim sistemleri için yeni uygulama yöntemleri araştırılmıştır. Bu yöntemler, bütün bir hareket denetim sisteminin alt parçalarını oluşturan, geleneksel ve modern denetim topolojileri ile çevresel birim arayüzleri için incelenmiştir. Geliştirilen yöntemler başlangıçta tek tek sınanarak bireysel başarımları ölçülmüş, sonunda ise parçalar biraraya getirilerek oluşturulan tasarım sınanmıştır. Bütün tasarımın sınanımları, bir CNC işleme merkezinin çevrimiçi donanım benzetimi kullanılarak gerçekleştirilmiştir. Geliştirilen yöntemler, başarımları, kaynak tüketimler ve erişilebilir örnekleme hızları bazında tartışılmıştır.

Anahtar kelimeler: Alan programlanabilir kapı dizini, Hareket denetimi

To My Parents

DON’T PANIC

ACKNOWLEDGEMENTS

I would like to thank to my supervisor Assist. Prof. Dr. Melik Dölen and co-supervisor Assist. Prof. Dr. A. Buğra Koku for giving me this research opportunity. I have been privileged to have mentors with a great deal of knowledge and I sincerely appreciate their guidance and support during this study.

I would also like to thank to The Scientific & Technological Research Council of Turkey, for financial support under project contract 108E048.

I would like to thank to my parents, Selma and Mehmet Mutlu, for always supporting me with their love and care. It is a blessing to know that regardless of the path I choose to take, they will be there for me.

I would like to thank to my colleagues, Ulaş Yaman, Serdar Üşenmez and Rasim Aşkın Dilan, for all the support and fun they have provided during prolonged hours of study in the laboratory.

I would like to thank to my bıdık Müge, for her love, understanding and patience, especially during the final period of this thesis. I have been lucky to have her by my side.

Finally, I would like to thank to Tuna Soncul, Can Özer, Sinan Değer, Can Özgiresun and all the others who cared enough for me to make this path significantly harder. If it wasn't for them, I would probably have ended up in the same place, but for all the wrong reasons.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ v
ACKNOWLEDGEMENTS vii
TABLE OF CONTENTS viii
LIST OF FIGURES xii
LIST OF TABLES xvi
CHAPTERS
1 INTRODUCTION 1
  1.1 Motivation 1
  1.2 Scope of the thesis 4
  1.3 Organization 5
2 LITERATURE SURVEY 6
  2.1 Hybrid control systems 10
    2.1.1 Control systems utilizing FPGA as an interface 10
    2.1.2 Control systems utilizing FPGA as a secondary processor 14
  2.2 FPGA-based control systems 15
    2.2.1 Control systems using softcore processors 16
    2.2.2 Hardware implementation of control systems 19
  2.3 Closure 23
3 PERIPHERAL DEVICE INTERFACING FOR MOTION CONTROLLERS 25
  3.1 Encoder Interface 26
    3.1.1 Incremental encoder decoding module 28

3.2 Pulse width modulation (PWM) 33
3.2.1 PWM generator module 34
3.2.2 PWM transmitter module 36
3.2.3 PWM receiver module 37
3.3 Finite pulse generation 37
3.3.1 Pulse generator module 38
3.4 Serial Peripheral Interface Bus (SPI) 42
3.4.1 SPI module 43
3.5 Other Interfaces 46
3.5.1 Custom parallel data receive 46
3.5.2 Pulse frequency modulation 47
3.5.3 RS-232 controller 47
3.5.4 SRAM controller 48
3.6 Closure 48
4 STATE-SPACE CONTROLLER AND OBSERVER DESIGN AND IMPLEMENTATION 50
4.1 Full-state feedback controller 51
4.2 Implementation 53
4.2.1 Method I: Matrix multiplier module 54
4.2.1.1 Customized multiplication modules 54
4.2.1.2 Method I-a: Integer multiplication (IMUL module) 57
4.2.1.3 Method I-b: Fixed point multiplication (FMUL module) 58
4.2.1.4 Method I-c : Floating point multiplication (FPU module) 58
4.2.1.5 Overall Architecture 60
4.2.2 Method II: Softcore processor IP module 62
4.2.2.1 Method II-a: Softcore processor with integer arithmetic 63

4.2.2.2 Method II-b: Softcore processor with fixed point arithmetic 64
4.2.2.3 Method II-c: Softcore processor with floating point arithmetic 64
4.3 Test case 65
4.4 Hardware-in-the-loop simulation results 68
4.5 Comparison and discussion 73
4.6 Closure 79
5 IMPLEMENTATION OF DIGITAL FILTERS 80
5.1 Digital Filters 80
5.2 Implementation 82
5.2.1 Method I: Generic filter module 82
5.2.2 Method II: Softcore processor 84
5.3 Hardware-in-the-loop simulation results 85
5.4 Comparison and discussion 89
5.5 Closure 94
6 ADVANCED CONTROLLERS 95
6.1 Controller Topologies 96
6.1.1 Fuzzy Controller 96
6.1.2 Sliding Mode Controller 98
6.2 Test Setup 100
6.3 Results 102
6.4 Closure 105
7 HARDWARE-IN-THE-LOOP SIMULATION & RESULTS 106
7.1 Real system 107
7.1.1 CNC machining center 107
7.1.2 MATLAB/Simulink model 113

7.2 Controller design 116
7.2.1 Controller selection 116
7.2.2 Linearized system model 117
7.2.3 Design via root locus technique 118
7.2.4 Simulink simulation results 123
7.3 Implementation of the system 128
7.3.1 Implementation of the control system on the FPGA 129
7.3.2 Realization of the plant via hardware in the loop simulation. 131
7.4 HILS Results 133
7.5 Closure 143
8 CONCLUSIONS AND FUTURE WORK 144
8.1 Conclusion 144
8.2 Future work 147
REFERENCES 148
APPENDICES
A LIST OF VERILOG HDL FILES 152
B LIST OF C CODES FOR NIOS II IMPLEMENTATION 153
C SAMPLE C CODE FOR NIOS II IMPLEMENTATION 154
D LIST OF MATLAB M-FILES 157
E SAMPLE MATLAB M-FILE FOR HILS 158

LIST OF FIGURES

FIGURES

Figure 1.1 – Schematic of a typical motion controller 2

Figure 1.2 – Increase in number of logic cells of Xilinx and Altera FPGA chips over years 3

Figure 2.1 – Generic schematic for FPGA implementation of a motion controller 8

Figure 3.1 – Ideal output of an incremental encoder................................. 27

Figure 3.2 – Encoder signal generated by the FPGA (6.25 kHz).............. 31

Figure 3.3 – Encoder signal generated by the FPGA (625 kHz)................ 31

Figure 3.4 – Frequency vs. decoding error.. 32

Figure 3.5 – Output of the PWM module with 25% duty cycle............... 35

Figure 3.6 – Output of the PWM transmit module.................................... 36

Figure 3.7 – Sequence of 5 pulses generated at 5 kHz................................ 41

Figure 3.8 – Sequence of 26 pulses generated at 6.25 kHz......................... 41

Figure 3.9 – SPI bus between a master and two slave chips 42

Figure 3.10 – Transmission of an 8-bit data via SPI bus 45

Figure 3.11 – Transmission of an 16-bit data via SPI bus 45

Figure 3.12 – Schematic representations of the developed modules obtained via Quartus II schematic tool 49

Figure 4.1 – Block diagram of the control system...................................... 52

Figure 4.2 – Floating point unit 59

Figure 4.3 – Matrix multiplier module 61

Figure 4.4 – Controller/Observer module 62

Figure 4.5 – Generic model for pendulum drive system 65
Figure 4.6 – HILS result of method I-a 70
Figure 4.7 – HILS result of method II-a.. 70
Figure 4.8 – HILS result of method I-b 71
Figure 4.9 – HILS result of method II-b.. 71
Figure 4.10 – HILS result of method I-c 72
Figure 4.11 – HILS result of method II-c 72
Figure 4.12 – Resource utilization of Method I-a on Cyclone II FPGA.... 74
Figure 4.13– Resource utilization of Method I-b on Cyclone II FPGA 74
Figure 4.14 – Resource utilization of Method I-c on Cyclone II FPGA.... 75
Figure 4.15 – Resource utilization of Methods II-a & II-b on Cyclone II FPGA 75
Figure 4.16 – Resource utilization of Method II-c on Cyclone II FPGA .. 76
Figure 5.1 – Generic filter module.. 83
Figure 5.2 – Cascaded control system 86
Figure 5.3 – HILS result of method I with IMUL....................................... 87
Figure 5.4 – HILS result of method I with FPU.. 88
Figure 5.5 – HILS result of method II with IMUL 88
Figure 5.6 – HILS result of method II with FPU 89
Figure 5.7 – Resource consumption of Method I with IMUL 90
Figure 5.8 – Resource consumption of Method I with FPU 90
Figure 5.9 – Resource consumption of Method II with integer arithmetic 91
Figure 5.10 – Resource consumption of Method II with FPU.................. 91
Figure 6.1 – Fuzzy controller implementation... 97
Figure 6.2 – Membership functions of the fuzzy controller 97
Figure 6.3 – Sliding mode controller implementation 99

Figure 6.4 – Simplified model of a typical turning center 100
Figure 6.5 – Simplified model of the system 101
Figure 6.6 – Test setup 101
Figure 6.7 – Controller performances under disturbance input............ 103
Figure 6.8 – Zoomed controller performances to highlight disturbance rejection 104
Figure 7.1 – First MCV-1100 3-Axis CNC Machining center 108
Figure 7.2 – X-axis feed drive for CNC machining center....................... 108
Figure 7.3 – Z-axis feed drive for CNC machining center....................... 109
Figure 7.4 – Torque capability curve for CNC machining center axis motors 112
Figure 7.5 – MATLAB/Simulink model of a single axis of the CNC machining center 115
Figure 7.6 – Root locus plot of the uncompensated system 119
Figure 7.7 – Root locus plot of the system after addition of a pole and a zero 120
Figure 7.8 – Closed-loop Bode plot of the system 122
Figure 7.9 – Simulink model of the overall system 124
Figure 7.10 – Reference trajectory for the X-axis...................................... 125
Figure 7.11 – Disturbance inputs for light and heavy machining conditions 126
Figure 7.12 – Simulink results of the designed controller in terms of encoder counts 126
Figure 7.13 – Resource utilization of the single axis solution 130
Figure 7.14 – Resource utilization of the implemented 3-axis solution. 131
Figure 7.15 – Schematic of the hardware in the loop simulation system.. 132

Figure 7.16 – Reference trajectories for a plastic bottle injection mold (selected portions indicated with red color) 133
Figure 7.17 – Reference trajectories for X,Y and Z axes (t = 0-20s) 134
Figure 7.18 – Reference trajectories for X,Y and Z axes (t = 201-241s) ... 134
Figure 7.19 – First trajectory motor position error in X-axis (t = 0-20s) . 136
Figure 7.20 – First trajectory motor position error in Y-axis (t = 0-20s) . 136
Figure 7.21 – First trajectory motor position error in Z-axis (t = 0-20s) . 137
Figure 7.22 – Second trajectory motor position error in X-axis (t = 201-241s) 137
Figure 7.23 – Second trajectory motor position error in Y-axis (t = 201-241s) 138
Figure 7.24 – Second trajectory motor position error in Z-axis (t = 201-241s) 138
Figure 7.25 – Second trajectory cart position error in X-axis with backlash (t = 201-241s) 139
Figure 7.26 – Second trajectory cart position error in Y-axis with backlash (t = 201-241s) 139
Figure 7.27 – Second trajectory cart position error in Z-axis with backlash (t = 201-241s) 140

LIST OF TABLES

TABLES

Table 3.1 – Verilog HDL code segment for encoder channel buffering .. 29

Table 3.2 – Verilog HDL code segment for pulse and direction detection 29

Table 3.3 – Verilog HDL code segment for encoder count calculation ... 30

Table 3.4 – Verilog HDL code segment for encoder count calculation ... 35

Table 3.5 – Verilog HDL code segment for *init* rising edge detection 39

Table 3.6 – Verilog HDL code segment for *clock divider* utilization.......... 39

Table 3.7 – Verilog HDL code segment for pulse generation 40

Table 3.8 – Verilog HDL code segment for *SPI* utilization........................ 44

Table 4.1 – Terminology used in FPU.. 60

Table 4.2 – Inverted pendulum parameters... 67

Table 4.3 – Resource/Time Costs of Proposed Methods on Altera Cyclone II FPGA chip.. 77

Table 6.1 – Membership functions of the fuzzy controller........................ 98

Table 6.2 – Resource costs of different controller topologies on the Xilinx Virtex-5 FGPA.. 103

Table 6.3 – Minimum attainable sampling periods of controllers on the Xilinx Virtex-5 FGPA 105

Table 7.1 – MATLAB/Simulink model of a single axis of the CNC machining center 114

Table 7.2 – Controller design parameters for x, y and z axes 123

Table 7.3 – Mean, max and standard deviation values of the Simulink results (in counts) 127
Table 7.4 – Mean, max, and standard deviation values of the Simulink results (in μm) 128
Table 7.5 – Root mean square and standard deviation values of the HILS results for first trajectory (in encoder counts) 141
Table 7.6 – Root mean square and standard deviation values of the HILS results for first trajectory (in μm) 141
Table 7.7 – Root mean square and standard deviation values of the HILS results for second trajectory (in encoder counts) 142
Table 7.8 – Root mean square and standard deviation values of the HILS results for second trajectory including backlash model under heavy machining (in μm) 143
Table A.1 – List of Verilog HDL files employed in the thesis 152
Table B.1 – List of Nios II C files employed in the thesis 153
Table D.1 – List of MATLAB M-files employed in the thesis 157

CHAPTER 1

INTRODUCTION

1.1 Motivation

In general terms, motion control can be described as the position and/or velocity control of a machine part such as the end effector of a robotic arm, cutting tool of a CNC machine, tip of an automated welding machine and etc. Evidently, motion control plays a key role in industrial automation systems and success of the motion control system directly affects the production speed, quality and quantity. Considering that industrial automation systems are used globally in production and assembly lines of many industrial plants including automotive, aviation, packaging, printing, textile and semiconductor production industries, better, faster and more robust motion control systems are always desirable. Schematic of a typical motion control system is shown in Figure 1.1.

Motion control systems are being developed by both academia and private entities, in correlation with the technological advances in semiconductor technology. As new semiconductor devices with faster processing speeds and higher resources are available in the market, new motion controllers are developed by these entities that fully utilize the features of these novel chips.

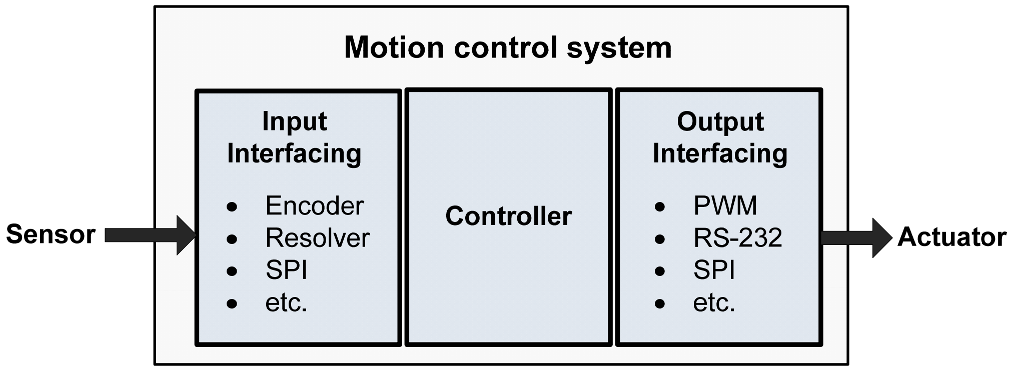


Figure 1.1 – Schematic of a typical motion controller

In the currently available motion control systems, the most commonly employed chip is the digital signal processors (DSPs) which appear in almost all of the recent products developed by both academia and private manufacturers, including the most recent products of the two major motion controller manufacturers, including the DMC-18x6 by Galil and PMAC2 by Delta-Tau. The situation is similar in the academic literature and many studies published in the last decade employ DSPs in motion control systems, especially if the emphasis of the study is on the controller's theory rather than the implementation.

On the other hand, FPGAs are also suitable options for real-time motion control applications, exhibiting some superior qualities over traditional processors (micro-processors, micro-controllers and DSPs) such as parallel processing capability, high sampling rates, flexibility in design, and reliability. Therefore, in the academic end, field programmable gate arrays (FPGAs) also start to take significant parts in motion controller designs. As further discussed in the literature survey chapter, there are many recent studies that employ FPGAs in motion controller designs and obtain very successful results. Especially after the FPGAs' resources have increased significantly in the first half of this decade, there has been a considerable increase in these studies that utilize an FPGA in the motion control system. Figure 1.2 shows the increase in resources (in terms of logic cells/elements) of FPGAs that are produced by two major FPGA manufacturers: Xilinx and Altera in the last 15 years.

As can be seen in Figure 1.2, resources of the FPGAs have increased approximately 700% in the last decade, which resulted in the academia's increased level of interest in FPGAs for motion control applications.

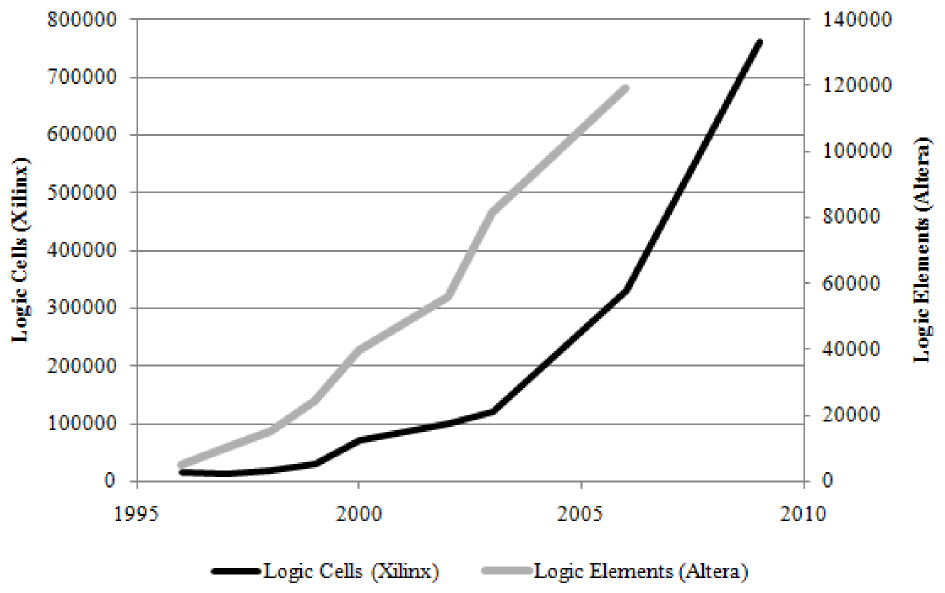


Figure 1.2 – Increase in number of logic cells of Xilinx and Altera FPGA chips over years

It can be estimated that this interest will increase or at least stay at this level, as FPGA manufacturers continue to increase the capabilities of their FPGA chips and develop new and more advanced intellectual properties (softcore processors, SoPC tools, etc.) for embedded designers. Therefore, developing methods for an FPGA based motion control system is desirable, as the FPGAs' role in motion control applications increases.

1.2 Scope of the thesis

As presented in the previous section, the main objective of this study is to develop an FPGA based motion controller and by FPGA based motion controller, a "single chip" solution that handles all the necessary tasks of a typical motion control system is implied. Requirements of a typical motion control system are well-known and the developed system must be able to:

1. Provide a convenient interface for obtaining sensor data from common sensors that are employed in motion control applications and interpret this data as required.
2. Obtain reference inputs from an outside source via a suitable interface and/or a communication protocol or generate the reference commands for the controller itself.
3. Execute a controller algorithm that is able to provide the desired performance that is comparable to the modern motion controllers in the literature.
4. Provide means for transmitting the manipulated input computed by the controller via a suitable interface or a communication protocol to a driver, servo-amplifier or the actuator itself.

Evidently, there are many device/design choices to be made for each item and it is not possible to cover every aspect of FPGA based motion controller design in a single study. However, it is possible to develop methods for the most common choices in the academic studies and industrial applications and if possible extrapolate the results to others. For instance, in Chapter 3, a design method for an incremental encoder interface is provided, leaving out the absolute encoders. However, even though the interface required by an absolute encoder would be different, the obtained result that is hardwired implementation by HDL design is more efficient for the encoder interface is valid for both of the sensor devices. Therefore, most of the results are applicable to similar devices or design choices.

1.3 Organization

This thesis is divided into 8 chapters. The second chapter provides a review on the studies relevant to the motion control applications, in a limited context of FPGA employing designs. In the third chapter, peripheral interfaces that are developed and implemented on the motion controller design are introduced. In the fourth chapter, design and implementation of a state-space controller with a Luenberger-type observer on an FPGA is presented. In the fifth chapter, digital filters are introduced and their implementation methods on FPGAs are discussed. In the sixth chapter, a test case is provided where the developed FPGA modules are tested on a hardware-in-the-loop simulation platform. In the seventh chapter, a preliminary study on advanced controllers is presented. In the last chapter, conclusions are drawn and future work is discussed.

CHAPTER 2

LITERATURE SURVEY

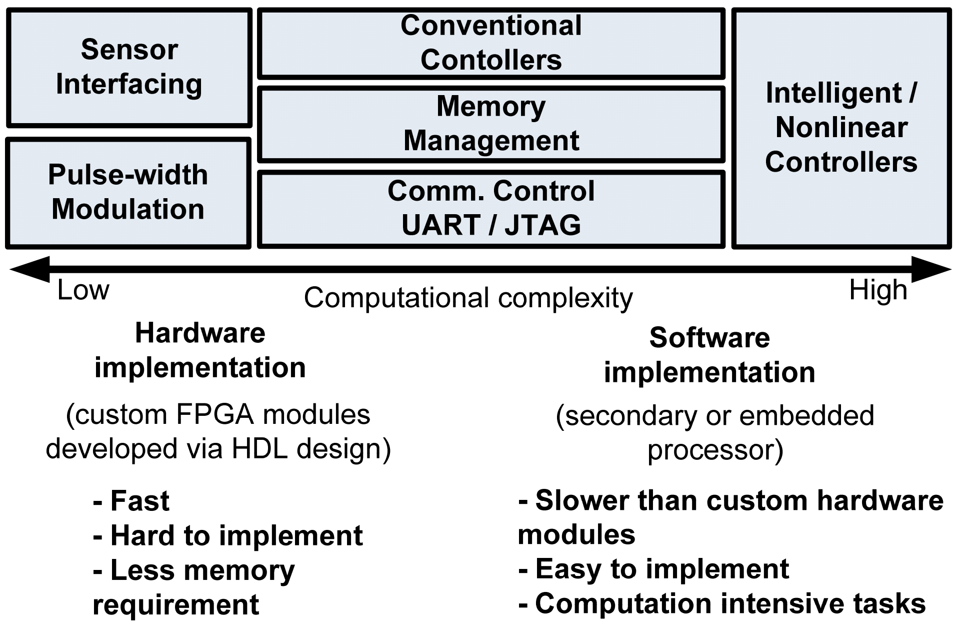
Since the last decade, FPGAs are becoming gradually dominant in control applications. In numerous studies, implementation methods for embedded controller design and application results of different topologies on FPGAs are presented. In this chapter, a general overview of the relevant technical literature is provided.

Due to their flexible hardware, uses of FPGAs in control literature differ significantly among different design topologies. For instance, while there are implementations of an FPGA as an interface between the controller chip and peripheral devices in some designs, there are also architectures in which a complex motion controller is realized by only utilizing a single FPGA chip. Therefore, it is necessary to classify FPGA employing controller designs into two categories as: "Hybrid control systems" and "FPGA-based control systems".

In hybrid control systems, FPGA may be employed in two different schemes. In the first scheme, FPGA is used for peripheral device interfacing and it has minimal amount of computational tasks. On the other hand, in the second scheme, FPGA is utilized as a secondary processor, and a noteworthy computational load is handled by the FPGA, in addition to its interfacing tasks. Nevertheless, it is obvious that in these hybrid designs there needs to be a central processor employed in the design, which is usually selected as a digital signal processor (DSP) or a microprocessor.

In a generic scheme of a hybrid design, peripheral device interfacing (sensor interfacing, pulse-width modulation generation, SPI and etc.) is a task assigned to the FPGA. This is due to the fact that FPGA implementation of those modules is fast, relatively simple and requires a small amount of resources. On the other hand, complex controller algorithms such as intelligent (fuzzy, neural-network) or nonlinear (sliding mode, variable structure and etc.) controllers are generally implemented on the processor, as well as the memory management units (SRAM, EEPROM and etc.) and communication controllers (UART, JTAG and etc.). It can be seen that, as the complexity of the task increases and the speed requirement decreases, the load shifts from the hardwired FPGA implementation to the processor, as shown in Figure 2.1.

As can be seen from Figure 2.1, the conventional controllers find their place in between this speed/complexity distinction, which means that there are designs that both include or exclude a conventional controller unit on the FPGA and both options are equally feasible. Hence, these hybrid control systems are also classified according to the existence of a controller load on the FPGA as it is further discussed in Section 2.1.



Sensor Interfacing
Pulse-width Modulation
Conventional Contollers
Memory Management
Comm. Control UART / JTAG
Intelligent / Nonlinear Controllers
Low
Computational complexity
High
Hardware implementation
(custom FPGA modules developed via HDL design)
- Fast
- Hard to implement
- Less memory requirement
Software implementation
(secondary or embedded processor)
- Slower than custom hardware modules
- Easy to implement
- Computation intensive tasks

Figure 2.1 – Generic schematic for FPGA implementation of a motion controller

As a final remark for hybrid designs, the presented scheme is not a universal approach and while it is a common approach to use FPGA for interfacing and implementation of simpler controllers; complex controller algorithms, memory management units and communication controllers are also applicable by FPGA implementation and in some designs there are topologies that do not correspond to the generic scheme provided in Figure 2.1.

In the FPGA-based designs, there exist a single FPGA chip that is responsible for the main computational task as well as the interfacing, memory management and communication functions. However though, a single chip implementation does not mean that every task is handled by a custom hardwired FPGA module. Since there are many embedded processor IPs developed by FPGA manufacturers, it is possible to employ an embedded processor in an FPGA based design. If an embedded processor is employed on an FPGA-based design, it can be seen that the embedded processor implementation along with the custom hardware modules is very similar to the hybrid design approach. The difference would be that the connecting wires between the custom modules and the processor are external on the hybrid design and processor's resources don't consume FPGA's resources. Therefore, the generic scheme provided in Figure 2.1 applies with a slight difference on the employed processor.

Nevertheless, it is also possible that the overall motion control system is implemented on the FPGA chip with custom hardwired modules. It is a significantly different approach than the processor-based designs; therefore it is convenient to classify the FPGA-based designs into two by the utilization of an embedded processor.

In FPGAs, there are different ways that computations can be handled. In hardwired implementation via HDL design, it is possible to include or develop custom arithmetic module IPs for fixed-point arithmetic. Furthermore, it is also feasible to insert a floating point unit (FPU) in the design to perform floating point calculations, such as the one devised by R. Usselmann [1] [1] as an open-core IP. Nevertheless, it should be noted that the development platform of hardware description languages such as Verilog HDL or VHDL does not support fixed/floating point number representations (and accompanying calculations) directly and therefore the design paradigm is long and tedious. On the other hand, in the embedded processor development environment, it is possible to use different number types conveniently by coding through a high-level language like C.

A similar problem occurs when implementing a memory management unit or a communication controller on the FPGA as a hardwired module. In this hardware solution, it is the designer's job to arrange and synchronize the clock signals, read/clear buffers etc. On the other hand, if a softcore processor is employed, it provides a system on programmable chip (SoPC) solution for these tasks and it is significantly easier to implement an SRAM controller or a UART controller when an embedded processor is employed. As a matter of fact, the designer does not interact at all with the memory management unit or the UART controller, after including those units to the SoPC design interface and connecting them to the selected embedded processor.

2.1 Hybrid control systems

In this section, designs are discussed in which FPGA is either employed as a sole interface between a main processor and peripheral devices or utilized as a secondary processor in addition to its interfacing task.

2.1.1 Control systems utilizing FPGA as an interface

In many previous studies, FPGAs are generally considered as an interface between a processor unit that handles the controller (such as a DSP or a microprocessor) and its peripheral units. This is due to the fact that FPGA implementations of peripheral interfaces are particularly fast and efficient by design with hardware description languages (such as Verilog HDL and VHDL) on FPGAs. For instance, a PWM generator is a very common and necessary interface in motion controller designs and in a study conducted by Arbit et al. [2] it has been shown that different shapes of waveforms of PWM signals can be generated precisely on an Altera Spartan II FPGA and even simpler FPGAs could handle the task quite efficiently. Therefore, implementing an FPGA based input/output interface is a good option for motion controllers, as further discussed in Chapter 3. In this section, designs in which FPGA is employed as a peripheral device interface are presented.

In such configurations, FPGAs do not share the computational burden of the main processor. For instance, in the design proposed by Dong et al. [3], the control system consists of a TMS320LF2407 DSP of Texas Instruments accompanied by an EP1K30-144 FPGA of Altera. While this system is presented as a dual-core system, there is a significant difference between the computational loads of the cores. In this design, FPGA core deals with the external circuitry (that are FPGA modules) to process position encoder signals, some keyboard inputs and displays while also managing some I/O functions as well as a data bus to the DSP core. On the other hand, the DSP core handles all the computations of the PID controller. Therefore, although the system is presented as a dual core system, the primary computation load is on the DSP core.

A similar scheme appears in the study of Birou and Imecs [4] where the FPGA is coupled to a TMS32OC50 DSP of Texas Instruments. In this configuration, an FPGA module generates pulse width modulation

(PWM) signals for the power converters and another FPGA module decodes incremental encoder signals. Controller is again implemented on the digital signal processor (DSP) and FPGA has no computational load.

In another study conducted by Lee et al [5], a DSP 2812 controller board employing a TMS320F2812A DSP is utilized with a Altera Cyclone FPGA chip in order to realize a RBF neural network controller for nonlinear systems. Analogous to the previous studies, the scheme is topologically the same as the previous studies and the FPGA is used for the sole purpose of encoder interfacing/counting and PWM generation.

Another recently published study by Caporal and Pacas [6], presents implementation of a direct mean torque controller that is realized on a hybrid system consisting of an ADSP 21062 DSP by Analog Devices and a Xilinx XC4010E FPGA. Nevertheless, FPGAs duty for interfacing remains with no significant computational load.

On the other hand, the situation is analogous in industrial motion controllers developed by private manufacturers. For instance, "DMC-18x6" which is the most recent and highest performance motion controller of Galil [7], utilizes a Xilinx FPGA chip for interpreting the encoder signals, while PID compensation with velocity and acceleration feedforward is realized by a 32-bit Motorola processor. Likewise, another product "PMAC2" which is developed by Delta-Tau [8], employs an Altera FPGA chip as an encoder interface and the central processor is selected as a Freescale DSP563xx.

These recent studies show that even the capabilities of the FPGAs are significantly increased in the last two decades; they are still employed for simply interfacing purposes regardless to their increasing capacity and processing capabilities. Nevertheless, it is also important to note that even if FPGAs do not carry heavy computational loads in the mentioned designs, their role is crucial in demanding control applications where sensor signals need to be decoded in a fast and efficient manner and complex output signals need to be generated. Therefore in these examples, the fact that FPGAs do not have a computational burden does not imply that their role in the controller design is unimportant.

For instance, in the study conducted by Al-Ayasrah et al. [9], an N-Motor speed controller for brushless DC motors is proposed by the implementation of a ADSP-21992 DSP by Analog Devices that is employed with a Xilinx Virtex-E FPGA chip. The key feature of this design is the N-motor speed control, which is realizable by the PWM module implemented on the FPGA chip.

Another study conducted by Toh et al. [10] presents implementation of torque and flux controllers for direct torque control (DTC) of an induction machine. In this study, complex torque/flux estimations are computed and a P flux controller along with a PI torque controller are realized on a DS1102 DSP board from dSPACE, which is based on a TMS320C31 DSP. However the success of the control algorithm depends on the high speed generation of the output signals and therefore it is vital that a faster hardware circuit handles that operation, which is realized on an Altera EPF10K20 FPGA device.

2.1.2 Control systems utilizing FPGA as a secondary processor

As evidently shown by the presented studies in the previous section, FPGAs are widely used for interfacing purposes in controller designs coupled with a digital signal processor, having none or very limited computation loads. However, there are also other hybrid control systems in which FPGAs share more amount of computational load than just an interfacing chip; deserving its role as a secondary processor.

In the design proposed by Jung and Kim [11], Texas Instrument's TMS320C6711 DSP is employed with Altera's EP20K300EQC240 FPGA chip, similar to the schemes in previous section. The controller topology is composed of a neural network controller and a PID controller; however in this case the PID controller is implemented on the FPGA; therefore contrary to the previous designs, the computation load of the FPGA is increased.

A study conducted by Yu et al. [12] proposes a novel multi-redundancy electro-mechanical actuator (EMA) controller, implemented on a TMS320VC33 DSP by Texas Instruments and Cyclone II FPGA by Altera. In this design, DSPs functions include receiving instructions of position, and carrying out calculations of the position loop and velocity loop; whereas the FPGAs functions are to provide interface for position signals, velocity signals and current signals, to carry out the calculation of the current loop, and to generate the PWM signal that is required by the multi-redundancy EMA. As can be seen, in this configuration FPGA handles the current controller as well as its typical interfacing tasks, earning its title as a secondary processor.

It is obvious that in these hybrid designs, FPGAs computation load as a secondary processor vary significantly from design to design. For instance, in a novel AC servo system implementation proposed by Esmaeli et al. [13], a Spartan FPGA of the Xilinx is employed with a TMS230LF2407 DSP of Texas Instruments. In the system, DSP chip implemented a position loop control which includes a PI regulator, velocity and acceleration feedforward and a digital notch filter. On the other hand, FPGA chip implemented the speed loop and current loop control, which also includes a PI regulator as well as a vector transformation module. This design can be considered as a truly hybrid design, because of the fact that the computational loads of both chips are equivalent and demanding.

As demonstrated by the provided examples, FPGAs can be utilized in many different controller topologies along with another processor, to achieve tasks at a range of complexities from relatively simple to utterly complex. However, there are also designs that employ a single FPGA chip for the overall motion controller design, and they are presented in the following section.

2.2 FPGA-based control systems

Recent research studies tend to shift the computational load from the primary processor to the FPGA, as the capabilities of FPGAs significantly increased in time over the last two decades. Many examples of hybrid control systems are provided in the previous section, however in those studies the FPGA has always been accompanied by a primary processor, which has mostly been a digital signal processor or a microprocessor.

Nevertheless, it is also possible to realize a control system on a single FPGA chip, and in this section, designs in which FPGA is solely employed to realize the entire motion control system are presented.

2.2.1 Control systems using softcore processors

FPGAs require low-level (i.e. logic-level) circuit design through hardware description languages (HDL) and such a design effort using HDLs is a relatively long and tedious process. To overcome this difficulty, FPGA manufacturers offer many intellectual properties (IPs) including embedded processor IPs. Therefore, many motion controller designs that facilitate a single FPGA makes good use of an embedded processor IPs.

While some designs use softcore processors only to implement controllers, the others utilize the embedded processor for complex calculations along with some “hardwired” custom modules for simpler tasks. As a matter of fact, utilizing an embedded processor does not necessarily mean that the whole implementation is realized on the processor. Therefore, it should be noted that while the control systems presented in this section are classified as systems with embedded processors, most of the designs in this section also utilize custom hardwired modules developed by HDL design tools. However, modules for complex arithmetic operations are not generally included since the processor easily handles those operations.

In a study conducted by Ni et al [14], an integrated motion control system is realized on an Altera Cyclone FPGA with a Nios II processor. This hardware design scheme realizes all sensor data acquisition, SPWM generation, motor vector control, torque control and trajectory generation in a single FPGA chip. Since a Nios II processor is employed, all the calculations are performed with floating point arithmetic.

In another study, Li et al [15] developed an FPGA-based servo controller for PMSM drives on an Altera Cyclone EP1C20 FPGA. In this design, while a custom HDL designed module performs the current/speed loop control for PMSM drives, which includes vector control strategy, the PI regulator, coordinate transformation and the SVPWM generator; Nios II processor performs the function of position control, based on the discrete-time sliding mode variable structure control. Similar to the previous study, the softcore processor allows floating point arithmetic for the variable structure controller.

In a recent study, Kung et al [16] realized a motion control IC for an X-Y table on a Altera Stratix II EP2S60F672C5ES FPGA chip. In this implementation, two axes' fuzzy position controllers and P speed controllers, as well as a motion trajectory generator are implemented by software using Nios II softcore processor. On the other hand, two axes' current vector controllers, along with the SVPWM generators and the encoder interfaces are realized by custom hardware (HDL design) modules. It is noted that the Nios II processor has less than 1 kHz sampling frequency, while the custom hardwired implementation works at 16 kHz. Therefore this study is a good example for demonstrating the better speed characteristic of the custom hardware.

Das and Banerjee [17] developed a digital PID controller for precision control of a brushed DC servo motor, employing a Spartan 3E FPGA chip and utilizing a PicoBlaze softcore processor by Xilinx. As expected, the PID control algorithm is realized on the softcore processor and the supporting interface modules such as: RS232 controller and the PWM generator is developed as custom hardware modules by VHDL. It is also noted that since the PWM generation is a time critical job, it cannot be handled by the processor along with the PID controller an therefore a custom module is developed for PWM generation task.

Salem et al. [18] proposed a servo drive system in which the peripheral interface and speed control is handled by hardware and position control along with the networking functions are handled by softcore processor, and implemented it on a Xilinx Virtex-II Pro XC2VP30 FPGA chip available on a Xilinx ML310 board. Furthermore, two different RT kernels, that are μC/OS-II and Xilkernel are investigated on the PowerPC 405 processor and their performances are tested on a test case of a PI controlled DC motor emulator.

These examples demonstrate that using an embedded processor IP within the FPGA is proven to be a successful approach. Furthermore, it is also shown that using an embedded processor with custom hardware modules further increases the efficiency and speed of the design, and it is proven that oftentimes the softcore processor needs to be accompanied by custom hardware modules. Unsurprisingly, this remark raises the issue of the performance of an FPGA-based design that is fully hardwired HDL design. The following section presents studies that do not employ an embedded processor and the whole motion controller design is based on custom hardware modules.

2.2.2 Hardware implementation of control systems

While the current tendency is to use an embedded processor for complex calculations required by controllers, it is also possible to eliminate the embedded processor and develop a total hardware solution. This is a desirable feature since the hardware modules are significantly faster than the embedded processors and oftentimes more efficient in terms of resources and power consumption rates. However, it is a long task to develop custom hardwired modules and furthermore, in order to benefit from the efficiency of the flexible hardware, modules need to be optimized for a specific task to fully observe the improvement obtained by the hardware implementation.

It is important to note that controller implementation on FPGA is often a trade-off between resource and execution time of the controllers. The reason for that is; while it is possible to benefit from the parallel processing capability of the FPGA chip by calling many instances of a certain module which would certainly require more resources, it is also possible to use certain modules repeatedly in a sequential manner to increase the time, rather than the resource cost. Therefore, it is possible to modify a design according to the resource/time requirements of the controller.

In a very recent study, Cho et al [19] have proposed an FPGA-based multiple axis motion control chip with no embedded processor employed. The chip has all the essential features such as velocity profile generation, interpolation, inverse kinematics calculation and a PID controller which are required to control a multiple axis motion control system such as a robotic manipulator. As discussed earlier, certain methods need to be developed in order to avoid complex calculations and in this study, they managed to avoid floating point calculations by multiplying coefficients by constant integers. Using no embedded processor; they attained lower resource costs and power consumption rates. The hardware modules are developed by VHDL and implemented on a Xilinx XC2V6000 FPGA.

Chan et al [20] have conducted a study on PID controller implementation on an FPGA and they managed to decrease the resources required by a multiplier-based design significantly on the target platform that is a Xilinx Spartan-II-E FPGA. They have proposed to replace the multipliers by a distributed arithmetic based design utilizing look up tables and they managed to decrease the resource requirement down to 4 to 13% of the former design. However they increased the computation time from 1 cycle to 13 to 26 cycles. The increase in efficiency of this design refers to the resource cost of the controller in terms of slices and power consumption. A very similar study by Tao et al [21] managed to decrease the logic element requirement of a PID-based CNC position controller from 51.7% to 0.8-1.5% in an Altera Cyclone II FPGA, increasing the computation cycle from 1 to 32-64 cycles, utilizing two different methods. The methods provided in these studies are a mere result of trying to reduce the resource requirements of the conventional PID controller and while these studies offer good improvements for a PID controller, there exist many controller algorithms including intelligent, nonlinear and hybrid topologies and there is no single way to implement each of them more efficiently on an FPGA. However, they are valuable to demonstrate that without the embedded processor IPs, it is still possible to implement a conventional controller as hardwired modules successfully via the HDL design tools on an FPGA.

There are also some other studies where more complex controllers are implemented on the FPGA via custom hardwired modules. Fuzzy controller designed by Lanping et al [22] requires no embedded processor or a floating point unit to perform fuzzy control. The selected platform for development is EPF10KlOLC84-15 of FLEXlOK series from Altera. On the other hand, the proposed method is similar to a rule based control topology and the method is not generally applicable for different fuzzy controller topologies. In another study by Kung et al. [23], an adaptive fuzzy controller for AC motor drive is proposed and the speed control IP is fully realized on the hardware platform of an Altera Cyclone EP1C20 FPGA. As a matter of fact, a Nios II processor is also included in this design for SRAM and UART control, however since the controller is fully implemented as hardwired, it is included as a hardware implementation. In this design, the current loop sampling frequency is 16 kHz, while the speed control loop's sampling frequency is 2 kHz.

An Elman neural network implementation is proposed by Lin et al [24] for a linear ultrasonic motor, where a fixed point arithmetic unit is implemented to perform the calculations. The proposed design is implemented on a Xilinx XC2V1000 FPGA chip and 723 Hz sampling frequency is attained for the controller. In the previous section, a study by Kung et al. [16] is provided for realization of a motion control IC for a X-Y table, in a more recent study by the same group [25], a self-tuning PID controller is realized using RBF neural network and is applied to the X-Y table. Different from the previous study, in this case the Nios II processor is only employed for trajectory generation purposes and the rest of the design is implemented as hardwired custom modules, including the neural network. The same chip that is Altera Stratix II EP2S60, is employed in the design and current loop can be closed at 16 kHz while the position loop's sampling frequency is 500 Hz. From these examples, it can be noticed, as the complexity of the controller increases, the attainable sampling frequency decreases.

These studies prove that it is possible to implement different controller topologies on FPGAs utilizing arithmetic logic units or custom hardware solutions. It is also seen that PID controllers are easier to implement by custom hardware logic and in practice usually realized by hardware modules. On the other hand it is also possible to implement more complex topologies by custom hardwired modules.

2.3 Closure

In this chapter, a literature review was provided for motion controller topologies that employ an FPGA chip. A classification was made according to the utilization of the FPGA in the design as a secondary chip or a single chip; and a further classification was performed according to the FPGA's duty in the design for the secondary chip case and utilization of an embedded processor in the FPGA for the single chip case. This classification is non trivial and different categorizations are also possible; nevertheless the studies in this review are provided according to this classification, in order to better demonstrate the differences between the design approaches.

In the provided studies, it has been shown that FPGAs can be successfully used in tasks within a wide complexity range, from primitive interfacing tasks to complex controller algorithms. It is shown that FPGAs can perform tasks utilizing custom hardware IPs and/or embedded processors. However, when an FPGA is accompanied by a secondary processor chip, it is seen that FPGA is mostly utilized by custom hardwired modules.

As a general rule, in almost all of the studies the peripheral interfacing has been implemented by hardwired FPGA modules, however, for implementation of control algorithms, it is shown that both custom FPGA modules and embedded processors are viable options. For the case of memory management and communication control, embedded processor IPs are generally selected for their simplicity in design; however it is shown that, custom modules are also available for these tasks.

In the presented studies, it was shown that the custom modules developed by HDL design tools have less processing time and resource requirement than the embedded processors. This is due to the fact that, custom modules are developed and optimized according to the controllers' needs. On the other hand, while some parameters and configuration of the embedded processor can be modified before implementation, it is still much less flexible than a custom hardwired module. In this study, an FPGA-based solution is proposed and evaluated.

CHAPTER 3

PERIPHERAL DEVICE INTERFACING FOR MOTION CONTROLLERS

A motion controller is always required to receive and transmit signals and data from/to other devices employed within the control system; therefore it is necessary for a motion controller design to include some means for peripheral device interfacing. For instance, motion controllers need to receive some sort of sensor information from a mechanical/optical/magnetic sensor and produce a corresponding manipulation signal that can be interpreted by a driver or can drive the actuator itself. Furthermore, other peripheral units (such as a host PC for setting the design parameters of the controller online) may also be included in the control system, which would require a communication protocol (such as RS-232) or a bus interface (such as SPI) in the controller architecture.

Evidently, there exist various interface topologies employed in sensors, actuators and other devices; and it is not practical to include numerous interfacing modules in a single design. Therefore, it is necessary to limit these modules by considering the most common interfaces used in common practice and if possible taking the currently employed devices into account.

In this chapter, a number of interfacing modules are presented, which are selected conveniently from the commonly used devices in motion control applications.

All the modules are developed using a hardware description language (mostly Verilog HDL), in order to increase the processing rate and decrease the resource cost of the module, benefiting from the flexible hardware of an FPGA.

On the other hand, it should also be noted that while implementing a softcore processor is also a viable option, for primitive designs such as sensor/actuator interfacing or communication protocol controller modules, it is a more efficient design approach to develop customized modules via HDL design. Further discussion on softcore processor implementation on FPGAs is available in the following chapters.

3.1 Encoder Interface

In motion control applications, the most commonly employed sensors to obtain positional feedback are encoders. Therefore an encoder decoding module is the most critical sensor interface of a motion control system. Encoders are electromechanical devices that generate an analog or digital (commonly digital) signal, in order to provide linear or rotational position feedback. Encoders that provide absolute position feedback are called "absolute position encoders" and their output signals require no interpretation; since they provide absolute position information in proportion to their resolution. However, due to their limited resolution, absolute encoders do not succeed in high speeds and in industrial control applications, incremental encoders are generally preferred.

Incremental encoders provide digital signals as square wave forms from two channels, with a phase difference of 90°. Therefore, it is possible to count every rising/falling edge of channels A and B (which is called quadrature decoding method), and increase the resolution of the encoder to 4 times its initial resolution. Ideal output of an incremental encoder is shown in Figure 3.1.

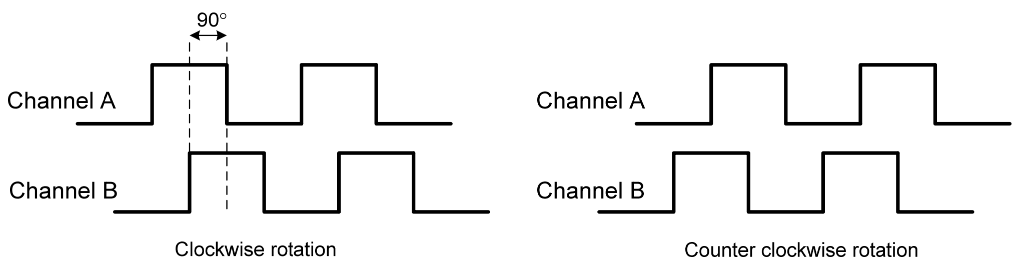


Figure 3.1 – Ideal output of an incremental encoder

The main advantage of incremental encoders is that the resolution is not correlated with the number of output channels, which is 2 for any resolution, unlike absolute encoders, which need to have 2^n output ports to provide n-bit resolution. Therefore, if an incremental encoder is employed, no hardware modification is required in the motion controller interface if a different encoder with higher (or lower) resolution is implemented.

On the other hand, the output of incremental encoders needs to be interpreted by the decoder chip, by means of counting the rising/falling edges of the output signals A and B of the encoder. It is obvious that the decoding process needs to be realized at a high sampling frequency since missing a logic level change of a signal would mean misinterpretation of the feedback.

FPGA is a suitable choice for high frequency decoding purposes, because of its logic level design capability, as stated in the introduction of this chapter. A perfectly optimized module is developed for the sole purpose of encoder decoding; in order to utilize the available clock source with full capacity (i.e. no computational delays). The developed encoder module is presented in the next section.

3.1.1 Incremental encoder decoding module

Incremental encoder decoding module, or as commonly referred in this text as the "encoder module" is developed using Verilog HDL and its symbolic representation as generated by Quartus II (Altera's programmable logic device design software that is used throughout this chapter) is presented in Figure 3.12.

As can be seen from Figure 3.12, the module takes two channels from the incremental encoder as its input, as well as a clock and reset signal. The output of the module is simply the result of the current pulse count, obtained via quadrature decoding. Note that *clk* signal is the input clock of the module and a sufficiently fast clock is necessary for successful decoding. In this implementation, the fastest clock on the *Altera DE1* board (50 MHz) is connected to this signal. For the implementation, first the inputs from channels A and B are buffered in a 3-bit register in order not to miss any logic level changes in the signal. Verilog HDL code of this implementation is as shown in Table 3.1. Note that signal buffering is realized with this approach throughout this design.

Table 3.1 – Verilog HDL code segment for encoder channel buffering

```
reg[2:0] inA_delayed, inB_delayed;
always @(posedge clk) inA_delayed <=
{inA_delayed[1:0], inA};
always @(posedge clk) inB_delayed <=
{inB_delayed[1:0], inB};
```

As can be observed, at the system clock frequency, input channels A and B are sampled and buffered into registers called inA_delayed and inB_delayed. Using these signals that are delayed 2 clock cycles (corresponding to 40 ns for the 50 MHz clock), it is possible to detect a pulse generated by the encoder, as well as determine its direction. Verilog HDL code of this implementation is shown in Table 3.2.

Table 3.2 – Verilog HDL code segment for pulse and direction detection

```
wire count_enable = inA_delayed[1] ^
inA_delayed[2] ^ inB_delayed[1] ^
inB_delayed[2];
wire count_direction = inA_delayed[1] ^
inB_delayed[2];
```

As can be seen, 3 XOR gates are implemented to detect the level change of the pulse and 1 XOR gate is implemented to determine the direction. After detecting a pulse from an encoder channel and determining its direction, it is trivial that a count register needs to be increased/decreased according to its input, as shown in Table 3.3.

Table 3.3 – Verilog HDL code segment for encoder count calculation

```
always @(posedge clk)
begin
  if(count_enable)
  begin
    if(count_direction) count<=count+1;
    else count<=count-1;
  end
end
```

At this point, comparing this design with a softcore processor design for the same purpose would be helpful to demonstrate the suitability of the customized module approach. Even if minimum specifications are selected for the processor, it would still require a significant processing time, where the design could also be implemented with a simple and efficient HDL code. Note that the success of the module depends on a high sampling clock and a simple design, which are fulfilled by this approach.

The generated module is tested by two means. In the first experiment, an incremental encoder generating 2000 pulses/rev (which corresponds to 8000 pulses/rev with quadrature decoding) is employed and the encoder shaft is driven manually for a certain period of time. In the end, when the encoder shaft is returned to its original position, a multiple of 8000 is read on the PC via RS-232 connection. After this relatively simple experiment, a more complex and convincing experiment is performed.

In the next experiment, an FPGA module to generate an encoder-like signal is developed to test the decoding module. This module is capable of generating encoder signals up to 6.25 MHz, which is limited by the fastest clock available (50 MHz) on the FPGA board. The output of this module is connected to the general purpose I/O ports of the FPGA board; which can be treated as output signals of a real encoder. Therefore, using this module, encoder signals in a wide range of frequencies could be generated for testing purpose. Two signals generated by this module with 6.25 kHz and 625 kHz frequencies are shown in Figure 3.2 and Figure 3.3.

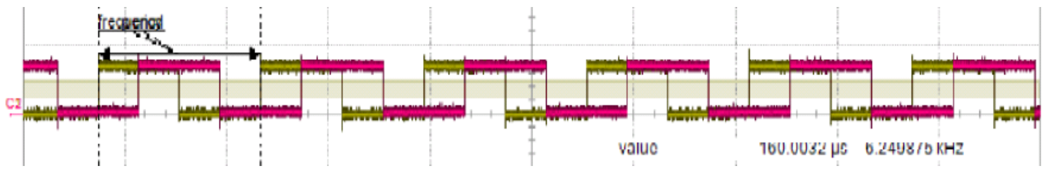


Figure 3.2 – Encoder signal generated by the FPGA (6.25 kHz)

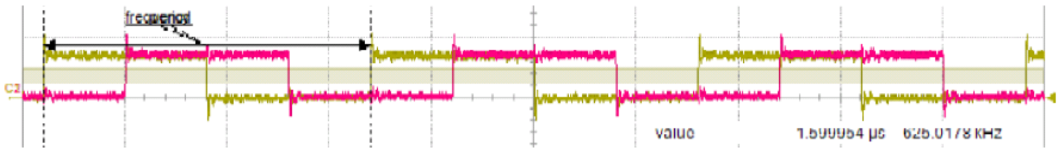


Figure 3.3 – Encoder signal generated by the FPGA (625 kHz)

As seen in Figure 3.2 and Figure 3.3, the signals successfully represent the ideal encoder signal, as shown in Figure 3.1. Using this module, an experiment is performed with different frequencies ranging from 200 Hz to 1 MHz, and the direction of rotation is changed numerously (between 5 to 20 times) via the on-off switches available on the FPGA board. The result of this experiment is shown in Figure 3.4.

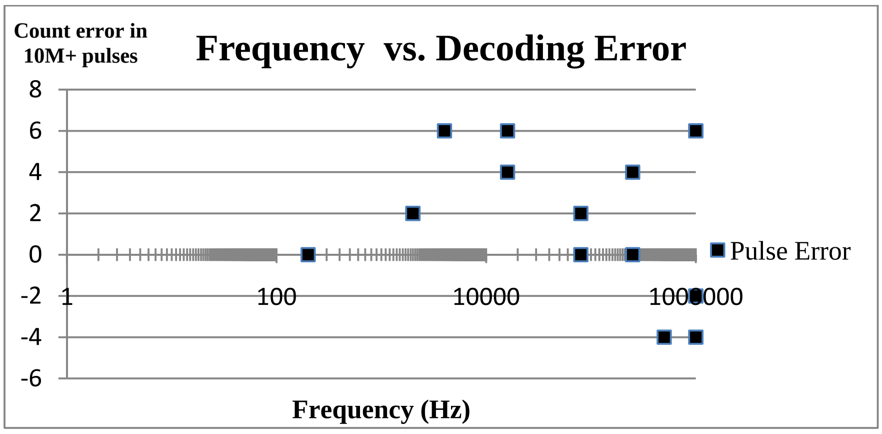


Figure 3.4 – Frequency vs. decoding error

As shown in Figure 3.4, in frequencies greater than 1 kHz (1000 rev/s), count error is equal to only a few pulses when a sum more than 10 Million pulses are generated by the encoder module. Those frequencies are already extremely high for industrial applications and it can be deduced that FPGA implementation of an incremental rotary decoder is sufficiently successful for industrial motion control applications.

3.2 Pulse width modulation (PWM)

Pulse width modulation (PWM) is a technique to provide intermediate electrical power by changing the duty cycle of a high frequency digital signal. Therefore, it is a commonly employed method to represent an equivalent analog signal with a digital signal. While it is possible to use PWM signal to directly drive a motor (via a servoamplifier), it is also possible to feed the PWM signal to a motor driver where PWM represents a manipulated input command (such as torque). Furthermore, PWM can also be used for serial communication, in a 2-wire scheme similar to SPI.

Essentially, frequency of the PWM signal is crucial and there are two factors affecting the attainable frequency of the PWM signal: frequency of the input clock and the desired resolution. The relationship between the system clock, resolution and the attainable PWM frequency is obtained as $f_{pwm} = f_{clk} / 2^R$ where f_{clk} and R correspond to the system clock and the resolution respectively. A calculation based on the fastest available clock on *Altera DE1* board (50 MHz) and a 10-bit resolution (which is a highly sufficient value for industrial applications) can be performed as 50MHz / 2^{10} = 48.8 kHz. This equation shows that an equivalent analog signal can be represented with 10-bit accuracy at 48.8 kHz, when a 50 MHz clock is available.

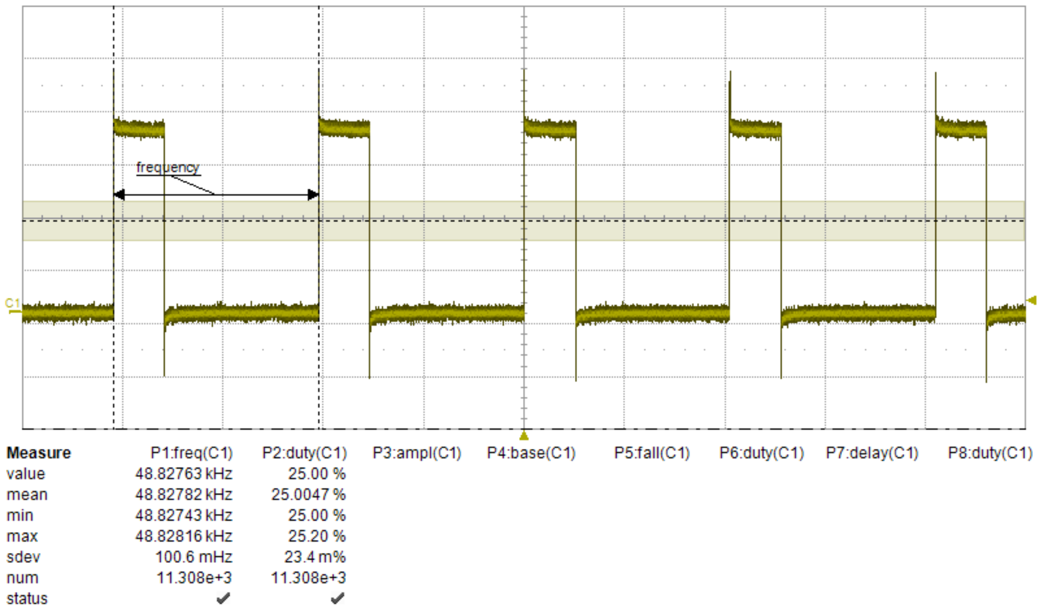
As discussed earlier, PWM can be employed in different schemes. It can be used directly to drive a servo-motor via a servo-amplifier: its output can represent a torque command for another controller or it can be used to transmit serial data to another chip. However, the design differs between the two cases when PWM signal represents an analog signal and is used to transmit data. Therefore, two modules called PWM generator (for producing an equivalent analog signal) and PWM transmitter (for serial communication) are developed and included in the motion controller design.

3.2.1 PWM generator module

PWM generator module (or commonly referred in this text as PWM module) is a simple module that can create a PWM signal that represents its input data. PWM module is developed using Verilog HDL and its symbolic representation as generated by Quartus II is presented in Figure 3.12. As can be seen in Figure 3.12, the module takes a 10-bit `duty_in` signal as its input, as well as a clock and enable signal. The output of the module is simply the generated PWM signal called `pwm_out`. Verilog HDL code of this implementation is shown in Table 3.4.As can be observed, as long as the `clkctr` is smaller than the `duty_in` signal, the output of the PWM is equal to 1. As can be seen, PWM module is simple; yet efficient in creating equivalent analog signals. Output of the PWM module is shown in Figure 3.5.

Table 3.4 – Verilog HDL code segment for encoder count calculation

```
always @(posedge clk)
begin
   clkctr <= clkctr+1;
   if(enable)
   begin
      if(clkctr < duty_in) pwm_out <= 1;
      else pwm_out <= 0;
   end
   else pwm_out <= 0;
end
endmodule
```



Measure	P1:freq(C1)	P2:duty(C1)	P3:ampl(C1)	P4:base(C1)	P5:fall(C1)	P6:duty(C1)	P7:delay(C1)	P8:duty(C1)
value	48.82763 kHz	25.00 %						
mean	48.82782 kHz	25.0047 %						
min	48.82743 kHz	25.00 %						
max	48.82816 kHz	25.20 %						
sdev	100.6 mHz	23.4 m%						
num	11.308e+3	11.308e+3						
status	✓	✓						

Figure 3.5 – Output of the PWM module with 25% duty cycle

Notice that the frequency value is 48.8 kHz, as calculated. Duty cycle is set as 25%, which can be adjusted with a 10-bit resolution.

3.2.2 PWM transmitter module

PWM transmitter module creates a PWM signal; however this signal is not continuous as a regular PWM signal and is generated only when a data needs to be transmitted. PWM transmitter module is developed using Verilog HDL and its symbolic representation as generated by *Quartus II* is presented in Figure 3.12.

As shown in Figure 3.12, PWM transmitter module has an extra input and output different from the PWM module. This is due to the fact that, since the PWM transmit module is used for serial communication, an initiate transmission signal `init_xmit` is necessary for the transmission start. Furthermore, a similar signal is required as the output, in order to inform the receiver module that the transmission has started. Output of this module is shown in Figure 3.6.

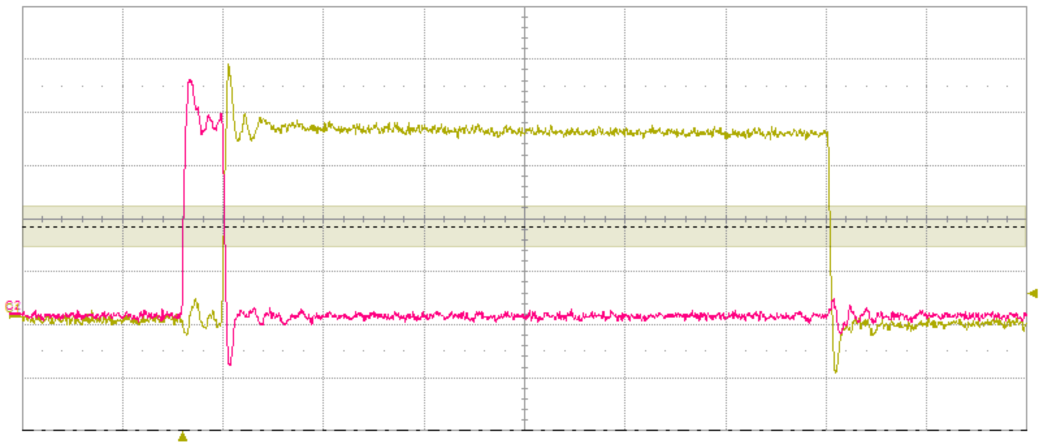


Figure 3.6 – Output of the PWM transmit module

In Figure 3.6, pink line represents the `xmit_clk` and the yellow line represents the `PWM_data_out`. As can be observed, falling edge of the `xmit_clk` initiates the transmission and `PWM_data_out` is set to 1 with the same method employed in the PWM module. Besides the `xmit_clk`, another difference of this module is the frequency of this signal, which is **not** equal to 48.8 kHz, since the module is initiated only when data transmission is necessary. In a controller topology, frequency of this signal would be equal to the sampling frequency of the controller.

3.2.3 PWM receiver module

PWM receiver module interprets the PWM signal created by the PWM transmitter module. PWM receiver module is developed using Verilog HDL and its symbolic representation as generated by *Quartus II* is presented in Figure 3.12.

As can be seen in Figure 3.12, input of the module matches the output of the transmitter module and outputs of the module are the interpreted data that is `PWM_recv_data` and `PWM_OK` signal which generates a rising edge when data transmission is complete.

3.3 Finite pulse generation

In motion control applications, oftentimes the controller needs to drive multiple axes that have different control requirements. For instance, if a 3-axes turning center is considered, position control of the cutting tool requires a much simpler control approach than the speed control of the spindle motor. Therefore, in cases where a simple controller is sufficient, it may be desirable to drive that axe via the "pulse control mode" of the driver.

In order to drive a controller in "pulse control mode", it is necessary to produce a number of square pulses at a certain frequency that corresponds to the speed of the motor shaft. Evidently, this frequency is limited by the specifications of the driver/motor couple.

3.3.1 Pulse generator module

Pulse generator module gets a number of pulses and frequency as its input and produces the specified number of pulses at the desired frequency. Pulse generator module is developed using Verilog HDL and its symbolic representation as generated by *Quartus II* is presented in Figure 3.12. As can be seen in Figure 3.12, finite pulse generation has three inputs (other than the system clock) called `init, pulse_no` and `pulse_freq_div`. `Init` signal represents the initialize signal for the generator and rising edge of this signal enables this module. Rising edge of this input is detected by buffering the signal in a 3-bit register and checking the consecutive bits, as shown in Table 3.5. Note that this Verilog HDL code shown in Table 3.5 is commonly employed for rising/falling edge detection of enable/initialize signals throughout this design.

Table 3.5 – Verilog HDL code segment for *init* rising edge detection

```
reg[7:0] init_delayed;

always @(posedge clk) init_delayed <=
{init_delayed[6:0], init};

wire init_risingedge =
(init_delayed[2:1]==2'b01);
```

`Pulse_freq_div` is employed to set the frequency of the generated finite pulse signal by utilizing another module called the *clock divider* module as shown in Figure 3.12. This clock divider module is utilized in various parts of the overall design, in order to generate clocks (infinite square waveforms) at different frequencies. As can be seen, its inputs are the system clock and a divider and its output is a clock with frequency f_{clk_out} that is equal to f_{clk_sys} / divider. This clock divider module is utilized in the pulse generator module as shown in Table 3.6.

Table 3.6 – Verilog HDL code segment for *clock divider* utilization

```
clk_divider clk_div1(.divider((pulse_freq_div >>
1)),
                           .sys_clk(clk),
                           .clk_out(clk_pulse));
```

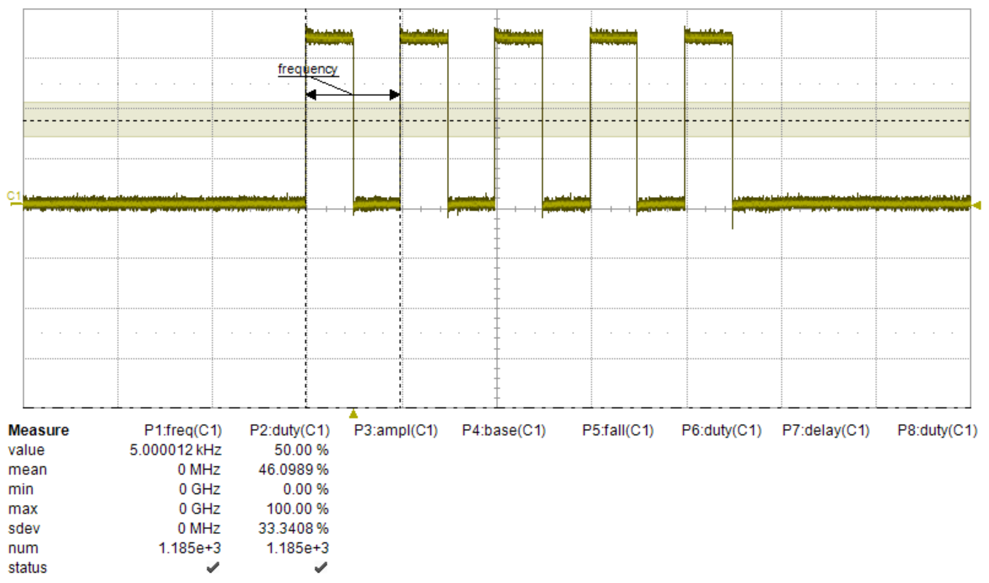
As can be seen in Table 3.6, `pulse_freq_div` signal is directly connected to the input of the divider module, with a bitshift to right, which would double the output frequency of the divider. The reason for this shift is revealed in the pulse generator segment of this code, shown in Table 3.7.

Table 3.7 – Verilog HDL code segment for pulse generation

```
always @(posedge clk)
begin

   if(init_risingedge)
   begin
   PULSE_on <= 1;
   clkctr <= 0;
   end

   if(PULSE_on && clk_pulse_risingedge)
   begin
   clkctr <= clkctr + 1;

   if(clk_pulse_risingedge) pulse_out <= (1-
pulse_out);

   end

   if((clkctr >> 1) == pulse_no)
   begin
   PULSE_on <= 0;
   pulse_out <= 0;
   end

end
```

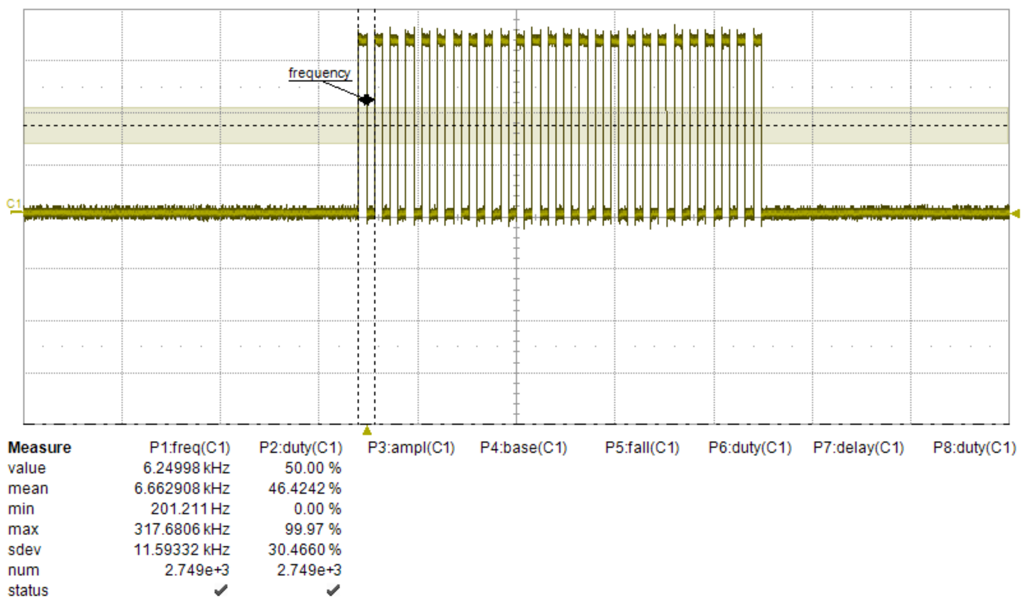
As can be seen in Table 3.7, a register called `pulse_on` is set to 1, when a rising edge of the initialize signal is detected. As long as this registers logical value is true and the clock pulse generated by the clock divider module has a rising edge (which has an analogous code to the one provided in Table 3.5) `pulse_out` signal changes its logical value. Since this process halves the frequency, the output of the clock divider module needs to be twice the desired output frequency, which explains the reason why `pulse_freq_div` signal is shifted 1 bit to right.

Using this module, different numbers of pulse sequences are generated at different frequencies. Two of these sequences are provided in Figure 3.7 and Figure 3.8.



```
Measure   P1:freq(C1)   P2:duty(C1)  P3:ampl(C1)  P4:base(C1)  P5:fall(C1)  P6:duty(C1)  P7:delay(C1)  P8:duty(C1)
value     5.000012 kHz      50.00 %
mean             0 MHz   46.0989 %
min              0 GHz      0.00 %
max              0 GHz    100.00 %
sdev             0 MHz   33.3408 %
num          1.185e+3    1.185e+3
status              ✓           ✓
```

Figure 3.7 – Sequence of 5 pulses generated at 5 kHz



```
Measure   P1:freq(C1)   P2:duty(C1)  P3:ampl(C1)  P4:base(C1)  P5:fall(C1)  P6:duty(C1)  P7:delay(C1)  P8:duty(C1)
value      6.24998 kHz     50.00 %
mean      6.662908 kHz   46.4242 %
min        201.211 Hz       0.00 %
max       317.6806 kHz     99.97 %
sdev      11.59332 kHz   30.4660 %
num           2.749e+3    2.749e+3
status               ✓           ✓
```

Figure 3.8 – Sequence of 26 pulses generated at 6.25 kHz

3.4 Serial Peripheral Interface Bus (SPI)

Serial Peripheral Interface Bus (*SPI*) is a serial communication protocol that is commonly used for data transmission between chips. SPI bus operates in full duplex mode and communication is achieved in master/slave mode where the master chip initiates the data transmission by selecting the target chip via the slave select signal (*SSEL*), followed by the serial clock signal (*SCLK*). During operation, data is exchanged from master out slave in (*MOSI*) and master in slave out (*MISO*) ports in full duplex mode. While the most common word size for transmission is 8-bits, other sizes are also commonly implemented. A simple schematic diagram of the SPI bus between a master and two slave chips is presented in Figure 3.9.

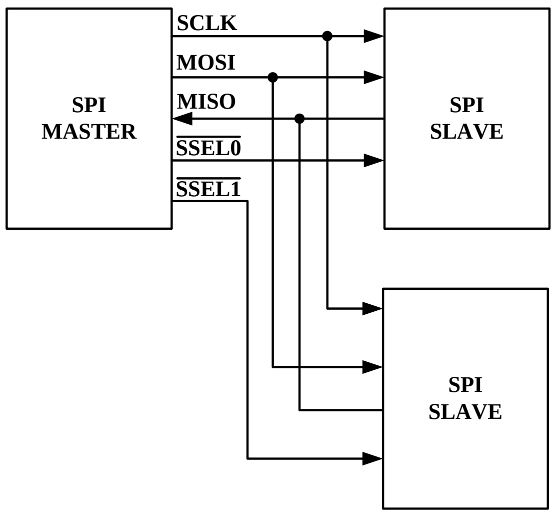


Figure 3.9 – SPI bus between a master and two slave chips

While there are different serial communication protocols such as I^{2}C and One-Wire that can also be implemented on an FPGA; in terms of data transmission speed and multiple chip support, SPI is a better choice for development. For instance, in terms of transmission rates, the closest rival of SPI is I^2C with transmission speeds of 100-400 kbit/s, whereas SPI can achieve 5-10 Mbit/s speed. Therefore, an SPI module is selected for development for the motion controller design and the details are presented in the next section.

3.4.1 SPI module

SPI module is developed using Verilog HDL and its symbolic representation as generated by Quartus II is presented in Figure 3.12. As can be seen in Figure 3.12, SPI module is configured for operating in slave mode; however it is also possible to change the operation mode from slave to master with a few modifications in the HDL code. Note that `clk` signal is the input system clock of the module and not relevant with the SPI bus. Verilog HDL code segment realizing the data transmission is provided in Table 3.8.

As seen in Table 3.8, SPI module is working only when *SSEL* signals logical value is 0. When a falling edge of the *SCK* signal is detected (analogous to the method explained in Table 3.5), ***`bitcnt`*** register is increased by 1 and ***`byte_data_received`*** is shifted left with the new bit coming from *MOSI*. On the other hand, at the rising edge of the *SCK* signal, *MISO* register is shifted 1 bit to left, in order to set the new bit for transmission.

Table 3.8 – Verilog HDL code segment for *SPI* utilization

```
always @(posedge clk)
begin
  if(SSEL)
    bitcnt <= 0;
  else
  begin
  if(SCK_fallingedge)
  begin
    bitcnt <= bitcnt + 1;
    byte_data_received <=
{byte_data_received[6:0], MOSI};
  end
  if(SCK_risingedge)
    MISO << 1;
  end
end
```

The developed SPI module is implemented on the FPGA board as the slave chip, and the communication is tested via an 8-bit microcontroller (PIC16F877A), operating as the master chip. Two instances obtained during operation of the SPI bus via an oscilloscope are presented in Figure 3.10 and Figure 3.11. In Figure 3.10 and Figure 3.11, D0 represents the data signal (*MOSI*), D1 represents the serial clock signal (*SCLK*) and D2 represents the slave select signal (*SSEL*). As can be observed, data transmission starts with falling edge of the *SSEL* signal, and then *SCLK* produces 8 pulses (since the word size for this application of SPI is selected as 8-bits). During this process, logic level of the *MOSI* signal changes according to the data that is to be transmitted at each rising edge of the *SCLK*; and read by the slave chip at each falling edge of the *SCLK*. As can be seen in Figure 3.11, before the next byte is transmitted, initial states of the *SCK* signal and the *SSEL* signal are identical while *MOSI* is different, since it represents the data.

Transmission tests are performed between the FPGA board and PIC 16F877A microcontroller with this module and the module is proven to be working successfully at speeds up to 250kbit/s, which is a highly sufficient value for the motion controller design problem.

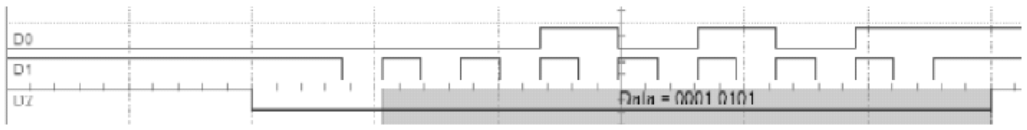


Figure 3.10 – Transmission of an 8-bit data via SPI bus

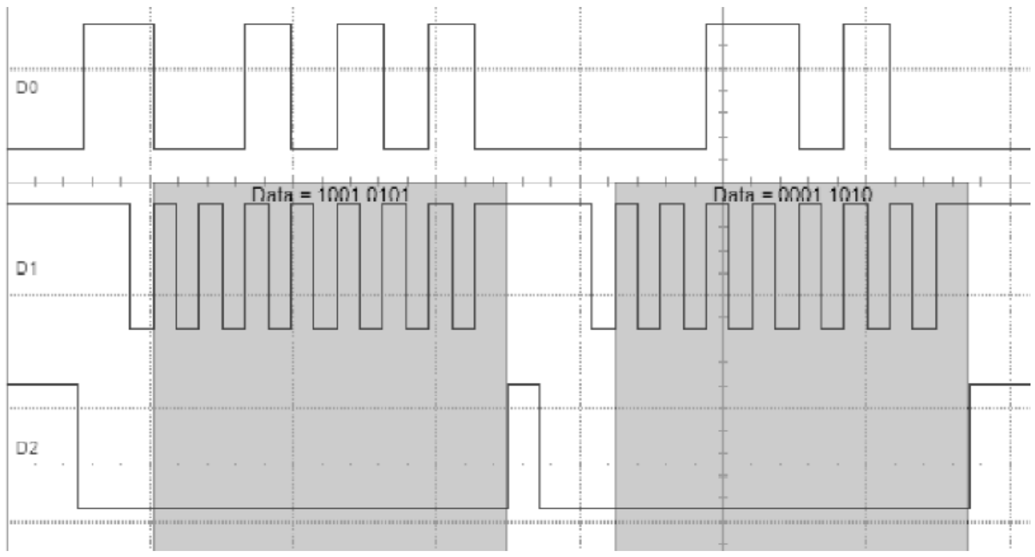


Figure 3.11 – Transmission of an 16-bit data via SPI bus

3.5 Other Interfaces

Other than the aforementioned interfaces and modules, there are a few more modules that are implemented in the design. These interfaces include a ***custom parallel data receive*** module, a custom ***pulse frequency modulation*** module as well as an ***RS-232 controller***. The reason why these modules are not discussed in detail is that, even if these modules are employed in some parts of the design, their applications are custom and their design methodology is very similar to the explained cases; and therefore further discussion would not provide significant information. On the other hand, while RS-232 controller is a relatively important design, it is adapted from an open core source as an intellectual property and therefore, is not comprehensively discussed.

3.5.1 Custom parallel data receive

This interface is developed for fast and simple data transmission between two FPGA chips. The transmission takes place in parallel with a 32-bit width and a clock is generated by the receiver to change the input data at the transmitter end. Symbolic presentation of this receive block is shown in Figure 3.12. As can be seen, *`cmnd_in`* signal is the 32-bit parallel input signal. Every time the *`clk_out`* produces a rising edge, the transmitter block sets the *`cmnd_in`* signal and therefore the new data can be read in parallel. This receive block buffers 9 set of 32-bit data and transmits this to the main module with a *`CMND_recv_ok`* signal indicating that the transmission is over. As can be seen, this is a custom module and is not applicable to industrial peripheral devices without modification in the HDL design.

3.5.2 Pulse frequency modulation

Pulse frequency modulation (PFM) method is a very similar method to pulse width frequency, however in this case the duty cycle of the signal is constant where the frequency is varying. The PFM module is used for transmitting data just as the PWM transmitter and receiver modules that are explained in sections 3.2.2 and 3.2.3. Symbolic presentations are not provided since they are analogous to PWM modules and Verilog HDL codes are not discussed since a more complex design is explained in the finite pulse generation module.

3.5.3 RS-232 controller

RS-232 controller is an important module, and is excessively used in the following chapters for realizing a hardware-in-the-loop simulation on a PC. While it is possible to develop a controller, it is a long and tedious design effort, especially when open core modules are available for utilization. Therefore, an open core IP is utilized in the design and is used throughout this study. The utilized module is "*Simple Asynchronous Serial Comm. Device*" that is developed by *R.Usselmann [1]* and is available at *www.opencores.org.* A symbolic presentation of the "Simple Asynchronous Serial Comm. Device (SASC)" module is presented in Figure 3.12. It should also be noted that this top level module utilizes three other modules that are also developed by the same author.

3.5.4 SRAM controller

SRAM controller is used for utilizing the SRAM available on the FPGA board. An open core IP that is also developed by *R.Usselmann [1]* is utilized in the design and is used throughout this study. A symbolic presentation of the "*SRAM controller*" module is presented in Figure 3.12.

3.6 Closure

In this section, a hardwired design methodology is presented for peripheral device interfaces which are used frequently in motion controller designs. The designs are developed via Verilog HDL, however it should be noted that any other hardware description language (such as VHDL) is evenly applicable for these designs. A number of these presented modules are also employed in Chapter 7 of this thesis.

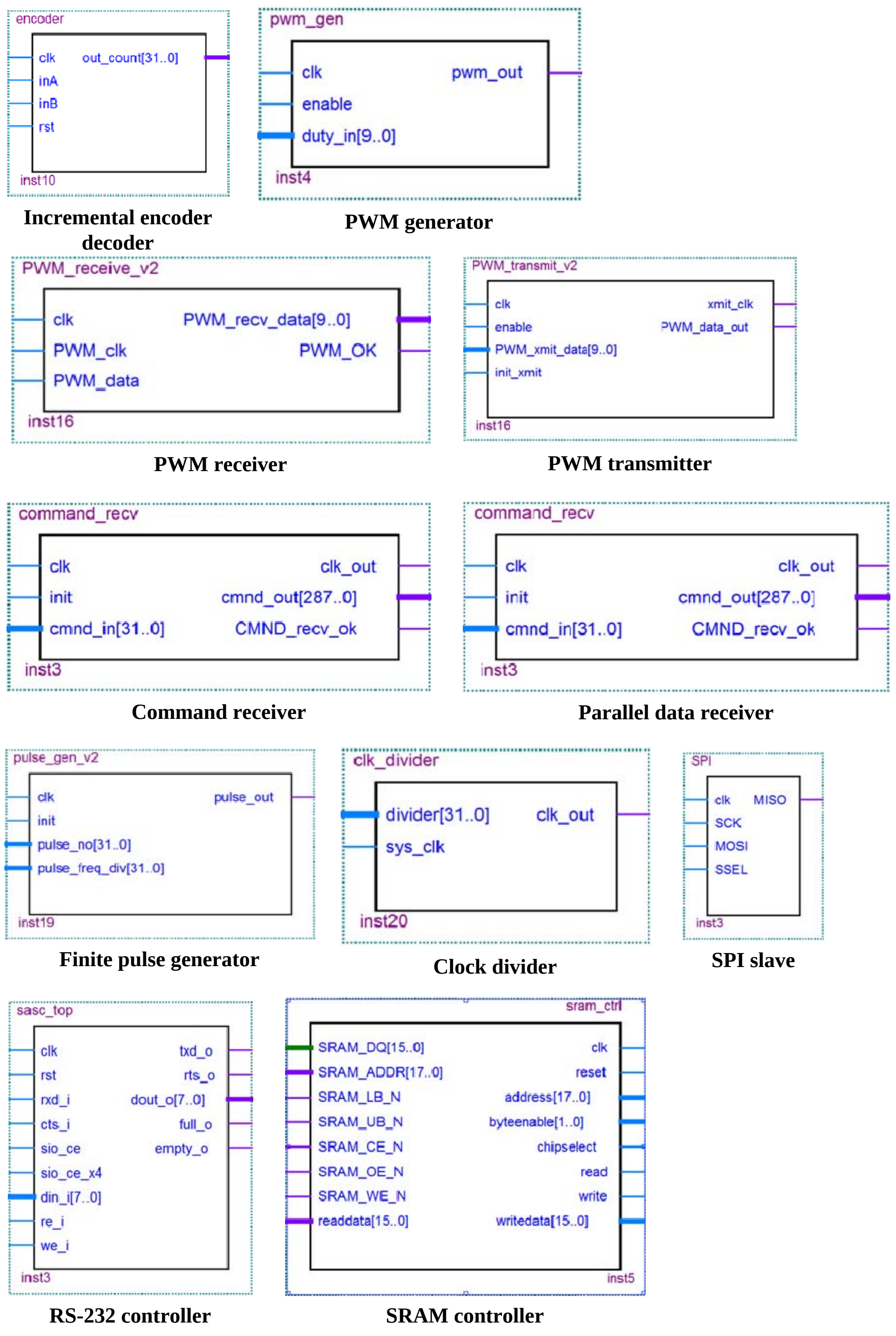


Figure 3.12 – Schematic representations of the developed modules obtained via Quartus II schematic tool

CHAPTER 4

STATE-SPACE CONTROLLER AND OBSERVER DESIGN AND IMPLEMENTATION

State-space controllers are convenient choices for multi-output systems, since they provide means for controlling multiple states of the plant using time-domain based design methodology. However, they exhibit a challenge since all the available states are not generally available in a control system. The general scheme of a motion control system is that the angular/linear position feedbacks are obtained via encoders but the time derivatives of these states are unavailable from sensor feedback; since providing a second feedback (velocity/acceleration) is generally costly or unfeasible. Therefore the controller is usually employed as coupled with a state observer, in order to estimate these unavailable states that are required by the controller.

The overall implementation of this controller and observer scheme can be realized by employing a series of matrix multiplications (as will be discussed in the following section); however the initial problem remains as the realization of the basic multiplication of two elements of these matrices and how these elements could be defined with different data types to reduce the computation load from the multiplication unit.

In this chapter, after a brief introduction of the controller/observer scheme that is to be implemented, two diverse approaches are presented for performing these calculations required by a full-state feedback controller and a Luenberger-type state observer. However, as it will be revealed later, applications of these methods are not limited to state-space controllers and can be used for realization of other algorithms. For instance, in Chapter 5, digital filter implementation on FPGAs is thoroughly discussed and the same methods presented here are applied for the digital filter design problem.

4.1 Full-state feedback controller

State space controllers find their use in almost all industrial motion control applications. The control law of a typical state-space controller can be simply expressed as;

$$\boldsymbol{u} = -\mathbf{K}\big(\hat{\mathbf{x}}(k) - \mathbf{x_r}(k)\big) \tag{4.1}$$

Here $\mathbf{u}$ is the manipulated input vector; $\hat{\mathbf{x}}$ is the estimated state vector; $\mathbf{x}_r$ is the reference state vector and $\mathbf{K}$ refers to the gain matrix. Once the system equations governing the dynamics of the plant are obtained, the gains in (4.1) can be adjusted to yield desired control characteristics using modern control theory. Note that since all states of the plant are not measured for all practical purposes, a full-state observer needs to accompany the design in order to estimate the missing components of the state vector. For instance, a Luenberger-type state observer takes the following form:

$$\hat{\mathbf{x}}(k+1) = (\mathbf{F} - \mathbf{LH})\hat{\mathbf{x}}(k) + \mathbf{G}\boldsymbol{u}(k) + \mathbf{L}\boldsymbol{y}(k) \tag{4.2}$$

Here **F**, **G**, **H** correspond to the system matrices formed by the estimated parameters; y(k) is the controlled output vector while **L** denotes the gain matrix of the observer. Figure 4.1 shows the block diagram of the state-space control system with the Luenberger-type state observer.

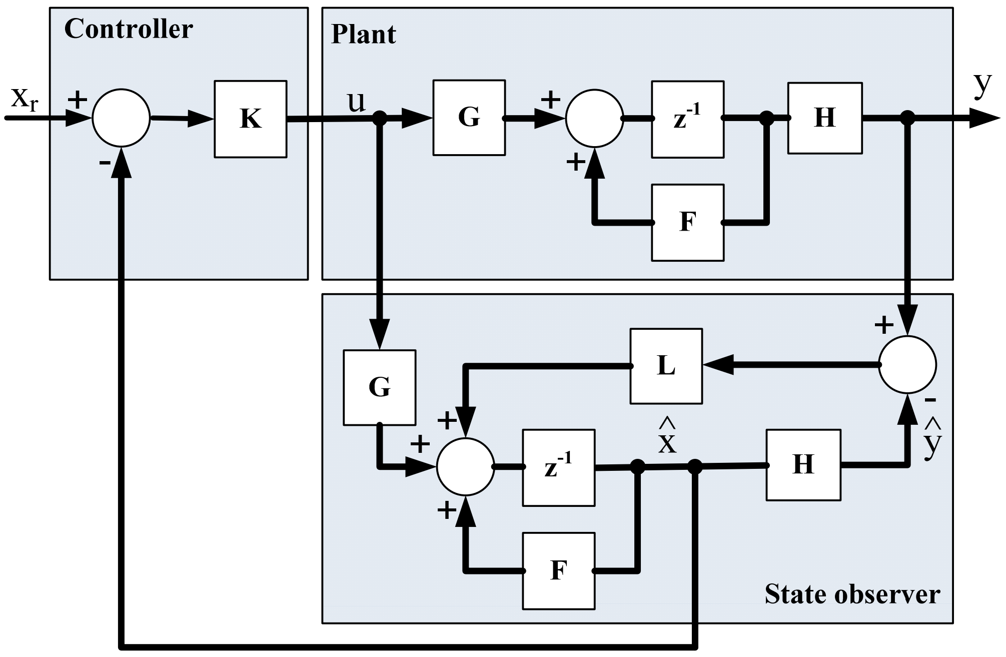


Figure 4.1 – Block diagram of the control system

In industrial motion control, the states of the system are frequently selected as angular position (θ) and (instantaneous) angular velocity (ω). Thus, the general approach is to estimate the velocity using the measured position. There exist versatile estimator algorithms in the literature for velocity estimation in digital systems employing encoders/resolvers.

However, no single estimation algorithm exists to cover all applications where dynamic operating conditions do vary considerably such as the one considered in this study. Hence, when good estimates on system parameters are available, it is more desirable to observe the unavailable states, rather than to estimate them using higher-order differencing methods. Implementation details of this controller/observer scheme are presented in the proceeding sections.

4.2 Implementation

In Chapter 2, many recent studies employing an FPGA in controller designs are presented and different architectures are discussed where FPGAs are used coupled with a DSP/MCU or used exclusively as the controller. Furthermore, for the FPGA based (non-hybrid) designs, various alternatives are discussed to handle the computations required by the controller algorithm; including embedded softcore processors and hardwired solutions (including embedded multipliers, floating point units, custom arithmetic modules and etc.)

Unlike embedded softcore processors, hardwired solutions require relatively long and tedious design effort with hardware description languages (HDLs) and therefore many studies in recent literature tend to include an embedded processor in their FPGA based designs. However, using an embedded processor may have disadvantages such as consuming a bulk amount of the resources of the chip and decreasing the flexibility and processing rate of the module. Therefore, in order to fully benefit from the flexible architecture of an FPGA, it is necessary to develop custom FPGA modules for computation.

In this section, two different approaches are presented for realization of the controller/observer scheme that is presented in the preceding section. The first approach uses a *matrix multiplier module*, which is developed by HDL design (with Verilog HDL) and incorporates other custom multiplication modules called *IMUL*, *FMUL* and *FPU*. Evidently, custom multiplier modules are the key features of this method and are discussed thoroughly. The second method is a softcore processor solution which utilizes the *Nios II* softcore processor developed by *Altera* and the design approach is drastically different from the modular approach, since a processor is involved in the design.

4.2.1 Method I: Matrix multiplier module

Matrix multiplier module utilizes a custom multiplication module (which can be selected as *IMUL*, *FMUL* or *FPU*) and an adder, in order to realize a matrix multiplication. However, before proceeding with the details of this module, it is necessary to present the custom multiplication modules and explain the methodology behind these modules as well as how these modules are developed and customized for a motion control application.

4.2.1.1 Customized multiplication modules

Digital control systems have unique properties that relax the usage of floating point arithmetic and thus there is a potential to develop inexpensive yet high-performance solutions. In order to explore the capabilities of a flexible hardware, all the IPs included in the design should be customized (and optimized) for a specific task at hand. Consequently, it is necessary to adapt low level open source IPs within the design or develop custom modules via HDL design methods for a specific controller topology (i.e. a state-space controller) combined with a specific control system (i.e. a motion control system with a motor drive and encoder feedback).

Some of the advantageous attributes of digital motion control systems are as follows:

- Outputs of all sensors used in controls technology are essentially amplitude-quantized and can be conveniently represented as (signed/unsigned) integers.
- Reference signals (i.e. the command vector), which are to be compatible with the sensory data, are to be generated as integer (number) sequences.
- Manipulated outputs of almost all control systems need to be amplitude-quantized while sending them out to the output interface.
- With proper scaling, the controller gains might be cast as integers without a significant change in the overall dynamics of the controlled system dynamics.

As explained in Chapter 3, in many motion control applications, incremental optical encoders (either linear or rotational) are exclusively employed to measure position of which is commonly characterized as integer counts of pulses being produced by these devices. Similarly, the manipulated output (torque or velocity command to the motor driver) is represented as a finite-length binary number to be latched onto a digital-to-analog converter. Note that within the context of this study, such systems will be referred to as **"quantized input/output"** control systems. Furthermore, if the control gains could be also cast as integers, the resulting system will be called **"fully quantized system."**

For a fully quantized system, a state-space controller can be a suitable choice since the pole placement techniques offer a margin for manipulation on the controller gains. Note that the casting of these gains as integers is not a straightforward task as the input arguments of the gain matrix must be pre-scaled which may in turn aggravate the quantization noise. Hence, the overall problem requires a fine balance among conflicting objectives.

It is critical to notice that if a system is fully quantized, one may employ integer-arithmetic entirely in all calculations. On the other hand, for a quantized input/output system, the decimal multiplication algorithms of digital signal processing (such as shift-and-add algorithm) can be utilized. These well-known algorithms are easy to implement on FPGAs with the low-level design tools provided by FPGA manufacturers. Since many industrial motion control applications employ the quantized input-output, such methods can reduce resource costs significantly.

The implementation methods for the state-space controller/observer is to be developed for a quantized input-output (hence including fully quantized) system. In this system, representing the input, output, and feedback states as floating-point numbers do not have an advantage in terms of accuracy. Therefore, it is appropriate to cast and store these quantities as signed/unsigned integers. Note that the selection of the word size is up to the designer and is to be chosen by considering both the system properties (such as feedback resolution) as well as the features of the implementation method. The methods elaborated in the following section make good use of the special properties for control systems.

Two techniques are proposed for matrix calculations which are essential in state-space controllers and observers. Using these techniques, two modules called "IMUL" and "FMUL" are designed specifically to take advantage of the aforementioned special characteristics of control systems. Furthermore, a custom floating point unit (FPU) is also included, in order to demonstrate the resource and time-wise pros and cons of the proposed methods.

4.2.1.2 Method I-a: Integer multiplication (IMUL module)

In FPGAs, it is possible to develop efficient integer multiplication/division algorithms with the logic-level (i.e. combinational circuit and/or embedded multipliers of the FPGA chip) design. Therefore, the first method, which employs a special multiplier module called "**IMUL**", focuses on multiplication of controller gains with system states employing integer arithmetic. To be specific, let us consider the following operation: $y = \lfloor ax \rfloor$ where a ($\in \Re$) is the multiplicand; x and y ($\in \mathbb{Z}$) are the multiplier and the result (product) respectively while $\lfloor\ \rfloor$ refers to floor function. It is obvious that one can represent the fractional number (a) in this operation as the ratio of two integers (N_a, D_a): $y \cong \frac{N_a x}{D_a}$. Hence, the overall problem is reduced to multiplication and division by integers. Note that the success of this approximation is directly correlated to the bit-length of the numerator and denominator.

4.2.1.3 Method I-b: Fixed point multiplication (FMUL module)

Similar to the previous one, this method focuses on performing multiplication/division where the multiplier is essentially an integer. In fact, the proposed method facilitates a fixed-point multiplication unit (called "**FMUL**") where bit-shift/add operations are successively employed to obtain the result. In this paradigm, the fractional number (*multiplicand*) is separated into an integer- and a fractional portion. Two instances of the multiplier module are used to multiply these portions in parallel. When both calculations are complete, the partial products are added to obtain the result.

4.2.1.4 Method I-c : Floating point multiplication (FPU module)

In previous sections, custom multiplier modules "IMUL" and "FMUL" are presented. Those modules are customized modules developed to perform a specific sort of multiplication; and their main advantage is their low amount of resource requirement. In turn, they may not be applicable to other controller topologies; since they are developed to perform a certain type of multiplication. Therefore, a floating point unit is a necessity, in order to perform more complex arithmetic operations required by advanced controllers. Furthermore, employing floating point unit in state space controller design will be also helpful in demonstrating the resource improvement provided by the custom multiplication modules "IMUL" and "FMUL". The implemented floating point unit is an open core IP, developed by *R. Usselmann [1]*. A schematic of this unit with its input/output ports is shown in Figure 4.2 and the terminology is explained in Table 4.1.

As can be seen in Table 4.1, the FPU performs single precision (32-bit) floating point operations, as well as integer to float and float to integer operations. The operation is performed in one-clock, however the output is provided after a 4 cycle delay period. Therefore, while 2 unrelated operations can be completed in 5 cycles, 2 consecutive operations need 8 clock cycles to be completed.

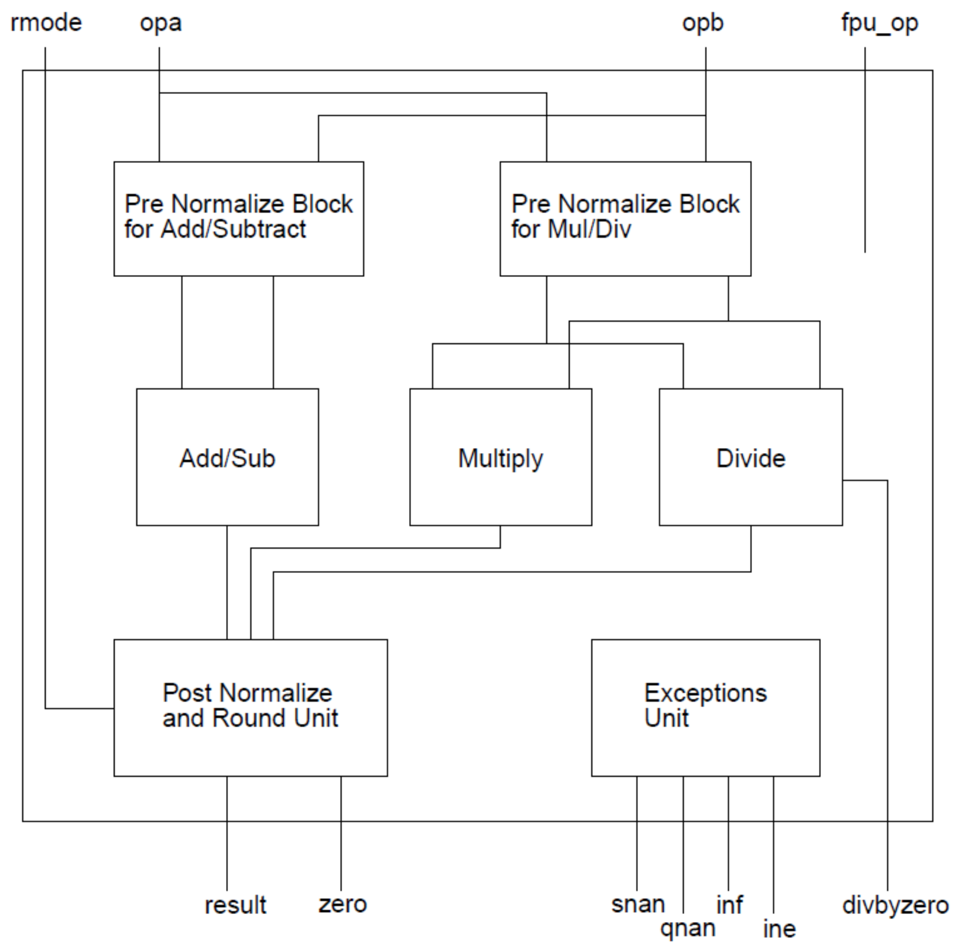

rmode
opa
opb
fpu_op
Pre Normalize Block for Add/Subtract
Pre Normalize Block for Mul/Div
Add/Sub
Multiply
Divide
Post Normalize and Round Unit
Exceptions Unit
result
zero
snan
qnan
inf
ine
divbyzero

Figure 4.2 – Floating point unit

This floating point unit is also adapted as a custom multiplication module, and implemented in the overall architecture as explained in the next section.

Table 4.1 – Terminology used in FPU

Signal name	Length [Bits]	Direction	Explanation
clk	1	Input	System clock
rmode	2	Input	Rounding mode
fpu_op	3	Input	Operation
opa-opb	32	Input	Inputs a and b
out	32	Output	Output
snan	1	Output	Result is not a number
qnan	1	Output	Result is not a number
inf	1	Output	Result is infinity
ine	1	Output	Result is indefinite
overflow	1	Output	There is overflow
underflow	1	Output	There is underflow
div_by_zero	1	Output	Division by zero
zero	1	Output	Result is zero

4.2.1.5 Overall Architecture

A matrix multiplier module can be designed, utilizing the afore-mentioned multiplication modules (IMUL, FMUL and FPU). Proposed module, which operates on (classical) *multiply-and-accumulate* principle, is illustrated in Figure 4.3. Note that the matrices are stored in the SRAM and thus the memory interface module sends out the relevant data to the registers of the multiplier controller unit (a finite state machine) on demand. Hence, the architecture provides flexibility in the controller design as the designers can change the data at will.

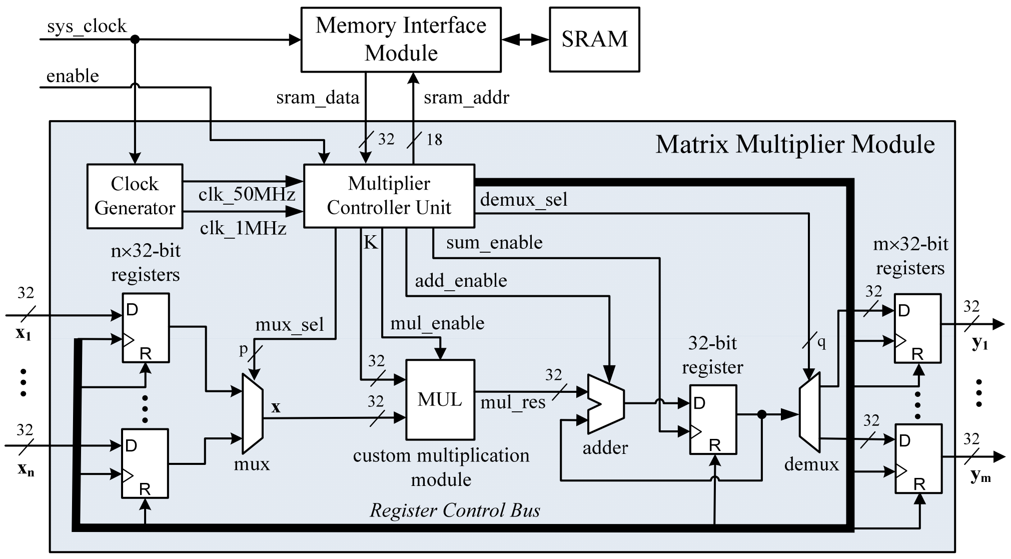


Figure 4.3 – Matrix multiplier module

Similarly, the overall design, which is built on these matrix multiplier units, is presented in Figure 4.4. This unit performs the computations in (4.2) sequentially to obtain $\hat{\mathbf{x}}(k)$ and then proceeds to calculate **u(k)** in (4.1). Note that in the shown architecture all the computations are performed in sequential fashion for the sake of reducing the hardware cost. However, the parallel implementation can be easily realized by eliminating the multiplexer /demultiplexer units in Figure 4.3 while using the instances of custom multiplication modules.

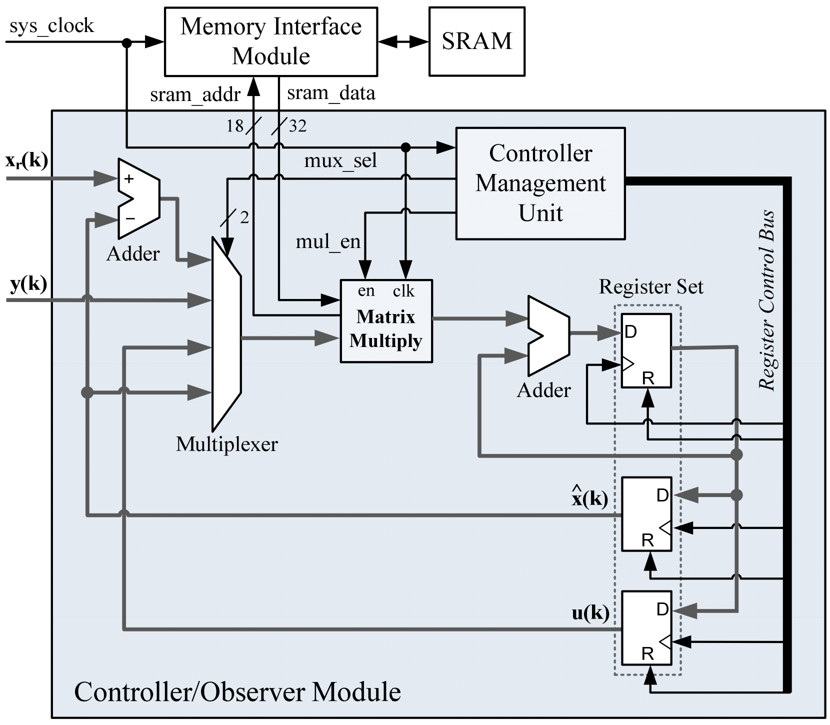


Figure 4.4 – Controller/Observer module

4.2.2 Method II: Softcore processor IP module

Softcore processors are embedded processor IPs developed by FPGA manufacturers, in order to decrease the long and tedious design periods required by low-level (i.e. logic-level) circuit design through hardware description languages (HDL). While shorter and easier design periods are favorable characteristics, the flexibility of the design inevitably decreases due to the bulk implementation of the processor and it may not always be a resource-wise and time-wise beneficial method.

The design approach of a softcore processor is significantly different from the modular approach used in state-space controller implementation and filter implementation via generic filter module. The difference is that, in modular design approach, all the modules are designed with HDLs or schematic design tools and all the connections (data buses, clock signals, enable/reset signals and etc.) between the modules are defined and implemented by the designer. On the other hand, utilizing a softcore processor is via a user-friendly GUI of the design tool provided by the FPGA manufacturer and all the parameters of the processor can be selected easily using this tool. Furthermore, all the peripheral controllers (such as memory management units, serial controllers, GPIOs and etc.) are also provided in this tool, making the overall design process a lot easier and faster.

After implementing the embedded processor along with peripheral controllers to the FPGA, it is possible to develop a C code to realize any algorithm that is supported by the specifications of the implemented processor. In this design method, FPGA implementation of the algorithm is handled by the compiler and therefore the development process is much faster than the modular design approach. Note that, this is one of the most significant differences between the two methods.

4.2.2.1 Method II-a: Softcore processor with integer arithmetic

Implementation details of the second method is as follows; first the specifications of the processor is selected and peripheral controllers such as SRAM and UART controller are added via SOPC (system on programmable chip) builder tool of *Altera Quartus II*. Note that a floating point unit is **not added** to the processor in order to reduce the amount of resources to a minimum. After the embedded processor is implemented to the FPGA, Nios II IDE is utilized to develop and implement the C code into the processor. Similar to methods I-a and I-b, all the parameters and states are stored as 32-bit integers in the C code. A significant advantage of employing a softcore processor is that the processor handles all the SRAM and UART operations, as well as the timing constraints and therefore the design process takes significantly less time than the HDL design case.

Note that the resource requirements of the design does not depend on the algorithm but only the specifications of the processor and peripheral control units. Therefore, only the execution time of the process inevitably depends on the algorithm. As discussed earlier, bulk implementation of the process drastically eases the design in cost of flexibility.

4.2.2.2 Method II-b: Softcore processor with fixed point arithmetic

Implementation details of this module is exactly the same with Method II-a, therefore the resource requirements of this method is also the same. On the other hand, the C code significantly differs from the previous case and multiplications are realized via the fixed point multiplication method presented in Method I-b. The only difference is that in this implementation the “FMUL” module is realized via a C function.

4.2.2.3 Method II-c: Softcore processor with floating point arithmetic

The only difference of this method from the preceding one is the floating point arithmetic unit employed in the design. Therefore, the only change in the methodology is in the specifications of the processor selected. In this method, an optional floating point arithmetic unit (which is not the same unit employed in Method I-c) is added via the SOPC builder tool to the processor and the C code is modified so that the parameters are stored as 32-bit floating point variables.

In the next section, the test case (a nonlinear inverted pendulum system) which is employed via hardware in the loop simulation is presented. In the subsequent section, results of the simulation are provided for all of the presented methods.

4.3 Test case

As a benchmark case, an inverted pendulum system shown in Figure 4.5 is considered.

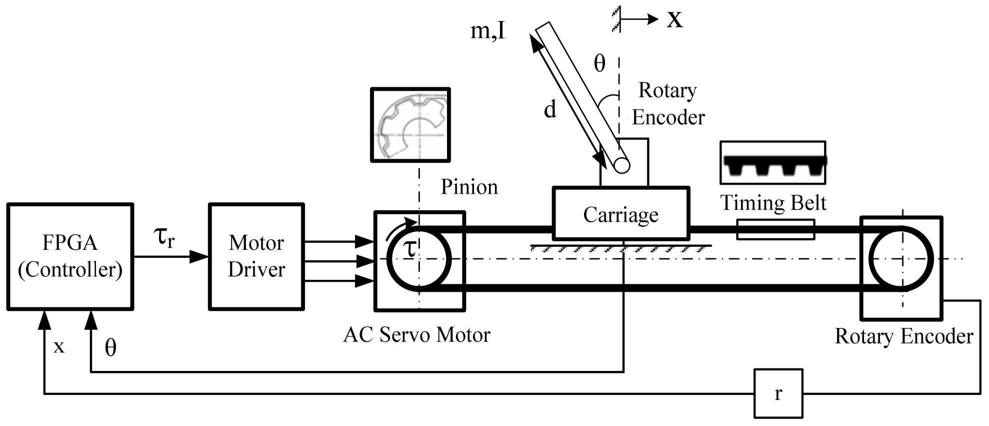


Figure 4.5 – Generic model for pendulum drive system

The system includes an AC servo-motor coupled directly to a timing belt. In this configuration, the motor driver is in the torque regulation mode where the motor along with its driver can be regarded as an ideal torque modulator. Hence, the state-space controller can directly generate the relevant (torque) commands through a digital-to-analog converter. Note that two rotary encoders (which can produce 10000 pulses/rev) are to supply feedback on the position of the carriage as well as the angular position of the pendulum. This choice is also convenient since the system has four states; $(x, \dot{x}, \theta \; and \; \dot{\theta})$ and given that only two states (x, θ) are available via encoder feedback in order to estimate $\dot{x} \; and \; \dot{\theta}$, an observer is required to be implemented in the design that satisfies the testing requirements for the developed methods.

Notice that this can be regarded as a "quantized input/output system" since all the sensor feedback is coming from incremental encoders as pulses and that the output of the FPGA is also an integer representing the motor torque. However, the system cannot be considered as a "fully quantized system" since there exist an observer in the design and while the coefficients of the controller can be cast to close integers via pole placement, it is not possible to implement an observer by using only integers.

The force acting on the carriage is related to the motor torque as F = τη/r; here τ is the motor torque [Nm]; r is the pinion (pitch circle) radius [m] and η is the overall transmission efficiency. Equations of motion for this system become

$$(M+m)\ddot{x} + b\dot{x} + m\frac{d}{2}\ddot{\theta}cos\theta - m\frac{d}{2}\dot{\theta}^2 sin\theta = \frac{\tau}{r}\eta \qquad (4.3)$$

$$\left(I + m\frac{d^2}{4}\right)\ddot{\theta} + mg\frac{d}{2}sin\theta = -m\frac{d}{2}\ddot{x}cos\theta \qquad (4.4)$$

Here, M is the mass of the carriage [kg]; b is its viscous damping [Nms]; m is the mass of the pendulum, I is its mass moment of inertia [kgm^2]; τ is the manipulated input (i.e. motor torque). Numerical values for the system parameters are provided in Table 4.2.

Table 4.2 – Inverted pendulum parameters

Parameter	M [kg]	m [kg]	I [kgm^2]	b [Nms]	d [m]	r [m]
Value	0.5	0.2	0.006	0.1	0.6	0.01

In order to design a controller and a state observer, the system is linearized around an operating point (θ = 0), since the goal of the control system is to hold the pendulum in upright position. Using (4.3) and (4.4), the state-space representation of the linearized system can be given as:

$$\frac{d}{dt}\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{-b}{C_2} & -m\frac{d}{2}\frac{C_1}{C_2}g & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{-C_1}{C_2}b & \frac{C_1}{C_2}(M+m)g & 0 \end{bmatrix}}_{\boldsymbol{A}} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ \frac{\eta}{rC_2} \\ 0 \\ \frac{\eta C_1}{rC_2} \end{bmatrix}}_{\boldsymbol{B}} \tau \tag{4.5a}$$

$$\mathbf{y} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{C}} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} \tag{4.5b}$$

$$C_1 = -m\frac{d}{2}\left(I + m\frac{d^2}{4}\right)^{-1}, C_2 = M + m + \frac{C_1 d}{2} \tag{4.6}$$

Having obtained the continuous-time state space representation, the discrete representations of (4.5) can be simply obtained as:

$$\boldsymbol{x}(k+1) = \mathbf{F}\boldsymbol{x}(k) + \mathbf{G}u(k) \tag{4.7a}$$

$$\boldsymbol{y}(k) = \mathbf{H}\boldsymbol{x}(k) \tag{4.7b}$$

where

$$\mathbf{F} = e^{\mathbf{A}T}; \quad \mathbf{G} = \left(\int_{\mathbf{0}}^{\mathrm{T}} e^{\mathbf{A}t}\, \mathrm{d}t\right)\mathbf{B}; \quad \mathbf{H} = \mathbf{C} \tag{4.8}$$

Here, **A**, **B** and **C** denote to the system matrices of (4.5) while T refers to the sampling period of the controller [s]. With the discrete-time model at hand, a state-space controller (along with an observer) can be designed using modern control theory.

4.4 Hardware-in-the-loop simulation results

In the simulation study, the controller topology presented is implemented on the Altera Cyclone II FPGA chip using Verilog hardware definition language. The nonlinear inverted pendulum system (as explained in the previous section) is realized via (non-real-time) hardware-in-the-loop simulation (HILS) in MATLAB environment.

The controller system (controller+observer) is implemented on the FPGA using the presented techniques: Method I-a (integer arithmetic - IMUL), Method I-b (fixed-point arithmetic – FMUL), Method I-c (floating-point arithmetic – FPU), Method II-a (integer arithmetic with softcore processor), Method II-b (fixed-point arithmetic with softcore processor), and Method II-c (floating point arithmetic with softcore processor).

In this scheme, the manipulated torque input ($u = \tau$) generated by the FPGA is transmitted to hardware-in-the-loop simulator running on MATLAB platform. Once the new system states are computed using the dynamic model of the controlled system, they are sent back to the FPGA via RS-232 communication. As explained in preceding sections, the word size allocated to the matrix elements is a major design parameter. In this study, a 2×16-bit word size is selected for each element of the controller/observer matrices: 32-bit for numerator/ denominator pair for IMUL multiplication module (Method I-a) or integer/ fractional portion pair for FMUL module (Method I-b). On the other hand, the system states are stored as 32-bit signed integers. However, this choice is also a design choice and can be modified according to the needs of a particular application.

Results of the HILS are presented in Figs. Figure 4.6 through Figure 4.11. Note that the initial condition of the state vector is selected as $[0\ 0\ -0.05\ 0]^T$ to start out with a pendulum angle to create a challenge for the controller. As can be seen in Figs. Figure 4.6 through Figure 4.11, all controllers react fast and bring the carriage around 0.02m to prevent pendulum arm from falling; and after the initial reaction, the implemented controllers can successfully hold the inverted pendulum around $\theta = 0$ and the carriage's position are bounded around its initial position. Since the system is nonlinear, small differences in calculation of the results between the proposed methods: I-a (II-a), I-b (II-b), and I-c (II-c) cause a relatively different path for the carriage after t = 2s; Nevertheless, all sub-methods of method I yield acceptable performance.

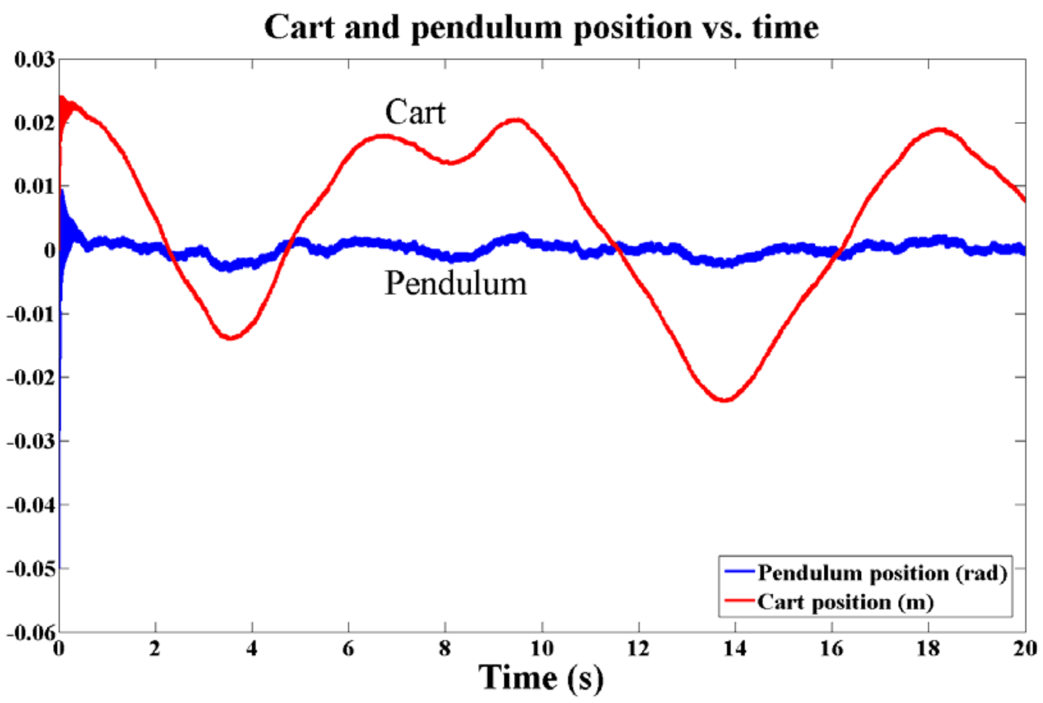


Figure 4.6 – HILS result of method I-a

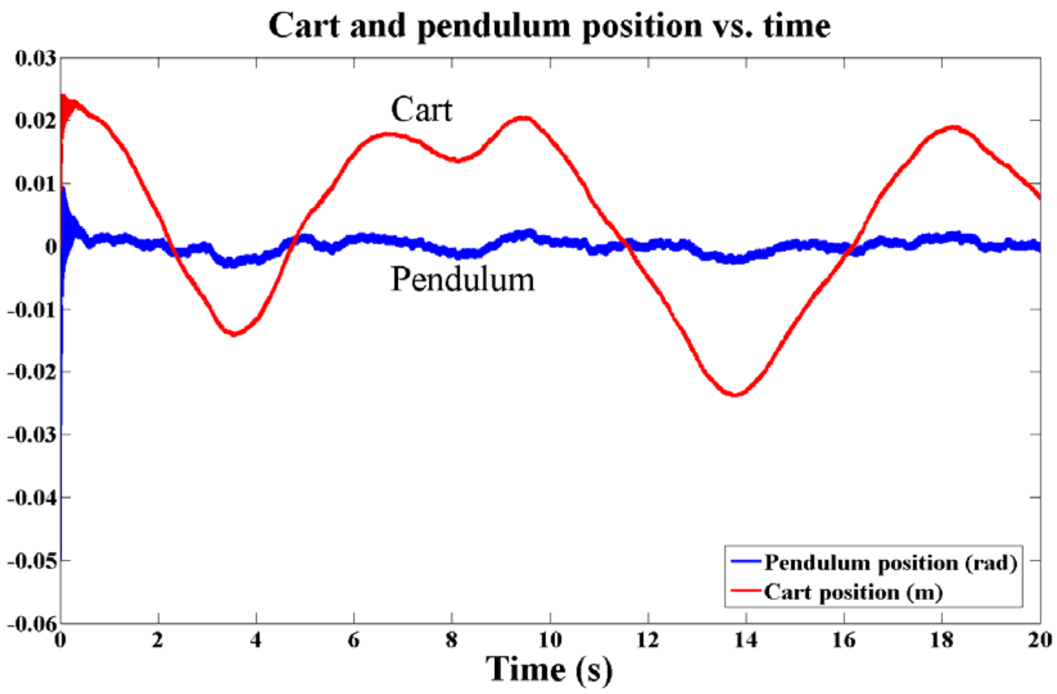


Figure 4.7 – HILS result of method II-a

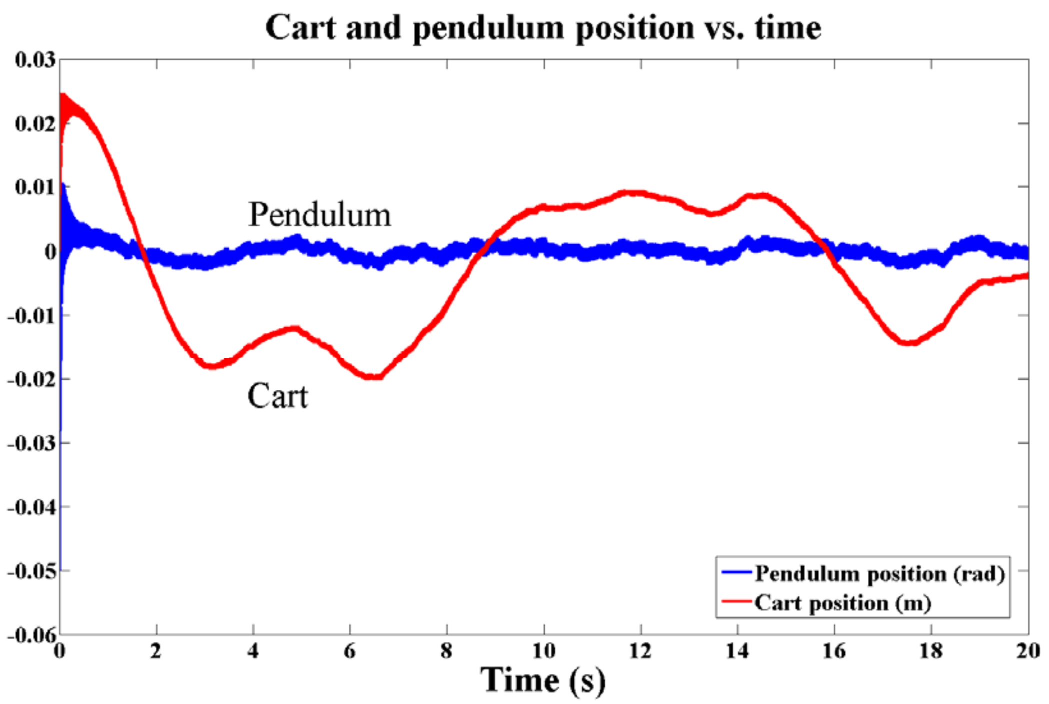


Figure 4.8 – HILS result of method I-b

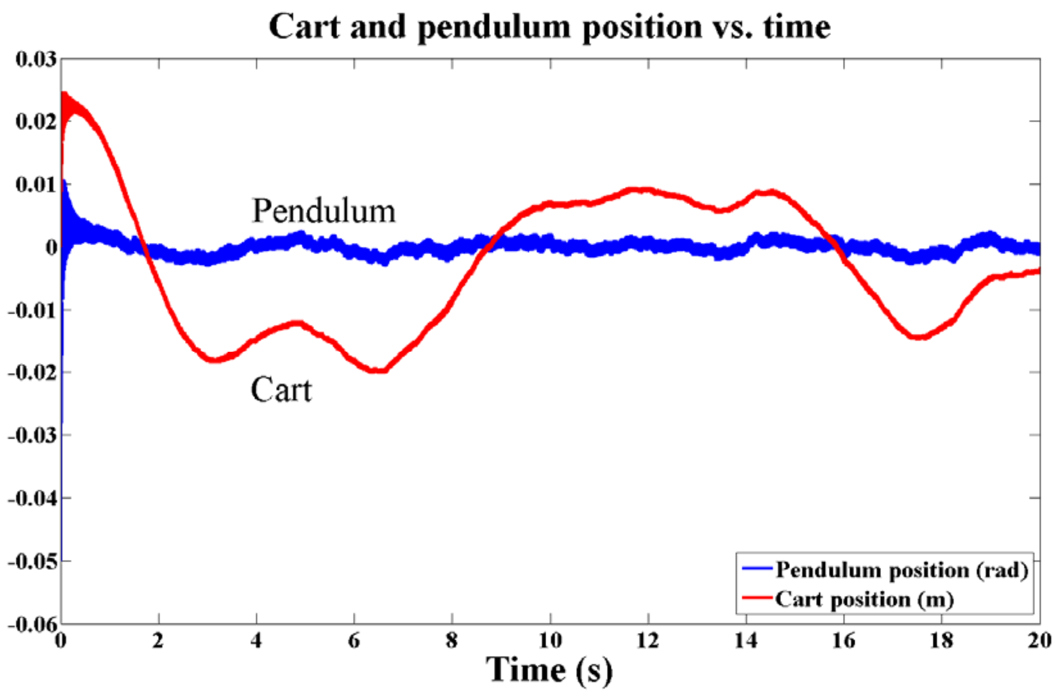


Figure 4.9 – HILS result of method II-b

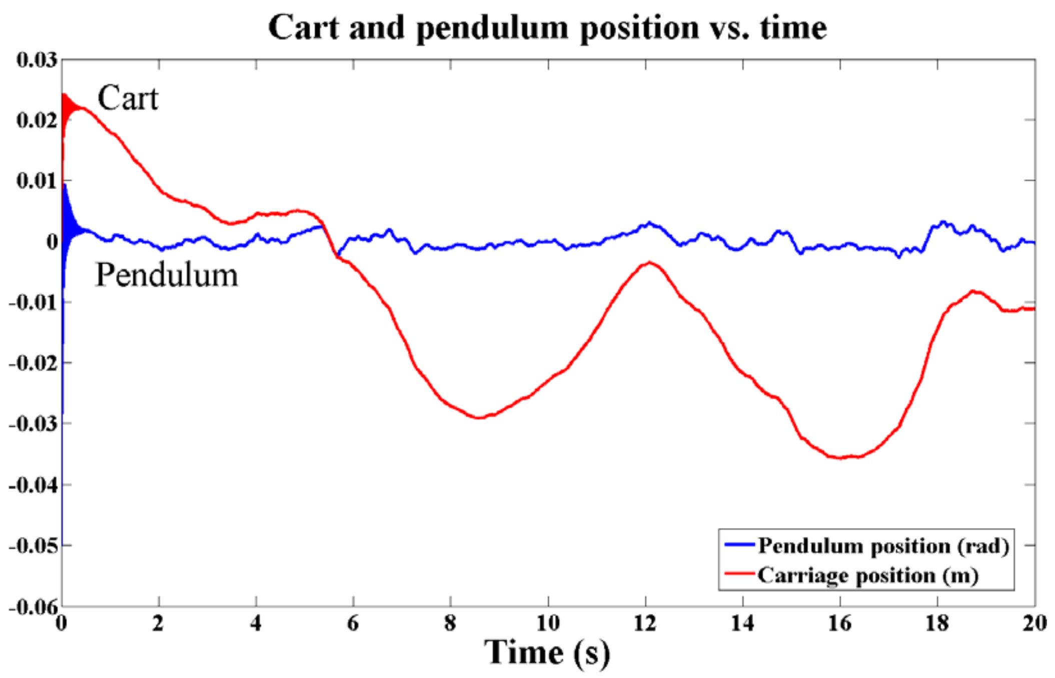


Figure 4.10 – HILS result of method I-c

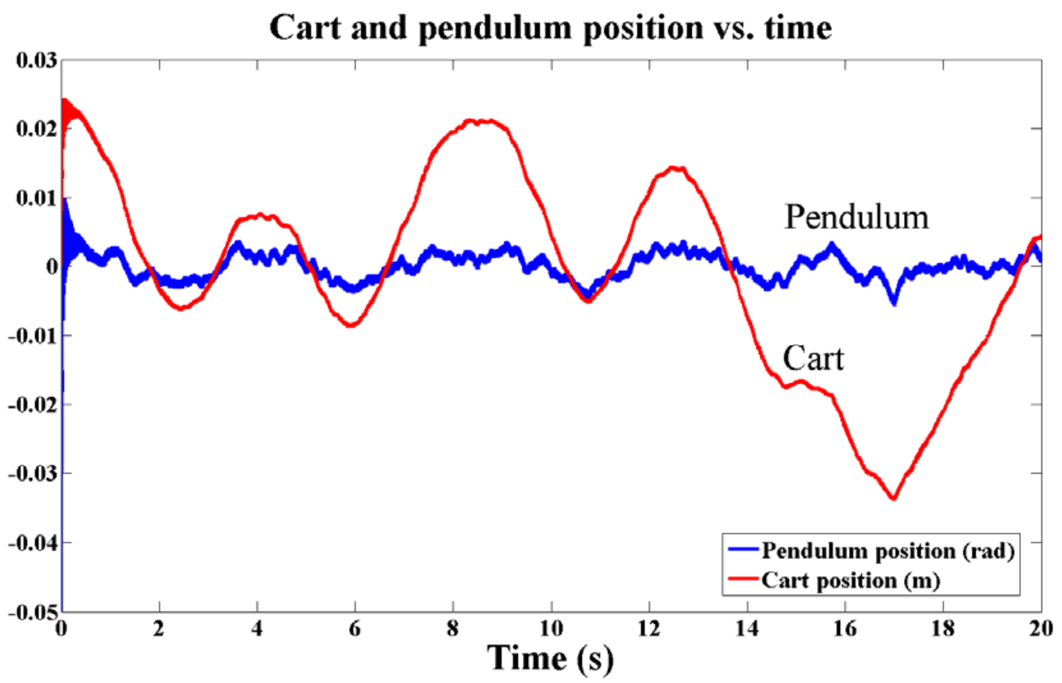


Figure 4.11 – HILS result of method II-c

As expected, result of the method II-a is exactly the same with method I-a, because even if the approach is different, both of the implementations are based on integer arithmetic and therefore this result is predicted. The case is also the same with method II-b and I-b, since the fixed point arithmetic algorithm is the same. On the other hand, as seen in Figure 4.11, result of method II-c is not the same with method I-c, even if they both employ a floating point unit for calculations. However, this is also expected since the floating point units are not the same (method I-c employs an open floating point by *Usselmann [1]* and method II-c employs a floating point developed by *Altera* for *Nios II* processors) and a small computational difference yields a different trajectory.

4.5 Comparison and discussion

In the previous section, it has been proven that both Method I and Method II are feasible options for implementing a state space controller and an observer on an FPGA. However, certain factors such as: total logic elements used, number of clock cycles to complete loop, hardware multipliers employed and etc. should also be taken into consideration, when developing a design for FPGAs. An illustrative presentation of the resource consumptions of these methods on the *Altera Cyclone II* FPGA chip are obtained using the *"Chip planner"* tool of *Quartus II* software and provided in Figs. Figure 4.12 through Figure 4.16.

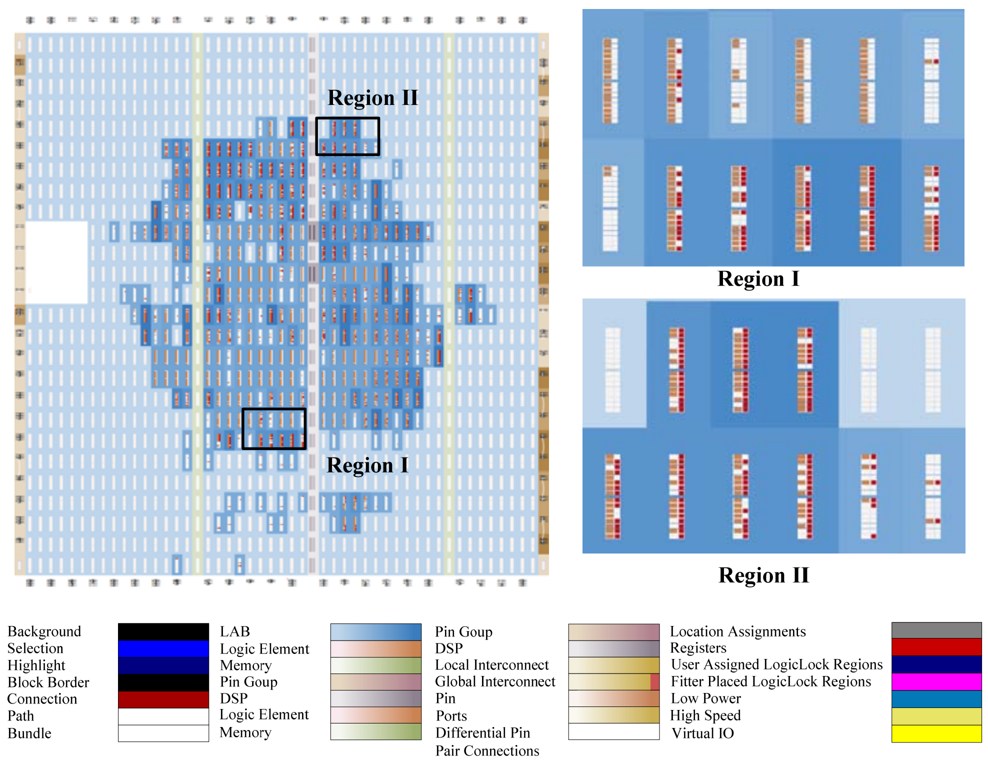


Figure 4.12 – Resource utilization of Method I-a on Cyclone II FPGA

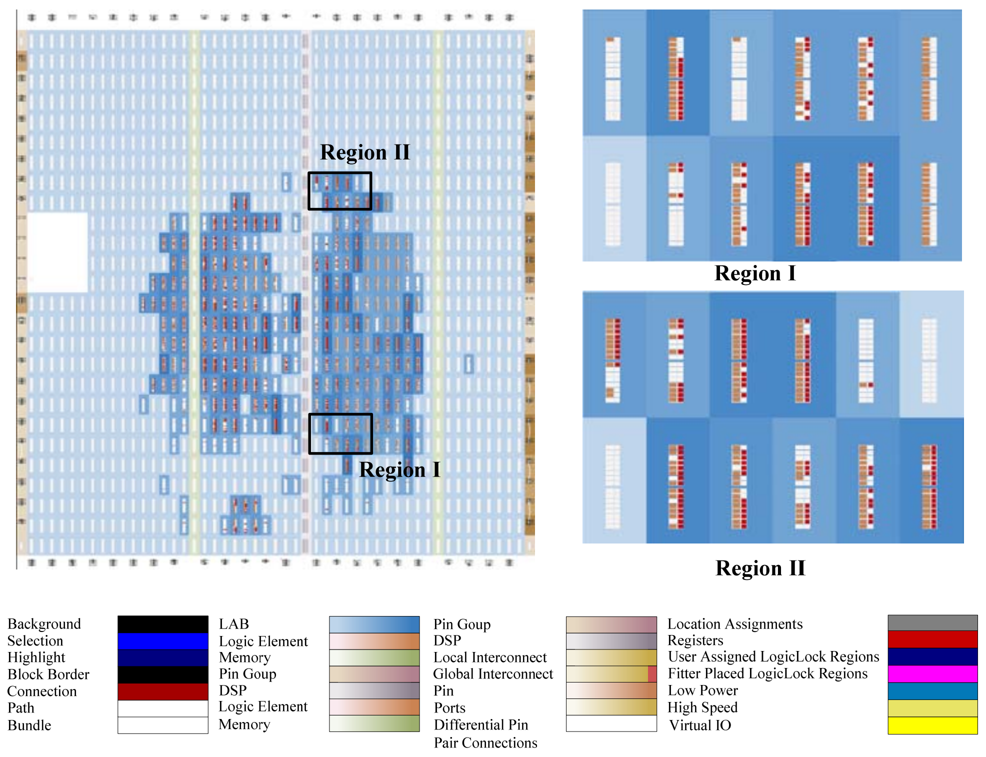


Figure 4.13– Resource utilization of Method I-b on Cyclone II FPGA

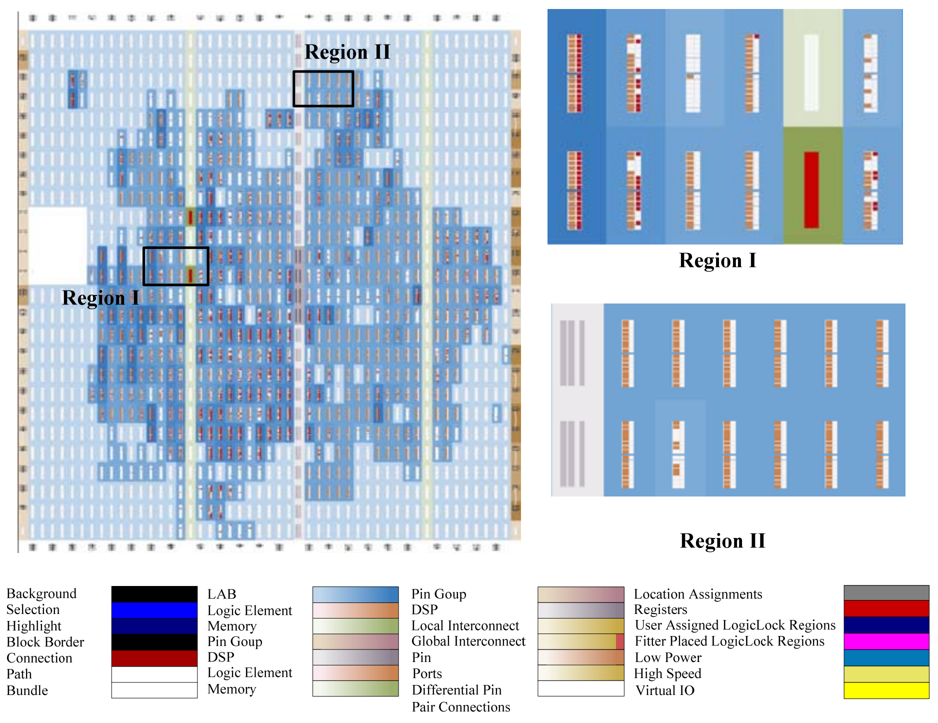


Figure 4.14 – Resource utilization of Method I-c on Cyclone II FPGA

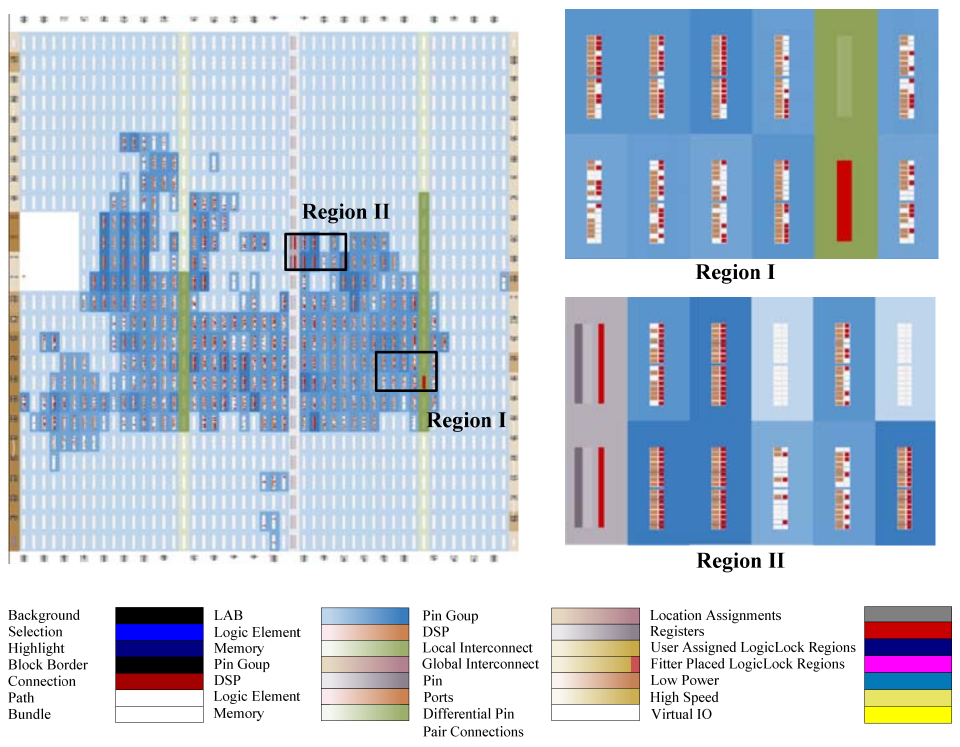


Figure 4.15 – Resource utilization of Methods II-a & II-b on Cyclone II FPGA

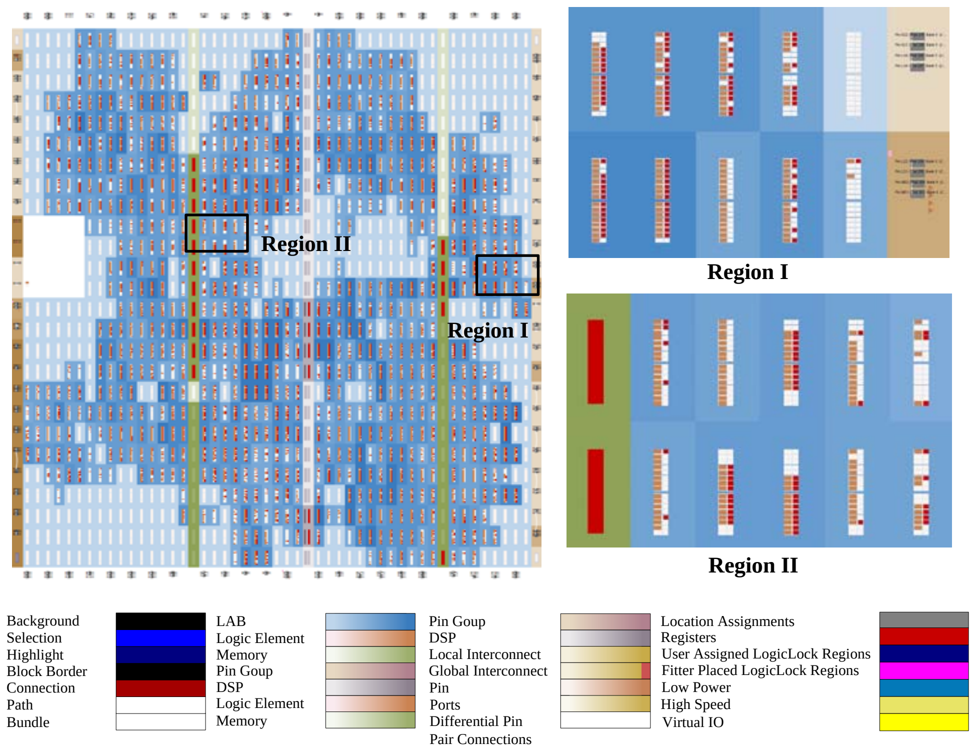


Figure 4.16 – Resource utilization of Method II-c on Cyclone II FPGA

Three illustrations that are provided for Method I-a, I-b and I-c include three different modules employed for different types of computation, which are IMUL, FMUL and FPU modules. On the other hand, in Method II, since a softcore processor is employed, the difference between the illustrations is the difference in the specifications of the processor, which is the floating point arithmetic module. Note that a different module than the FPU module which is used in method I-c. Similarly, resource requirements of each method are summarized in Table 4.3. It is important to notice that resource utilizations (i.e. resource requirements and floor plans) may vary slightly due to the optimization performed by the fitter of the IDE tool provided by the FPGA manufacturer.

Table 4.3 – Resource/Time Costs of Proposed Methods on Altera Cyclone II FPGA chip

Method	Type of Comp.*	Total logic elements	Embedded multipliers	Total clocks required	Max. frequency
I-a	INT	4449 (24%)	4 (8%)	4×32 cycles	400 kHz
I-b	FIXED	3528 (19%)	0 (0%)	32×32 cycles	50 kHz
I-c	FP	7778 (41%)	7 (13%)	20×32 cycles	80 kHz
II-a	INT	4504 (24%)	4 (8%)	3907	12.8 kHz
II-b	FIXED	4504 (24%)	4 (8%)	33185	1.5 kHz
II-c	FP	10595 (54%)	11 (21%)	16051	3.1 kHz

*abbreviations: INT= Integer, FIXED= Fixed point, FP= Floating Point

For method I, clock cycles necessary depend on not only the number of states of the controlled system but also the multiplication algorithm selected. The number of base cycles (N_c) can be expressed as follows:

$$N_c = n(n + 2m + n_y) \tag{4.9}$$

Here, n is the number of states; n_y is the number of measured states and m is the number of manipulated inputs. In this case, since n = 4, n_y = 2 and m = 1; N_c becomes 32 (as indicated in Table 4.3). The duration of each base cycle depends on the design of the multiplier unit.

As can be seen in Table 4.3, the proposed method I increases the attainable sampling rate of the controller significantly. As discussed earlier, this is due to the custom designed and application specific modules. It is also evident that the type of computation also has an effect on the attainable sampling rate, however the slowest frequency attainable by method I is still 3 times faster than the fastest frequency attainable by method II.

It is critical to note that since the speed of the serial communication (RS-232) used in this study is not sufficient for real-time simulation, a non-real time HILS is realized. For the simulation, a sampling frequency of 1 kHz is selected and thus the discrete-time state space representations are evaluated using Eqns. 4.7-4.8.

In terms of resource requirements, it is clear that when floating point operations are involved, resource requirements of the design increase drastically. This is proven by both of the methods with an increase of 3 to 6 thousands of logic elements and 3 to 11 embedded multipliers. While the most resource-wise efficient method seems to be method I-b, all the methods that are not employing floating point arithmetic (method I-a, method II-a and method II-b) are also comparable. This shows that for modules with high complexity, bulk implementation of the controller does not increase the resource requirement significantly, since the customized module would also require a significant amount of FPGA resources. In conclusion, for a mediocre FPGA chip such as *Altera Cyclone II*, utilizing floating point arithmetic is a costly method; especially when it is proven that good control performances can be attained by exploiting certain properties of motion control systems (such as quantized input/output property) as explained in section 4.2, in order to reduce the resource requirements of the design. It is also proven that customized HDL modules are more advantageous in terms of speed, rather than resource utilization, especially in complex tasks. However, complexity of the task is also a vague description and it is necessary to implement both methods to see which one would be more resource-wise efficient.

4.6 Closure

In this chapter, a state space controller with a state observer is designed and implemented for the inverted pendulum problem. In section 4.2, 5 different methods for computation of the necessary calculations are explained thoroughly under 2 different design methods. The HILS results suggest that while the modular approach presented as Method I offers a slight increase in terms of resource requirements, it is definitely faster than the softcore processor method that is presented as Method II.

CHAPTER 5

IMPLEMENTATION OF DIGITAL FILTERS

Generally, one or more filters are usually employed in a typical motion control system. In digital motion control applications, these filters are realized by digital filtering algorithms and can be implemented on various signal processing chips such as: DSPs, ASICs, FPGAs and etc. In this chapter, after a brief overview of digital filters that are commonly used in motion control applications, two different methods (one incorporating the previously mentioned multiplication methods and the other utilizing the *Nios II* softcore processor) for FPGA implementation of a general infinite impulse response (IIR) filter is presented, along with hardware-in-the-loop simulations (HILS) of a cascade control architecture realized by the two alternative methods. The chapter is finalized by a quantitative comparison between the two design methodologies.

5.1 Digital Filters

In this section, a brief overview of digital filters is presented, within a limited scope of digital motion control applications. A digital filter is commonly employed to modify certain aspects of a signal and can be classified into two categories, based on its impulse response or in particular, its feedback property. A finite impulse response (FIR) filter is a digital filter that generates its output signal by using only current and past values of its input. Therefore, it has no internal feedback and the impulse response dies out to zero, as the name implies. On the other hand, an infinite impulse response (IIR) filter contains an internal feedback and therefore its output depends on its past output values as well as past input values. The constant coefficient difference equation of an IIR filter is provided in (5.1).

It is more convenient to discuss the implementation of an IIR filter rather than the FIR, since it is more general and can also be modified to obtain an FIR, as can be observed in (5.1). Therefore, the subsequent discussions are based on the IIR filter.

$$y(k) = \sum_{i=0}^{n} b_i x(k-i) + \sum_{j=0}^{m} a_i y(k-j) \tag{5.1}$$

Using the generalized expression of IIR, it is possible to deduce that the implementation requires multiplication of the filter coefficients with past outputs and past inputs, followed by an addition operation, which is a very similar case to the state-space controller implementation that is presented in the previous chapter. Therefore, it is possible to realize an IIR filter (hence also an FIR filter) on an FPGA, utilizing the previously described methods.

Most of the filters such as: Notch filter, Low-pass filter and High-pass filter and etc. that are commonly used in motion control systems are all IIR filters and can be realized by a general IIR filter module. Furthermore, difference equation of a PID controller, as can be seen in (5.2):

$$m(k) = m(k-1) + b_0 e(k) + b_1 e(k-1) + b_2 e(k-2) \tag{5.2}$$

where m is the manipulated input, e is the error and k is the time step, is also a variation of (5.1) and can be implemented by the same module. Therefore it is desirable to develop a generic IIR filter module for FPGAs, in order to realize certain controller topologies, as well as digital filters. Implementation details of digital filters are explained in the following section.

5.2 Implementation

In this section, two alternative methods for implementation of digital filters on FPGAs are presented. The first method is development of a generic filter module utilizing a custom multiplication module, which is presented in the previous chapter. The second method presents the implementation of a softcore processor, Nios II and requires a significantly different design approach for the problem, as explained in the previous chapter.

5.2.1 Method I: Generic filter module

Generic filter module is analogous to the matrix multiplier module presented in the previous chapter. Similar to the previous case, this module utilizes a custom multiplication module (which is selected as IMUL in this case) and an adder, in order to calculate the result of (5.1). Figure 5.1 presents a schematic representation of the generic filter module. Here x represents input at the current time step x(k), y represents the output at the current time step y(k), x_n and y_m represents the input at the (k-n)th time step and output at the (k-m)th time step, that are x(k-n) and y(k-m) respectively.

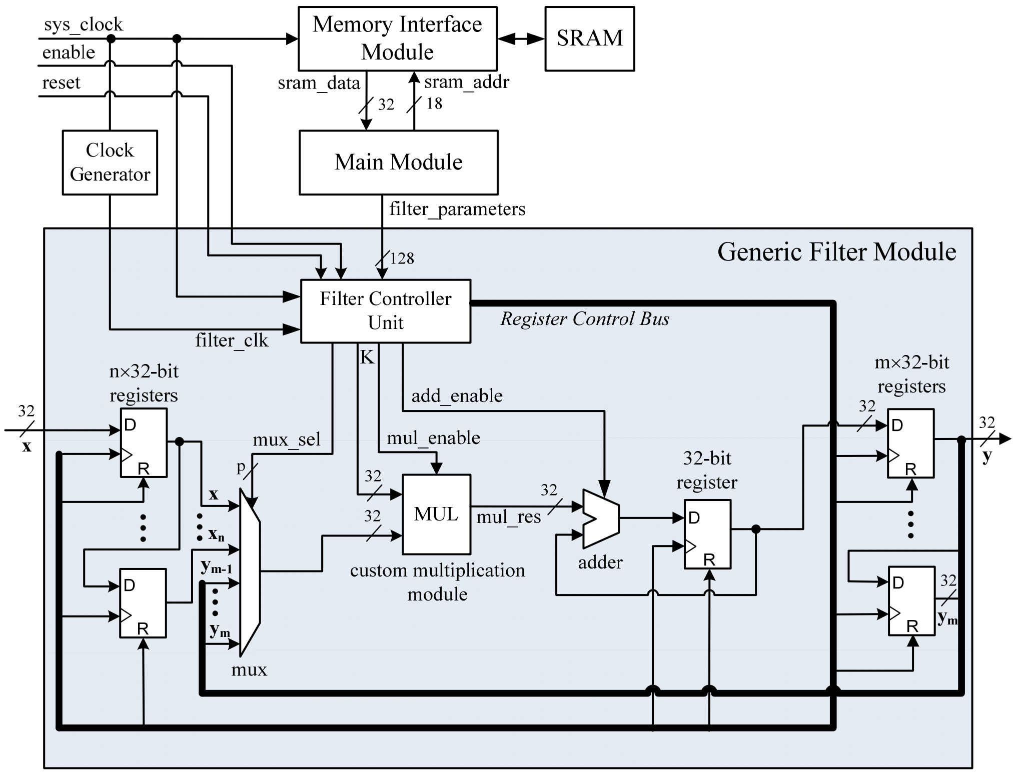


Figure 5.1 – Generic filter module

There are only two design parameters to be selected when utilizing this generic filter module; that are n (the number of past input values) and m (the number of past output values). While these parameters can be selected as relatively large integers, in order to avoid further modifications to the HDL code, it is also possible to select those parameters as required minimum values as required for a certain filter, to minimize resource requirements of the module.

5.2.2 Method II: Softcore processor

Softcore processors are embedded processor IPs developed by FPGA manufacturers, in order to decrease the long and tedious design periods required by low-level (i.e. logic-level) circuit design through hardware description languages (HDL). While shorter and easier design periods are favorable characteristics, the flexibility of the design inevitably decreases due to the bulk implementation of the processor and it may not always be a resource-wise and time-wise beneficial method.

The design approach of a softcore processor is significantly different from the modular approach used in state-space controller implementation and filter implementation via generic filter module. The difference is that, in modular design approach, all the modules are designed with HDLs or schematic design tools and all the connections (data buses, clock signals, enable/reset signals and etc.) between the modules are defined and implemented by the designer. On the other hand, utilizing a softcore processor is via a user-friendly GUI of the design tool provided by the FPGA manufacturer and all the parameters of the processor can be selected easily using this tool. Furthermore, all the peripheral controllers (such as memory management units, serial controllers, GPIOs and etc.) are also provided in this tool, making the overall design process a lot easier and faster.

After implementing the embedded processor along with peripheral controllers to the FPGA, it is possible to develop a C code to realize any algorithm that is supported by the specifications of the implemented processor. In this design method, FPGA implementation of the algorithm is handled by the compiler and therefore the development process is much faster than the modular design approach. Note that, this is one of the most significant differences between the two methods.

Implementation details of the second method is as follows; first the specifications of the processor is selected and peripheral controllers such as SRAM and UART controller are added via SOPC (system on programmable chip) builder tool of *Altera Quartus II*. After the embedded processor is implemented to the FPGA, Nios II IDE is utilized to develop and implement the C code into the processor. Note that the resource requirements of the design does not depend on the algorithm but only the specifications of the processor and peripheral control units. However, execution time of the process inevitably depends on the algorithm. Results of the hardware-in-the-loop simulations are provided in the following section.

5.3 Hardware-in-the-loop simulation results

Hardware-in-the-loop simulation is realized by the inverted pendulum system presented in the previous chapter. In order to demonstrate the performance of the developed methods, a cascade controller utilizing PID controllers as explained in section 5.1 is implemented. The cascade control system consists of a PD controller in the outer loop for control of the position of the cart and a PID controller in the inner loop for control of the angular position of the pendulum.

Figure 5.2 – Cascaded control system illustrates the control system.

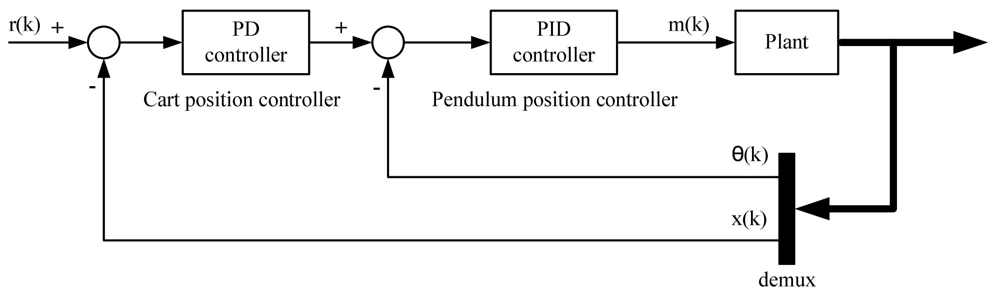


Figure 5.2 – Cascaded control system

In Figure 5.2, r is the reference input, m is the manipulated input, θ is the angular position of the pendulum and x is the linear position of the cart. As can be seen in Figure 5.2, two "generic filter module" instances are required to realize the cascade controller system. On the other hand, resource requirement of the softcore processor doesn't depend on the number of instances required, just as it would be the case in a regular microprocessor.

Results of the HILS using the generic filter module (Method I) with imul and FPU are shown in Figure 5.3 and 5.4 respectively. Results of the HILS using the Nios II softcore processor (Method II) are shown in Figure 5.5 and Figure 5.6 respectively. Note that the initial conditions for all simulations are selected as $x_0 = 0$ m and $\theta_0 = -0.05$ rad.

As can be observed from Figure 5.3, Figure 5.4, Figure 5.5 and Figure 5.6, the cascade control system is able to hold the inverted pendulum in an upward position while the position of the cart is bounded and settling to zero. Furthermore, it can be seen that there is no significant difference between the figures as expected, since the implemented controllers are the same even if the methodologies are drastically different. Having obtained these results, it is possible to evaluate and compare these methodologies in terms of their resource requirement, execution time and ease of implementation; which will be the subject of the following section.

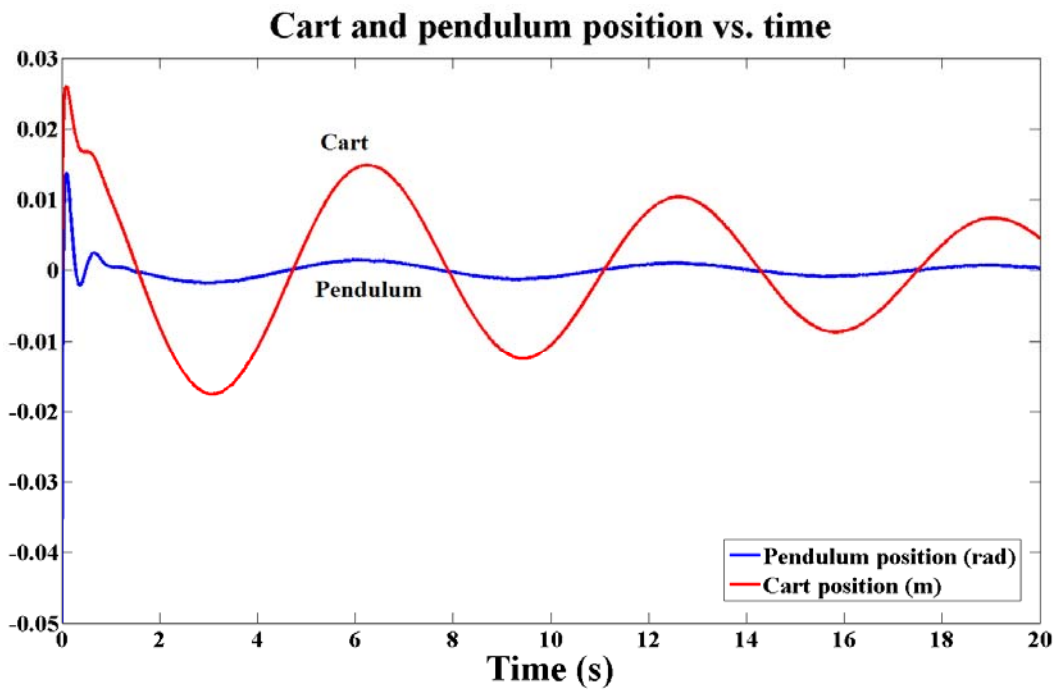


Figure 5.3 – HILS result of method I with IMUL

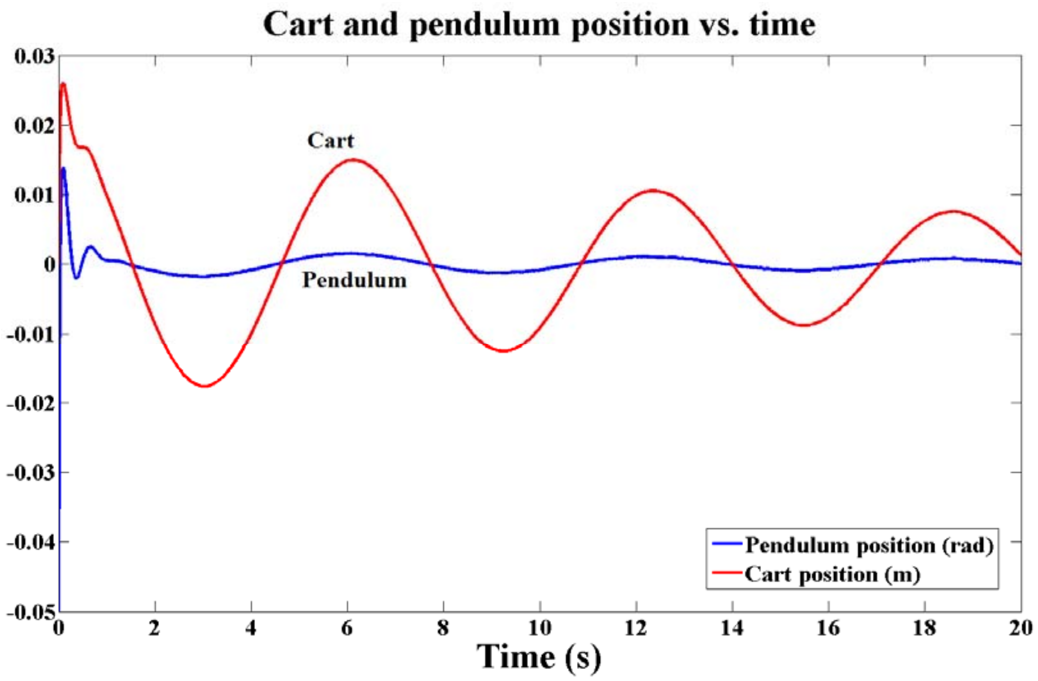


Figure 5.4 – HILS result of method I with FPU

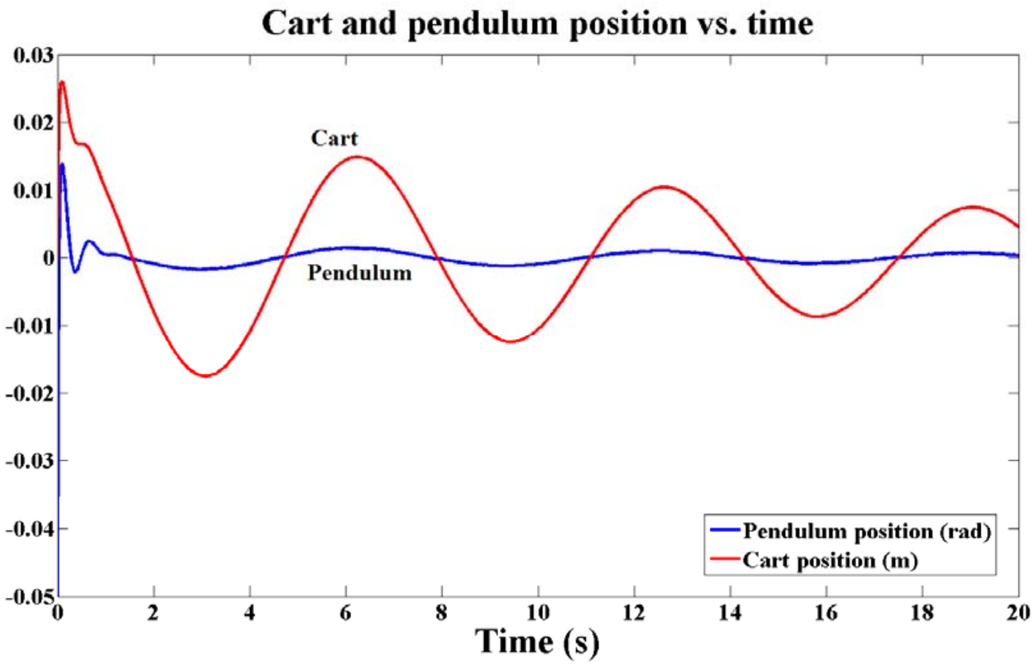


Figure 5.5 – HILS result of method II with IMUL

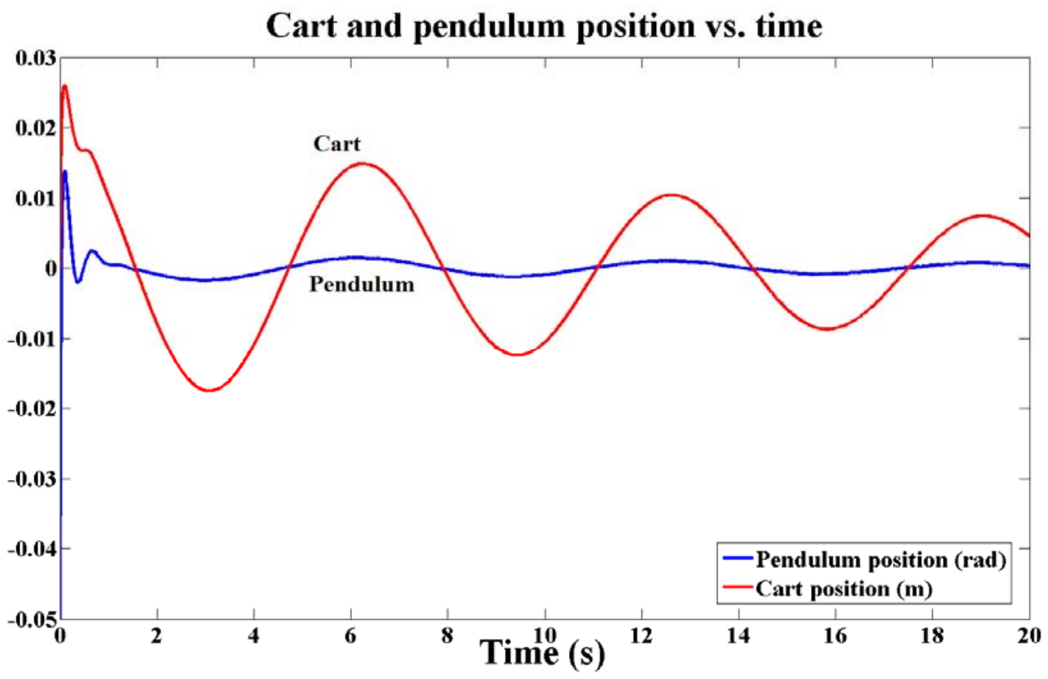


Figure 5.6 – HILS result of method II with FPU

5.4 Comparison and discussion

In the previous section, it has been shown that both method I and method II are feasible options for implementing digital filters on an FPGA. However, certain factors such as: total logic elements used, number of clock cycles to complete loop, hardware multipliers employed and etc. should also be taken into consideration, when developing a design for FPGAs. An illustrative presentation of the resource consumptions of these methods on the *Altera Cyclone II* FPGA chip are obtained using the "*Chip planner*" tool of *Quartus II* software and provided in Figure 5.7, Figure 5.8, Figure 5.9 and Figure 5.10.

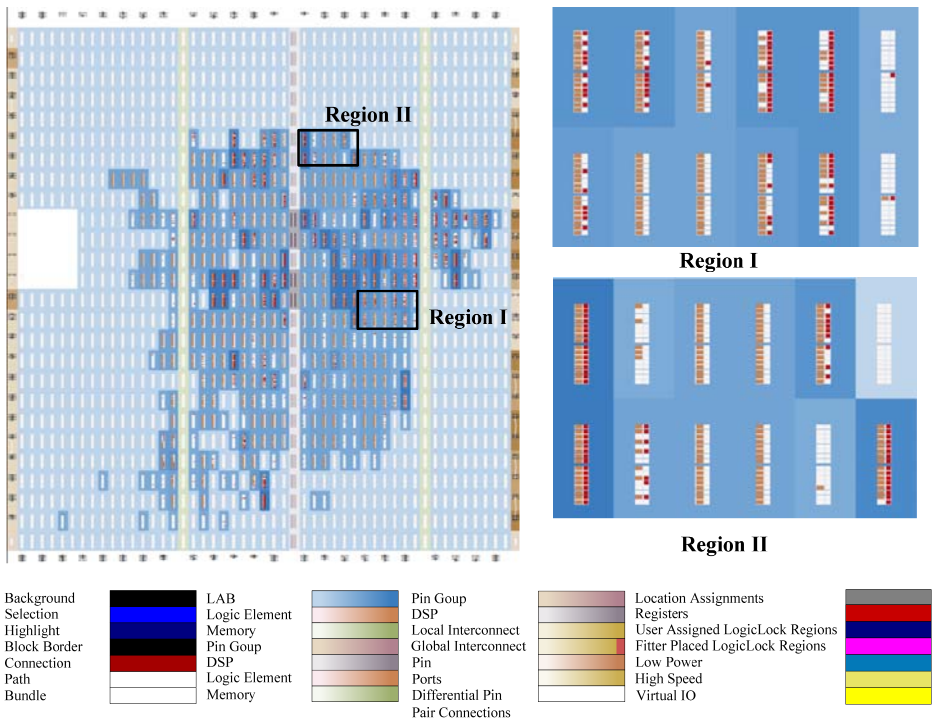


Figure 5.7 – Resource consumption of Method I with IMUL

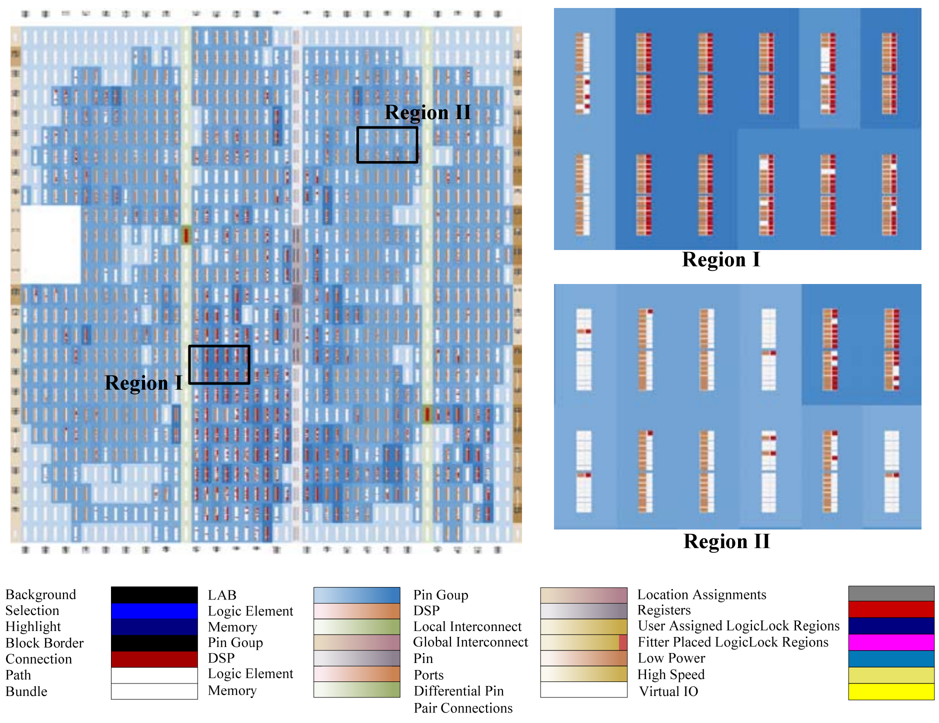


Figure 5.8 – Resource consumption of Method I with FPU

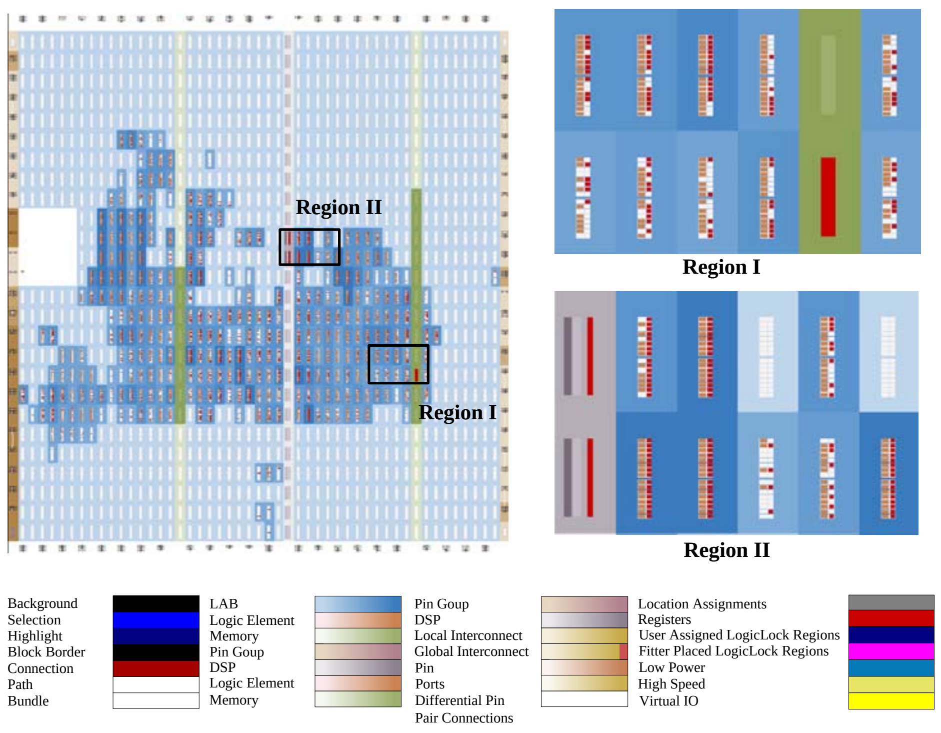


Figure 5.9 – Resource consumption of Method II with integer arithmetic

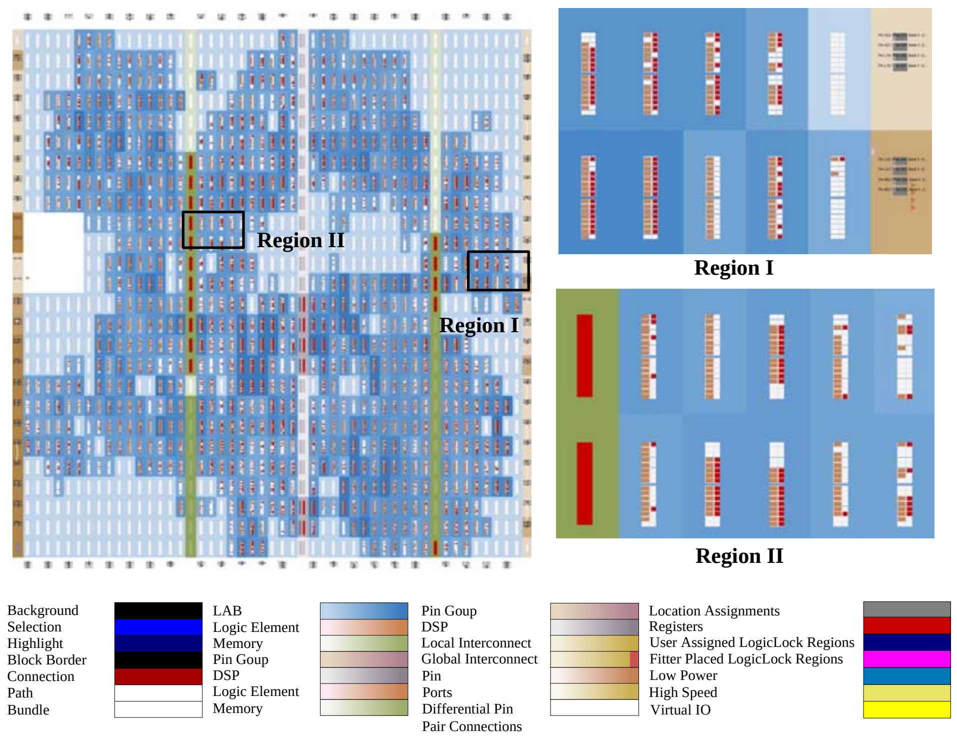


Figure 5.10 – Resource consumption of Method II with FPU

Two illustrations provided for Method I include two different modules employed for two different types of computation, which are IMUL and FPU modules. On the other hand, in Method II, since a softcore processor is employed, the difference between the illustrations is the difference in the specifications of the processor, which is the floating point arithmetic module. Note that this module is not the same module which is used in the generic filter module. The resource/time costs of the proposed methods are tabulated in Table 5.1.

TABLE 5.1 Resource/Time Costs of Proposed Methods on Altera Cyclone II FPGA chip

Method	**Type of Comp.***	**Total logic elements**	**Embedded multipliers**	**Total clock cycles required**	**Max. frequency**
I	INT	5012 (27%)	8 (15%)	4×4 cycles	3 MHz
I	FP	11180 (60%)	14 (27%)	20×4 cycles	625 kHz
II	INT	4504 (24%)	4 (8%)	400	125 kHz
II	FP	10595 (54%)	11 (21%)	3270	15.3 kHz

*abbreviations: INT= Integer, FP= Floating Point

As shown in Table 5.1, in terms of logic elements, resource requirements of both methods are very close. On the other hand, embedded multiplier requirement of the first method (modular approach) is double of the second, owing to the fact that two instances of the generic filter module is required to implement two filters (i.e. it is not possible to use same instance twice for different filters), since past values need to be stored. On the other hand, softcore processor handles the design with 4 embedded multipliers, which is an advantage on its part.

In terms of total clock cycles, method I has a significant advantage over method II, due to the optimizable modular design approach, as expected. It is not possible that this advantage arises due to the increased number of embedded multipliers that are employed, because even if the total clock cycles required by the first method is doubled, it is still significantly faster than the embedded processor. The total clock cycles required can be determined based on the number of multiplications in the filter. In this case, since 4 multiplications are required, total clock cycles required is obtained by multiplying the clock cycles needed by the custom multiplication module by 4.

Observing the simulation results and the resource/time properties of two methods, it can be seen that both methods are convenient options for digital filter implementations on FPGAs. It should be noted that, when the complexity of the design increases (i.e. the number of filters and etc.), the significant increase will be in terms of resource requirements for Method I, since new instances of the generic filter module will be required; on the other hand, for Method II, the significant increase will be in terms of required clock cycles, since the processor is already implemented and the change will only be in the algorithm. Therefore, it can be concluded that Method I -modular design approach- seems more suitable for demanding real time designs where speed is more crucial and Method II -softcore processor- seems more suitable if the design is very complex or a fast and efficient development period is desired.

5.5 Closure

In this chapter, different methodologies for digital filter implementation on FPGAs are presented. These methods are utilized by realizing a cascaded control system to solve a nonlinear inverted pendulum control problem. Furthermore, resource utilization and speed performances are also provided for these methodologies.

CHAPTER 6

ADVANCED CONTROLLERS

This chapter presents a study for some advanced real-time motion controller topologies implemented on the field programmable gate array (FPGA); which are selected as a sliding mode controller and a fuzzy controller. In this chapter, all the aforementioned methods are not implemented and a single implementation method (Method I-c) presented in Chapter 4 is adopted. In the context of Method I-c, controllers are developed using Verilog HDL (i.e. modular hardwired approach is adopted) and an open-core hardwired floating point unit is implemented for the complex calculations.

Unlike other chapters of this study, in this chapter the controllers are implemented on a ML505 development board with a Xilinx Virtex-5 FPGA chip (in other chapters a DE1 board with an Altera Cyclone II FPGA is employed). The tests are performed on a hardware-in-the-loop simulation of a field-oriented induction motor system, which is a similar system to the case provided in Chapter 7, however in this case speed control problem of a CNC turning center is considered.

Finally, simulation results are provided for the test case and implementation results are provided. The controller topologies are evaluated by certain criteria including resource cost (total memory space and logic unit requirement), attainable sampling rate and the success of the controller.

6.1 Controller Topologies

Controller topologies are selected from common advanced controller topologies, considering the suitability of the topology for FPGA implementation.

6.1.1 Fuzzy Controller

Fuzzy control is an intelligent control topology based on fuzzy set theory. It has been applied to many control applications including the control of drives [26]-[27]. Typical method for fuzzy control application in discrete-time control is to calculate error and change in error in each sampling time, then to define a linguistic representation of the error and change in error based on membership functions. As a final step, these linguistic representations having fuzzy memberships go through a defuzzification process to generate a manipulation signal based on a fuzzy rule base. A schematic for the implementation of the fuzzy controller on the FPGA is presented in Figure 6.1.

Membership functions are presented in Figure 6.2. Here, NB, NS, Z, PB, and PS correspond to “Negative big,” “Negative small,” “Zero,” “Positive big,” and “Positive small” respectively. Notice that the functions are formed by the pulse error per sampling time period. Therefore, the data from the encoder can be directly used without further calculations while the fuzzy memberships can be represented as integers.

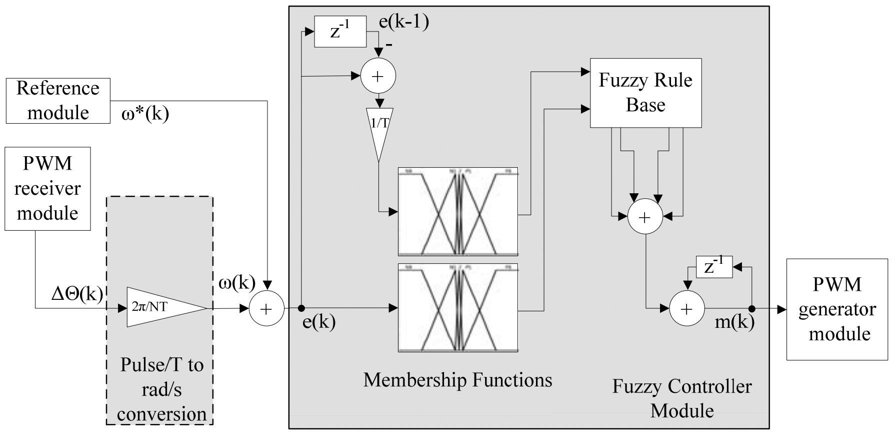


Figure 6.1 – Fuzzy controller implementation

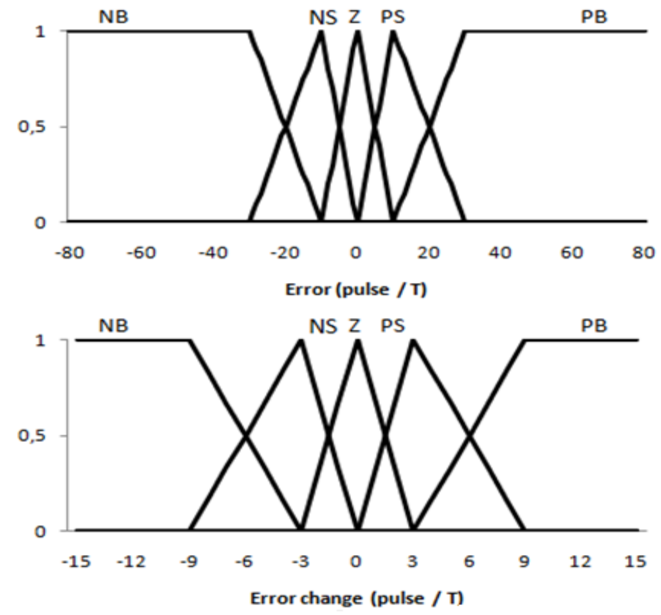


Figure 6.2 – Membership functions of the fuzzy controller

The defuzzification process depends on the fuzzy rule base presented on Table 6.1. It can be observed that the membership functions are formed based on pulse error per sampling time period. Therefore, the data from the encoder can be directly used without further calculations and the fuzzy memberships can be represented in integer forms. The defuzzification process is based on the fuzzy rule base presented on Table 6.1. Fuzzy membership functions and the rule base are selected based on the common experience in the literature and after a trial and error process.

Table 6.1 – Membership functions of the fuzzy controller

error / Δ error	NB	NS	Z	PS	PB
NB	NB	NB	NS	NS	Z
NS	NB	NS	NS	Z	PS
Z	NS	NS	Z	PS	PS
PS	NS	Z	PS	PS	PB
PB	Z	PS	PS	PB	PB

6.1.2 Sliding Mode Controller

Sliding mode control is a robust control method developed to deal with model uncertainties and unknown parameters in the expense of high computational cost [28]. Its applications include position/speed control of servo/induction motor drives [29]-[30]. Sliding mode controller is based on a control law with varying control structures. The basic idea is to force the trajectory of the system state to a sliding surface through switching of the control structures. The most general form of the sliding surface is

$$s = \dot{e} + \lambda e \tag{6.1}$$

where s is the sliding surface; e is the error of the controlled state and λ is a controller parameter. After calculating the sliding surface, an equivalent control term needs to be calculated such that applying equivalent control would make $\dot{s} = 0$. Hence, the sliding mode control law takes the form:

$$u = u_{eq} + Ksgn(s) \tag{6.2}$$

where u_{eq} is the equivalent control and u is control output. The implementation of the sliding mode controller is presented in Figure 6.3

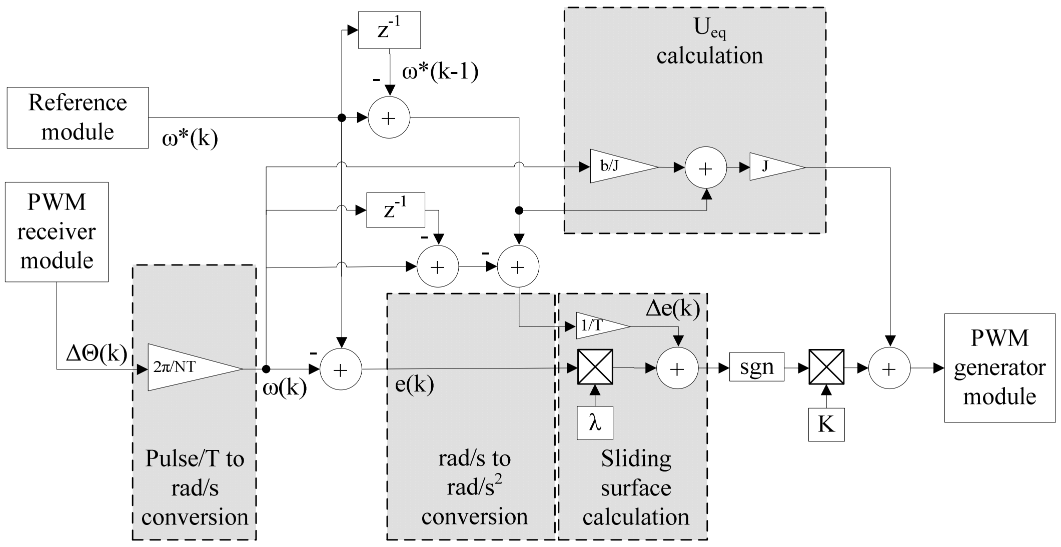


Figure 6.3 – Sliding mode controller implementation

The sliding mode controller parameters are selected as λ = 10000 and K = 15. These parameters are selected based on the disturbance rejection characteristics of the controller and the maximum admissible torque based on the rated torque of the motor.

6.2 Test Setup

The control of CNC turning centers is considered for performance evaluation of the above mentioned controller topologies. In CNC turning centers, the spindle housing the workpiece is the key component of the machine where a constant speed is required during most machining operations. Therefore, to sustain constant speed, the controller must effectively reject the disturbance torque observed in turning operations. Figure 6.4 illustrates the simplified model of a typical turning center. In this system, a field-oriented induction motor, which is further elaborated in [31], is employed. Induction motor parameters are as follows: motor power: 5.5kW; rated motor torque (T_r): 35Nm; rated speed (ω_r): 1500rpm; maximum speed (ω_p): 8000 rpm. It is critical to note that the presented system is realized via a hardware-in-the-loop simulation (HILS) performed on the Altera DE1 FPGA development board. For the sake of implementation, certain simplifications (ideal DTC motor drive, ideal timing belt) are made on the system. Figure 6.5 shows the block diagram of the resulting system.

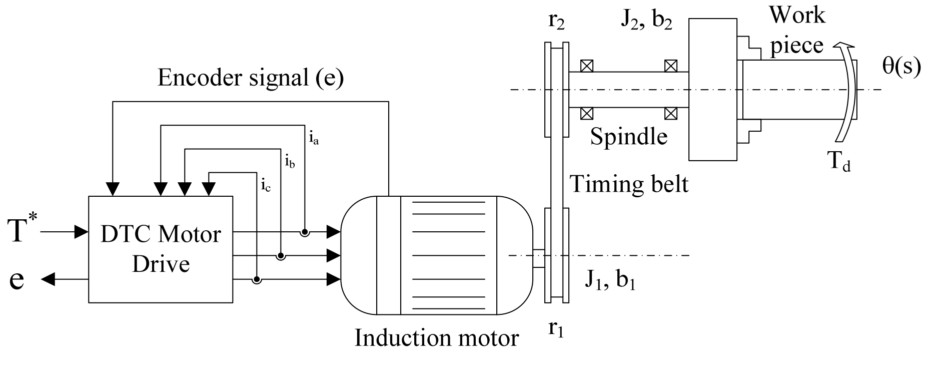


Figure 6.4 – Simplified model of a typical turning center

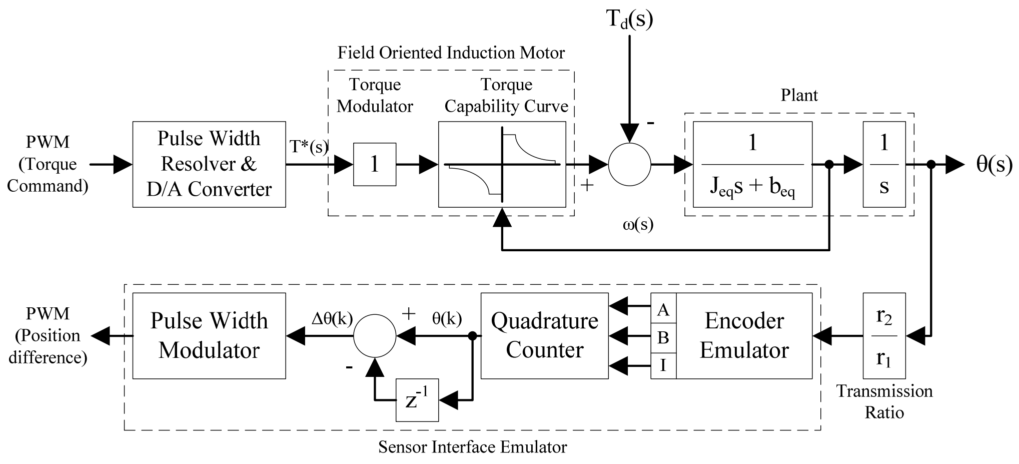
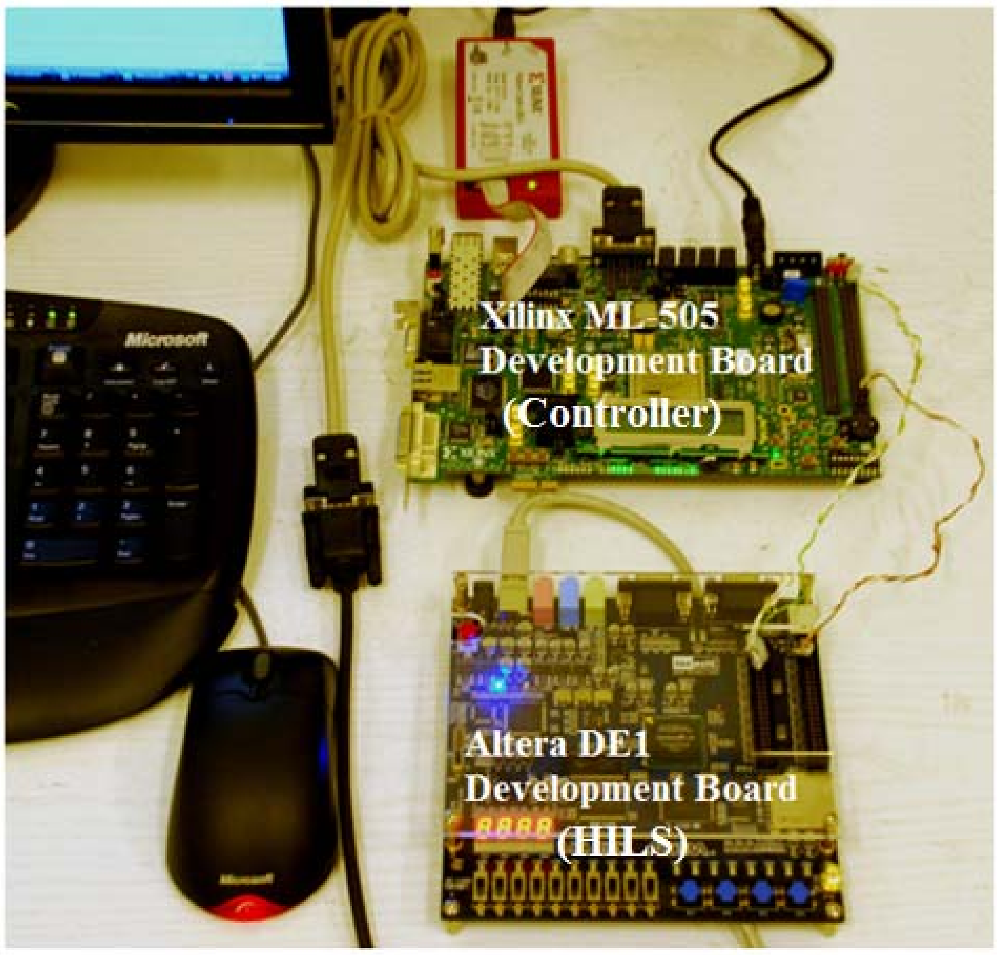


Figure 6.5 – Simplified model of the system



Xilinx ML-505 Development Board (Controller)
Altera DE1 Development Board (HILS)

Figure 6.6 – Test setup

As can be seen in Figure 6.5, the communication with the simulator is accomplished via PWM signals. A 10-bit resolution is selected for the PWM signal to represent torque command and position feedback. In order to prevent problems that could arise from insufficient resolution of the PWM signal, the position difference is transmitted from the HILS setup. Note that the real setup, which is shown in Figure 6.6, consists of two FPGA development boards. One of them is the Xilinx ML-505 development board on which the controller modules are implemented. Likewise, an Altera DE1 development board performs the HILS. Boards are connected through I/O pins, facilitating the PWM connection.

6.3 Results

In this study, different controller topologies are designed and implemented on an FPGA chip. In this section, the results of the HILS are presented and controller topologies are evaluated by their success (tracking performance, disturbance rejection) and their resource cost on an FPGA chip.

The reference input to the system is given as position difference between time samples and a constant 10 Nm torque is applied at t = 1s to simulate the interrupted machining operation. Responses of the system with different controllers are presented in Figure 6.7 and Figure 6.8 highlights the disturbance rejection characteristics of each controller. As can be observed, the controllers achieve similar performances until the disturbance kicks in at 1s. From this point on, the best performance is achieved by the sliding mode controller and while the system under fuzzy control is affected slightly, the controller acts fast enough to compensate the disturbance. Note that from the standpoint of performance, these controllers are comparable.

Table 6.2 – Resource costs of different controller topologies on the Xilinx Virtex-5 FGPA

	Slice LUTs	Slice Registers
Fuzzy	12863 (44%)	2448 (8%)
Sliding mode	4192 (14%)	1675 (5%)

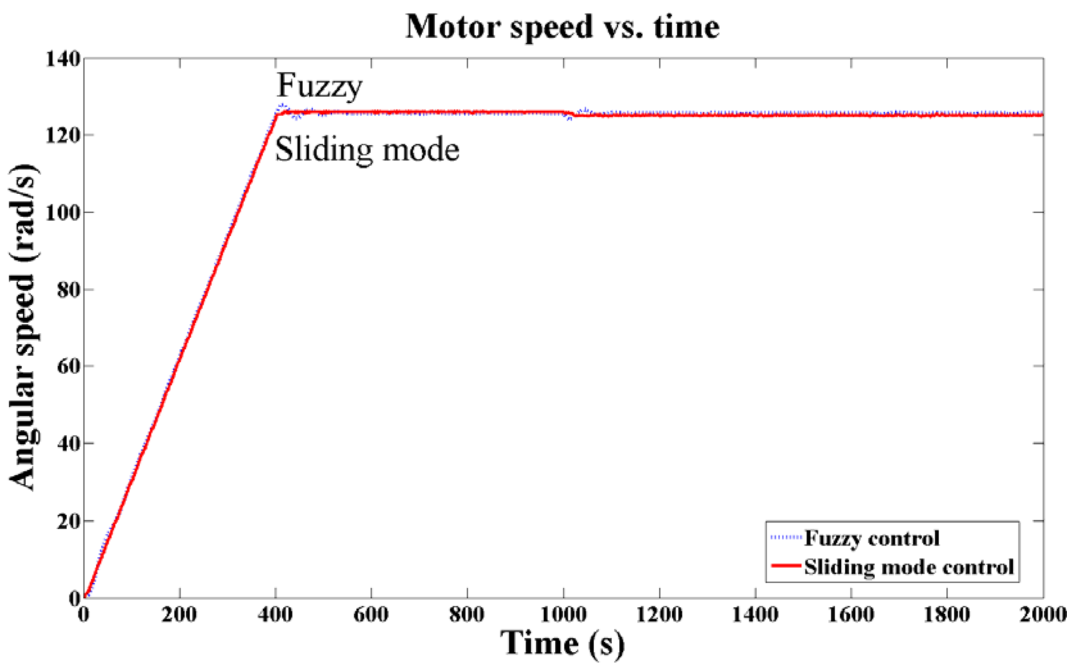


Figure 6.7 – Controller performances under disturbance input

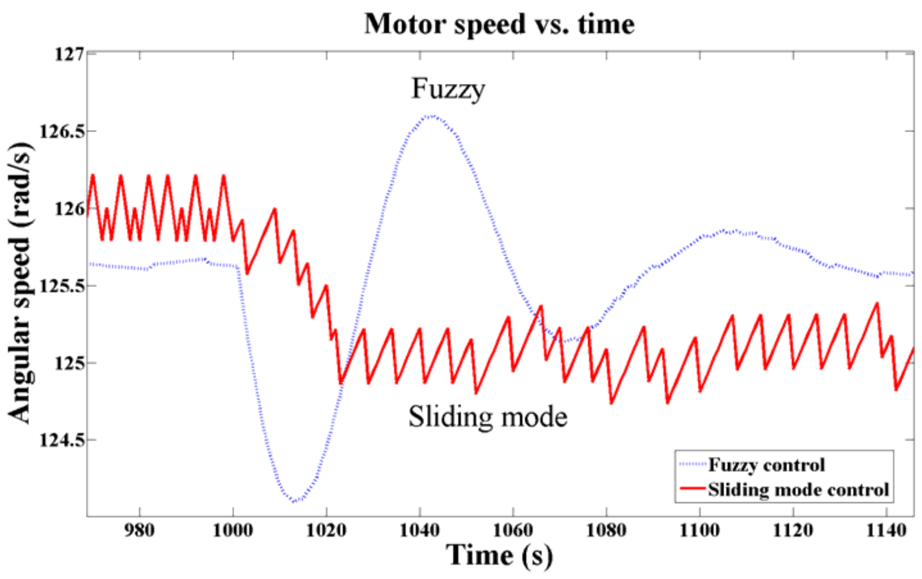


Figure 6.8 – Zoomed controller performances to highlight disturbance rejection

On the other hand, resource costs of the controllers vary in a wide range. These requirements are given in Table 6.2 in terms of slice look up tables (LUTs) and slice registers followed by the percentage consumed out of the available amount on the FPGA chip. At this point, it is critical to note that the resource requirements provided here are **not** in the same terms with the resource requirements provided in the previous chapters for Altera FPGAs, since Xilinx uses a different terminology for different amount of resources.

As can be seen from Table 6.2, the sliding mode controller is consuming around 5% of the registers and 15% of the LUTs available in the FPGA chip. If a multiple-axis solution is sought on the chip, these numbers are sufficient enough for driving 5-axis simultaneously. On the other hand, fuzzy controller requires high amount of resources as Slice LUTs and therefore does not seem applicable for multi-axis solutions.The complexity in the design affects not only the resources but also the attainable sampling period of the controller. In Table 6.3, maximum attainable sampling rates by controllers are presented. As seen in Table 6.3, the controllers can perform the required calculations around 85 cycles, which corresponds to 1.7 μs on an FPGA with a 50 MHz clock.

Table 6.3 – Minimum attainable sampling periods of controllers on the Xilinx Virtex-5 FGPA

Controller Type	Cycles to complete loop	Minimum period
Fuzzy	87 cycles	1.74 μs
SMC	84 cycles	1.68 μs

On the implementation end, sliding mode controller may be implemented with a sufficiently high effort. However, fuzzy controller is complicated to implement on an FPGA chip, especially when an embedded processor is not employed in the design, as in this case.

6.4 Closure

In this chapter, two advanced controller topologies are implemented on a Virtex5 FPGA. In order to take full advantage of the parallel processing capability of the FPGA, a softcore processor (explained as Method II in Chapter 4) is not employed; however as a preliminary study, only Method I-c is utilized for computation. In the results part, control algorithms are compared by their success and their resource cost on an FPGA chip. Results are discussed and important features are highlighted.

CHAPTER 7

HARDWARE-IN-THE-LOOP SIMULATION & RESULTS

The aim of this chapter is to demonstrate the utilization of the modules and methods that are developed in this thesis. Therefore, aforementioned pieces of a motion controller are put together to solve a real-world motion control problem.

As a matter of fact, most of the modules and design methods are investigated *individually* via different testing methods presented at each chapter and they are proven to be successful at their own right. However, it is critical to test the developed methods as a single motion controller design and demonstrate that the proposed design is effectively capable of dealing with real-world motion control problems. Furthermore, testing the overall system would also demonstrate the capabilities and success of the overall design paradigm, which would provide more meaningful results than individual experiments.

As a test case, a CNC machining center is selected. This selection is convenient since CNC machinery is one of the fundamental application areas of motion control systems. Furthermore, it also requires a multi-axis controller; which is also suitable for demonstrating the parallel processing capabilities of the FPGA based design.

The outline of this chapter is as follows: in the first section, the mathematical model of the CNC machining center selected as the test case is presented thoroughly. In the second section, the axis-controllers for this system are designed and the MATLAB/Simulink simulation results are provided. In the third section, the realization details of this problem via hardware-in-the-loop simulation (HILS) are briefly presented. In the fourth section, the HILS results are shown while some key results conclusions on the implementation are discussed in the last section.

7.1 Real system

The real system selected as the test case is a CNC machining center, which is a very important application area of a motion control system; since the success of the controller directly affects the productivity of the machine, as well as the quality of the product. Therefore, a CNC machining center is a suitable candidate for testing the motion controller design. Thus, this section starts with the details of the selected CNC center.

7.1.1 CNC machining center

The selected CNC vertical machining center shown in Figure 7.1 is located at the Machine Shop of the Mechanical Engineering Department of METU. It is a First MCV-1100 3-Axis CNC Machining center by Long Chang Machinery, equipped with an automatic tool changer, coolant and chip removal systems.

Figure 7.1 – First MCV-1100 3-Axis CNC Machining center

The axes of the machine are all mounted on friction (hydrostatic) guideways, and are driven by servomotors via ball screws. The *x*-axis carrying the cart (a.k.a. "table") on which the workpiece is mounted is illustrated in Figure 7.2 and the *y*-axis (a.k.a. "saddle") carries the entire *x*-axis assembly. On the other hand, Z-axis assembly is housed on the column and carries the entire headstock (main spindle shaft, motor, tool changing mechanism) as shown in Figure 7.3.

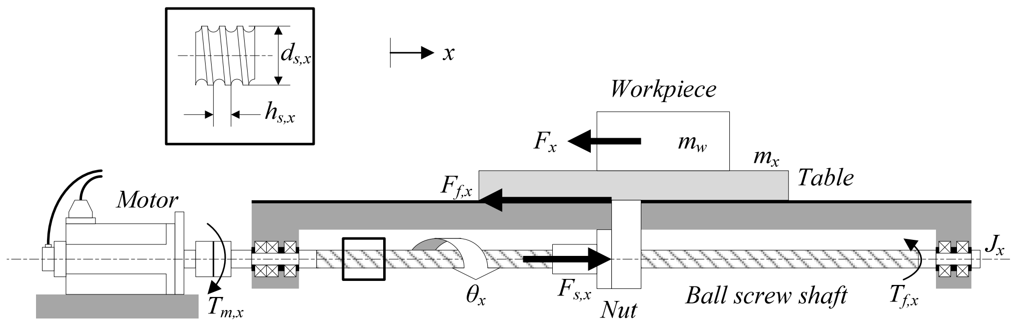


Figure 7.2 – X-axis feed drive for CNC machining center

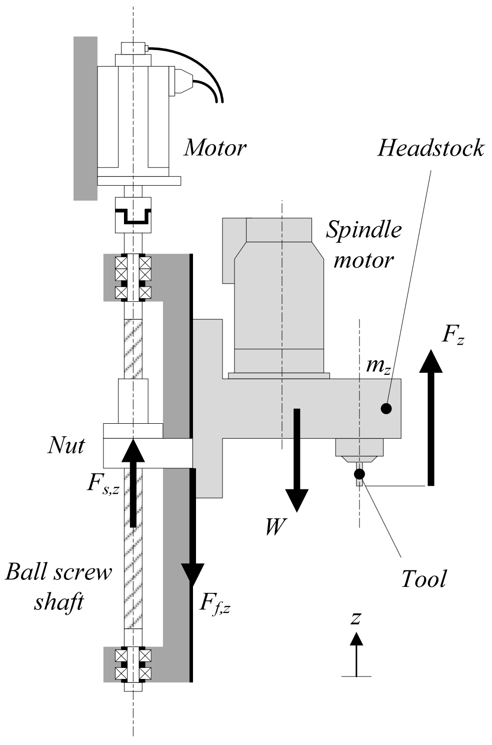


Figure 7.3 – Z-axis feed drive for CNC machining center

With the given information, the equation of motion for the x-axis cart can be written as

$$\ddot{x} = \left(\frac{1}{m_x + m_w} \right) \left(F_{s,x} - F_x - F_{f,x} \operatorname{sgn}(\dot{x}) \right) \tag{7.1}$$

where m_w stands for the mass of the workpiece, m_x is the mass of the cart, F_x is the cutting force on the axis, $F_{f,x}$ is the friction force (dry) and $F_{s,x}$ is the force exerted on the table by the ball screw nut. The equation of motion for the dynamic system (as reduced to the motor shaft) becomes

$$\left(\frac{1}{J_x} \right) \left(b_x \left| \dot{\theta}_x \right|^{\frac{2}{3}} \right) \operatorname{sgn}\left(\dot{\theta}_x \right) + \ddot{\theta}_x = \left(\frac{1}{J_x} \right) \left(T_{m,x} - \frac{h_{s,x}}{2\pi \eta_{s,x}} F_{s,x} + T_{0,x} \right) \tag{7.2}$$

where J_x is the total moment of inertia of the ball screw and rotor, $T_{m,x}$ is the torque applied by the motor, $T_{f,x}$ is the total (dry and viscous) friction torque on the ball screw and rotor, $h_{s,x}$ is the pitch of the ball-screw shaft and $\eta_{s,x}$ is the ball screw efficiency. When backlash exists in the ball-screw assembly (which is a rare situation in precision parts), equations (7.1) and (7.2) are coupled together with (7.3) as

$$F_{s,x} = \begin{cases} k_x\left(d_x - \frac{b_x}{2}\right) & , d_x > \frac{b_x}{2} \\ 0 & , \left|d_x\right| < \frac{b_x}{2} \\ k_x\left(d_x + \frac{b_x}{2}\right) & , d_x < -\frac{b_x}{2} \end{cases} \tag{7.3}$$

where

$$d_x = \frac{h}{2\pi}\theta_x - x_x \tag{7.4}$$

If the ball screw is assumed to be backlash-free, these equations can be reduced to a single equation of motion that uses an equivalent set of parameters. That is, using (7.1) and (7.2) yields

$$\ddot{\theta}_x = \left(\frac{1}{J_{eq,x}}\right)\left[T_{m,x} - \frac{h_{s,x}}{2\pi\eta_{s,x}}F_x - \left(b_x\left|\dot{\theta}_x\right|^{\frac{2}{3}} + T_{0,eq,x}\right)\mathrm{sgn}\left(\dot{\theta}_x\right)\right] \tag{7.5}$$

Here, the equivalent inertia $J_{eq,x}$ is defined as

$$J_{eq,x} = J_x + \frac{h_s^{\ 2}}{4\pi^2\eta_s}\left(m_x + m_w\right) \tag{7.6}$$

Since the table's position is linearly dependent on the angular position of the ball screw under no-backlash condition, it immediately follows that the velocities are also linearly dependent and $\mathrm{sgn}(\dot{x}) = \mathrm{sgn}(\dot{\theta})$ holds. Hence, utilizing equations (7.1) and (7.2), the equivalent dry friction $T_{f,eq,x}$ can be simply written as

$$T_{0,eq,x} = \frac{h_{s,x}}{2\pi\eta_{s,x}} F_{f,x} + T_{0,x} \tag{7.7}$$

The equations of motion regarding the *y*- and *z*-axes can be similarly obtained as

$$\ddot{y} = \left(\frac{1}{m_y + m_x + m_w} \right) \left[F_{s,y} - F_y - F_{f,y}\,\mathrm{sgn}(\dot{y}) \right] \tag{7.8}$$

$$\ddot{\theta}_y = \left(\frac{1}{J_y} \right) \left\{ T_{m,y} - \frac{h_{s,y}}{2\pi\eta_{s,y}} F_{s,y} - \left[b_y \left| \dot{\theta}_y \right|^{\frac{2}{3}} + T_{0,y} \right] \mathrm{sgn}\left(\dot{\theta}_y \right) \right\} \tag{7.9}$$

$$\ddot{z} = \left(\frac{1}{m_z} \right) \left[F_{s,z} + F_z - F_{f,z}\,\mathrm{sgn}(\dot{z}) - W \right] \tag{7.10}$$

$$\ddot{\theta}_z = \left(\frac{1}{J_z} \right) \left[T_{m,z} - \frac{h_{s,z}}{2\pi\eta_{s,z}} F_{s,z} - \left(b_z \left| \dot{\theta}_z \right|^{\frac{2}{3}} + T_{0,z} \right) \mathrm{sgn}\left(\dot{\theta}_z \right) \right] \tag{7.11}$$

Under no-backlash condition, these can be expressed in a simpler form similar to (7.5) as

$$\ddot{\theta}_y = \left(\frac{1}{J_{eq,y}}\right)\left\{T_{m,y} - \frac{h_{s,y}}{2\pi\eta_{s,y}}F_y - \left[b_y\left|\dot{\theta}_y\right|^{\frac{2}{3}} + T_{0,eq,y}\right]\operatorname{sgn}\left(\dot{\theta}_y\right)\right\} \tag{7.12}$$

$$\ddot{\theta}_z = \left(\frac{1}{J_{eq,z}}\right)\left\{T_{m,z} + \frac{h_{s,z}}{2\pi\eta_{s,z}}\left(F_z - W\right) - \left[b_z\left|\dot{\theta}_z\right|^{\frac{2}{3}} + T_{0,eq,z}\right]\operatorname{sgn}\left(\dot{\theta}_z\right)\right\} \tag{7.13}$$

Note that the weight of the headstock assembly (*W*) is included to the model of the *z*-axis drive. It is critical to notice that the feed-drive axes are driven by Fanuc α Series AC Servo Motors while the spindle motor is a Fanuc α Series AC Spindle (Induction) Motor. As specified in the user manuals, the speed-torque characteristics of the servo motors have a linearly decreasing tendency in the torque region up to the rated speed [1]. Beyond this point, the motor enters the constant power region as shown in Figure 7.4.

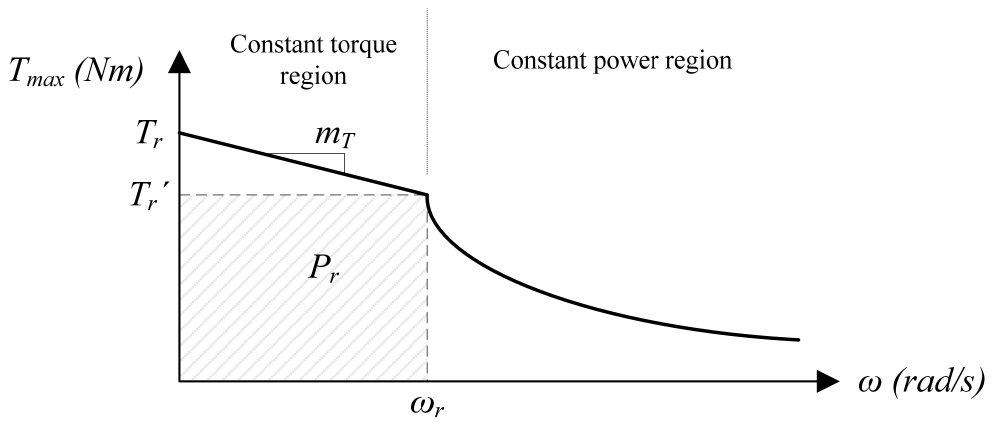


Figure 7.4 – Torque capability curve for CNC machining center axis motors

The torque envelope of the motor, T_{max}, (See Fig. 7.4) is then as follows

$$T_{max} = \begin{cases} T_r + m_T\left|\omega\right| & ,\left|\omega\right| < \omega_r \\ \dfrac{P_r}{\left|\omega\right|} & ,\left|\omega\right| \geq \omega_r \end{cases} \tag{7.14}$$

where T_r and ω_r represent the rated torque and rated speed, respectively. T_r' is the torque produced by the motor and P_r is the power output, both at the rated speed while $m_T = (T_r - T_r') / \omega_r$ and $P_r = T_r'\omega_r$. The numerical values for the parameters defining the plant are provided in Table 7.1.

Finally, it should be noted that the motor position data is obtained from the axes via an incremental encoder that generates 10000 ppr which would yield a resolution of 40000 pulses per revolution with quadrature (4X) decoding.

7.1.2 MATLAB/Simulink model

The system's governing equations in terms of equivalent torque and inertia are provided in the previous section in (7.5), (7.12), and (7.13) for x, y and z axes respectively. Using these equations and the system parameters provided in Table 7.1; it is possible to develop a Simulink model for the overall system. Figure 7.5 shows the dynamic model of a single axis (x-axis) of the system developed by the MATLAB/Simulink package.

Table 7.1 – MATLAB/Simulink model of a single axis of the CNC machining center

Parameter	**Sym.**	**Unit**	***X***	***Y***	***Z***
Mass	m	kg	130	331.97	260
Dry friction force	F_f	N	200	200	200
Moment of inertia	J	kg m^2	7.994×10^{-3}	16.484×10^{-3}	19.745×10^{-3}
Dry friction torque	T_f	N	1.1	1.5	2.1
Viscous friction coefficient	B	Nms/rad	0.0005	0.0005	0.0005
Equivalent moment of inertia	J_{eq}	kg m^2	0.00834	0.01737	0.02044
Equivalent dry friction	$T_{f,eq}$	N m	1.435	1.835	2.435
Ball screw lead	h_s	m	0.010	0.010	0.010
Ball screw efficiency	η_s	-	0.95	0.95	0.95
Rated torque	T_r	N m	12	22	30
Rated speed	ω_r	rad/s	209.44	209.44	209.44
Rated power	P_r	W	2094.4	3769.9	4398.2
Torque-speed slope	m_T	Nms/rad	-0.00955	-0.01910	-0.04297

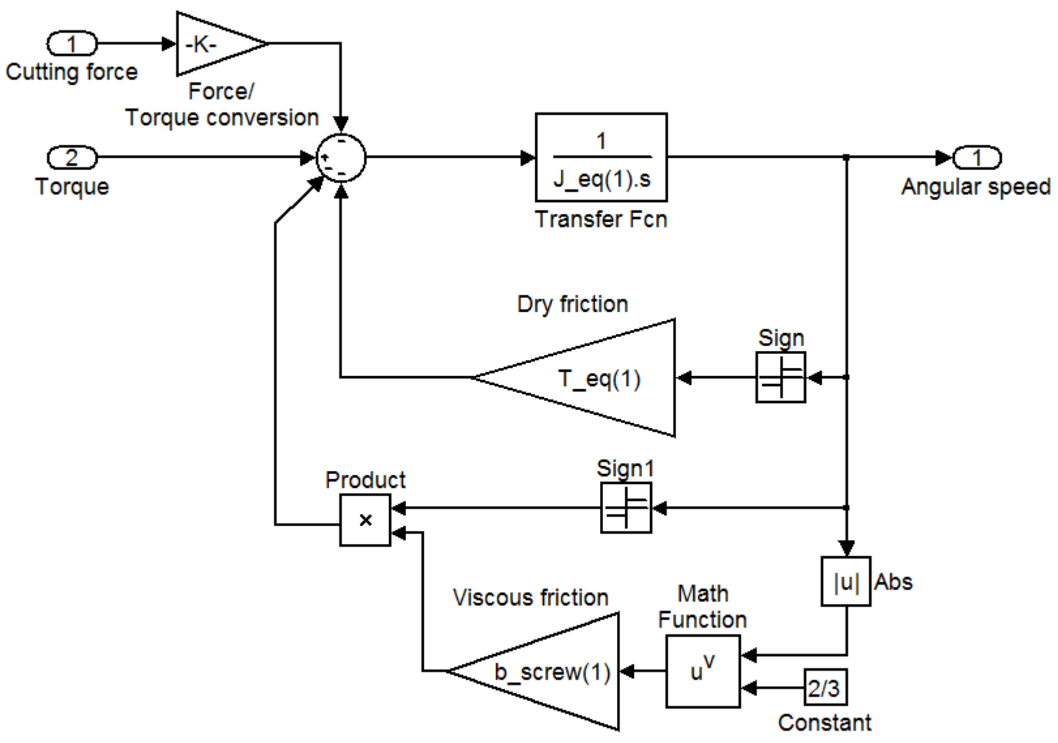


Figure 7.5 – MATLAB/Simulink model of a single axis of the CNC machining center

As can be seen in Figure 7.5, the Simulink model has a transfer function that relates the input torque to the angular speed of the motor shaft. The model includes two different friction models (dry and viscous) and the disturbance input that are exactly represented in the governing equations of the system. This model is employed in all of the three axes of the machining center, with a change in the parameters: equivalent inertia (J_eq), equivalent dry friction coefficient (T_eq), viscous friction coefficient of screw (b_screw), screw efficiency (eff_scr) and pitch of the screw (h). Note that these parameters can easily be changed by changing the index of the parameter (i.e. in J_eq(1) index 1 represents the parameter for the x axis).

While testing/debugging the designed controller, the model in Figure 7.5 is utilized as the controlled plant in the MATLAB/Simulink simulations. The next section explains the details of the controller design.

7.2 Controller design

In section 0, the CNC machining center that is considered as the test case of the motion controller design is introduced and its governing equations, Simulink model, and the relevant system parameters are provided. In this section, a controller is developed based on the introduced model and the simulation results obtained in MATLAB/Simulink are presented.

7.2.1 Controller selection

As discussed in Chapter 2, there are many motion controller topologies that can be implemented on an FPGA, including both conventional and novel/intelligent controllers. However, in order to demonstrate the presented methods in the previous chapters, the choice for the test case controller is to be made between the state-space controller (presented in Chapter 4) and the filter implementation (discussed in Chapter 5). Even though both of these controllers are equally applicable, the filter implementation seems a more convenient choice since the classical SISO controller topologies (like the industry-standard PID) can be easily realized. Furthermore, it is a more efficient method (in terms of expended resources) on FPGA: As demonstrated in Chapters 4 and 5, two instances of the filter can be realized with 27% of the logic cells (i.e. a single instance consumes 13%) while a state-space controller can realized utilizing 24% of the logic cells (via integer multiplication method). Therefore, filter implementation is selected to build the controller topology for the test case and thus its FPGA implementation is carried out by the method presented in Chapter 5.

7.2.2 Linearized system model

Considering the system model provided in Figure 7.5, the transfer function (G_{pw}) between the input torque and the output angular speed is G_{pw} (s) = 1 / (J_{eq} s) when the nonlinear terms (friction) are discarded under the assumption that they could be conveniently visualized as a part of the disturbance. Since a position control is desired, the output of the transfer function G_{pw} needs to be integrated, which would lead to the transfer function between the input torque (manipulation) and the output position of the motor shaft, that is $G_p = 1 / (J_{eq}\ s^2)$. This continuous time transfer function of the plant is considered as the system model for the controller design via root locus method.

Note that since the position feedback is obtained from an encoder, there is an encoder gain of 40000/(2πh) in the feedback loop. Therefore, the reference trajectory may also obtained in terms of encoder pulses (hence integers) as previously discussed in Chapter 4, while commenting on the advantageous attributes of digital motion control systems. Therefore, the controller is designed according to an encoder gain applied to both the reference and the position feedback.

As a final mark, it is important to note that there exist 3 different axes to be controlled and hence 3 different plant models are obtained. However, the following procedure is presented for a single axis (x), since the plants and the design scheme are very similar for each axis.

7.2.3 Design via root locus technique

After the system model is obtained, the controller may be designed in either continuous-time domain or discrete-time domain. While both approaches are equally acceptable, in this study the latter approach is adopted and therefore as a first step, the system model needs to be converted to an equivalent discrete-time domain (i.e. z-domain) representation.

As an initial step for discretization of the system, a convenient sampling time (t_s) should be selected. In modern real-time control systems, 1 kHz sampling frequency is a highly sufficient for even demanding control applications, thus $t_s = 0.001s$ is selected for the sampling period of the controller. After the selection of t_s, the system is discretized using zero-order hold method to obtain the discrete time model of the system $G_P(z)$ via *"c2d"* function of MATLAB.

Once the discrete-time transfer function is obtained for the plant, the root-locus design is performed via *sisotool,* which is a convenient tool provided by MATLAB for root locus design method. The interface allows the user to select closed-loop pole locations by adjusting the controller gain. That is, the user can modify the open-loop poles and zeros via an interactive root locus plot. Using *sisotool,* the root locus plot of the uncontrolled system is provided in Figure 7.6. Notice that the system is unstable since the closed-loop poles are outside the unit circle and it is not possible to stabilize the system by a simple proportional controller since one of the poles end up at infinity as gain increases. Therefore, another controller needs to be designed to obtain a stable system that yields desirable tracking- and disturbance rejection performances.

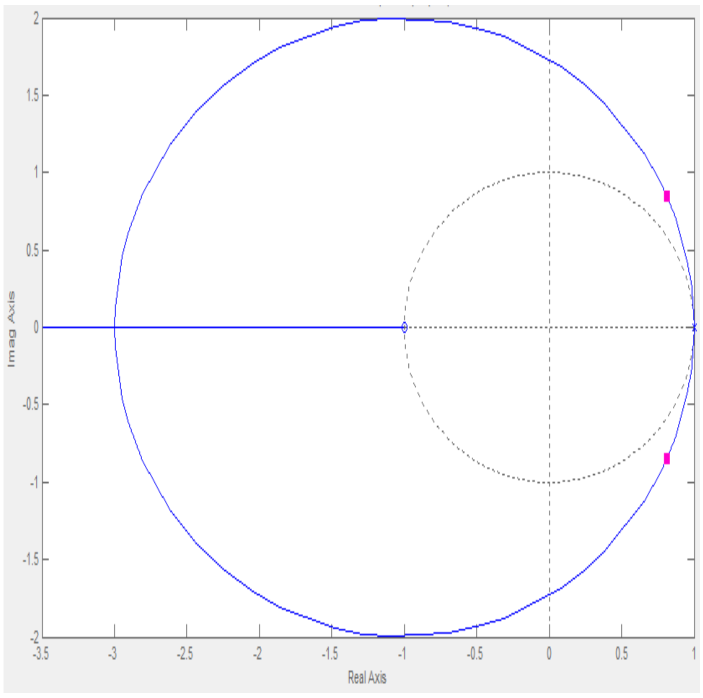


Figure 7.6 – Root locus plot of the uncompensated system

As an initial step, a zero is added at 0.8 on the real axis in order to stabilize the system. After the addition of the zero, the speed of the system is further increased by placing a pole at 0.1. Hence the root locus plot has become as shown in Figure 7.7.

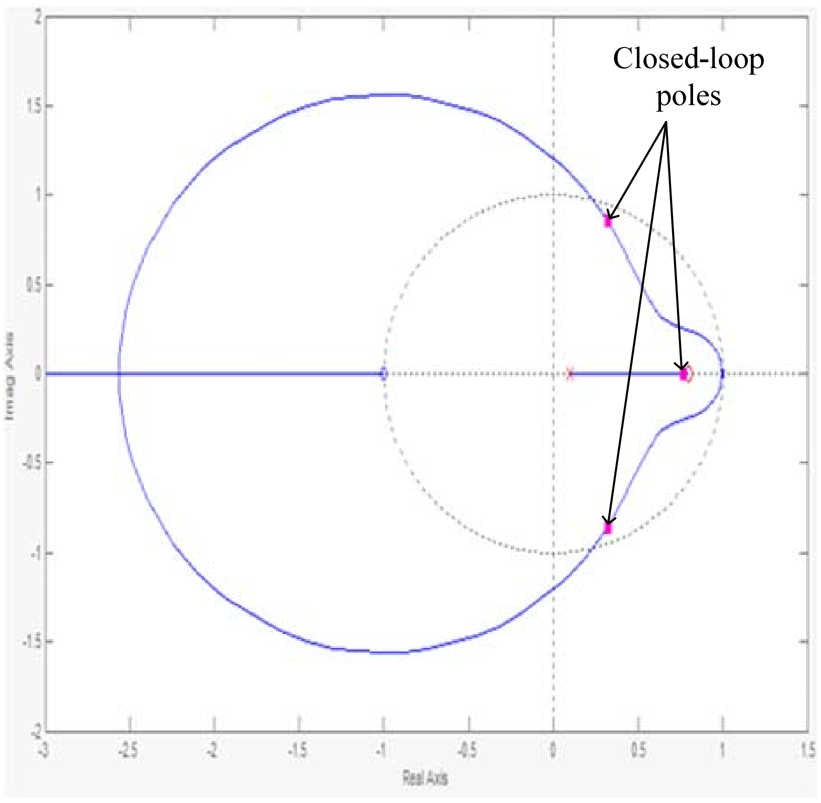


Figure 7.7 – Root locus plot of the system after addition of a pole and a zero

As can be seen in Figure 7.7, the system may be still unstable since two of the closed-loop poles could be outside the unit circle; however after the addition of the pole/zero, now it is possible to stabilize the system by simple gain adjustment. Note that the initial gain is selected as unity by default in *sisotool*, therefore by decreasing the gain, the system can be stabilized.

By decreasing the gain gradually, it can be seen that the closed-loop poles cross the unit circle when gain (K) = 0.492 and therefore values below this value should yield a stable system response. Considering the actual plant, it is known that during machining process, cutting forces act on the system as a disturbance. Therefore, a sufficiently high K value is desired for increasing the system's dynamic stiffness (to disturbances). Hence, K = 0.4 is chosen conveniently for the gain value of the controller. The resulting closed-loop poles are also shown in Figure 7.7

On the other hand, it is also necessary to check the closed-loop bode plot, in order to obtain the bandwidth frequency of the controlled system (which is also provided in the *sisotool* interface). Bode plot of the closed-loop system is shown in Figure 7.8. As can be seen in Figure 7.8, the bandwidth frequency of the system is around 300 Hz (< half the sampling frequency = 500 Hz), which is deemed sufficient for the most CNC vertical machining centers.

Proceeding with the designed controller, its discrete-time domain expression can be obtained as follows:

$$G_c(z) = \frac{2z-1.6}{1.1z-0.1} \tag{7.15}$$

Using (7.15) the developed controller may be tested via Simulink to observe the performance of the controller with the desired trajectory of the x axis.

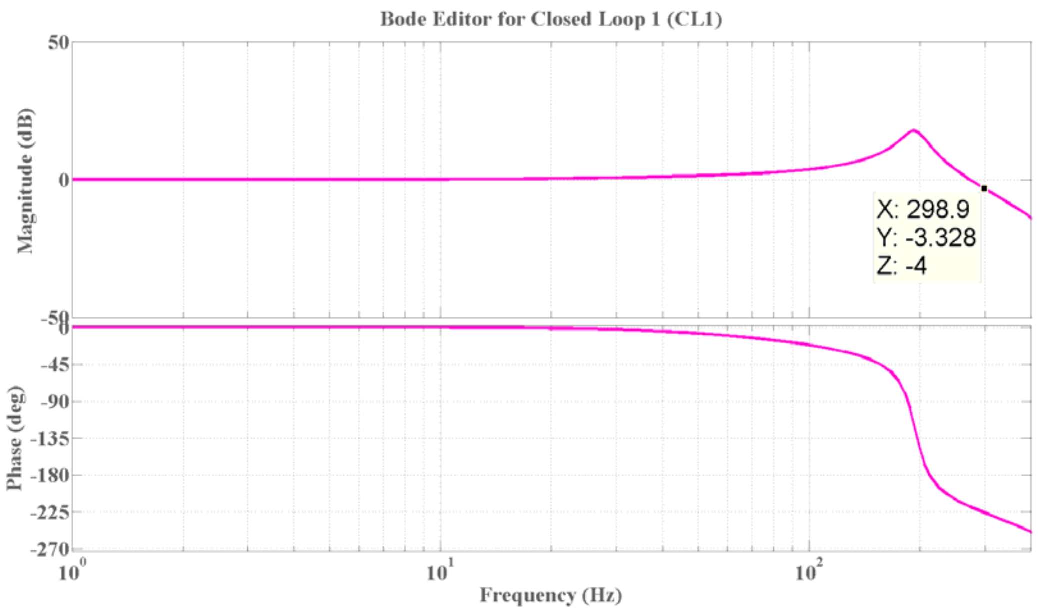


Figure 7.8 – Closed-loop Bode plot of the system

It is important to note that while this design method is explained for the x axis of the CNC machining center, the design method for the other axes are very similar to the x-axis; since the same plant with slightly different parameters are considered. As a matter of fact, the same controller is applied to the all of the axes with a slight change in the controller gains for the y and z axes. The reason behind this choice is that, the root locus plots of the other axes allow more increase in the controller gain, while keeping the closed-loop poles within the unit circle. Therefore, the gain values for y and z axes are selected as 0.6, while the other controller parameters are essentially the same.

Table 7.2 – Controller design parameters for x, y and z axes

Axis	Gain	Closed-loop poles	Bandwidth frequency [Hz]
X	0.4	0.771, 0.321±0.86i	296
Y	0.6	0.755, 0.425±0.69i	257
Z	0.6	0.741, 0.469±0.61i	236

The next section presents the Simulink results of the designed controller.

7.2.4 Simulink simulation results

In Figure 7.9, the overall system model in Simulink is shown. As can be seen, the overall system incorporates the following systems: **i)** the feed-drive (axis) model shown in Figure 7.5 (shown as x-Axis), **ii)** the controller (the lead-lag compensator), **iii)** the feedforward dry friction compensator, **iv)** the torque generation model for the motor. Note that an encoder gain is placed behind the scope to convert the output position signal from radians to encoder counts. To simulate the real encoder behavior, a floor function is placed after the encoder gain. Since the axis model provides the output in terms of angular speed, an integrator is added after the axis model to obtain the angular position. The inputs, which are previously defined reference trajectory and cutting force (disturbance), are obtained from the Matlab workspace. Notice that the provided model represents the x-axis of the machining center; however the model can be applied to all axes with proper changes in the parameter indices.

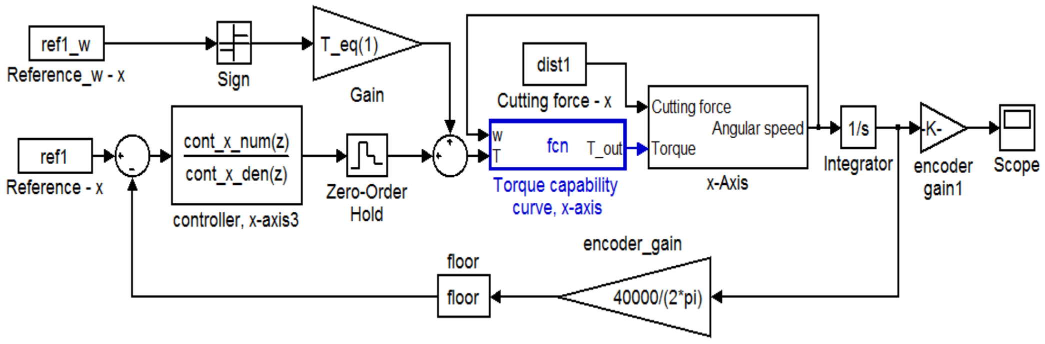


Figure 7.9 – Simulink model of the overall system

The reference trajectory, which has a duration of 20 seconds, is a portion extracted from a real machining operation and is shown in Figure 7.10. As can be seen in Figure 7.10, the reference is provided in terms of encoder counts. On the other hand, the disturbance inputs in Figure 7.11 are not exactly the same as the disturbance on the real system; since the real disturbance (machining) process is relatively hard to model in Simulink environment and is beyond the scope of this thesis. However, an approximation of the actual case, which would provide sufficient information about the disturbance resistance characteristics of the system, can be implemented with ease. Therefore, similar results should be expected from the real test in terms of tracking error. The disturbance inputs corresponding to light- and heavy machining processes are provided in Figure 7.11.

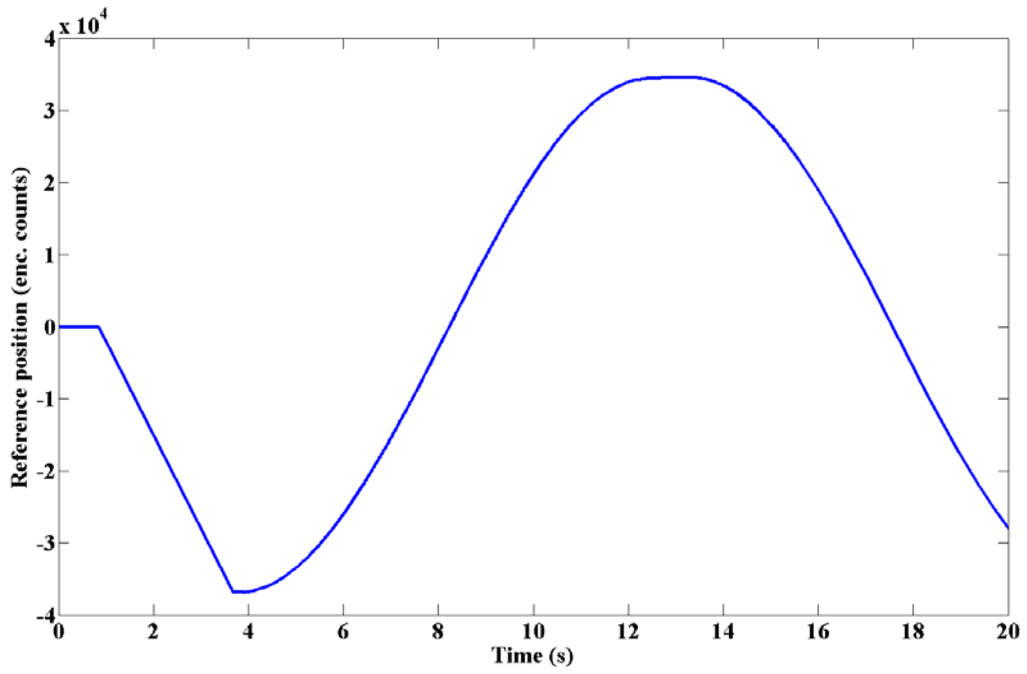


Figure 7.10 – Reference trajectory for the X-axis

It is critical to note that for this particular application, the machining accuracy of the center is designated as ±10μm. Hence, the maximum tracking error of the controlled system under the worst case scenario is expected to be less than 40 encoder counts.

Simulink results are obtained by using the reference trajectory (shown in Figure 7.10), the disturbance input (in Figure 7.11), and the system model provided in Figure 7.9. The results are obtained for both cases of disturbance inputs. The tracking errors are shown in Figure 7.12 in terms of encoder counts.

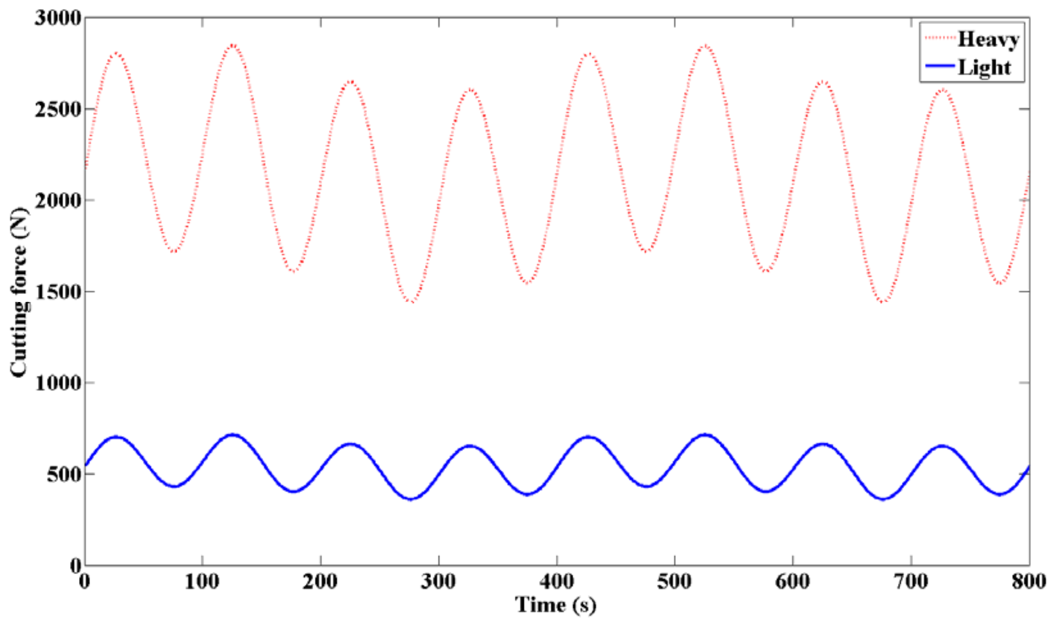


Figure 7.11 – Disturbance inputs for light and heavy machining conditions

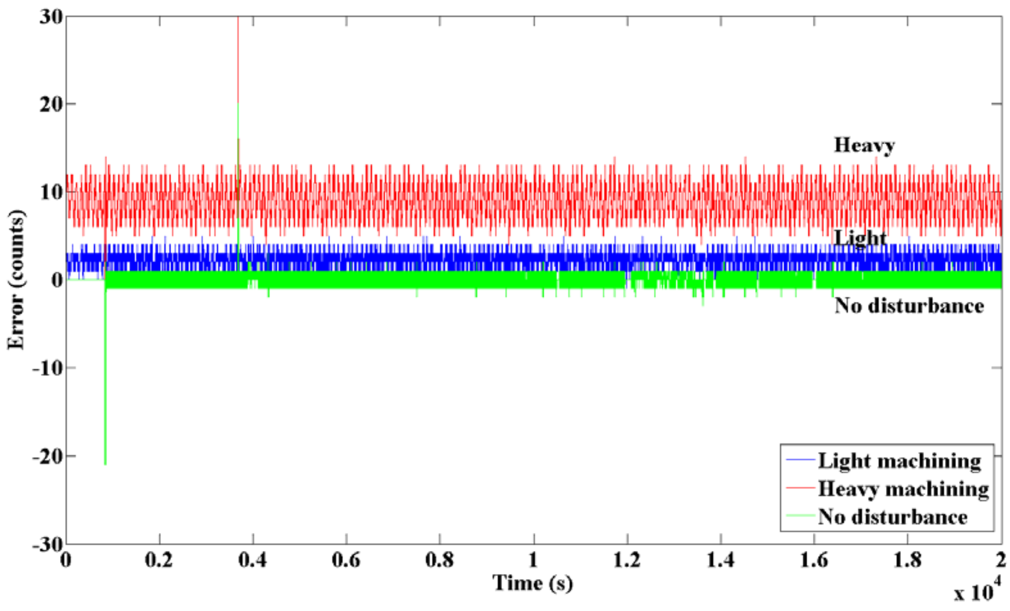


Figure 7.12 – Simulink results of the designed controller in terms of encoder counts

As can be seen in Figure 7.12, when no disturbance is acting on the system, the control system is able to follow the trajectory within an error band of 1 encoder count. However, when a disturbance (i.e. machining force) is present, the tracking error of the system becomes (roughly) 4 and 13 counts for light- and heavy machining (simulation) respectively. Table 7.3 shows the means and standard deviations of error obtained through the simulation study.

Table 7.3 – Mean, max and standard deviation values of the Simulink results (in counts)

Disturbance type	**Mean**	**Max**	**Standard deviation**
No disturbance	0.005	20	0.804
Light disturbance	2.252	23	0.908
Heavy disturbance	9.005	30	1.874

Taking into account the resolution of the encoder (10000 ppr) as well as the pitch of the ballscrew shaft (10 mm), one can determine that 10 encoder counts correspond to a table displacement of 2.5 μm. Thus, Table 7.4 illustrates these table/cart displacement errors.

As can be seen in Table 7.4, the results are quite successful in terms of mean error and standard deviation of the error. The maximum error is also sufficient for control purposes.

Table 7.4 – Mean, max, and standard deviation values of the Simulink results (in μm)

Disturbance type	Mean	Max	Standard deviation
No disturbance	0.001	5	0.201
Light disturbance	0.563	5.75	0.227
Heavy disturbance	2.251	7.5	0.469

If Figure 7.12 is carefully observed, it can be seen that the maximum errors are attained in the sharp transitions of Figure 7.10, which corresponds to "rapid travel/motion" along the trajectory (the phase until t = 4s in Figure). Notice that in CNC technology, rapid travel is a point-to-point motion and position control along the trajectory lying between the initial- and target position is not needed. Therefore, it can be concluded that the designed controller can be applied to the real case.

7.3 Implementation of the system

In the preceding section, the controller design via classical root locus technique is explained and simulation results demonstrating the command tracking and disturbance rejection properties are given for the designed controllers. The results have been proven to be successful and therefore the controller is to be implemented on the FPGA along with other necessary modules for encoder interfacing, PWM generation, etc. In this section, the implementation of the control system and the hardware-in-the-loop simulation (HILS) of the plant are presented.

7.3.1 Implementation of the control system on the FPGA

The control system is implemented on the DE1 FPGA board with an Altera Cyclone II FPGA chip by utilizing the methods and modules provided in the previous chapters. The interface modules that are employed are as follows:

- Incremental encoder decoding module: For gathering angular position information from the axes of the CNC machining center. 3 instances are employed for 3 axes of the machining center.
- PWM transmitter module: For transmitting the calculated torque command to the motor driver (in torque/current modulation mode). 3 instances are employed for 3 axes of the machining center.
- RS-232 controller: For setting the controller parameters and reference values via PC. Single instance is employed.
- SRAM controller: For storing the controller parameters and reference values on the FPGA board. Single instance is employed.

Along with these modules, three controllers are implemented on the chip by employing three instances of the *generic filter module* presented in Chapter 5, using Method I-a (IMUL). Consequently, all the interface modules and three controller modules are implemented on the FPGA chip.

It is important to note that, for control of 3-axis, this design consumes 7550 logic elements (LE), corresponding to 40% of the FPGA's LE resources, along with 12 embedded 9-bit multipliers, corresponding to 23% of the

FPGA's available multiplier resources. On the other hand, if a single axis solution is required, this requirement drops to 3027 logic elements (16%) and 4 embedded 9-bit multipliers (8%). Considering that Altera Cyclone II FPGA is a relatively aged chip (having 18752 total logic elements, 52 embedded 9-bit multipliers), these results prove that the proposed solution is an extremely resource-wise efficient solution. Resource utilization of a single- and 3-axes solution (i.e. synthesized digital circuitry) are presented in Figure 7.13 and Figure 7.14

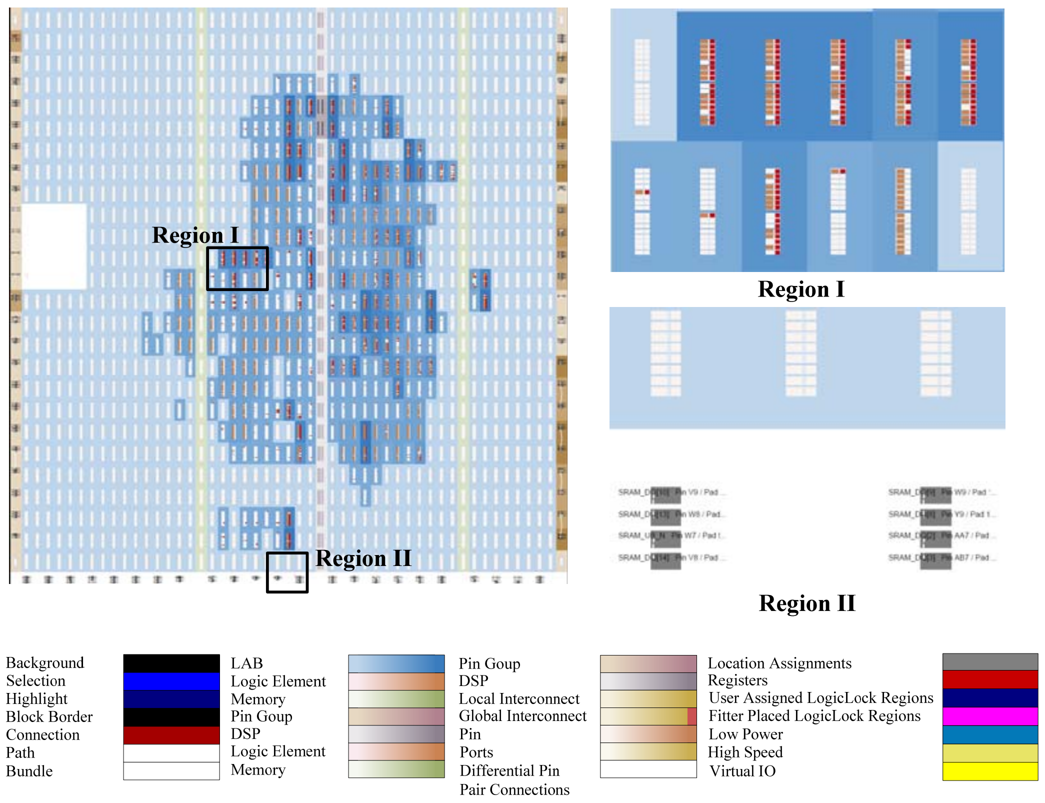


Figure 7.13 – Resource utilization of the single axis solution

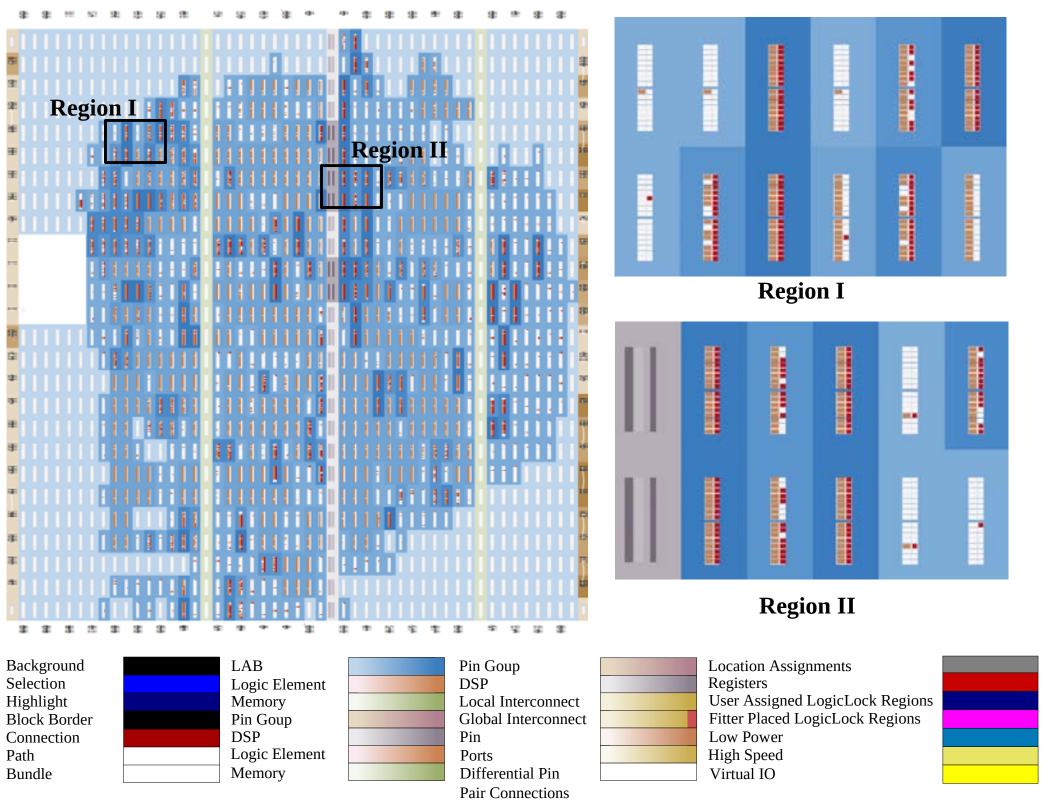


Figure 7.14 – Resource utilization of the implemented 3-axis solution

7.3.2 Realization of the plant via hardware in the loop simulation

HILS of the real system is realized by a hybrid design which includes an Altera FPGA and an Atmel processor. It can provide encoder signals and receive PWM data, exactly the way that a regular CNC machining center would provide and receive. Therefore, within the context of this study, the HILS system is treated as a real plant since there is no difference from the controller's point of view. Figure 7.15 shows the controller system coupled to the HIL simulator.

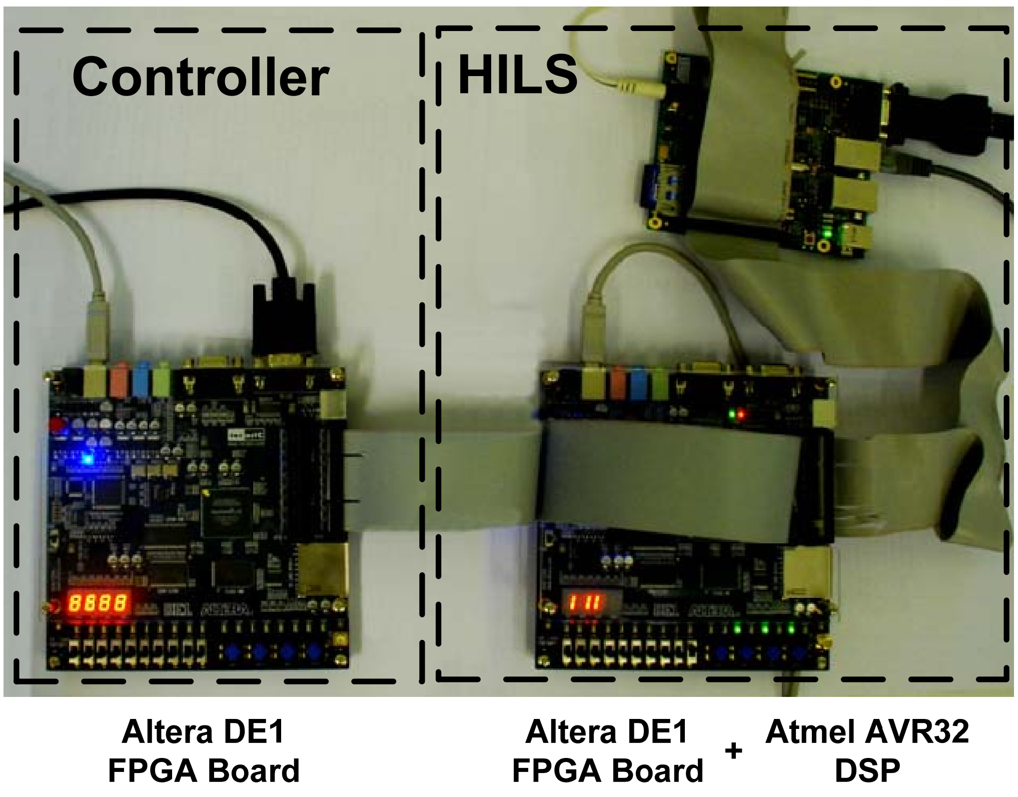


Figure 7.15 – Schematic of the hardware in the loop simulation system

It is critical to notice that the modules within the HILS system is **not** related to the modules presented in Chapter 3. The HILS system is a result of another study [32], and is adopted in this study for the test case simulation of the developed controller.

In chapter 5, the attainable sampling frequency of the digital filter is presented as 385 kHz; therefore the selected sampling frequency (1 kHz) is attainable on the controller end. On the other hand, the computations on the HIL simulator take longer than the selected sampling frequency and inevitably HIL simulation is realized non-real time at 100 Hz. However though, the simulation results represent the real sampling frequency of the system and therefore applicable to a real-time control problem. The results provided in the next section are obtained by utilizing the HILS system.

7.4 HILS Results

HILS results are obtained by using two different reference trajectories, which are portions obtained from a CNC code generated to manufacture a plastic bottle injection mold, as shown in Figure 7.16. The first trajectory is shown in Figure 7.17 and the second one is shown in Figure 7.18.

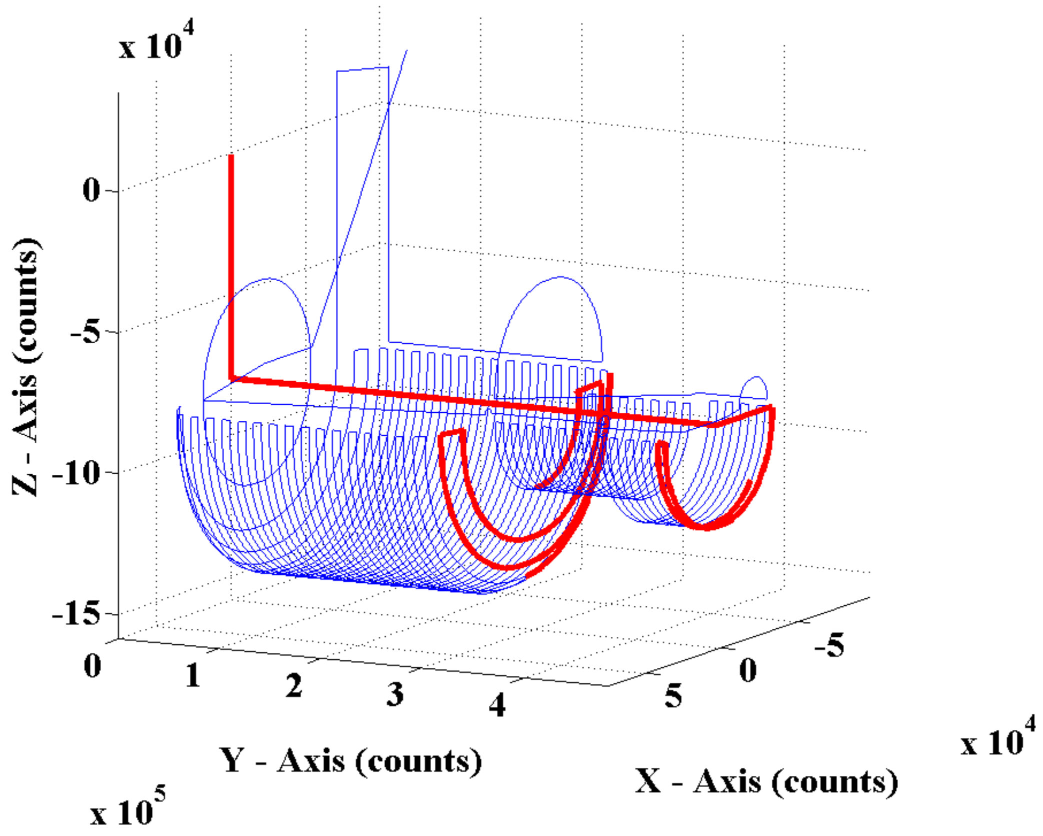


Figure 7.16 – Reference trajectories for a plastic bottle injection mold (selected portions indicated with red color)

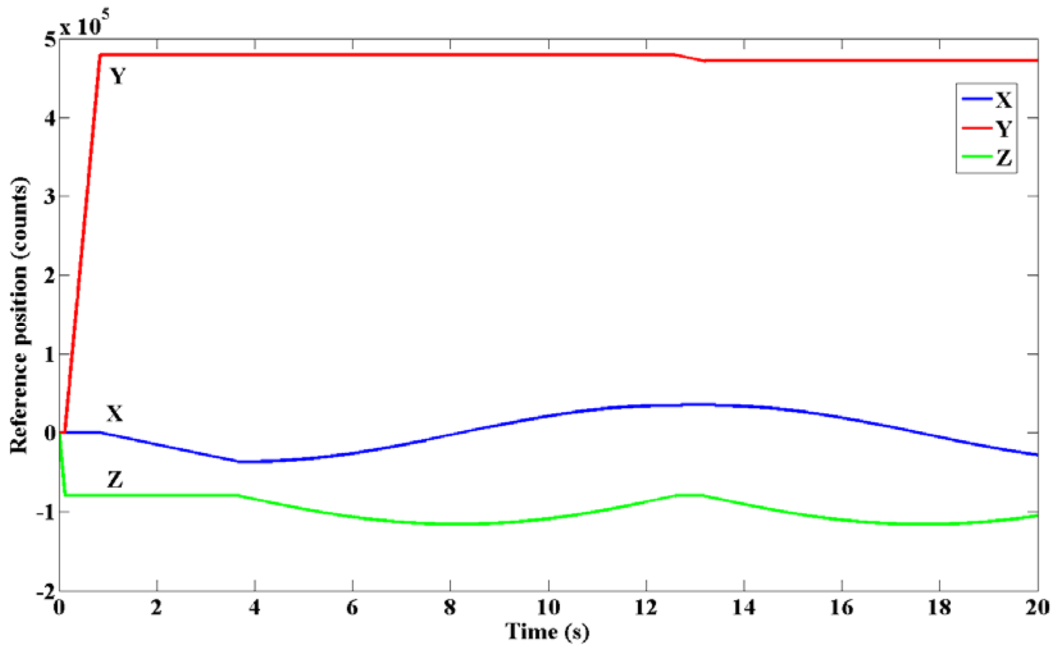


Figure 7.17 – Reference trajectories for X,Y and Z axes (t = 0-20s)

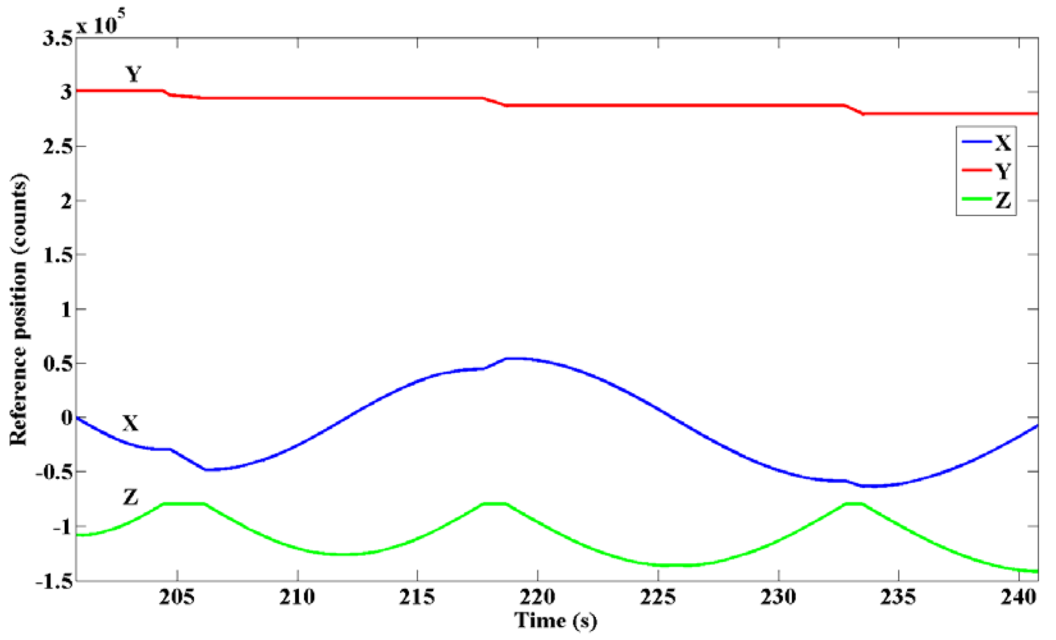


Figure 7.18 – Reference trajectories for X,Y and Z axes (t = 201-241s)

As can be seen from Figure 7.17 and Figure 7.18, the trajectories are portions from a real trajectory set for a CNC machine. The reason why certain portions are selected is that the data to be stored in the SRAM device is limited with 512 kb. However, when a suitable reference command generator is coupled to the design, there is no limit in prolonging the simulation times as the reference data can then be fed to the controller in a continuous fashion. On the other hand, the disturbances that are employed are already discussed in the preceding section and therefore will not be further discussed here.

The results of the HILS are presented in the following order:

1. Figure 7.19 – First trajectory motor position error in X-axis (t = 0-20s)
2. Figure 7.20 – First trajectory motor position error in Y-axis (t = 0-20s)
3. Figure 7.21 – First trajectory motor position error in Z-axis (t = 0-20s)
4. Figure 7.22 – Second trajectory motor position error in X-axis (t = 201-241s)
5. Figure 7.23 – Second trajectory motor position error in Y-axis (t = 201-241s)
6. Figure 7.24 – Second trajectory motor position error in Z-axis (t = 201-241s)
7. Figure 7.25 – Second trajectory cart position error in X-axis with backlash (t = 201-241s)
8. Figure 7.26 – Second trajectory cart position error in Y-axis with backlash (t = 201-241s)
9. Figure 7.27 – Second trajectory cart position error in Z-axis with backlash (t = 201-241s)

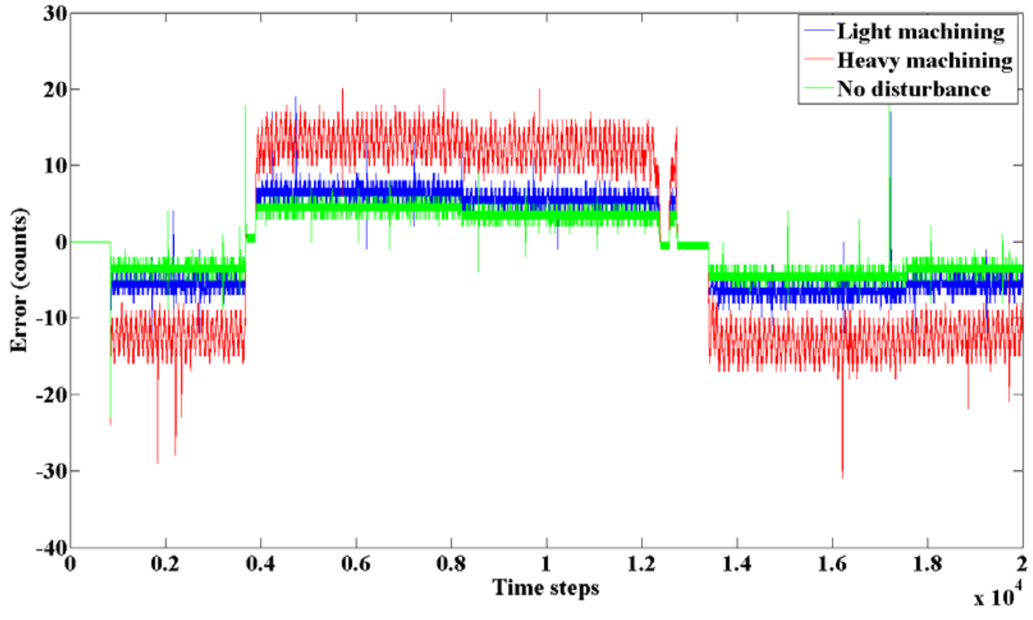


Figure 7.19 – First trajectory motor position error in X-axis (t = 0-20s)

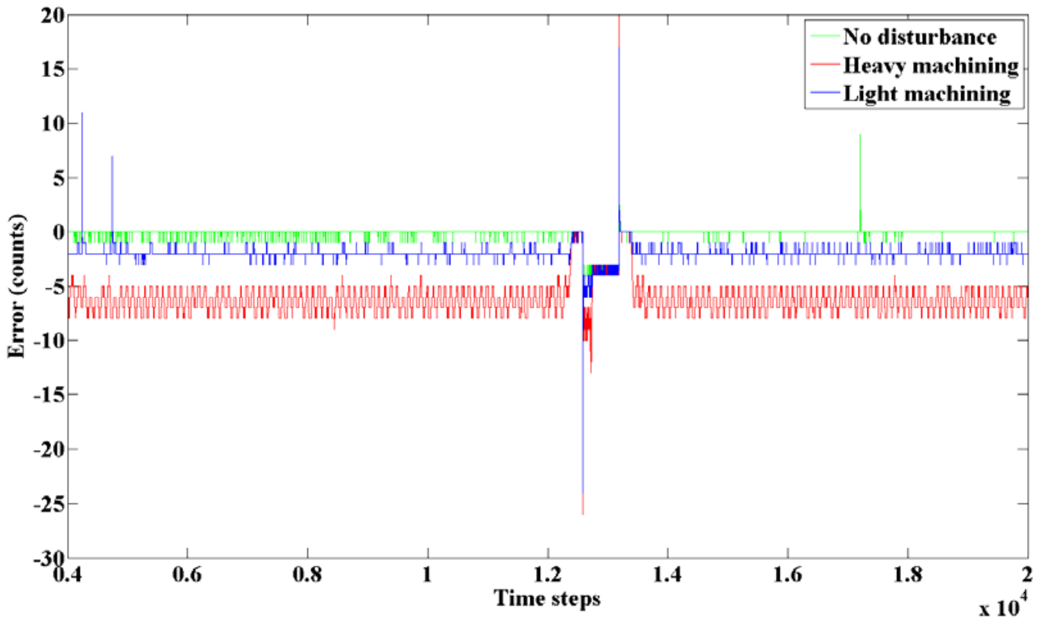


Figure 7.20 – First trajectory motor position error in Y-axis (t = 0-20s)

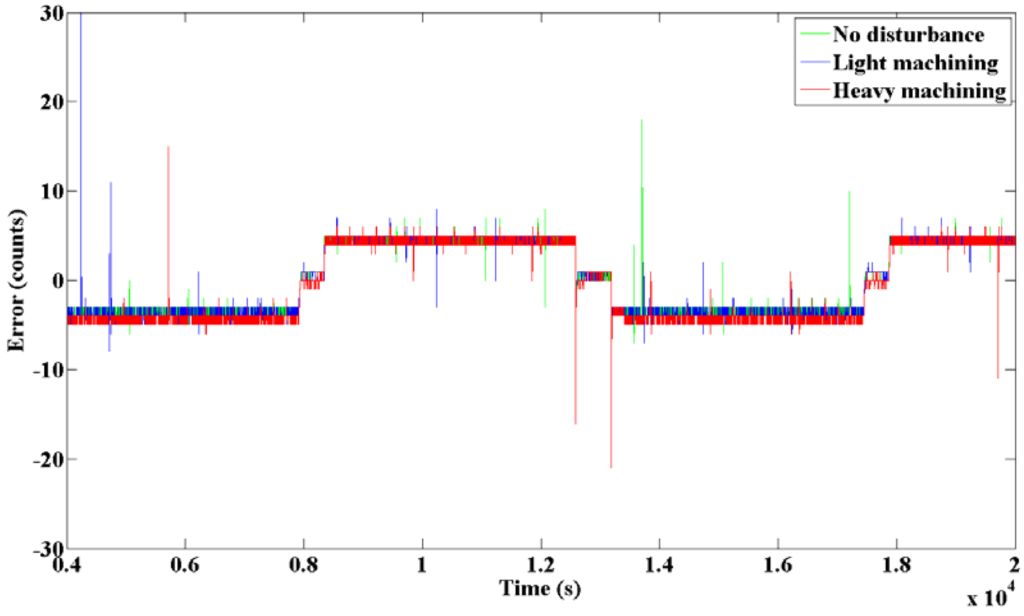


Figure 7.21 – First trajectory motor position error in Z-axis (t = 0-20s)

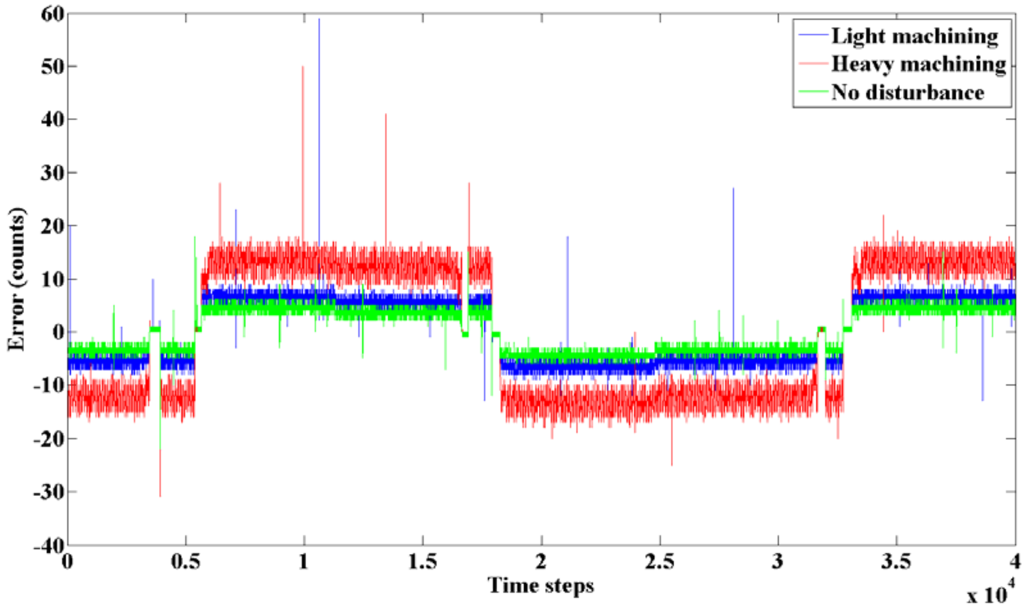


Figure 7.22 – Second trajectory motor position error in X-axis (t = 201-241s)

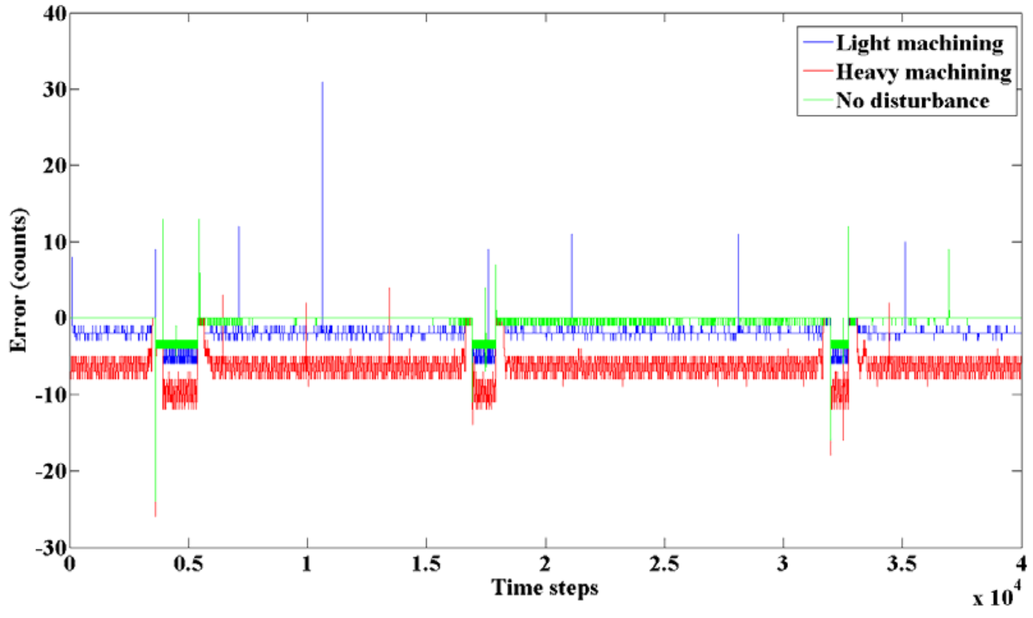


Figure 7.23 – Second trajectory motor position error in Y-axis (t = 201-241s)

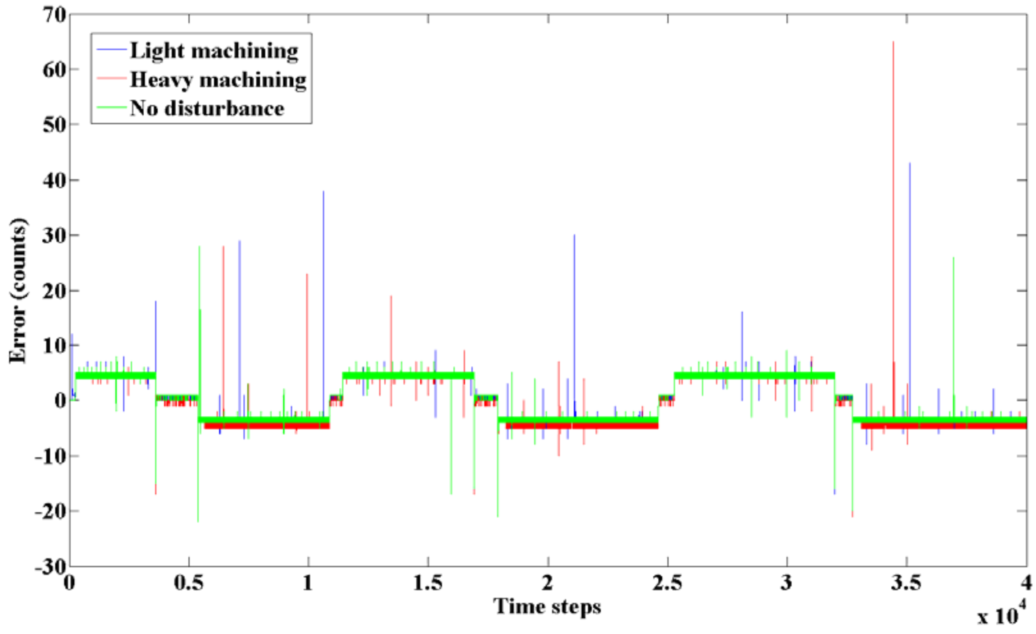


Figure 7.24 – Second trajectory motor position error in Z-axis (t = 201-241s)

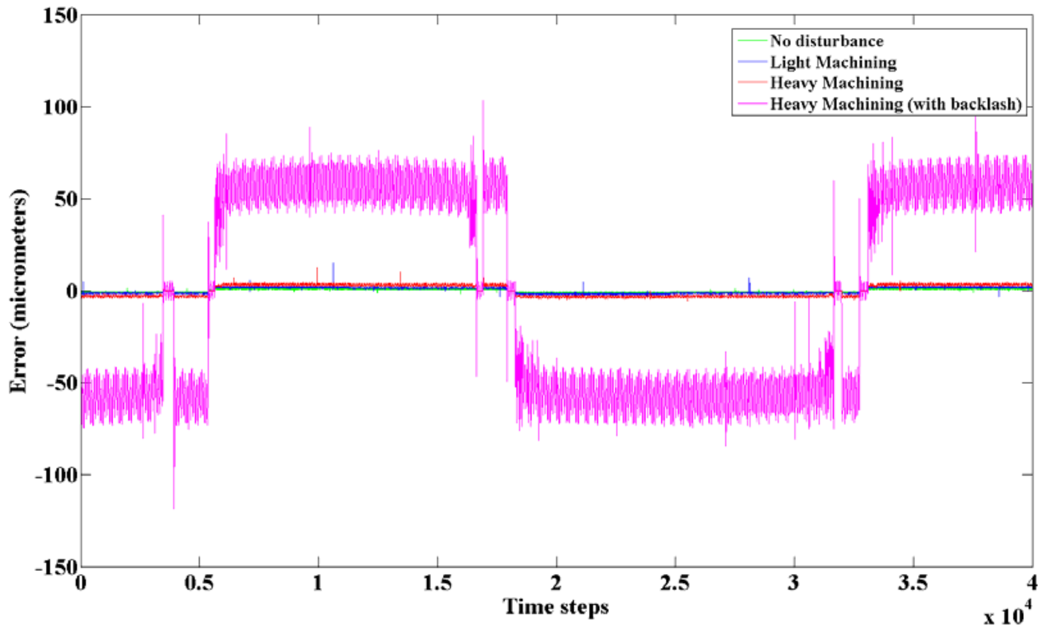


Figure 7.25 – Second trajectory cart position error in X-axis with backlash (t = 201-241s)

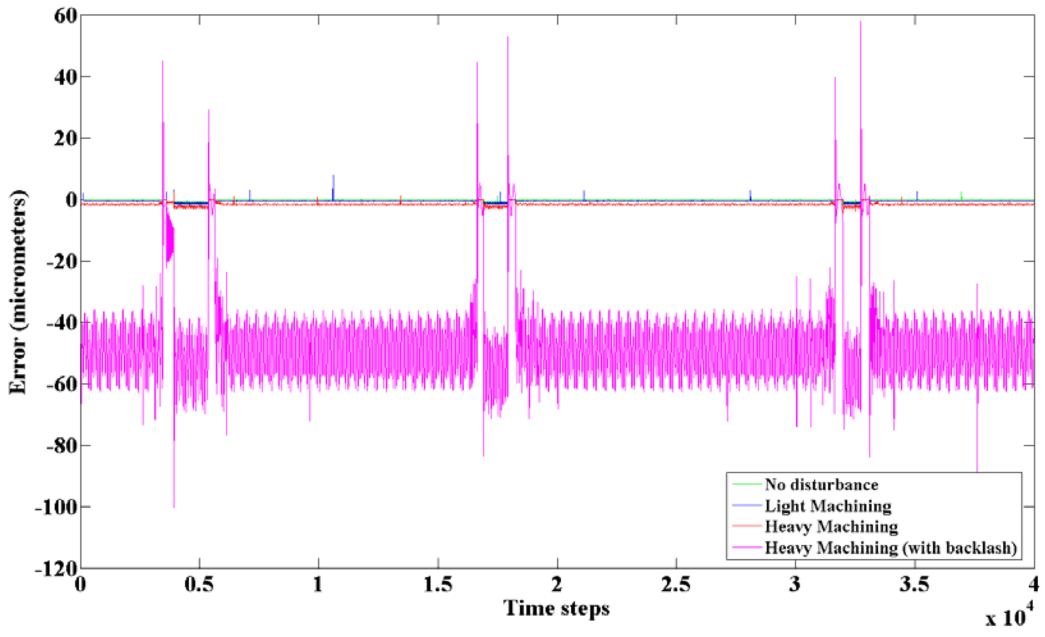


Figure 7.26 – Second trajectory cart position error in Y-axis with backlash (t = 201-241s)

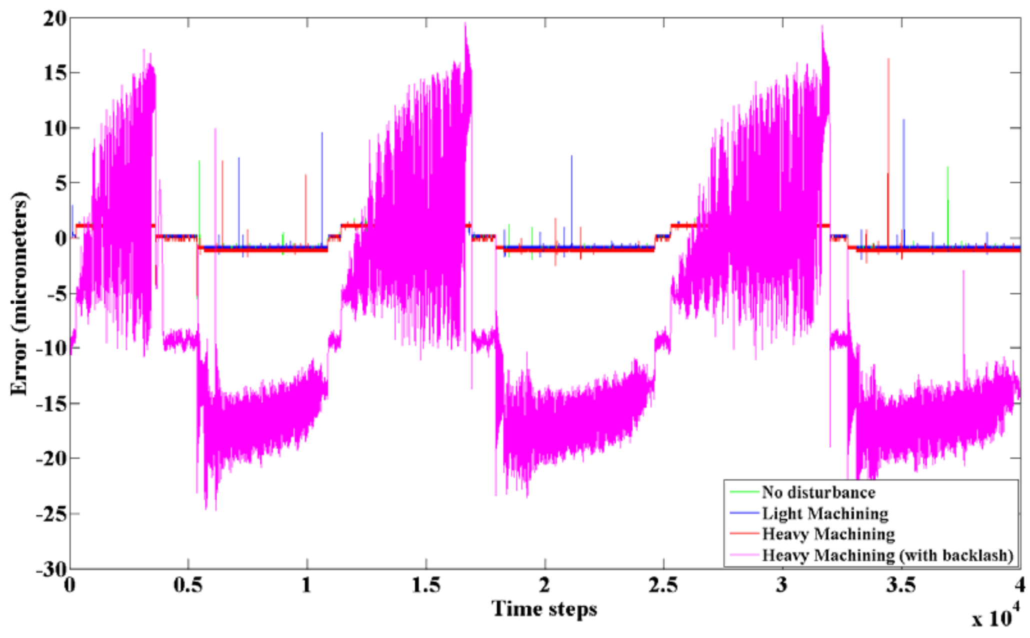


Figure 7.27 – Second trajectory cart position error in Z-axis with backlash (t = 201-241s)

As could be observed from the results, the controller seems successful in the trajectory tracking, as well as disturbance resistance. However, when backlash model is present in the HILS, the error significantly increases in the cart's position. Statistical data provided in Table 7.5 to Table 7.8 would be more useful in interpreting the results. Note that root mean square (RMS) is defined as:

$$\text{RMS} = \sqrt{\frac{1}{K}\sum_{k=0}^{K}[x(k) - x^*(k)]^2} \qquad (7.16)$$

where K is the length of the data, x is the real value of the data and x* is the reference value of the data.

Table 7.5 – Root mean square and standard deviation values of the HILS results for first trajectory (in encoder counts)

Axis	**X**		**Y**		**Z**	
Dist. type	**RMS**	**STD***	**RMS**	**STD**	**RMS**	**STD**
No dist.	3.80	3.78	0.87	0.86	4.08	4.08
Light dist.	6.01	5.98	2.12	0.74	4.21	4.21
Heavy dist.	12.59	12.51	6.38	1.56	4.13	4.10

*STD: Standard deviation

Table 7.6 – Root mean square and standard deviation values of the HILS results for first trajectory (in μm)

Axis	**X**		**Y**		**Z**	
Dist. type	**RMS**	**STD**	**RMS**	**STD**	**RMS**	**STD**
No dist.	0.95	0.95	0.22	0.21	1.02	1.02
Light dist.	1.50	1.49	0.53	0.19	1.05	1.05
Heavy dist.	3.15	3.13	1.60	0.39	1.03	1.03

Table 7.7 – Root mean square and standard deviation values of the HILS results for second trajectory (in encoder counts)

Axis	**X**		**Y**		**Z**	
Dist. type	**RMS**	**STD**	**RMS**	**STD**	**RMS**	**STD**
No dist.	3.79	3.78	1.17	1.12	4.03	4.03
Light dist.	6.02	6.01	2.43	1.13	4.17	4.17
Heavy dist.	12.57	12.57	6.65	1.97	4.13	4.10

As can be seen from Table 7.5 to Table 7.8, the results are very successful in terms of mean error and standard deviation of the errors. Even in the heavy machining case, the maximum error appears in the X axis with an RMS value around 12.6 counts (3.15 μm) and a standard deviation around 12.5 counts (3.14 μm), when the HILS has no backlash model. However, when a backlash model between the ball screw and the cart is included in the HIL simulation, it directly increases the error, both in terms of RMS and STD, as can be seen in Table 7.8. This is an expected result since the controller has no effect on backlash compensation.

When the figures are observed, even there exist large errors in rare occasions; they are still within an acceptable range for the CNC machining center control task. Therefore, it can be concluded that the designed controller is successful in trajectory tracking, even under heavy machining condition.

Table 7.8 – Root mean square and standard deviation values of the HILS results for second trajectory including backlash model under heavy machining (in μm)

Axis	**X**		**Y**		**Z**	
Dist. type	**RMS**	**STD**	**RMS**	**STD**	**RMS**	**STD**
No dist.	0.948	0.95	0.29	0.28	1.01	1.01
Light dist.	1.506	1.50	0.61	0.28	1.04	1.04
Heavy dist.	3.14	3.14	1.66	0.49	1.03	1.03
Heavy d.+ BL*	55.94	55.94	49.17	13.56	12.49	9.38

(* d. + BL = disturbance + backlash)

7.5 Closure

In this chapter, most of the aforementioned modules and methods are utilized in an FPGA based motion controller and the results of the test case have proven that the proposed solution is a successful design. The results are comparable with industrial motion controllers in terms of performance and significantly efficient in terms of resource requirements. Therefore, it can be concluded that the proposed solution (FPGA based implementation) have proven to be a useful for motion control (or CNC) applications.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this chapter, an overall assessment of this thesis is presented, along with a number of suggestions that can follow and contribute to this study.

8.1 Conclusion

In this thesis, methods for developing an FPGA based motion control system are investigated. In this perspective, efficient and successful design methods are developed for each element of a typical motion control system, including peripheral interfaces and the controller parts. At certain points, some methods are highlighted as the best design approach; whereas at other points, methods have proven to be compatible with each other. However though, it can be seen that in the majority of this study, hardwired approach is adopted rather than the embedded processor design method.

The reasoning behind this selection can be justified as to obtain the maximum capability of the FPGA chip. Low-level design methodology required by hardwired approach allows the designer to customize a segment of the FPGA, to execute a specific task in an efficient manner. Furthermore, many instances of the designed module can also be used in parallel, as demonstrated in the previous chapters. However, utilizing an embedded processor for a certain task decreases the process time, consumes abundant amount of resources and eliminates the chance of parallel implementation.

On the other hand, employing an embedded processor has also certain advantages such as faster development and easy to use interfaces. Furthermore, when processing speed is not crucial and the design is complex, the embedded processor implementation may be comparable to hardwired design in terms of resource requirements. Therefore, in Chapters 4 and 5 of this thesis, both design methodologies have been evaluated and the results are presented for both approaches. Notice that in those chapters, state-space controller (and observer) and filter designs are presented, where processing speed is not crucial (clearly above a certain limit) and the design is relatively complex. Results approve that the resource consumptions are comparable and the process times significantly differ.

It can be observed that in this thesis two different FPGAs are employed from the two leading FPGA manufacturers; that are Altera (Cyclone II) and Xilinx (Virtex-5). However though, in the majority of the thesis (except Chapter 6) Altera's DE1 board with the Cyclone II (EP2C20F484C7N) chip is employed even though the Xilinx Virtex-5 is a more recent and resourceful chip. The reason for this choice is that development tools that are provided by Xilinx require significantly more time to compile, are more error-prone and harder to debug.

Certain advanced controller topologies are also investigated in this thesis, however due to the shortcomings of the Xilinx FPGA chip and lack of time, only a single method could be implemented to demonstrate the utilization of advanced controllers on an FPGA. Furthermore, the test case provided in Chapter 6 is also different from the test cases provided in Chapter 4 and 5. Nevertheless, the results have been successful and implementation of an advanced controller is demonstrated, even with a single method.

An important aspect of this thesis is that, all the chapters that include a design process contain a test case within itself, to demonstrate the success of the method. However, a test case is also provided in Chapter 7 for the assembled solution, which shows the success of the overall motion control system, including the peripheral interfaces as well as the controller. Therefore, the methods are both proven individually and as an assembled product. Notice that HILS is employed for testing the assembled solution, instead of a real test setup (a CNC machining center), however from the control system's point of view there is no difference between the two, since the simulator provides encoder signals and accepts PWM signal, as it would be in the real case. Although the HILS is not conducted real-time (due to limitations of the simulator) the proposed system is proven to be capable of real-time control, in the respective design chapters of the control system's elements. Therefore the results are evenly applicable to the real-time control case.

8.2 Future work

As addressed in the previous section, an important contribution to this study would be to utilize this system to control a real-world system, preferably a CNC machining center, in order to compare the results obtained via HILS. Although the HIL simulator reflects the real-world application in a very good manner, it is always desirable to utilize the designed control system in a real-world control application.

In this study, it can be deduced that the conventional control methodologies are covered quite thoroughly in Chapters 4 and 5. However in Chapter 6, only a single method could be presented for the advanced controller implementations. As a future study, other methods provided in the previous Chapters could be implemented to obtain a more detailed discussion between different designs. Furthermore, it would also be more meaningful if the study is conducted on an Altera Cyclone II FPGA, to provide a comparison between the conventional and intelligent controllers' implementation on the same chip.

As a final remark, it should also be noted that in Chapter 7, due to the unavailability of a command generator, the reference commands are written to SRAM before the simulation; which resulted in a limitation of the simulation time. However, as a future work, the commands could be received from an outside source to run the full-time simulation, utilizing the developed modules that are presented in Chapter 3.

REFERENCES

[1] R. Usselmann [1], Open Floating Point Unit Manual, www.opencores.org, 2000.

[2] Arbit, A.; Pritzker, D.; Kuperman, A.; Rabinovici, R.; , "A DSP-controlled PWM generator using field programmable gate array," Electrical and Electronics Engineers in Israel, 2004. Proceedings. 2004 23rd IEEE Convention of , pp. 325- 328, 6-7 Sept. 2004

[3] Xu Dong; Wang Tianmiao; Wei Hongxing; Liu Jingmeng, "A new dual-core Permanent Magnet Synchronous Motor Servo System," Industrial Electronics and Applications, 2009. ICIEA 2009. 4th IEEE Conference on , pp.715-720, 25-27 May 2009.

[4] Birou, I.; Imecs, M., "Real-time robot drive control with PM-synchronous motors using a DSP-based computer system," Power Electronics and Motion Control Conference, Proceedings of the Third International IPEMC 2000., vol.3, pp.1290-1295 vol.3, 2000.

[5] Geun-Hyung Lee; Sung-Su Kim; Seul Jung; , "Hardware Implementation of a RBF Neural Network Controller with a DSP 2812 and an FPGA for Controlling Nonlinear Systems," Smart Manufacturing Application, 2008. ICSMA 2008. International Conference on , pp.167-171, 9-11 April 2008

[6] Morales-Caporal, R.; Pacas, M.; , "Digital implementation of a direct mean torque control for AC servo drives based on a hybrid DSP/FPGA controller system," Power Electronics Congress, 2008. CIEP 2008. 11th IEEE International , pp.77-83, 24-27 Aug. 2008

[7] DMC-18x6 reference manual, Galil Motion Control Co., 2010

[8] PMAC2 hardware reference manual, Delta-Tau Co., 2010

[9] Al-Ayasrah, O.; Alukaidey, T.; Pissanidis, G.; , "DSP Based N-Motor Speed Control of Brushless DC Motors Using External FPGA Design," Industrial Technology, 2006. ICIT 2006. IEEE International Conference on , pp.627-631, 15-17 Dec. 2006

[10] Toh, C.L.; Idris, N.R.N.; Yatim, A.H.M.; Muhamad, N.D.; Elbuluk, M.; , "Implementation of a New Torque and Flux Controllers for Direct Torque Control (DTC) of Induction Machine Utilizing Digital Signal Processor (DSP) and Field Programmable Gate Arrays (FPGA)," Power Electronics

Specialists Conference, 2005. PESC '05. IEEE 36th , pp.1594-1599, 16-16 June 2005

[11] Seul Jung; Sung su Kim, "Hardware Implementation of a Real-Time Neural Network Controller With a DSP and an FPGA for Nonlinear Systems," IEEE Transactions on Industrial Electronics, vol.54, no.1, pp.265-271, Feb. 2007.

[12] Kaiping Yu,; Hong Guo,; Dayu Wang,; Lanfeng Li,; , "Design of multi-redundancy electro-mechanical actuator controller with DSP and FPGA," Electrical Machines and Systems, 2007. ICEMS. International Conference on , pp.584-587, 8-11 Oct. 2007

[13] Esmaeli, A.; Li Bo; Sun Li; , "A Novel AC Servo System Implementation," 9th International Multitopic Conference, IEEE INMIC 2005 , pp.1-5, 24-25 Dec. 2005

[14] Ni, F.L.; Jin, M.H.; Xie, Z.W.; Shi, Sh.C.; Liu, Y.Ch.; Liu, H.; Hirzinger, G., "A Highly Integrated Joint Servo System Based on FPGA with Nios II Processor," Proceedings of the 2006 IEEE International Conference on Mechatronics and Automation, pp.973-978, 25-28 June 2006.

[15] Li, Yan; Zhuang, Shengxian; Zhang, Luan, "Development of an FPGA-Based Servo Controller for PMSM Drives,", 2007 IEEE International Conference on Automation and Logistics, vol. 18, no. 21, pp.1398-1403, Aug. 2007.

[16] Ying-Shieh Kung; Rong-Fong Fung; Ting-Yu Tai, "Realization of a Motion Control IC for X-Y Table Based on Novel FPGA Technology," IEEE Transactions on Industrial Electronics, vol.56, no.1, pp.43-53, Jan. 2009.

[17] Das, A.; Banerjee, K.; "Fast prototyping of a digital PID controller on a FPGA based softcore microcontroller for precision control of a brushed DC servo motor," Industrial Electronics, 2009. IECON '09. 35th Annual Conference of IEEE , pp.2825-2830, 3-5 Nov. 2009

[18] Ben Salem, A.K.; Ben Othman, S.; Ben Saoud, S.; Litayem, N.; , "Servo drive system based on programmable SoC architecture," Industrial Electronics, 2009. IECON '09. 35th Annual Conference of IEEE , pp.2961-2966, 3-5 Nov. 2009

[19] Jung Uk Cho; Quy Ngoc Le; Jae Wook Jeon, "An FPGA-Based Multiple-Axis Motion Control Chip," IEEE Transactions on Industrial Electronics, vol.56, no.3, pp.856-870, March 2009.

[20] Chan, Y.F.; Moallem, M.; Wang, W., "Efficient implementation of PID control algorithm using FPGA technology," 43rd IEEE Conference on Decision and Control, vol.5, pp. 4885-4890 Vol.5, 14-17 Dec. 2004.

[21] Yao dong Tao; Hu Lin; Yi Hu; Xiaohui Zhang; Zhicheng Wang, "Efficient implementation of CNC Position Controller using FPGA,".. 6th IEEE International Conference on Industrial Informatics (INDIN 2008), pp.1177-1182, 13-16 July 2008.

[22] Jia Lanping; Zhou Runjing; Liang Zhian, "Realization of position tracking system based on FPGA," 2004. Proceedings. ICSP '04. 2004 7th International Conference on Signal Processing, vol.3, pp. 2588-2591 vol.3, 31 Aug.-4 Sept. 2004.

[23] Ying-Shieh Kung; Ming-Shyan Wang; Chung-Chun Huang; , "Digital Hardware Implementation of Adaptive Fuzzy Controller for AC Motor Drive," Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE , pp.1208-1213, 5-8 Nov. 2007

[24] Faa-Jeng Lin; Ying-Chih Hung, "FPGA-based Elman Neural Network Control System for Linear Ultrasonic Motor," IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency, vol.56, no.1, pp.101-113, January 2009.

[25] Ying-Shieh Kung; Ming-Shyan Wang; Tzu-Yao Chuang; , "FPGA-based self-tuning PID controller using RBF neural network and its application in X-Y table," Industrial Electronics, 2009. ISIE 2009. IEEE International Symposium on , pp.694-699, 5-8 July 2009

[26] Liaw, C.-M.; Wang, J.-B., "Design and implementation of a fuzzy controller for a high performance induction motor drive," Systems, Man and Cybernetics, IEEE Transactions on , vol.21, no.4, pp.921-929, Jul/Aug 1991.Arbit, A.; Pritzker, D.; Kuperman, A.; Rabinovici, R.; , "A DSP-controlled PWM generator using field programmable gate array," Electrical and Electronics Engineers in Israel, 2004. Proceedings. 2004 23rd IEEE Convention of , pp. 325- 328, 6-7 Sept. 2004

[27] Alexei, Z.; Sandor, H., "Robust speed fuzzy logic controller for DC drive," Intelligent Engineering Systems, 1997. INES '97. Proceedings., 1997 IEEE International Conference on , pp.385-389, 15-17 Sep 1997.

[28] Sio, K.C.; Lee, C.K., "Identification of a nonlinear motor system with neural networks," Advanced Motion Control, 1996. AMC '96-MIE. Proceedings., 1996 4th International Workshop on , vol.1, pp.287-292 vol.1, 18-21 Mar 1996.

[29] Slotine, J.J.E., and Li, W., Applied Nonlinear Control, Prentice-Hall, 1991.

[30] Brock, S.; Deskur, J.; Zawirski, K., "Modified sliding-mode speed controller for servo drives," Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on , vol.2, pp.635-640 vol.2, 1999.

[31] Usenmez, S.; Dilan R.A; Dolen M.; Koku A.B., “Real-Time Hardware-in-the-Loop Simulation of Electrical Machine Systems Using FPGAs”, to appear in the Proc. of the International Conference on Electrical Machines and Systems (ICEMS), 2009.

[32] Usenmez, S. “Design of an integrated hardware-in-the-loop simulation system”, Master of Science Thesis, Department of Mechanical Engineering, Middle East Technical University, Graduate School of Natural and Applied Sciences, 2010.

APPENDIX A

LIST OF VERILOG HDL FILES

In this appendix, a list of Verilog HDL files that are employed in different parts of this study is provided, along with their related chapters and corresponding functions.

Table A.1 – List of Verilog HDL files employed in the thesis

Name	**Related chapter**	**Function**
`encoder.v`	Chapter 3	Incremental encoder interface
`pwm_gen_v2.v`	Chapter 3	PWM signal generator
`pwm_transmit_v2.v`	Chapter 3	PWM data transmitter
`pwm_receive_v2.v`	Chapter 3	PWM data receiver
`clk_divider.v`	Chapter 3	Clock generator
`pulse_gen_v2.v`	Chapter 3	Finite pulse generator
`SPI.v`	Chapter 3	SPI slave module
`command_recv.v`	Chapter 3	Custom parallel data receiver
`sasc_top.v`	Chapter 3	RS-232 controller
`sram_ctrl.v`	Chapter 3	SRAM controller
`imul.v`	Chapter 4	Integer multiplication module
`fmul.v`	Chapter 4	Fixed-point multiplication module
`fpu.v`	Chapter 4	Floating point unit
`mul_kx.v`	Chapter 4	Fractional / Quantized multiplication module
`filter_v2.v`	Chapter 5	Generic filter module

APPENDIX B

LIST OF C CODES FOR NIOS II IMPLEMENTATION

In this appendix, a list of C codes developed for Nios II softcore processor implementation of methods presented in Chapter 4 and Chapter 5 is presented.

Table B.1 – List of Nios II C files employed in the thesis

Name	**Related chapter**	**Function**
`nios_ss_imul.c`	Chapter 4	Method II-a implementation
`nios_ss_fmul.c`	Chapter 4	Method II-b implementation
`nios_ss_fpu.c`	Chapter 4	Method II-c implementation
`nios_filter_imul.c`	Chapter 5	Method II-a implementation
`nios_filter_fmul.c`	Chapter 5	Method II-b implementation
`nios_filter_fpu.c`	Chapter 5	Method II-c implementation

APPENDIX C

SAMPLE C CODE FOR NIOS II IMPLEMENTATION

In this appendix, a sample C code developed for Nios II softcore processor implementation of method II-a presented in Chapter 4 is presented.

```c
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include "altera_avalon_performance_counter.h"
#define perfctr_base (void*)0x00900020

short readshort();
unsigned short readshortu();
void writeshort(short val);
void writeint32(int val);
int fmul(short k_int, short k_frac, int x);

int main()
{
  signed int perfctr_count;
  unsigned int perfctr_rate = alt_get_cpu_freq();
  int i,j,k;

  short K_ni[4];
  short K_df[4];
  short F_ni[4][4];
  short F_df[4][4];
  short G_ni[4];
  short G_df[4];
  short L_ni[4][4];
  short L_df[4][2];
  short x_read[2];
  int x[4];
  int xkm1[4];

  int u = 0;
  int ukm1 = 0;

      for(k=0;k<4;k++){
        xkm1_f[k] = 0;
        xkm1[k] = 0;
        x_f[k] = 0;
        x[k] = 0;
    }


  while(1){
```

```
  for(i=0;i<4;i++){
  K_ni[i] = readshort();
  K_df[i] = readshort();
  }


  for(i=0;i<4;i++){
      for(j=0;j<4;j++){
        F_ni[i][j] = readshort();
        F_df[i][j] = readshort();
      }
  }

  for(i=0;i<4;i++){
  G_ni[i] = readshort();
  G_df[i] = readshort();
  }

  for(i=0;i<4;i++){
      for(j=0;j<2;j++){
        L_ni[i][j] = readshort();
        L_df[i][j] = readshort();
      }
  }

  for(i=0;i<200;i++){

  x_read[0] = readshort();
  x_read[1] = readshort();

  PERF_RESET(perfctr_base);
  PERF_START_MEASURING(perfctr_base);

// SS with int

  for(k=0;k<4;k++){

      for(j=0;j<4;j++){
      x[k] = x[k] + (F_ni[k][j]*xkm1[j])/F_df[k][j];
      }

      x[k] = x[k] + (G_ni[k]*ukm1/G_df[k]);

      for(j=0;j<2;j++){
      if(j==0) x[k] = x[k] + (L_ni[k][j]*(x_read[j]-
xkm1[0])/L_df[k][j]);
      if(j==1) x[k] = x[k] + (L_ni[k][j]*(x_read[j]-
xkm1[2])/L_df[k][j]);
      }
  }

  for(k=0;k<4;k++){
  u = u + (K_ni[k]*xkm1[k]/K_df[k]);
  }

  PERF_STOP_MEASURING(perfctr_base);
```

```c
  perfctr_count = perf_get_total_time(perfctr_base);

  writeint32(u);
  writeint32(perfctr_count);

    ukm1 = u;
    u = 0;

    for(k=0;k<4;k++){
        xkm1_f[k] = x_f[k];
        xkm1[k] = x[k];
        x_f[k] = 0;
        x[k] = 0;
    }
  }
  } //End of while(1)
  return 0;
}

short readshort()
{
short val;
short temp;
val = getchar();
temp = getchar();
val = val | (temp << 8);
return val;
}

void writeshort(short val)
{
    putchar((char)val);
    usleep(2000);
    putchar((char)(val >> 8));
}

void writeint32(int val)
{
    putchar((char) val);
    usleep(2000);
    putchar((char)(val >> 8));
    usleep(2000);
    putchar((char)(val >> 16));
    usleep(2000);
    putchar((char)(val >> 24));
}
```

APPENDIX D

LIST OF MATLAB M-FILES

In this appendix, a list of MATLAB M-files developed for HILS employed in Chapter 4 and Chapter 5 is presented.

Table D.1 – List of MATLAB M-files employed in the thesis

Name	**Related chapter**	**Function**
`sim_ss_imul.m`	Ch. 4	Sending initial parameters to FPGA and realizing HILS for hardwired implementation
`sim_ss_fmul.m`	Ch. 4	
`sim_ss_fpu.m`	Ch. 4	
`sim_ss_nios_imul.m`	Ch. 4	Sending initial parameters to FPGA and realizing HILS for softcore implementation
`sim_ss_nios_fmul.m`	Ch. 4	
`sim_ss_nios_fpu.m`	Ch. 4	
`sim_filter_imul.m`	Ch. 5	Sending initial parameters to FPGA and realizing HILS for hardwired implementation
`sim_filter_fmul.m`	Ch. 5	
`sim_filter_fpu.m`	Ch. 5	
`sim_filter_nios_imul.m`	Ch. 5	Sending initial parameters to FPGA and realizing HILS for softcore implementation
`sim_filter_nios_fmul.m`	Ch. 5	
`sim_filter_nios_fpu.m`	Ch. 5	

APPENDIX E

SAMPLE MATLAB M-FILE FOR HILS

In this appendix, a sample M-file (`sim_ss_nios_imul.m`)is presented.

```
clc;
clear;
s = serial('COM5');
set(s,'BaudRate', 57600, 'DataBits', 8, 'Parity',
'none','StopBits', 1, 'FlowControl', 'none');

state_no = 4; %Hence the total number of 16-bit packages to send
is equal to 8 (state_no*2(for K matrix)).
known_state_no = 2;
simulation_time = 20; %in seconds
t_sample = 0.001; %in seconds

state_history = simulation_time/t_sample;

ctr1=1;
T_new = zeros(state_history,1);
F_new = zeros(state_history,1);
F_read = zeros(state_history,1);
F_dene = zeros(state_history,1);
Y_new = zeros(state_history,4);
Y_new(1,1) = 0; %Set inital position of the cart
Y_new(1,3) = -0.05; %Set inital position of the pendulum
Y_send = zeros(state_history,2);
Y_read = zeros(state_history,3);
states = zeros(state_history,4);
X_current = zeros(4,1);

K = [-0.9975   -2.3195   29.0515   17.6788]; %define K Matrix

[K_n,K_d] = rat(K);

F_matrix = [1.0000    0.0010    0.0000    0.0000;
         0    0.9998    0.0027    0.0000;
         0   -0.0000    1.0000    0.0010;
         0   -0.0005    0.0312    1.0000];

[F_n,F_d] = rat(F_matrix);

G= [0.0000;
     0.0018;
     0.0000;
     0.0045];
```

```
[G_n,G_d] = rat(G);



L=     1.0e+002 * [0.0026   -0.0000;
    1.7286   -0.0153;
   -0.0000    0.0026;
   -0.0110    1.7294];

[L_n,L_d] = rat(L);

F = 0;
u = 0.2;

Y_send(1,1) = round(Y_new(1,1)*10000); %x = cart position
Y_send(1,2) = round(Y_new(1,3)*10000); %theta = angular position
of pendulum

fopen(s);

%IMUL

for i1=1:1:state_no
   fwrite(s, K_n(i1), 'int16');
   fwrite(s, K_d(i1), 'int16');
end

pause(0.01);

for i1=1:1:state_no
   for i2=1:1:state_no
   fwrite(s, F_n(i1,i2), 'int16');
   fwrite(s, F_d(i1,i2), 'int16');
   end
end

pause(0.01);

for i1=1:1:state_no
   fwrite(s, G_n(i1,1), 'int16');
   fwrite(s, G_d(i1,1), 'int16');
end

pause(0.01);

for i1=1:1:state_no
    for i2=1:1:known_state_no
    fwrite(s, L_n(i1,i2), 'int16');
    fwrite(s, L_d(i1,i2), 'int16');
    end
end
pause(0.01);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
for i=t_sample:t_sample:simulation_time

fwrite(s, Y_send(ctr1,1), 'int16');
fwrite(s, Y_send(ctr1,2), 'int16');
Y_read(ctr1,1) = fread(s,1,'int32');
Y_read(ctr1,2) = fread(s,1,'int32');

F = (-Y_read(ctr1,1)/10000);

[T,Y] = ode45(@(t,y) myode(t,y,F),[i-t_sample i],[Y_new(ctr1,1)
Y_new(ctr1,2) Y_new(ctr1,3) Y_new(ctr1,4)]);

Y_new(ctr1+1,1) = Y(size(T,1),1);
Y_new(ctr1+1,2) = Y(size(T,1),2);
Y_new(ctr1+1,3) = Y(size(T,1),3);
Y_new(ctr1+1,4) = Y(size(T,1),4);

X_current(1,1) = Y_new(ctr1+1,1);
X_current(2,1) = Y_new(ctr1+1,2);
X_current(3,1) = Y_new(ctr1+1,3);
X_current(4,1) = Y_new(ctr1+1,4);

Y_send(ctr1+1,1) = round((Y_new(ctr1+1,1))*10000); %x = cart
position
Y_send(ctr1+1,2) = round((Y_new(ctr1+1,3))*10000); %theta =
angular position of pendulum

if(abs(Y_send(ctr1+1,1)) > 32765)
    Y_send(ctr1+1,1) = sign(Y_send(ctr1+1,1))*32765;
end

if(abs(Y_send(ctr1+1,2)) > 32765)
    Y_send(ctr1+1,2) = sign(Y_send(ctr1+1,2))*32765;
end
F_new(ctr1+1,1) = F;
T_new (ctr1,1) = i;
ctr1 = ctr1 + 1;
end
fclose(s);

Y_new = Y_new(1:state_history,1:4);
F_new = F_new(1:state_history,1);
Y_send = Y_send(1:state_history,1:2);

plot(T_new, Y_new(:,3));
hold on;
plot(T_new, Y_new(:,1),'r');
title('Plot of x as a function of time');
xlabel('Time'); ylabel('Y(t)');
```