METHODOLOGIES FOR FPGA IMPLEMENTATION OF FCS-MPC FOR ELECTRIC MOTOR DRIVES

METHODOLOGIES FOR FPGA IMPLEMENTATION OF FINITE CONTROL SET MODEL PREDICTIVE CONTROL FOR ELECTRIC MOTOR DRIVES

BY

ALEX LAO, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2019) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	Methodologies for FPGA Implementation of Finite Control Set Model Predictive Control for Electric Motor Drives
AUTHOR:	Alex Lao B.Eng., (Computer Engineering) McMaster University, Hamilton, Ontario, Canada
SUPERVISORS:	Dr. Nicola Nicolici Ph.D. (University of Southampton) Dr. Ali Emadi Ph.D. (Texas A&M University)
NUMBER OF PAGES:	xviii, 120

Dedicated to those pushing the boundaries of knowledge, design, and manufacturing to conquer humanity's greatest challenges and to the family and friends that make it all possible.

Abstract

Model predictive control is a popular research focus in electric motor control as it allows designers to specify optimization goals and exhibits fast transient response. Availability of faster and more affordable computers makes it possible to implement these algorithms in real-time. Real-time implementation is not without challenges however as these algorithms exhibit high computational complexity. Field-programmable gate arrays are a potential solution to the high computational requirements. However, they can be time-consuming to develop for. In this thesis, we present a methodology that reduces the size and development time of field-programmable gate array based fixed-point model predictive motor controllers using automated numerical analysis, optimization and code generation. The methods can be applied to other domains where model predictive control is used. Here, we demonstrate the benefits of our methodology by using it to build a motor controller at various sampling rates for an interior permanent magnet synchronous motor, tested in simulation at up to 125 kHz. Performance is then evaluated on a physical test bench with sampling rates up to 35 kHz, limited by the inverter. Our results show that the low latency achievable in our design allows for the exclusion of delay compensation common in other implementations and that automated reduction of numerical precision can allow the controller design to be compacted.

Acknowledgements

I would like to thank my co-supervisors Dr. Nicola Nicolici and Dr. Ali Emadi not just for their mentorship throughout my graduate studies but during my time as an undergraduate student as well. Dr. Nicolici introduced me to digital logic design, pushed me to solve challenging problems, and helped find me an opportunity to learn more and travel as part of an internship at Altera in San Jose, California. I cannot thank Dr. Emadi enough for being a faculty advisor of the McMaster Formula Hybrid and McMaster Engineering EcoCAR3 teams that I have had the opportunity to be a part of throughout my undergraduate studies. Being able to solidify the knowledge I gained in courses within a practical setting has been invaluable.

This work would not have been possible without the help of Dr. Shamsuddeen Nalakath who's doctoral work on control algorithms served as a basis for this work. Dr. Nalakath always found time to answer my questions, no matter how busy he was. The amount that I have learned from him and his work cannot be overstated, and I am grateful for his dedication.

Thank you to all the researchers and students in the labs I have been a part of and surrounding labs for sharing their projects, asking for help and helping me out over the years. Discussing ideas in our vastly different, yet similar projects have been great. An extra thank you to Aaron Pitcher for taking the time to review my electrical designs and Jing Zhao for her help during testing while we shared the test setup.

Many of the skills I have gained over the years is thanks to my colleagues at previous internships positions I have held at Design and Integration, Canada's Wonderland, Altera, Intel, Clearpath Robotics, and Kepler Communications. Together, we have worked on so many different things. From industrial machines, roller coasters, memory interfaces, robots, to satellite payloads. Each person I have run into has taken the time and patience to share their craft and teach me how to design, build or care for another one of humanity's inventions. They have given me the confidence to make anything no matter how complex and daunting it may seem at first.

I must thank our undergraduate capstone mentor Dr. Hubert deBruin and the members of my project team, Christina Riczu, Emilie Corcoran and Thomas Phan for their dedication during that project. It was a lot of fun and I learned a lot from everyone involved.

Thank you to the undergraduates I have had the pleasure of teaching and mentoring. Their patience, hard work, curiosity, and inquisitive questions have made teaching not only great fun but an excellent learning experience as well.

We would like to acknowledge CMC Microsystems for the provision of products and services that facilitated this research, including workstations and CAD tools.

This research would not have been possible without the support of the Ontario Graduate Scholarship Program and the Canadian Graduate Scholarship-Master's Program.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Notation and Abbreviations

Notation - Electrical and Mechanical

abc	Three-phase stationary frame of reference
dq	Two axis rotating frame of reference
I_a, I_b, I_c	Current components in the abc frame of reference
I_{abc}	Current vector in the abc frame of reference
I_d, I_q	Current components in the dq frame of reference
I_{dq}	Current vector in the dq frame of reference
L_d, L_q	Inductance components in the dq frame of reference
λ	Permanent magnet flux linkage
ω	Angular Velocity
R_s	Series Resistance
T_s	Sampling Time
θ	Angular Position
V_{abc}	Voltage vector in the abc frame of reference
V_d, V_q	Voltage components in the dq frame of reference
V_{dq}	Voltage vector in the dq frame of reference

Notation - Optimization and Code Generation

c_{rate}	Cooling rate in the simulated annealing process
F	Fixed-point format of a node in Q format
FV	Fixed value
IU	Input uncertainty interval
OP	Operator
PMV	Post-multiply value
PSU	Pre-shift uncertainty interval
PSV	Post-shift value
$Q_{F_a.F_b}$	Fixed-point format with an integer and fractional word-length of F_a and F_b respectively
R	Range interval
RV	Real value
T	Temperature in the simulated annealing process
U	Uncertainty interval
UR	Uncertainty requirement interval
$\dot{x}, \ddot{x}$	Node property x of the first and second operand referenced from the operator node
$[\underline{X}, \overline{X}]$	Interval X with lower bound $\underline{X}$ and upper bound $\overline{X}$
$x_{integer}$	Integer variable named x
x_{real}	Real variable named x

Abbreviations

AA	Affine Arithmetic
AC	Alternating Current
ADC	Analog to Digital Converter
BFM	Bus Functional Model
DC	Direct Current
DFG	Data Flow Graph
DSP	Digital Signal Processors or Digital Signal Processing
DTC	Direct Torque Control
FCS-MPC	Finite Control Set Model Predictive Control
FIFO	First in First out
FOC	Field-Oriented Control
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FWL	Fractional Word-Length
GA	Genetic Algorithm
HDL	Hardware Description Language
HIL	Hardware in Loop
HLS	High-Level Synthesis
IA	Interval Arithmetic
IPMSM	Interior Permanent Magnet Synchronous Motor
IWL	Integer Word-Length
LSB	Least Significant Bit
LUT	Lookup Table

MCU	Microcontroller Unit
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MPC	Model Predictive Control
MSB	Most Significant Bit
PC	Personal Computer
PI	Proportional Integral
PMSM	Permanent Magnet Synchronous Motor
RMSE	Root Mean Square Error
RPM	Revolutions Per Minute
RTL	Register Transfer Level
SA	Simulated Annealing
SMT	Satisfiability Modulo Theories
TS	Tabu Search
USB	Universal Serial Bus
WL	Word-Length

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Electric motors are widely used in our modern world and market growth is expected. The amount of electricity used in the industrial sector by electric motors is over 50% of the total electrical energy use in industry, this shows a high demand for motors (Schwartz *et al.* (2017)). Between 2016 and 2017 sales of electric vehicles increased by 54%, showing market growth and an increase in demand for electric motors (International Energy Agency (2018)). The existing market and growth highlight the importance of high-performance electric motor solutions in our world. These markets demand advanced solutions catering to their specific needs, they require methodologies to reduce time to market, decrease cost and increase performance. In this chapter, we introduce electric motors, inverters and the controllers required to build high-performance electric motor drives. We also introduce some details involved in the design process. After, we provide a summary of thesis objectives before providing a chapter summary.

1.1 Electric Motors, Control, and Implementation

Many individual components are required to build an electric drive. Generally, we require the motor, power electronics, control algorithm, computer, as well as, design tools and methodologies. These components are used to build systems that take user demands and convert them into control actions that satisfy the users' requirements in real-time.

Electric motors convert electrical to mechanical power. Motors often require a controller that implements a control method that precisely regulates power, torque, speed, and position. Different control methods influence how controlled variables deviates away from the desired values. Better regulation may result in a system with lower torque ripple, less noise or higher efficiency.

An inverter is a power electronic component that is sometimes required to provide AC power to a motor from a DC source or AC power from an unsuitable AC source with a different voltage and frequency. For synchronous motors, the frequency of the input power must be synchronized to the rotation of the motor. A task an inverter is well suited for with the help of sensors and a controller.

Controllers can be implemented using digital computers such as microprocessors or microcontrollers. This implementation method is not new and has been used since the 1980s (Gabriel *et al.* (1980)). A control algorithm, implemented on a computer, manipulates numbers representing data sampled from sensors to make control decisions. Numbers can be represented in many ways on a computer, resulting in different amounts of rounding errors internally, as well as a difference in speed, size, and cost of the resulting controller. While rounding reduces size and cost, too much rounding

error may change control decision and result in unacceptable performance degradation and instability (Shien-Ru Ko and Wen-Shyong Yu (2000)). Therefore, controller designers might want to limit the amount of uncertainty within their controller implementation. Methods and tools exist to analyze uncertainty in computers and can be applied to controller development (Darulova and Kuncak (2013)).

Different types of computers exist, such as microcontrollers, digital signal processors, and programmable logic. The choice results in varying levels of computational performance. Faster computers may be able to make more calculations or decisions per second resulting in the designers' goals being met more effectively. However, more exotic computing platforms may cost more or take longer design for.

A designer might not want to be involved with all the details in designing every aspect of the system. Tools can be developed to automate part of the design process. Equations required in a controller can be represented inside of tools which can manipulate the design to optimize for a user-configurable uncertainty limit with optimized size and cost. Code can then be generated, ready to be integrated into a larger design.

Predictive control is a hot topic in power electronics and motor control (Rodriguez *et al.* (2013)). In predictive control, equations are used to predict the behavior of a system. The predictions are used to influence control decisions while trying to meet the behavior characteristics desired by the designer. Predictive control enables the design of a controller that allows for arbitrary control goals to be set by the designer. They can also respond quickly to changing demands. However, their computational requirements often require designs to be implemented on fast digital computers to meet real-time constraints (Kosan *et al.* (2018)).

1.2 Thesis Objectives

There are two major objectives in this thesis. The first objective is to develop and demonstrate methods to partially automate the creation of a prediction based motor controller using programmable logic that can make millions of predictions per second. The second objective is a demonstration of how that workflow can be applied to the development of a motor controller and then show cost savings associated with using such a design process. Then we show the performance degradation and caveats that come with these savings.

Along the way, we describe all the steps from design methodology to physical realization. We also describe the simulation environments useful for development and performance evaluation purposes. The behavior characteristics, performance metrics, and figures of merit of multiple controller variants implemented using our methodologies will then be evaluated using these simulation environments and the physical setup.

1.3 Thesis Organization

This chapter introduced the importance of electric motor drives and the high-level details involved with the implementation of such systems before introducing the concept of predictive control. After, we summarized the objectives of this thesis.

In Figure 1.1 we show where the topics we introduced fit into the electric drive development process or resulting system.

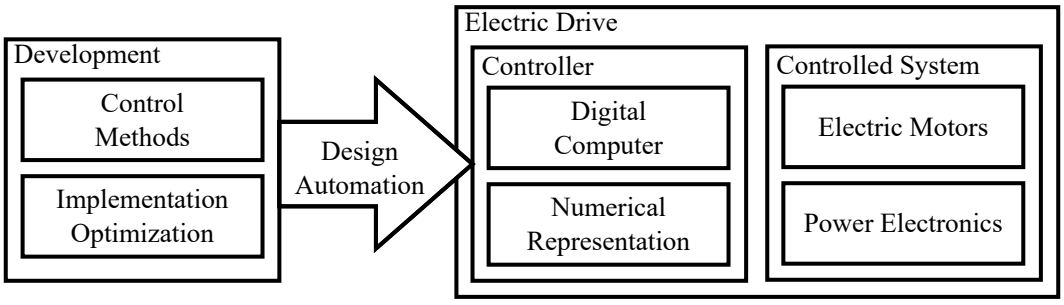

Figure 1.1: Components of an electric drive development process and product

A summary of the literature in this field and a brief introduction to the major topics of this thesis will be presented in Chapter 2.

Chapter 3 details our implementation tools and methodologies for creating digital circuits for evaluation of equations such as those required for predictive control.

Chapter 4 demonstrates usage of our tools combined with traditional design techniques through implementing a predictive control algorithm for an electric drive. The associated required electronics and simulation environments will also be detailed.

Chapter 5 is where we present our results. Our findings may also interest those developing motor controllers without the methods presented in Chapter 3.

We then conclude by summarizing our findings and present ideas for future work in Chapter 6.

Chapter 2

Background and Literature Review

One of our goals is to develop tools and methodologies for predictive motor controller design that uses programmable logic. When building a predictive motor controller, the specific details of every part of the design can greatly influence system performance and where effective optimizations can be made across the implementation, therefore we must gain a deeper understanding of all the details involved. Furthermore, we want to partially automate the design process, choices made when developing the automated tools restrict the design space explored by the tool.

In this chapter, we provide a literature review and provide background on a wide range of topics related to electric drive development. We cover topics including electric motors, inverters, and control algorithms. Then we look at implementation platforms for the control algorithms before covering analytical methods and methodologies for design automation related to programmable logic. We then look at works in electric motor control and other fields that involve implementation details and analysis methodologies related to ours.

2.1 Electric Motors

Electric motors convert electrical to mechanical power by creating torque using magnetic field interactions (Krishnan (2001)). Many types of electric motors exist, common types include brushless DC, induction, and permanent magnet synchronous motors (PMSM). Due to availability of a physical test setup, we will focus on the control of an interior permanent magnet synchronous motor (IPMSM) to evaluate our controller implementation methodology with the understanding that similarities in design requirements will make the methodology reusable with adaptation.

2.1.1 Interior Permanent Magnet Synchronous Motor

An IPMSM is a variant of PMSMs with the magnets embedded inside the rotor. As this is a synchronous motor, the frequency of the power must be synchronized to the rotation of the motor. This means a controller in conjunction with power electronics to generate the variable frequency voltage waveform is often required (Krishnan (2001)). PMSMs are often smaller, more efficient and have less rotating mass which allows for faster response when compared to induction motors which may encourage some to select this motor type (Pillay and Krishnan (1991)). Discrete models suitable for use in a predictive controller can be commonly found (J. Rezaie (2007)).

2.2 Power Electronics

Power electronics are used to perform electric power conversion between the voltage and current from the supply to what is required by the load (Ned Mohan (1995)). As Mohan explains, AC power from the electric utility grid often needs to be converted into DC power for use by our electronic devices, this conversion is known as rectification. In electric vehicles, power for the motor might come from a DC source such as a battery. Therefore, an inverter is used to drive the AC motor commonly used in these vehicles.

Many more types of power electronic converters exist but most use electronic switches such as metal oxide field-effect transistors (MOSFETs) or insulated gate bipolar transistors in conjunction with switch control signals known as gating signals from a controller to perform their task (Ned Mohan (1995)). The higher switching frequency can result in better power electronics or motor performance but will result in more switching losses (Shirabe *et al.* (2014)). The higher switching frequencies without excessive switching loss can be made possible by using new transistor materials such as gallium nitride (Shirabe *et al.* (2014)). However, controllers generating the gate signals also need to perform better to make all the control decision in real-time.

2.2.1 Motor Inverters

Motor inverters are used to power an electric motor from a DC source, they generate the voltages and currents required to drive the AC motor and allow for torque and speed control. A typical inverter fed motor drive is shown in Figure 2.1. It includes an inverter, controlled by a controller with position and current feedback. The inverter, shown in Figure 2.2, generates the three-phase AC power required by the motor from a DC source.

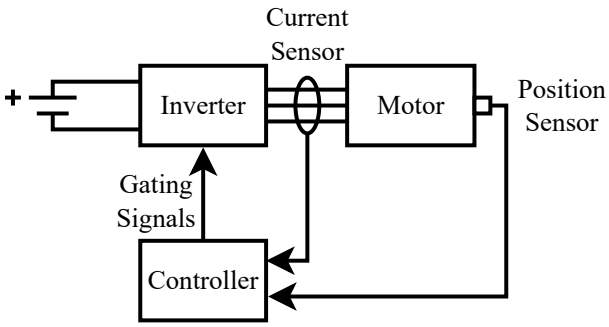

Figure 2.1: A typical inverter fed drive

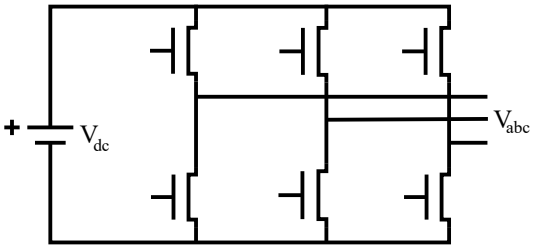

Figure 2.2: A three-phase inverter

2.3 Electric Motor and Inverter Control Methods

An electric drive that exhibits high-performance in a system often requires advanced control techniques implemented in real-time. Many control techniques exist for PMSMs such as field-oriented control (FOC), direct torque control (DTC), and model predictive control (MPC) (Wang *et al.* (2011)). Each of these control methods exhibits different characteristics that may be desired by the designer and must be chosen for their specific application. These control methods are used to generate the gating signals for the inverter. Each set of valid gating signals constitutes an inverter switch state. Methods for gate signal generation will be covered in this section.

2.3.1 Coordinate Transformations

Three-phase power is generated by a rotating magnetic field. Therefore, three-phase power can be seen as rotating. By rotating our frame of reference with the rotation of the motor we can generate two DC components (dq) from three AC components (abc). Using these DC equivalent values in our models can simplify them. In a PMSM, the stator current can be split into d and q axis components using this method (Wang *et al.* (2011)). The d axis current (I_d) represents the magnitude of the magnetic field in the air gap between the stator and rotor and the q axis current (I_q) produces useful torque (Wang *et al.* (2011)). This transformation is performed using a Clarke and Park transform (Krishnan (2001)).

2.3.2 Control Methods

The control method selected to regulate the torque output of a motor result in different transient response times, steady-state ripple, inverter switching frequency, and computational load. All control methods require low latency as delayed control response adversely affects performance (Shien-Ru Ko and Wen-Shyong Yu (2000)).

Field-oriented control (FOC), also known as vector control, utilizes regulators such as linear PI controllers to regulate the I_d and I_q currents (Wang *et al.* (2011)). These linear controllers produce a continuously variable output voltage request that is met using a pulse width modulator which results in a fixed switching frequency.

Direct torque control (DTC) is a method that controls the motor torque by estimating the rotor torque and stator flux before selecting an inverter switch state based on this estimate using a hysteresis controller (Wang *et al.* (2011)). A modulator is

not used and switching frequency can vary but it can be regulated by changing the hysteresis thresholds.

A model predictive controller (MPC) uses predictions to select a state that results in the best response as evaluated by a cost function (Rodriguez *et al.* (2013)). Finite control set model predictive control (FCS-MPC), a variant of MPC presents some benefits over FOC and DTC for motor control such as fast transient response and the ability to optimize for any output variable using a cost function. If such a cost function involves I_{dq} currents, then the behavior can be like FOC but without the need for linear controllers that require tuning. Consequently, if the cost function is based on torque and stator flux then the behavior can be like DTC. For a PMSM, FCS-MPC has been demonstrated to exhibit less flux and torque ripple then FOC or DTC (Wang *et al.* (2011)). Unlike with FOC or DTC however, the switching frequency is not controlled which may become a problem as we will see our test results.

FCS-MPC is well suited for motor control because an inverter has a finite set of legal switch states. FCS-MPC evaluates the prediction equations for every valid switch state over a prediction horizon. It then evaluates the trajectory using a cost function to select the best switch state to use. At the next time step, all the predictions will be repeated with new sensor data as model inaccuracies and external disturbances will result in a different response then the one predicted. More switching states, longer prediction horizons, and higher sampling frequencies may be required to meet performance targets, however (Rodriguez *et al.* (2013)). This increases the computational complexity, posing a challenge for real-time implementations.

In many implementations of FCS-MPC where computational latency is high enough to be close to the sample time, a compensation step can be applied to make up for the

delay (Yang *et al.* (2017)). The previously selected state is applied to the system at the start of the next time step and the outcome is predicted before performing all the predictions within the horizon. Once the predictions are complete and the next state is selected the controller waits until the start of the next time step before applying it.

An illustration of the internal operation of an FCS-MPC algorithm without delay compensation is shown in Figure 2.3. With delay compensation, the predictions are shifted forward from K to K+1 and extend to K+4 instead of K+3. The state used at K would have been decided in the previous time step.

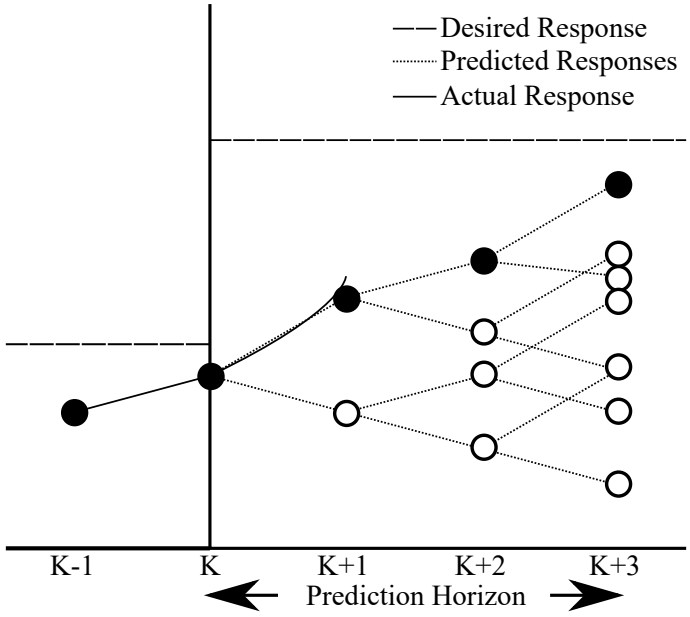

Figure 2.3: FCS-MPC with two valid states and a prediction horizon of three

2.4 Digital Computers

Digital computers are often used to implement controllers due to their low cost and re-programmability. All control methods must be implemented in real-time, which means processing must be performed within time limits or system performance degradation or failure will occur. Options available to designers building a controller include microcontrollers (MCUs), digital signal processors (DSPs) and field-programmable gate arrays (FPGAs).

2.4.1 Microcontrollers and Digital Signal Processors

MCUs and DSPs are devices that execute software instructions to carry out processing and control tasks. MCUs are often seen as general-purpose controllers and DSPs are seen as specialized controllers for tasks that require processing that would overburden an MCU. However, the line between these two device types has blurred over time as MCU architectures have gained more digital signal processing abilities and DSP architectures have become better at more general tasks. Therefore, we can lump them into a single category, software processors that execute instructions. Software processors have the benefit that they can be very low-cost and easy to develop for. Unfortunately, they can usually only do one operation at a time. Multi-core devices exist where multiple instructions can be executed at the same time to increase parallelism, but their parallelism is still very limited. This can be a problem when you require high-throughput and low latency. This is an underlying reason why many implementations of FCS-MPC require a delay compensation step.

2.4.2 Field-Programmable Gate Arrays

FPGAs are a type of programmable logic device that contains at a minimum, inputs and outputs, lookup tables (LUTs), registers and a routing network (Brown *et al.* (1996)). LUTs are used to implement combinational logic circuits, registers are used to create sequential logic when used to segment the combinational logic, and the routing network is used to connect all the logic, inputs and outputs. FPGAs may also have multipliers integrated on the chip so less of the more generic logic resources are used when implementing multiplication. An $m \times m$ hard multiplier can work with two, m bit inputs. Figure 2.4 shows a simple FPGA architecture that includes LUTs, registers and multipliers.

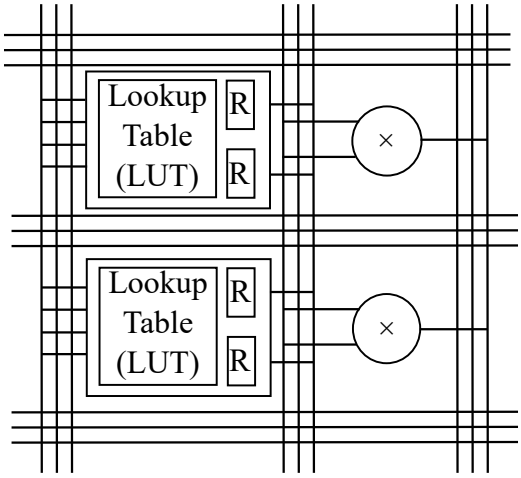

Figure 2.4: Simple FPGA architecture

FPGAs are an option that enables either high throughput, low latency or a combination of both for digital designs. From networking (Lockwood *et al.* (2007)), to compute acceleration (Che *et al.* (2008)) and computer vision (Jin *et al.* (2010)), FPGAs provide the computational power for demanding tasks.

In the context of control systems for power electronics and drive application, parallelism in FPGAs enable high-performance designs that can allow for high switching frequencies exceeding one MHz (Monmasson *et al.* (2011)). Quasi-instantaneous execution is highlighted as a benefit of FPGA based controllers when compared to MCU or DSP implementations of control algorithms by Monmasson et al. (Monmasson and Cirstea (2007)).

Parallel processing on an FPGA means algorithms can potentially be accelerated greatly when implemented on FPGAs instead of a software processor. Designs can be built in such a way as to have extremely predictable and low latency which can enable us to build an FCS-MPC without delay compensation.

Designs for FPGAs are often written in a hardware description language (HDL) such as Verilog (IEEE Computers Society (2006)), SystemVerilog (IEEE Computers Society (2018)) or VHDL (IEEE Computers Society (2009)). Using these HDLs, logical expressions and data transfer between registers are described, this is known as a register transfer level (RTL) description. Controller implementation on FPGAs often requires more time and effort than on software processors which must be addressed before FPGAs can be used in more designs.

FPGA area utilization is a critical cost metric as larger FPGAs cost more, require more power, and may take up more circuit board area. Techniques to estimate FPGA area include lookup tables, equations, and curve fits (Deng *et al.* (2008)).

2.5 Numerical Formats in Digital Computing

Numerical algorithms can generally be implemented in two ways, with floating-point or fixed-point arithmetic. Both are examples of finite precision arithmetic. Floating-point units are more complex to implement in digital systems while fixed-point units are simpler. All methods of storing decimal numbers digitally result in different range, precision limitations, as well as, uncertainty characteristics.

2.5.1 Floating-Point Representation

Floating-point arithmetic is often used within the science and engineering community as it allows the representation of large and small numbers in binary. Floating-point representation of real values is like scientific notation where a normalized value is scaled by an exponent. In scientific notation, the base of the exponent is typically ten while in floating-point representation it is two. This method of storing values on a computational platform results in large values being stored with low precision and small values being stored with high precision. Typically, the IEEE-754 standard is implemented to support floating-point arithmetic on digital computers (IEEE Computers Society (2008)).

2.5.2 Fixed-Point Representation

Fixed-point arithmetic may be preferred over floating-point because the digital circuit required to operate on them is much simpler. However, because the radix point does not move on its own, the range of representation is limited by the integer word-length (IWL), and the precision limited by the fractional word-length (FWL). Appropriate

word-length (WL) must be allocated appropriately to prevent integer overflow or excessive precision loss. Shorter WLs require less logic to implement, a benefit for all implementations of computers.

Focusing on signed values using twos complement, the WLs of a fixed-point format can be expressed in $Qa.b$ format where a and b is the IWL and FWL respectively (Erick L. Oberstar (2007)). To convert a value from real to fixed-point it must be scaled and then truncated to zero decimal places as shown in Equation 2.1. Truncation causes quantization error which results in some precision loss. Therefore, the result of the conversion is an integer value that represents the real value with some error.

$$x_{integer} = Truncate(x_{real} \times 2^b) \tag{2.1}$$

Conversion of real values to fixed-point requires that the resulting integer resides in the interval shown in Equation 2.2 or overflow will occur.

$$x_{integer} = \Big[-2^{a+b}, 2^{a+b} - 1 \Big] \tag{2.2}$$

Also discussed by Oberstar, addition, and subtraction of fixed-point values require that both operands have the same FWL. If the operands have different FWLs then the radix point must be aligned before the operation can be performed. The result of the operation will have an IWL one higher than the IWL of the operand with the largest IWL. Multiplication does not require that the radix point be aligned. The product will have an IWL and FWL that is the sum of the two operands IWL and FWL respectively. This WL growth often means the results or operands in the implementation may have to be truncated or logic utilization will increase too much.

2.6 Fixed-Point Word-Length Optimization

Optimization of fixed-point WLs in implementations can be done automatically using searching and optimization methods in conjunction with uncertainty, range, complexity, and cost calculators and estimators (Han *et al.* (2006), Dong-U Lee *et al.* (2005)).

Automatic data type conversion and optimization of word-length is available in MATLAB (The MathWorks, Inc. (2019a)), however their use of a simulation method for determining if range and precision bounds are met means their results cannot be guaranteed (Osborne *et al.* (2007)).

In this section, we review the available methods for searching, range estimation, precision analysis and cost evaluation that are often used for WL optimization. Many combinations of any of these methods can be used for WL optimization.

2.6.1 Meta-Heuristic Algorithms

Meta-heuristic algorithms are designed to generate potential solutions within an enormous design space to find good solutions, possibly even ones that are close to optimal. They do so by using rule sets that guide it towards better solutions. Popular meta-heuristics include genetic algorithms (Holland (1992)), simulated annealing (Kirkpatrick *et al.* (1983)), and tabu search (Glover (1986)).

Genetic Algorithms

Genetic algorithms (GAs) are inspired by the process of adaptation and evolution in the natural world (Holland (1992)). Evolution of the solution is achieved through genetic operators, crossover, inversion, and mutation. Solutions are then evaluated using a fitness function.

Simulated Annealing

Simulated annealing (SA) mimics the process of metallurgical annealing (Kirkpatrick *et al.* (1983)). Small random changes are made to the solution to generate neighbouring ones. Probability of accepting a worse solution is governed by the temperature of the system which decreases over time. A slowly decreasing probability of accepting worse solutions gives the meta-heuristic a chance at finding better local minima in the global search space.

Tabu Search

Tabu search (TS) performs local search and accepts a worse solution if a better one does not exist (Glover (1986)). It lowers the probability of revisiting a solution by using memory. TS, when compared to SA and GA searches the entire neighbouring space for the best solution before accepting a worse one. This can have a significant impact on run time, preventing good coverage of the global search space if solutions have a lot of variables.

2.6.2 Range and Precision Analysis

Simulation and analytical approaches exist to estimate the maximum and minimum range of values and their uncertainty if the input range and uncertainty is known for algebraic expressions (Han *et al.* (2006)). We need to analyze the range and uncertainty to efficiently implement the equations necessary for coordinate transformations and predictions in a controller. Range and uncertainty can be analyzed using interval arithmetic, affine arithmetic or satisfiability modulo theory analysis.

Interval and Affine Arithmetic

Interval arithmetic (IA) and affine arithmetic (AA) can be used to reliably calculate the range and uncertainty bounds of finite precision arithmetic (Stolfi and Henrique De Figueiredo (1998)). These methods are used in Gappa (Melquiond (2019)), among others (Linderman *et al.* (2010), Dong-U Lee *et al.* (2005), Osborne *et al.* (2007)). One issue with IA is that it does not capture the relationships between variables where range or uncertainty is reduced due to variables partially or fully canceling out (Fang *et al.* (2003)). AA is a similar method where relationships between variables are tracked, therefore AA produces tighter bounds then IA but is more complex to implement, but still provides a pessimistic result (Stolfi and Henrique De Figueiredo (1998)). Both of these methods are analytical methods that produce bounds that will hold with certainty under the assumption that the initial input bounds hold.

A detailed explanation of how IA can be used to store and compute these bounds on finite precision arithmetic operations is provided by Stolfi and Henrique de Figueiredo which we will summarize here. IA consists of interval extension operations that correspond with arithmetic operations including truncation. We begin by representing the range and uncertainty of our input variables as intervals as shown in Equations 2.3.

$$x = \left[\underline{x}, \overline{x}\right] \tag{2.3a}$$

$$\delta x = \left[\underline{\delta x}, \overline{\delta x}\right] \tag{2.3b}$$

Addition of intervals to compute range or uncertainty is simply performed using Equations 2.4.

$$x + y = \left[\underline{x} + \underline{y}, \overline{x} + \overline{y}\right] \tag{2.4a}$$

$$\delta(x + y) = \left[\underline{\delta x} + \underline{\delta y}, \overline{\delta x} + \overline{\delta y}\right] \tag{2.4b}$$

Similarly, for subtraction we use Equations 2.5.

$$x - y = \left[\underline{x} - \overline{y}, \overline{x} - \underline{y}\right] \tag{2.5a}$$

$$\delta(x - y) = \left[\underline{\delta x} - \overline{\delta y}, \overline{\delta x} - \underline{\delta y}\right] \tag{2.5b}$$

Equations 2.6 are used to compute bounds on multiplication. They are more complicated as uncertainty is affected by range.

$$a = \left\{\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy}\right\} \tag{2.6a}$$

$$b = \left\{(\underline{x} + \underline{\delta x})(\underline{y} + \underline{\delta y}), (\underline{x} + \underline{\delta x})(\overline{y} + \overline{\delta y}), (\overline{x} + \overline{\delta x})(\underline{y} + \underline{\delta y}), (\overline{x} + \overline{\delta x})(\overline{y} + \overline{\delta y})\right\} \tag{2.6b}$$

$$x \times y = \left[min(a), max(a)\right] \tag{2.6c}$$

$$\delta(x \times y) = \left[min(b) - min(a), max(b) - max(a)\right] \tag{2.6d}$$

When working with intervals that represent uncertainty, errors incurred through conversion from real to fixed-point or from one fixed-point format to another can be achieved by expanding the uncertainty interval as shown in Equation 2.7 when converting real to fixed, or Equation 2.8 when converting from a fixed-point format with a larger FWL to a smaller one. No error increase is incurred when the FWL is expanded.

$$\delta(x) = \left[-2^{-b}, 2^{-b} \right] \tag{2.7}$$

$$\delta(x) = \left[\underline{x} - 2^{-b_{new}} + 2^{-b_{old}}, \overline{x} + 2^{-b_{new}} - 2^{-b_{old}} \right] \tag{2.8}$$

As we do not use AA in our work, we will not discuss it farther. However, it should be noted that AA can be used in substitution of IA.

Satisfiability Modulo Theories

Satisfiability modulo theory (SMT) solvers are constraint solvers that can work with equations that contain reals and integers making them suitable for analyzing differences between real and fixed-point implementations of algorithms. With a suitable constraint set, they can prove even tighter range and uncertainty bounds then AA but due to the long run-times associated with SMT solvers, they may not be suitable for analyzing large equations without an incremental approach (Eldib and Wang (2014)). Many SMT solvers are available such as Z3 (Microsoft Research (2019)), Yices (SRI International (2018)), and CVC4 (Barrett *et al.* (2011))), among many others.

In this thesis, integer and real SMT variables are denoted with subscript integer and real respectively.

SMT solvers prove if a set of constraints is satisfiable or not. For example, the constraint set shown in Equation 2.9 is satisfiable.

$$(0 \leq x_{real} \leq 5) \wedge (y_{real} = 2 \times x_{real}) \wedge (y_{real} \equiv 7.2) \tag{2.9}$$

However, the one shown in Equation 2.10 is not because an integer multiple of an integer cannot be a decimal.

$$(0 \leq x_{integer} \leq 5) \wedge (y_{real} = 2 \times x_{integer}) \wedge (y_{real} \equiv 7.2) \tag{2.10}$$

2.7 Model-Based Design

Model-based design is a design entry technique often used to build simulations (The MathWorks, Inc. (2019b)). The technique involves connecting blocks to build mathematical and logical expressions. It can also be used to implement controllers on MCUs, DSPs, and FPGAs through code generation.

For simulation, the model can be executed in the design environment where the user can plot and collect data or code generation can be used to generate modular blocks that can be integrated into other simulation tools.

For implementation, code can be generated to target software processors or FPGAs. The code generation process, when targeting FPGAs is also known as high-level synthesis.

2.8 High-Level Synthesis

High-level synthesis (HLS), is one method of reducing the effort required when developing for FPGAs through automatic code generation from higher-level descriptions of algorithms. HLS is the process of taking a data flow or algorithm and generating HDL code for logic synthesis (Borriello and Detjens (1988)). HLS can include lexical analysis, data flow analysis, scheduling and resource allocation (Meredith (2004)).

An example HLS flow that works with algebraic equations is shown in Figure 2.5. In the example, an input file is parsed and a data flow graph (DFG) is built. Then, a scheduling algorithm is used to choose the timestep to execute operations. The resource binder then counts the maximum number of each resource needed each time step before allocating reusable units to perform the required operations.

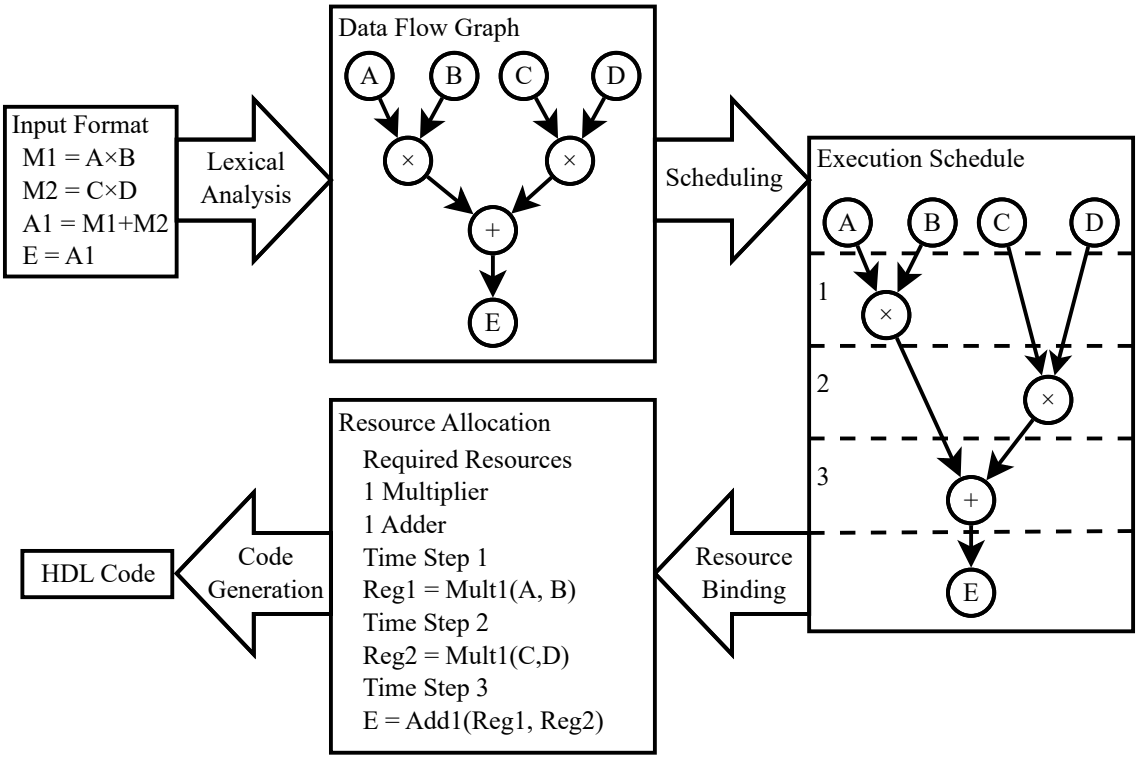

Figure 2.5: An illustration of a simple HLS flow

Extensive research on each step involved in HLS including static schedulers (Paulin and Knight (1989)), dynamic schedulers (Lapotre *et al.* (2013)), FPGA area aware resource sharing (Casseau and Le Gal (2009)) and WL optimization (Ye and Kapre (2014)).

Today, many commercial products are available including Vivado HLS (Xilinx Inc. (2019)), High-level Synthesis Compiler (Intel Corporation (2019a)), Synphony (Synopsys, Inc. (2009)), Catapult (Mentor, a Siemens Business (2019a)), and Stratus (Cadence Design Systems, Inc. (2019b)), among others. Also, tools within science and engineering suits such as LabView FPGA (National Instruments (2019b)) and HDL Coder for MATLAB and Simulink (MathWorks Inc. (2019a)) exists to combine model-based design with HLS.

2.9 System Simulation

Simulation of physical systems is often performed using general-purpose simulation packages such as Simulink (The MathWorks, Inc. (2019c)), LabVIEW (National Instruments (2019c)) or MapleSim (Maplesoft (2019)), among many others. Often these tools are configured using a scripting language or using blocks in a model-based design environment. Control algorithms can be modeled within these environments allowing for algorithm development.

FPGA simulation environments allow designers to work on designs without the hardware and provide greater design visibility by allowing all signals to be available for analysis. Commercial tools for FPGA simulation include ModelSim (Mentor, a Siemens Business (2019b)), Incisive (Cadence Design Systems, Inc. (2019a)), and

VCS (Synopsys, Inc. (2019)), among others. Modeling of components outside of the FPGA must be provided using bus functional models (BFMs).

Co-simulation between general-purpose simulation packages and FPGA simulators is provided by tools such as HDL Verifier for Simulink (MathWorks Inc. (2019b)) and LabVIEW FPGA (National Instruments (2019b)). This can be used to validate the behavior of an FPGA based controller. As an alternative, using the code generation facilities in some simulation packages we can generate BFMs to be integrated into our FPGA simulation testbench if a co-simulation environment is not desired. This type of functionality can also be found in the same co-simulation tools.

Hardware in loop (HIL) simulation is the process of testing a controller implemented on the real hardware with an emulator instead of the physical test setup. In the context of FPGA design, this testing exposes problems associated with inaccurate BFMs since the actual hardware is used. HIL support can often be found within the same simulation packages as we mentioned previously. Platforms for HIL simulation is available from dSpace (dSPACE GmbH (2019b)), National Instruments (National Instruments (2019a)), and others.

2.10 Related Works and Inspiration

Many pieces of related work exist that combine different combinations of topics we discussed in this chapter. We draw inspiration from these works to create a methodology useful for integration into an FCS-MPC controller design process.

We discussed that analytical methods and SMT solvers can be used to prove bound on range and uncertainty. The Z3 SMT solver was used previously to refine IA or AA results (Duralová (2014)). Similar techniques can be found in Rosa which is a

tool that compiles software programs with reals into finite precision implementations using IA, AA and SMT solvers (Darulova and Kuncak (2013)). A similar project uses the LLVM compiler framework (Lattner and Adve (2004)) and the Yices SMT solver to reduce errors and prevent overflow in fixed-point software programs (Eldib and Wang (2014)). Eldib and Wang's work also demonstrates segmentation to address SMT issues with scalability. All these works exploit the speed of IA or AA with the smaller bounds that SMT solving can provide. The authors of these works target software implementations.

Targeting FPGA implementations, MixFX-Score (Ye and Kapre (2014)) is an HLS tool that combines inspiration from the FX-Score (Martorell and Kapre (2012)) framework. It applies WL optimization by using Gappa++ (Linderman *et al.* (2010)) which performs AA. SA is used to generate candidate designs that implement SPICE circuit models. FX-Score produces homogeneous fixed-point designs where the fixed-point format is the same throughout while MixFX-Score produces heterogeneous fixed-point designs. Other custom HLS processes targeting Xilinx FPGAs with DSP48E1 that utilizes Gappa (Melquiond (2019)) to perform IA for error minimization has been demonstrated (Bajaj (2016)). While all these processes that target FPGAs have different input languages and are intended for different processing needs, they internally use similar methods.

In power electronic control, we have works where manual WL optimizations were applied to an FPGA based active front end FCS-MPC showing minor controller performance degradation but significant area utilization reduction (Hamidi *et al.* (2017)). Manually developed FPGA based FCS-MPC for induction motors exist that can make 8 predictions in 1.6 uS, potentially achieving a 625 kHz sample rate with a prediction

horizon of 1 (Kosan *et al.* (2018)). Others have gone for model-based approaches by using HLS tools within MATLAB and Simulink (MathWorks Inc. (2019a)) to produce FPGA implementations of FCS-MPC for motor control (Wendel *et al.* (2017a)), highlighting a user that can benefit from having additional optimization features in their HLS tools.

We draw inspiration from the overview of works in this section. Specifically, we draw on the analytical techniques' others have applied to software program design analysis and optimization and how others have already applied these techniques to FPGA design automation through HLS. We identify others that have applied WL optimization for FCS-MPC on FPGAs to save area, a process that others have already automated in other fields. Then we look at performance achieved in manually developed FCS-MPC implementations on FPGA for motor control highlighting that others are already looking at FPGAs to alleviate computational complexity problems when applying FCS-MPC to motor control. We then identified users of HLS for FCS-MPC development that would benefit from our methods as an additional feature in the tools they already use. We see there is an existing set of techniques that can apply to the problems being solved by researchers of FCS-MPC implementations on FPGA for motor control and from this, we developed our objectives highlighted in Section 1.2.

2.11 Chapter Summary

In this chapter, we summarized the basics and state of the art of major works related to ours. We identified where processes in other research fields can be applied to the work others are doing on FCS-MPC for motor control using FPGAs.

In Chapter 3, we will discuss the internal details of our high-level design tools for FPGA modules implementing equation evaluation. It applies WL optimizations to reduce the size of the FPGA required using a combination of SA, IA and SMT solvers so uncertainty bounds can be guaranteed. We chose to build a custom tool instead of depending on a set of existing tools because it allows better control over what is happening during the automated design process. In the future, our techniques can be incorporated into other HLS tools, increasing the efficiency of designs produced by automated design processes. We then apply these tools to the development of a predictive controller in Chapter 4.

In Chapter 4 we only target motor control using the tools and techniques we describe in Chapter 3. However, they may apply to other applications where equations must be evaluated in an FPGA such as in power electronic control.

Those implementing FPGA based controllers for power electronics or motor control without HLS or our optimization methods may still find details of our implementation in Chapter 4 and results in Chapter 5 applicable to their work.

Chapter 3

Tools and Methodologies

Our optimization tools automate word-length (WL) optimization of circuits that evaluate algebraic equations which are commonly found in FCS-MPC models. This chapter focuses on the internal operation of our tools while Chapter 4 utilizes these tools within a controller design. This chapter can be skipped for readers interested in either using, rather than developing, tools for equation modeling or those looking to avoid these types of high-level optimization tools. Techniques in Chapter 4 can still apply to those who intend to use traditional FPGA development techniques alone to design an FCS-MPC for power electronics or motor control.

Our tools target fixed-point implementation on small FPGAs with hard multipliers. WL optimization reduces the logic utilization of the circuits so smaller FPGAs can be used, reducing cost.

We first introduce how we can perform arithmetic operations in fixed-point on an FPGA. We will then discuss how we represent equations within our tools. After, we detail the analytical methods used to ensure that the fixed-point WLs are allocated

such that integer overflow does not occur and precision of the output meet requirements so long as input restrictions are respected. After, we discuss how we estimate and optimize the FPGA area utilization of the circuit generated by combining these techniques into a WL optimization and code generation tool.

While we integrate our tools into a traditional FPGA development flow as shown in Figure 3.1 the techniques we present can be integrated into more sophisticated high-level synthesis (HLS) environments or combined with their techniques in the future. Combining previous work could allow for resource sharing and pipelining, potentially resulting in even better performance with less area.

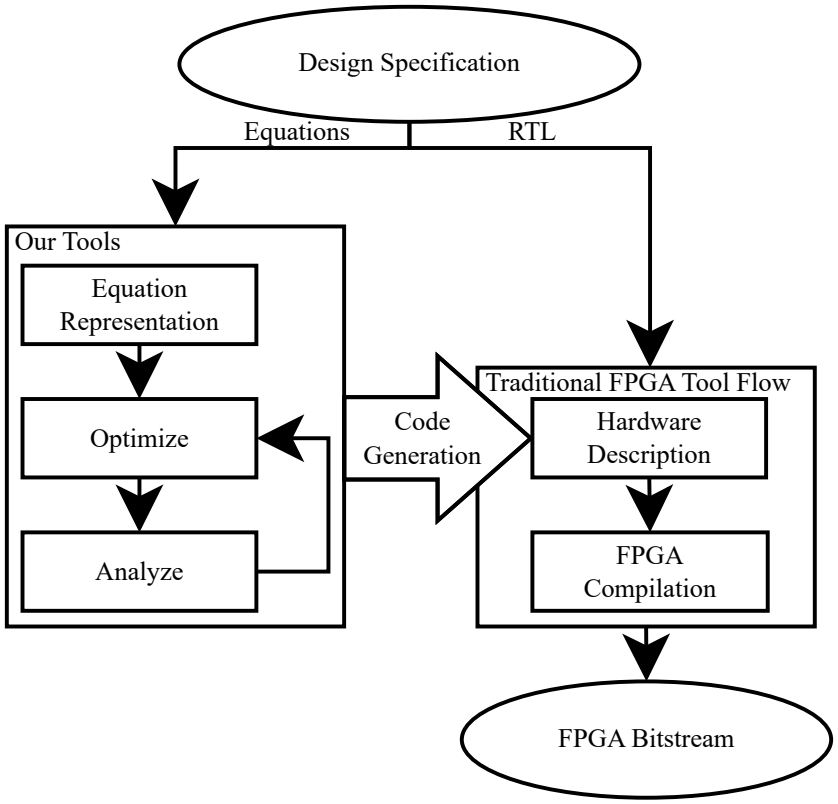

Figure 3.1: Integration of our tools into traditional FPGA tool flows

3.1 FPGA Fixed-Point Arithmetic Model

FPGA logic can be synthesized to shift, add, subtract, multiply or divide values expressed in fixed-point format. In this chapter, we show WLs in Q format as explained in Section 2.5.2. Values with longer WLs require more logic on an FPGA to store or operate on.

Additions and subtractions are often implemented using lookup tables (LUTs). The larger the WL of the operands, the more LUTs are needed.

Multiplication is often implemented using dedicated multiplier elements available in almost all FPGAs to save logic resources. Division by a constant can be replaced with multiplication by a reciprocal to save logic resources as well. Often, a division can be substituted in this manner. Therefore, we can exclude division to simplify our tool flow yet still support many modeling equations required in FCS-MPCs.

Shifting is often used in FPGA designs. Shifting by a constant is simply implemented with a change in routing on an FPGA so there is essentially no logic overhead. Multiplication or division by powers of two can be replaced with left and right arithmetic shift respectively, saving resources. Shift operations can also be used to move the radix point or to truncate or add LSBs. Right shifting or truncation of LSBs results in rounding towards zero to the new, larger quantization step while left shifting or addition of LSBs decreases the size of the quantization step.

Addition and subtraction operations on an FPGA can be broken down into three stages, the pre-shift stage to align the radix point of the operands, the arithmetic operation itself, and a truncation step if our result has an IWL longer then what we need. Figure 3.2 shows an example of an addition or subtraction operation between operands with FWLs above and below the output FWL and IWLs that result in a WL above the output IWL. In our work, we collectively call this a Q3.6 addition or subtraction node.

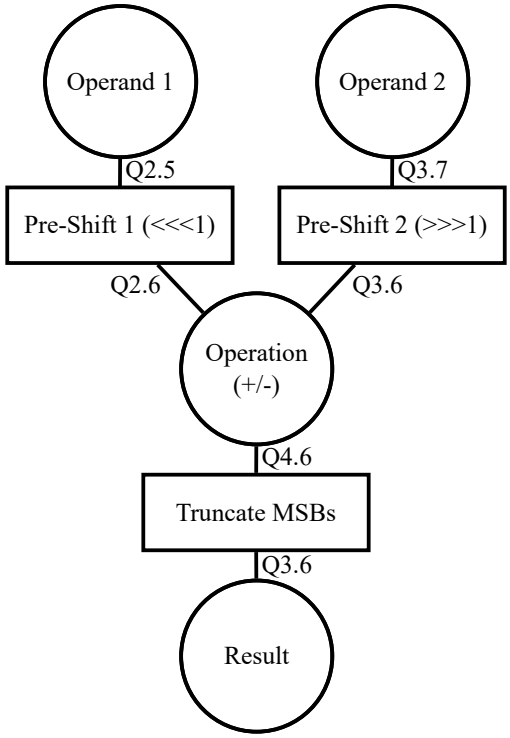

Figure 3.2: The internal structure of a Q3.6 addition or subtraction node

When multiplying, even though the alignment of the radix point is not required, we decided to apply a shift operation on the operands to limit the size of the multiplier required. The output of the multiplier has a WL that is the sum of the input operands. An MSB and LSB truncation operation can be applied to reduce the output WLs which reduces storage requirements. The choice to apply either pre-shift, post-truncation or both is up to the designer. If desired all three options can be explored by the tool. We choose to perform pre-shifting if the FWL of the operand exceeds the output FWL of the node before performing post-truncation of both MSB and LSB. Limiting the options available to the optimizer reduces the complexity of the tool and we found it to be effective still. An example of a multiplication node requiring pre-shifting on one input is shown in Figure 3.3.

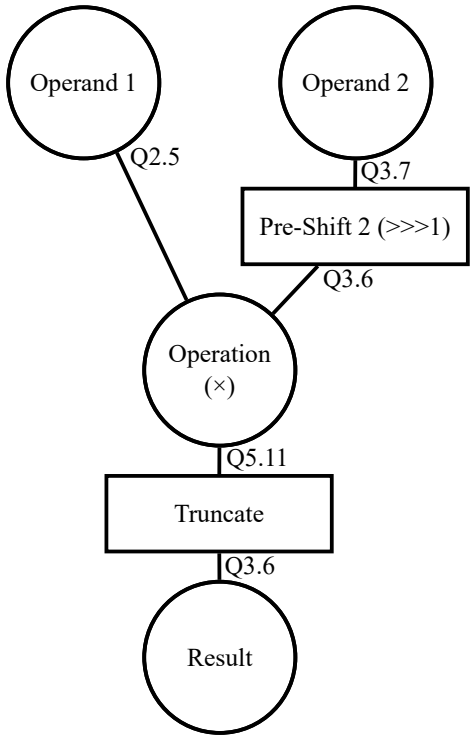

Figure 3.3: The internal structure of a Q3.6 multiplication node

3.2 Equation Representation

The equations we are trying to implement in a circuit is converted into nodes and connected into a data flow graph (DFG) as discussed in Section 2.8. In our tool, the user is required to construct this DFG manually for the rest of the stages. An example of a DFG that represents Equation 3.1 is shown in Figure 3.4 along with the traversal order.

$$D(A, C) = (A \times B) + C \tag{3.1}$$

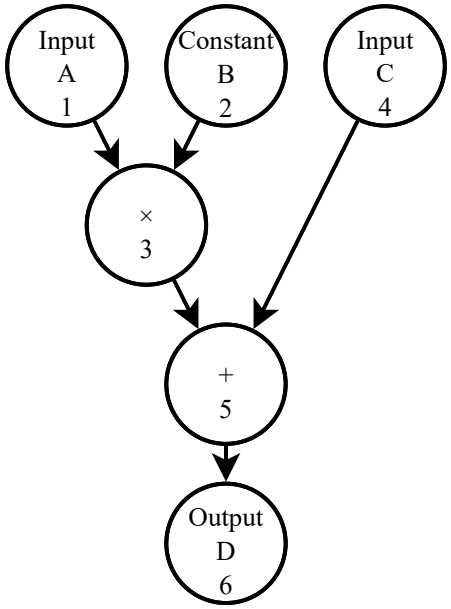

Figure 3.4: Equation 3.1 as a DFG with processing order

The traversal order of the DFG is the same for all processing steps in this chapter. Before we can work with a node, we must work with its inputs first, propagating node properties forward during analysis or generating dependencies during code generation.

The DFG allows us to store nodes and the connectivity between them. The nodes contain information related to the analysis stages or final implementation. Once the

equation is stored in a DFG it can be analyzed and manipulated by traversing the graph and changing properties stored at each node. Source nodes are used to represent constants or inputs, intermediate nodes are used to represent arithmetic operations such as addition, subtraction or multiplication, and finally sink nodes are used to represent the outputs of the circuit. Each node contains constraints configured by the user, and variables explored by our tool.

3.2.1 Source Node Properties

For the source nodes that represent an input, the input fixed-point format (F) is required from the user as it defines the input connectivity of the generated module. Table 3.1 shows the constraints the user must supply. The range (R) and input uncertainty (IU) must be supplied by the user since any inputs might contain errors. To simplify the tool, the interval IU must include 0.

Table 3.1: Source Node Properties

Node Property	User-Provided Constraint	Tool Controlled Variable
Input Uncertainty (IU)	$\left[\underline{IU}, \overline{IU}\right], \underline{IU} \leq 0, \overline{IU} \geq 0$	
Range (R)	$\left[\underline{R}, \overline{R}\right]$	
Uncertainty (U)		$\left[\underline{U}, \overline{U}\right]$
Format (F)	$Q_{F_a.F_b}$	

Source nodes that represent a constant have an input range such that $\overline{R}$ and $\underline{R}$ are the same. Input uncertainty can be defined for constants if the constant varies, for example with environmental factors.

3.2.2 Arithmetic Node Properties

The operation (OP) must be supplied by the user for an intermediate node representing an arithmetic operation as shown in Table 3.2. The tool will derive all the other variables using information from connected operand nodes.

Table 3.2: Arithmetic Node Properties

Node Property	User-Provided Constraint	Tool Controlled Variable
Operator (OP)	$OP \in \{+, -, \times\}$	
Range (R)		$\left[\underline{R}, \overline{R}\right]$
Uncertainty (U)		$\left[\underline{U}, \overline{U}\right]$
Format (F)		$Q_{F_a.F_b}$

3.2.3 Output Node Properties

For sink nodes that represent outputs, Table 3.3 make up the required properties. The output format is required from the user as it defines the output connectivity of the generated module. The output uncertainty requirement (UR) is the uncertainty the WL optimizer must stay below. The rest of the variables are propagated from the output nodes' input.

Table 3.3: Output Node Properties

Node Property	User-Provided Constraint	Tool Controlled Variable
Range (R)		$\left[\underline{R}, \overline{R}\right]$
Uncertainty (U)		$\left[\underline{U}, \overline{U}\right]$
Format (F)	$Q_{F_a.F_b}$	
Output Uncertainty Requirement (UR)	$\left[\underline{UR}, \overline{UR}\right]$	

3.3 Range and Precision Analysis

During the WL optimization process, we first apply IA until result show constraints might not be met before applying SMT analysis to determine if the solution meets constraints. SMT analysis is used with a configurable timeout as it may take a long time, if the process times out, we assume the IA results correctly identified that constraints are not met and move forward.

3.3.1 Interval Arithmetic

We use the IA techniques we summarized in Section 2.6.2 to propagate range and uncertainty bounds through the DFG. We keep the range and uncertainty computations separate when propagating bounds to facilitate optimization of IWL and FWL separately. The range (R) and uncertainty (U) intervals we discussed in Section 3.2 are calculated during IA analysis.

During IA we create a temporary interval, pre-shift uncertainty (PSU) to represent precision loss due to the pre-shift operations discussed in Section 3.1.

Source Node Bounds

For source nodes, we refer to Table 3.1 and see that uncertainty (U) must be computed. The uncertainty includes the input uncertainty and additional error caused by conversion from real to fixed-point. This means we can use Equation 2.8 discussed previously to compute the uncertainty as shown in Equation 3.2.

$$U = \left[\underline{IU} - 2^{-F_b}, \overline{IU} + 2^{-F_b}\right] \tag{3.2}$$

Arithmetic Node Bound Propagation

Recall that the arithmetic nodes contain the tool controlled variable, format (F), range (R), and uncertainty (U) as shown in Table 3.2. The fixed-point format (F) will be set in a different step in the optimization process. R and U will be computed using IA operations. Properties associated with operand nodes of the arithmetic node will be referred to using one dot (left operand) or two dots (right operand) above property names, an example of this is shown in Figure 3.5.

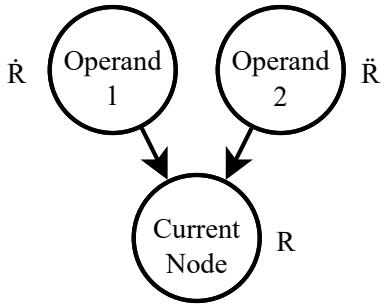

Figure 3.5: Referencing R interval of operand nodes from the current node

Uncertainty bound propagation for addition, subtraction and multiplication is initially the same. Addition and subtraction operations require that we include any precision loss from truncation of LSBs when aligning the radix point. According to our arithmetic model discussed in Section 3.1, we perform pre-shifting on each of the operands of a multiplier if the operands FWL is larger then the nodes FWL. Since for addition and subtraction, increasing the FWL does not cause any precision loss, it turns out that for all three operations the uncertainty expansion is the same.

Modeling of precision loss due to radix alignment operations and efforts to reduce the WL of a multiplier is achieved using Equations 3.3 which is based on Equation 2.8. The uncertainty bounds after pre-shifting are stored in a temporary PSU interval.

$$(\dot{F}_b > F_b) \implies P\dot{S}U = \left[\underline{\dot{U}} - 2^{-F_b} + 2^{-\dot{F}_b}, \overline{\dot{U}} + 2^{-F_b} - 2^{-\dot{F}_b}\right] \tag{3.3a}$$

$$(\dot{F}_b \leq F_b) \implies P\dot{S}U = \left[\underline{\dot{U}}, \overline{\dot{U}}\right] \tag{3.3b}$$

$$(\ddot{F}_b > F_b) \implies P\ddot{S}U = \left[\underline{\ddot{U}} - 2^{-F_b} + 2^{-\ddot{F}_b}, \overline{\ddot{U}} + 2^{-F_b} - 2^{-\ddot{F}_b}\right] \tag{3.3c}$$

$$(\ddot{F}_b \leq F_b) \implies P\ddot{S}U = \left[\underline{\ddot{U}}, \overline{\ddot{U}}\right] \tag{3.3d}$$

For addition and subtraction, uncertainty bounds can then be calculated using one of Equations 3.4 depending on the operation.

$$(OP \equiv +) \implies U = P\dot{S}U + P\ddot{S}U \tag{3.4a}$$

$$(OP \equiv -) \implies U = P\dot{S}U - P\ddot{S}U \tag{3.4b}$$

Subsequently, range bounds for these operations can be computed using one of Equations 3.5 depending on the operation.

$$(OP \equiv +) \implies R = \dot{R} + \ddot{R} \tag{3.5a}$$

$$(OP \equiv -) \implies R = \dot{R} - \ddot{R} \tag{3.5b}$$

Propagation of range or uncertainty for multiplication operations is more complicated as the two are linked. Recall that we have already computed uncertainty due to pre-shifting using Equations 3.3. We then compute bounds using the uncertainty after pre-shifting by using Equations 2.6 which were previously discussed. Equations 3.6 perform the IA operations required for multiplication nodes.

$$a = \left\{ (\underline{\dot{R}})(\underline{\ddot{R}}), (\underline{\dot{R}})(\overline{\ddot{R}}), (\overline{\dot{R}})(\underline{\ddot{R}}), (\overline{\dot{R}})(\overline{\ddot{R}}) \right\} \tag{3.6a}$$

$$b = \left\{ \begin{array}{c} (\underline{\dot{R}} + \underline{P\dot{S}U})(\underline{\ddot{R}} + \underline{P\ddot{S}U}), (\underline{\dot{R}} + \underline{P\dot{S}U})(\overline{\ddot{R}} + \overline{P\ddot{S}U}), \\ (\overline{\dot{R}} + \overline{P\dot{S}U})(\underline{\ddot{R}} + \underline{P\ddot{S}U}), (\overline{\dot{R}} + \overline{P\dot{S}U})(\overline{\ddot{R}} + \overline{P\ddot{S}U}) \end{array} \right\} \tag{3.6b}$$

$$R = \Big[min(a), max(a) \Big] \tag{3.6c}$$

$$U = \Big[min(b) - min(a), max(b) - max(a) \Big] \tag{3.6d}$$

3.3.2 SMT Analysis

To check if a tighter bound on range and uncertainty exists when IA suggests that our design might not satisfy our requirements, we need a suitable constraint set for our SMT solver. We must represent the operations as a sequence of constraints on reals and integers. If all the constraints can be satisfied, then the bounds are violated.

Source Node Constraints

For source nodes, we build the constraint from the properties introduced previously in Table 3.1. The constraint on real SMT variable RV_{real} in Equation 3.7 represents all possible real values with input uncertainty (IU) included.

$$(\underline{R} + \underline{IU}) \leq RV_{real} \leq (\overline{R} + \overline{IU}) \tag{3.7}$$

Equations 3.8 shows the constraints on the integer SMT variable $FV_{integer}$ that models real to fixed-point conversion with rounding towards zero. Input uncertainty (IU) is included here by expanding the real values represented by each integer value.

$$(RV_{real} \geq 0) \implies \left\{ \begin{array}{l} (RV_{real} < (((FV_{integer} + 1) \times 2^{-F_b}) - \underline{IU})) \\ \qquad \wedge \\ (RV_{real} \geq ((FV_{integer} \times 2^{-F_b}) - \overline{IU})) \end{array} \right\} \tag{3.8a}$$

$$(RV_{real} < 0) \implies \left\{ \begin{array}{l} (RV_{real} \leq ((FV_{integer} \times 2^{-F_b}) - \underline{IU})) \\ \qquad \wedge \\ (RV_{real} > (((FV_{integer} - 1) \times 2^{-F_b}) - \overline{IU}) \end{array} \right\} \tag{3.8b}$$

Arithmetic Node Constraints

The arithmetic models in Section 3.1 require unique constraints depending on the operation due to the different rules regarding radix point alignment for addition and subtraction or size reduction for the multiplier.

In this section, we again refer to the node properties associated with the operand nodes using a dot (left operand) or two dots (right operand) above property names of SMT variables as shown previously in Figure 3.5.

For arithmetic nodes representing an addition or subtraction, the first constraint is chosen from Equations 3.9 depending on the operation. It represents the exact result.

$$(OP \equiv +) \implies RV_{real} = \dot{\mathrm{RV}}_{\mathrm{real}} + \ddot{\mathrm{RV}}_{\mathrm{real}} \tag{3.9a}$$

$$(OP \equiv -) \implies RV_{real} = \dot{\mathrm{RV}}_{\mathrm{real}} - \ddot{\mathrm{RV}}_{\mathrm{real}} \tag{3.9b}$$

The constraints in Equations 3.10 scale each fixed-point operand to represent pre-shifting for radix alignment and stores the result in the $PSV_{integer}$ SMT variable.

$$\dot{PSV}_{integer} = \dot{FV}_{integer} \times 2^{F_b - \dot{F}_b} \tag{3.10a}$$

$$\ddot{PSV}_{integer} = \ddot{FV}_{integer} \times 2^{F_b - \ddot{F}_b} \tag{3.10b}$$

Next, we operate on the radix aligned fixed-point values depending on the operation as shown in Equations 3.11.

$$(OP \equiv +) \implies FV_{integer} = \dot{PSV}_{integer} + \ddot{PSV}_{integer} \tag{3.11a}$$

$$(OP \equiv -) \implies FV_{integer} = \dot{PSV}_{integer} - \ddot{PSV}_{integer} \tag{3.11b}$$

The first constraint for multiplication nodes is like the addition and subtraction nodes as it just produces the product of the real values as shown in Equation 3.12.

$$RV_{real} = \dot{\mathrm{RV}}_{\mathrm{real}} \times \ddot{\mathrm{RV}}_{\mathrm{real}} \tag{3.12}$$

Pre-shifting for multiplication nodes is a little different as we only pre-shift values if the operands' FWL exceeds our output FWL since alignment of the radix point is not required and reducing the FWL just serves to reduce the area of a multiplier. This pre-shift operation is represented in Equation 3.13.

$$(\dot{F}_b > F_b) \implies \dot{PSV}_{integer} = \dot{FV}_{integer} \times 2^{F_b - \dot{F}_b} \tag{3.13a}$$

$$(\dot{F}_b \leq F_b) \implies \dot{PSV}_{integer} = \dot{FV}_{integer} \tag{3.13b}$$

$$(\ddot{F}_b > F_b) \implies \ddot{PSV}_{integer} = \ddot{FV}_{integer} \times 2^{F_b - \ddot{F}_b} \tag{3.13c}$$

$$(\ddot{F}_b \leq F_b) \implies \ddot{PSV}_{integer} = \ddot{FV}_{integer} \tag{3.13d}$$

Then multiplication of the pre-shifted fixed-point values is performed in Equations 3.14 and stored in temporary variable $PMV_{integer}$.

$$PMV_{integer} = \dot{PSV}_{integer} \times \ddot{PSV}_{integer} \tag{3.14}$$

Finally, Equations 3.15 show how truncation of the LSBs of the result is modeled.

$$((\dot{F}_b \leq F_b) \wedge (\ddot{F}_b \leq F_b)) \implies FV_{integer} = PMV_{integer} \times 2^{F_b - \dot{F}_b - \ddot{F}_b} \tag{3.15a}$$

$$((\dot{F}_b \leq F_b) \wedge (\ddot{F}_b > F_b)) \implies FV_{integer} = PMV_{integer} \times 2^{F_b - \dot{F}_b} \tag{3.15b}$$

$$((\dot{F}_b > F_b) \wedge (\ddot{F}_b \leq F_b)) \implies FV_{integer} = PMV_{integer} \times 2^{F_b - \ddot{F}_b} \tag{3.15c}$$

$$((\dot{F}_b > F_b) \wedge (\ddot{F}_b > F_b)) \implies FV_{integer} = PMV_{integer} \times 2^{-F_b} \tag{3.15d}$$

Output Node Range Constraints

Output nodes contain the uncertainty requirement. To check if this requirement is met we need a constraint that detects if the fixed-point value converted back into a real deviates away from the real value by too much. In this section, we refer to the output nodes' input node properties with a dot above property names or SMT variables.

First, we need to perform a scaling operation on the input fixed-point value if the input of the output node does not have the same FWL. This scaling is modeled by Equation 3.16.

$$FV_{integer} = \dot{FV}_{integer} \times 2^{F_b - \dot{F}_b} \tag{3.16}$$

Next, we need to convert our fixed-point value back into a real using Equation 3.17.

$$FV_{real} = FV_{integer} \times 2^{-F_b} \tag{3.17}$$

Then, detecting if the bounds are violated is done with Equation 3.18.

$$(FV_{real} \geq \dot{RV}_{real} + \overline{UR}) \vee (FV_{real} \leq \dot{RV}_{real} - \underline{UR}) \tag{3.18}$$

If multiple output nodes exist in the DFG then the constraint set in Equation 3.18 can be combined with the same constraint set for the other output nodes using a logical OR between all of them. Therefore, if any of the output nodes fail to meet their constraints, the SMT solver will tell us that the entire constraint set is satisfiable.

Overflow Constraints

Alternatively, we can choose to exclude the uncertainty bound violation checks in Equation 3.18 and instead check for integer overflow on any or all of the nodes in the DFG. This can be accomplished for any or all of the nodes using Equation 3.19. Similarily to uncertainty checks, all the constraints for overflow checks must be combined using logical ORs.

$$(FV_{integer} \geq 2^{F_a+F_b}) \vee (FV_{integer} < -2^{F_a+F_b}) \tag{3.19}$$

3.4 Implementation Cost Estimation

We use an equation-based method for estimating the number of $m \times m$ multiplier elements required to implement our multiplication nodes as they are scarce in small FPGAs. We ignore the number of LUTs required to implement our equations since a reduction of the size of our multipliers will indirectly reduce the number of LUTs required for other operations in the equation.

FPGAs combine smaller hard multipliers into larger ones by shifting and adding up partial products (Intel Corporation (2004)). The FPGAs our tool targets have $2m \times 2m$ hard multipliers that can be fractured into $m \times m$ multiplier elements. Multipliers larger than $2m \times 2m$ requires cascading which significantly increases the number of multiplier elements required. Cascading of multiplier elements involves shifting and adding partial products. When multiplying a $2m$ wide input by an m wide input a full $2m \times 2m$ multiplier is needed.

To determine the size of the multipliers required we first determine if we need a multiplier at all. If either of the inputs is a constant one or zero then the multiplication can either be implemented with a shift operation or the result is always zero respectively, the cost, in these cases, is calculated using Equation 3.20. A and B refer to the two operands of the multiplication operation.

$$(((A \equiv 0) \vee (A \equiv 1)) \vee ((B \equiv 0) \vee (B \equiv 1))) \implies (Cost(A \times B) = 0) \tag{3.20}$$

If Equation 3.20 does not apply, we round the total WL including a sign bit, of each input up to the nearest multiple of m. The operands are then segmented into as

many blocks $2m$ wide as possible before segmenting any remaining bits into m wide blocks if needed. An illustration of this process is shown in Figure 3.6.

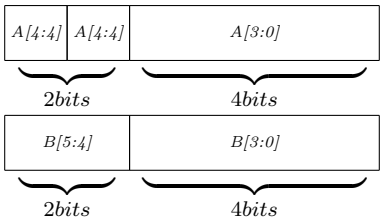

Figure 3.6: Segmentation of an input with a WL of 5 before sign extension and input with a WL of 6

Computing the number of multiplier elements required now requires that we count all the unique pairs of segments and adding up the number of multiplier elements needed as shown in Equation 3.21. Note that we need two elements if either of the operands is $2m$ in width.

$$Cost(A \times B) = 2(n_{A_{2m} \times B_{2m}} + n_{A_{2m} \times B_m} + n_{A_m \times B_{2m}}) + n_{A_m \times B_m} \tag{3.21}$$

3.5 WL Optimization and Code Generation

We allow the fixed-point format to be different throughout the FPGA design, by doing so we open up a vast design space that is difficult to work with manually, so we developed a tool that automatically explores this space. The tool optimizes WLs stored in the format (F) properties described in Section 3.2 which denote IWL and FWL of a node with F_a and F_b respectively. The tool we will be detailing in this section is summarized in Figure 3.7.

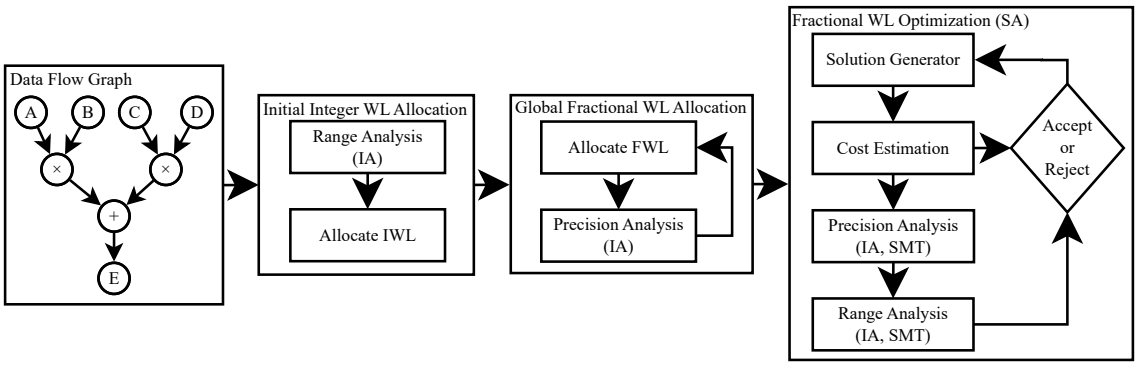

Figure 3.7: WL optimization process

During initial IWL allocation we compute the minimum required IWL for each node in the DFG using the range (R) calculated using the IA techniques we described in Section 3.3.1. This initial IWL allocation does not including uncertainty effects which can increase the range to the point where overflow can occur, but this result is required for cost estimation. For each node, we select the lowest IWL such that Equation 3.22 is satisfied.

$$(2^{F_a} - 1 \geq \overline{R}) \wedge (-2^{F_a} \leq \underline{R}) \tag{3.22}$$

Since we do not include the range increase that the fractional bits contribute, we decrease the chance that the second range check later in the optimization process will discover that overflow is possible. This allows us to avoid a second round of cost estimations without a significant loss in cost estimation accuracy.

Our optimizer follows the same type of process as MiniBit+ (Osborne *et al.* (2007)) where a coarse FWL optimization step is used to find the minimum global FWL. After, each nodes' FWL is refined individually using simulated annealing (SA).

Initially, every node with a configurable FWL has it set to zero. All the configurable FWLs are then increased by one until the uncertainty (U) on all the output nodes obtained through IA shrinks enough to meet the output uncertainty requirement (UR) set on the output nodes. In other words, once Equation 3.23 is satisfied for all output nodes in the DFG, we stop increasing the global FWL. We set this as the initial base solution used in the next stage.

$$(\underline{UR} \leq \underline{U}) \wedge (\overline{UR} \geq \overline{U}) \tag{3.23}$$

After, we begin the fine-grained FWL optimization. We use SA for solution exploration, but many other meta-heuristic algorithms would work as we discussed in Section 2.6.1. SA was chosen for its simplicity and popularity in other WL optimization tools. Our SA based algorithm generates neighboring solutions by randomly selecting an arithmetic node and randomly increasing or decreasing the FWL by one. The cost of the generated solution will then be evaluated. If the cost is better then the best so far, we immediately go onto the next step. If the cost is worse, we randomly generate a number between 0 and 1. If the value is below the hypothetical temperature (T) of the system, also between 0 and 1 we accept it, else we reject it and go back to

solution generation. Every time we generate a solution, we use a cooling function to decrease the temperature of the system, thereby decreasing the chance of accepting worse solutions the more solutions we try. Equation 3.24 is the cooling function we used, c_{rate} being a user-configurable cooling rate, another value between 0 and 1.

$$T_{new} = c_{rate} \times T_{old} \tag{3.24}$$

Every solution that makes it to the next step is checked to see if uncertainty bounds are met using our IA techniques. If our IA techniques tell us that uncertainty bounds might be exceeded, we try using an SMT solver to see if tighter bounds can be found that pass our requirements. We then need to perform an IA range check again with the range and uncertainty bounds combined since uncertainty can cause the range to increase. If our IA techniques tell us that range bounds are exceeded, we switch to SMT analysis to check if the solution is acceptable. If the solution is validated through IA then it is accepted by the SA algorithm as the next base solution, if it was validated by SMT then it is just reported as a good solution to avoid trapping the optimizer in a state where IA always fails. If the cost of this solution is a new minimum, we save that as the current best score. We repeat this process until the temperature of the system drops below a user-configurable value. After, we can choose to repeat the SA process with the same base solution given to us through global FWL optimization to find more solutions.

Once we have a solution we are satisfied with from the SA step, we move onto code generation. Since the cost estimation might not be perfectly accurate due to optimizations that might be performed by the FPGA tools, it might be worth taking a few of the best solutions generated and trying them.

3.5.1 Code Generation

Our tool generates SystemVerilog code but targeting a different HDL is a trivial task since all the language features we use have analogues in Verilog and VHDL. The code generation step in our tool is very simple since we do not perform any form of resource sharing, pipelining or scheduling. However, it should be noted that there is no reason why those stages cannot be included between the WL optimization and code generation stage. In the next chapter, we will calculate the overall computational latency reduction and increase in throughput that could be gained with the addition of an automated pipelining step. This step is left out since the demonstration of automatic pipelining is not a goal of this work and adds additional complexity to the code generation step. To prevent the generated module from decreasing the achievable clock rates in the rest of the design we add one register after each arithmetic node being generated.

As an example, we will use Equation 3.1 and the associated DFG in Figure 3.4 with values generated by the WL optimization stage shown in Table 3.4 to demonstrate our code generator.

Table 3.4: Example DFG Configuration

Node	Format (F)	Constant
Input A	$Q_{3.8}$	
Constant B	$Q_{2.6}$	3.14159
Input C	$Q_{4.6}$	
Multiply	$Q_{6.7}$	
Add	$Q_{6.6}$	
Output D	$Q_{6.6}$	

Our code generator first generates all the input and output nodes to build the generated modules input and output ports. The total WL of each node includes a sign bit, the FWL and the IWL. We also include the clock for the module at this point for internal registers.

```
module equation (
    input                                clk,
    input         signed [11:0]  input_A,
    input         signed [10:0]  input_C,
    output logic  signed [12:0] output_D_reg
    );
```

We then generate all the internal wires and registers that we will assign values to throughout the design. Note that we converted 3.14159 into an integer by scaling the real value to create a fixed-point value with an FWL of 8.

```
    logic signed [12:0] output_D;

    parameter signed [8:0] constant_B = 804;

    logic signed [13:0] multiply_0;
    logic signed [13:0] multiply_0_reg;
    logic signed [12:0] add_0;
    logic signed [12:0] add_0_reg;
```

Next, we generate the computation logic in the traversal order shown in Figure 3.4, this is to ensure that dependencies are calculated before they are used. Note that inputs and constants are skipped since they already have their value. Here we follow the shifting rules defined in Section 3.1.

```
    always_comb begin
        multiply_0 = ((input_A>>>1) * (constant_B))>>>6;
        add_0 = ((multiply_0>>>1) + (input_c));
        output_D = add_0;
    end
```

After, we generate the register logic and end the module.

```
    always_ff @ (posedge clk) begin
        multiply_0_reg <= multiply_0;
        add_0_reg <= add_0;
        output_D_reg = output_D;
    end
endmodule
```

The resulting circuit modeled by this generated SystemVerilog module does not have a balanced pipeline since there are varying numbers of registers on each path all input nodes to all output nodes. In the results, we calculate the benefits that can be expected with the addition of an automated pipelining algorithm.

3.6 Chapter Summary

In this chapter, we discussed a process for automated WL optimization of equations destined to be implemented on a small FPGA with hard multiplier elements. The process analyzes a DFG given by the user using IA and SMT based techniques to allocate integer and fractional bits for fixed-point implementation of equation evaluating circuits on FPGAs. This is useful for implementing equations required in a predictive motor controller.

In the next chapter, we will be using our tools and methods in conjunction with traditional design techniques to create part of an FPGA based FCS-MPC for controlling an IPMSMs. In Chapter 5, the design will be used to evaluate the FPGA area savings made with our method and demonstrate the control performance degradation we are trading off by using it.

Chapter 4

Controller Design and Test Setup

To evaluate our tools from Chapter 3, we use them to generate part of a design that implements finite control set model predictive control (FCS-MPC) for an interior permanent magnet synchronous motor (IPMSM). Our design consists mostly of hand-written hardware description language (HDL) code with the addition of a tool generated module. We will use this design to evaluate both the FPGA area utilization improvements and detrimental effects on control performance when our tools are utilized. A controller will be built and integrated with an inverter, motor and test setup as shown in Figure 4.1. Our simulation environment will also be discussed.

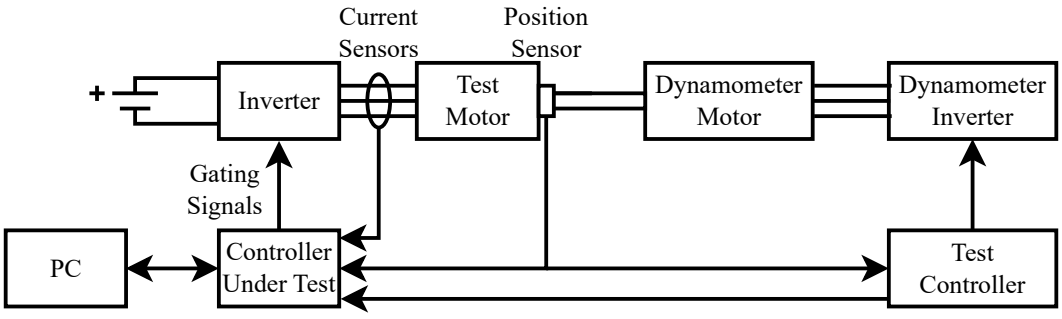

Figure 4.1: Test system architecture

4.1 Physical Test Setup

We use a modified version of the experimental test setup described in Appendix B of Nalakath dissertation (Nalakath (2018)). The same dynamometer which consists of a Yaskawa A-1000 drive unit and a 5 kW induction motor is utilized. The dynamometer is controlled by a dSPACE MicroAutobox II (dSPACE GmbH (2019a)) and is used to spin the motor under test at a constant speed. In Nalakath's experiments, the motor control algorithms are also implemented on the dSPACE, but for our purposes, we will implement the control algorithms on a custom circuit board which contains our FPGA based controller instead. For safety purposes, the dSPACE is connected to our controller so it can disable it under fault conditions. The test setup is shown in Figure 4.2.

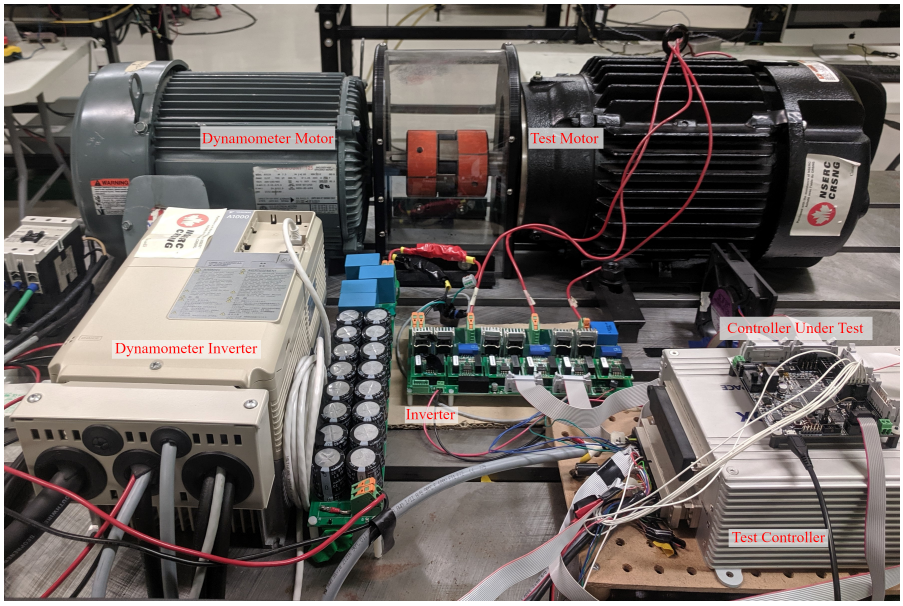

Figure 4.2: Test setup

Since the IPMSM under test is the same, we use the values measured by Nalakath to fill in the constants that will be required by the IPMSM model. The specifications of the motor under test are shown in Table 4.1

Table 4.1: IPMSM Nominal Parameters and Specifications

Constant	Value
Pole count	10
Rated Current	9.4 A
Rated Speed	700 RPM
Nominal d axis inductance (L_d)	11 mH
Nominal q axis inductance (L_q)	14.3 mH
Nominal permanent magnet flux linkage (λ)	333.3 mWb
Nominal stator resistance (R_s)	400 mΩ

Connected to the shaft of the IPMSM is a BEI Sensors DHO5S-14-P-G5-9-800000-G3-R020 (BEI Sensors SAS (2019)) incremental quadrature encoder for rotational position feedback. This sensor produces two signals which when combined, transitions 320 000 times every rotation.

The inverter connected to the IPMSM is powered from a 300 V DC supply and consists of six CREE C2M0025120D (Cree, Inc. (2015)) silicon carbide power MOSFETs. Gating signals for these MOSFETs come from our FPGA board. For current sampling, the inverter has a CKSR 25-NP (LEM International SA (2009)) current sensor on each phase. Figure 4.3 shows an overview of the inverter.

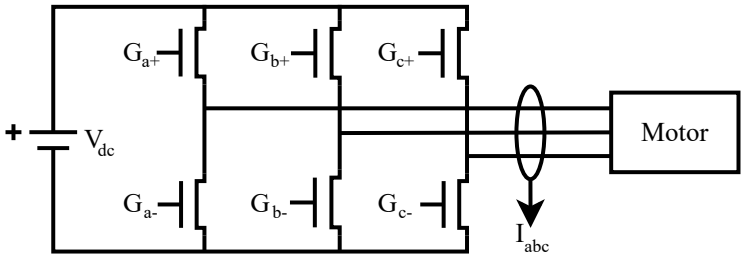

Figure 4.3: IPMSM inverter

All combinations of switch states where the state of the lower switch on each branch is an inversion of the upper switch state are valid. If the upper switch in a specific leg of the inverter is on, then 300 V will be applied to that motor phase, else, 0 V is applied. The result is 8 valid switch states. When any switch changes state, deadtime is required. Deadtime is the time period when both switches in one leg of the inverter are turned off. This is required because switches cannot change states instantaneously. There is a time period while transitioning between states where the switch is partially conductive. If both switches in the same leg of the inverter are partially conductive at the same time, a current will flow through them. Wasting power or potentially damaging the inverter.

4.2 Control Algorithm

To control the IPMSM, we use an FCS-MPC algorithm adapted from Nalakaths' work (Nalakath (2018)). The variant we will implement is without parameter estimation and has a prediction horizon of one. Our goal is to demonstrate higher sampling rates on the same physical test setup but with a controller implemented in fixed-point on small FPGAs. The algorithm selected controls I_d and I_q currents due to its correlation with torque production as described in Section 2.3.2.

FCS-MPC requires that we have a model for the behavior of the controlled system in reaction to something we can influence. Regulating the I_{dq} vector requires that we can measure it first since we need to make our predictions of future I_{dq} vectors starting with measured ones. As the current sensors on the inverter measure I_{abc}, we need to use the Clarke and Park transformation discussed in Section 2.3.1 to obtain I_{dq}. Equations 4.1 (Krishnan (2001)) are needed to perform this transformation.

$$d = \frac{2}{3}(a(\cos(\theta)) + b(\cos(\theta - \frac{2\pi}{3})) + c(\cos(\theta + \frac{2\pi}{3}))) \tag{4.1a}$$

$$q = -\frac{2}{3}(a(\sin(\theta)) + b(\sin(\theta - \frac{2\pi}{3})) + c(\sin(\theta + \frac{2\pi}{3}))) \tag{4.1b}$$

The prediction equations required to build an FCS-MPC controlling I_{dq} of an IPMSM is given in Equations 4.2 (J. Rezaie (2007)).

$$I_{d_{n+1}} = I_{d_n} - \frac{T_s R_s}{L_d} I_{d_n} + \frac{T_s L_q}{L_d} \omega I_{q_n} + \frac{T_s}{L_d} V_d \tag{4.2a}$$

$$I_{q_{n+1}} = I_{q_n} - \frac{T_s R_s}{L_q} I_{q_n} - \frac{T_s L_d}{L_q} \omega I_{d_n} + \frac{T_s}{L_q} V_q - \frac{T_s \lambda}{L_q} \omega \tag{4.2b}$$

T_s is the sampling time, while Ω is the rotational velocity of the motor which we can get from the incremental encoder. I_{d_n} and I_{q_n} are the values we get from measuring I_{abc} and performing the Clarke and Park transformation on them. V_{dq} is the voltage vector we are evaluating using our prediction model since we apply a voltage vector V_{abc} using the inverter, we must also perform the Clarke and Park transformation on all the voltage vectors the inverter can supply before using the model for predictions.

A cost function is then used to evaluate the outcome predicted by the model. We use a simple cost function, shown in Equation 4.3. $I_{d_{target}}$ and $I_{q_{target}}$ are our desired values. Minimum cost is what the control algorithm will optimize for using the available voltage vectors. Using this cost function, we optimize for minimum total deviation away from our target values.

$$Cost = |I_{d_{target}} - I_{d_{n+1}}| + |I_{q_{target}} - I_{q_{n+1}}| \tag{4.3}$$

The algorithm must be executed for each control cycle timed T_s apart. As discussed in Section 2.4.2, FPGA designs that exhibit extremely low latency can be built so we exclude delay compensation from our implementation. As we will see, delay compensation is not strictly necessary. The algorithm summarized in Figure 4.4 shows how the equations we discussed in this section are used to find the best inverter gating signals to apply during the next time step.

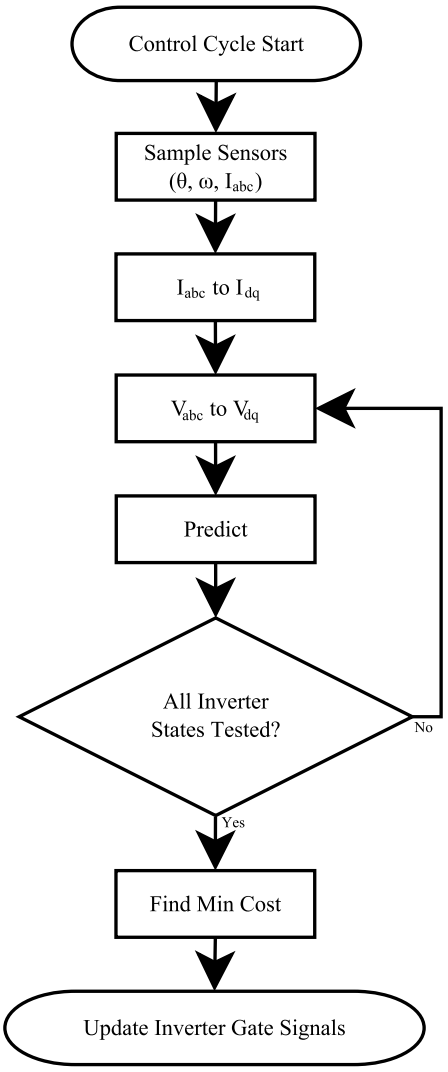

Figure 4.4: Steps taking by the control algorithm every control cycle

4.3 Circuit Board Design

The controller was implemented on a custom circuit board and contains the FPGA. Along with the FPGA, it contains an analog to digital converter (ADC) for reading the current sensors, a differential line receiver to interface with the incremental encoder, and a data interface that uses the universal serial bus (USB) standard (USB-IF (2019)) which is widely supported by many PCs. A high-level overview of the connectivity on the circuit board is provided in Figure 4.5.

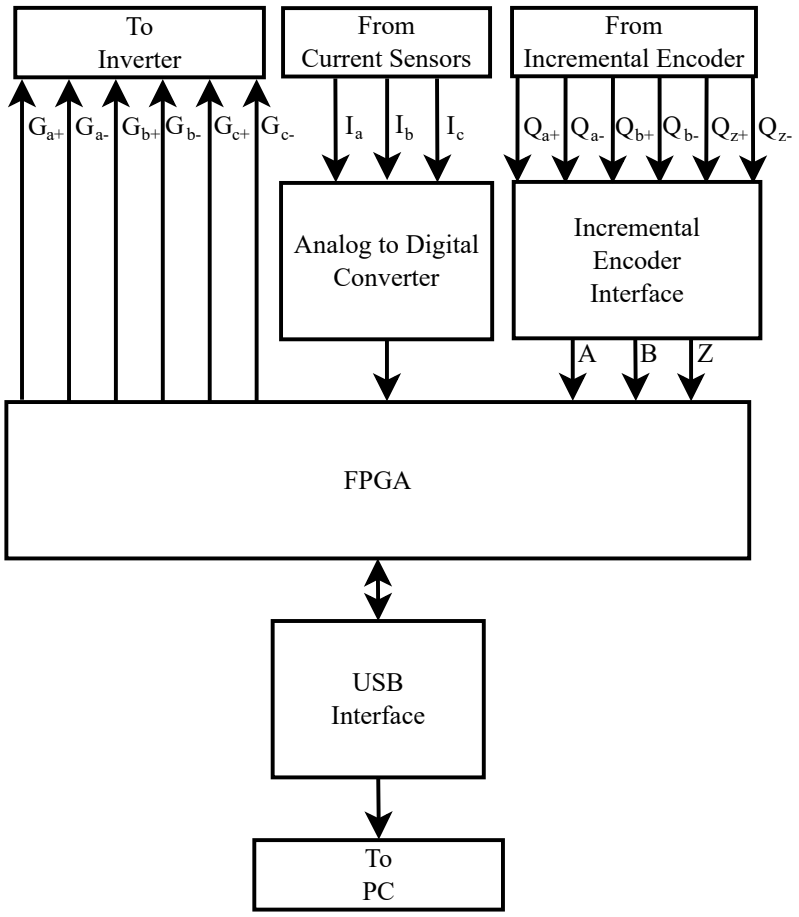

Figure 4.5: High-level overview of the circuit board

The circuit board which measures 10 cm by 10 cm is shown in Figure 4.6.

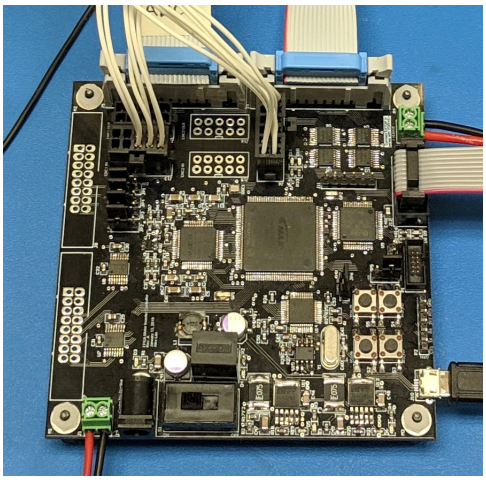

Figure 4.6: Assembled custom controller circuit board

Gating signals produced by the FPGA as shown in Figure 4.5 go to the MOSFETs shown in Figure 4.3 to inject the selected voltage vectors into the motor.

The magnitude and direction of current flow through the high voltage phase cables feeding the motor windings in the IPMSM is converted to a voltage which is then converted into a digital signal by an ADC. We use an ADC that has a minimum sampling time much smaller than the control cycle time (T_s). Therefore, a new sample of I_{abc} is available for each control cycle with negligible latency. The ADC we use is also directly timed by the FPGA, therefore latency between current measurement and updating the gate signals is at a minimum and does not vary, reducing timing jitter in the control loop. Its inputs are also simultaneously sampled. Therefore, all three current sensors are sampled at the same time, not doing so would produce erroneous readings when currents change rapidly. The specific ADC used is a Maxim Integrated MAX11047 with 4 inputs, 4 us sampling time and a 1 us conversion time (Maxim Integrated (2019a)).

Industrial incremental encoders often use higher voltages and differential signaling to increase noise immunity, these signals are incompatible with the FPGA. Therefore, a suitable differential line receiver is used between the sensor and the FPGA. The differential receiver used is a MAX3094E (Maxim Integrated (2019b)), compatible with incremental encoder outputs up to ± 25 V making it suitable for sensors powered using 24 V commonly found in industrial environments. It also supports a maximum input frequency of 10 MHz which far exceeds the frequencies we should see on the individual signals with the motor spinning at its top speed of 700 RPM.

The FPGA, an Intel MAX 10 10M16SCE144C8G with 15840 logic elements, 562176 bits of embedded memory and 90 9x9 multipliers is a member of Intel's FPGA family focusing on affordability and ease of integration (Intel (2019)). Its small size and low cost make it suitable for use in motor controllers intended for cost-sensitive markets where incorporation of fast FCS-MPC may not have been possible if more advanced parts were needed to meet computational requirements.

Run-time configuration and data acquisition capability are provided using an FTDI FT232H USB chip operating in synchronous first in first out (FIFO) mode providing up to 40 MB/s of throughput between the FPGA and PC (FTDI Ltd. (2019)).

For safety purposes, during development, signals from the current sensors are split off and sent to a dSPACE MicroAutobox II (dSPACE GmbH (2019a)) for overcurrent protection. The dSPACE is given control of an inhibit pin on the gate drive circuitry on our controller board allowing it to disable the FPGAs ability to drive the MOSFETs. This will put the inverter in a safe state automatically in case of faults.

4.4 FPGA Design

FPGA designs process data in parallel and can be described as a bunch of modules operating at the same time. Here, we present the high-level architecture of the design in Figure 4.7 before diving into the function of each module. The module that interfaces to the USB controller is not shown, it is connected to every module so access to any value can be added as required.

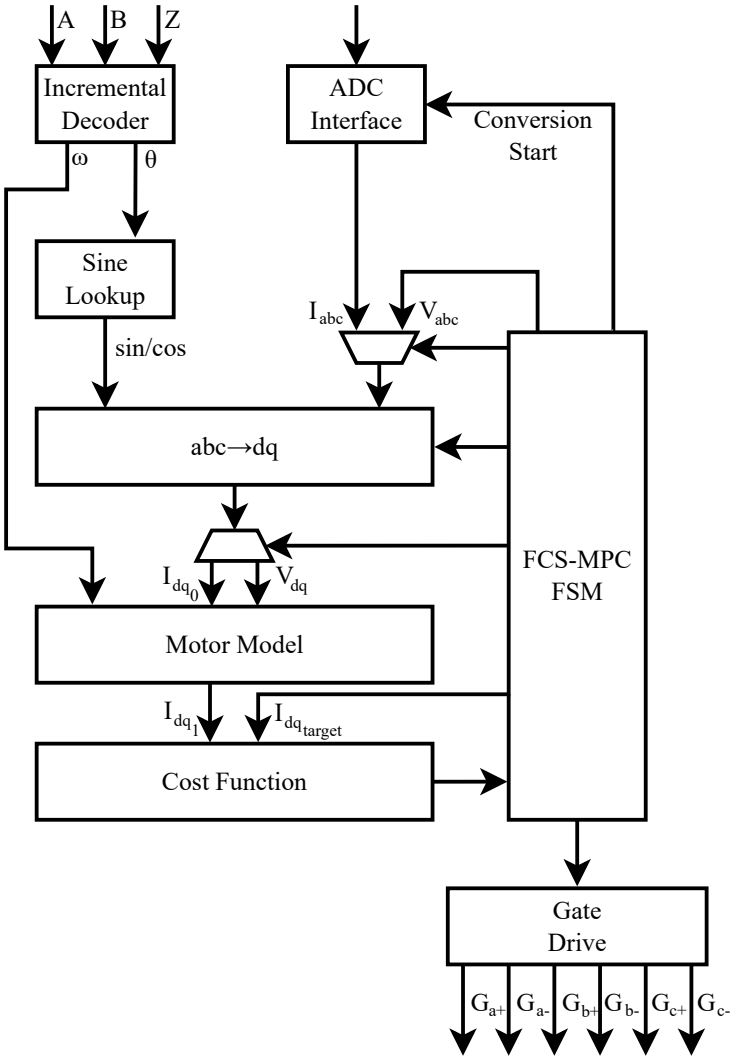

Figure 4.7: Architectural overview of the FPGA design

4.4.1 Traditionally Designed Modules

Finite State Machine

The operation of the controller is coordinated by the FCS-MPC finite state machine (FSM). This module times the ADC sampling. It also controls when values such as θ and I_{dq_0} are sampled and held constant throughout the control cycle. It supplies all the V_{abc} vectors that need to be evaluated and then takes the output of cost function when it is ready. Finally, it ranks the result before using them to select the next gate drive vector.

Incremental Decoder

Incremental quadrature encoders output a pulse once a rotation on its Z output to report the zero position of the motor, subsequent pulses on the A and B outputs which are 90 degrees out of phase with each other allow the FPGA to detect the direction of rotation. Using the phase of the signals and counting the transitions in them allows the FPGA to measure the angle (θ) if the number of pulses per rotation for the sensor is known. An example of the signals output by an incremental encoder is shown in Figure 4.8.

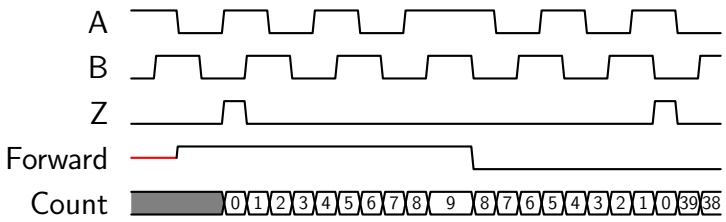

Figure 4.8: Incremental encoder signals with direction change around the index position for a hypothetical sensor with 10 counts (40 edges) per rotation

The relative phase between signals A and B is used to infer the direction of rotation. Two edges are required to detect the phase. With the direction known, incrementing or decrementing a counter is performed to track the change in position relative to the zero index which is signaled by the sensor with a rising edge of the Z signal. The θ output is invalid until the FPGA sees the Z signal transitions high for the first time. Every time the incremental decoder receives a pulse on any of its inputs it will update its count. The incremental encoder used is an 80 000 count incremental encoder which provides us with 320 000 edges per rotation on the A and B signals combined.

The motor we use is a 10 pole motor. One mechanical rotation results in five electrical rotations. Therefore, internally the quadrature decoder resets its count every 64 000 edges to track the electrical angle. 64 000 edges per rotation is still a very high resolution, therefore we keep an internal fast edge count and output a slower edge count that wraps every 16 000 edges. Our resulting θ output is dimensionless and is used directly by the sine lookup table.

Rotational speed can be measured by counting the number of edge transitions in both the A and B signals combined within a certain time period. With our sensor, speed can be accurately measured this way. If a lower resolution sensor is used instead, the reader is directed to investigate techniques suitable for FPGA implementation such as the MT method which can be used to improve velocity measurement by predicting the arrival of the next edge (Hace and Čurkovič (2018)).

Sine Lookup Table

The sine lookup table is used to obtain all 6 of the required scaled sine and cosine values used in the Clarke and Park transform detailed in Equations 4.1. By scaling the values in the lookup table by $\frac{2}{3}$ we alleviate the need to perform this operation at run-time in the Clarke and Park transformation module. Using a single dual-port memory instance on the FPGA we can lookup all 6 values in 3 clock cycles.

Storing a full sine wave with a 15-bit FWL and a sign bit requires 256000 bits if we have one sample for each of the 16 000 θ values from our incremental decoder. 256000 bits is about half of the memory resources available on the FPGA we chose. If required, memory can be saved by only storing a quarter of the sine wave. However, this will complicate the lookup process. Another alternative is to interpolate between the stored samples using the higher position knowledge available from the incremental encoder. We took the simpler approach without an attempt at saving memory as this portion of the design is required but is not being evaluated in this work.

ADC Interface

The ADC interface is timed by the FCS-MPC FSM. The ADC we elected to use gives us control over the sample and hold circuit. Opening of the switches on the sample and hold circuit begins the analog to digital conversion which takes 1 uS. By synchronizing the start of conversion with the control cycle we hold the latency between current sampling and the updating of gate drive signals constant.

Clarke and Park Transformation

The Clarke and Park transformation module (abc $\rightarrow$ dq), implements Equations 4.1. This module is used for transformation of both voltage test vectors and the current measurements. This module could be generated using the methods we described in Chapter 3. However, to isolate most of the quantization effects to the motor model, we build this module using 16-bit FWL and forego additional FWL optimizations.

Cost Function

The cost function module simply evaluates Equation 4.3 using a 16-bit FWL. This operation can be performed in 1 clock cycle. The results are transferred to the FCS-MPC finite state machine to be ranked.

Gate Drive

Finally, the gate drive module generates switch deadtime for each positive and negative switch pair if the selected switch state is different from the previous one. Deadtime introduces a delay when changing the output switch state. In our controller, we elect to have a 1 us deadtime.

Fault detection is also included in this block, disabling gate drive output in the event of overcurrent faults. This protection is redundant with the protection provided by the dSPACE but due to the low latency nature of the FPGA design, it can react more quickly.

4.4.2 Generated Motor Model

The motor model produces predictions for the cost function to evaluate using sensor data. It implements Equations 4.2 to do so. Those cost evaluations are directly used by the FCS-MPC FSM to make control decision. Uncertainty in the predictions can be detrimental to control performance. This module was selected to be generated using our tools because of its role in control decision making. Also, it uses the most multiplier elements in the design if developed with a fractional word-length FWL of 16-bits throughout. If desired, this module can be manually written instead.

The FPGA area impact and effects the optimizations have on control performance will be evaluated using multiple implementation variants. All inputs and outputs of this module have an FWL of 16-bits but internally the FWL will vary as they will be tuned by our optimizer. The difference between implementation variants generated by our tool is a different uncertainty limit imposed on the outputs $I_{d_{n+1}}$ and $I_{q_{n+1}}$. Our tools allow us to include uncertainty values for all inputs so we can calculate and include major sources of uncertainty. It should be noted that we do not include every source of error. More can be included at the discretion of the designer, but these are the sources included in the generated modules used for our results in the next chapter.

Input Range and Uncertainty

The inputs of the motor model primarily consist of I_{dq} or V_{dq} values and the rotational velocity all derived from sensor readings. Therefore, we need to propagate range and uncertainty from these sensors up to the motor models input so our tools can work off of them.

The Clarke and Park transformation block use the sine values from the sine lookup table. The lookup table utilizes the position readings from the incremental decoder to get them. Therefore, we must compute the maximum error on the output of the sine table with 16 000 entries. The highest rate of change in the sine function is at 0, so we calculate the maximum error around this point. The sine table can also be programmed with an offset equal to half the distance between two entries, this essentially means instead of rounding down the position from the incremental decoder we round to the nearest. Unfortunately, 16 000 is not integer divisible by 3 and many of the sine and cosine values required in Clarke and Park transformation requires a $\frac{1}{3}$ rotational offset. This means many of the lookups are off by $\frac{1}{3}$ of an entry. Since we store the sine wave using 15-bits of FWL we need to include the error incurred by the real to fixed-point conversion as well. With all of this in mind, the uncertainty interval of the sine values coming from the sine lookup table is given by Equation 4.4.

$$SineError = \Big[-\sin\Big(\frac{4}{3}\frac{-2\pi}{16000}\Big) - 2^{-15}, \sin\Big(\frac{4}{3}\frac{-2\pi}{16000}\Big) + 2^{-15}\Big] \tag{4.4}$$

The range of values for the sine outputs is given by Equation 4.5. The range is computed using the maximum value of the sine function and the scaling we applied ahead of time, so we don't need to perform the multiplication by $\frac{2}{3}$ required in the Clarke and Park transform.

$$SineRange = \Big[-\frac{2}{3}, \frac{2}{3}\Big] \tag{4.5}$$

For the three-phase current transformation. The CKSR 25-NP (LEM International SA (2009)) sensors produce 25 mV/A of current flow. Our ADC has an input

range of 0 to 5V and a resolution of 16-bits. Therefore, it has a resolution of approximately 76.3 uV per least significant bit (LSB). Using those values, we can calculate that our current measurement has a resolution of about 3.052 mA. The maximum current we should ever expect to see is below the 9.4 A current rating listed in Table 4.1, therefore a range of $[-10, 10]$ is reasonable. Alternatively, for three-phase voltage transformation we have a range of $[0, 300]$ and for simplicity sake, no uncertainty. However, uncertainty can be included if the voltage from the DC supplies varies.

We use the interval arithmetic (IA) steps described in Section 3.3.1 to get the range and uncertainty of the outputs of the transformation module. To do this we use the range and uncertainty values shown in Table 4.2 and 4.3 to calculate bounds on current and voltage transformation respectively. The values come from calculations earlier in this section. Where applicable, the values are all rounded away from zero.

Table 4.2: Current Transform Input Properties

Inputs	Range	Uncertainty
Sine Values	$\left[-\frac{2}{3}, \frac{2}{3}\right]$	$\left[-0.00056, 0.00056\right]$
a,b,c	$\left[-10, 10\right]$	$\left[-0.004, 0.004\right]$

Table 4.3: Voltage Transform Input Properties

Inputs	Range	Uncertainty
Sine Values	$\left[-\frac{2}{3}, \frac{2}{3}\right]$	$\left[-0.00056, 0.00056\right]$
a,b,c	$\left[0, 300\right]$	$\left[0, 0\right]$

To apply IA to calculate range and uncertainty bound, we construct data flow graphs (DFGs) shown in Figure 4.9 and 4.10, for Equations 4.1. All the nodes have an FWL of 16-bits except the sine inputs which have an FWL of 15-bits.

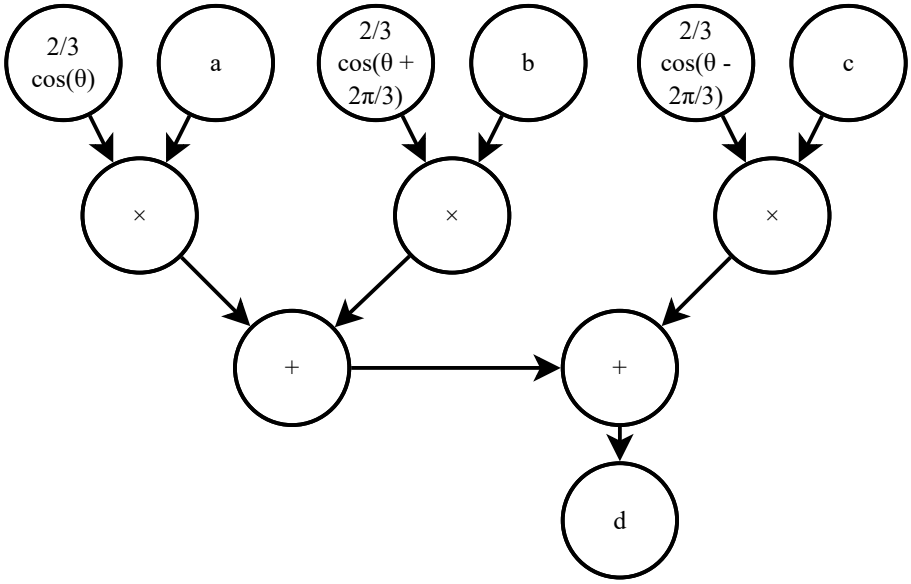

Figure 4.9: DFG for analyzing the d axis Clarke and Park transform

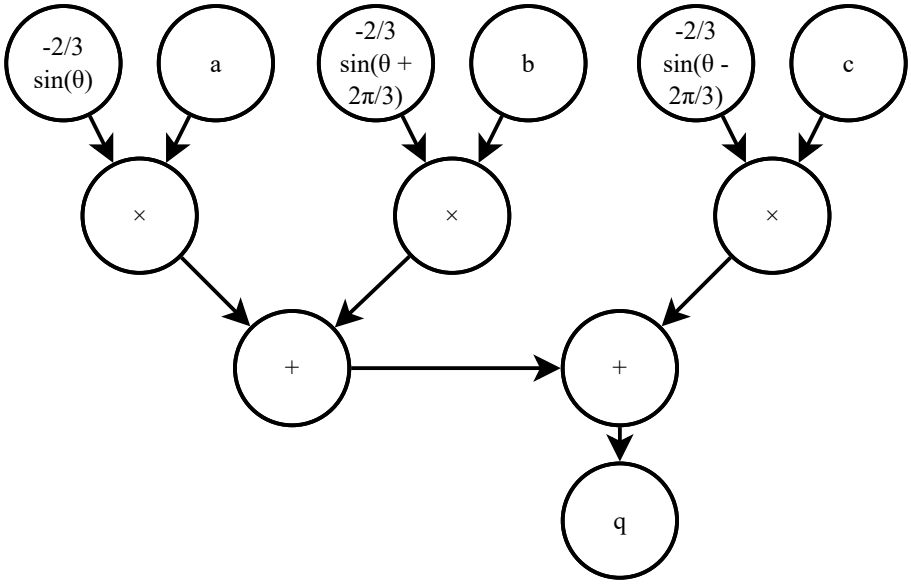

Figure 4.10: DFG for analyzing the q axis Clarke and Park transform

Performing the IA steps from Section 3.3.1 yields the results in Table 4.4 and 4.5 for current and voltage transformation respectively.

Table 4.4: Current Transform Output Properties

Output	Range	Uncertainty
d, q	$\left[-20, 20\right]$	$\left[-0.025, 0.025\right]$

Table 4.5: Voltage Transform Output Properties

Inputs	Range	Uncertainty
d, q	$\left[-600, 600\right]$	$\left[-0.169, 0.169\right]$

Due to the correlation of the sine and cosine values we know that the output d or q value can't exceed $\frac{2}{3}$ times the largest values a, b, c can have if they all have the same range. IA is unable to capture such correlations. Therefore, we make a manual optimization before moving on. We modify the d, q current range to be $[-6.67, 6.67]$ and the d, q voltage range to be $[-200, 200]$.

Rotational velocity is obtained though frequency counting on the edges from the rotary encoder. Due to the high resolution of the rotary encoder, we assume no uncertainty in this value. The range of this value in rad/s is limited by the maximum speed of the IPMSM listed in Table 4.1. 700 RPM is 73.4 rad/s, however since we have a 10 pole motor, the maximum electrical speed is 367 rad/s. Therefore, the range of values for ω is $[0, 367]$.

Module Generation and Usage

To create the motor model, we need to implement Equations 4.2. To reduce the number of operations required to be performed at run-time, we solve for the numerical values that are made up of multiple constants. By doing so we go from Equations 4.2 to Equations 4.6 using the constants in Table 4.1 and our selected sample time (T_s).

$$I_{d_{n+1}} = I_{d_n} - C_1 I_{d_n} + C_2 \omega I_{q_n} + C_3 V_d \tag{4.6a}$$

$$I_{q_{n+1}} = I_{q_n} - C_4 I_{q_n} - C_5 \omega I_{d_n} + C_6 V_q - C_7 \omega \tag{4.6b}$$

A DFG is built to evaluate Equations 4.6 and is shown in Figure 4.11 and 4.12. The DFG is separated for clarity. Using the tools in Chapter 3 and the input range and uncertainty intervals we calculated in the last section, SystemVerilog code implementing the DFG is produced using different uncertainty requirements.

We see that the longest path in the DFG will have 5 register stages. Our implementation of the Clarke and Park transformation required for each prediction requires 3 register stages. The initial current transformation to get initial values for the prediction models will also require the Clarke and Park transformation unit. An additional clock cycle is required by the cost function. Therefore, to evaluate all 8 valid switch states we require 68 clock cycles. At 100 MHz, this creates a computational latency of 0.68 us. With automated pipelining, a feature we excluded from our tools, it would be possible to feed the Clarke and Park transformation and motor model unit every clock cycle after the initial current transformation is completed. We will get our first result after an initial delay of 8 clock cycles. Therefore, only 20 clock cycles would be required to perform all our predictions.

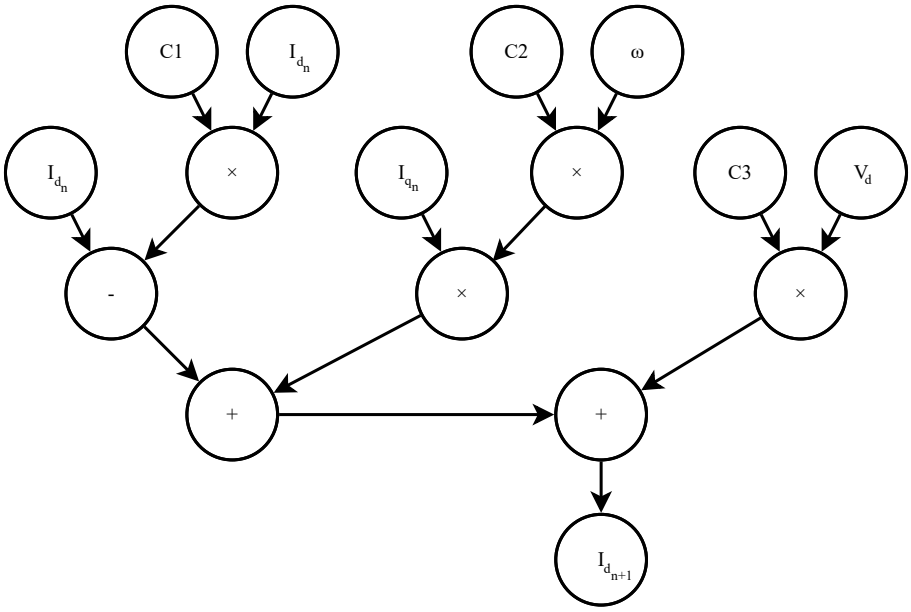

Figure 4.11: d axis portion of the DFG for motor model generation

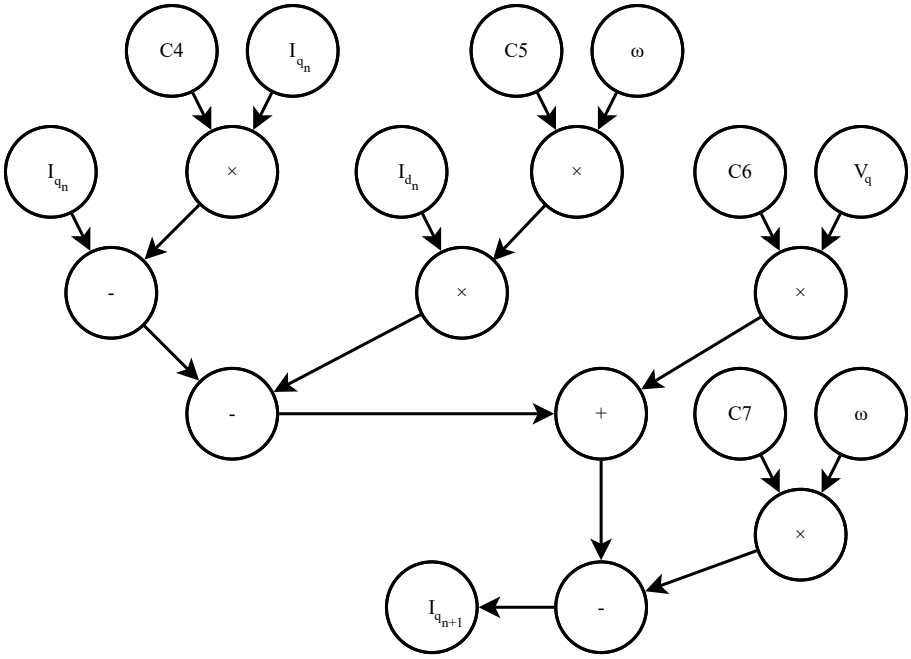

Figure 4.12: q axis portion of the DFG for motor model generation

4.5 Model in Loop FPGA Simulation Environment

Software-based simulation is the primary way of testing FPGA designs as it allows for the highest amount of design visibility. When an FPGA design is implemented on a device, it is generally not possible to capture all the signals within a design. This makes verification and debugging extremely difficult. Therefore, we build a simulation environment as shown in Figure 4.13.

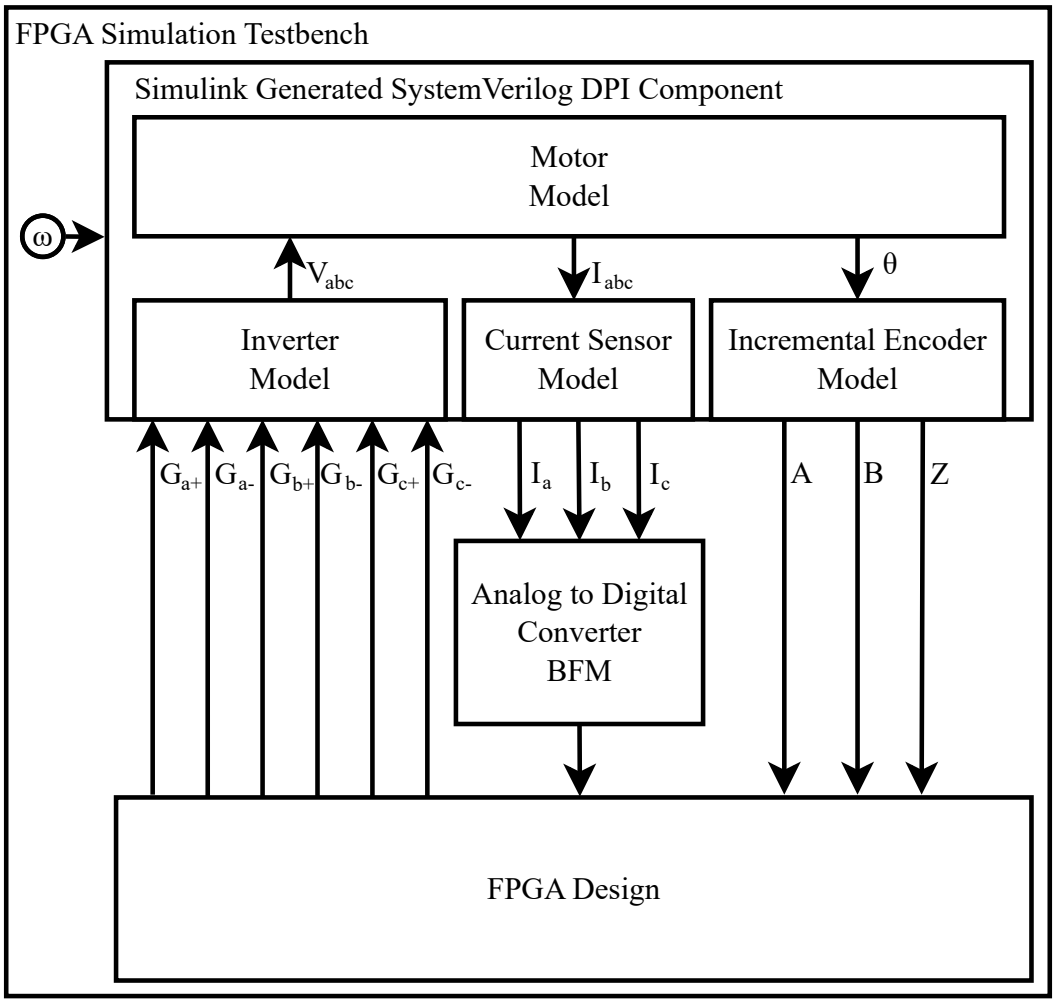

Figure 4.13: High-level overview of the model in loop FPGA simulation environment

Within the simulation environment the components external to the FPGA design such as circuit board components and sensors must be emulated using bus functional models (BFMs). As discussed in Section 2.9, tools such as MATLAB and Simulink can be used to generate BFMs that model physical behavior. Generally, BFMs that model bus interactions between the FPGA and another device such as our ADC is written in an HDL.

Models representing the motor, inverter, current sensors, and incremental encoder are built in Simulink then generated to a SystemVerilog DPI component suitable for use in our FPGA simulator. The Simulink model is executed at 10 MHz which is necessary for the incremental encoder model to generate high fidelity waveforms. The high execution rate is not required for accurate modeling of the other components. If desired, a multi-rate model can be used. It should be noted though, this is not a performance case since the simulation of the FPGA design largely dominates simulation execution time.

The ADC BFM performs floating-point to fixed-point conversion on the values coming out of the Simulink model. The BFMs digital interface to the FPGA is built in accordance with the component datasheet and the model accurately depicts the conversion latency incurred by the component by emulating the conversion time.

The controller board interface software and USB interface are not simulated in the simulation environment. Instead, register values are forced at the start of the simulation. The software and USB interface will be tested during hardware in loop testing.

In a simulation, any variable in the design can be logged to a file for analysis later. Data is captured synchronously to the ADC sampling in the same way it is done in the physical implementation.

To provide a control performance baseline, we will be replacing the FPGA design with a double-precision floating-point control model for some test. In these tests, the latency will be matched with the FPGA design.

4.6 Data Acquisition and Control Software

The data acquisition and control software we developed allows us to configure the FPGA based design over USB. Values can be written to the controller at run time to change the target I_{dq} values in real-time. Internal variables and fault conditions can be logged and presented to the user in real-time.

Using the USB chip we chose, we can transfer up to 40 MB/s of data from the controller to the PC. This allows 80 16-bit variables to be transferred to the PC for logging at the 250 kHz maximum sample rate of the ADC. Plenty for our uses.

Working at these data rates when both plotting and logging data can be a challenge. Selection of the right tools to assemble the software system is critical in achieving 40 MB/s of throughput. We chose to construct the software in C++ (ISO (1998)) using the Qt application framework (The Qt Company (2019)) in conjunction with the FTD2XX library (FTDI Ltd. (2012)) from the manufacturer of the USB chip.

Headers are transmitted by the FPGA with the data. These headers are verified by the software to ensure samples are not lost.

4.7 Hardware in Loop Test Environment

Hardware in loop testing (HIL) is optional but highly recommended for systems where the improper operation of components can present a safety hazard or cause physical damage to the test setup. In our case, it is used to verify that our BFMs of circuit board components is accurate since we wrote those ourselves. A HIL test validates that our interpretation of the component datasheets is accurate by utilizing the actual components with the FPGA. It also provides an opportunity to test safety features such as overcurrent protection by intentionally triggering faults and provides an environment for testing our software.

Purpose-built HIL simulators exist that provide high fidelity stimulus for a controller (dSPACE GmbH (2019b)). We did not use purpose-built hardware but instead used the same dSPACE hardware that we use for dynamometer control. This results in some caveats that we must keep in mind. The MicroAutobox II is designed for real-time control applications and was only able to run the simulation model at 20 kHz, therefore, the incremental encoder signals cannot be generated if the virtual motor is spinning at high speed. This limits the virtual motor speeds to about 2 RPM, an unrealistically low value. However, this test still validates our BFMs, safety systems, and software functionality well. However, the results from these HIL test cannot be used beyond that.

The HIL model deployed on the MicroAutobox II is a modified version of the model used in the FPGA simulations depicted in Figure 4.13. Using the digital IO pins, the inverter gating signals are sampled, and the incremental encoder signals are generated. Then the current sensor signals are simulated using the digital to analog converters available on the MicroAutobox II.

4.8 Chapter Summary

In this chapter, we discussed the physical system we are testing on and the custom controller hardware we are using. After we describe the internal architecture of our FPGA design and detail how the tools from Chapter 3 are used. After, we detailed our simulation environments used for functional validation. The simulation environment will also be used for testing of additional controller variants that are impractical to implement for physical testing such as those where switching frequencies exceed the capability of our inverter. Then, we describe the configuration and data acquisition software necessary to configure our controller and collect data from it at run-time.

In the next chapter, we provide additional details on controller variants we will be testing. Then we describe our individual simulation and physical tests and provide our interpretation of the results.

Chapter 5

Test and Evaluation

So far, we discussed in Chapter 3, methodologies and tool development for automated fixed-point word-length (WL) optimization for field-programmable gate array (FPGA) implementation of equations. In Chapter 4, we discussed a potential use case, finite control set model predictive control (FCS-MPC) of an interior permanent magnet synchronous motor (IPMSM). In this use case, we used our tools to automatically generate an implementation of the motor modeling equations required in FCS-MPC. After, we presented a complete, physically realized, controller design that includes our automatically generated model implementation and other required components. Then we detail our simulation environment that provides another test platform for our controller. In Chapter 2, we discussed potential problems associated with precision loss and latency in control loops. We also highlighted that large FPGA designs cost more to fit into a real device.

In this chapter, we investigate achievable throughput, cost savings, and detrimental effects our optimizations can have on control performance.

5.1 Controller Variants

Using our tools, we generated motor model variants with different uncertainty requirements on the I_d and I_q prediction outputs for controllers targeting different sample rates.

Each modules' uncertainty requirement was selected to demonstrate gradual performance degradation after initial testing in simulation. We stop increasing the uncertainty requirement on controllers with the same sampling rate when our WL optimization tool is able to reach a multiplier element utilization close to zero. Further optimizations made past that point will not result in significant FPGA area savings.

An additional 16-bit fractional word-length (FWL) un-optimized variant is also included to give baseline area and performance numbers. We will confirm later in simulation that our 16-bit FWL variants exhibit performance that closely resembles a floating-point implementation. The un-optimized modules were analyzed using the same interval arithmetic (IA) techniques we discussed in Chapter 3 to get actual uncertainty intervals.

Table 5.1 lists the properties of the generated modules we will be testing with identifiers we use throughout this chapter. We list the actual uncertainty interval achieved by the WL optimizer calculated using IA since the uncertainty requirement is just a limit. Some of the uncertainty intervals calculated are higher than the requirement, this occurs when the SMT techniques are able to find better uncertainty bounds then IA alone. While we only tested sample rates achievable through integer division of the maximum, 250 kHz sample rate of our analog to digital converter (ADC) to ease data acquisition. Any sample rate up to 250 kHz is possible since the ADC we chose is timed by the FPGA as discussed in Section 4.3.

Table 5.1: IPMSM Nominal Parameters and Specifications

Identifier	Sample Rate	Uncertainty Requirement	Actual I_d Uncertainty	Actual I_q Uncertainty
MPC-10k-16bit	10 kHz		[-0.049, 0.049]	[-0.030, 0.034]
MPC-10k-0.10	10 kHz	[-0.10, 0.10]	[-0.054, 0.054]	[-0.076, 0.080]
MPC-10k-0.25	10 kHz	[-0.25, 0.25]	[-0.071, 0.071]	[-0.076, 0.215]
MPC-10k-0.50	10 kHz	[-0.50, 0.50]	[-0.354, 0.354]	[-0.296, 0.435]
MPC-10k-1.00	10 kHz	[-1.0, 1.0]	[-0.637, 0.637]	[-0.501, 0.640]
MPC-10k-2.00	10 kHz	[-2.0, 2.0]	[-0.627, 0.627]	[-0.890, 1.745]
MPC-25k-16bit	25 kHz		[-0.044, 0.044]	[-0.028, 0.029]
MPC-25k-0.10	25 kHz	[-0.10, 0.10]	[-0.087, 0.087]	[-0.029, 0.029]
MPC-25k-0.25	25 kHz	[-0.25, 0.25]	[-0.173, 0.173]	[-0.153, 0.154]
MPC-25k-0.50	25 kHz	[-0.50, 0.50]	[-0.451, 0.451]	[-0.295, 0.295]
MPC-25k-1.00	25 kHz	[-1.0, 1.0]	[-0.513, 0.513]	[-0.290, 0.633]
MPC-35k-16bit	35 kHz		[-0.044, 0.044]	[-0.046, 0.049]
MPC-35k-0.10	35 kHz	[-0.10, 0.10]	[-0.076, 0.076]	[-0.060, 0.064]
MPC-35k-0.25	35 kHz	[-0.25, 0.25]	[-0.193, 0.193]	[-0.096, 0.162]
MPC-35k-0.50	35 kHz	[-0.50, 0.50]	[-0.264, 0.264]	[-0.106, 0.350]
MPC-50k-16bit	50 kHz		[-0.054, 0.054]	[-0.028, 0.031]
MPC-50k-0.10	50 kHz	[-0.10, 0.10]	[-0.095, 0.095]	[-0.067, 0.067]
MPC-50k-0.25	50 kHz	[-0.25, 0.25]	[-0.166, 0.166]	[-0.152, 0.234]
MPC-125k-16bit	125 kHz		[-0.054, 0.054]	[-0.043, 0.044]
MPC-125k-0.10	125 kHz	[-0.05, 0.05]	[-0.054, 0.054]	[-0.043, 0.067]
MPC-125k-0.25	125 kHz	[-0.10, 0.10]	[-0.101, 0.101]	[-0.056, 0.080]

5.2 Reference Model Testing

Before we evaluate the control performance of our controller variants, we must compare our 16-bit FWL implementation to a double-precision floating-point implementation at all our selected FCS-MPC sample rates in simulation. We need to do this because it is impractical for us to implement a floating-point controller with the same latency as our fixed-point design for physical testing. After, our baseline performance evaluation for the other tests will be obtained with the 16-bit FWL designs. As such, we need to confirm in simulation that the 16-bit FWL designs perform similarly to the double-precision floating-point reference design. This will also confirm that our 16-bit FWL designs are good baselines for our area evaluations.

To obtain our double-precision floating-point performance numbers we modify the model in loop FPGA simulation environment described in Section 4.5. We replace the analog to digital converter (ADC) bus functional model (BFM) and FPGA design with a floating-point implementation of the control loop. We also bypass the incremental encoder model by using the position and velocity values from the physical simulation model directly. In the control model, we simulate the same sampling, computation and gate deadtime latency that exists in the FPGA design. We will test the controller implementations at 100 RPM and 500 RPM with a target I_d and I_q currents set to 0 A and 5 A respectively. In these tests, MPC-ref refers to our reference controller implemented in floating-point while MPC-16-bit refers to our fixed-point controller with an FWL of 16-bits. We will use root mean squared error (RMSE) to quantify deviation from the target current values during steady-state operation.

In Figures 5.1 and 5.2 we compare our resulting average switching frequency for both controller types across our selected controller sampling rates at 100 and 500 RPM.

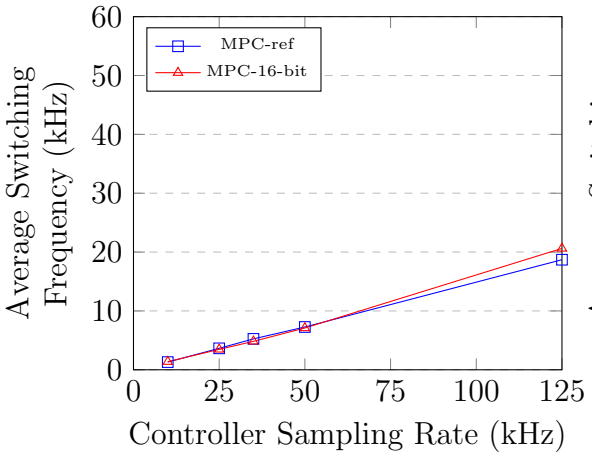

Figure 5.1: Average switching frequency of the reference controllers at 100 RPM

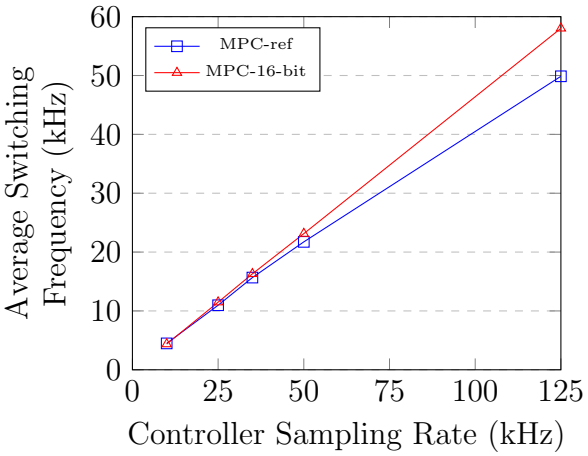

Figure 5.2: Average switching frequency of the reference controllers at 500 RPM

Next, we compare in Figures 5.3 and 5.4, the RMSE between I_d and its target value at 100 and 500 RPM respectively.

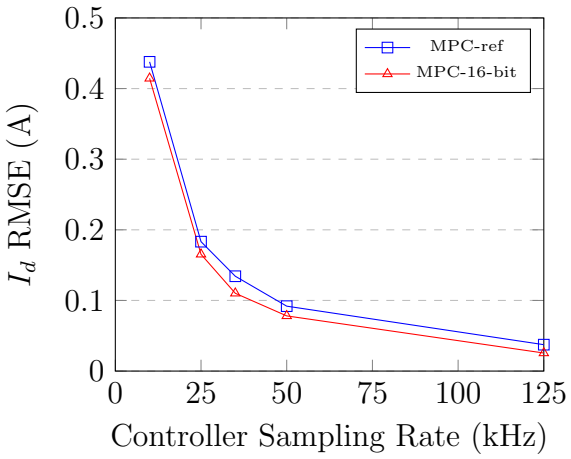

Figure 5.3: Reference controller I_d regulation accuracy at 100 RPM

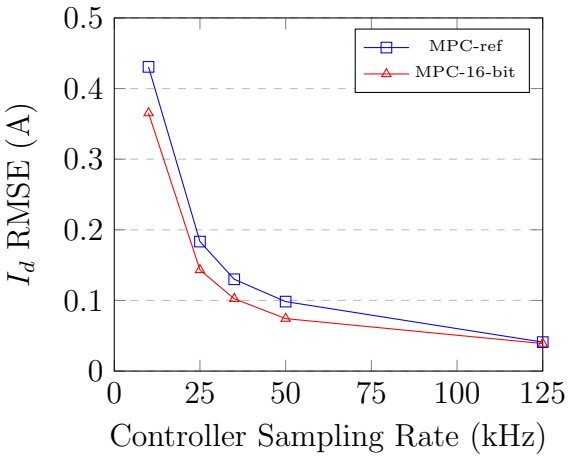

Figure 5.4: Reference controller I_d regulation accuracy at 500 RPM

After, in Figures 5.5 and 5.6 we compare the RMSE between I_q and its target value at 100 and 500 RPM respectively.

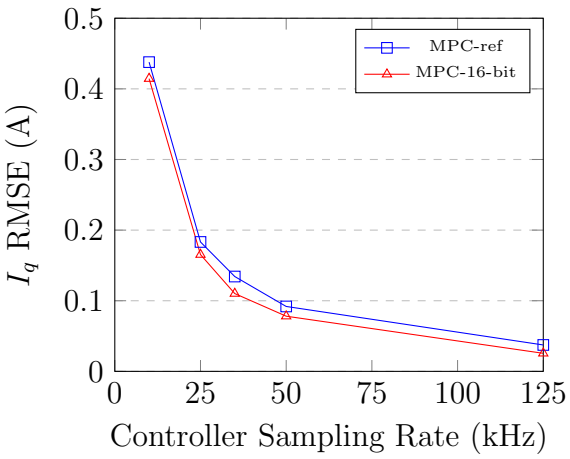

Figure 5.5: Reference controller I_q regulation accuracy 100 RPM

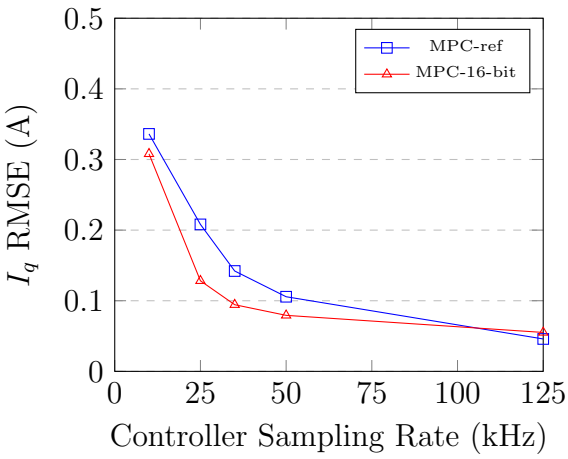

Figure 5.6: Reference controller I_q regulation accuracy at 500 RPM

We see that our fixed-point reference controller with an FWL of 16-bits provides comparable performance when compared to our floating-point reference controller. The fixed-point controller performs slightly better regarding current regulation but slightly worse when we look at switching frequency. There is a larger increase in switching frequency at 125 kHz for the fixed-point controller. The reasons for this change in performance will become clear when we explore further in simulation, it is due to the controller's higher sensitivity to prediction uncertainty when sampling rates are increased. While the performance of the two reference controllers is not identical, the similarities give us confidence that our fixed-point reference implementation can serve as a good baseline for FPGA area and controller performance evaluation.

5.3 FPGA WL Optimizer

In this section, look at how well our implementation cost estimation and WL optimization tool performed in terms of FPGA area reduction.

In Section 3.4 we discussed a cost estimation technique. Using the estimations from Equation 3.21, our optimizer can reduce the estimated costs as shown in Figure 5.7 when we allow a larger uncertainty interval. These variants were generated using a cooling rate (c_{rate}) of 0.95 in Equation 3.24, 100 random changes within the simulated annealing (SA) algorithm, and 100 repetitions of the optimization process to find better solutions. The SMT solver was used with a timeout of 10 seconds.

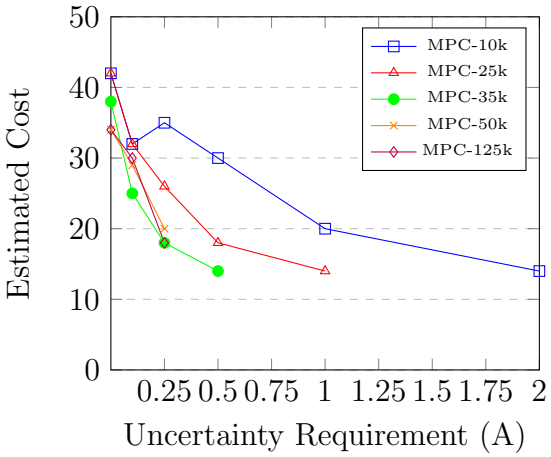

Figure 5.7: Estimated cost of our FCS-MPC implementations

We see an anomaly in the data with variant MPC-10k-0.25 where the best cost estimation found is higher than a variant with less uncertainty. Outcomes like this are possible because the SA algorithm used is guided by previous solutions it finds. With a larger allowable uncertainty interval, a good solution may be found early so less high-cost solutions are evaluated resulting in a search path through the solution space that leads to a worse outcome overall. We could have run the SA algorithm a

few more times and the random searches could eventually find a solution that fits our trend better, however, we elected to leave this anomalous data point to illustrate some potential problems with SA. We suggested in Section 3.5 that it might be desirable to repeat the optimization process multiple times, and this is why.

In Figure 5.8, we show the progress that is made in finding solutions with lower cost as the optimization process for different 10 kHz sample rate controllers is repeated 100 times. Each time the optimization process is repeated, 100 random changes are made to the base solution. Only points where a lower cost solution is found is shown. Initially, new solutions are found quickly, but after a while, better solutions are rarely, or not found at all. On an Intel Xeon E5-2620 v4 (Intel Corporation (2019b)) processor using one thread this optimization process usually takes about 10 seconds.

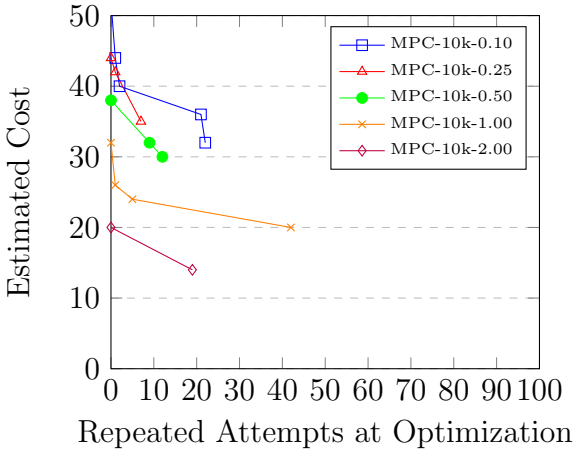

Figure 5.8: Estimated cost reduction as optimization is retried

In our testing, we found that the SMT techniques were only used rarely and only sometimes finds a better solution when the IA techniques cannot. We see in Table 5.1 only one variant has an uncertainty bound calculated by IA larger than the uncertainty requirement. This means only one variant in our set of controllers benefited from the SMT techniques.

We used a timeout of 10 seconds for the SMT solver. We found that increasing this timeout to 1000 seconds did not return better results. Since our SA algorithm only evaluates solutions where the cost is better than the best it has found so far, the SMT solver is often invoked less than 5 times in the 10000 opportunities it could be used with our settings. Therefore, even if the SMT solver does not return with a conclusive result in 10 seconds every time, the worst run-time we saw was less then 60 seconds. These outcomes suggest that affine arithmetic or just using IA alone might have been a better choice for the equations we are trying to optimize. However, more advanced techniques exist that can be used to reduce the run-time of SMT which could make it useful in our case (Eldib and Wang (2014)). Different outcomes as far as run-times should be expected if the optimization methods are used on different equations.

Now we have an idea of how the WL optimizer can reduce estimated cost and how long it takes to do so for our equations. We now must look at the actual FPGA resource utilization for our design variants after implementation by the FPGA tools since the optimizer just estimates cost.

In Figure 5.9 we show actual multiplier element usage. We see that our estimations are not entirely accurate due to other optimizations the FPGA tools perform but the optimization methods remain effective at reducing multiplier element utilization with an increase in allowable uncertainty.

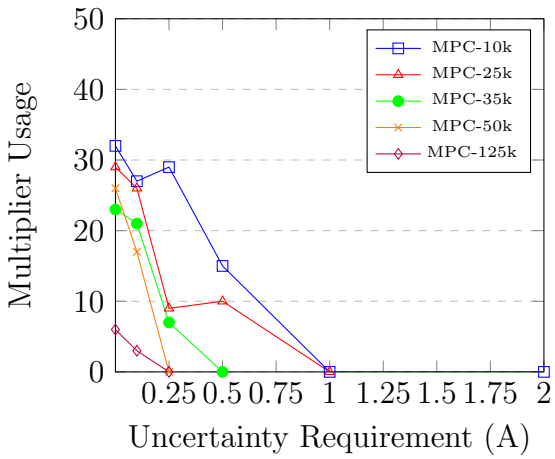

Figure 5.9: Multiplier element usage of our FCS-MPC implementations

We mentioned in Section 3.4 that we should expect LUT utilization to reduce with a lower estimated cost since cascading elements to form larger multipliers requires logic and smaller multipliers produce smaller results for our adders and subtractors to work with elsewhere in the circuit. We confirm this trend with the actual LUT utilization values shown in Figure 5.10.

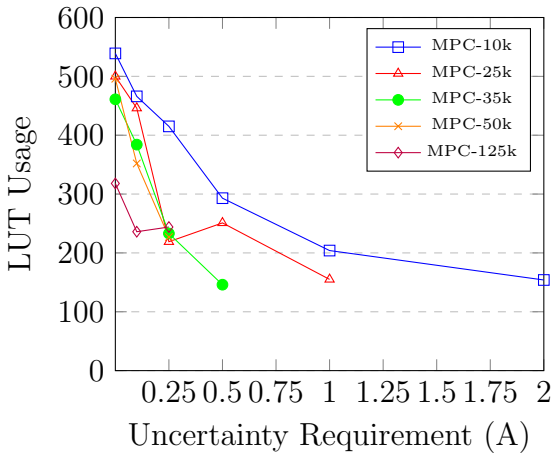

Figure 5.10: LUT usage of our FCS-MPC implementations

Throughout our results we see that the faster controllers require less area, this is because the constants in Equation 4.2 are shrinking as the sampling time (T_s) shrinks. Smaller values are easier to work with so the FPGA tools can make more optimizations.

We see that our tools and methods are capable of significantly reducing the resource utilization of implementations of the modeling equations required in our FCS-MPC. However, FPGA area reduction is not useful if we end up with a controller that behaves vastly worse then an un-optimized implementation. In the next section, we will investigate controller performance in simulation.

5.4 Performance Evaluation in Simulations

We evaluate the performance of the controller variants with different uncertainty requirements discussed in Section 5.1 against the reference fixed-point implementation tested in the previous section. Our goal is to find control performance degradation that might arise with increasing uncertainty in the prediction results within our FCS-MPC. To do this we use the simulation environment discussed in Section 4.5 at 100 and 500 RPM with a target I_d and I_q currents set to 0 A and 5 A respectively.

In Figure 5.11 and 5.12, we compare the resulting average switching frequency for all our fixed-point controller variants across our selected controller sampling rates. We see that initially, with small amounts of uncertainty, our controllers have about the same switching frequency as the reference fixed-point implementation. However, at some point, the switching frequency increases. The increase is less at higher speed since the controller is already switching quickly due to the more rapid changes in angular position.

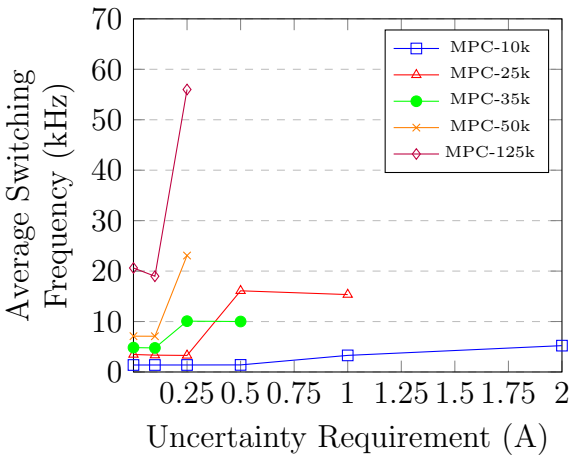

Figure 5.11: Average switching frequency for all our controller variants in simulation at 100 RPM

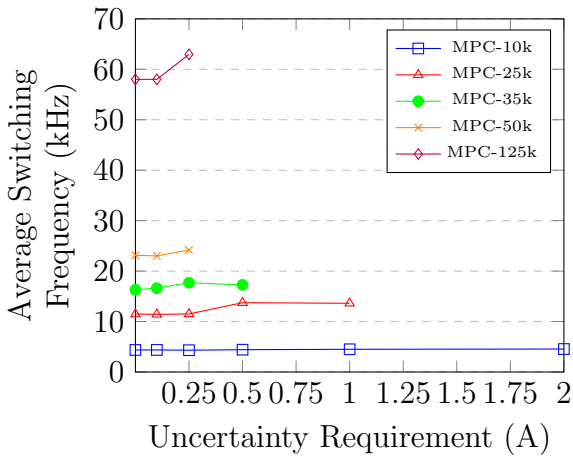

Figure 5.12: Average switching frequency for all our controller variants in simulation at 500 RPM

Next, we compare the RMSE between I_d and its target value in Figures 5.13 and 5.14.

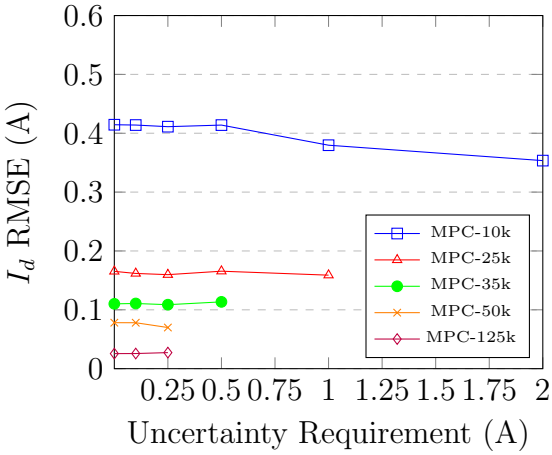

Figure 5.13: I_d regulation accuracy for all our controller variants in simulation at 100 RPM

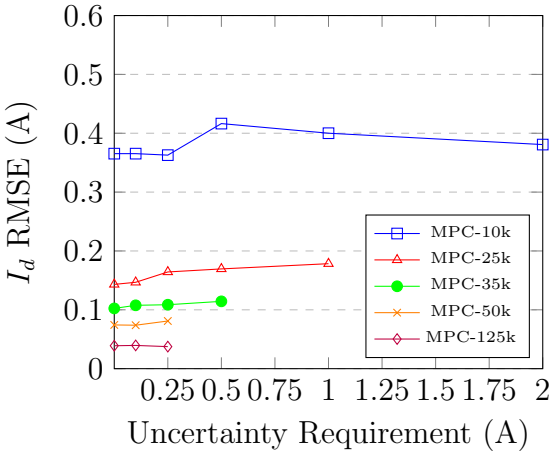

Figure 5.14: I_d regulation accuracy for all our controller variants in simulation at 500 RPM

After, we compare the RMSE between I_q and its target value in Figures 5.15 and 5.16.

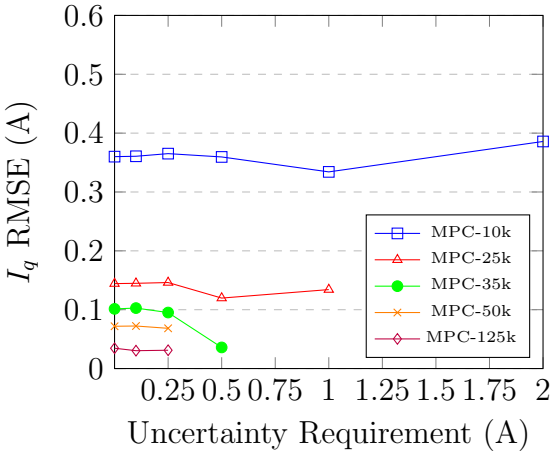

Figure 5.15: I_q regulation accuracy for all our controller variants in simulation at 100 RPM

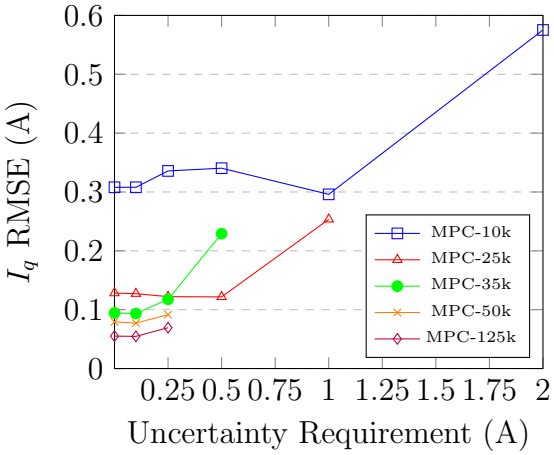

Figure 5.16: I_q regulation accuracy for all our controller variants in simulation at 500 RPM

We see a bigger change in the ability of the controller to regulate I_q current accurately with more uncertainty at high speed because the constants that are multiplied with rotational velocity is rounded to zero after WL optimization. In Equations 4.2 we see that exclusion of the portion of the equation that models the effects of velocity would cause the prediction equation to overestimate I_q current significantly. The result is that the controller ends up regulating I_q current with a negative bias. A similar effect is seen in I_d current regulation, but it is less severe since velocity has less of an effect on I_d.

We see switching frequency increase when uncertainty in the prediction results used within the FCS-MPC is increased past a certain point. In Figure 5.17, we compare the resulting I_q currents when using two different controller variants both trying to maintain I_q currents of 5 A at 100 RPM. One is our reference fixed-point controller and the other is one where we allowed the WL optimizer enough uncertainty to cause the rise in switching frequency we previously observed.

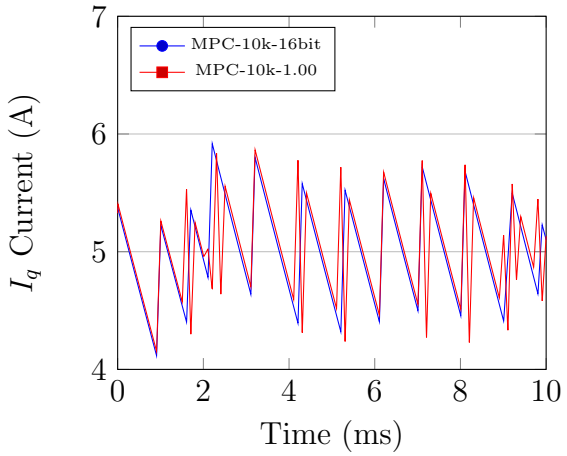

Figure 5.17: Unnecessary switching in controllers with too much uncertainty in simulation

We see that the controller exhibiting an increased switching frequency has a problem with what we will call erroneous ranking. This erroneous ranking phenomenon occurs when the inputs to the Find Min Cost step in the FCS-MPC algorithm shown previously in Figure 4.4 has enough errors to cause the inverter switch state that results in the minimum cost switch places with another switch state.

Next, we look at the transient response of our controller when the target I_q value is changed from 4 to 5A at 100 RPM. For all the controller variants, we did not see any increase in rise-time from 10 to 90% of the new target value. This is not unexpected since a change in the target values causes a dramatic increase in the cost function being evaluated by the controller. This makes selection of the best switch state very easy for the controller and even large amounts of uncertainty in the predictions will not compromise decision making. Once the new target values are reached however, we see the same behavior as we presented in steady state where erroneous ranking will appear for controller variants with too much uncertainty. In Figure 5.18 we show a comparison of the transient behaviour between two controller variants.

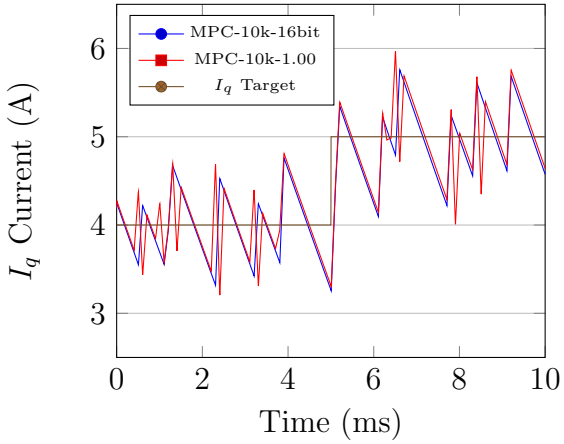

Figure 5.18: Transient behavior of our reference 10 kHz controller and one with a lot of uncertainty in simulation

5.5 Performance Evaluation with Physical Tests

Now that we have found some characteristic behavior of controllers with too much uncertainty in its predictions in simulation, we can proceed with physical testing. We want to test the same controllers we tested in simulation on the physical test setup described in Section 4.1 to confirm that we see the same characteristic trends. Unfortunately, due to the limited switching frequency that can be provided by our inverter, we limit testing to the controllers with a sampling frequency of 35 kHz and below. Our controllers are tasked with maintaining an I_d current of 0 A and an I_q current of 5 A at 100 RPM.

We begin by looking at the average switching frequency again in Figure 5.19. The last two variants of the 35 kHz sampling rate controllers with 0.25 and 0.5 A uncertainty limits are left out because the inverter switching frequency increased past our 30 kHz limit.

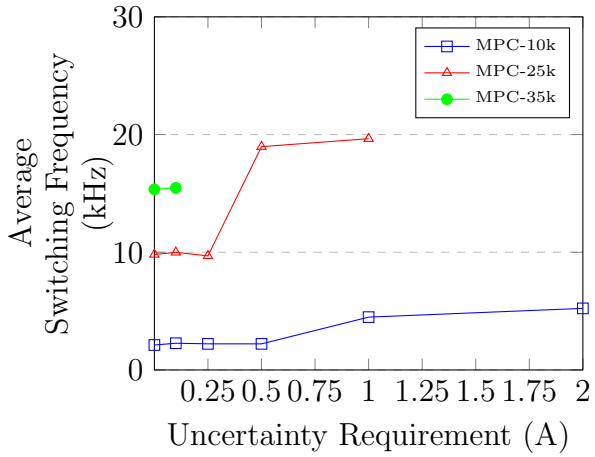

Figure 5.19: Average switching frequency of our physically tested controller variants

We see that the switching frequency is higher during physical testing then the results of our simulations in Figure 5.11 showed. This was primarily due to noise

being picked up by our current sensors from the dynamometer inverter. We were able to reproduce this in simulation by introducing a 100 mA peak to peak error on the I_d and I_q inputs to our motor model. Ignoring this, we see that the average switching frequency follows the same trend as we saw in simulations. Furthermore, not only is the trend the same, the controller variant with the smallest uncertainty requirements in each set of controllers exhibits the average switching frequency increase in both simulation and physical testing.

Next, we look at the I_d and I_q current regulation performance in Figures 5.20 and 5.21. Again, we quantify these performance numbers using RMSE.

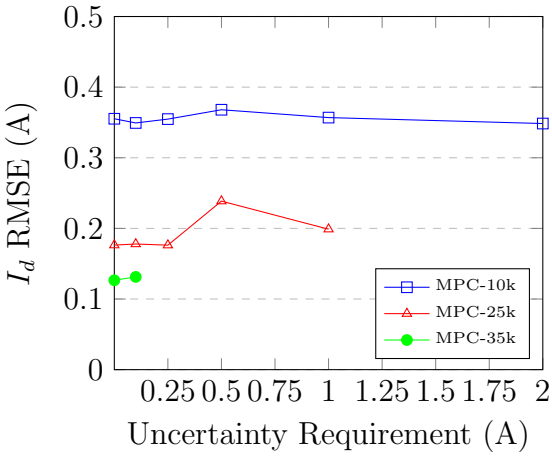

Figure 5.20: I_d regulation accuracy of our physically tested controller variants

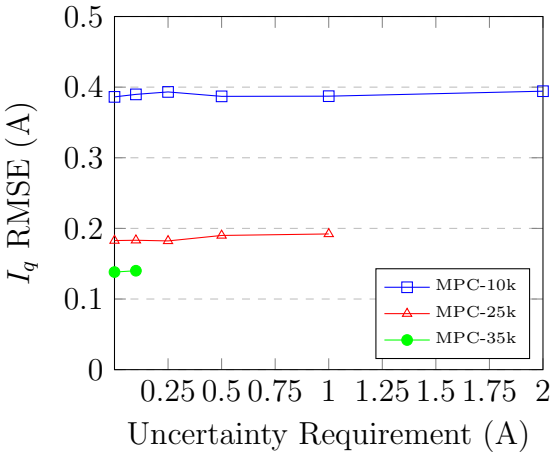

Figure 5.21: I_q regulation accuracy of our physically tested controller variants

Our current regulation measurements from the physical tests confirm what we saw in simulation. Current regulation ability is not significantly affected by uncertainty in the predictions used by the controller if we stay below the uncertainty threshold where we observe erroneous ranking and an increase in switching frequency. Once

we see switching frequency rise, we know that erroneous ranking is occurring and a change in current RMSE can occur as we saw in the simulation results.

In Figure 5.22 we see the unnecessary switching due to erroneous ranking as we saw in simulation. The reference fixed-point controller also exhibits some erroneous switching due to the current sensor noise we mentioned previously but the effects of the larger uncertainty bounds on the second controller are still visible. Unlike in simulation, the waveforms do not match up exactly since the test conditions are not as well-controlled, the position of the motor between simulations is exactly correlated with time while in physical tests this level of control is not possible.

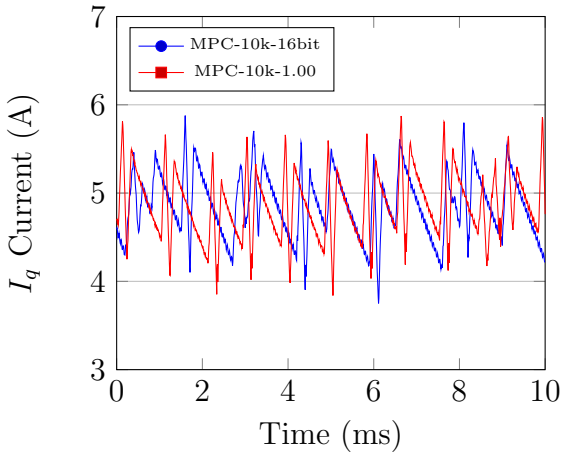

Figure 5.22: Unnecessary switching in controllers with too much uncertainty during a physical test

We look at the transient behavior of two of our controller variants on the physical test bench in Figure 5.23. We see that results agree with what we saw in simulation once again. The behavior during the transient event is very similar between a reference fixed-point controller and one with a lot of uncertainty in its predictions. Again, the waveforms do not line up as well when compared to simulation results because, during physical tests, the level of control over exact test conditions is much more difficult.

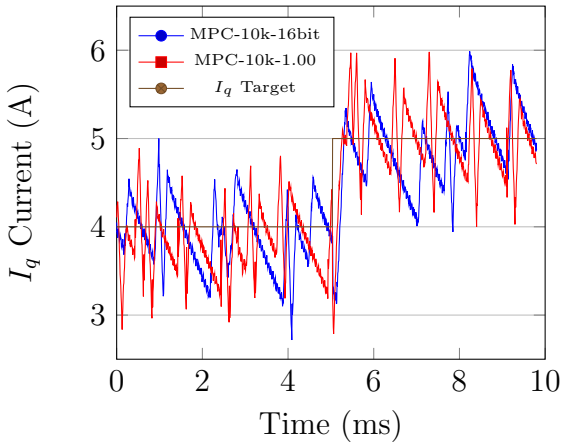

Figure 5.23: Transient behavior of our reference 10 kHz controller and one with a lot of uncertainty during a physical test

5.6 FPGA Area Savings Evaluation

We can now look at our results so far and draw some conclusions on FPGA area savings with the performance of the controller in mind. The increase in average switching frequency we observed as uncertainty increases too much is extremely undesirable in a controller. Therefore, we will exclude those variants when discussing area savings our tools were able to achieve.

Our physical testing and simulations validated that excessive switching begins with the same variants in each set of controller implementations that share the same sampling rate. Therefore, we will use our simulation results to select controller variants where control performance did not degrade significantly when compared to the fixed-point reference controller.

In Table 5.2 we compare the smallest controller variant that exhibited minimal performance degradation with the 16-bit FWL implementation. We see that we were able to significantly reduce multiplier element and LUT usage. Recall that our FPGA has a total of 90 multiplier elements so saving even a few can have a significant impact on what can fit in a small FPGA. For instance, it might make it possible to fit more than one motor modeling unit into the FPGA thereby multiplying our prediction throughput, lowering our computational latency, or enabling longer prediction horizons.

Table 5.2: FPGA Area Reduction

Identifier	Multipliers Used	LUTs Used	Multiplier Usage Decrease	LUT Usage Decrease
MPC-10k-16bit	42	539		
MPC-10k-0.50	30	293	12 (28.6%)	246 (45.64%)
MPC-25k-16bit	42	500		
MPC-25k-0.25	26	219	16 (38.1%)	281 (56.2%)
MPC-35k-16bit	38	461		
MPC-35k-0.10	25	384	13 (34.2%)	77 (16.7%)
MPC-50k-16bit	34	496		
MPC-50k-0.10	29	352	5 (14.7%)	144 (29.0%)
MPC-125k-16bit	34	318		
MPC-125k-0.10	30	236	4 (11.76%)	82 (25.8%)

5.7 Chapter Summary

In this chapter, we demonstrated in simulations that our tools, when used in conjunction with our FPGA controller design for an IPMSM can provide similar performance when compared to a floating-point design. We then show that our tools can reduce FPGA resource utilization using a cost estimator that estimates multiplier element usage. After, we demonstrated that the FCS-MPC algorithm in our use case can perform similarly to our reference design with some uncertainty in the predictions used within. This uncertainty allowance gives the potential for resource utilization reduction. Low resource utilization could allow the use of a smaller FPGA or duplication of the module that makes predictions, allowing for a higher prediction throughput or reduction in computational latency. It could also enable the use of FPGA based FCS-MPC in cost-sensitive applications.

In the next chapter, we provide our concluding remarks and ideas for future work.

Chapter 6

Conclusion and Future Work

In our introduction, we set out two major objectives in this thesis.

Our first objective was to develop and demonstrate methods that can be used to partially automate the creation of a prediction based motor controller capable of making millions of predictions per second. In Chapter 3, we described tools and methodologies that can be used to generate the FPGA designs that evaluated the prediction equations needed in an FCS-MPC. In Chapter 4, we demonstrated that our tools can be used to produce part of the logic required to apply FCS-MPC to IPMSM control. In Section 4.4.2 we calculate that with our design we can perform all 8 of the predictions required in one control cycle within 0.68 us. This means we achieved a prediction throughput of just under 12 million predictions per second. This performance is in line with what is achieved in an FCS-MPC implementation on FPGA for an induction motor where they were able to perform their predictions for 8 valid voltage vectors in 1.6 us (Kosan *et al.* (2018)). We should note that this shows that throughput is within a reasonable range, but comparisons between our implementations should not be too involved as the application, controller architecture,

and FPGA used are not identical. However, we can note that the design demonstrated by Kosan et al. could benefit from our optimization methodologies due to the similar requirements and constraints involved in our designs.

Our second objective was to apply our workflow to the development of a motor controller and show the FPGA resource utilization reduction our tools and methodologies can achieve and identify potential control performance issues that can arise if we push our design optimizations too far. Using the design, simulation, and test environments described in Chapter 4, we demonstrated between 10-38% reduction in multiplier element usage and 16-56% reduction in LUT usage for controller variants that exhibited degradation of control performance in Chapter 5. This could enable high-sample rate FCS-MPC implemented on FPGAs at lower price points. We also demonstrated that if we push our optimizations too far, the controller begins to make erroneous decisions which result in undesirable excessive state switching which would increase losses in the system.

We demonstrated the high throughput and low latency achievable on an FPGA. However, if software processors such as microcontrollers or digital signal processors are fast enough for your application of FCS-MPC we advise against using an FPGA due to challenges associated with FPGA development. With that said, when a real-time workload exceeds the capabilities of software processors, we showed that FPGAs are a compelling platform to use instead. They can be scaled up in performance easily through duplication of components such as the prediction module to enable even longer prediction horizons, more complex equations, and more switching states. The implementation optimizations we demonstrated can help by allowing duplication with less of an impact on the size of FPGA required.

Throughout our work, we emphasized that our optimizations are performed within guaranteed range and uncertainty bounds. Because of these guaranteed range bounds during integer word-length allocation, we never observe any overflow in the prediction model in any of our tests. We reiterate that it is important that overflow never occurs since overflow would result in grossly incorrect predictions which can cause catastrophic failure of the controller. The observant reader will notice that the uncertainty bounds that we guarantee are not required for the fractional word-length optimizations we make. The guarantees we make may even be invalidated because we leave out some sources of uncertainty and noise. However, the reason it is important we maintain this path instead of going for a simulation approach to error bounding is because it builds a foundation for future work. If others can develop a way to guarantee that erroneous ranking within a fixed-point FCS-MPC does not occur, there is a possibility that they require that we guarantee how far off the predictions within the controller are.

At this point, we see multiple paths for future work with different focuses.

In our experience, we found IA very effective and the improvements that could arise from using SMT techniques did not, pre-processing steps or more advanced SMT methods may improve this situation (Eldib and Wang (2014)). With that in mind, we encourage the reader to also investigate affine arithmetic (AA) techniques instead of SMT if they require better bounds then IA in a use case similar to ours. An additional step could also be investigated where IA, AA, or SMT methods could be selected depending on the equation being analyzed.

Those well trained in the analysis of control algorithms can explore ways in which to handle error in the predictions required in an FCS-MPC. They could also explore

ways in which to find upper limits on prediction uncertainty before detrimental effects on control performance arise.

Those with experience developing high-level synthesis (HLS) tools can propagate some of the methodologies we demonstrated to the code generation and HLS tools already being used by control system designers. In the literature on FCS-MPC for power electronics and motor control, there is often mention that higher computational throughput may be desired (Rodriguez *et al.* (2013)). Our work and others (Kosan *et al.* (2018), Wendel *et al.* (2017a), Hamidi *et al.* (2017), Wendel *et al.* (2017b)) have shown that the answer may be implementation on FPGAs. Cost of large FPGAs can be a problem for mass production. Therefore, we targeted methodologies to reduce cost. Another problem we did not address fully, however, is the time and effort required to design for FPGA implementation. As such, we need the methods we discussed in our work propagated to the tools control systems designers already use. The thousands of lines of HDL code required to build the design we presented in Chapter 4 may not acceptable in the general sense. We cannot always expect a control systems designer to also be an expert in FPGA design. For our work, and the work that others have done on FPGA based power electronics or motor FCS-MPC to have a big impact in this area, there needs to be more tool automation to lower the barrier to entry.

References

Bajaj, R. (2016). *Exploiting DSP Block Capabilities in FPGA High Level Design Flows*. Ph.D. thesis, Nanyang Technological University.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., and Tinelli, C. (2011). CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer. Snowbird, Utah.

BEI Sensors SAS (2019). ATEX Incremental Encoder for ATEX Zone 2 and 22, DHO5S Range. `http://www.beisensors.com/pdfs/dho5-atex-optical-incremental-encoder.pdf`. [Online; Accessed July-2019].

Borriello, G. and Detjens, E. (1988). High-level synthesis: current status and future directions. In *25th ACM/IEEE, Design Automation Conference.Proceedings 1988.*, pages 477–482.

Brown, S., , Brown, S., and Rose, J. (1996). Architecture of fpgas and cplds: A tutorial. *IEEE Design and Test of Computers*, **13**, 42–57.

Cadence Design Systems, Inc. (2019a). Incisive Enterprise Simulator. http://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html. [Online; Accessed August-2019].

Cadence Design Systems, Inc. (2019b). Stratus High-Level Synthesis. http://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html. [Online; Accessed July-2019].

Casseau, E. and Le Gal, B. (2009). High-level synthesis for the design of fpga-based signal processing systems. In *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 25–32.

Che, S., Li, J., Sheaffer, J. W., Skadron, K., and Lach, J. (2008). Accelerating compute-intensive applications with gpus and fpgas. In *2008 Symposium on Application Specific Processors*, pages 101–107.

Cree, Inc. (2015). C2M0025120D Silicon Carbide Power MOSFET. http://www.wolfspeed.com/media/downloads/161/C2M0025120D.pdf. [Online; Accessed August-2019].

Darulova, E. and Kuncak, V. (2013). On sound compilation of reals. *CoRR*, **abs/1309.2511**.

Deng, L., Sobti, K., and Chakrabarti, C. (2008). Accurate models for estimating area and power of fpga implementations. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1417–1420.

Dong-U Lee, Gaffar, A. A., Mencer, O., and Luk, W. (2005). MiniBit: bit-width optimization via affine arithmetic. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 837–840.

dSPACE GmbH (2019a). MicroAutoBox II. `http://www.dspace.com/en/inc/home/products/hw/micautob/microautobox2.cfm`. [Online; Accessed August-2019].

dSPACE GmbH (2019b). SCALEXIO Rack Overview. `http://www.dspace.com/en/inc/home/products/system/scalexio/scalexio-rack-system.cfm`. [Online; Accessed August-2019].

Duralová, E. (2014). *Programming with Numerical Uncertainties*. Ph.D. thesis, École polytechnique fédérale de Lausanne.

Eldib, H. and Wang, C. (2014). An smt based method for optimizing arithmetic computations in embedded software code. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **33**(11), 1611–1622.

Erick L. Oberstar (2007). Fixed-point representation & fractional math. `http://www.superkits.net/whitepapers/Fixed%20Point%20Representation%20&%20Fractional%20Math.pdf`. [Online; Accessed June-2019].

Fang, C. F., Rutenbar, R. A., and Tsuhan Chen (2003). Fast, accurate static analysis for fixed-point finite-precision effects in dsp designs. In *ICCAD-2003. International*

Conference on Computer Aided Design (IEEE Cat. No.03CH37486), pages 275–282.

FTDI Ltd. (2012). D2XX Programmer's Guide. `http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer's_Guide(FT_000071).pdf`. [Online; Accessed July-2019].

FTDI Ltd. (2019). FT232H - Hi-Speed Single Channel USB UART/FIFO IC. `https://www.ftdichip.com/Products/ICs/FT232H.htm`. [Online; Accessed May-2019].

Gabriel, R., Leonhard, W., and Nordby, C. J. (1980). Field-oriented control of a standard ac motor using microprocessors. *IEEE Transactions on Industry Applications*, **IA-16**(2), 186–192.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, **13**(5), 533–549.

Hace, A. and Čurkovič, M. (2018). A novel divisionless mt-type velocity estimation algorithm for efficient fpga implementation. *IEEE Access*, **6**, 48074–48087.

Hamidi, A., Karimi, S., Ahmadi, A., and Ahmadi, M. (2017). Digital FCS-MP control of an ac-dc power converter to improve dynamic response. In *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 326–329.

Han, K., Olson, A. G., and Evans, B. L. (2006). Automatic Floating-Point to Fixed-Point Transformations. In *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, pages 79–83.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

IEEE Computers Society (2006). IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590.

IEEE Computers Society (2008). IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70.

IEEE Computers Society (2009). IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626.

IEEE Computers Society (2018). IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315.

Intel (2019). Intel MAX 10 FPGA. `https://www.intel.ca/content/www/ca/en/products/programmable/fpga/max-10.html`. [Online; Accessed May-2019].

Intel Corporation (2004). Implementing Multipliers in FPGA Devices. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an306.pdf`. [Online; Accessed July-2019].

Intel Corporation (2019a). Intel High Level Synthesis Compiler. `http://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`. [Online; Accessed July-2019].

Intel Corporation (2019b). Intel Xeon Processor E5-2620 v4. `https://ark.intel.com/content/www/us/en/ark/products/92986/`

intel-xeon-processor-e5-2620-v4-20m-cache-2-10-ghz.html. [Online; Accessed August-2019].

International Energy Agency (2018). Global EV Outlook 2018. http://webstore.iea.org/global-ev-outlook-2018/. [Online; Accessed May-2018].

ISO (1998). *ISO/IEC 14882:1998: Programming languages — C++*. Available in electronic form for online purchase at http://webstore.ansi.org/ and http://www.cssinfo.com/.

J. Rezaie, M. Gholami, R. F. T. A. K. S. (2007). Interior permanent magnet synchronous motor (ipmsm) adaptive genetic parameter estimation.

Jin, S., Cho, J., Pham, X. D., Lee, K. M., Park, S., Kim, M., and Jeon, J. W. (2010). Fpga design and implementation of a real-time stereo vision system. *IEEE Transactions on Circuits and Systems for Video Technology*, **20**(1), 15–26.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *SCIENCE*, **220**(4598), 671–680.

Kosan, T., Talla, J., Janous, S., and Blahnik, V. (2018). Fpga-based accelerator for model predictive control of induction motor drive. In *2018 18th International Conference on Mechatronics - Mechatronika (ME)*, pages 1–6.

Krishnan, R. (2001). *Electric Motor Drives*. Prentice Hall.

Lapotre, V., Coussy, P., Chavet, C., Wouafo, H., and Danilo, R. (2013). Dynamic branch prediction for high-level synthesis. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–6.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA.

LEM International SA (2009). CAS/CASR/CKSR series Current Transducers. `http://www.lem.com/sites/default/files/marketing/lem%20leaflet%20cas%20casr%20cksr.pdf`. [Online; Accessed August-2019].

Linderman, M. D., Ho, M., Dill, D. L., Meng, T. H., and Nolan, G. P. (2010). Towards program optimization through automated analysis of numerical precision. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 230–237, New York, NY, USA. ACM.

Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. (2007). Netfpga–an open platform for gigabit-rate network switching and routing. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, pages 160–161.

Maplesoft (2019). MapleSim. `http://www.maplesoft.com/products/maplesim/`. [Online; Accessed August-2019].

Martorell, H. and Kapre, N. (2012). Fx-score: A framework for fixed-point compilation of spice device models using gappa++. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 77–84.

MathWorks Inc. (2019a). HDL Coder. `http://www.mathworks.com/products/hdl-coder.html`. [Online; Accessed July-2019].

MathWorks Inc. (2019b). HDL Verifier. http://www.mathworks.com/products/hdl-verifier.html. [Online; Accessed July-2019].

Maxim Integrated (2019a). MAX11047 - 4-/6-/8-Channel, 16-/14-Bit, Simultaneous-Sampling ADCs. https://www.maximintegrated.com/en/products/analog/data-converters/analog-to-digital-converters/MAX11047.html. [Online; Accessed May-2019].

Maxim Integrated (2019b). MAX3094E. http://www.maximintegrated.com/en/products/interface/transceivers/MAX3094E.html. [Online; Accessed July-2019].

Melquiond, G. (2019). Gappa. http://gappa.gforge.inria.fr/. [Online; Accessed May-2019].

Mentor, a Siemens Business (2019a). Catapult High-Level Synthesis. http://www.mentor.com/hls-lp/catapult-high-level-synthesis/. [Online; Accessed July-2019].

Mentor, a Siemens Business (2019b). ModelSim. http://www.mentor.com/products/fv/modelsim/. [Online; Accessed August-2019].

Meredith, M. (2004). A look inside behavioral synthesis. http://www.eetimes.com/document.asp?doc_id=1217645. [Online; Accessed July-2019].

Microsoft Research (2019). Z3 Theorem Prover. https://github.com/Z3Prover/z3. [Online; Accessed June-2019].

Monmasson, E. and Cirstea, M. N. (2007). Fpga design methodology for industrial control systems—a review. *IEEE Transactions on Industrial Electronics*, **54**(4), 1824–1842.

Monmasson, E., Idkhajine, L., Cirstea, M. N., Bahri, I., Tisan, A., and Naouar, M. W. (2011). Fpgas in industrial control applications. *IEEE Transactions on Industrial Informatics*, **7**(2), 224–243.

Nalakath, S. (2018). *Robust Position Sensorless Model Predictive Control for Interior Permanent Magnet Synchronous Motor Drives*. Ph.D. thesis, McMaster University.

National Instruments (2019a). Explore the Hardware-in-the-Loop Platform. `http://www.ni.com/hil/platform/`. [Online; Accessed August-2019].

National Instruments (2019b). NI FPGA. `https://www.ni.com/fpga/`. [Online; Accessed July-2019].

National Instruments (2019c). What is LabVIEW? `http://www.ni.com/en-ca/shop/labview.html`. [Online; Accessed August-2019].

Ned Mohan, Tore M. Undeland, W. P. R. (1995). *Power Electronics*. Wiley.

Osborne, W. G., Cheung, R. C. C., Coutinho, J. G. F., Luk, W., and Mencer, O. (2007). Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *2007 International Conference on Field Programmable Logic and Applications*, pages 617–620.

Paulin, P. G. and Knight, J. P. (1989). Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **8**(6), 661–679.

Pillay, P. and Krishnan, R. (1991). Application characteristics of permanent magnet synchronous and brushless dc motors for servo drives. *IEEE Transactions on Industry Applications*, **27**(5), 986–996.

Rodriguez, J., Kazmierkowski, M. P., Espinoza, J. R., Zanchetta, P., Abu-Rub, H., Young, H. A., and Rojas, C. A. (2013). State of the art of finite control set model predictive control in power electronics. *IEEE Transactions on Industrial Informatics*, **9**(2), 1003–1016.

Schwartz, L., Wei, M., Morrow, W., Deason, J., Schiller, S. R., Leventis, G., Smith, S., Leow, W. L., Levin, T., Plotkin, S., Zhou, Y., and Teng", J. (2017). Electricity end uses, energy efficiency, and distributed energy resources baseline. `http://emp.lbl.gov/publications/electricity-end-uses-energy`. [Online; Accessed May-2018].

Shien-Ru Ko and Wen-Shyong Yu (2000). Stability analysis of state regulator systems with finite word length effects. In *2000 26th Annual Conference of the IEEE Industrial Electronics Society. IECON 2000. 2000 IEEE International Conference on Industrial Electronics, Control and Instrumentation. 21st Century Technologies*, volume 2, pages 1422–1427 vol.2.

Shirabe, K., Swamy, M. M., Kang, J., Hisatsune, M., Wu, Y., Kebort, D., and Honea, J. (2014). Efficiency comparison between si-igbt-based drive and gan-based drive. *IEEE Transactions on Industry Applications*, **50**(1), 566–572.

SRI International (2018). Yices SMT Solver. `http://yices.csl.sri.com/`. [Online; Accessed June-2019].

Stolfi, J. and Henrique De Figueiredo, L. (1998). Self-validated numerical methods and applications.

Synopsys, Inc. (2009). Synopsys Introduces Synphony High Level Synthesis. http://news.synopsys.com/index.php?s=20295&item=123096. [Online; Accessed July-2019].

Synopsys, Inc. (2019). VCS. http://www.synopsys.com/verification/simulation/vcs.html. [Online; Accessed August-2019].

The MathWorks, Inc. (2019a). fxpopt. https://www.mathworks.com/help/fixedpoint/ref/fxpopt.html. [Online; Accessed June-2019].

The MathWorks, Inc. (2019b). Model-Based Design with Simulink. http://www.mathworks.com/help/simulink/gs/_model-based-design.html. [Online; Accessed August-2019].

The MathWorks, Inc. (2019c). Simulink. http://www.mathworks.com/products/simulink.html. [Online; Accessed August-2019].

The Qt Company (2019). Qt. http://www.qt.io/. [Online; Accessed August-2019].

USB-IF (2019). USB-IF. http://www.usb.org/. [Online; Accessed July-2019].

Wang, T. S., Guo, J. G. Z. Y. G., Lei, G., and Xu, W. (2011). Simulation and experimental studies of permanent magnet synchronous motor control methods. In *2011 International Conference on Applied Superconductivity and Electromagnetic Devices*, pages 252–255.

Wendel, S., Dietz, A., and Kennel, R. (2017a). Area-efficient fpga implementation of finite control set model predictive current control. In *2017 IEEE Southern Power Electronics Conference (SPEC)*, pages 1–6.

Wendel, S., Dietz, A., and Kennel, R. (2017b). Fpga based finite-set model predictive current control for small pmsm drives with efficient resource streaming. In *2017 IEEE International Symposium on Predictive Control of Electrical Drives and Power Electronics (PRECEDE)*, pages 66–71.

Xilinx Inc. (2019). Vivado High-Level Synthesis. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`. [Online; Accessed July-2019].

Yang, Y., Wen, H., and Li, D. (2017). A fast and fixed switching frequency model predictive control with delay compensation for three-phase inverters. *IEEE Access*, **5**, 17904–17913.

Ye, D. and Kapre, N. (2014). Mixfx-score: Heterogeneous fixed-point compilation of dataflow computations. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 206–209.