OPTIMIZED EMBEDDED ARCHITECTURES FOR MODEL PREDICTIVE CONTROL ALGORITHMS FOR BATTERY CELL MANAGEMENT SYSTEMS IN ELECTRIC VEHICLES

by

ANNE KULBITSKI MADSEN

MSEE, Naval Postgraduate School, 1995
BS, United States Naval Academy, 1990

A dissertation submitted to the Graduate Faculty of the University of Colorado Colorado Springs in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Department of Electrical and Computing Engineering

2020
This dissertation for the Doctor of Philosophy degree by

Anne Kulbitski Madsen

has been approved for

Department of Electrical and Computing Engineering

by

Darshika G. Perera, Chair

T.S. Kalkur

M. Scott Trimboli

Heather Song

Kristen Walcott

Date: 8 October 2020
Madsen, Anne Kulbitski (Ph.D., Electrical Engineering)

Optimized Embedded Architectures for Model Predictive Control Algorithms for Battery Cell Management Systems in Electric Vehicles

Dissertation directed by Assistant Professor Darshika G. Perera

ABSTRACT

With the ever-growing concerns about carbon emissions and air pollution throughout the world, electric vehicles (EVs) and hybrid electric vehicles (HEVs) are some of the most viable options for clean transportation. EVs are typically powered by battery packs created from a large number of individual cells such as lithium-ion cells. In order to enhance the durability and prolong the useful life of the battery pack, it is imperative to monitor and control the battery packs at the cell level. The best techniques to provide maximum cell performance, longevity and safety are also the techniques that are the most computationally expensive and least suited towards the constrained resources of embedded systems. Advances in the power and computing capability of both microprocessors and field programmable gate arrays (FPGA) make it possible to bring the computationally expensive algorithms and methods into embedded systems.

Model predictive control is an adaptive control approach that uses models to predict the outcome of a control move and then optimize the control move based on the system constraints. This control approach incorporates prediction, optimization and constraints to determine the best input to achieve a desired output and is computationally expensive. The ability to incorporate constraints allows reduction of the safety margins, increasing the capacity or productivity of a system. This approach is well established in industrial applications and is very desirable for use in battery management systems. Currently in MPC for battery cell management there are two popular model approaches for battery cell management, the Equivalent Circuit model (ECM) and the physics-based model (PBM). ECM has the advantage of simplicity and uses an empirical approach
to describing battery cell interactions. The disadvantage to ECM is the limit on the operating range of the model, though the model operates very well within the boundaries set by the data collection parameters ECM does not perform well outside the boundaries. The ECM also does not model the internal electrochemical reactions of the battery that lead to battery degradation. The PBM, being based on first order principals that describe the electrochemical reactions of the battery, is more flexible and can handle a wider range of operating conditions and supports monitoring and control of the internal reactions that lead to degradation. The challenge of PBM is reducing the complexity of the algorithm so that it can be implemented in a resource constrained environment while maintaining the ability to accurately model the battery. As it is difficult to physically measure the internal electrochemical reactions of the batteries while in use, an observer is required to estimate the state of the reactions in support of the control algorithm. The non-linear nature of the PBM adds complexity to both the MPC approach and the observer. MPC using PBM has long been considered too complex for resource constrained systems.

Our goal in this research is to investigate and architect embedded systems software and hardware co-designs that support the computationally expensive model predictive control algorithms, models, and approaches that maximize both performance and longevity of battery packs at the cell level previously considered too computationally complex for embedded systems applications.

The first contribution of this work introduces one software and two hardware embedded architectures that address the computational complexity of ECM based MPC for the at-rest fast charging of an HEV battery. The second contribution introduces novel and efficient software and hardware embedded architectures for a non-linear observer based on the Extended Kalman filter using a physics-based models based on first order principals. These architectures demonstrate the utility and accuracy of the state space PBM developed using the discrete realization algorithm (DRA). Experiments are run on the embedded systems that show the capability to maintain the advantages of PB approaches while using minimal resources. Results are provided that show the
resource constrained architectures retain the accuracy and flexibility advantage of the physics-based models using the minimum resources required for mobile battery management.

The final contributions are embedded architectures for a unique adaption of model predictive control to create a smart sensor for battery cell management that focuses on preventing cell degradation. These architectures incorporate the PB EKF and a PB MPC on a single system. Two architectures are presented, a software architecture on a 32-bit 128KB microcontroller and a hardware architecture on an FPGA. Experiments run on the actual embedded systems demonstrate that the current design approaches provide excellent performance and easily support the required control intervals.

In each case the experiments are run on the actual embedded systems and demonstrate that the current state of research for physics-based battery cell management can be achieved and are becoming suitable for resource constrained embedded systems.
ACKNOWLEDGEMENTS

First, I wish to thank my dissertation committee. I am very grateful for the time and resources you were willing to dedicate to helping me succeed in this endeavor. Each of your contributions, large and small, enhanced and improved my experience on this journey. It was a pleasure working with you.

Deepest thanks go to my advisor and Committee Chair, Dr. Darshika Perera. I am constantly amazed and grateful for your dedication to my success. Thank you for being so generous with your time and energy. You have been with me every step of this journey, offering guidance, support, encouragement and the occasional kick to keep me on task. You have been a tremendous role model and I am very proud to call you my mentor.

A special thank you to Dr. Scott Trimboli and his graduates, Dr. Marcello Xavier and Dr. Gustavo Florentino for being so generous with your research, ideas, and approaches. As well as always finding time to answer questions. I could not have done this without you.

Finally, thank you to my wonderful husband and amazing children, none of this would have been possible without your love and support.
# TABLE OF CONTENTS

CHAPTER

1 INTRODUCTION ..................................................................................................................... 1

   1.1 Overview ......................................................................................................................... 1
   1.2 Research Objectives ......................................................................................................... 8
   1.3 Contributions .................................................................................................................. 8
   1.4 Organization ................................................................................................................... 9

2 BACKGROUND ..................................................................................................................... 10

   2.1 Overview ....................................................................................................................... 10
   2.2 Battery Management Systems (BMS) .......................................................................... 10
       2.2.1 Battery Models ....................................................................................................... 10
       2.2.2 Model Predictive Control ...................................................................................... 17
       2.2.3 Optimization ......................................................................................................... 21
       2.2.4 Hildreth’s Quadratic Programming Technique .................................................... 23
   2.3 Embedded Systems ....................................................................................................... 25

3 FAST CHARGE MPC DESIGN ............................................................................................. 27

   3.1 Background: Model Predictive Control for the Fast-Charge MPC .............................. 27
   3.2 Dynamic Model ............................................................................................................. 28
       3.2.1 Prediction of State and Output Variables ............................................................... 31
       3.2.2 Optimization ......................................................................................................... 33
       3.2.3 Applying Control Signal ....................................................................................... 35
   3.3 Design Approach and Development Platform ............................................................. 35
   3.4 Embedded Hardware and Software Architectures for MPC ....................................... 38
       3.4.1 Embedded Software Architecture ........................................................................ 40
       3.4.2 Embedded Hardware Designs .............................................................................. 42
3.5 Experimental Results and Analysis

3.5.1 Functional Verification – Comparison with Baseline Model

3.5.2 Performance Metrics – Execution Time and Resource Utilization

3.5.3 Analysis of Existing Works on Embedded Designs for MPC

3.6 Conclusion

4 EXTENDED KALMAN FILTER FOR THE REDUCED-ORDER PBM

4.1 Architecture Overview

4.1.1 Module 1

4.1.2 Module 2

4.1.3 Module 3

4.1.4 Module 4

4.1.5 Module 5

4.1.6 Stages 6 and 7

4.2 Experimental Results and Analysis

4.2.1 Functional Verification

4.2.2 Performance Analysis: Execution Time and Speedup

4.2.3 Resource Utilization

4.2.4 Comparison with Related Work

4.3 Conclusion

5 EMBEDDED SOFTWARE ARCHITECTURE FOR PB MPC AS A SMART SENSOR

5.1 Architecture Overview

5.2 Background – PBM, EKF, and MPC

5.2.1 Physics-based Model (PBM)

5.2.2 Extended Kalman Filter (EKF) observer

5.2.3 MPC

5.3 Embedded Software Architecture for Physics-Based MPC
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.1</td>
<td>EKF</td>
<td>128</td>
</tr>
<tr>
<td>5.3.2</td>
<td>MPC</td>
<td>132</td>
</tr>
<tr>
<td>5.4</td>
<td>Experiments and Results</td>
<td>134</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Functional Verification</td>
<td>134</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Performance analysis: execution times, code Size</td>
<td>137</td>
</tr>
<tr>
<td>5.5</td>
<td>Related Work – PB-EKF-MPC</td>
<td>138</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusion</td>
<td>139</td>
</tr>
<tr>
<td>6.1</td>
<td>Architecture Overview</td>
<td>140</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Stage 7</td>
<td>142</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Stage 8</td>
<td>150</td>
</tr>
<tr>
<td>6.2</td>
<td>Experiments and Results</td>
<td>150</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Functional Verification</td>
<td>153</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Performance Analysis: Execution Time and Speedup</td>
<td>157</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Resource Utilization</td>
<td>158</td>
</tr>
<tr>
<td>6.3</td>
<td>Conclusion</td>
<td>159</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>160</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>161</td>
</tr>
</tbody>
</table>

BIBLIOGRAPHY ........................................................................................................ 166
LIST OF TABLES

Table 1: Software algorithm for Fast-charge MPC ................................................................. 41
Table 2: Temperature regions for Cubic Spline ................................................................. 45
Table 3: Determining the Open Circuit Voltage from the State of Charge ..................... 62
Table 4: HQP algorithm ....................................................................................................... 70
Table 5: Execution times: embedded HW and SW designs and baseline Matlab model ...... 88
Table 6: Resource utilization: embedded HW_v2 versus embedded HW_v1 ................. 90
Table 7: Software algorithm to find the \( \beta \) indices and producing the scaling factors .... 100
Table 8: Performance comparison: Embedded HW, SW, and baseline Matlab ............... 119
Table 9: Resource utilization: embedded hardware ............................................................. 119
Table 10: Generation of output matrices ........................................................................... 131
Table 11: Embedded software code size breakdown .......................................................... 137
Table 12: Worst-Case Time & No. of Battery Cells Supported per Interval ..................... 138
Table 13: Smart Sensor Performance comparison: Embedded HW and Embedded SW ..... 157
Table 14: Resource utilization comparison: embedded hardware .................................... 158
LIST OF FIGURES

Figure 1: Equivalent circuit model (ECM) for battery cell charging [6] ......................................... 12
Figure 2: Cell schematic showing variables and their position with respect to x or z [41] ............ 15
Figure 3: System-level architecture for Fast-charge MPC .......................................................... 37
Figure 4: Four high-level stages of the Fast-charge MPC algorithm ........................................... 39
Figure 5: Functional/data flow for Stage 1 .................................................................................. 42
Figure 6: Parameter Module for HW_v2 .................................................................................... 45
Figure 7: Augmented model for HW_v1 ...................................................................................... 46
Figure 8: Internal architecture for Φ for HW_v1 and HW_v2 ....................................................... 48
Figure 9: Organization of ΦÃ ....................................................................................................... 48
Figure 10: Comparison of Φ and ΦÃ .......................................................................................... 49
Figure 11: Internal architecture for gain matrices and ΦÃ for HW_v2 ........................................ 49
Figure 12: Internal architecture for ΦÃ module for HW_v1 ........................................................ 50
Figure 13: Internal architecture for the HW_v1 E module .......................................................... 52
Figure 14: Internal architecture for F_sub module for HW_v1 .................................................. 53
Figure 15: E, F, M, and P modules for HW_v2 .......................................................................... 55
Figure 16: Pipelined MACx module for HW_v2 ........................................................................ 56
Figure 17: Overview of Stage 2 ............................................................................................... 60
Figure 18: Internal architecture for F and Δu° module for HW_v1 .......................................... 63
Figure 19: Internal architecture for γ module for HW_v1 ........................................................ 63
Figure 20: a) VS, b) V-V, and c) Comparison of constraints for HW_v1 .................................. 64
Figure 21: Functional architecture for stage 2 for HW_v2 ....................................................... 65
Figure 22: Internal architecture for SVM_v1 and Wp3 modules for HW_v1 ............................. 74
Figure 23: Stage 3 Converge_v1 and New Δu_v1 modules for HW_v1 ..................................... 75
Chapter 1

INTRODUCTION

1.1 Overview

The adoption of alternative fuel vehicles is considered as one of the major steps towards addressing the issues related to oil dependence, air pollution, and most importantly climate change. Among many options, electricity and hydrogen fuel cells are the top contenders for the alternative fuel for vehicles. Despite numerous public and private initiatives around the world to enhance the usage of Electric Vehicles (EVs), we continue to face many challenges to promote the wider acceptance of EVs by the general public. Some of these major challenges include charging time of the battery and the maximum driving distance of the vehicle [1]. In recent years, major EV manufacturers such as Tesla have been making numerous strides in the electric vehicle industry; however, we still have to overcome the distance traveled, high cost, and charging time constraints to gain the market acceptance.

Power and energy management of EV and HEV is critical to addressing the shortcomings of clean energy vehicles. Power management strategies have to take into account multiple vehicle subsystems and profiles such as electric motor systems, internal combustion engines (for HEV), drive train systems, vehicle dynamics, battery packs, DC-to-DC converters (EVs) and driving profiles [2]. The goals of power and energy management systems include high performance, reduced fuel/energy consumption, lower pollution emissions for HEVs, and battery management. EVs and HEVs come in several configurations, most combine a dynamic energy source with an energy storage system such as batteries or supercapacitors that is also secondary energy source when needed. Some common combinations for vehicles are internal combustion engine and electric motor with battery pack as in [2] or photovoltaic (PV) cell with battery pack as in [3],[4].
It is interesting that HEVs that contain combustion engines and have a combination of Alternating and Direct current (AC and DC) focus on power management while HEVs and EVs that contain natural energy sources and are strictly DC focus on energy management. There are a number of techniques and approaches to power and energy management, rule based policies have been based on heuristics and human expertise and are effective for real-time supervisory control, model based techniques such as dynamic programming and the equivalent consumptions minimization strategy (ECMS) approach that solve a global or local optimization problem and include prediction and finite horizon techniques are popular, and modeless techniques such as support vector machines and reinforcement learning approaches [2] are proposed. Each of these methods manage multiple subsystems which increases complexity. It makes sense for each of these approaches to include lower level control systems that can focus on the constraints inherent to that subsystem and feed that information up to the top level power or energy management system to reduce the top level complexity. The subsystem of focus for this dissertation is the battery management system.

Of the energy storage systems (ESS) utilized by electric vehicles (EVs) such as battery packs, fuel cells, capacitors, super capacitors, and combinations of the above, Lithium-ion (Li-ion) battery packs are one of the most widely employed ESS in EVs mainly because of their light-weight, long life, and high energy density traits [5]. ESS battery packs are typically created from individual Li-ion cells arranged as series and/or parallel modules. The long-term performance (durability) of the Li-ion battery pack is significantly affected by the choice of the cell charging strategy. For instance, exceeding the current and voltage constraints of the Li-ion battery cell can cause irreversible damage and capacity loss that would degrade the long-term performance and curtail the effective life of the battery pack [6]. Conversely, operating within the current and voltage constraints would enhance the durability and prolong the useful life of the battery pack. This requires monitoring and controlling the battery packs at the cell-level. However, most of the existing research on the Battery Management System (BMS) focuses on system-level or pack-level control and monitor, as in [5], instead of cell-level. Thus, it is crucial to investigate and provide
efficient techniques and design methodologies: to monitor and control the battery packs at cell-levels, and to optimize the parameters of the individual cells, in order to enhance the durability and useful life of the battery packs.

Model Predictive Control (MPC) has been investigated as a viable technique for cell-level monitoring and controlling of the battery packs [6]. MPC algorithm is a popular control technique that enables incorporating constraints and generating predictions, while allowing the systems to operate at the thresholds of those constraints. For some time, MPC algorithm has been utilized in the industrial processes, typically in resource-rich environments; however, in recent years, this algorithm is gaining interest in the resource-constrained environments, including cyber-physical systems and hybrid automotive fuel cells[6]. The effectiveness of the MPC algorithm for cell-level monitor/control depends on the accuracy of the mathematical model of the battery cell. These mathematical models include Equivalent Circuit Models (ECMs) and Physics-Based Models (PBMs). The main advantage of the ECM model is the simplicity and solid performance in control algorithms. The simplicity of the model makes ECM well suited for implementation in resource constrained environments such as embedded systems. In [6], the authors prove the efficacy of controlling and providing a fast-charge mechanism for Li-ion battery cells by integrating the MPC algorithm with an ECM model. This fast-charge MPC mechanism incorporates various constraints such as maximum current, current delta, cell voltages, and cell state of charge, which keep the Li-ion battery cell within its optimal operating parameters while reducing the charging time. The main disadvantage of ECM is the inability to incorporate the internal electrochemical reactions in the battery that lead to battery degradation and possibly improved performance. Additionally, the ECM are developed using empirical methods, which limits effective use of the model to the operating conditions used to gather the empirical data[7]. Conversely, the PBMs are based on the internal dynamics of the battery cell. PBMs have the advantage of accounting for the chemical reactions that lead to degradation in performance and battery life and, because they are mathematically based, the models are adaptive and not limited to a specific operating profile [8][9].
The main disadvantage of PBMs are the computational complexity involved. PBMs are highly non-linear and not as easily adaptable as ECMs to a discrete state-space representation that supports a resource constrained implementation required for portable devices, including those used in vehicles. For the last decade, research has focused on creating physics-based battery models that can be represented in state-space form for use in model-based control algorithms. The stated goal of much of the research is to produce algorithms and models suitable for use in embedded systems. To date, most of the research and experiments on creating reduced-order PBMs has been concentrated on creating the models and proving their functionality. Most of this is done using Matlab or other desktop programs to run the experiments, simulations, and verifications. Very little has been done to demonstrate viability of embedded architectures running on microcontrollers or FPGAs. The only research found that goes as far as implementing a physics-based model on an embedded system is that of T.K. Lee, who demonstrated the feasibility of using a reduced order PBM model, using Matlab created C-code, in an existing EPROM BMS system. The PBM state-space implementation used held all parameters constant, significantly reducing area and resource utilization including processing time. The BMS is a proprietary artifact of Ford Motor Company which means there is no information on the control approach or the observer approach, nor is there any discussion on the effect of calculating the parameters based on input changes such as temperature and SOC, making it difficult to use for comparisons.

Combining the PBM insight into the internal mechanisms of the battery, with the MPC ability to proactively handle constraints should produce a BMS that allows maximum use of the battery capacity while protecting the battery against aging and degradation and preserving system safety. An observer or state-estimator is required to utilize the advantages provided by the PBM. One drawback to battery cells is the inability to measure the internal reactions of the battery without invasive techniques. This requires the addition of an observer that can estimate the non-measurable components. The value most often estimated for batteries regardless of model is the state of charge (SOC), using a PBM would also allow us to estimate the state of the internal chemical reactions
such as overpotential or flux and then be able to use these as constraints in the control action. The Kalman filter (KF) is one of the most widely used state estimator/observers for these models [9]. The KF typically provides robust and effective state estimation for linear systems. The PBM model is highly non-linear, therefore a non-linear adaptation of KF is required. The Extended Kalman Filter (EKF) is one of many non-linear adaptations of KF [10], selected due to its simplicity [11] and adaptability to MPC. Thus far, the computation complexity of the PBM has prevented this model as a viable option for portable BMS. The author in [8] envision that with powerful microcontrollers (consisting of substantial on-chip memory), it is possible to support and utilize the physics-based EKF and physics-based MPC, in order to obtain the maximum performance and lifetime from the batteries. Investigation on related work revealed that most of the existing works [12][13][14][15] on FPGA-based EKF for batteries were based on the ECM approach for estimating SOC and state of health (SOH) of the batteries. Even with the adaptations proposed in [7][11], the implementation of a PBM based MPC with an EKF observer requires a powerful embedded systems platform.

Even as research into clean energy has increased, the advent of the smart phone has driven an increase in the capacity and capability of embedded processors and embedded systems. Smart technology research, innovation and commercial competition have produced increasingly capable microprocessors. Field programmable gate arrays (FPGAs) have also been increasing in capability and utility. FPGAs were once limited to use by hardware engineers specializing in one of the main hardware description languages (HDL), VHDL, System Verilog, and Verilog. FPGA companies are now offering high level synthesis (HLS) development tools that translate software languages into register transfer level (RTL) code or HDLs that can be used to program FPGAs to make FPGAs accessible to a wider community. Xilinx provides HLS which translates C, C++, or System C into HDL code which can be used to program an FPGA. Altera, now owned by Intel, also offers an HLS that is advertised to take a C++ input and generate “production quality” HDL for implementation on Intel FPGAs [16]. The Khronos Group, an open industry consortium, has
created Open CL, which is a framework that allows programmers to use combinations of processing units such as central and graphics processing units (CPUs, GPUs), FPGAs, Digital Signal Processors (DSPs) and more[17][18]. Additionally, FPGAs are becoming less expensive; low end FPGA development kits can be found for under $30, such as the Lattice Semiconductor Corporation’s LCMXO3L-6900CS-EVN Development kit for the MACHXO3L FPGA offered on digikey.com. With the ability to provide partial and dynamic reconfiguration after the chips are produced, FPGAs are moving from being mainly a prototyping platform to reduce time to market for ASICs and other chips, to being utilized in research and production system designs. As an example, a main market for FPGAs is the advanced driver assistance systems (ADAS) in the automotive industry. Xilinx’s ADAS sales have grown at a rate of 60% per year since 2013, having shipped more than 40 million chips for the ADAS market [19]. These advances create an opportunity to utilize low cost FPGAs and powerful microcontrollers in battery management embedded systems.

This research explores the possibilities and advantages of implementing ECM and PBM MPC on embedded platforms. Investigating the existing research on MPC algorithms, as well as the existing research on embedded systems designs for MPC algorithms in the literature we found that most of the research on discrete linearized state-space MPC for ECM focused on reducing either the complexity of the Quadratic Programming (QP) or increasing the speed of the computation of the QP, or both. The existing research on on-line MPC methods include fast gradient[20][21], active set [22][23][24], interior point [25][26][27][28][29][30], Newton’s method[23][31][32], Hildreth QP [33] and others [34]. In[35], a faster on-line MPC was achieved by combing several techniques such as Explicit MPC, Primal barrier interior point method, warm start and Newton’s method. In [23] and[32], the Logarithmic Number System (LNS) based MPC was designed on an Field Programmable Gate Array (FPGA) to produce integer-like simplicity. The existing research on embedded systems designs for MPC algorithm focused on FPGAs[22][25][26][31][36][37], System-on-Chip (SOC)[23][30], Programmable Logic
Controllers (PLC)[38], and embedded microprocessors[39]. Although there were interesting MPC algorithms/designs among the existing research works, none of the previously existing research incorporated the feed-through term required by the battery cell model introduced with fast-charge MPC algorithm in[6]. The impact of the feed-through term is discussed in detail in Chapter 3. Additionally, while there have been recent advances in developing physics based models for battery cells [7][9][40][41][42], very little research, has been done for embedded architectures of physics-based model predictive control to this point. The most promising works are those done by[43] and[6] for ECM MPC and by and [44] for PBM MPC. In [43] and [11] the authors create Matlab designs to prove the efficacy of their models and MPC algorithm approaches. The design choices made in [43] and [11] to reduce complexity make both designs excellent candidates for a successful embedded systems architecture. The ECM MPC in [43] is the starting point for the embedded architectures described in chapter 3 while [11] is the starting point for the observer embedded architectures in chapter 4. Both [11] and [44] will be the basis for the full MPC hardware and software architectures that will become chapter 5. Each architecture will explore further methods and approaches that support a robust embedded BMS that reduces complexity while preserving the advantages provided by using model predictive control in battery management systems.

Electric and Hybrid Electric Vehicles bring new challenges to the automotive industry in terms of complexity in terms of battery management, safety functions, increased driver comfort aids such as assisted driving (ADAS), and entertainment systems. With the push to develop and incorporate more sophisticated modeling and control for improved battery management systems and the maturation of state-space based PBMs, research opportunities are wide open in the embedded systems field. Multi-core microprocessor systems and reconfigurable hardware such as FPGAs could conceivably cope with the computational demands, not just of battery management but overall safety and comfort functions of the vehicle[45].
1.2 Research Objectives

The main objective of our research work is to provide microcontroller-level software support and chip-level hardware support for battery cell management applications for Electric and Hybrid Electric Vehicles. In order to achieve our main research objective, we divide our research into three related stages. The objectives for each progressive stage are

- In the first stage, investigate the feasibility and potential performance gain of using hardware for battery cell management using an ECM based MPC algorithm for Fast Charging a battery in a vehicle at rest.
- In the second stage, investigate the feasibility of using embedded systems to support physics-based model observer for battery cell management. Both hardware and software architectures are explored.
- In the third stage, explore architectures for embedded hardware and software that support physics-based MPC control or monitoring of battery cells to improve performance or increase life expectancy.

1.3 Contributions

For the first stage, we introduce a unique, novel and efficient embedded hardware and software architectures for the fast-charge MPC algorithm. The architectures are generic and adaptable to similar MPC algorithms. The proposed architectures have the ability to control the fast-charge of multiple batteries within a single sample time reducing the hardware required for battery management at the cell level. The architecture is unique in that it can process a non-zero feed through term as well as the traditional zero-value feed through term. The experiments to evaluate the feasibility and efficiency of our design were performed in real time with the designs running on the FPGA. This work has led to the publication [46].

For the second stage we have proven that an Extended Kalman filter observer using a reduced order physics-based model can be successfully implemented on both hardware and
software embedded systems. The software runs on a 128Kb microprocessor and the hardware on an FPGA. We provide algorithms and approaches that further reduce the complexity of the PBM model while retaining the accuracy and insight of a physics-based approach. Parts of this work has led to publication [47].

The third stage will extend the work done by[11][44], and [47] continuing to explore methodologies and hardware-software co-design approaches that enable an embedded systems physics-based model predictive controller using an EKF observer. The goal of the third stage is to produce viable software and hardware embedded architectures and evaluate the feasibility and efficiency of the designs using the Xilinx Virtex-7 FPGA development platform.

1.4 Organization

This dissertation is organized as follows; Chapter 2 contains the background on the various fields that comprise an embedded architecture for a battery management system. Chapter 3 describes the software and hardware embedded architectures for the at-rest, fast-charge MPC algorithm utilizing and ECM model. Chapter 4 describes the software and hardware architectures for an EKF observer for an in-motion charging/discharging MPC algorithm based on the PBM model. Proposed Future Research is covered in Chapter 5.
Chapter 2

BACKGROUND

2.1 Overview

This chapter provides background information for the various areas of research that support an embedded systems architecture for a battery management system. The purpose of a battery management system is discussed with a brief explanation of the two battery models considered in this research, ECM and PBM. An overview of model predictive control theory and the basic algorithm is provided followed by a discussion on embedded systems and FPGAs.

2.2 Battery Management Systems (BMS)

Battery management systems are the interface between the vehicle control system and the battery. The BMS is responsible for optimizing the conflicting goals of performance and longevity while also protecting the battery from damage. BMS normally operate at the pack level, however having cell level management systems that optimize cell performance and feed the necessary information back to the pack-level BMS can provide both better protection to the battery cell and increased performance.

2.2.1 Battery Models

When using model-based prediction or estimation algorithms, selecting a suitable model is imperative, since the prediction performance depends on how well the dynamics of the system are represented by the model. The models for batteries tend to follow two categories: empirical models and first principles models[48]. ECM is an empirical model while the PBM is a first-principles model [6]. Both models depend on a combination the measurable current and voltage and
temperature variables to describe and determine the electrochemical battery behavior; these variables are the only easily measurable variables of the battery. The equivalent circuit model uses electrical entities such as resistors and capacitors to model battery behavior. The model uses empirical data gathered on the performance of the battery and some form of curve-fitting to determine the parameter values. This type of model is excellent for situations that are well understood. Fast-charging of a battery while a vehicle at rest is a well-defined situation suited towards empirical models. Physics-based models use the theory of electrochemical reactions to determine performance. Because they are based on first-order principles, these models are inherently adaptive to changes [7] and would be a better fit for battery management of a vehicle in motion. In real life, driving paths and driving conditions are unpredictable; models that require a priori knowledge for the best performance are hampered by the need to define and cover every possible situation. A naturally adaptive model would be ideal for battery management during driving conditions.

2.2.1.1 Equivalent Circuit Model

For the fast-charge of Li-ion batteries in [43], the authors employed an Equivalent Circuit Model (ECM) shown in Figure 1. The sheer simplicity of the ECM leads to a dynamic model that provides a suitable MPC performance for many applications. [6][43]. This ECM model is used in the fast-charge MPC architecture described in Chapter 3. The design and development of the model is fully detailed in [43] with a summary explanation provided in here.
As illustrated in Figure 1 and following the development in [43], the model is based on a Thevenin circuit representation. The series resistor $R_0$ represents the instantaneous response ohmic resistance that occurs when a load is connected to the circuit. In the ECM model, the $R_0$ represents the feed through term, $D_m$, in the MPC general state space equation (3) [6][43][49]. The $R_1C_1$ ladder models the diffusion process. The voltage source, $OCV(z(t))$, represents the state of charge (SOC) dependent open circuit voltage (OCV). Finally, $v(t)$ is the measured voltage. In this model, the dependence of $OCV$ on SOC is non-linear, thus can be implemented as a look-up-table (LUT) in hardware and software. If $Q$ represents the physical capacity of the battery cell or the charge removed when discharging a cell from 100% to 0%, the SOC can be modeled mathematically using current as shown:

$$\frac{dz}{dt} = \frac{-i(t)}{Q} = \dot{z}(t)$$  \hspace{1cm} (2.1)$$

and it follows that $z(t)$ is

$$z(t) = z(t_0) - \frac{1}{Q} \int_{t_0}^{t} i(\tau) d\tau,$$  \hspace{1cm} (2.2)$$

where the integral equation tracks the number of coulombs used divided by the total number of coulombs available ($Q$).

The voltage $v(t)$ in Figure 1 can be represented by the equation:

$$v(t) = OCV(z(t)) - v_{c1}(t) - i(t)R_0$$  \hspace{1cm} (2.3)$$

and

$$\frac{dv_{c1}}{dt} = -\frac{1}{R_1C_1}v_{c1}(t) + \frac{1}{C_1}i(t) = \dot{v}_{c1}(t)$$  \hspace{1cm} (2.4)$$
The ECM models depend on the ability to tie the measurable voltage and/or current to the SOC. There are a multitude of SOC estimation approaches that depend on either voltage or current or a combination of the two. A combination of both is used in this research. Preparing to create the state-space model of Figure 1, we declare $z(t)$ and $v_{c1}(t)$ to be the state variables and the cell voltage to be the output. In matrix form the first two equations form the state equation and the cell voltage is the output equation as follows.

$$\begin{bmatrix} \dot{z}(t) \\ \dot{v}_{C1}(t) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{1}{R_1 C_1} \end{bmatrix} \begin{bmatrix} z(t) \\ v_{C1}(t) \end{bmatrix} + \begin{bmatrix} -\frac{\eta(t)}{Q} \\ \frac{i(t)}{C_1} \end{bmatrix}$$

(2.5)

$$v(t) = \begin{bmatrix} 0 & -1 \end{bmatrix} \begin{bmatrix} z(t) \\ v_{C1}(t) \end{bmatrix} - [R_0]i(t) + OCV(z(t))$$

(2.6)

Cells are not perfect therefore $\eta(t)$ is used to represent the non-ideal cell efficiency to provide more realistic results. Converting to discrete time and assuming current is constant over the sampling period $\Delta T$, the state equation (2.5) becomes

$$\begin{bmatrix} z_{k+1} \\ v_{C1,k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ e^{-\Delta T/R_1 C_1} \end{bmatrix} \begin{bmatrix} z_k \\ v_{C1,k} \end{bmatrix} + \begin{bmatrix} -\eta_k \Delta T/Q \\ R_1 \left(1 - e^{-\Delta T/R_1 C_1}\right) \end{bmatrix} i_k$$

(2.7)

and the output equation (2.6) becomes

$$v_k = \begin{bmatrix} 0 & -1 \end{bmatrix} \begin{bmatrix} z_k \\ v_{C1,k} \end{bmatrix} - [R_0]i_k + OCV(z_k)$$

(2.8)

The values for the parameters (resistances and capacitances) are based on lab-measured data under controlled conditions. While not explicitly stated in the above equations, each of the elements is also dependent on temperature. Because the ECM is an empirical model, it is most effective when the operating conditions closely match the measurement conditions. The stationary fast-charge profile is a good candidate for accurate representation by the ECM model.

The ECM model has a single control input (i.e., the current), and two outputs (i.e., the terminal voltage $v(t)$ and the SOC $z(t)$). The main goal is to bring the battery cell to full SOC with the least amount of time. As a result, the $z(t)$ becomes the output to be controlled, which makes this MPC a Single Input Single Output (SISO) system. The state-space representation of the ECM is
further described in Chapter 3. The creation of the embedded architectures for the ECM MPC algorithm is based on the MPC algorithms presented in [6][11][43][49][50], with many modifications to cater to the embedded platforms. The feed-through term and dual-mode adjustments are based on the ones in [6][43][49]. The basic MPC algorithm is presented in 2.2.2 while the modifications needed to produce the ECM-based fast-charge MPC are presented in Chapter 3.

### 2.2.1.2 The physics-based model

The physics-based model is a first principles, first order model based on the electrochemical reactions, and often incorporate thermodynamics and transport phenomena. None of these are externally measurable and so an observer or estimator is required to determine the state of these reactions as an input into the MPC algorithm. Typically, physics-based models are comprised of partial differential equations (PDEs) which are not suitable for an embedded systems approach. In this case a method to transform the model PDEs into state-space like representations would be necessary. [11] utilizes the Discrete Realization Algorithm to transform the first order physics model developed in [9] into a discrete state-space representation with non-linear corrections to accommodate the non-linear nature of the model. The following explanation summarizes the model from [9][11].

There are 5 basic reactions with corresponding battery cell parameters that comprise the reduced order PBM used in this research

1. Charge conservations in the solid where $\varphi_s$ represents the electric potential in the solid
2. Mass conservation in the solid where $c_{s,e}$ is the lithium concentration in the solid at the surface
3. Mass conservation of the electrolyte where $c_e$ is the lithium concentration in the electrolyte
4. Charge conservation in the electrolyte where $\varphi_e$ is the electric potential in the electrolyte
5. Lithium flux where $j$ represents the flux and $\eta$ is the reaction overpotential tied to the open circuit voltage, represented by $U_{ocp}$.

Each of these reactions is represented by partial differential equations (PDE) that provide a high-fidelity cell model at the cost of high-computational complexity. Such complexity currently prevents use of the cell model in real-time or time-sensitive applications. Transforming the PDEs into a finite order state-space system that can be utilized in EKF and MPC applications is not trivial and more detailed explanations can be found in [11] and [41]. Transforming the PDEs is conducted using the two processes below

1. Create transfer functions from the PDEs, based on [42]

2. Use the discrete realization algorithm [41] to convert the continuous time, infinite poles, transfer functions to an optimal discrete time state-space model.

$$x(t + \Delta t) = Ax(t) + Bi(t) \quad (2.9)$$

$$y(t) = Cx(t) + Di(t) \quad (2.10)$$

Where $\Delta t$ is the sampling period and $i(t)$ is the applied current. In [41] the authors split the battery cell into three regions: negative electrode, separator and positive electrode as shown in Figure 2 from [41] to create a more optimized state-space model.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Negative electrode (n)} & \text{Separator (m)} & \text{Positive electrode (p)} \\
\hline
\phi_\varphi(x,t) & \phi_\varphi(x,t) & \phi_\varphi(x,t) \\
\rho_{\varphi,e}(x,t) & \rho_{\varphi,e}(x,t) & \rho_{\varphi,e}(x,t) \\
j_\varphi(z,t) & j_\varphi(z,t) & j_\varphi(z,t) \\
\eta(z,t) & \eta(z,t) & \eta(z,t) \\
\hline
\end{array}
\]

$z = 0$  $z = 1$  $z = 1$  $z = 0$

$x = 0$  $x = L_n$  $x = L_n + L_m$  $x = L$

Figure 2: Cell schematic showing variables and their position with respect to $x$ or $z$ [41]

While there is a single input to the state-space model, there can be multiple outputs. The outputs are determined by the variables that we want to model. Each variable has at least four spatial locations as defined by Figure 2. Therefore, if we solved for all 6 variables at all four locations,
we would produce a total of 28 outputs for $y(t)$. It is important to note that the reduced order model produced by the DRA above is completely linear. In order to recover the non-linear cell behavior and be able to estimate the non-linear output voltage, we conduct a series of non-linear corrections developed in [41] refined in [11] and explained in Chapter 4. As part of the DRA process the constant biases were removed where possible from the variables of interest. The output vector, $y(t)$ is thus made up of debiased variables. The debiased variables are indicated with a tilde, such as $\tilde{c}_e$ representing the debiased concentration in the electrolyte. The only variable from those listed in steps 1-5 above that does not require a correction is the reaction flux, $j$. The non-linear corrections, state and output equations are more fully detailed in Chapter 4.

Through experiment and analysis to determine the best number of states for both accuracy and simplicity, the authors of [41] found that there was little improvement in the accuracy for PBM systems that used more than five states. Knowing that, our model going forward utilizes the 5 states as determined by [41] and refined in[11]. Because of the linearization done while creating the transfer functions of the model, the prediction errors will increase as the cell state moves away from the initial set point. Similar to any linearization technique, the model is most accurate around the set point and the accuracy will degrade the further the system moves from the set point. To account for this, numerous models were generated around different temperature and SOC set-points. Though fine for a desktop implementation using Matlab, this many models would create too large of a LUT for an embedded system. In [11] the author introduces the concept of a blended model, where the model developed for SOC = 60% at $T = 0^\circ C$ and one for SOC = 60% and $T = 25^\circ C$ are blended together to create a single set of parameters that can be used in the LUTs. This drastically reduces the amount of data that required storage in an embedded system. The resulting blended database required about 50Kb of data. This is trivial on a desktop system, however on a 128Kb microprocessor, this is more than a third of the available memory. This does not consider the program memory or other data memory required to implement the design on an embedded microprocessor.
Having given an overview of the two models currently under investigation by this research, the next section provides a brief overview of the model predictive control process.

### 2.2.2 Model Predictive Control

Model Predictive Control (MPC) utilizes a model of a system (under control) to predict the system’s response to a control signal. Using the predicted response, the control signals are adjusted until the target response is achieved, and then the control signals are applied. For instance, in autonomous vehicles, this model can be used to predict the path of the vehicle. If the predicted path does not match the reference or target path, adjustments are made to the control signals, until the two paths are within an acceptable range. Our version of model predictive control uses a moving window of control and the receding horizon principle. Think of driving a car and how your horizon changes as the car moves. You are aiming the car towards the horizon that is in the direction you wish to travel (this is the prediction horizon $N_p$), yet you are basing your immediate control decision on objects and available paths much closer to your current position (this is the control horizon $N_c$) because the horizon is always moving or changing the theory of receding horizon only applies the most current of the calculated control signals and throws out the rest. The next interval calculates a new set of control signals and the cycle is repeated until you reach the objective.

The MPC algorithms can be generic or customized to a specific application or a specific task, based on the requirements of a given application/task. The customized MPC typically reduces the execution overhead required for certain decision-making logic that would otherwise be essential for the generalized MPC. Furthermore, embedded architectures are usually designed for a specific application or a specific computation. The above facts demonstrate that the customized MPC algorithms specific to a given model and given constraints are appropriate for embedded hardware/software architectures. The first step in MPC is to create the state-space equations, we do this here for a single input and single output system (SISO).
Following the development of MPC from Wang [50], the standard MPC state-space model of a plant is as follows where \( x \) is an \( n \)-dimensional vector describing the state of the system and \( y \) is an \( p \)-dimensional vector of measured outputs and \( u \) is a \( q \)-dimensional input vector:

\[
\begin{align*}
x_{m,k+1} &= A_m x_{m,k} + B_m u_k \\
y_k &= C_m x_{m,k}.
\end{align*}
\]

(2.11)  
(2.12)

In order to eliminate steady-state error, a differencing operation is performed on both sides of equations (2.11) and (2.12) which is a method to implicitly introduce integral action into the MPC:

\[
\Delta x_{m,k+1} = A_m \Delta x_{m,k} + B_m \Delta u_k
\]

where \( \Delta x_{m,k} = x_{m,k} - x_{m,k-1} \) and \( \Delta u_k = u_k - u_{k-1} \). This process is referred to as augmentation. Treating the output similarly:

\[
y_{k+1} - y_k = C_m \Delta x_{m,k+1} = C_m A_m \Delta x_{m,k} + C_m B_m \Delta u_k.
\]

(2.14)

The new augmented state vectors become:

\[
x_k = \begin{bmatrix} \Delta x_{m,k} \\ y_k \end{bmatrix} \quad \text{and} \quad x_{k+1} = \begin{bmatrix} \Delta x_{m,k+1} \\ y_{k+1} \end{bmatrix}.
\]

(2.15)

Combining (2.13) and (2.14) results in the following:

\[
x_{k+1} = \begin{bmatrix} \Delta x_{m,k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} A_m & 0^T_m \\ C_m A_m & I \end{bmatrix} \begin{bmatrix} \Delta x_{m,k} \\ y_k \end{bmatrix} + \begin{bmatrix} B_m \\ C_m B_m \end{bmatrix} \Delta u_k = A x_k + B \Delta u_k.
\]

(2.16)

\[
y_k = \begin{bmatrix} 0_m & I \end{bmatrix} \begin{bmatrix} \Delta x_{m,k} \\ y_k \end{bmatrix} = C x_k.
\]

(2.17)

The next step is to calculate the predicted plant output using the input control signal \( \Delta u_k \) from (2.16) within an optimization window \( N_p \) also called the prediction horizon. The \( \Delta u_k \) signal is known as the adjustable signal; we will later use the optimization to determine the best \( \Delta u_k \) for the desired output. The state variable, \( x_k \) describes the current state of the plant. In our case \( x_k \) cannot be measured directly and therefore requires estimation. The estimation approach we use for ECM is discussed in Chapter 3 and for PBM in Chapter 4. The prediction horizon determines the number of future state variables denoted as:

\[
x_{k+1|k}, x_{k+2|k}, \ldots, x_{k+N_p-1|k}, x_{k+N_p}
\]
where the $k+1|k$ means the first prediction at $k+1$ given the input at the current interval, $k$. Similarly, the number of control or input variables is determined by the control horizon $N_c$. The control horizon is always chosen to be less than or equal to the prediction horizon. The vector of control variables is denoted below:

$$\Delta U = [\Delta u_k, \Delta u_{k+1}, \cdots, \Delta u_{k+N_c-1}]^T.$$ 

Using equation (2.16) the state variables are calculated sequentially:

$$x_{k+1|k} = Ax_k + B\Delta u_k$$
$$x_{k+2|k} = Ax_{k+1|k} + B\Delta u_{k+1|k} = A^2 x_k + AB\Delta u_k + B\Delta u_{k+1}$$
$$\vdots$$
$$x_{k+N_p|k} = A^{N_p} x_k + A^{N_p-1} B\Delta u_k + A^{N_p-2} B\Delta u_{k+1} + \cdots + A^{N_p-N_c} B\Delta u_{k+N_c-1}.$$ 

Using (2.17) the predicted output variables are calculated by multiplying the state variables by $C$ as shown below:

$$y_{k+1|k} = CAx_k + CB\Delta u_k$$
$$y_{k+2|k} = CA^2 x_k + CAB\Delta u_k + CB\Delta u_{k+1}$$
$$\vdots$$
$$y_{k+N_p|k} = CA^{N_p} x_k + CA^{N_p-1} B\Delta u_k + CA^{N_p-2} B\Delta u_{k+1} + \cdots + CA^{N_p-N_c} B\Delta u_{k+N_c-1}.$$ 

Note that the predictions are formulated in terms of the current state, $x_k$, and the input or future control signal $\Delta u_{k+j}$, where $j = 0, 1, 2..N_c - 1$. A vector of the output variables is organized as follows:

$$Y = [y_{k+1|k}, y_{k+2|k}, \cdots, y_{k+N_p|k}]^T.$$ 

Understanding that the dimension of $\Delta U$ is $N_c$ and the dimension of $Y$ is $N_p$, the output equations can then be written in compact matrix form:

$$Y = \Phi A x_k + G \Delta U$$

(2.18)
where \( \Phi = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{N_p-1} \end{bmatrix} \) and \( G = \begin{bmatrix} CB & 0 & 0 & \cdots & 0 \\ CAB & CB & 0 & \cdots & 0 \\ CA^2B & CAB & CB & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ CA^{N_p-1}B & CA^{N_p-2}B & CA^{N_p-3}B & \cdots & CA^{N_p-N_c-1}B \end{bmatrix} \). (2.19)

you might also find (2.18) in the form \( Y = Fx_k + \Phi \Delta U \) where the first term of \( F \) is \( CA \) and so on. However, the MPC used in Chapter 3 follows the terminology in (2.18). Having built the prediction matrix, the next step is optimization. Given a goal or target, also called a set-point signal, the objective is to find the control signal \( \Delta U \) that will bring \( Y \) as close as possible to the target \( R_s \).

In order to make the optimization calculations work, the target vector has a length of \( N_p \) and is written generically as:

\[
R_s^T = [1 \ 1 \ \cdots \ 1] \eta_k
\]

and the optimization or cost function is written as:

\[
J_k = (R_s - Y_k)^T (R_s - Y_k) + \Delta U_k^T \bar{R} \Delta U_k.
\] (2.20)

The first term in \( J_k \) has the objective of minimizing the difference or error between the target and the predicted output while the second term is used to make the size of the input vector as small as possible. \( \bar{R} \) is a diagonal matrix with a tuning parameter \( r_w \) along the diagonal and could be 0 if the size of the input vector is not a consideration. Typically, the optimization is accomplished by taking the derivative of the cost function with respect to \( \Delta U_k \) and setting it to 0.

\[
\frac{\partial J_k}{\partial \Delta U_k} = 0
\] (2.21)

Replacing \( Y \) with (2.18) results in

\[
\frac{\partial J_k}{\partial \Delta U_k} = -2G^T (R_s - \Phi Ax_k) + 2(G^T G + \bar{R}) \Delta U_k = 0
\] (2.22)

Solving for \( \Delta U_k \):

\[
\Delta U_k = (G^T G + \bar{R})^{-1} G^T (R_s - \Phi Ax_k)
\] (2.23)

creates a generic solution that computes the vector containing future optimal control inputs at each time step, \( k \) up to the length of the control horizon \( k + N_c - 1 \). In optimization literature the Hessian matrix contains the second derivative information and is represented by the \((G^T G + \bar{R})^{-1}\)
term in (2.23). The receding horizon control principle means that only the first control signal, $\Delta u_k$ is applied to the plant. This solution is further adapted for each of the MPC approaches used in this research. Chapter 3 introduces the concept of near future and far future to further simplify the MPC calculations.

To calculate the optimal control signal, we must incorporate the constraints or operating limitations of the plant under control. MPC does this implicitly using the following relationship:

$$M\Delta U_k \leq \gamma$$ (2.24)

where $M$ is a matrix reflecting the constraints. The constraints are unique to each model and MPC algorithm and thus development of the constraints is done in Chapter 3 for the ECM MPC.

In developing MPC we assume that the current state information, $x_k$, is available. Usually this means that the state is made up of measurable parameters. For battery cells this is not the case. As mentioned in the model development, the only available measurable values are the current and the voltage. The other quantities are determined based on estimation. In control engineering the process or method of estimating the state is called an observer. Chapter 4 introduces the Extended Kalman Filter Observer that is used with the physics-based model to predict the internal characteristics of the battery and subsequently the SOC. The fast-charge MPC algorithm in Chapter 3 does not use a traditional observer for the SOC, instead we use the empirically derived SOC to OCV relationship to determine the SOC.

### 2.2.3 Optimization

Methods and approaches for solving optimization problems forms its own branch of research. I am focused on techniques that translate well to minimum resource embedded systems. While FPGAs are well suited to parallel processing, single core microcontrollers are limited to one thread of processing and are excellent at iterative approaches. Two optimization techniques that can be used to reduce complexity are the use of active constraints and the primal dual method. The active constraint approach limits the calculation to those constraints that are active in a solution.
interval. The active constraint can change from one interval to another but by eliminating the inactive constraints, calculations have the potential to be reduced. In the case of the ECM approach in Chapter 3 there are 32 possible active constraints, typically only 2 are active for any given solution interval. The primal dual approach simultaneously determines the active constraints and allows the optimization to reduce a 2 variable minmax problem to be solved using a single variable max problem. A brief overview of the approach is as follows. Starting with a generic cost or objective function

$$J = \frac{1}{2} x^T E x + x^T F$$  \hspace{1cm} (2.25)

In order to calculate the optimal control signal, we must incorporate the constraints or operating limitations of the plant under control. MPC does this implicitly using the following relationship:

$$M x \leq \gamma.$$  \hspace{1cm} (2.26)

If we multiply the equality constraint in (2.26) with the Lagrange multiplier, \(\lambda\) and add this to the objective function (2.25) we have a new augmented objective function that enables us to minimize the objective function by excluding inactive constraints from the solution.

$$J = \frac{1}{2} x^T E x + x^T F + \lambda^T (M x - \gamma)$$  \hspace{1cm} (2.27)

Taking the partial derivatives of (2.27) with respect to \(x\) and \(\lambda\) and setting them to zero leaves us with

$$\frac{\partial J}{\partial x} = E x + F + M^T \lambda = 0$$  \hspace{1cm} (2.28)

$$\frac{\partial J}{\partial \lambda} = M x - \gamma = 0$$  \hspace{1cm} (2.29)

Using the partial derivatives to build the Lagrangian matrix allows us to explicitly solve for \(x\) and \(\lambda\).

$$x = -E^{-1}(M^T \lambda + F)$$  \hspace{1cm} (2.30)

$$\lambda = -(M E^{-1} M^T)^{-1}(\gamma + M E^{-1} F)$$  \hspace{1cm} (2.31)
The Primal-Dual method of solving optimization problems uses a dual method to identify the constraints that are not active or triggered by the equality constraint (2.26) and therefore can be eliminated from the solution. The Primal problem for (2.27) is:

$$\max_{\lambda \geq 0} \min_x \left[ \frac{1}{2} x^T E x + x^T F + \lambda^T (M x - \gamma) \right]$$

(2.32)

The dual problem is developed by recognizing that the minimum over x is expressed by (2.30). Substituting (2.30) into (2.32) reduces the optimization problem to

$$\max_{\lambda \geq 0} \left( -\frac{1}{2} \lambda^T P \lambda - \lambda^T K - \frac{1}{2} F^T E^{-1} F \right)$$

(2.33)

Where

$$P = ME^{-1} M^T$$

(2.34)

$$K = \gamma + ME^{-1} F$$

(2.35)

The active constraints identified by the dual method are denoted as $M_{\text{act}}$ and $\lambda_{\text{act}}$. Reorganizing (2.30) and utilizing active constraints allows $x$ to be written as

$$x = -E^{-1} F - E^{-1} M_{\text{act}}^T \lambda_{\text{act}}$$

(2.36)

If we substitute MPC notation into (2.36) and replace $x$ with $\Delta u$, we get the constrained control solution

$$\Delta u = \Delta u^o - E^{-1} M_{\text{act}}^T \lambda_{\text{act}}$$

(2.37)

Where $\Delta u^o$ represents the unconstrained control solution and $E^{-1} M_{\text{act}}^T \lambda_{\text{act}}$ is the correction factor. The correction factor can be found using Hildreth’s Quadratic Programming method, this method is an iterative approach to solving the dual optimization problem as expressed in (2.33).

2.2.4 Hildreth’s Quadratic Programming Technique

Hildreth’s Quadratic Programming (HQP) has two main advantages for embedded systems, the first is that it is an iterative process that does not require matrix conversion and is executed using basic math functions, multiply, divide, add and subtract, therefore it does not require advanced math libraries or functions. Second, and perhaps more importantly, it always provides
an executable solution. The solution might be sub-optimal; however, it can be used safely by or on
the system. Solvers that cannot provide a solution could cause software or hardware to enter an
unrecoverable state or crash. HQP uses the equations developed in 2.3.1 and can be separated into
two main steps. The first step determines the unconstrained solution $\Delta u^o$ and the second step
determines the correction factor if it is needed. In the first step, the unconstrained solution set is
calculated and then compared to the constraints using

$$M \Delta u^o \leq \gamma \quad (2.38)$$

If (2.38) is true for all constraints and elements of $\Delta u^o$, the unconstrained solution is the optimal
control signal for this control step and the optimization step is complete. However, if (2.38) is false
for any constraint, the HQP algorithm invokes the second step to calculate the correction
factor, $E^{-1} M_{act}^T \lambda_{act}$ . The correction factor depends on the vector of Lagrange multipliers, $\lambda$. In
HQP the $\lambda$ vector is created iteratively by varying one element at a time. With a starting vector
($\lambda^m$), a single element ($\lambda_i^m$) of the vector is modified, utilizing $P$ and $K$ ((2.34) and (2.35)
respectively) to minimize the cost function (2.27), which creates $\lambda_i^{m+1}$. If the modification requires
$\lambda_i^m < 0$, the constraint is inactive and $\lambda_i^{m+1} = 0$. Then, the next element ($\lambda_{i+1}^{m+1}$) of the vector is
considered, and this process continues until all the elements of the entire $\lambda^m$ vector are modified.
This modification is computed using equation (2.40):

$$\lambda_i^{m+1} = \max(0, w_i) \quad (2.39)$$

where,

$$w_i = -\frac{1}{p_{ii}} [k_i + \sum_{j=1}^{i-1} p_{ij} \lambda_j^{m+1} + \sum_{j=i+1}^{n} p_{ij} \lambda_j^{m}] \quad (2.40)$$

In this case, $k_i$ and $p_{ij}$ are the scalar $i^{th}$ and $ij^{th}$ elements of $K$ and $P$ respectively. This is an
iterative process, which continues either until the $\lambda$ converges (so that $\lambda^{m+1} \approx \lambda^m$) or until a
maximum number of iterations is reached. This process concludes with a $\lambda^*$ of either 0 or positive
values. The positive values are the active constraints in the system during this time step. The next
step is to utilize $\lambda^*$ made from the elements found by (2.39), to obtain the correction factor:
\[ HQPdu = -E^{-1}M^T\lambda^* \]  

(2.41)

The correction factor is added to the unconstrained solution to create the constrained solution

\[ \Delta u_{k+1} = \Delta u_{k+1}^0 - E^{-1}M^T\lambda^* \]  

(2.42)

The constrained solution is optimal when the algorithm converges, if the algorithm does not converge by the maximum number of iterations, a sub-optimal but still acceptable solution is created.

2.3 Embedded Systems

Embedded systems are a combination of hardware and software dedicated to a specific set of tasks. Alternatively, they are intelligent systems with a dedicated purpose embedded into larger systems. Embedded systems range from microcontrollers to FPGA based systems and include programmable logic systems, microprocessors, and programmable and non-programmable application specific integrated circuit (ASIC) chips. Embedded systems control integrated hardware peripherals such as sensors, timers, and networks. In the last decade, FPGAs have become more prevalent in embedded systems due to their flexibility and capacity[51]. They are well suited to support faster response times, custom logic, and a multitude of peripherals. Today, most FPGAs also provide either a hard or soft microprocessor. A hard microprocessor is made up of dedicated circuitry that cannot be reused by the FPGA, whereas a soft processor uses the same FPGA fabric as user design logic and can be implemented if required by the user. Typically, customized hardware is optimized for a specific application and avoids the high execution overhead of fetching and decoding instructions as in microprocessor-based designs, thus providing higher speed performance, lower power consumption, and area efficiency, than equivalent software running on general-purpose microprocessors.

With the expansion of smart and automated devices, FPGAs and microprocessors are rapidly increasing in capability. FPGAs that contain microprocessors are uniquely suited for hardware and software codesign, as the hardware accelerator(s) and the microprocessor are
implemented on the same fabric, allowing them to communicate via internal busses and signals, reducing the communication induced latency experienced by hardware accelerators that are external to the microprocessor.

Our previous analyses illustrated [83],[84] that FPGA-based systems provide numerous advantages, including flexibility, area efficiency, reduced time-to-market, and relatively low cost, which are important to support compute-intensive applications, such as MPC on embedded devices. Furthermore, our previous work on FPGA-based accelerators, architectures, and techniques for various compute/data-intensive applications, including data mining/analytics [85],[86],[87],[88],[89],[90],[91], control systems [46],[47], security [92],[93], machine learning [94],[95],[96],[100] and bioinformatic [97], demonstrates that FPGA-based systems are the best avenue to support and accelerate complex algorithms such as MPC on embedded devices, considering the constraint associated with these devices and the requirements of the applications running on these devices.

FPGAs are now capable of dynamic and/or partial reconfiguration that allows reuse of the FPGA fabric for different accelerators. This can be automated and occur without intervention by a human, increasing the capacity and utility of the, typically, more expensive FPGA chips. Finally, with the advent of high-level synthesis (HLS) opening the use of FPGAs to non-hardware engineers, FPGAs are becoming more popular and therefore more available to industry. HLS allows designers to use high level languages such as C and C++ to create register-transfer level (RTL) designs that would normally require one of the hardware description languages of Verilog or VHDL. Bypassing the need to learn the HDLs.

The FPGA advantage of supporting both microprocessors and user designed hardware accelerators makes the uniquely suited to support the computational complexity of the model predictive control algorithms for ECM and PBM models. Chapters 3, 4 and 5 explore the utility of FPGA to support embedded implementations ECM and PBM based algorithms.
Chapter 3

FAST CHARGE MPC DESIGN

3.1 Background: Model Predictive Control for the Fast-Charge MPC

In this chapter, our objective is to investigate the feasibility and potential performance gain of using tailored software and hardware architectures. Our investigation [46] focuses on the at-rest charging profile of an electric vehicle (EV). When at rest, the battery charging profile involves only charging with no discharge and the goal is to charge the battery as fast as possible without violating any constraints. Constraints are in place to preserve battery life, performance and safety.

An investigation on the existing MPC algorithms revealed that the MPC design in [6] provides a simple, robust, and efficient algorithm for the fast charging of lithium ion battery cells. Hence, this MPC algorithm [6] could be potentially suitable for creating embedded hardware and software designs. The simplicity of this algorithm is based on two major design decisions that reduce the computational complexity of the algorithm: use of the Dual-Mode MPC technique, and the Hildreth’s Quadratic Programming technique [43].

The Dual-Mode MPC technique addresses the computational issue of the infinite prediction horizons. This technique divides the problem space into the near future and the far future solution segments. This enables the prediction horizons and control horizons to be decreased significantly, while maintaining the performance on par with the infinite prediction horizons [43]. The application of this technique to the fast charge of batteries with a feed-through term is detailed in [43]. As discussed in [43], reducing the prediction horizon dramatically reduces the size of the matrices utilized in MPC, which in turn reduces the computation complexity. Trimboli’s group [6],[43] evaluated various control horizons and prediction horizons for the optimal performance using the near future and the far future approach, and determined that the optimal control and prediction horizons to be $N_c = 1$ and $N_p = 10$. 

27
The Hildreth’s Quadratic Programming (HQP) technique is an iterative process that is deemed suitable for the embedded systems software designs [43],[50]. This technique is part of the active set Dual-Primal Quadratic Programming (QP) solution, which consists of two main features that are beneficial for embedded designs: (1) no matrix inversion is required, hence manages poorly conditioned matrices; (2) the computations are run on scalar instead of matrices, thus reducing the computation complexity [50]. With the HQP, the intention of the MPC is to bring the battery cell to a fully charged position in the least amount of time. In order to reduce the computational effort [6], the pseudo min-time problem is implemented to achieve the same results as the explicit optimal min-time solution. As a result, the HQP technique is deemed appropriate, although it might produce a sub-optimal solution, in the case where the solution fails to converge in the allotted iterations [38]. Conversely, HQP will always produce a solution. A recent study [38] revealed that the HQP technique performed faster than the commercial solvers. The main drawback again identified is the tendency to provide the sub-optimal solutions more often than other solvers. The performance of HQP is dependent on selecting the optimal number of iterations for the application. In this study [38] the clock speed per iteration of the HQP technique was approximately 15 times faster than the most robust state-of-the-art active set solver (qpOASES).

The next sections describe the architectures for the fast-charge MPC and the various adaptations made to the designs presented in [6],[43] to support MPC on embedded systems.

3.2 Dynamic Model

To accomplish the fast-charging of Li-ion batteries, the authors in [6],[43] employed the Equivalent Circuit Model (ECM) shown below chosen for its simplicity and the fact that the fast-charge profile of a car at rest is well understood.
Revisiting Figure 1, shown here, we continue the development of the state-space equations specific to the fast-charge MPC. Here the series resistor $R_0$ is the instantaneous response ohmic resistance that occurs when a load is connected to the circuit, which essentially provides a feedback signal or feedthrough term that is not normally present in MPC. This term is represented by $D_{m,k}u_k$ in the output equation shown in (3.3). The $R_1C_1$ ladder models the diffusion process; the state of charge (SOC) dependent voltage source, i.e., $OCV(z(t))$, represents the open circuit voltage (OCV).

The current $i(t)$, which is the control input signal, is represented in the state space equations as $u_k$. Employing the MPC algorithm against this model, our intention is to find the best control input, $i(t)$, in order to produce the fastest charge while respecting the physical constraints of the cell; therefore $z(t)$ is the signal to be controlled and $v(t)$ is the measurable output.

The general state-space equations for the ECM model are designed and developed in Chapter 2 based on Figure 1 [6]. The physical parameters of the model are $Q_{\text{charge}}$, $R_0$, $R_1$, and $\tau = R_1C_1$. Typically, the parameters or the elements of the ECM model are temperature dependent. The values for these parameters have been gathered by empirical measurements at a controlled temperature for this iteration of the Fast-charge design. In this case, the un-augmented state variables are considered as: the $z(t)$, which is the state of charge (SOC) of the open circuit voltage (OCV), and the $v_{CI}(t)$, which is the voltage across the capacitor. The terminal voltage $v(t)$ is the output and the current $i(t)$ is the input control signal. The discretized state space variables are $Z_k$, $v_{CI,k}$, $v_k$, and $u_k$. The general state-space equation (2.11) is repeated below:

$$x_{m,k+1} = A_m x_{m,k} + B_m u_k$$
With the state equation from (2.7):

\[
\begin{bmatrix}
    z_{k+1} \\
    v_{C_1, k+1}
\end{bmatrix} = \begin{bmatrix}
    1 & 0 \\
    0 & e^{-\Delta t / R_1 C_1}
\end{bmatrix}
\begin{bmatrix}
    z_k \\
    v_{C_1, k}
\end{bmatrix} + \begin{bmatrix}
    -\frac{\eta \Delta t}{Q} \\
    R_1 \left( 1 - e^{-\Delta t / R_1 C_1} \right)
\end{bmatrix} u_k
\]

The general formula for the measured outputs from (2.12) now includes the feed through term identified in the ECM for the Fast-charge model:

\[ y_k = C_m x_{m,k} + D_m u_k \] (3.1).

Adapting (3.1) for the terminal voltage, the general equation becomes:

\[ v_k = C_{m,v} x_{m,k} + D_{m,v} u_k + OCV(z_k) \] (3.2)

Comparing the above to equation (2.8) we see that the \([-R_0]i_k\) term is the feed through term where \(u_k\) represents the current \(i_k\) and the terminal voltage output equation (3.2) becomes:

\[ v_k = \begin{bmatrix}
    0 & -1
\end{bmatrix}
\begin{bmatrix}
    z_k \\
    v_{C_1, k}
\end{bmatrix} + \begin{bmatrix}
    -R_0
\end{bmatrix} u_k + OCV(z_k) \] (3.3)

The equation for the output to be controlled, SOC, is developed similarly for (3.4):

\[ z_k = C_{m,z} x_{m,k} + D_{m,z} u_k \]

\[ z_k = \begin{bmatrix}
    1 & 0
\end{bmatrix}
\begin{bmatrix}
    z_k \\
    v_{C_1, k}
\end{bmatrix} + \begin{bmatrix}
    0
\end{bmatrix} u_k \] (3.4)

Going forward, the sampling time (\(\Delta t\)), and the cell efficiency (\(\eta\)) are considered as 1 second and 0.997, respectively. These values are determined from [6],[43] based on a Li-ion battery manufactured by the LG Chem Ltd [49]. Next, the model is augmented to incorporate integral action and the feed-through term. The integral action is incorporated by determining the difference between the state signals (\(\Delta x_{m,k}\)) and the control signals (\(\Delta u_k\)). The addition of the feedthrough term necessitates an adjustment to the state variable. Ordys and Pike came up with a unique solution that redefines the state variable and allows the use of the normal augmentation equations shown in (2.16) and (2.17). The new state variable is:

\[ \chi_k = [x_k \quad u_k], \text{ where } x_k = [\Delta x_{m,k} \quad y_k] \] (3.5)

The final augmented state-space equations are presented below, based on the design in [6]:
\( \chi_{k+1} = \tilde{A}\chi_k + \tilde{B}\Delta u_{k+1} \)  \hspace{1cm} (3.6) \\
\( v_k = \tilde{c}_v\chi_k + OCV(z_k) \)  \hspace{1cm} (3.7) \\
\( z_k = \tilde{c}_z\chi_k, \)  \hspace{1cm} (3.8) \\

where the A, B and C are redefined as follows:

\[
\begin{bmatrix} A_m \\
0 \\
I \end{bmatrix}, \quad \tilde{B} = \begin{bmatrix} 0 \\
I \end{bmatrix}, \quad \tilde{c}_v = \begin{bmatrix} C_{m,v} & D_{m,v} \end{bmatrix}, \quad \text{and} \quad \tilde{c}_z = \begin{bmatrix} C_{m,z} & D_{m,z} \end{bmatrix}. \quad (3.9)
\]

### 3.2.1 Prediction of State and Output Variables

The feed-through term incorporated by Trimboli’s group [43],[49] was built upon and extended from the work done in [52]. A detailed description of the extended work can be found in [43],[49] and the synopsis of this approach can be found in [6]. For illustration purposes, the summary of this approach is presented below.

After completing the augmented model, the gain matrices are computed. To achieve this, the state equation (3.6), as demonstrated below, is propagated to obtain the future states. Due to the definition of the new state variable, this is remarkably similar to the gain matrices computations presented in chapter 2.

\[
\begin{align*}
\chi_{k+1} &= \tilde{A}\chi_k + \tilde{B}\Delta u_{k+1} \\
\chi_{k+2} &= \tilde{A}\chi_{k+1} + \tilde{B}\Delta u_{k+2} = \tilde{A}(\tilde{A}\chi_k + \tilde{B}\Delta u_{k+1}) + \tilde{B}\Delta u_{k+2} \\
&= \tilde{A}^2\chi_k + \tilde{A}\tilde{B}\Delta u_{k+1} + \tilde{B}\Delta u_{k+2} \\
\chi_{k+3} &= \tilde{A}^3\chi_k + \tilde{A}^2\tilde{B}\Delta u_{k+1} + \tilde{A}\tilde{B}\Delta u_{k+2} + \tilde{B}\Delta u_{k+3} \\
& \vdots \\
\chi_{k+N_p} &= \tilde{A}^{N_p}\chi_k + \tilde{A}^{N_p-1}\tilde{B}\Delta u_{k+1} + \tilde{A}^{N_p-2}\tilde{B}\Delta u_{k+2} + \cdots + \tilde{A}\tilde{B}\Delta u_{k+N_c} + \tilde{B}\Delta u_{k+N_c} \\
\end{align*}
\]  \hspace{1cm} (3.10)

Next, the output equation (3.8) is propagated and substituted with the elements of equation (3.6), in order to obtain the predicted output as equation (3.11).

\[
y_{k+1} = \tilde{c}\chi_{k+1} = \tilde{c}\tilde{A}\chi_k + \tilde{c}\tilde{B}\Delta u_{k+1}
\]
\[ y_{k+2} = \hat{c}x_{k+2} = \hat{c}\hat{a}^2x_k + \hat{c}\hat{a}\hat{b}\Delta u_{k+1} + \hat{c}\hat{b}u_{k+2} \]

\[ y_{k+3} = \hat{c}x_{k+3} = \hat{c}\hat{a}^3x_k + \hat{c}\hat{a}^2B\Delta u_{k+1} + \hat{c}\hat{a}\hat{b}\Delta u_{k+2} + \hat{c}\hat{b}\Delta u_{k+3} \]  
(3.11)

\[ y_{k+N_p} = \hat{c}x_{k+N_p} = \hat{c}\hat{a}^N\hat{p}x_k + \hat{c}\hat{a}^{N-1}\hat{p}\Delta u_{k+1} + \hat{c}\hat{a}^{N-2}\hat{p}\Delta u_{k+2} + \cdots + \hat{c}\hat{a}^{N-N_c}\hat{p}\Delta u_{k+N_c} \]

Rewriting the equation (3.11) in matrix form produces the following equations (3.12) and (3.13).

\[
Y_k = 
\begin{bmatrix}
\hat{c} & 0 & 0 & \cdots & 0 \\
\hat{c}\hat{a} & \hat{c}\hat{a}\hat{b} & \hat{c}\hat{b} & \cdots & 0 \\
\hat{c}\hat{a}^2 & \hat{c}\hat{a}^2\hat{b} & \hat{c}\hat{a}\hat{b} & \hat{c}\hat{b} & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\hat{c}\hat{a}^{N-1} & \hat{c}\hat{a}^{N-1}\hat{b} & \hat{c}\hat{a}^{N-2}\hat{b} & \hat{c}\hat{a}^{N-3}\hat{b} & \cdots & \hat{c}\hat{a}^{N-N_c}\hat{b} \\
\end{bmatrix}
\begin{bmatrix}
x_k \\
\Delta u_{k+1} \\
\Delta u_{k+2} \\
\Delta u_{k+3} \\
\vdots \\
\Delta u_{k+N_c} \\
\end{bmatrix}
\]

\[ Y_k = \Phi \hat{a}x_k + G\Delta U_k \]  
(3.12)

(3.13)

In order to use the far-future control technique, the \( G \) matrix and \( \Delta U_k \) matrix are partitioned into the near-future (nf) and the far-future (ff) elements, where the near future gain \( G_{nf} \) is a \( N_p \times N_c \) matrix and the far future gain \( G_{ff} \) is a \( N_p \times N_p - N_c \) matrix as below:

\[
\Delta U_k = \begin{bmatrix}
\Delta U_{k,nf} \\
\Delta U_{k,ff}
\end{bmatrix}, \quad G = \begin{bmatrix}
G_{nf} & G_{ff}
\end{bmatrix}
\]

(3.14)

In other words, with \( N_c = 1 \), \( G_{nf} \) is the first column of \( G \) and \( G_{ff} \) is comprised of the remaining columns of \( G \). Partitioning \( G \) in equation (3.13) results in

\[ Y_k = \Phi \hat{a}x_k + G_{nf}\Delta U_{k,nf} + G_{ff}\Delta U_{k,ff} \]  
(3.15)

As discussed in [49], \( \Delta U_{k,nf} \) from (3.12), being able to express \( \Delta U_{k,ff} \) in terms of \( \Delta U_{k,nf} \) prevents the need to add another row and column to \( G \), and, as developed in [49], results in the following representation (3.16):

\[ \Delta U_{k,ff} = -(v\Delta U_{k,nf} + u_k) \]  
(3.16)
where $\mathbf{v}_{1 \times N_c} = [1 \ 1 \ 1 \ \cdots \ 1]$. Furthermore, substituting equation (3.16) into (3.15) results in equation (3.17):

$$
\mathbf{Y}_k = \Phi \tilde{A}_k + G_{nf} \Delta \mathbf{U}_{k,nf} - G_{ff} \mathbf{v} \Delta \mathbf{U}_{k,nf} - G_{ff} u_k
$$

(3.17)

The steps are required to process and complete the MPC algorithm. For the Fast-charge embedded architectures, temperature is held constant and so the above gain matrices need only be calculated once at startup.

### 3.2.2 Optimization

With the embedded systems design, the objective is to create a control signal that brings both the output signal $Y_k$ and the reference or set-point signal $R$, as close as possible. In this case, it is assumed that $R$, remains constant inside the prediction window. The cost function that reflects this optimization goal for the Fast-charge profile is written in a matrix form as below [43]:

$$
J_k = (\mathbf{Y}_k - \mathbf{R}_s)^T (\mathbf{Y}_k - \mathbf{R}_s) + P_1 \Delta \mathbf{U}_{k,nf}^T \Delta \mathbf{U}_{k,nf}.
$$

(3.18)

In the above equation (3.18), $R_s$ is a vector of set-point or target information, and $P_1$ is a penalty factor based on the given constants $\omega_r$ and $\lambda_p$. Substituting equation (3.18) with the elements of equation (3.17), utilizing properties of the symmetric matrices, and grouping the terms, results in the following cost function:

$$
J_k = \Delta \mathbf{U}_{k,nf}^T (G_{nf}^T G_{nf} + P_1 I - G_{nf}^T G_{ff} \mathbf{v} - \mathbf{v}^T G_{ff} G_{nf} + \mathbf{v}^T G_{ff}^T G_{ff} \mathbf{v}) \Delta \mathbf{U}_{k,nf} - 2 \Delta \mathbf{U}_{k,nf}^T (G_{nf}^T \mathbf{R}_s + \mathbf{v}^T G_{ff}^T \mathbf{R}_s - G_{nf}^T \Phi \tilde{A}_k - \mathbf{v}^T G_{ff} \Phi \tilde{A}_k - G_{nf}^T G_{ff} u_k - \mathbf{v}^T G_{ff} G_{ff} u_k) + (\Phi \tilde{A}_k - \mathbf{R}_s - G_{ff} u_k) \Phi \tilde{A}_k - \mathbf{R}_s - G_{ff} u_k).
$$

(3.19)

For this architecture, we are using HQP to solve the primal-dual optimization problem as described in Chapter 2. Step 1 of HQP is to determine the unconstrained solution, $\nabla \mathbf{u}^o = E^{-1} \mathbf{F}$. Recalling the cost function used by HQP from chapter 2 is (2.27), shown below for convenience:

$$
J = \frac{1}{2} x^T Ex + x^T F + \lambda^T (Mx - \gamma)
$$
We can relate (3.19) to (2.27) to infer $E$ and $F$ and produce the following equations (3.20) and (3.21):

$$E = 2(G_{nf}^T G_{nf} + P_1 - G_{nf}^T G_{ff} v - v^T G_{ff}^T G_{nf} + v^T G_{ff}^T G_{ff} v)$$  
(3.20)

$$F = -2(G_{nf}^T R_s + v^T G_{ff}^T R_s - G_{nf}^T \Phi \Delta X_k - v^T G_{ff}^T \Phi \Delta X_k - G_{nf}^T G_{ff} u_k - v^T G_{ff}^T G_{ff} u_k)$$

$$F = -2 \left( (G_{nf} + v^T G_{ff}) R_s - (G_{nf} + v^T G_{ff}) \Phi \Delta X_k - (G_{nf}^T G_{ff} + v^T G_{ff}^T G_{ff}) u_k \right)$$  
(3.21)

Optionally, a weight vector ($m$) can be added to further enhance the performance of the MPC algorithm. The $m$ vector is a $1 \times N_p - N_c$ vector that is typically computed off-line in Matlab and stored either in registers or in BRAMs. In this case, the $P_2$ is an extra penalty factor added to improve the performance [43]. Considering that the SOC is the output to be controlled, and the gain matrices used are $G_z$ and $\Phi_z$, then $E$ and $F$ become:

$$E = 2(G_{nfz}^T G_{nfz} + P_1 - G_{nfz}^T G_{ffz} m - m^T G_{ffz}^T G_{nfz} m + m^T G_{ffz}^T G_{ffz} m + m^T m P_2)$$  
(3.22)

$$F = -2 \left( (G_{nfz} + m^T G_{ffz}) R_s - (G_{nfz} + m^T G_{ffz}) \Phi \Delta X_k - (G_{nfz}^T G_{ff} + m^T G_{ffz}^T G_{ffz} u) \right)$$  
(3.23)

and the unconstrained solutions is

$$\Delta u^c = (G_{nfz}^T G_{nfz} + P_1 - G_{nfz}^T G_{ffz} m - m^T G_{ffz}^T G_{nfz} m + m^T G_{ffz}^T G_{ffz} m + m^T m P_2)^{-1} \left( (G_{nf} + v^T G_{ff}) R_s - (G_{nf} + v^T G_{ff}) \Phi \Delta X_k - (G_{nf}^T G_{ff} + v^T G_{ff}^T G_{ff}) u_k \right)$$  
(3.24)

Next, the constraints are developed. Possible constraints for this model are the control input $u_k$, the control input increment $\Delta u_k$, the terminal voltage $v$, and the maximum SOC $z_k$. In this architecture we are controlling $u_k$, $v$, and $z_k$. The developments of $M$ and $\gamma$ are detailed in [49], the final equation (3.25) is presented below.

$$M = \begin{bmatrix} 1 \\ -1 \\ (G_{nfv} + G_{ffv} m) \\ -(G_{nfv} + G_{ffv} m) \\ (G_{nfz} + G_{ffz} m) \end{bmatrix} \quad \text{and} \quad \gamma = \begin{bmatrix} u_{max} - u_k \\ -u_{min} + u_k \\ v_{max} - (\Phi_v \Delta X + G_{ffv} m u_k + OCV) \\ -v_{min} - (\Phi_v \Delta X + G_{ffv} m u_k + OCV) \\ z_{max} - \Phi_z \Delta X - G_{ffz} m u_k \end{bmatrix}$$  
(3.25)
We can now use the inequality from (2.26) and shown below to determine if the unconstrained solution is the optimal solution given our specified constraints:

\[ M \Delta u^o \leq \gamma \]

If the above equation fails for any element of the constraint vectors, the correction factor is computed and applied using step 2 of the HQP to iteratively calculate the correction factor (2.41) for the optimal control increment, \( \Delta u_{k+1} \) from (2.42).

### 3.2.3 Applying Control Signal

The control signal and the state signal are computed and updated using equation (3.6). The first element of the \( \Delta U_k \) is used to update the control signal as shown in equation (3.26).

\[
u_{k+1} = u_k + \Delta u_{k+1}
\]

(3.26)

Next, the new control signal is used to determine the states for the next iteration, as presented in equation (3.27):

\[
x_{k+1} = A_m x_k + B_m u_{k+1}
\]

(3.27)

The state of charge (SOC), (i.e., \( x_{k+1}[0] = z_{k+1} \)), is compared to a reference value to determine if the Li-ion battery is fully charged. If the SOC is less than the reference values \( (z_{k+1} < \text{reference}) \) or not fully charged, then the MPC algorithm is repeated to compute the next control signal.

### 3.3 Design Approach and Development Platform

The proposed embedded architectures for the fast-charge MPC algorithm are inspired by the modified MPC algorithm for the Lithium-ion battery cell-level MPC modeled by Trimboli’s group \([6],[43],[49]\). Trimboli’s research group provided the source code written in Matlab for the fast-charge MPC algorithm \([43]\) described above. This validated Matlab model is used as the baseline for the performance and functionality comparison presented in Section 3.5 of this chapter.

For all the Fast-charge experiments, both software and hardware versions of various computations are implemented using a hierarchical platform-based design approach to facilitate
component reuse at different levels of abstraction. The designs consist of different abstraction levels, where higher-level functions utilize lower-level sub-functions and operators. The fundamental operators such as add, subtract, multiply, divide, compare, and square-root are at the lowest-level; the vector and matrix operations including matrix multiplication/addition/subtraction are at the next level; the four stages of the MPC, i.e., Model Generation, Unconstrained Optimal Solution, Hildreth’s QP Process, and State and Plant Update, are at the third level of the design hierarchy; and the MPC is at the highest level.

All Fast-Charge hardware and software experiments are carried out on the ML605 FPGA development board [53], which utilizes a Xilinx Virtex 6 XC6VLX240T-FF1156 device. The development platform includes large on-chip logic resources (37680 slices), MicroBlaze soft processors, and 2MB on-chip BRAM (Block Random Access Memory) to store data/results. The hardware modules are designed in mixed VHDL and Verilog. They are executed on the FPGA (running at 100MHz) to verify their correctness and performance. Xilinx ISE 14.7 and XPS 14.7 are used for the hardware designs. ModelSim SE and Xilinx ISim 14.7 are used to verify the results and functionality of the designs. The software modules are written in C and executed on the 32-bit RISC MicroBlaze soft processor (running at 100 MHz) on the same FPGA. The soft processor is built using the FPGA general-purpose logic. Unlike the hard processors such as the PowerPC, the soft processor must be synthesized and fit into the available gate arrays. Xilinx XPS 14.7 and SDK 14.7 are used to design and verify the software modules. The hardware modules for the fundamental operators are designed using single precision floating-point units [54] from the Xilinx IP core library. The MicroBlaze is also configured to use single precision floating-point unit for the software modules. Conversely, the baseline Matlab model was designed using double-precision floating-point operators. This has caused some minor discrepancies in certain functionalities of the fast-charge MPC algorithm. These discrepancies are detailed in Section 3.5 of this chapter.

36
The speedup resulting from the use of hardware over software is computed using the following formula:

\[
\text{Speedup} = \frac{\text{Baseline Execution Time (Software)}}{\text{Improved Execution Time (Hardware)}}
\]  

(3.28)

**System-Level Design:** For all the designs in this chapter, on-chip BRAMs are used to store the input data needed to process the MPC algorithm and to store the final and intermediate results from the MPC algorithm. As shown in Figure 3, the AXI (Advanced Extensible Interface) interconnect acts as the glue logic for the system.

![System-level architecture for Fast-charge MPC](image)

Figure 3: System-level architecture for Fast-charge MPC

Both hardware versions also incorporate a MicroBlaze soft processor configured to have 128KB of local on-chip memory. As illustrated in Figure 3, the user-designed hardware module communicates with the MicroBlaze processor and with the other peripherals via the AXI bus [55], through the AXI Intellectual Property Interface (IPIF) module, using a set of ports called the Intellectual Property Interconnect (IPIC). For the hardware designs the MicroBlaze processor is only employed: to initiate the control cycle; to apply the control signals to the plant; and to determine the plant output signal. The user-designed hardware module performs the calculations for the entire fast-charge MPC algorithm. The execution times for the hardware as well as the software on MicroBlaze are obtained using the AXI Timer [56] running at 100MHz.
3.4 Embedded Hardware and Software Architectures for MPC

Apart from the main objective, one of the design goals is to create these embedded architectures to monitor and control not only one battery cell but also multiple battery cells individually, in a time-multiplexed fashion, thus significantly reducing the hardware resources required for BMS.

Initially, We investigate the functional flow of the MPC algorithm in [43], and then decompose the algorithm into four high-level stages (as shown in Figure 4) to simplify the design process. The operations of the four consecutive stages are as follows:

- **Stage 1: Model Generation** – Compute the augmented model and gain (or data) matrices.
- **Stage 2: Unconstrained Optimal solution** – Check the plant state (i.e., whether the charging is completed or not); compute the global (unconstrained) optimal solution and determine whether the constraints are violated or not.
- **Stage 3: HQP** – Compute the correction factor using the HQP procedure and apply to the control increment, executed if and only if one or more constraints are violated.
- **Stage 4: State and Plant Update** – Compute the new plant states and plant outputs. It should be noted that for experimental purposes the terminal voltage, is computed in Stage 4, however, in a real-world scenario, the terminal voltage would be a measured value.
In order to enhance the performance and area efficiency of both embedded hardware and software designs, all the time-invariant computations are relocated to Stage 1 from other stages of MPC algorithm. In this case, Stage 1 is considered the start-up phase, which is performed only once at the beginning of the Control Prediction Cycle, whereas, subsequent stages (Stages 2, 3, and 4) are performed in every sampling interval of the Control Prediction Cycle in iterative fashion. Relocating the time-variant computations to Stage 1 dramatically reduces the time taken to perform the subsequent stages and enhances the overall speedup of the MPC algorithm. For example, consider the P matrix (2.34) used in HQP typically associated with Stage 3. This P is created by multiplying a 32-word vector by a 32-word vector to create a 32X32 matrix, which comprises 1024 multiplications. This computation would usually take 1032 clock cycles per iteration, if we employ a FPU multiplier, which produces a multiplication results every clock cycle after an initial latency of 8 clock cycles. With the original fast-charge MPC algorithm [43], the P parameter is computed every sampling cycle, when the Stage 3 is executed. By moving the P parameter computation to Stage 1, we save 1032 clock cycles per iteration. These execution times and speedups are detailed in Section 3.5 of this chapter.
There are two major advantages of using the modified fast-charge MPC algorithm for embedded systems designs over other MPC algorithms in the existing literature:

- The fast-charge MPC algorithm contains only one matrix inversion, which is time-invariant, thus needs to be computed only once, provided that the temperature remains constant.
- The dual-mode approach allows for a short prediction horizon \((N_P = 10)\) and a short control horizon \((N_C = 1)\), which reduces the size of the matrices while maintaining the required stability. It also reduces the single matrix inversion to a scalar inversion, thus eliminating matrix inversion entirely.

The proposed embedded architectures for the fast-charge MPC are detailed in the following sub-sections.

### 3.4.1 Embedded Software Architecture

The software for the fast-charge MPC algorithm is written in C using the XCode integrated development environment. This software design is executed on a desktop computer with dual core i7 processor. The results are compared and verified with the baseline results from the Matlab code[11]. Both the C and Matlab results are also used to verify the results from the embedded software and hardware designs.

Due to the limited resources of the embedded devices, it is imperative to reduce the code-size of the embedded software design. The desktop software design above is drastically modified to fit into the embedded microprocessor, i.e., MicroBlaze. In this case, we make the code leaner and simpler, in such a way that it fits into the program memory available with the embedded microprocessor, without affecting the basic structure and the functionalities of the algorithm. Many design decisions for hardware optimizations are also employed to optimize the embedded software design whenever possible, including reordering certain operations to reduce the redundancy (e.g. computing \(P\) in Stage 1). We also incorporate techniques to reduce the use of for-loops
appropriately, and perform loop unrolling when the speed is important. Furthermore, we identify parts of the program, where off-line computations can be done without exceeding the data memory requirements.

The embedded software is designed to mimic the hardware designs. Apart from the usual computation modules, embedded software design consists of two sub-modules. One sub-module computes the temperature dependent model parameters of resistances $R_0$ and $R_1$, time-constant $\tau$, and $Q_{\text{charge}}$, whereas the other sub-module computes the open circuit voltage (OCV) from the state of charge (SOC). The required parameters for the software design are computed from the measured data using a Cubic Spline technique. Since the empirical data are unlikely to change, the Cubic Spline data are computed off-line with Matlab. The software flow for the fast-charge MPC is presented in Table 1.

Table 1: Software algorithm for Fast-charge MPC

<table>
<thead>
<tr>
<th>Stage</th>
<th>MPC Software Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Create cell parameters based on temperature</td>
</tr>
<tr>
<td></td>
<td>Build A, B, C and D matrices</td>
</tr>
<tr>
<td></td>
<td>Calculate gain matrices, $\Phi$ and $G$</td>
</tr>
<tr>
<td></td>
<td>Create $G_{nf}$ and $G_{ff}$ ($nf =$ near future and $ff =$ far future dual mode data)</td>
</tr>
<tr>
<td></td>
<td>Calculate $E$ and time-invariant portions of $F$</td>
</tr>
<tr>
<td></td>
<td>Calculate $P$ (matrix for optimization by Hildreth QP)</td>
</tr>
<tr>
<td></td>
<td>Build $M$ (constraints vector)</td>
</tr>
<tr>
<td>2.</td>
<td>Start loop – compare $x_{m}[0]$ (SOC) to determine charge level, if full, exit.</td>
</tr>
<tr>
<td></td>
<td>Calculate $F$</td>
</tr>
<tr>
<td></td>
<td>Solve for optimal unconstrained solution, $\Delta u^e$</td>
</tr>
<tr>
<td></td>
<td>Solve for the open circuit voltage (OCV) from SOC</td>
</tr>
<tr>
<td></td>
<td>Build $\gamma$ (constraints vector)</td>
</tr>
<tr>
<td></td>
<td>Compare: $M\Delta u^e \leq \gamma$. If True, GOTO 4, else GOTO 3</td>
</tr>
<tr>
<td>3.</td>
<td>Hildreth QP, create $\lambda$ until convergence or maximum loops</td>
</tr>
<tr>
<td></td>
<td>develop new $\Delta u$ correction factor that meets constraints</td>
</tr>
<tr>
<td>4.</td>
<td>Calculate the next control signal, update the state, terminal voltage and SOC</td>
</tr>
<tr>
<td></td>
<td>GOTO Start Loop (2)</td>
</tr>
</tbody>
</table>
3.4.2 Embedded Hardware Designs

Two hardware versions have been developed for the Fast-charge MPC algorithm: a register-based hardware version 1 (HW_v1) and the on-chip BRAM-based hardware version 2 (HW_v2). With HW_v1, a customized and parallel processing architecture is introduced to perform the matrix computations in parallel by mostly utilizing registers to store the data/results. By employing a parallel processing architecture, we anticipated an enhancement of the speedup of the overall MPC algorithm. With HW_v2, a fully pipelined and optimized architecture is introduced to address certain issues that have arisen with HW_v1. By employing on-chip BRAMs to store the data/results, we expected a reduction in overall area, since the registers and the associated interconnects (in HW_v1) typically occupy a large space on chip. Conversely, the existing on-chip BRAMs are limited to two access ports, hence could potentially hinder parallel processing of computations.

The register-based HW_v1 is designed in such a way to follow the software functional flow of the MPC algorithm presented in Figure 5; thus, has similar characteristics as the embedded software design. In this case, the registers are used to hold the matrices, which is analogous to the indexing of the matrices in C programming. It should be noted that HW_v1 is a proof-of-concept work; where HW_v2 addresses certain issues that have arisen with HW_v1.

![Functional/data flow for Stage 1](image)

Xilinx offers two types of floating-point IP cores: AXI-based and non-AXI-based. For the register-based HW_v1, the standard AXI-based IP cores are used for the fundamental operators. These IP cores provide standardized communications and buffering capabilities, and occupy less area on chip, at the expense of higher latency. For the BRAM-based HW_v2, the non AXI-based
IP cores are used for the fundamental operators. These IP cores allow the lowest latency adder (5-cycle latency) and multiplier (1-cycle latency) units to support 100MHz system clock, at the expense of occupying more area on chip. The non AXI-based cores have less stringent control and communication protocols, thus proper timing of signals is required to obtain accurate results. With HW_v2, we utilize fewer IPs that are more complex but offer lower latency but more resource intensive IP cores, since it consists of fewer multipliers and adders. With HW_v1, we have to use higher latency but less resource intensive IP cores, since it comprises large number of multipliers and adders, due to the parallel processing nature of the design.

Initially, the embedded hardware architectures for each stage as designed and developed as separate modules are analogous to our hierarchical platform-based design approach. The hardware designs for each stage consist of a data path and a control path. The control path manages the control signals of the data path as well as the BRAMs/registers. Next, a Top-Level Module is designed to integrate the four stages of the MPC algorithm, and to provide necessary communication and control among the stages. Among various control and communication signals, the Top-Level Module ensures that the plant outputs, the state values, and the input control signals are routed to the correct stages at proper times. The control path of the Top-Level Module consists of several Finite State Machines (FSMs) and multiplexers to control the timing, routing, and internal architectures of the designs. The internal hardware architectures of the four stages of the MPC algorithm are detailed in the following sub-sections.

3.4.2.1 Stage 1: Augmented Model and Gain Matrices

Stage 1, the initial phase of the MPC algorithm, is performed only once at the beginning of the Control Prediction Cycle. All the time-invariant computations, which are deemed independent of \( x_k \) and \( u_k \) are relocated and performed in Stage 1, to ease the burden of the compute-intensive iterative portions of the MPC algorithm.
The general functional and data flow of Stage 1 (for both HW_v1 and HW_v2) is depicted in Figure 5. As illustrated, the relocated computations include $E$, $M$, $P$ and the sub-matrices for $F$. The Stage 1 also consists of the Augmented Model and Gain Matrices for both the hardware versions, and a Parameter Module only for HW_v2. Initially, Augmented Model (in Figure 5) is created from equations (6), (7), and (8) depending on the temperature dependent parameters, initial states $x_k = [0, 0.5]$, and initial control input $u_k = 0$.

### 3.4.2.1.1 Computing Parameters

Since varying temperature is inevitable in the real-world scenario, for HW_v2, we integrate an additional Parameters Module to compute the four temperature dependent variables $Q$, $R_0$, $R_1$, and $r$, utilized in the Augmented Model. These variables are computed using a Cubic Spline Interpolation of empirical data provided for Li-ion batteries. We use four Cubic Spline equations to compute the four variables. The general formula for a Cubic Spline Interpolation is: $y = a_3 x^3 + a_2 x^2 + a_1 x + a_0$, where $x=T-ref$; in this case, $T$ is the temperature, and $ref$ is (min) from Table 2. As presented in Table 2, the Cubic Spline approach uses six temperature regions. For HW_v2, initially, the coefficients (i.e., $a_3$, $a_2$, $a_1$, and $a_0$) of the equations for all four variables are produced by Matlab codes and stored in a BRAM configured as a ROM. If the temperature varies, the base address of the temperature region in use ($ref$) is passed to the Parameter Module, and the corresponding variables (parameters) are computed.

For HW_v1, in Stage 1, the Parameter Module is excluded due to the resource constraints on chip. In this case, for HW_v1, the temperature dependent parameters are considered as constants, and stored in the registers, with the premise that the temperature will remain constant [49]. For the experimental results and analysis (in Section 3.5 of this chapter), we consider the temperature to be constant for both hardware versions. With the current experimental setup, the additional Parameter Module does not impact the precision or the performance of the proposed embedded designs.
Table 2: Temperature regions for Cubic Spline

<table>
<thead>
<tr>
<th>Region</th>
<th>Range</th>
<th>Reference</th>
<th>Region</th>
<th>Range</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-15°C ≤ T &lt; -5°C</td>
<td>-15°C</td>
<td>4</td>
<td>15°C ≤ T &lt; 25°C</td>
<td>15°C</td>
</tr>
<tr>
<td>2</td>
<td>-5°C ≤ T &lt; 5°C</td>
<td>-5°C</td>
<td>5</td>
<td>25°C ≤ T &lt; 35°C</td>
<td>25°C</td>
</tr>
<tr>
<td>3</td>
<td>5°C ≤ T &lt; 15°C</td>
<td>5°C</td>
<td>6</td>
<td>35°C ≤ T</td>
<td>35°C</td>
</tr>
</tbody>
</table>

The internal architecture of the Parameter Module (from Figure 5) for HW_v2 is depicted in Figure 6. This module executes a Cubic equation for each of the temperature dependent variables. The regions contain different coefficients based on empirical data. As illustrated in Figure 6, these coefficients are stored in ROM; and the region defines the memory location of the coefficients and the reference values. To execute the Cubic equation the Parameter Module uses an 8-cycle multiplier, 12-cycle adder and multiplexers. There are four Cubic equations, one for each parameter. This module initially computes the $x$ term for all four equations, and then adds the constants. Next, the $x^2$ term is calculated and multiplied by the four corresponding coefficients, and the resulting value is added to the previous terms. This is repeated for the $x^3$ term. This multiply-add approach is timed in such a way to eliminate the need for extra registers to hold the values. Once the add completes, the next multiply is ready to be added to the total.

![Figure 6: Parameter Module for HW_v2](image)

3.4.2.1.2 Creating Augmented Model

After computing the parameters, we design and develop the matrices of the Augmented Model. The elements of the modified fast-charge MPC state-space equations (i.e., equations (1)-(8) [49]) are presented below in (3.29).
\[ A_m = \begin{bmatrix} 1 & 0 \\ 0 & e^{-\Delta t/R_1C_1} \end{bmatrix}, \quad B_m = \begin{bmatrix} -\frac{\eta \Delta t}{Q} \\ R_1 \left( 1 - e^{-\Delta t/R_1C_1} \right) \end{bmatrix}, \quad \text{and } x_{m,k} = \begin{bmatrix} z_k \\ v_{c_{1,k}} \end{bmatrix} \] (3.29)

The augmented state-space equation matrices are given in equation (3.9) (in Section 3.2), where, \( \Delta t \) is the sampling time (considered as 1s), and \( \eta \) is the cell efficiency (considered as 0.997). Also, the \( e^{-\Delta t/\tau} \) term is currently stored as a constant and an input for both hardware versions. For both HW_v1 and HW_v2, the Augmented Model computes all the elements in equation (3.29), and then stores the values in the correct order of the matrices, in registers (for HW_v1) and in BRAMs (for HW_v2). In addition, the Augmented Model for HW_v2 computes the penalty factors, \( P_1 \) and \( P_2 \), used in equation (3.24).

The internal architecture of the Augmented Model for HW_v1 is shown in Figure 7. In order to compute the values in equation (3.29) for the augmented model, a subtraction FPU, multiplication FPU, a division FPU and three multiplexers are required. The results are stored in registers to be forwarded directly to the subsequent modules.

![Figure 7: Augmented model for HW_v1](image)

### 3.4.2.1.3 Computing Gain Matrices

Next, the gain matrices are computed, including the \( \Phi \), \( G_{vf} \), and \( G_{zf} \). Each gain matrix has identical computations, which are independent of each other. In this design, the \( \Phi \) and \( G \) matrices are developed for both the terminal voltage \( v_k \), and SOC \( z_k \) separately, resulting in \( \Phi_v, \Phi_z, G_v, \) and \( G_z \). The gain matrices are derived from equation (3.12), where \( \Phi_v \) and \( \Phi_z \) are:
\[ \Phi_v = \begin{bmatrix} \tilde{C}_v \\ \tilde{C}_v \tilde{A} \\ \tilde{C}_v \tilde{A}^2 \\ \vdots \\ \tilde{C}_v \tilde{A}^{N_p-1} \end{bmatrix} \quad \text{and} \quad \Phi_z = \begin{bmatrix} \tilde{C}_z \\ \tilde{C}_z \tilde{A} \\ \tilde{C}_z \tilde{A}^2 \\ \vdots \\ \tilde{C}_z \tilde{A}^{N_p-1} \end{bmatrix} \] (3.30)

It should be noted that in our design, from equation (3.9), the \( \tilde{B} \) is considered as \([0 \ 0 \ 1]^T\), thus each column of \( G \) is derived from the third column of \( \Phi \). This simply requires an arrangement of the elements of the \( G \) matrix in registers or BRAMs, instead of computing these elements. In this case, \( G_{nf} \) is a \( N_p \times N_c \) matrix and \( G_{ff} \) is a \( N_p \times N_p - N_c \) matrix. As shown in equation (3.31), for \( N_c = 1 \), \( G_{nf} \) is the first column of \( G \), from equation (3.12), and \( G_{ff} \) is comprised of the rest of the columns. Utilizing \( \tilde{A}, \tilde{B}, \tilde{C}_v \) and \( \tilde{C}_z \), which incorporate the feed-through term from equation (3.9), we create \( G_{nfv}, G_{ffv}, G_{nfz}, \) and \( G_{ffz} \).

\[
G = \begin{bmatrix}
\tilde{C} \tilde{B} & 0 & 0 & \cdots & 0 \\
\tilde{C} \tilde{A} \tilde{B} & \tilde{C} \tilde{B} & 0 & \cdots & 0 \\
\tilde{C} \tilde{A}^2 \tilde{B} & \tilde{C} \tilde{A} \tilde{B} & \tilde{C} \tilde{B} & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\tilde{C} \tilde{A}^{N_p-1} \tilde{B} & \tilde{C} \tilde{A}^{N_p-2} \tilde{B} & \tilde{C} \tilde{A}^{N_p-3} \tilde{B} & \cdots & \tilde{C} \tilde{A}^{N_p-N_c} \tilde{B} \\
G_{nf} & & & & G_{ff}
\end{bmatrix}
\] (3.31)

The internal architecture for computing the \( \Phi \) matrix (for both HW_v1 and HW_v2) is shown in Figure 8. The size of \( \Phi \) is determined by the prediction horizon, \( N_p \), the number of states, \( N_s \), and the number of inputs, \( N_{in} \) and is an \( N_p \times (N_s+N_{in}) \) matrix. As illustrated, the \( \Phi \) includes three multiply-and-accumulate units to compute three elements of each row in parallel. Instead of adding a zero (0) to the first term, as in a typical multiply-and-accumulate unit; in this case, the first term is placed in a register until the second term is ready for the add operation. After the addition of the first two terms, the rest of the terms are subjected to multiply-and-accumulate operation.
As shown in Figure 8, the internal architecture also comprises a feedback-loop unit, which determines the appropriate values to be loaded in each iteration. In this case, each subsequent row of $\Phi$ is the previous row multiplied by $\tilde{A}$. Our design comprises three multiply-and-accumulate (MAC) units that compute each column of $\tilde{A}$ (as shown in Figure 9) in parallel.

$$
\begin{bmatrix}
\Phi_{row}
\end{bmatrix} =
\begin{bmatrix}
A_0 & A_1 & A_2
\end{bmatrix} = \Phi_{row\_next}
$$

Figure 9: Organization of $\Phi\tilde{A}$

As demonstrated in Figure 8, both hardware versions have the same internal architecture for computing the $\Phi$ matrix. In this case, HW_v1 waits until $\Phi$ matrix computation is completed, and then loads $G_{rf}$ and $G_{gf}$. Also, HW_v1 employs two Gain Matrices Modules to compute $[\Phi_v, G_{nf}v, G_{ff}v]$ and $[\Phi_z, G_{nf}z, G_{ff}z]$ matrices in parallel.

Conversely, HW_v2 computes each row of the $\Phi$ matrix, and then saves the row term in an appropriate memory location, to subsequently build $\Phi$, $G_{rf}$ and $G_{gf}$ utilizing an addressing algorithm. Furthermore, HW_v2 computes and saves the $\Phi_v\tilde{A}$ and $\Phi_z\tilde{A}$ matrices. As illustrated in Figure 10, the calculation of $\Phi$ and $\Phi\tilde{A}$ only differs by one row. Hence, by merely computing one
additional row, \( \Phi \tilde{A} \) can be built in the same fashion and at the same time as \( \Phi, G_{rf} \) and \( G_{ff} \), using one extra iteration.

\[
\Phi = \begin{bmatrix}
\tilde{C} \\
\tilde{CA} \\
\tilde{CA}^2 \\
\vdots \\
\tilde{CA}^{N_r-1}
\end{bmatrix}
\text{ and } \Phi \tilde{A} = \begin{bmatrix}
\tilde{CA} \\
\tilde{CA}^2 \\
\vdots \\
\tilde{CA}^{N_r}
\end{bmatrix}
\]

Figure 10: Comparison of \( \Phi \) and \( \Phi \tilde{A} \)

Unlike HW_v1, HW_v2 computes the \([\Phi_v, G_{nfv}, G_{ffv}, \Phi_v \tilde{A}]\) and \([\Phi_z, G_{nfz}, G_{ffz}, \Phi_z \tilde{A}]\) sequentially. The functional architecture of the Gain Matrices for HW_v2 is depicted in Figure 11. The hardware module for computing the \( \Phi \) matrix (from Figure 8) is reused in this module.

Figure 11: Internal architecture for gain matrices and \( \Phi \tilde{A} \) for HW_v2

HW_v1 computes \( \Phi \tilde{A} \) in a separate module (as in Figure 12), after completing the \( \Phi \) matrix computation. In this case, we employ 10 MAC units to compute all the elements in each column of \( \Phi \tilde{A} \) in parallel. As illustrated in Figure 12, the columns are computed sequentially. Also, HW_v1 employs two \( \Phi \tilde{A} \) modules to compute \( \Phi_v \tilde{A} \) and \( \Phi_z \tilde{A} \) in parallel.
3.4.2.1.4 Time-Invariant Computations for HW_v1

As mentioned in Section 3.4.2.1, all the time-invariant computations (E, M, P, and sub-matrices of F), which are deemed independent of $\chi_k$ or $u_k$ (from Stages 2 and 3) are relocated to Stage 1, thus significantly reducing the computation burden in other stages. For HW_v1 and HW_v2, these computations are designed using different techniques. For register-based HW_v1, we employ parallel processing architecture, whereas BRAM-based HW_v2 is executed in pipeline fashion.

**E Module for HW_v1**: The architecture for HW_v1 is presented first since it intuitively follows the order of operations. Considering the equation (3.20) from Section 3.2.2, there are no $\chi$ or $\Delta u$ terms. As a result, and barring temperature changes, $E$ remains constant and can be performed in Stage 1. The $E$ computation is decomposed into several sub-functions in such a way that each sub-function comprises only one matrix computation. Then for HW_v1, separate sub-modules perform the different matrix computations such as: a vector-scalar multiplication ($V S$), a vector-vector multiplication ($V V$), a vector-matrix multiplication ($V M$), and vector-matrix transpose multiplication ($V^T M^T$). The decomposed computations are presented in equations (3.32) – (3.40):

$$E_1 = G_{nfz}^T G_{nfz} \quad \quad \quad \quad (3.32)$$

$$P_1 = rw(1 - \gamma_p) \quad \quad \quad \quad (3.33)$$

$$E_2 = E_2a m, E_{2a} = G_{nfz}^T G_{ffz} \quad \quad \quad \quad (3.34)$$

$$P_2 = rw\gamma_p \quad \quad \quad \quad (3.35)$$

$$E_3 = E_{3a} G_{nfz}, E_{3a} = m^T G_{ffz}^T \quad \quad \quad \quad (3.36)$$

$$E = E_1 + P_1 + E_2 + E_3 + E_4 + E_5 \quad \quad \quad \quad (3.37)$$
\[ E_4 = E_{4a} \mathbf{m}, \quad E_{4a} = E_{3a} g_{ffz} \]  \hspace{1cm} (3.38) \hspace{1cm} E_{inv} = E^{-1}  \hspace{1cm} (3.39)

\[ E_5 = E_{5a} p_2, \quad E_{5a} = m^T \mathbf{m} \]  \hspace{1cm} (3.40)

In this case, the control horizon is \( N_c = 1 \), and \( E \) and the inverse of \( E \) are scalars, which significantly reduces the complexity of the MPC algorithm. Since division and inversion floating-point operations typically incur the highest latency, by computing the \( E^{-1} \) in Stage 1, the subsequent stages mostly comprise multiplications operations with much lower latency (1- to 8-cycles based on the FPU). For HW_v1, the final internal architecture for \( E \) module is derived from equation (3.20) from Section 3.2.2. From the equation for \( F \) (3.21), it is observed that the last term is in fact \( (E_2 + E_4 + E_5) u_k \). In this case, integrating equation (3.41) to the \( E \) Module, reduces the number of outputs of \( F_{3a} \), i.e., from three 32-bit values to a single 32-bit value.

\[ F_{3a} = E_2 + E_4 + E_5 \]  \hspace{1cm} (3.41)

As illustrated in Figure 13(a), the \( E \) Module for HW_v1 computes the equations (3.32) – (3.40). As shown, the \( E \) Module for HW_v1 comprises several sub-modules to compute various vector and matrix operations. These sub-modules utilize MAC units (Figure 13(c)) to perform the necessary vector/matrix operations. Our MAC unit is designed in such a way to reduce the clock cycles of each final MAC result by 12 clock cycles. In our designs, the vector-vector multiplication (\( \text{VV} \)) is identical to vector squared (\( \text{V}^2 \)) except the former accepts two separate vectors, whereas the latter accepts only one. Vector-matrix multiplication (\( \text{VM} \)) and vector-matrix transpose multiplication (\( \text{VTM}^T \)) are also identical, except the former uses the number of columns of the matrix to determine the number of processing elements (PEs), whereas the latter uses the number of rows of the matrix to determine the number of PEs. Furthermore, as depicted in Figure 13(b), we design a separate sub-module to compute the tuning parameters \( P1 \) and \( P2 \), which is executed in parallel with the \( E \) Module. This significantly reduces the control logic required for the \( E \) Module.
**F_sub Module for HW_v1:** We design the F_sub Module to compute the sub-matrices for $F$. This module computes all the $F$ terms, presented in equations (3.42)-(3.45), which are derived from equation (3.21).

$$F_{1a} = G_{nfx}^T + E_{3a} \quad (3.42)$$

$$F_{2a} = F_{1a} \Phi_z \quad (3.43)$$

$$F_1 = F_{1a} R_s \quad (3.44)$$

$$F_{2c} = F_{2a} \tilde{A} \quad (3.45)$$

The internal architecture of the F_sub Module is depicted in Figure 14, which consists of a vector-addition module ($\mathbf{V+V}$), a vector-accumulation module ($\mathbf{V\text{Acc}}$), two $\mathbf{VM}$ modules, and a FPU multiplier. In this case, the former two sub-modules ($\mathbf{V+V}$ and $\mathbf{V\text{Acc}}$) utilize FPU adders to perform the required operations.
Figure 14: Internal architecture for F_sub module for HW_v1

**M Module for HW_v1:** The M Module is designed to compute $M$ constraints. The $M$ and $\gamma$ are presented in equation (3.25). All the elements of $M$ and some elements of $\gamma$ can be computed with the equations (3.46)-(3.52) as follows:

- $M_{posv} = G_{ffv}m$  \hspace{1cm} (3.46)
- $M_{posz} = (G_{nfz} + M_{posz})$  \hspace{1cm} (3.47)
- $M_{posz} = G_{ffz}m$  \hspace{1cm} (3.48)
- $\Psi_A v = \Psi \bar{A}$  \hspace{1cm} (3.49)
- $M_{posv} = (G_{nfv} + M_{posv})$  \hspace{1cm} (3.50)
- $\Phi_A z = \Phi \bar{A}$  \hspace{1cm} (3.51)
- $M_{negv} = -M_{posv}$  \hspace{1cm} (3.52)

HW_v1 employs separate modules to perform $G_{ff}m$ and $\Phi \bar{A}$. The internal architecture for $\Phi \bar{A}$ is demonstrated in Figure 12, and the architecture of $G_{ff}m$ computation is similar to the VM sub-module. With the M Module, the negation operations are performed by reversing the most significant bit (MSB) of the 32-bit floating-point values, thus reducing the logic utilized for these operations.

**P Module for HW_v1:** The P Module for HW_v1 is derived from equation (2.34), $P = ME^{-1}M^T$. As discussed in Section 2.2.4, the Hildreth’s Quadratic Programming (HQP) utilizes this equation and (2.35) to compute $\lambda$ vector. Equation (2.34) is decomposed into equations (3.53) and (3.54) as follows:

$$M_{condinv} = ME^{-1}$$  \hspace{1cm} (3.53)
where, equation (3.67) performs a vector-scalar multiplication.

\[ P = M_{conEinw}M^T. \]  

(3.54)

In this case, \( P \) is a square symmetric matrix; hence, the number of columns and rows are equal to the length of \( M \) (in our case 32). To compute this matrix, we use an efficient computation assignment algorithm developed by [57]. Utilizing this algorithm, elements of the \( P \) matrix are executed in parallel using several parallel PEs. In this case, \( n \) number of PEs process \( n \) number of elements (of the matrix) at a time and computes the whole \( P \) matrix with no idle time.

Due to the size of the \( P \) matrix (32×32), registers are not suitable to store the matrix on chip. The attempt to store the matrix using registers caused my initial design to exceed the chip resources by 25%. Therefore, BRAM is integrated into the P Module to store the \( P \) matrix in HW_v1. In this case, only 2 PEs are used to compute elements of the \( P \) matrix, due to the port limitations of the BRAMs. The PEs consist of a multiplier and logic elements to ensure that the inputs to the multiplier are ready every clock cycle to reduce latency. The results of \( P \) matrix computation are utilized in Stage 3.

In summary, HW_v1 is designed with separate modules, including \( GFFm, \Phi A, E, F_{sub}, M, \) and \( P, \) to execute various computations in Stage 1. In this case, two \( GFFm \) modules compute equations (3.46) and (3.48) in parallel, and two \( \Phi A \) modules compute equations (3.49) and (3.51) in parallel. The \( F_{sub} \) module computes equations (3.42)-(3.45), the \( M \) module computes equations (3.47),(3.50) and (3.52), and finally \( P \) module computes equations (3.53) and (3.54).

### 3.4.2.1.5 Time-Invariant Computations for HW_v2

For the internal architecture for HW_v2, we use a novel and unique approach to perform the \( E, F, M, \) and \( P \) computations. A unique pipelined multiply-and-accumulator (MACx) module is designed to perform various vector and matrix multiplication operations in sequence. The MACx has a wrapper, which handles reading/writing from/to the BRAMs during the vector and matrix operations.
For HW_v2, the matrix addition and the scalar operations are typically performed in the E, M, and P Modules. In this case, the E Module organizes the scalar addition, multiplication, and division necessary to generate $E^{-1}$. The M module performs the scalar addition and multiplication to generate $M$ (for equations (3.47), (3.50) and (3.52)), and $F_{t_o}$ (for equation (3.42) and (3.47)). When using BRAMs to store the vectors. The equations (3.42) and (3.47) generate the same values.

Figure 15 shows the top-level architecture for HW_v2 for the time-invariant computations $E$, $F$, $M$, and $P$. As illustrated, the multiply, adder, and divider FPUs are shared among the E, M and P Modules, and multiplexers are utilized to control the routing and internal architecture of these modules. The outputs of these FPUs are forwarded to the E, M and P Modules, and the results are stored in the BRAMs.

The internal architecture of the pipelined MACx module is depicted in Figure 16. The MACx is primarily designed to perform vector multiplications. The Input module of the MACx decomposes the matrix computations into vector operations. The pipelined MACx (for HW_v2) executes the vector operations (for 3 or more vectors) faster than its parallel HW_v1 counterpart. The FPUs are carefully configured to have the lowest latency without compromising the system-clock frequency (100MHz). For HW_v2, the FPUs for the multiplier and the adder have 1-cycle and 5-cycle latencies respectively, whereas for HW_v1, the FPUs for the multiplier and the adder
have 8-cycle and 12-cycle latencies respectively. However, there is a trade-off; low-latency IP cores occupy more area on chip. This might not be an issue for the BRAM-based HW_v2, since the overall design occupies less area on chip compared to the register-based HW_v1. This is not only because HW_v2 employs BRAMs instead of registers to store the data/results, but also it utilizes far less IP cores than HW_v1.

Figure 16: Pipelined MACx module for HW_v2

Furthermore, in HW_v1, computations such as $G_{ffz}m$, are not available for subsequent operations until the whole computation has been completed (i.e., all the elements are computed). Conversely, in HW_v2, after one element is computed in one operation, that element can be used in subsequent operations. For instance, for HW_v2, when MACx completes the first vector computation (i.e. $G_{ffz\_row0} * m$), the resulting element and the first element of $G_{nfz}$ in equation (3.63) are available for the M Module to generate the first element of $M_{posz}$. This dramatically reduces the time required to execute Stage 1, as detailed in Section 3.5 of this chapter.

With the pipelined MACx, the input wrapper module controls the order of the operations (i.e., execution order). Since the computations are performed sequentially, the order of execution is carefully determined to minimize the wait or stall time for dependent operations and to optimize the utilization of the limited memory ports. The two performance bottlenecks of Stage 1 (for HW_v2) are the limited memory ports and the IP core latency. The design uses 3 types of BRAM memory: a dual port ROM that stores constants; a single port RAM-low; and a dual port RAM-high. The input wrapper has access to a single read port in each of the memories. The ports are reserved only when the vectors are being fetched from the memory, and freed once the data is
loaded into the MACx input buffers. The execution order using the pipelined MACx for HW_v2 is as follows:

1. $E_{5a} = m^T m$, equation (3.40). In this case, a single ROM port is utilized to preload the $m$ vector into both input buffers of the MACx. This occurs in parallel with the $\Phi$ and the Gain matrices calculations. After the multiply and add operations of the MACx are completed, the Output MACx module sends a signal to the E module, indicating that this value is ready. The E module accesses the value from the MACx output register and multiplies this value with $P_2$ to create $E_5$. The MACx output register is also the input register used to store the data in RAM-high. This value is stored in the memory and sent to the adder for the E module simultaneously.

2. $E_{3a} = m^T G_{ffz}^T = G_{ffz} m = Mposz_\alpha$, equation (3.36). From step 1, the $m$ vector is already loaded into one input buffer of the MACx, and single RAM-low port is required to load a row of $G_{ffz}$ into the other input buffer of MACx. The multiplier sends a signal to the Input module to preload the next row of $G_{ffz}$ into the MACx input register. The $m$ vector remains in the input buffer until cleared or overwritten. This step continues until all the rows of $G_{ffz}$ have been entered. Once the required vector is available, the Output MACx module sends a signal to the M and Input modules, and then loads the vector into RAM-high. The M module uses this vector ($E_{3a}$) to create $F_{1a}$ and $E_{3a}$ is also used in step 5 to create $E_3$. Next, steps 3 and 4 are selected to be executed, since inputs to these steps are already available. Furthermore, these two steps can be executed in the pipeline with no stall states.

3. $E_1 = G_{nffz}^T G_{nffz}$, equation (3.32). Since $G_{nffz}$ is a vector, a single RAM port is required to load $G_{nffz}$ into both MACx input buffers. After completing this computation, the Output MACx module sends a signal to the E module, indicating that this value is ready. The E module adds this value ($E_1$) to $E_5$ and stores it in a temporary register.

4. $E_{2a} = G_{nffz}^T G_{ffz}$, equation (3.34). From step 3, $G_{nffz}$ is already loaded into the MACx input buffer and a single RAM port is used to pre-fetch the columns of $G_{ffz}$. Once the Multiplier
indicates that it starts executing, the Input module preloads the next column of $G_{ffz}$ into the input buffer to compute the next term of $E_{2a}$. This step continues for all the columns of $G_{ffz}$. In step 6, $E_{2a}$ is used to create $E_2$. As a result, $E_{2a}$ is stored in RAM-low, and a signal is forwarded to the Input module once it is completed.

5. $E_3 = E_{4a} = E_{3a}G_{ffz}$, equation (3.36). The time it takes to load steps 3 and 4 ensures that the operation started in step 2 ($E_{3a}$) is completed. The Input module ensures that this value is ready by checking the complete signal. One port from each RAM is used to preload $E_{3a}$, while a column of $G_{ffz}$ is loaded into the MACx input buffers. This step continues until all the columns of $G_{ffz}$ have been loaded. Upon completion of the MACx operations the Output module sends a signal to the $E$ module indicating that the value ($E_3$) is available. The $E$ module accesses the MACx output register to add this value to $E_5+E_1$.

6. $E_2 = E_{2a}m$, equation (3.34). The $m$ vector is loaded into one MACx input buffer using a ROM port. Simultaneously, $E_{2a}$ is completed, and step 5 is being executed. Then $E_{2a}$ is loaded into the other input buffer using a RAM-low port. Once the MACx operation is completed, the Output module sends a signal to the $E$ module, and the $E$ module accesses the MACx output register to add this value to $E_5+E_1+E_3$.

7. $Mposv_a = G_{ffv}m$, equation (3.46). As mentioned before, the $m$ vector is already present in the input buffer of the MACx. Hence, a RAM port is required to load the rows of $G_{ffv}$ into the other MACx input buffer. This step continues until all the rows of $G_{ffv}$ have been operated on. Once the MACx operations are completed, the Output module sends a signal to the $M$ module. The $M$ module uses this value to build the $M$ constraint vector.

8. $E_4 = E_{4a}m = E_3m$, equation (3.38). For step 8, the $m$ vector is still present in the input buffer, and $E_3$ is completed, while step 7 is being executed. A single RAM-low port is required to load the $E_3$ into the MACx input buffer. Upon completion of the MACx operations, the Output module sends a signal to the $E$ module indicating that $E_4$ is completed. The $E$ module accesses
the RAM input data register to add $E_4 + E_5 + E_1 + E_3 + E_2 + P_1$ to create the final $E$ value.

9. $F_{2a} = F_{1a} \Phi_z$, equation (3.43). $F_{1a}$ is calculated in the M module using the output from step 7 and loaded into a FIFO buffer to eliminate any memory access for step 9. $F_{1a}$ is loaded into the input buffer from the FIFO. Simultaneously, the first column from $\Phi$ is loaded into the other input buffer from RAM. This step continues until all 3 columns of $\Phi$ have been loaded into the MACx. Once the MACx computations for $F_{2a}$ are completed, the MACx Output module sends a signal to the MACx Input module to initiate the execution of step 10.

10. $F_{2c} = F_{2a} \bar{A}$, equation (3.45). Once the Input module receives a signal that step 9 is completed, the $F_{2a}$ vector is loaded into one input buffer and the first column of $\bar{A}$ is loaded into the other input buffer of MACx. This step continues until the three columns of $\bar{A}$ have been loaded into the MACx. Once the computations for $F_{2c}$ are completed, this value is stored in the memory and a Done signal is set to indicate the completion of this step.

### 3.4.2.2 Stage 2: Unconstrained Solution

Stage 2 computes the unconstrained optimal solution and then determines whether the unconstrained solution meets or violates the constraints. If the constraints are violated, Stage 3 is invoked and performs the HQP algorithm to compute a correction factor for a suitable solution. If the constraints are met, Stage 3 is bypassed, and Stage 4 executes. It is necessary to perform the following steps in Stage 2. These steps are also illustrated in Figure 17.

1. Determine whether the battery has reached a full charge, i.e., $x_m[0] \geq 0.9$, which indicates that the state of charge (SOC) is greater than or equal to 90% full. This limit is designed to prevent overcharging of the battery.

2. Compute the current open circuit voltage (OCV) value based on the input SOC ($x_m[0]$).

3. Compute the unconstrained general optimal solution for the control input, $\Delta u^o = -E^{-1} F$, from equation (3.24).

4. Compute the $\gamma$ constraint vector from equation (3.25).
5. Compute $M\Delta u^0$.

6. Compute $K$ from equation (2.35).

7. Compute an element by element comparison, $M\Delta u^0 \leq \gamma$ to determine if the unconstrained solution is optimal or fails constraints.

From the above steps, vector $K$ is computed in Stage 2 because it only needs to be computed once per control interval, although it is utilized in Stage 3. The time interval for controlling the charging of a battery is one second. For instance, the control signal is updated every second for charging or discharging a battery cell. In this case, steps 2 and 3 are performed in parallel; next steps 4 and 5 are performed in parallel; and finally steps 6 and 7 are performed in sequence.

![Figure 17: Overview of Stage 2](image)

### 3.4.2.2.1 Computing OCV for HW_v1 and HW_v2

In step 2, OCV is computed based on the current SOC ($x_m[0]$) value, using a linear interpolation between two data points from the two tables discussed below. The internal architectures to compute the OCV are quite similar for both hardware versions. The only difference being the tables and values required by HW_v2 are stored in the BRAM, whereas the HW_v1 values are stored in registers. In both cases, the linear interpolation uses two tables ($OCV_0$ and $OCV_{rel}$) of empirical data, which depend on the operation of the Li-ion battery. For both HW_v1 and HW_v2, the algorithm for computing OCV using SOC is presented in
Table 3:
Table 3: Determining the Open Circuit Voltage from the State of Charge

<table>
<thead>
<tr>
<th>OCV from SOC</th>
</tr>
</thead>
</table>

1. **Determine the boundary conditions:**
   - If \((x_m[0] < 0)\) use the minimum pre-calculated OCV
   - Else if \((x_m[0] > 1)\) use the maximum pre-calculated OCV
   - Else compute OCV using steps 2 to 4.

2. **Find the Index, \(i\)**
   \[ i = (\text{int})(200 \times x_m[0]) \text{- convert to an integer} \]

3. **Find the difference (\(D\)) and offset (\(S\))**
   \[ D = i - 200x_m[0] \]
   \[ S = 1 - D \]

4. **Compute the OCV using temperature (\(T\))**
   \[ OCV = OCV_0[i] \times S + OCV_0[i + 1] \times D + T(OCV_{ref}[i] \times S + OCV_{ref}[i + 1] \times D) \]

3.4.2.2.2 **Computing unconstrained general optimal solution for HW_v1**

In step 3 of Stage 2 we compute the unconstrained global optimal solution \(\Delta u^o\) for the control input and we complete the remaining computations for \(F\) that are not computed with the \(F\_sub\) Module in Stage 1, which include the final multiplications by \(\chi_k\) and \(u_k\), as well as the final summation terms of equation (3.21). These computations are illustrated in equation (3.55), and are derived from equations (3.41), (3.44), and (3.45).

\[
F = -2(F_1 - (F_{2c})\chi_k - (F_{3a})u_k) \quad (3.55)
\]

For HW_v1, as demonstrated in Figure 18, the Final F Module consists of: a \(VV\) module to compute \((F_{2c})\chi_k\), an adder to sum the terms, a multiplier to compute \((F_{3a})u_k\), -2(sum), and \(\Delta u^o\).
Computing $\gamma$ constraint vector for HW_v1: Next, we compute the $\gamma$ constraint vector. For HW_v1, the internal architecture for $\gamma$ module is depicted in Figure 19. As illustrated, the $\gamma$ module computes the $G_{ffu_k}$ vectors and $\Phi \tilde{\chi}$ vectors in parallel, by employing two VS modules and two MV modules, respectively. Then, two $V+V$ modules are employed to compute the intermediate terms $\Phi_{v} \tilde{\chi} + G_{ffv}mu_k$ and $\Phi_{z} \tilde{\chi} + G_{ffz}mu_k$ in parallel. An adder is utilized to compute the scalar addition. Next, three $V+S$ modules are employed to compute the final terms in $\gamma$ constraint vector in parallel, in order to generate equation (3.25) from Section 3.2.2.

Computing $M\Delta u^o$ for HW_v1: For HW_v1, the $M\Delta u^o$ is designed in such a way to be performed in parallel with $\gamma$. As shown in Figure 20(a), the $M\Delta u^o$ module is a dedicated VS
module, which consists of a single multiplier and a feedback-loop logic to multiply each element of the vector by the scalar.

**Computing K vector for HW_v1:** In HW_v1, the \( K \) vector is computed before the final step 7 (in Stage 2), which is to perform the comparison operation. The \( K \) vector is one of the first operands of Stage 3. The \( K \) vector computation requires a minimum of 32 subtractions. In this case, in order to ensure that \( K \) is ready for Stage 3, \( K \) vector is computed before performing the comparison as presented in equation (3.31), \( M\Delta u^o \leq \gamma \). As illustrated in Figure 20(b), K Module is a simple V-V module, which consists of a subtractor to subtract each element of the input vectors.

**Computing Comparison for HW_v1:** In the final step in Stage 2, for HW_v1, the two vectors \( M\Delta u^o \) and \( \gamma \) and compared element by element using a FPU comparator. The internal architecture of the Comparison module is illustrated in Figure 20(c). In this case, if the constraints are not violated, the Comparison module performs all 32 compare operations, and then goes to Stage 4. However, if a constraint is violated, the Comparison module triggers Stage 3 and relinquishes the execution of remaining compare operations.

Figure 20: a) VS, b) V-V, and c) Comparison of constraints for HW_v1

3.4.2.2.3 Computing Unconstrained Solutions for HW_v2

In Stage 2 for the internal architecture for HW_v2, we use the pipelined MACx module for the matrix and vector multiplication operations. The utilization of the MACx module (for HW_v2) drastically reduces the occupied area on chip for Stage 2 compared to that of HW_v1. For instance,
for the OCV module, HW_v1 uses 20 dedicated IP cores, whereas HW_v2 uses only 8 dedicated IP cores. The space analysis is detailed in Section 3.5.

Figure 21: Functional architecture for stage 2 for HW_v2

As depicted in Figure 21, the internal architecture for HW_v2 consists of the OCV module, a MACx module, AU (arithmetic unit) module for arithmetic operations, and a module to perform additional memory operations not managed by the MACx. In order to minimize the memory access bottleneck due to the limited number of memory ports, as well as to reduce the complexity of the memory controller, we incorporate a FIFO buffer to preload the necessary vectors for the MACx and for the input AU modules, in certain scenarios, where memory ports are not available. In this case, the MACx module and OCV module are executed in parallel. The MACx module performs the following computations in sequence:

1. $F_{2c\chi}$ for $F$ in equation (3.55)
2. $\phi_{x\tilde{A}\chi}$ for $\gamma$ in equation (3.25)
3. $\phi_{y\tilde{A}\chi}$ for $\gamma$ in equation (3.25)

Since the maximum length of the individual vectors is 3, the 5-stage pipelined MACx module uses only the first 3 pipeline stages, reducing the overall execution time.
The Input AU module sends the necessary operands to the AU module, which performs the remaining operations (not performed by MACx) in Stage 2. The Output AU module forwards the results to be stored in the BRAM. With the AU module, multiplication results are generated every clock cycle after an initial latency of 1 clock cycle, and addition/subtraction results are also generated in every clock cycle after an initial latency of 5 clock cycles.

Handshaking protocol is used to communicate between the Input AU and Output AU modules. After completing any intermediate computations, the Output AU module sends a signal to the Input AU module, indicating that the intermediate data (results from previous arithmetic operations) are ready for subsequent arithmetic operations. Utilizing two modules (i.e., Input AU and Output AU) to read from the memory and write to the memory separately, significantly reduces the complexity of the control path for both modules. This also minimizes the setup and hold time violations, thus improving the overall efficiency of Stage 2.

In HW_v2 design, the comparison (final step 7) is performed while computing K, instead of using a separate comparator module as in HW_v1. Considering the equation (2.35), $K = \gamma - M\Delta u^0$, and the comparison equation (2.26) $M\Delta u^0 \leq \gamma$, if $K \geq 0$ then the comparison is true and $\Delta u^0$ is the optimal control signal and Stage 4 is executed bypassing Stage 3. In HW_v2, if one or more elements violate the constraints, then we start executing Stage 3 immediately after performing the K computation. A binary single bit comparison significantly reduces the time taken to perform the compare operations from the HW_v1 approach of utilizing a separate floating point compare module. As illustrated in Figure 21, HW_v2 has an integrated solution for Stage 2, whereas HW_v1 has a modular solution (depicted in Figure 18, Figure 19, and Figure 20).
3.4.2.3 Stage 3: Hildreth’s Quadratic Programming

In Stage 3, we compute the correction factor for the constrained optimal control input using Hildreth’s Quadratic Programming (HQP) approach. With this approach, the $\Delta u^\circ$, which is known as the global optimal solution, is adjusted by $E^{-1}M^T\lambda$ (as in equation (2.42)), where $\lambda$ is a vector of Lagrange multipliers.

For Stage 3, we use the Primal-Dual method for active set approach, which reduces the total constraints down to active constraints (i.e., non-zero $\lambda$ elements), thus reducing the computation complexity (3 or less computations versus 32 computations). Apart from reducing the computation complexity of Stage 3, this approach also reduces the computation complexity of Stage 4, since the Stage 4 design needs to compute only 1 to 3 active elements of the lambda vector versus computing all 32 elements.

Next, we use the HQP technique, which further simplifies the above computations by finding the vector of Lagrange multipliers, $\lambda$, for the HQP solution one element at a time. This HQP technique eliminates the need for matrix inversion in optimization. In this case, the $\lambda$ vector has either positive non-zero values for active constraints, or zero values for inactive constraints.

Typically, not all the constraints are active at the same time, making $\lambda$ a sparse vector. Since only the active constraints need to be considered, both hardware versions are designed in such a way to execute the sparse vector to reduce the total computations involved for the operation.

It should be noted that the HQP technique does not always converge. Therefore, a suitable iteration length (number of iterations) is selected, to provide the greatest possibility for convergence, as well as to provide a reasonable sub-optimal solution when lambda does not converge.

As implied above, HQP is an iterative process. This is typically implemented as two nested loops. The inner loop computes the individual elements of the $\lambda$ vector, in which, the number of iterations depends on the length of $\lambda$. The outer loop determines whether the $\lambda$ vector converges.
The outer loop executes until the λ vector converges or until the maximum number of iterations (in our case, 40 iterations) are reached. The functional flow of Stage 3 is as follows.

1. Compute individual elements of λ vector from equations (2.39) and (2.40).
2. Determine whether the λ vector meets the convergence criteria.
3. If it does, compute the new Δu using the updated λ vector, else go to step 1.

For both hardware versions (HW_v1 and HW_v2), we decompose Stage 3 into the above three main modules, illustrated in Steps 1 to 3. Firstly, the λ module (Wp3) computes the first λ vector. Secondly, the convergence module (Converge_v1) determines whether the current λ vector converges or not; simultaneously, the λ module computes the next λ vector. If the current λ vector converges then the λ module stops the execution of the next λ vector. In this case, the λ module performs the computations of equations (2.39) and (2.40) on each element.

The HQP technique, which includes these two equations, (for both HW_v1 and HW_v2) is illustrated in the algorithm (in
Table 4). Since $E^{-1}M^T$ is computed in Stage 1, it is reused in Stage 3, instead of re-computing in each iteration. The elements of the $\lambda$ vector are calculated using the $P$ matrix from Stage 1 and $K$ vector from Stage 2.
Table 4: HQP algorithm

<table>
<thead>
<tr>
<th>Hildreth’s Quadratic Programming Technique (HQP algorithm)</th>
</tr>
</thead>
</table>

for loop = 0; loop < 40;
1. Calculate λ
   \[ \lambda_{previous} \leftarrow \lambda_{current} \]
   for i = 0; i < M_{size};
     for j = 0; j < M_{size};
       \[ w = w + P_{i,j} \lambda_{j} \]
     end
   \[ w = w + K_{i} - P_{i,i} \lambda_{i} \]
   \[ \lambda_{test} = -w/P_{i,i} \]
   \[ \lambda_{i} = \max \{0, \lambda_{test}\} \]
   end
2. Check convergence
   calculate the Euclidean length of \( \lambda_{previous} \)
   calculate the Euclidean length of \( \lambda_{current} \)
   if \( \| \lambda_{current} \| / \| \lambda_{previous} \| < \) reference, then calculate \( \Delta u_{corr} \)
   else repeat \( \lambda \).
3. Calculate correction factor \( \Delta u_{corr} \)
   for j = 0; j < M_{size};
   1. \[ \Delta u_{corr} = \Delta u_{corr} + M_{j}^{T} \lambda_{j} \]
   end
   2. \[ \Delta u_{k+1} = \Delta u^{\circ} - \Delta u_{corr}/E \]
   end loop

3.4.2.3.1 For HW_v1

HW_v1 consists of three main modules, including \textbf{Wp3}, \textbf{Converge_v1} and \textbf{New_Au_v1}, and a sub-module (\textbf{SVM_v1}) for sparse vector multiplication.

From our experimental results (presented in Section 3.5), it is observed that any \( \lambda \) vectors typically have a maximum number of 3 non-zero elements. Hence, our hardware is designed to operate only on the non-zero elements of \( \lambda \) and \( P \). In order to generate all the elements of the \( \lambda \) vector, the computations for calculating \( \lambda \) (section 1 in
Table 4) must be repeated 32 times. By focusing only on the active, non-zero elements, our hardware design dramatically reduces the time taken to generate the active $\lambda$ elements, since certain steps are by-passed in
The functional flow of the sparse vector multiplication module (SVM_v1) is illustrated in Figure 22(a). As demonstrated, SVM_v1 Module checks each element of the input vectors, and only forwards the non-zero elements to the MAC unit. As depicted in Figure 22(b), Wp3 Module (λ module) employs SVM_v1 to compute sub-step 2.b of the HQP algorithm (in
Table 4). The Wp3 Module also consists of other modules including a multiplier and adder, to compute the remaining sub-steps, i.e., 2.c to 2.f of the HQP algorithm.

In this case, the $\lambda$ vector is updated, after 32 iterations. Then the updated $\lambda$ vector is forwarded to the convergence module (Converge_v1). The Converge_v1 Module computes step 2 of the HQP algorithm (in
Table 4; simultaneously the New_Au_v1 Module computes step 3 of the HQP algorithm (to generate $\Delta u_{k+1}$) in anticipation of a convergence. At the same time, the $\lambda$ module (Wp3) starts computing the next $\lambda$ vector in the event the current $\lambda$ does not converge. If the convergence fails, the $\Delta u_{k+1}$ value is discarded. If the convergence succeeds, a signal is sent to Wp3 Module to terminate the next $\lambda$ vector computation, and then the subsequent stage (Stage 4) is started with input $\Delta u_{k+1}$.

Figure 22: Internal architecture for SVM_v1 and Wp3 modules for HW_v1

As shown in Figure 23(a), the Converge_v1 Module consists of the SVM_v1 Module, an adder, a multiplier, a square-root, and an inverse square-root. It also consists of a comparator to compare the ratio value to a reference value to determine the convergence. The internal architecture for the New_Au_v1 Module is depicted in Figure 23(b). As depicted, the New_Au_v1 Module computes the $\Delta u_{k+1}$ value (step 3 of HQP algorithm) and consists of a SVM_v1 module and a subtractor. In this case, the New_Au_v1 Module is executed in parallel with the Converge_v1 Module.
3.4.2.3.2 For HW_v2

The high-level architecture for HW_v2 for Stage 3 is illustrated in Figure 24. Apart from the fundamental operators, this consists of five custom modules.

For HW_v2, Win Module computes the equations (2.39) and (2.40), (i.e., computes sub-step 1 of the HQP Algorithm (}
Table 4). Also, **Win Module** acts as an interface/control module, and interfaces with the memory and drives the inputs for other modules. The functional/data flow of the **Win Module** is shown in Figure 25. In this case the FPU multiplier, adder, and subtractor are external to the **Win module** as illustrated in Figure 24.

![Internal architecture for Win module for HW_v2](image)

**Figure 25: Internal architecture for Win module for HW_v2**

For **HW_v2**, similar to **HW_v1**, we introduce another sparse vector multiplication (**SVM_v2**) module, in order to utilize only the active set (or non-zero values of the \( \lambda \) vector), thus enhancing the efficiency of the design. This is because the pipelined MACx is not efficient for single vector multiplication operations. In **Win Module**, addressing logic is incorporated to track the non-zero elements of the \( \lambda \) vector. These non-zero \( \lambda \) elements and the corresponding indexes are stored as vectors in the BRAMs. The indexes are used to find the corresponding \( P \) and \( ME^{-1} \) values, thus reducing the number of operations without compromising the accuracy of these values. In this case, the number of operations is reduced from 32 to 3 or less.

The internal architecture of the **SVM_v2 Module** is demonstrated in Figure 26(a). Although the multiplier is external to this module, it is included in dotted lines (in Figure 26(a)) to facilitate our discussion below. As illustrated, our **SVM_v2 Module** utilizes a counter to determine the number of accumulation loops, instead of using the length of the vector. The **SVM_v2 Module** employs an adder, a FIFO buffer, and a multiplier (external) to perform the necessary operations. This module can execute a vector of any length, in this case, up to the maximum number of counts, depending on the size of the counter.
First, the **Win Module** sends the vector elements to the multiplier and signals the **SVM_v2 Module** that the sparse vector operation is initiated. Next, if the results are valid, the valid signal is asserted to start incrementing the counter and to start loading the results to the FIFO buffer. In this case, the counter is incremented if the multiplier valid signal is asserted (high), the counter is decremented if the adder valid signal is asserted; and the counter is on hold if both the valid signals are asserted or de-asserted (low) simultaneously. The FIFO buffer is used to bridge the latency between the multiplier and the adder. If the count is 1, the **SVM_v2 Module** forwards the multiplication results to the output, by-passing the adder.

The internal architecture of the convergence module (**Converge_v2 Module**) is shown in Figure 26(b). To determine the convergence of the \( \lambda \) vector, the Euclidean Distance is computed. In HW_v2, the squared terms of the Euclidean Distance are calculated as each element of the \( \lambda \) is computed, one element at a time. Conversely, in HW_v1, this distance is measured sequentially after all 32 elements are computed. In HW_v2, the **\( \lambda \)-norm** module (in Figure 26(b)) takes the scalar \( \lambda_i \) as inputs, squares the \( \lambda_i \), and then adds the squared value to the previous element. After computing the final \( \lambda \) element, which is \( \lambda_{31} \), the output of **\( \lambda \)-norm** is then forwarded to a square-root module. The result from the square-root module is the Euclidean Distance. The value is used
twice, as both the current lambda norm for the current iteration and the previous lambda norm for
the next iteration, reducing the number of square root operations.

In this case, the Win Module sends $\lambda$, to the multiplier, and signals the $\lambda$-norm module that
the required data is ready. Next, the $\lambda$-norm module waits until the multiplier valid signal is
asserted, then accumulates the outputs using an adder. After Win Module informs that the iterations
for $\lambda$ are completed, the final accumulator result of the $\lambda$-norm module is sent to the FPU square-
root module to initiate the execution of the Converge_v2 Module. The Converge_v2 Module
typically waits for the square-root valid signal to be asserted. During this time, the Converge_v2
Module inverts the previous $\lambda$ length value using a divider. The internal architecture of the
convergence module (Converge_v2 Module) is shown in Figure 26(b).

The entire process is repeated up to 40 times. The system either converges naturally or,
after the 40th iteration, is considered to be converged. Next, we start executing the New_Au_v2
Module. In this case, the Win Module loads the $\lambda$ and the $E^{-1}M^T$ values into the multiplier for the
SVM_v2 Module to process, and sends a signal to the New_Au_v2 Module to initiate the execution.
Depending on the length of the active set (non-zero elements) in the $\lambda$ vector, the New_Au_v2
Module selects either the output of the multiplier or the output of SVM_v2 to be the input to its
subtractor. The result of the subtraction is $\Delta u_{k+1}$ value, which is forwarded to Stage 4 for processing.

Finally, in Stage 3, a clear operation is performed to clear the FIFO, which occurs at the
end of vector multiplication by SVM_v2 Module. This ensures that invalid data is not incorporated
in any computations. The clear operation takes four clock cycles and asserts a ready signal to
indicate that the result of the SVM_v2 Module is ready for the next operation. The ready signal
also indicates the SVM_v2 Module is free to address the next computation.

3.4.2.4 Stage 4: State and Plant

In Stage 4, we compute and update the plant state and the plant outputs, using the new $\Delta u$
(computed in Stages 2 or 3), and utilizing $\chi$, which contains the current states and the current control
signal $u$. In a real-world scenario, the plant outputs are measured, and the control signals are sent to the plant input or actuators.

The updated plant states and the input control signals are forwarded to Stage 2 for the next iteration. Prior to starting the next iteration, the Top-Level Module (in Section 3.4.2) determines whether the plant state value ($x_m^0$) is fully charged or whether we have reached the maximum number of iterations.

During Stage 4, we compute the plant output, which determines the current terminal voltage ($v_k$), and then the state of charge ($z_k$) from equations (3.3) and (3.4), respectively. Then the control signal and the state signals are updated. In this case, the first element of the $\Delta U_k$ is used to update the control signal from equation (3.26), and the new control signal is used to determine the states for the next iteration from equation (3.27).

### 3.4.2.4.1 For HW_v1

The overview of Stage 4 for HW_v1 is depicted in Figure 27. As illustrated, in HW_v1, the plant outputs and next states are computed in parallel, since these computations are independent of each other. Conversely, in HW_v2, these computations are performed in sequence, but in highly efficient fashion to reduce the performance bottleneck.
In this case, as shown in Figure 27, for HW_v1, the voltage $v_k$ and the state of charge $z_k$ are computed in the Plant Module, and the control signal $u_{k+1}$ and the states $x_{k+1}$ are computed in the State Module. The Plant and State Modules are executed in parallel.

As demonstrated in Figure 28(a), for HW_v1, the Plant Module consists of a customized module to perform the two $\tilde{C}_v x_k$ computations in sequence to save occupied area on chip. Since these computations are performed on small vectors, the execution overhead due to sequential operations is negligible. In this case, first, the $\tilde{C}_v x_k$ is performed; second, the $\tilde{C}_z x_k$ is performed, simultaneously adding OCV to $\tilde{C}_v x_k$. As illustrated in Figure 28(b), for HW_v1, the State Module reuses the MV, VS, and V+V Modules from previous stages to perform various vector/matrix operations. It also consists of an adder module to compute the control signal $u_{k+1}$. 

Figure 27: Top-level architecture for stage 4 for HW_v1
3.4.2.4.2 For HW_v2

The internal architecture of Stage 4 for HW_v2 is depicted in Figure 29. As illustrated, HW_v2 consists of four major modules: the Input Module, SVM_v2 Module, SVM_store Module, and an Output Module. In this case, the Output Module computes and updates the plant state, the plant outputs, and the control signals.
necessary data and control signals to the multiplier and the adder to perform the computations. The **Input** Module sends data ready signals to the **SVM_v2** Module and to the **Output** Module. Handshaking protocol is used to communicate/control among the modules. The **SVM_store** Module and the **Output** Module are executed in parallel, and the intermediate and final results are stored in the two BRAMs. In Stage 4, for **HW_v2**, we carefully arrange the computations in the following sequence (from step 1 to step 14), to obtain the results with the least amount of time.

(1) \( C_v x_k \rightarrow \text{by SVM_v2 Module} \)

(2) \( D_v u_k \rightarrow \text{by multiplier} \)

(3) \( C_z x_k \rightarrow \text{by SVM_v2 Module} \)

(4) \( D_z u_k \rightarrow \text{by multiplier} \)

(5) \( A_{m_{row0}} x_k \rightarrow \text{by SVM_v2 Module} \)

(6) \( A_{m_{row1}} x_k \rightarrow \text{by SVM_v2 Module} \)

(7) \( u_k + \Delta u_{k+1} \rightarrow \text{by adder} \)

(8) \( C_v x_k + D_v u_k \rightarrow \text{by adder} \)

(9) \( C_z x_k + D_z u_k \rightarrow \text{by adder} \)

(10) \( B_{m0} u_{k+1} \rightarrow \text{by multiplier} \)

(11) \( B_{m1} u_{k+1} \rightarrow \text{by multiplier} \)

(12) \( C_v x_k + D_v u_k + OCV(z_k) \rightarrow \text{by adder} \)

(13) \( A_{m_{row0}} x_k + B_{m0} u_{k+1} \rightarrow \text{by adder} \)

(14) \( A_{m_{row1}} x_k + B_{m1} u_{k+1} \rightarrow \text{by adder} \)

With the above arrangement, we manage to overlap the **SVM_v2** Module computations with the multiplier/adder computations; thus, reducing the overall execution time for Stage 4. In this case, the multiplier and adder modules produce results every clock cycle, and these results are forwarded to the **Output** Module to be stored in a BRAM. Conversely, the time taken for the **SVM_v2** Module to produce results varies and often depends on the length of the input vectors, and these results are forwarded to the **SVM_store** Module to be stored in a BRAM. Hence, the final result of step 2 is available (in BRAM) before the final result of step 1 is available (in BRAM). This concurrent execution of operations significantly reduces the performance bottleneck in Stage 4. For **HW_v2** in Stage 4, we reuse the **SVM_v2** Module from Stage 3. The adder and multiplier IP cores are also reused in other stages to reduce the overall space occupied on chip.
After Stage 4 computations are completed, we start computing Stage 2. In Stage 2, the updated state of charge (SOC) value is compared with the reference value to determine whether the battery is fully charged. The MPC algorithm iterates through Stages 2, 3, and 4, until the battery reaches its fully charged condition.

3.5 Experimental Results and Analysis

We perform experiments to evaluate the feasibility and efficiency of our proposed embedded hardware and software architectures for the fast-charge Model Predictive Controller (MPC). We also compare our proposed embedded architectures with the baseline model of the fast-charge MPC written in Matlab [43], in order to evaluate and validate the correctness and functionalities of our designs. The evaluation setup for our embedded designs is based on real implementations, whereas the evaluation setup for the baseline Matlab model is based on simulation. Our embedded hardware and software results are obtained in real-time, while these designs are actually running on the Virtex-6 chip. Conversely, the baseline Matlab results are obtained through the simulation on a desktop computer. Apart from embedded designs, our software design written in C is also executed on a desktop computer and the corresponding results are compared with the baseline Matlab results. All our experiments are performed with a sample time of 1 second, temperature at 25°C, and iterations of 3600.

3.5.1 Functional Verification – Comparison with Baseline Model

It is imperative to ensure that our embedded hardware and software architectures operate correctly; hence, we compare our proposed embedded architectures with the baseline model written in Matlab [43].

As stated in [43] it is necessary to determine the applied current profile that drives the state of charge (SOC) to the desired reference value, while ensuring that the terminal voltage does not exceed its operational constraints. The convention used for the current for the batteries is: negative if charging current, and positive if discharging current. Since this is a fast-charge model, the values
of the current are all negative. Figure 30 illustrates the desired charging profile for the fast-charge MPC implementation in [43]. In this case, the battery cell is considered to be completely empty. The charging profile is the standard constant current (CC) and the constant voltage (CV). The current is constant until the voltage reaches its maximum, and then the voltage is constant. As in Figure 30, initially, the current starts out at its maximum value, and stays or is held constant until the terminal voltage reaches its allowed maximum. Once the terminal voltage reaches its maximum value, the voltage is held constant while the current starts to decay towards zero. The current continues to decay until the SOC reaches its full charge (in our case, this is at least 90% of capacity). Once the battery reaches its full charge, the current goes to zero and the terminal voltage returns to its No Load state. The intention of the experiments (in this sub-section) is to ensure that the charging profiles for the embedded hardware and software architectures are identical to that of the baseline Matlab implementation in [43].

As discussed earlier, the MPC algorithm consists of three main elements, i.e., state of charge (SOC), terminal voltage, and battery cell current ($I_{cell}$). We perform experiments to verify and evaluate the functionalities of these three main elements for our embedded architectures. The results are obtained and presented with Figure 31(a), Figure 32(a), and Figure 33(a) respectively. As illustrated in these three figures, the charging profiles of our embedded hardware and software
architectures are almost identical to that of the baseline Matlab in [43], since the graphs are overlapping. There are some slight discrepancies, which are negligible.

Figure 31(a) and Figure 31(b) depict the SOC of the battery as a percentage. As illustrated in these graphs, our embedded hardware architectures (HW_v1 and HW_v2) and our embedded software architecture show similar behavior as that of the baseline Matlab for the SOC.

![Cell state of charge](image)

**Figure 31: State of Charge comparison: HW and SW designs vs baseline Matlab Model**

Although at a glance, the SOC graphs (Figure 31(a)) seem identical for all four designs, at a closer look, there are some discrepancies. As illustrated in Figure 31(b), the SOC increases sharply with the embedded systems designs, whereas SOC increases gradually with the baseline Matlab design. In this case, both designs reach full charge before the expected time of 1216s, which is determined from the baseline experiments.

Figure 32(a) and Figure 32(b) depict the terminal voltage of the battery. As illustrated in these graphs, our embedded hardware architectures (HW_v1 and HW_v2) and our embedded software architecture show similar behavior as that of the baseline Matlab design for the terminal voltage. As demonstrated in Figure 32(b), the output voltage does not exceed 4.2V; this illustrates that the system’s behavior respects the constraints intended to extend the useful-life of the battery.
Similar to SOC graphs, at a glance, the terminal voltage graphs (Figure 32(a)) seem identical for all four designs; at a closer look (Figure 32(b)), there are some discrepancies. For instance, at time t=0s, the initial Cell Terminal Voltage value for the embedded systems designs is 3.92V, whereas for the baseline Matlab design is 4.11V. Further experiments and analysis confirm that this discrepancy does not affect the overall functionalities of the system or the final outcome of the MPC algorithm.

As illustrated in Figure 32(b), the Cell Terminal voltage increases gradually and smoothly with the baseline Matlab design, whereas the Cell Terminal voltage increases sharply in the beginning and then decreases gradually with the embedded systems designs. In this case, the difference in value between the above two is merely 1.2mV.

Figure 33(a) and Figure 33(b) depict the control signal, i.e., the current, generated by the designs that drive the terminal voltage and the SOC responses. As illustrated in these graphs, our embedded hardware architectures (HW_v1 and HW_v2) and our embedded software architecture show similar behavior as that of the baseline Matlab design for the control signal ($I_{cell}$). In this case, a negative value for the current means that the current is flowing into and charging the battery, rather than flowing out of the battery and being used in the system.
Figure 33: Control signal ($I_{cell}$) comparison: HW and SW designs vs baseline Matlab model

The current starts out at a constant, with a maximum allowed value of (-15A). The negative value indicates that current is charging the battery instead of powering the system. The current starts to gradually decay to zero once the terminal voltage reaches its maximum voltage, and then the current is held constant. The current shows a steep decay at around 1200s, which is when the SOC is 90% and the battery is considered fully charged.

Similar to terminal voltage graphs, at a glance, the control signal graphs (Figure 33(a)) seem identical for all four designs; at a closer look (Figure 33(b), there are some discrepancies. As illustrated in Figure 33(b), the discrepancies are prominent between the timeline 1090s and 1120s. However, these discrepancies are negligible as they do not affect the overall functionality and the final results of the designs.

**Summary:** From these results and analysis, we can conclude that our embedded designs show similar behaviors and functionalities as that of the baseline Matlab Model, thus confirming the correctness and functionality of our designs. There are some slight discrepancies in the order of milli-volts for the voltage and milli-amps for the current. These slight discrepancies are mainly because we use single-precision floating-point-units for our embedded hardware and software architectures, whereas a baseline Matlab model was created using double-precision floating-point units. In addition, we use different techniques to solve the linear algebra equations, instead of the
existing techniques used in the baseline model, which might further contribute to these discrepancies. Further experiments and analysis reveal that these discrepancies are too small to have an impact on the overall functionalities and the performance of the fast-charge MPC.

3.5.2 Performance Metrics – Execution Time and Resource Utilization

We perform experiments to evaluate the feasibility and efficiency of our embedded hardware and software architectures in terms of speed-performance and resource utilization on chip.

3.5.2.1 Execution Times and Speedup: Embedded HW vs. SW on MicroBlaze and Intel i7

The total time taken to execute the fast-charge MPC algorithm for the two embedded hardware designs and the embedded software design is presented in Table 5. The execution time for each design is measured 10 times, and the average is presented. In this case, embedded hardware architectures are executed on the Virtex-6 FPGA running at 100MHz, whereas embedded software architecture is executed on the MicroBlaze processor running at 100MHz on the same FPGA for fair comparison purposed. The total time is considered as the time taken to execute the fast-charge MPC algorithm for a specific number of iterations (in our case, 3600 iterations) for all 3 embedded systems designs in Table 5.

The total time taken to execute the baseline Matlab design is also presented in Table 5. The execution time for the Matlab model is also measured 10 times, and the average is presented. The baseline design is executed on an Intel i7 processor running at 3.1GHz on a desktop computer.

Table 5: Execution times: embedded HW and SW designs and baseline Matlab model

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Execution time/ms</th>
<th>Speedup over embedded SW</th>
<th>Speedup over baseline Matlab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded SW on MicroBlaze (at 100MHz)</td>
<td>3958.04</td>
<td>-</td>
<td>0.21</td>
</tr>
<tr>
<td>Baseline Matlab on i7 processor at (3.1GHz)</td>
<td>848.331</td>
<td>4.67</td>
<td>-</td>
</tr>
<tr>
<td>HW_v1 (at 100 MHz)</td>
<td>468.557</td>
<td>8.45</td>
<td>1.81</td>
</tr>
<tr>
<td>HW_v2 (at 100 MHz)</td>
<td>39.774</td>
<td>99.51</td>
<td>21.33</td>
</tr>
</tbody>
</table>
From Table 5, considering the total execution time, our embedded hardware version 2 (HW_v2) is almost 100 times faster, and our embedded hardware version 1 (HW_v1) is almost 9 times faster than the equivalent software (SW) running on the embedded MicroBlaze processor. Furthermore, our HW_v2 is 21 times faster, and our HW_v1 is almost 2 times faster than the baseline Matlab model running on an Intel i7 processor. It should be noted that all our embedded systems designs are running at 100MHz, whereas the Matlab model is running at 3.1GHz.

Unlike the embedded hardware and software designs, the Matlab model is designed in such a way, so that it terminates the execution of Stages 2 and 3 once the system meets the threshold for the fully charged. Next, the Matlab model only executes Stage 4 for the remainder of the MPC computation. Hence, the total time obtained for the Matlab model (presented in Table 5) is not the time taken to execute the fast-charge MPC for 3600 iterations, but much less than that. As a result, it is difficult to make a direct execution time comparison between the baseline Matlab model and the embedded systems designs. However, as illustrated in Table 5, our embedded hardware designs still achieve better speedup compared to the Matlab model running on a high-performance processor. With these speedups, our proposed hardware designs should be able to monitor and control multiple battery cells individually.

From the above results and analysis, it is observed that our register-based HW_v1 is much slower than the BRAM-based HW_v2. Typically, the register-based designs should provide better computing power compared to the memory-based designs, since there is an execution overhead associated with reading/writing from/to the on-chip memory in the latter. In this case, the read operation, and the write operation from/to on-chip memory take one clock cycle each. However, our memory-based HW_v2 design achieves higher speed-performance. This is mainly because our initial experience gained throughout the design and development of HW_v1, enable us to enhance the efficiency of HW_v2. Furthermore, the speed-performance is also impacted by the compact nature and area-efficiency of the memory-based design, as discussed in the following sub-section.
3.5.2.2 Resource Utilization: Register-Based HW_v1 vs. BRAM-Based HW_v2

The cost analysis on space is carried out on our two embedded hardware versions to examine the area-efficiency of our hardware designs. The resource utilization for register-based HW_v1 and BRAM-based HW_v2 is presented in Table 6. As illustrated, the total number of occupied slices, the total number of BRAMs, and the total number of DSP slices required for HW_v1 are 34,193, 62, and 688, respectively. Conversely, the total number of occupied slices, the total number of BRAMs, and the total number of DSP slices required for HW_v2 are 10,277, 35, and 73, respectively.

Table 6: Resource utilization: embedded HW_v2 versus embedded HW_v1

<table>
<thead>
<tr>
<th>Configuration</th>
<th># of occupied slices</th>
<th># of BRAMs (36E1)</th>
<th># of DSP48E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW_v1 (at 100 MHz)</td>
<td>34,315</td>
<td>62</td>
<td>688</td>
</tr>
<tr>
<td>HW_v2 (at 100 MHz)</td>
<td>10,277</td>
<td>35</td>
<td>73</td>
</tr>
</tbody>
</table>

As observed from Table 6, with the BRAM-based HW_v2, we achieve 70% of space saving in terms of total number of occupied slices, and 89% space saving in terms of total number of DSP slices, compared to the register-based HW_v1. Furthermore, we also achieve 44% space saving in terms of total number of BRAMs, with HW_v2 compared to HW_v1, which is unexpected, since it is assumed that the BRAM-based designs naturally would utilize more BRAMs than the register-based designs.

From the above results and analyses, it is evident that the BRAM-based HW_v2 is significantly more area efficient than the register-based HW_v1; hence the former is more suitable for embedded devices due the stringent area constraints of these devices.

3.5.2.3 Analysis of Iteration Time per Cycle for BRAM-Based HW_v2

We analyze the per iteration time only for our BRAM-based HW_v2, since this hardware version is superior to HW_v1, the embedded software design, and the baseline Matlab model, in terms of speed and area.
It should be noted that each of the embedded systems designs performs well within the required 1 second (s) sampling time (or interval) required for the fast-charge MPC algorithm. For instance, based on our experimental results, for HW_v2, the iteration time per cycle varies from 5.2μs to 450μs (4.5x10^{-4} s) as shown in Figure 34(a). The maximum iteration time per cycle, which is 450μs, is illustrated in Figure 34(b). This maximum iteration time incurs when the Hildreth’s Quadratic Programming (HQP) technique fails to converge within 40 iterations, leading to sub-optimal results being employed. This still leaves a significant margin of 0.9995s in the 1s sampling cycle.

In this case, the execution overhead of the Augmented Model in Stage 1 is approximately 24μs and considered to be minimal. This execution overhead is the time difference between the first iteration (which includes the processing time through stages 1 to 4 during an iteration, as illustrated in Figure 34(b)) and the second iteration (which includes the processing time through stages 2 to 4 during an iteration, as illustrated in Figure 34(c)). For both the first and the second iterations, Stage 3 (for HQP) converges in 2 loops, which takes approximately 20.5μs. Hence, we logically deduce that the difference in execution time between the iteration 1 and 2 is the time taken for Stage 1 to complete. In this case, Stages 2 and 4 require 5.18μs to process (as in Figure 34(d)), thus, leaving the remainder of the time for Stage 3. The time to process Stage 3 depends on two factors: the number of non-zero λ elements and the number of iterations required for convergence.

For our proposed embedded HW_v2 for the fast-charge MPC algorithm, the processing time for Stage 3 typically varies from 15.3μs to 444.5μs. The minimum time (15.3μs) is associated with 1λ element and 2 iterations; and the maximum time (444.5μs) is associated with 2λ elements and 40 iterations. In the worst-case scenario, by assuming that the first iteration does not converge, the worst-case iteration time is 474μs (i.e., adding 24μs to 450μs). In this case, the fast-charge MPC algorithm could execute more than 2100 times, in a 1 second sample time; allowing our embedded architecture to control multiple battery cells individually.
3.5.3 Analysis of Existing Works on Embedded Designs for MPC

Earlier we discussed and analyzed the existing research work on embedded architectures for MPC algorithm. From this investigation, it was evident that similar work does not exist, specifically for the fast-charge MPC algorithm. Therefore, it was difficult to make a fair comparison between the algorithms. However, we extended our investigation and selected a few existing works that had slightly closer traits to our proposed embedded designs. These designs are discussed and analyzed as follows: A closely related work was presented in [58], which proposed a hardware-software co-design design for MPC. This design comprised a microprocessor and a matrix co-processor. The design utilized a Logarithmic Number System (LNS) instead of a floating-point, and Newton’s algorithm instead of HQP, as in our design. Unlike our design, in [58], the model parameters were pre-calculated off-line and stored in the microprocessor. In [33], an MPC-dedicated processor was proposed, which utilized a mix of fixed-point and floating-point
numbers. Similar to our design, this design also utilized the HQP technique, but with Laguerre functions. The processor was designed using Matlab and evaluated using Simulink, however, no actual hardware architecture was implemented. In [30], a fixed-point MPC solution was proposed with two separate QP solvers as user-designed modules: Primary Dual Interior Point QP for sparse matrices and Fast Gradient QP for dense matrices. Unlike our design, this design utilized the MicroBlaze processor to handle all communication and control of the two user-designed modules. Furthermore, most of the existing designs had different control horizons and prediction horizons, which significantly impacts the total execution time of the MPC algorithm. Also, all the above designs were implemented on different platforms, affecting the resource utilization. The above facts made it difficult to perform a direct comparison between the algorithms in terms of speed and space.

In addition, it is evident that our architectures are the only embedded designs in the published literature that support a non-zero feed-through term for instantaneous feedback.

### 3.6 Conclusion

In this chapter, we introduced unique, novel, and efficient embedded hardware and software architectures for the fast-charge Model Predictive Control (MPC) for battery cell management in electric vehicles based on the ECM model. Our embedded hardware and software architectures are generic and parameterized. Hence, with minor changes to the internal architectures, our embedded designs can be adapted for other control systems applications that employ similar MPC algorithms.

Our BRAM-based HW_v2 achieved superior speedup (100 times faster than its software-counterpart) and our register-based HW_v1 also achieved substantial speedup (9 times faster than the equivalent software). Furthermore, our BRAM-based HW_v2 achieved significant space saving compared to our register-based HW_v1. In this case, 70% of space saving in terms of total number of occupied slices. Thus, it is important to consider the speed-space tradeoffs, especially in embedded devices due to their limited hardware footprints. These two unique embedded hardware
versions can be used in different scenarios, depending on the requirements of the applications and the available resources of the embedded platforms.

Our novel and unique embedded software architecture is also created to be lean, compact, and simple; thus, it fits into the available program memory (in this case 128Kb) with the embedded processor, without affecting the basic structure and the functionalities of the algorithm. We could potentially reduce the program memory usage significantly by constraining the flexibility of the embedded software design. This would allow the embedded processor to incorporate other functionalities and algorithms, if necessary.

Due to the superior speedup, our embedded hardware architecture HW_v2, as a single processing unit, could potentially monitor and control multiple battery cells, while treating each battery cell individually. Considering a typical battery pack made up of 84 cells, our single embedded hardware processing unit could easily execute the fast-charge MPC algorithm for all the 84 cells with the required 1s sample time, since the worst-case iteration time per cycle is a mere 474μs.

The work in this chapter has been published in the EURASIP Journal on Embedded Systems, 2018 [46].

While the ECM model is very well suited to a fast-charge application where the empirical parameters can be determined to closely match real-life and are well understood, ECM is not as well suited towards on the road driving patterns where the conditions change unpredictably. Also, the ECM model does not allow the internal electrochemical reactions to be considered. These reactions form the basis for aging mechanism and performance degradation. Having the ability to adjust the control signal based on these reactions could potentially extend the life of the battery and allow the BMS to increase the performance of the battery. The physics-based model (PBM) is mathematically based and, unlike empirical models, is very adaptable to unknown conditions. Conversely, it is computationally intensive. However, the potential benefit to the longevity and performance of battery cells is worth investigating the possibility of an embedded implementation.
[43],[6],[7] have produced a simplified reduced-order physics-based model that approaches ECM simplicity that shows great promise. In order to implement an MPC using this model, an observer that is able to estimate/observe the internal state of the battery is required as the internal reactions are not easily measurable without invasive techniques. Chapter 4 discusses the embedded architecture for such an observer.
Chapter 4

EXTENDED KALMAN FILTER FOR THE REDUCED-ORDER PBM

4.1 Architecture Overview

In [11] the authors have developed a reduced-order state-space physics-based battery model based on [7],[8],[9], that approaches the simplicity of the ECM and is suitable for an embedded architecture. The physics-based model of the battery provides valuable insight into the factors that affect the performance, the SOC and the life of the battery. ECM models are widely used in industry due to their simplicity, however they cannot provide insight into the internal chemical processes of the battery that affect performance and aging. Insight into these processes allow the safety margins of the batteries to be more accurately placed, increasing the useable capacity of the battery while ensuring that the physical processes that cause aging are respected, preserving the life of the battery. Recent work done by Florentino [11] describes a reduced-order state-space physics-based battery model that approaches the simplicity of the ECM. In this Chapter, our main objective is to provide an efficient FPGA-based hardware accelerator for the physics-based EKF observer for predicting the state of the battery cell. This proposed accelerator would ultimately be utilized in a physics-based MPC. With our proposed hardware accelerators, we strive to retain the accuracy and the insight provided by the PBM model, while reducing the design/computational complexity.

The EKF presented here uses this state-space PBM model to provide an observer for lithium plating, a main aging mechanism, by estimating the state of the contributing physical properties and the SOC. The EKF is one of the non-linear adaptations of the Kalman filter that linearizes the model around the current state [10]. In [7] the authors investigate an interacting-
multiple-model (IMM) Kalman filter framework to select the model that best reflects the current operating conditions of a battery cell. They use a combination of the sigma point Kalman filter (SPKF) and the central difference Kalman filter (CDKF) to compensate for the EKF tendency to produce error covariance estimates that are too small. In [7] the authors goal is to use the IMM to select the most appropriate PBM model (currently defined by SOC and temperature set-points) that matches the current battery performance, with the intent to use that model in the BMS, thus improving performance. Each model pair contains roughly 50KB of data for the lookup tables and various constants and uses over a third of the memory of a 128Kb microprocessor. We adopt the model corresponding to the two standard temperatures 0°C and 25° and a SOC of 60% and later propose a method to switch between models based on temperature or based on SOC that applies to FPGAs. The approach in [7] could result in better model selection as it is also based on current and voltage measured from the battery cell but will likely require much larger microcontroller capacity.

For the PBM we are using in the EKF, the input is battery current, the outputs are the battery voltage, the SOC, and the debiased variables discussed in chapter 2. The state space matrices are developed using look up tables calculated off-line. Throughout the design is the use of two-dimensional interpolation developed by [11] as the lookup tables require both temperature and SOC as inputs.

The main process of an EKF follows that of the Kalman Filter shown in Figure 35 with the addition of the non-linear corrections required by the PBM described in Chapter 2.
Figure 35: Kalman Filter Process

A general explanation of the EKF process can be found in [10], an EKF for ECM is found in [8], and a detailed explanation of EKF for this particular model is found in [11].

We initially investigate the functional flow of the EKF algorithm for the physics-based model [11], and then decompose the algorithm into seven stages to simplify the design process. As illustrated in Figure 36, these seven stages are built with only five main modules. The associated EKF equations of these hardware modules are detailed below.

Figure 36: Physics-based EKF hardware flow

The steps of the extended Kalman filter are discussed in more detail as we move through the description of the hardware design.

4.1.1 Module 1

In Stage 1, Module 1 consists of a finite state machine (FSM) that retrieves relevant constants from the read only memories (ROMs) and then writes these into the designated memory.
locations in the random-access memories (RAMs). In this case, ROMs are preloaded with a text file comprising the constants. Utilizing a text file enables the designers to easily modify the constants (i.e., initialization parameters of EKF), without changing the underlying hardware architecture. This module is executed only once during the initialization. After the initialization process, $x_k$, SOC, and $\sigma_x$ are updated continuously by the feedback loop of EKF (in Figure 36: Physics-based EKF hardware flow).

4.1.2 Module 2

In Stage 2, Module 2 executes the first step in EKF by computing the state matrix ($A$), followed by the state time update ($x_k$). The high-level architecture of the Module 2 is illustrated in Figure 37.

The state time update uses the a priori information; thus, the subscripts indicate previous state ($k$-1) and the current state ($k$):

$$x_k = Ax_{k-1} + Bu_{k-1} \quad (4.1)$$

The state matrix ($A$) is built using look-up-tables (LUTs) and 2D-interpolation. Temperature and SOC are the inputs to the LUTs. Figure 3 illustrates the internal architecture of the index finder, which determines the high and low SOC indices. The generic approach to producing the scaling factors, $\alpha$ and $\beta$, and the SOC indices is presented in equations (4.2) and (4.3). In this case, if $Z$ represents the array of 20 SOC values, and $T$ represents the array of 2 temperatures, $\alpha_0$ and $\beta_0$ are computed with these two equations (4.2) and (4.3):

$$\alpha_0 = \frac{T_{k,\text{C}} - T[\text{Low}]\text{C}}{T[\text{High}]\text{C} - T[\text{Low}]\text{C}} = \frac{T_{k,T[0]}}{T[1] - T[0]} = \frac{T_{k,0}}{25} = 0.04(T_{k}) \quad (4.2)$$
\[
\beta_0 = \frac{SOC_k - Z_{[Low]}^{[High] - Z_{[Low]}}}{0.05} = 20(SOC_k - Z_{[Low]})
\]  

The software algorithm that implements the above equations is shown in Table 7. Embedded C on the MicroBlaze soft processor truncates the floating-point value when casting it to an integer; in other words, always rounds down to the nearest integer.

**Table 7: Software algorithm to find the \(\beta\) indices and producing the scaling factors**

<table>
<thead>
<tr>
<th>Algorithm for index finder</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha_0 = 0.04*Tk);</td>
</tr>
<tr>
<td>(\alpha_1 = 1-\alpha_0;)</td>
</tr>
<tr>
<td>soc_val = SOC*20;</td>
</tr>
<tr>
<td>Low = (int)soc_val;</td>
</tr>
<tr>
<td>High = Low +1;</td>
</tr>
<tr>
<td>(\beta_0 = \text{soc_val} - \text{Low};)</td>
</tr>
<tr>
<td>(\beta_1 = 1-\beta_0;)</td>
</tr>
</tbody>
</table>

Our hardware architecture for the index finder is shown in Figure 38. In this case, we must integrate a check condition (i.e., \(\beta < 0\)) to the result to ensure that “Low” represents the lower bounding integer, since the FPI (floating-point-to-integer) component rounds to the nearest integer instead of always rounding down. This Check module determines whether \(\beta_0\) is negative or not, by examining the most significant bit (MSB). If the MSB is 1, the value is negative. A floating-point comparator could be used, but integer comparison is both faster and cheaper in terms of resources. In the case \(\beta_0\) is negative, the FPI rounded up to create the integer and we execute the top or truce branch of the decision. High=Low=(int)soc_val and Low=High-1. In our hardware design, the computation of “soc_val” is followed by two parallel operations: (1) one that performs FPI, in order to find “Low”, and then converting “Low” back to floating point; (2) another one that computes “soc_val+1”. For both parallel operations, only one adder is utilized, due to careful timing of the computations. We also design a separate “Check_T” module to compute \(\alpha\) values and determine the case value input to the user-designed exponential \(e^x\) module employed in Module 4. The Check_T module uses a multiplier and an adder to create \(\alpha_1\) and \(\alpha_0\) and a floating-point comparator.
to determine case value. The exponential module is optimized for 5 different temperature ranges and the comparator determines which coefficients to use in the module. The input case and $\alpha$ values are stored in on-chip BRAMs for subsequent use.

\[ A[i] = \alpha_1 \beta_1 (BigA_0[Low][i]) + \alpha_0 \beta_1 (BigA_1[Low][i]) + \alpha_1 \beta_0 (BigA_0[High][i]) + \alpha_0 \beta_0 (BigA_1[High][i]) \]  

(4.4)

The resulting $\alpha$ and $\beta$ are used to create the state matrix (A) as presented in equation (4.4), where $[i]$ represents the states:

\[ x_k[i] = A[i]x_{k-1}[i] + u_{k-1}. \]  

(4.5)

The internal architecture for Index Finder

The original LUT for A matrix (BigA) was a 3D matrix, with dimensions of temperature, state of charge (SOC), and system states, which led to a 2x20x5 matrix. For our design, we split the LUT into two 20x5 matrices, BigA0 containing the values for $T = 0^\circ C$, and BigA1 containing values for $T=25^\circ C$. This also allows us to use two dual-port BRAMs to store the LUTs and reduces the overhead associated with addressing a 3-dimensional matrix. Since A is a diagonal matrix, A is stored as a vector in our hardware module, with the implied zeros and A[i] represents the diagonal of the 5X5 state matrix. The state $x_k$ is also a vector; hence, the operation $A x_{k-1}$ does not need any additions. The B matrix is defined as a set of ones therefore $Bu_{k-1} = u_{k-1}$. As a result, the state time update (4.1) is reduced to:

\[ x_k[i] = A[i]x_{k-1}[i] + u_{k-1}. \]  

(4.5)

The internal hardware architecture to compute the A matrix and $x_k$ (i.e., equations (4.4) and (4.1)) are shown in Figure 39. As illustrated, the first two parallel multipliers compute the four terms in equation (4.4) for each state i. In this case, Mult1 and Mult2 perform the multiplications.
on the BigA0 and BigA1 data, respectively. Next, the addition operation (using Add1) is performed to add the first two terms and the last two terms together. The result of the Add1 is forwarded to Add2 to obtain the final A[i] matrix. Add2 result is written to the BRAM, while this result is also multiplied with $x_{k-1}[i]$ using Mult3. Next, Mult3 result is added to $u_{k-1}$, and the final addition result $x_k[i]$ is also stored in the BRAM.

![Figure 39: Internal architecture for A Matrix computation](image)

Our overall Module 2 is designed using only two multipliers and two adders shared by the sub modules. In this case, each multiplier and adder comprise a ready signal to indicate that the output is valid, and uses an input valid signal for the subsequent multiply/addition operations. These signals reduce the overhead of the FSM. In the standard Kalman filter process from Figure 35, the next step is to compute the error covariance time update:

$$\sigma_{x,k} = A\sigma_{x,k-1}A^T + \sigma_w$$

(4.6)

However, in our hardware design, this computation is delayed and done in Module 5 with the Covariance Matrix computation.

### 4.1.3 Module 3

Following the design flow in Figure 36 the next step is to compute the output prediction. Module 3 is used in both the time and measurement output updates in Stage 3 and Stage 6. In both stages, Module 3 updates the SOC, creates the observation matrices (C, D), and computes the output function ($y_{var,k}$). In this case, the output function comprises de-biased variables, which have physical meaning only when the nonlinear corrections are applied [11].
The top-level architecture of Module 3 is illustrated in Figure 40. As shown, in the beginning Module 3 computes the SOC based on the time update, \( x_k \), and the lithium concentration in the solid surface at the negative electrode, \( c_s^{neg} \). For our design, we use the cell model parameters from the Doyle cell parameters in [11]. These parameters are held constant for the temperature range used for this design.

The SOC calculation is as follows:

1. Calculate the average solid surface concentration in the negative electrode, \( c_{s,avg}^{neg} \) using \( x_k \) and cell model parameters that are held constant for this range of temperatures.

\[
\begin{align*}
c_{s,avg}^{neg} &= c_{s,\text{gain}}^{neg} x_k[4] + c_{s,max}^{neg} \quad SOC_{neg} = m_{\text{avg}} x_k[4] + b_{\text{avg}} 
\end{align*}
\]

2. Calculate the ratio of the average to the maximum solid surface concentration in the negative electrode:

\[
\theta_{neg} = \frac{c_{s,avg}^{neg}}{c_{s,max}^{neg}} = \frac{c_{s,avg}^{neg}}{c_{s,\text{max}}}^{c_{s,\text{max}}^{-1}}
\]

3. Calculate the SOC using \( \theta_{neg} \) and precalculated m and b constants.

\[
SOC = \frac{\theta_{neg} - \theta_0}{\theta_{100} - \theta_0} = m_{SOC} \theta_{neg} + b_{SOC} \tag{4.7}
\]
In each of the above steps we combine the cell model constants and rearranging the terms of the SOC computations from [11] to simplify the design. In steps 1 and 3, we manage to use the standard equation form \((y = mx + b)\) and in step two we replace division with multiplication as it requires less resources and less clock cycles. The \(m\) and \(b\) constants are pre-calculated and stored in BRAM.

The internal architecture of the SOC module is depicted in Figure 41. In this case, in loop 2 (L2) necessary data is sent to the adders and multipliers from the on-chip memory. In loop 3 (L3), the check condition checks whether \(0 < \theta < 1\), to ensure that this value falls within the bounds before calculating the final SOC which is then stored in memory. The updates of the SOC drive the updates to the SOC indices as well as the \(\beta\) terms, thus reusing the two sub-modules (i.e., Index Finder and Alpha Beta) from Module 2.

As illustrated in Figure 40, the C and D matrices are created reusing the A Matrix module (in Figure 39). For the D matrix module, 19 iterations are used instead of 5 iterations as in A matrix. For the C matrix module, an outer loop is added to the A matrix module to create an additional dimension. The equations for the C and D matrix are shown below:

\[
C[i][j] = \alpha_1 \beta_1 (BigC_0[Low][i][j]) + \alpha_1 \beta_0 (BigC_0[High][i][j]) + \\
\alpha_0 \beta_1 (BigC_1[Low][i][j]) + \alpha_0 \beta_0 (BigC_1[High][i][j])
\]

\[
D[j] = \alpha_1 \beta_1 (BigD_0[Low][j]) + \alpha_1 \beta_0 (BigD_0[High][j]) + \\
\alpha_0 \beta_1 (BigD_1[Low][j]) + \alpha_0 \beta_0 (BigD_1[High][j])
\]

After computing the C and D matrices, the output equation:

\[
y_{var,k} = C_k x_k + D_k u_k
\]  

is computed using the Y_out module. In order to compute \(C_k x_k\), we create an optimized linear multiply accumulate (MAC) module, which consists of one FP multiplier and 3 FP adders (as shown in Figure 42). Our primary goal of designing this efficient MAC module is to reduce the area, and the secondary goal is to optimize the speedup. In this case, the parallel computation of the \(19 \times 5\) \(C_k\) matrix by our proposed MAC module would optimize the execution time. While the
MAC module performs $C_k x_k$, the Index Finder module simultaneously computes the open circuit index values and the $\beta$ terms for the LUTs for subsequent analysis in Module 4. Unlike the design in Chapter 3, the proposed MAC module for EKF does not use input buffers at each stage. Instead, as shown in Figure 42, the “ready signals” from the preceding multiplier/adder modules are used to trigger the subsequent adder modules.

For the MAC module, the input C matrix is partitioned into 19 vectors, and fed into the MAC module one vector at a time. The result of the MAC module is added to the corresponding $D_k u_k$ term, in order to compute the output function $y_{var,k}$. After computing $y_{var,k}$, the non-linear corrections are computed and applied.

4.1.4 Module 4

Module 4 is used in Stage 4 and Stage 7. In both stages, Module 4 computes the non-linear corrections and the output voltage. As illustrated in Figure 43, Module 4 is created with five sub-modules and various FP IPs. In this case, the non-linear corrections computation is iterative, thus is not a good candidate for parallelism. The output voltage $y_{volt,k}$ is dependent on the values found using the non-linear corrections and is shown below:
\[ y_{\text{volt},k} = y_{\text{var},k}[17]FR_{\text{film}+} - y_{\text{var},k}[15]FR_{\text{film} -} + \varphi_e + \Delta \eta - u_kR_{\text{term}} + \Delta U_{ocp} \] (4.9)

\( FR_{\text{film}+}, FR_{\text{film} -}, \) and \( R_{\text{term}} \) are resistances in the film and terminal and are precalculated constants applicable to the temperature range and SOC for this model and are stored in ROM. Ultimately the output voltage is used to calculate the voltage residual or innovation, \( V_{\text{res}} = v_k - y_{\text{volt},k} \), used to determine the adjustments required for the state measurement update performed in the second half of the EKF.

The first four corrections are the concentration in the electrolyte \( c_e \), which are performed first, followed by the solid surface concentrations corrections \( c_s \), for both ends of the terminal represented by 0 and L as in Figure 2:

\[
\begin{align*}
    c_{e,0} &= y_{\text{var},k}[0] + 1 & \text{(4.10)} \\
    c_{e,L} &= y_{\text{var},k}[3] + 1 & \text{(4.11)} \\
    c_{s,0} &= y_{\text{var},k}[7] + b_{c,\text{neg}} & \text{(4.12)} \\
    c_{s,L} &= y_{\text{var},k}[7] + b_{c,\text{pos}} & \text{(4.13)}
\end{align*}
\]

The above corrections comprise simple memory calls and additions (by FP adder). The concentrations in the electrolyte results are inverted to calculate \( 1/c_{e,0} \) and \( 1/c_{e,L} \), and stored in the memory for subsequent analysis. Our FP divider takes 20 clock cycles to generate the first division result, thus other computations are performed during the division operation. The variables of the corrections (i.e., \( b_{c,\text{neg}} \) and \( b_{c,\text{pos}} \)) are the functions of the maximum negative and positive SOC, and the negative and positive \( c_{s,\text{max}} \), respectively. However, all these variables are constant for this temperature range, hence are pre-calculated and placed in the ROM. The next two corrections, which are potential in the electrolyte and flux computations, are interleaved to avoid waiting for any long-lead time operations such as square root, division and natural logarithmic. The equation (4.14) below presents the computation of the total potential in the electrolyte (\( \varphi_e \)). In this case, \( T_k \)
is the current temperature and $\phi_{const}$ is a pre-calculated value based on the Doyle parameters. We use the Xilinx FP natural logarithmic IP to compute $\varphi_e$:

$$\varphi_e = y_k[6] + T_k \phi_{const} \ln \frac{c_eL}{c_{e,0}}$$ (4.14)

Though the flux values do not need correction, the negative and positive flux values are used in the over-potential correction and are calculated as follows:

$$j_{0,neg} = \sqrt{\left( (c_{s,neg,max} - c_{s,0})c_{s,0}c_{e,0} \right)} \cdot e^{-m_{e,neg} \frac{1}{T_k} + b_{e,neg}}$$ (4.15)

$$j_{0,pos} = \sqrt{\left( (c_{s,pos,max} - c_{s,L})c_{s,L}c_{e,L} \right)} \cdot e^{-m_{e,pos} \frac{1}{T_k} + b_{e,pos}}$$ (4.16)

In this case, the $m_{e,neg}, m_{e,pos}, b_{e,neg}$ and $b_{e,pos}$ are pre-computed constants stored in the ROM, and $1/T_k$ is computed by the Check_T in Module 2 and stored in the RAM. The internal architecture of the flux computation is shown in Figure 44: Internal architecture for Flux computation. Since there are different values for $m_{e,neg}, m_{e,pos}, b_{e,neg},$ and $b_{e,pos}$ we implement two different Taylor series to generate accurate $e^x$ results. Both Taylor series are 6th order, however, their coefficients differ. Both are executed in parallel to reduce the execution time. We also split the temperature range into six sections to further reduce the errors from the numerical approximation of $e^x$. After selecting the focal point, the constants of the Taylor series become the multiples of the base values. These base values (for the six temperature regions) are stored in the memory and each of the expansion terms are computed using a selected base value.
In the flux computation (in Figure 44), square-root, $e_{neg}^x$, and $e_{pos}^x$ are computed in parallel to reduce the overall execution time. These results are sent through an inverter to generate the inverse of $j_{0,pos}$ and $j_{0,neg}$ and are stored in the memory for subsequent analysis.

The over-potential correction ($\Delta \eta$) is obtained using the following equation (4.17):

$$\Delta \eta = \eta_L - \eta_0 = \frac{T_k2R}{F} \cdot \left( \text{asinh}\left(\frac{y_k[17]}{j_{0,pos}}\right) - \text{asinh}\left(\frac{y_k[15]}{j_{0,neg}}\right) \right)$$  (4.17)

To create the equation (4.17), we design our own IP for hyperbolic arcsine (asinh) utilizing the Chebyshev Approximation, since Xilinx IP is not available for this operation. As stated in [59] the Chebyshev Approximation spreads the error over the interval, allowing a single equation to be used for the entire temperature region. Furthermore, no division is required, resulting in more efficient design. We use the following 8th order Chebyshev equation in [59] to approximate asinh:

$$\text{asinh}(x) = -0.017123x^7 + 0.0629397944x^5 - 0.16460894x^3 + 0.9999157047x$$

The 8th order function results in a 0.00957% error versus the 0.19% error of the 6th order function; and use only 3 more multiplications and one addition to compute the asinh. This is deemed as an acceptable tradeoff. The internal architecture of the Asinh is shown in Figure 45: Internal architecture for Asinh computation.
The open circuit voltage (OCV) or open circuit potential (OCP) is the final linear correction needed to compute $y_{volt,k}$. The OCV and its derivative are non-linear functions, which are represented by two LUTs. The index computations are performed in Module 3 and stored in the memory. The $\Delta U_{OCP}$ is computed using the interpolation presented in equation (4.18). This interpolation is organized in such a way to reduce the address computations needed for the memory calls, as well as to minimize the number of adders and multipliers used.

$$
\Delta U_{ocp} = \beta_1 (BigUocp_{pos}[Low] - BigUocp_{neg}[Low]) + \\
\beta_0 (BigUocp_{pos}[High] - BigUocp_{neg}[High])
$$

(4.18)

The same equation (4.18) with different LUTs is utilized to compute the derivative, $\Delta dU_{ocp}$, in order to determine the $\hat{C}$ in Stage 5. Hence, the module for $\Delta U_{OCP}$ is re-executed to generate $\Delta dU_{ocp}$ as well. The second execution for $\Delta dU_{ocp}$ is performed in parallel with the computation of $y_{volt,k}$ as there are no dependencies between the two calculations. Once the linear-correction results are available, the output voltage is computed using the equation (4.9) shown again here:

$$
y_{volt,k} = y_{var,k}[17]FR_{film+} - y_{var,k}[15]FR_{film-} + \tilde{\phi}_e + \Delta \eta - u_k R_{term} + \Delta U_{ocp}
$$

We chose to calculate the difference between the measured and predicted voltage (innovation) in module 5 as the value is not necessary for stage 7. With the completion of the predicted voltage, we move to stage 5 to calculate the Kalman Gain.
4.1.5 Module 5

Figure 46: Module 5 - top-level architecture for stage 5

In Stage 5, Module 5 computes the error covariance time and measurement updates, the Kalman gain and the state measurement updates. Because of the non-linear nature of the output equation $y_{volt,k}$, a further correction is required for the output matrices C and D. The state equation, $x_k$, is fully linear and no corrections are required for the A and B matrices. In [11] the author develops the non-linear Kalman gain calculation and defines new output matrices $\hat{C}$ and $\hat{D}$. We will be using these linearized matrices for our future physics-based MPC design as well, which will dramatically reduce the number of computations in the MPC algorithm.

The overall architecture of the Module 5 is depicted in Figure 11. In this case, the Chat_Dhat module consists of a FSM that controls the operations. The Chat_Dhat module serves as the input module for Stage 5. During the execution of Chat_Dhat module, all the associated memory ports are utilized. As a result, the SigmaX module, which computes the time update to the error covariance matrix, must wait to be executed until after the execution of Chat_Dhat module. As in [60], multi-ported memories could be utilized to execute the Chat_Dhat and the SigmaX modules in parallel.

The development of the new output matrices, $\hat{C}$ and $\hat{D}$ separates the calculation into four terms that reflect $y_{volt,k}$ in equation (4.9) and are summed to produce the new output matrices:
\[ \hat{C} = \hat{C}_1 + \hat{C}_2 + \hat{C}_3 + \hat{C}_4 \]  
\[ \hat{D} = \hat{D}_1 + \hat{D}_2 + \hat{D}_3 + \hat{D}_4 \]

for each part that follows, \([i]\) represents the states 0 through 4

Part 1 is based on the film resistance:

\[ \hat{C}_1[i] = C[17][i]FR_{film,pos} - C[15][i]FR_{film,neg} \]  
\[ \hat{D}_1 = D[17]FR_{film,pos} - D[15]FR_{film,neg} \]

Part 2 is based on the potential in electrolyte, \(\varphi_e\):

\[ \hat{C}_2[i] = C[6][i] + T_k\varphi_{const} \left(C[3][i] \frac{1}{\epsilon_{e,L}} - C[0][i] \frac{1}{\epsilon_{e,b}}\right) \]  
\[ \hat{D}_2 = D[6] + T_k\varphi_{const} \left(D[3] \frac{1}{\epsilon_{e,L}} - D[0] \frac{1}{\epsilon_{e,b}}\right) \]

Part 3 is based on the overpotential term:

\[ \hat{C}_3[i] = T_k \frac{R}{F} \left(C[17][i] \frac{1}{j_{o,pos}} - C[15][i] \frac{1}{j_{o,neg}}\right) \]  
\[ \hat{D}_3 = T_k \frac{R}{F} \left(D[17] \frac{1}{j_{o,pos}} - D[15] \frac{1}{j_{o,neg}}\right) \]

Part 4 is based on the derivative of the OCV, \(\Delta dU_{ocp}\):

\[ \hat{C}_4[i] = dSOC[i] \cdot \Delta dU_{ocp} \]  
\[ \hat{D}_4 = 0 \]

The constants for each part, \(dSOC[i]\), \(R/F\), \(\varphi_{const}\), \(FR_{film,pos}\), and \(FR_{film,neg}\), are all stored in ROM. There is no actual division in the above calculations as we instead use the inverse values created in Module 4. The Chat_Dhat module uses a finite state machine to control the FP operations above. After the output matrices are completed, we are able to start the calculation of the error covariance time update using equation (4.6) shown below:

\[ \sigma_{x,k} = A\sigma_{x,k-1}A^T + \sigma_w \]

The internal architecture of the SigmaX_time module is shown in Figure 12. In this case, \(\sigma_w\) is the uncertainty of the current sensor, and is a constant 5×5 matrix stored in the ROM.
Examining $A\sigma_{x,k-1}A^T$ and utilizing the properties of a diagonal matrix, we can rearrange the terms and execute $AA^T$ which creates a $5\times5$ matrix of multipliers for $\sigma_{x,k-1}$. The individual terms of $\sigma_{x,k}$ can be calculated as follows:

$$\sigma_{x,k}[n][m] = AA^T[n][m] \cdot \sigma_{x,k-1}[n][m] + \sigma_w[n][m] \quad (4.29)$$

Referring to Figure 47, $AA^T[n][m]$ is calculated using Mult1 whose output_valid signal is registered to provide a one clock cycle delay before forwarding it as the input_valid signal of Mult2. The other input to Mult2 is $\sigma_{x,k-1}[n][m]$. Similarly, the output_valid signal of Mult2 is registered and forwarded as the input_valid signal of Add1. The output of Mult2 and $\sigma_W[n][m]$ are the inputs to the Add1. The results of Add1, $\sigma_{x,k}[n][m]$, are stored in the memory for subsequent computations of Module 5. This approach reduces the resources needed compared to calculating equation (4.6) in the traditional multiply-accumulate manner.

![Figure 47: Internal architecture for SigmaX computation](image)

Given $\sigma_{x,k}$ we compute the Kalman gain utilizing the Update_SigmaX module (in Figure 46). The Kalman Gain matrix ($L$) is computed based on the equation (4.30):

$$L = \frac{\sigma_{x,k}\hat{C}^T}{\sigma_y} \quad (4.30)$$

Where $\sigma_y$ is the voltage error covariance and computed using equation (4.31).

$$\sigma_y = \hat{C}\sigma_{x,k}\hat{C}^T + \hat{D}\sigma_v\hat{D}^T \quad (4.31)$$

In this case, $\sigma_v$ is the error in voltage measurement. Since $\hat{C}$ is a vector, $\sigma_{x,k}\hat{C}^T$ is computed first, stored in the memory, and then reused for $L$, $\sigma_y$, and $\sigma_{x,k+1}$ equations (4.30), (4.31), and (4.33) respectively. The MAC module (in Figure 11) is used to compute the $\sigma_{x,k}\hat{C}^T$, and then the MAC
result is loaded into a local register array (ChatSigX). Simultaneously, the MAC result is multiplied with an appropriate term from Ĉ, and the result is forwarded to an adder to perform the final accumulation for σ_y. A FP divider is employed to compute the final division of the Kalman Gain, and the division result is forwarded to the Update_Xhat module with a ready signal. The final step of the Update_SigmaX module is to compute the measurement update of the error covariance matrix in equation (4.32):

\[ \sigma_{x,k+1} = \sigma_{x,k} + L\sigma_y L^T \]  

(4.32)

Since σ_y results in a scalar, we can simplify the computation for the error covariance measurement update by replacing L with \( \frac{\sigma_{x,k} \hat{C}^T}{\sigma_y} \) in (4.32) and cancelling the σ_y terms:

\[ \sigma_{x,k+1} = \sigma_{x,k} + (\sigma_{x,k} \hat{C}^T)L^T \]  

(4.33)

The second term of (4.33), \( (\sigma_{x,k} \hat{C}^T)L^T \), is accomplished with a single multiplier and results in a 5×5 matrix that is added, term by term, to \( \sigma_{x,k} \). The result of the adder is written to the on-chip memory, overwriting the current \( \sigma_{x,k-1} \) to support the next iteration.

The state measurement update (4.34) is obtained by the Update_Xhat module executed in parallel with the Update_SigmaX module. In this case, the Update_Xhat module consists of an adder and a multiplier and starts when the Update_SigmaX module indicates that L is ready:

\[ x_{k+1} = x_k + L(v_{k,in} - y_{vol{k},k}) \]  

(4.34)

The innovation term \( v_{k,in} - y_{vol{k},k} \) is a scalar making this update simple and fast. The results are stored in memory, completing stage 5.
4.1.6 Stages 6 and 7

Figure 48: Stages 6 and 7

The final updates required are the SOC and the output matrices \( \hat{C} \) and \( \hat{D} \). These are accomplished in stages 6 and 7. Rather than creating a specialized hardware module targeted to just those updates, Modules 3 and 4 are re-executed using the state measurement update, \( x_{k+1} \) as the input. Upon completion of stage 7, the top-level FSM checks for an active exit signal that is driven by the MicroBlaze microcontroller. The iterative process (in Figure 48) continues until the exit signal is asserted. Reusing Module 3 and Module 4 reduces the occupied area on chip. Furthermore, the updated \( \hat{C} \) and \( \hat{D} \) matrices will be utilized in the future design for the PBM MPC algorithm.

4.2 Experimental Results and Analysis

We perform experiments to evaluate the feasibility and efficiency of our proposed FPGA-based hardware accelerator for the PBM EKF observer designed to support an embedded PBM based MPC. We also compare our hardware accelerator with the baseline model of the physics-based EKF written in Matlab [11], in order to evaluate and validate the correctness and functionalities of our proposed design. Furthermore, we create an embedded software design for the physics based EKF in order to evaluate our embedded hardware design. Prior to the embedded software design, we created the software for physics-based EKF in C and executed on the desktop computer. These desktop results are compared and verified with the baseline results as well as with our embedded hardware/software results.
All our experiments are carried out on the Xilinx ML605 FPGA development board, which utilizes a Virtex 6 XC6VLX240T-FF1156 device [53]. All the hardware modules are designed in mixed VHDL and Verilog using Xilinx ISE and XPS 14.7 design tools and executed on the FPGA (running at 50MHz). Xilinx ISim is used to verify the results and functionalities of the designs. Our embedded software modules are written in C and executed on a 32-bit 128kB MicroBlaze soft-processor (running at 100MHz) on the same FPGA using Xilinx XPS and SDK 14.7 design tools. For both the hardware and software, the fundamental operators, such as adders and multipliers, are designed using single-precision floating-point (FP) units [54] from Xilinx IP core library, whereas the baseline Matlab model [11] was designed using double-precision FP operators. The experiments are performed using an Urban Dynamometer Driving Schedule (UDDS) [61] current profile for the input \( u_k \) and the measured output \( v_k \). The physics-based EKF is initialized using a SOC of 60% and an initial state covariance of diagonal matrix \([10, 100, 1000, 1000, 5\times10^7]\) [11]. This physics-based EKF is designed to support the normal driving of HEV with the charging and discharging based on the UDDS driving profile [11].

### 4.2.1 Functional Verification

It is crucial to confirm that the proposed embedded architectures operate correctly. In this case, we utilize the measured voltage value supplied by the UDDS driving profile to determine whether \( y_{\text{volt},k} \) is correct or quite similar. The difference between \( y_{\text{volt},k} \) and the driving profile is then used to compute the Kalman Gain. The first two graphs are the input current and voltage that represent a UDDS driving profile. The current in Figure 49 is the \( u_k \) primarily used in the time update portion of the EKF algorithm. The voltage in Figure 50 is the measured or reference voltage \( v_{k,\text{in}} \) used in equation (4.34). Typically, the UDDS driving profile represents the driving conditions within the city, and is used by the Environmental Protection Agency (EPA) for light vehicle testing [61].
The graph (in Figure 51) illustrates the performance of the HW architecture and the embedded software architecture in comparison to the baseline MATLAB SOC prediction.
As depicted in Figure 51, the embedded hardware results start out slightly noisy compared to the baseline MATLAB results but settles down fairly quickly. Considering that the SOC is unitless and is expressed as a percentage or a value ranging from 0 to 1, with 1 or 100% being fully charged, the mean error is 0.00017809 and the standard deviation is also very small at 1.258E-4 demonstrating that the hardware, at single precision, compares favorably with the double precision MATLAB output. The embedded software performance is even better with a mean error of 8.996E-7 and a standard deviation of 5.781E-7.

Figure 52 shows how the HW and embedded SW compares to the MATLAB baseline for the predicted voltage. The HW performs even better on the predicted voltage $y_{volt,k}$ from equation (4.9). For the embedded hardware $y_{volt,k}$ values, the standard deviation is 88.1μV with a bias or mean error of 10.6μV compared to the baseline MATLAB results. The embedded software error
has a standard deviation of 0.36 \( \mu V \) with a mean error of 0.001 \( \mu V \). Overall the single precision floating point hardware architecture shows very good performance results and matches the baseline MATLAB results very closely.

The difference in performance of the embedded HW architecture is most likely due to the user designed estimators for \( e^x \) and \( \text{asinh} \, x \), additionally the output from the floating point square root IP provided by Xilinx is not an exact match for the square root operation in the embedded software or MATLAB. It is likely the error can be improved with more robust implementations of these functions; however, it will be at a cost of area and execution time.

### 4.2.2 Performance Analysis: Execution Time and Speedup

The total time to execute an iteration of the physics-based EKF for embedded hardware and embedded software is presented in Table 8. The total time to execute the baseline Matlab software design is also presented in Table 8. The execution time for each design is measured 10 times and the average is presented. In this case, Matlab software design is executed on the Intel Xeon processor running at 2.4GHz, whereas the embedded designs are executed at 50MHz or 100MHz on the ML605 board.

From Table 8, considering total execution time, our embedded hardware (at 50MHz) for physics-based EKF is 28 times faster than the baseline Matlab design. Furthermore, our embedded hardware is 21 times faster than the equivalent embedded software running on a MicroBlaze processor. From these results and analysis, it is evident that utilizing FPGA-based hardware accelerators can achieve significant speedup compared to the software-based designs. With these speedups, our proposed hardware should be able to monitor and control multiple battery cells individually.

From the various runs of our embedded hardware design, the execution time only varies slightly. The fastest iteration takes 35.5\( \mu s \), while the slowest iteration takes 36.3\( \mu s \) with the difference of 0.8\( \mu s \). This execution time is excellent considering the 1 second update interval of the
battery cells. Thus, our proposed physics-based EKF, coupled with the an MPC design similar to Chapter 3 (which takes 450 μs per cycle) on a single chip, can be utilized to monitor and control approximately 2000 battery cells.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Embedded SW (100MHz)</th>
<th>Matlab (2.4GHz)</th>
<th>Embedded H@ (50MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (µs)</td>
<td>752.00</td>
<td>994.00</td>
<td>36.02</td>
</tr>
<tr>
<td>Speedup over embedded SW</td>
<td>--</td>
<td>0.76</td>
<td>20.88</td>
</tr>
<tr>
<td>Speedup over Matlab</td>
<td>1.32</td>
<td>--</td>
<td>27.60</td>
</tr>
</tbody>
</table>

### 4.2.3 Resource Utilization

The cost analysis on space is carried out to examine the area-efficiency of our embedded hardware designs. The resource utilization for our embedded hardware design presented in Table 9 comprises the resource utilization for the overall system-level design, similar to the space analysis in Chapter 3. As illustrated, the total number of occupied slices, total number of BRAMs, and total number of DSP slices required for embedded hardware design are 10702, 40, and 49 respectively. As demonstrated in Chapter 3, most of the area-efficient embedded hardware architectures with the system-level designs, typically occupy similar or more area on chip.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Occupied Slices</th>
<th>Slice Registers</th>
<th>LUTs</th>
<th>BRAMS</th>
<th>DSP48E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB EKF embedded HW</td>
<td>10702</td>
<td>19501</td>
<td>32169</td>
<td>40</td>
<td>49</td>
</tr>
</tbody>
</table>

### 4.2.4 Comparison with Related Work

We investigate the existing work on hardware architecture for ECM-based EKF as well as physics-based EKF. From our extensive investigation and to the best of our knowledge, similar work does not exist in the published literature that provides FPGA-based hardware accelerator for EKF specifically for physics-based battery cell models. Most of the existing EKF architectures
were designed for the equivalent circuit model (ECM). Two works [12],[62], performed similar experiments to obtain the speedup and space. In [12], a dual EKF was implemented using the ECM for the battery cell. This was implemented on Altera FPGAs without the system-level architecture. Another ECM-based EKF was designed to estimate the state of charge (SOC) using fixed-point arithmetic, which was implemented on Altera Cyclone II FPGA [62]. Since these existing works were created for different EKF algorithms and were implemented on different development platforms, it is challenging to make a fair comparison in terms of space and speedup.

4.3 Conclusion

In this chapter, we introduced a novel, unique, and efficient FPGA-based hardware accelerator for a physics-based EKF observer for the battery cells. Our embedded hardware accelerator is optimized for the PBM in [11] but is set up to handle PBM models developed at different temperatures and SOC. Our hardware accelerator executed up to 21 times faster than equivalent embedded software, and 27 times faster than the baseline Matlab software on a desktop computer (at 2.4 GHz).

Utilization of a physics-based model enables closer observation as well as tracking and controlling of the chemical aging mechanisms in lithium ion batteries, which could otherwise be disregarded when utilizing ECM. The ability to monitor the aging mechanisms facilitates the accurate settings of the safety margins, which in turn would enhance the performance and extend the life of the battery. Our proposed hardware accelerator was designed in such a way that the linearized output matrices, state matrices, and SOC estimation can be used by the physics-based MPC, thus making both more area-efficient.
Chapter 5

EMBEDDED SOFTWARE ARCHITECTURE FOR PB MPC

AS A SMART SENSOR

5.1 Architecture Overview

Having successfully developed functional embedded architectures for a physics-based observer for a battery cell in Chapter 4 this chapter explores the possibility of what can be achieved using a resource limited 32-bit microprocessor with 128KB total memory and the embedded EKF observer. The focus remains on creating an embedded battery management system that extends the life of the battery by preventing a known aging electrochemical reaction. In this chapter, we introduce a smart sensor that controls and constraints the operations of the individual battery cells to prevent aging and extend the life of the battery cell.

Recently a modular BMS with MPC was proposed in [11],[44] that reduces the computational load on the battery cell controller while retaining the ability to protect the battery from degradation. The proposed BMS enables the control burden to be split among the energy management sub-systems effectively reducing the computational load for the battery cell controller. In a sense it is a similar idea to edge computing where the subsystems handle as much of the computation as possible and only provide the minimum necessary information to the overarching system. To achieve this, the modular BMS requires a smart sensor based on a model predictive control (MPC) that will regulate the cell current for safe and reliable performance of individual battery cells while the DC-DC converter handles the SOC balancing of the battery cells and the bus voltage regulation for powering auxiliary systems.

For the MPC as a smart sensor, the goal is to monitor the control input requested by the higher level management system and ensure that the requested control does not violate any of the
constraints that prevent aging and ensure safety. The smart sensor will utilize MPC’s “look-ahead” strategy to predict how the requested control signal would impact the battery cell, and accordingly optimizes the signal based on the model and the constraints. The optimization step will only occur if the requested control signal violates the constraints the smart sensor is monitoring. Because the desired control signal is provided externally, the computation of the unconstrained solution is greatly reduced. Instead the smart sensor focuses on monitoring the requested control input and only if the control input violates the constraints does the smart sensor execute the MPC optimization to calculate a correction factor for the requested control signal that will keep the battery cell operating at peak performance. If the constraints are not violated, then the smart sensor acts as a pass-through.

The contribution of this chapter is a unique and efficient embedded software architecture for physics-based MPC smart sensor for lithium-ion (Li-ion) battery cell control, suitable for an embedded 32-bit Microcontroller with 128KB of memory that functions within a time interval of 1 second. Since the electrochemical reactions are not physically measurable in a normal battery, the smart sensor utilizes an extended Kalman filter (EKF) as an observer to determine the internal states. Both the PB-MPC and EKF use state-space equations with overlapping functions, which in turn reduces the computation workload on embedded system.

While there are many existing works focused on reducing the computational workload for model predictive control while simultaneously increasing the accuracy and effectiveness of the battery models to facilitate realizing PB-MPC for embedded battery control such as [7],[8],[9],[11],[41],[44],[46],[47],[63],[64],[65] Few if any have taken their research to an embedded microcontroller software design. To the best of our knowledge, no similar work exists in the published literature that provides a functional embedded software architecture for a combination of EKF and MPC for physics-based battery management system for a microprocessor with a limited memory of 128 KB. Expanding the research beyond battery cell management there were a few designs for PB-MPC with an observer, most of these were simulated on desktop
computers, [9],[63] or an embedded system with 1GB of memory [64]. This work adapts the EKF-MPC proposed in [11],[44] to an embedded software architecture. The DC-DC converter control software is resident on the DC-DC converter and is not a part of this effort. This paper organized as follows, Section 5.2 provides a short background review, Section 5.3 discusses the software design, Section 5.4 describes the experiments and results with Section 5.5 covering the conclusion.

5.2 Background – PBM, EKF, and MPC

In this section, we review the physics-based model (PBM), extended Kalman filter (EKF) algorithm, and model predictive control (MPC) algorithm utilized to create our efficient embedded software architecture for the PB-EKF-MPC smart sensor.

5.2.1 Physics-based Model (PBM)

There were several research works that focused on developing a simplified physics-based model (PBM), which strived to retain the benefits of adaptability and knowledge of electrochemical reactions to enhance the performance and life by preventing conditions that lead to degradation.

One early approach utilized a single particle model (SPM) to create each electrode as a single particle, while neglecting the electrolyte dynamics. The SPMe was an extension of SPM, which incorporated electrolyte dynamics back into the SPM [63], whereas others focus on incorporating thermal effects [65].

The approach in this research work utilizes a reduced order model (ROM) of a lithium-ion battery cell based on the work in [7],[8],[9]. As stated in [7],[8],[9], this approach was especially suitable for embedded applications/devices, based on first order principles. In this approach, the state-space representations are created using the Discrete Realization Algorithm (DRA) developed in [41] using a set points of temperature and state of charge (SOC).

In [11], the author strives to reduce the complexity by combining two models of different temperature set points and the same SOC, in order to widen the accuracy of the resulting model, while reducing the amount of stored data required for the application. This combined (or blended)
model is the basis of the state and output matrices utilized in both PB-MPC and PB-EKF algorithm as well as the non-linear corrections detailed in PB-EKF in Section 5.3. The baseline model for the PB-EKF-MPC is detailed in [11].

The analysis in [41] demonstrated that the PBM using five states is the best-case scenario considering the accuracy, performance, and complexity tradeoffs. Figure 2 is shown again below to illustrate the reactions that the PBM centers around. In this case, the minimum reactions required for the five states are the concentration in the electrolyte \( c_e \), solid surface concentration \( c_s \), total potential in the electrolyte \( \varphi_e \), electric potential in the solid \( \varphi_s \), flux \( j_n \), and overpotential \( \eta \).

The side-reaction overpotential \( \varphi_{s-e} \) is an additional reaction used in BMS to regulate or prevent lithium plating, which is positioned in the PBM similar to \( \varphi_s \).

<table>
<thead>
<tr>
<th>Negative electrode ((n))</th>
<th>Separator ((m))</th>
<th>Positive electrode ((p))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varphi_n(z,t) )</td>
<td>( \varphi_s(x,t) )</td>
<td>( \varphi_p(z,t) )</td>
</tr>
<tr>
<td>( c_{e,n}(z,t) )</td>
<td>( c_{e,s}(x,t) )</td>
<td>( c_{e,p}(z,t) )</td>
</tr>
<tr>
<td>( j_{n,n}(z,t) )</td>
<td>( j_{n,s}(x,t) )</td>
<td>( j_{n,p}(z,t) )</td>
</tr>
<tr>
<td>( \eta(z,t) )</td>
<td>( \eta(x,t) )</td>
<td>( \eta(z,t) )</td>
</tr>
</tbody>
</table>

\[ z = 0 \quad z = 1 \quad z = 1 \quad z = 0 \]

\[ x = 0 \quad x = L_{ni} \quad x = L_{ni} + L \quad x = L \]

Figure 53: (Figure 2) Three region cell model illustrating variables and their positions [41]

### 5.2.2 Extended Kalman Filter (EKF) observer

For this architecture, the observer is the extended Kalman filter (EKF) from Chapter 4 which is based on the work done in [11]. The EKF was developed based on the work in [7],[8],[9],[41] using a PB-ROM that depends on the five basic reactions: \( c_s, \varphi_e, \varphi_s, j_n, \) and \( \eta \). The EKF is employed to observe the non-measurable states that describe the internal electrochemical reactions of the battery. Although, the observer is often utilized to monitor the SOC for the MPC, in this work, the SOC balancing is accomplished in the DC-DC converter and our smart sensor is being employed to reduce the aging caused by lithium plating, thus, the EKF monitors the model states and the side reaction overpotential \( \varphi_{s-e} \).
5.2.3 MPC

The PB-MPC algorithm follows the standard MPC flow. The difference comes when developing the cost function that will produce the desired smart sensor behavior. Given architecture of a distributed control system as shown in Figure 54, the goal of the smart sensor is to monitor the current $i_{g\text{ref}}$ and only actuate when the current is going to violate a constraint. If a constraint is violated then PB MPC will adjust $i_{g\text{ref}}$ to $i^*_{g\text{ref}}$. Additionally, if the current does not violate constraints, the smart sensor should be a pass through. For this smart sensor the cost function is based on the amplitude of the current, $u_k$ rather than the amplitude of the control increment, $\Delta u_k$ as developed previously. Using the same technique to organize future predictions of $u_k$ as is shown in Section 2.2.2 for the state and output predictions and where $u_k = \Delta u_k + u_0$ and $u_{k+1} = \Delta u_{k+1} + \Delta u_k + u_0$ and so on, [11] develops the following:

$$
\begin{bmatrix}
  u_k \\
  u_{k+1} \\
  \vdots \\
  u_{k+N_p-1}
\end{bmatrix} = 
\begin{bmatrix}
  1 & 0 & \ldots & 0 \\
  1 & 1 & \ldots & 0 \\
  \vdots & \vdots & \ddots & \vdots \\
  1 & 1 & \ldots & 1
\end{bmatrix}
\begin{bmatrix}
  \Delta u_k \\
  \Delta u_{k+1} \\
  \vdots \\
  \Delta u_{k+N_p-1}
\end{bmatrix} + 
\begin{bmatrix}
  1 \\
  1 \\
  \vdots \\
  0
\end{bmatrix} u_0
$$

That can be expressed as $U = \Sigma \Delta U + U_0$ or alternatively as $I_g = \Sigma \Delta U + U_0$. The basic cost function then becomes:

$$J = (R_s - I_g)^T (R_s - I_g) \quad (5.1)$$

Substituting for $I_g$ produces:

$$J = (R_s - \Sigma \Delta U - U_0)^T (R_s - \Sigma \Delta U - U_0) \quad (5.2)$$

Expanding and rearranging terms as in [11] produces the following:

$$J = \frac{1}{2} \Delta U^T 2\Sigma^T \Sigma \Delta U + \Delta U^T (-2\Sigma^T (R_s - U_0)) \quad (5.3)$$

Where $E = 2\Sigma^T \Sigma \Delta U$ and $F = -2\Sigma^T (R_s - U_0)$. The unique design for a distributed control system proposed in [11] has the subsystems effectively handling voltage regulation and SOC balancing, leaving the smart sensor to handle the battery current and protect the battery cell from aging and degradation. This greatly simplifies the MPC implementation in that the calculation for
the unconstrained global optimal control is greatly reduced, and with a control horizon of 1 and prediction horizon of 2, is almost eliminated. In order to act as a pass through if the proposed control signal $i_{\text{ref}}$ elicits no constraint violations, $i_{\text{ref}}$ must be the unconstrained global solution, $\Delta u^\circ$.

Finally, for the PB-MPC, the current state provided by PB-EKF is the observed state $x_{k+1}$. Due to the integral action to the state space equation, the PB-MPC requires the difference between the current-state and next-state as shown in (5.4).

$$\Delta x_{m,k} = x_{m,k+1} - x_{m,k}$$ (5.4)

Where $x_{m,k}$ is the observed state output from the PB-EKF. An extra state update is implemented to obtain this value utilizing the state and output matrices from PB-EKF.

In summary, the PB-MPC is a single-input/output approach that evaluates the cell current control requested by the higher level BMS. Where the requested cell current is within acceptable operating conditions, the PB-MPC simply acts as a pass through. If the requested cell current would cause lithium plating (a known aging mechanism), then the PB-MPC intervenes to take appropriate action and calculates an acceptable cell current. For this PB-MPC, the primary constraint is based on the side reaction overpotential ($\varphi_{s-e}$), which is a major factor in reduction of battery life and degradation in performance. Detailed description of this PB-MPC can be found in [11] and is discussed in Section 5.3
5.3 Embedded Software Architecture for Physics-Based MPC

Figure 54: Block diagram of BMS utilizing PB-MPC for smart sensor

Figure 54 illustrates the block diagram of the BMS (battery management systems) that employs the PB-MPC. This figure is modified from [44] to add the EKF and related functions. As depicted, the inputs to the EKF are the cell current and voltage, whereas the outputs of the EKF, i.e., the current-state of the battery, state of $\varphi_{s-e}$, state and output matrices ($A, B, \hat{C}, \hat{D}$), are the inputs to the PB-MPC. In this case, the cell current is a combination of string current ($I_{\text{string}}$) and control current ($i_g$).

Since PB-MPC incorporates integral action, PB-MPC operates on the change-in state from current-state to next-state and computes the change-in current, whereas the EKF operates on the current-state. This change-in current is added to the control current to produce a change-in state for the PB-MPC with the aid of EKF. Then, the PB-MPC outputs an updated control signal, $i_{\text{g ref}}^*$ that meets the current and $\varphi_{s-e}$ constraints to prevent aging and battery degradation, while satisfying the required performance. In this work, both MPC and EKF use physics-based model, thus, can be called PB-MPC and PB-EKF.
5.3.1 EKF

Our software functional flow for the PB-EKF is shown in Figure 55. Recalling the EKF flow from Chapter 4 Figure 36, note that the original stage 5 is split into two stage and stage 7 is no longer a repeat of stage 4. The EKF in Chapter 4 is designed to provide all the available chemical reactions as outputs for a follow-on MPC or other control algorithm. In this application, the PB EKF need only observe the SOC (for the DC-DC converter) and the side-reaction overpotential, \( \psi_{S-e} \) for the smart sensor. Thus, the flow is adapted in stage 7 to only include the non-linear correction for \( \psi_{S-e} \). This does not reduce program size but does reduce execution time. The software functional flow of the PB-EKF algorithm is shown in Figure 55. The software development closely follows that of the hardware architecture in Chapter 4 but is reiterated here for convenience. In Stage 1, the EKF is initialized with a SOC = 0.6 and, \( N_s \) is the number of states. In Stage 2, the EKF computes the state matrix (A) in equation (1). The state matrix (A) is built using look-up-tables (LUTs) and 2D-interpolation based on the temperature and SOC (state of charge) values. The A matrix is computed using the following interpolation.

\[
A[i] = (1 - \alpha)(1 - \beta)BigA_0[Low][i] + \alpha\beta BigA_1[High][i] + (1 - \alpha)\beta BigA_0[High][i] + \\
\alpha(1 - \beta)BigA_1[Low][i]
\]

(5.1)

In equation (5.1), the \( \alpha \) and \( \beta \) are the weight values based on the temperature and SOC, respectively, Low and High are the SOC floating point values and \( i \) is the state[11].
The challenge of the software architecture is to reduce the amount of stored data as much as possible, where possible. A compiler will attempt to find sections large enough to store contiguous data. Breaking up or partitioning large tables into smaller tables allows the compiler more flexibility in memory placement and prevents the situation while there is plenty of available memory, there is no single block large enough to store the original LUT. This split is also used to advantage in the hardware design in chapter 4. For example, the original LUT for the A matrix (BigA) was a 3D matrix, with inputs of temperature, state of charge (SOC), and system state values. For our software design and based on the blended model approach discussed in chapter 4 developed by [11], we split the LUT into two matrices (BigA0 for 0°C and BigA1 for 25°C). Additionally, the state matrix A is a 5x5 diagonal matrix, allowing A to be represented by a 5x1 vector of diagonal values. The state matrix B in this case is a vector of all 1’s and so is not stored or used explicitly but rather used implicitly in the calculations.

For the PB-EKF, the control signal \(i_{cell}\) is denoted by \(u_k\), state is denoted by \(x_k\), the error covariance is \(\sigma_{x,k}\) and \(\sigma_w\) represents the uncertainty of the current sensors. Once the state matrices are computed, the PB-EKF performs the state and error covariance time update based on the previous state, control signal, and error covariance matrix using equations (1) and (2) in Figure 55.

In Stage 3, in order to compute the output prediction, the PB-EKF initially estimates the SOC based on the state time update \(x_k\), and the lithium concentration in the solid surface at the negative electrode \(c_{s^-}\). For our software design, we use the blended cell model parameters from[11]. These parameters are held constant for the blended model temperature range used in our software design. To further reduce data, the constant elements used in calculating the state of charge are combined so that only 5 variables are required. This approach is used throughout the software. The drawback to this approach is the inability to infer where the values originated without the use of comments in the code. As a result, similar to equation (4.7) in Chapter 4, our output SOC prediction computation is performed as follows:
The development of the software architecture closely follows the development described in Chapter 4 of the EKF. Having determined the SOC the second goal of Stage 3 is to compute the standard output equation function using equation (3) in Figure 55. To facilitate, we create the output matrices C and D similar to the interpolation in equation (5.1) using the LUTs derived from the DRA process. The output function, $y_{var,k}$

Each element of the output function, $y_{var,k}$ represents the debiased value of an electrochemical reaction at a given position. The debiased value only represents a physical reaction after preforming and incorporating the non-linear corrections. Hence, in Stage 4, non-linear corrections are performed and predicted output voltage ($y_{volt,k}$) is computed as follows:

$$y_{volt,k} = y_{var,k}[17]FR_{film}^+ - y_{var,k}[15]FR_{film}^- + \varphi + \Delta\eta + u_kR_{term} + \Delta OCV$$  \hfill (5.2)

The above parameters, $FR_{film}^+/-$ and $R_{term}$, (film and terminal resistances), are pre-computed constants, applicable to the temperature range and SOC of the blended model for PBM. The terms $\varphi$ and $\Delta\eta$ are outcomes of non-linear corrections and are computed using the following equations (12) and (13). The final term, $\Delta OCV$, is the open circuit voltage correction based on SOC and calculated using interpolation and LUTs.

$$\varphi = y_k[6] + T_k\varphi_{const} \ln \frac{c_{ek}}{c_{ep}}$$  \hfill (5.3)

$$\Delta\eta = \eta_l - \eta_0 = \frac{T_k2R}{P} \cdot \left( \text{asinh} \left( \frac{y_k[17]}{j_o,pos} \right) - \text{asinh} \left( \frac{y_k[15]}{j_o,neg} \right) \right)$$  \hfill (5.4)

Recognizing that $y_{volt,k}$ is a non-linear output equation for voltage and that we need to be able to represent it in the standard form for use in the Kalman Gain computation. To do this, we will have to create new linearized output C and D matrices. In Stage 5, new linearized $\hat{C}$ and $\hat{D}$ matrices are
created using the elements of $y_{volt,k}$. These matrices can also be used in MPC to facilitate optimizing or constraining an output or solution based on the voltage. As recalled from Chapter 4, $\hat{C}$ and $\hat{D}$ are constructed similarly to $y_{volt,k}$ Considering $\hat{D}$ as an example, $\hat{D}$ is computed as follows:

$$
+ D[17]R_{cell}^+ - D[15]R_{cell}^+ + T_K\varphi_{const}(D[3](1/c_{e,l}) - D[0](1/c_{e,o}))
$$

Noting that each term in this computation is a scalar value, $\hat{D}$ is also a scalar value. $\hat{C}$ is similarly calculated but the original C matrix is a 19x5 matrix, $\hat{C}$ ends up as a 5x1 vector. Further reducing the number of computations for the PB-EKF and PB-MPC. The final effort of Stage 5 is to compute the Kalman Gain (L) and voltage error covariance ($\sigma_y$) using equations (5) and (6), respectively, from Figure 55.

In Stage 6, the PB-EKF computes the state and error covariance measurement update, using equations (7) and (8), respectively from Figure 55. After computing measurement update, PB-EKF is technically completed. However, to support PB-MPC and SOC compensator, EKF repeats Stage 3 to update the SOC for the DC-DC converter and computes the necessary non-linear correction and related output function for the PB MPC in Stage 7.

In Stage 7, PB-EKF computes the derivative OCV value, and linearized C and D output matrices for $\varphi_{s-e}$ using the $x_{k+1}$ state. Thus, $x_{k+1}$ becomes the current-state of PB-MPC. To accomplish this, the following steps are executed

Table 10: Generation of output matrices

<table>
<thead>
<tr>
<th>Side reaction overpotential and supporting Linearized $\hat{C}$ and $\hat{D}$ output matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>$OCP^− = (1 - \beta_{OCP})BigUoCp^−[low] + \beta_{OCP}BigUoCp^−[high]$</td>
</tr>
<tr>
<td>$\varphi_{s-e,1} = y_{var,k}[12] + OCP^−$</td>
</tr>
<tr>
<td>$dOCP^− = (1 - \beta_{OCP})BigDUoCp^−[low] + \beta_{OCP}BigDUoCp^−[high]$</td>
</tr>
<tr>
<td>$\hat{C}<em>{gse} = C[12][l] + \left(dOCP^−/n</em>{s,max}\right)C[8][last]$</td>
</tr>
<tr>
<td>$\hat{D}<em>{gse} = D[12] + \left(dOCP^−/n</em>{s,max}\right)D[8]$</td>
</tr>
</tbody>
</table>
5.3.2 MPC

Recalling that MPC incorporates integral action by operating on the differences between current and next or current and last states. The first step of the MPC is to obtain $\Delta x_{m,k}$ from (5.4). This is done using the standard state equation with the state matrices provided by PB-EKF.

$$x_{m,k+1} = A_m x_{m,k} + B_m u_k$$

Where $x_{m,k}$ is the current state prediction from PB-EKF. The functional flow of our PB-MPC software is as follows:

1. Calculate next state for the state vector based on current state from EKF
2. Calculate change increment for the state vector and control signals
3. Compute $E$ and $F$ from the cost optimization forms
4. Compute the unconstrained solution $\Delta u^0 = E^{-1}F$
5. Build $M$ and $\gamma$ based on $G_{\varphi se}$ and $\Phi_{\varphi se}$ from equation (18)
6. If $M (\Delta u^0) \leq \gamma$ is true, END MPC, do nothing.
7. Else execute Hildreth QP to provide constrained solution increment $\Delta U_k[0]$
8. Update control signal, $i_{\text{gref}}^* = i_{\text{gref}} + \Delta U_k[0]$
9. Final constraint check on $i_{\text{gref}}^*$ to insure it stays within operating bounds.

PB-MPC not only accounts for the integral action but also allows for a non-zero $D$ term. The state variable then becomes $\chi$ as described in Chapter 3 (3.5) where the output term is the $\varphi_{s-e}$ computed by PB-EKF.

PB-MPC computes the next-state using $A$ and $B$ state matrices generated from PB-EKF, and then computes the difference between states, for steps 1 and 2, in the above software functional flow, as follows:

\[
\begin{align*}
\text{for } i=1 \text{ to } N_s: \quad x_{\text{mpc},k+1} &= A[i]x_{k+1} + u_k \\
\text{for } i=1 \text{ to } N_s: \quad \Delta \chi_{\text{mpc},k}[i] &= x_{\text{mpc},k+1}[i] - x_{k+1}[i] \\
\Delta \chi_{\text{mpc},k}[\text{last}] &= \varphi_{s-e} \\
\Delta u_{\text{mpc},k+1} &= i_{\text{gref,}k+1} - i_{\text{gref,}k}
\end{align*}
\]

To compute step 3, it is imperative to understand that the cost function operates on the amplitude of current, instead of the difference between states ($\Delta u_{\text{mpc}}$). This process is performed...
for each timestep up to the prediction horizon. Recalling the cost function from equation (5.3) the
development of 3 and 4 are as follows:

\[
\begin{align*}
E &= 2\Sigma \\
F &= -2\Sigma \Delta u_{mpc,k+1} \\
\Delta u^o &= -F/E
\end{align*}
\]

Technically \(\Delta u^o = \Delta u_{mpc,k+1} = i_{gref,k+1} - i_{gref,k} \) which confirms that \(i_{gref} \) will pass-through unchanged if no constraint violations are present. In PB-MPC, the main constraint is based on \(\varphi_{s-e} \) that causes lithium plating. The output vector for \(\varphi_{s-e} \) is written as follows:

\[
Y_{\varphi_{se}} = G_{\varphi_{se}} \Delta \chi_{mpc,k} + \Phi_{\varphi_{se}} \Delta U_k
\]

To compute step 5, the constraint matrix \(M\) and vector \(\gamma\) are created from \(G\) and \(\Phi\) from equation (5.5). The constraints for the smart sensor are based on the maximum and minimum current amplitude, the maximum and minimum \(\Delta u\) and then the bounds on \(\varphi_{s-e}\). Totaling 6 constraint elements for the two constraint vectors as follows where \(\Sigma\) and \(I\) are both 1.

\[
M = \begin{bmatrix}
\Sigma \\
-\Sigma \\
I \\
-1 \\
\Phi_{\varphi_{se}} \\
\end{bmatrix}
\quad \text{and} \quad
\gamma = \begin{bmatrix}
\Delta \chi_{mpc,k} \\
\Delta u_{max} \\
\Delta u_{min} \\
G_{\varphi_{se}} \Delta \chi - 0.14 \\
\end{bmatrix}
\]

(5.6)

Step 6 executes the inequality. Should a constraint be triggered, the software executes the HQP algorithm as defined in chapter 3,
Table 4. There are different ways to calculate the convergence of the $\lambda$ vectors in HQP. In this and the follow-on hardware implementation, if the ratio of the norm of the current to previous $\lambda$ vectors is very close to 1, the system converges. Theoretically the ratio should equal 1 for convergence but this is unachievable in a discrete, binary, digital system. The approach in Chapter 3 uses $|1 - \|\lambda_k\|/\|\lambda_{k-1}\|| \leq \text{ref}$ whereas [11] uses $\|\lambda_k - \lambda_{k-1}\|/\|\lambda_k\| < \text{ref}$ where the double brackets indicate the Euclidean norm. Both approaches converge when the ratio is close to zero. Our approach has the same reference or tolerance value but requires at least one less computation and has the same result. The final steps (8,9) of PB-MPC are to update the control signals if needed.

5.4 Experiments and Results

We perform experiments to evaluate the feasibility and efficiency of our embedded software architecture for PB-EKF-MPC smart sensor. We also compare our embedded software with PB-MPC/EKF for BMS written in Matlab [11], to evaluate and validate the correctness and functionalities of our design.

Our experiments are carried out on Xilinx V707 board. Our software modules are written in C and executed on 32-bit MicroBlaze processor running at 100MHz on Virtex-7 FPGA using Xilinx Vivado. For embedded software, fundamental operators are designed using single-precision floating-point units (FPU), whereas baseline Matlab [11] used double-precision FPUs. Experiments are based on urban dynamometer driving schedule (UDDS) profile for $I_{\text{string}}$ input. Our PB-EKF and PB-MPC are initialized with a SOC of 0.6 and 0.0, respectively.

5.4.1 Functional Verification

We utilize three main outputs of PB-EKF and PB-MPC, including control signal ($i_{g_{\text{ref}}}$), $\varphi_{ve}$, and SOC, to determine the correct operation of our smart sensor. We compare our embedded software for PB-EKF-MPC to baseline Matlab PB- MPC/EKF in [11]. To execute our embedded PB-EKF-MPC, it is essential to record: cell input current, cell reference voltage for PB-EKF, and input
current $i_{\text{g.ref}}$ for PB-MPC, at a standard UDDS driving cycle of 2000. Firstly, we examine the SOC and control signal $i_{\text{g.ref}}$. As in Figure 56: Control signals $i_{\text{g.ref}}$: proposed embedded software vs. baseline Matlab., our embedded software results are quite close to Matlab results, except for intermittent differences that starts at 1225th iterations and peaks at 1408th iteration. In this case, the standard deviation of error is 512μA, i.e., a percent error of -0.0000032%, which is negligible.

![Figure 56: Control signals $i_{\text{g.ref}}$: proposed embedded software vs. baseline Matlab.](image)

Secondly, we compare the SOC output from PB-EKF, that is typically forwarded to the SOC compensator. The SOC output is a ratio, thus presented as a percentage. As in Figure 57: SOC output: proposed embedded software vs. baseline Matlab., our embedded software overlaps with Matlab. The standard deviation of error between these two is 7.30014E-08. This might be due to single precision versus double precision FPUs used by our embedded software and Matlab, respectively.
Thirdly, we compare the side-reaction overpotential ($\phi_{s-e}$), the key variable to prevent aging and performance degradation. As in Figure 58, our embedded software overlaps with Matlab. The standard deviation of error between these two is $3.26202E-08$, similar to the SOC output in Fig. 4. The $\phi_{s-e}$ does not exhibit any spikes. All these results and analyses illustrate that our proposed embedded software architecture operates correctly.
5.4.2 Performance analysis: execution times, code Size

For our experiments, MicroBlaze processor is configured to have the maximum available cache memory of 128KB. We also set the stack and heap sizes to 20KB each, to execute our embedded software efficiently. Based on the code size in Table 11, our embedded software for PB-EKF-MPC can seamlessly be executed on a 32-bit embedded processor with limited memory.

Table 11: Embedded software code size breakdown

<table>
<thead>
<tr>
<th>Executable code (bytes)</th>
<th>Data (bytes)</th>
<th>Total size (bytes)</th>
<th>Total size (kilobytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60,352</td>
<td>41,023</td>
<td>101,616</td>
<td>99.23KB</td>
</tr>
</tbody>
</table>

In Table II, we report the “worst-case time” (measured with AXI Timer) to execute a single iteration of our embedded software PB-EKF-MPC smart sensor. The “worst-case time” (in column 2) incurs when it is absolutely must to execute HQP, due to a violation of a constraint by the unconstrained solution; thus depends on HQP iteration count (in column 1) which is varied to
achieve an optimal solution. Our highest worst-case time for the highest HQP iterations is 68.2ms (row 1, column 2) to process our embedded software PB-EKF-MPC smart sensor.

We observe that HQP is invoked only 3.7% of the time during first 2000 iterations of our embedded software, with an average execution time (time per iteration) of 4ms. To determine how many battery cells a single processor can support, we utilize worst-case time to ensure ample margin for good performance.

Although the standard interval time for a battery cell is 1s [11],[43], only a portion of this can be utilized to execute our embedded software, since other control algorithms must be executed during this 1s. Table 12 shows the number of battery cells that can be supported for 1s, 0.5s, 0.4s, and 0.3s intervals for varying HQP iterations, in columns 3, 4, 5, and 6, respectively. Hence, if the goal is to support 6 battery cells using one processor, and if only one third of the interval (i.e., 0.3s) is available to execute our embedded software, the maximum HQP iterations would be 300, to balance performance with available resources.

Table 12: Worst-Case Time & No. of Battery Cells Supported per Interval

<table>
<thead>
<tr>
<th>HQP iteration count</th>
<th>Worst-case time (seconds)</th>
<th>No. of battery cells supported per available control interval</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1.0 s interval</td>
</tr>
<tr>
<td>500</td>
<td>0.068213</td>
<td>14</td>
</tr>
<tr>
<td>400</td>
<td>0.055254</td>
<td>18</td>
</tr>
<tr>
<td>300</td>
<td>0.042236</td>
<td>23</td>
</tr>
<tr>
<td>250</td>
<td>0.035809</td>
<td>27</td>
</tr>
<tr>
<td>200</td>
<td>0.029254</td>
<td>34</td>
</tr>
</tbody>
</table>

5.5 Related Work – PB-EKF-MPC

We investigated existing works on embedded software for PB-EKF-MPC in the literature. Most of the existing works focused on Matlab modeling to verify and improve the physics-based (PB) approaches. Our research work is inspired by the Matlab model using DRA approach in [11],[41],[44]. There were a few PB-EKF-MPC software designs utilized for non-battery fields, but none were embedded software designs. In [64], PB-EKF-MPC software tracking system was
proposed for underwater vehicles, which was executed on a 16GB i7 CPU (at 2.9GHz), with an average execution time of 50ms. Conversely, our embedded software design, executed on 128KB embedded processor (at 100MHz), obtains an average time of 4ms.

5.6 Conclusion

In this chapter, we introduced a novel, unique, and efficient embedded software architecture for PB-EKF-MPC smart sensor for li-ion battery cell control, which was executed on an embedded processor with very limited memory. Our embedded software design performed well within the 1s update intervals of BMS when running at only 100MHz. To the best of our knowledge, no similar work exists in the literature that provides embedded software for PB-EKF-MPC, specifically for battery cell control. In the next chapter we create an efficient embedded hardware for PB-EKF-MPC smart sensor.
Chapter 6

EMBEDDED HARDWARE ARCHITECTURE FOR PB MPC
AS A SMART SENSOR

6.1 Architecture Overview

The advantage to using LUTs is the reduced computational burden and fast execution time. One of main drawbacks of an off-line approach such as LUTs is the amount of memory required, another drawback is the limited operating range provided by a static LUT. To increase the operating range either larger LUTs or additional LUTs are required and so the memory burden also increases. This is balanced by the fast performance and reduced computational burden LUTs provide. As we can see from the previous chapter, the program code size, even using LUTs, has utilized roughly 78% of the available memory. With a data size of 40 KB and remaining memory of 29 KB it is unlikely that a second blended model can be implemented to increase the operating range of the smart sensor on the current microprocessor. However, one of the advantages to an FPGA HW-based approach is the ability to scale the memory as needed. Additionally, the speed of the HW allows the FPGA to serve more battery cells than the microcontroller. This may help balance the cost of the more expensive FPGA against needing multiple microcontrollers.

This chapter builds on the theory and approach of the last two chapters to create a hardware architecture for the PB-EKF-MPC. This hardware is designed for the Virtex 7 chip which is an upgrade from the Virtex 6 chip used in chapters 3 and 4. Subsequently a new development environment is required as the previous environment, Xilinx ISE, does not support Virtex 7 development. The new Xilinx Vivado tools are not fully backward compatible with Xilinx ISE and so required adaptation of EKF architecture for it to work on the Virtex 7 chip. The result of the adaptation is an EKF HW design that performs at 100 MHz vs the original 50 MHz, and the work
to adapt the architecture design resulted in bug fixes that improved the overall performance of the
HW EKF from the version presented at the dissertation proposal.

Building off of the hardware flow in Figure 36 of Chapter 3, we again alter the stage 7
module to provide a tailored solution suitable for the smart sensor. This stage now includes the
final non-linear corrections, linearized output matrices and the MPC algorithm. A final stage is
included that enables the hardware accelerator to provide the results to the microcontroller, update
the temperature and input signals and initialize the next control cycle. Recalling the overall
distributed control design from [44], Stage 7 is equivalent to the MPC block shown in Figure 60
for convenience,
Figure 60: Block diagram of BMS utilizing PB-MPC for smart sensor

The hardware design for stages 1 through 6 is almost identical to that in Chapter 4, upgrading to the Virtex board required updating of all the Xilinx intellectual property (IP) floating point cores to the Advanced eXtensible Interface (AXI) IP cores as the previous IP cores were not supported. AXI is a type of microcontroller bus that provides improved design mobility through the use of industry standards and interface protocol standardization, this type of standardization means designers need only learn a single protocol for use with any Xilinx IP core [67]. This switch enabled timing flow changes to allow for a design speed of 100MHz. While the AXI protocol allows for burst transfers and streaming, this design uses AXI4-lite which has a small logic footprint and supports single memory mapped transactions suitable for use with control and status registers which is well suited to our design approach.

6.1.1 Stage 7

Stage 7 is organized hierarchically with Module 6 acting as the top module of this stage, using a finite state machine FSM to coordinate the different actions, typically a top module is simply a control module or multiplexor to interface the sub-modules with the Xilinx IP and each other and does not handle data directly, however in this case the Module 6 FSM also conducts the constraint comparison for determining if HQP is invoked. Module 6 provides access to the system’s 2 dual port RAMS and 2 single port ROMS. Module 6 also contains a number of floating-
point IP cores necessary to execute the operations of MPC. Keeping the arithmetic logic units at
this level of the hierarchy allows the user IP modules to control the data flow and outputs can be
shared amongst modules without adding extra circuitry and interconnects. Stage 7 uses a finite
state machine (FSM) to control the flow of the smart sensor. Internal to stage 7 there 11 states in
the FSM and 10 user designed submodules to execute the steps shown in Figure 59. The FSM of
Stage 7 is shown in Figure 61

Figure 61: States of Stage 7

Following the organization started in Chapter 4, the stage 7 Top module controls the access
to the two dual-port RAMs and two single port ROMS located in the overarching Stage 7 Top module.

6.1.1.1 PHISE

The first state, PHISE has three user designed IP working in parallel. This state and its
corresponding modules execute the initial MPC steps through creating the constraint vectors. The
Xekf_state module is responsible for calculating $\Delta \chi$, $\Phi_{pse}$ and part of $\gamma$ The Phi_se module is
responsible for $\varphi_{S-e}$, $\hat{\chi}_{pse}$, $\hat{D}_{pse}$, $G_{pse}$, M and the rest of $\gamma$ Phi_se_Store is responsible for writing
critical outputs to memory. During the PHISE state, the Phi_se module controls a read port address
of RAM1, RAM0 and ROM1, an FP adder, ADD1, and an FP multiplier, MULT1. The Xekf_state
controls the read port address of ROM 1, ADD2, and MULT2. Finally, Phi_se_store controls the write port addresses of RAM1 and RAM0 and a read port address of ROM0. While the RAMs are true dual port memories, as much as possible the design dedicates one port as a read port and the other as a write port. This is done to reduce the possibility of a race condition or collision.

To reduce the complexity of the multiplexors in Module 6 and to reduce the number of state switches due to memory port availability, Phi_se_Store controls the memory reads for Xekf_state. For example, in the initial operation, Xekf_state executes the state equation using the state and state matrices provided by the EKF. From equation (2.16) we have

\[ x_{k+1} = Ax_k + B\Delta u_k \]

For the hardware design, this reduces to a simple vector multiplication \( A_{diag} \ast x_k \), with \( \Delta u_k \) added to each result. Phi_se_store pulls A from RAM0 and x from RAM1 and the RAM outputs are accessible by all the modules, therefore nothing special is required for Xekf_state to send the A and x values to MULT2 and then ADD2. The addition is automated by tying the add_valid signal to the mult_rdy signal from the multiplier. Automated in this case means, it is set independently of the current FSM state. This is done using a separate always block in Verilog. A single clock cycle delay is added to the mult_rdy signal to allow the multiplier result to be loaded into the input register of the adder. Controls are added so that the tied signals and one cycle delay signal are only active during the operations and are tied to virtual ground otherwise. Because of the parallel nature of hardware designs, if these controls are not implemented, the one-cycle delay signal would be active the entirety of the time the system is running. Since the signal is physically tied to MULT2, anytime MULT2 is accessed by any of the modules, this signal would also be active incurring a power penalty for each change in state between 1 and 0. Adding the controls suppresses switching action, tying the signal to ground and reducing the active power requirement.

Phi_se_Store calls \( x_k \) again, timed so that it is available when the output of ADD2 is rdy, the output of ADD2 and the \(-x_k\) are fed to the input of the ADD2 to achieve the \( \Delta x_k \) terms for \( \Delta \chi_k \).
The last term of $\Delta \chi_k$ is $\varphi_{s-e}$ which was calculated in parallel by the Phi_se module. Phi_se_Store, stored the $\varphi_{s-e}$ value in the memory address that corresponds to the last term of $\Delta \chi_k$ so nothing further is required.

Because the MPC system uses a control horizon of 1 and a prediction horizon of 2 [11], the computations for the PHISE state are fairly straight forward. The gain matrices, $\Phi_{qse}$ and $G_{qse}$ are computed using the following approach:

\begin{align*}
\Phi_{qse}[0] &= \tilde{D}_{qse} \\
\Phi_{qse}[1] &= [\tilde{C}_{qse}|\tilde{D}_{qse}][B|1]^T
\end{align*}

Since B is a vector of 1’s, this becomes the accumulate portion of the multiply accumulate action. $G_{qse}$ uses an augmented $\tilde{C}_{qse}$ matrix, $[\tilde{C}_{qse} \ 1]$.

\begin{align*}
G_{qse,row 0} &= [\tilde{C}_{qse} \ 1] \\
G_{qse,row 1} &= [\tilde{C}_{qse}(A_{diag} + 1)^T \ 1]
\end{align*}

The first row of $G_{qse}$ is another accumulate function, while the order of operations for the second row of $G_{qse}$ is to add 1 to $A_{diag}$ and then multiply-accumulate (MAC) $\tilde{C}_{qse}$ and the result together. The final terms of 1 in each row are simply stored into memory. The Phi_se module calculated $\tilde{C}_{qse}$ and $\tilde{D}_{qse}$ according the Table 10 in Chapter 5. From equation (5.6) shown again below only the first and last two terms of $\gamma$ are calculated. The last two terms of M are simply the two terms in the $\Phi_{qse}$ gain matrix. The remaining terms are constants determined by the operating conditions of the battery.

\[
M = \begin{bmatrix}
\Sigma & -u_{max} - u_{mpc,k} \\
-\Sigma & -u_{min} + u_{mpc,k} \\
I & \Delta u_{max} \\
-I & -\Delta u_{min} \\
\Phi_{qse} & G_{qse}\Delta \chi - 0.14
\end{bmatrix}
\]

The first two terms of $\gamma$ are simple addition/subtraction operations. The final term is a MAC of $G_{qse}$ and $\Delta \chi$. Working within the limitation of memory ports, the two terms of $G_{qse}\Delta \chi$ are
calculated in parallel using the three modules, Phi\_se pulls the $\Delta\chi$ terms and first row of $G_{\phi s e}$ from memory, while Phi\_se\_Store pulls the second row of $G_{\phi s e}$ from memory. Phi\_se executes the MAC of row 1 and Xekf\_state executes the MAC of row 2. The MAC for this design is very similar to the MACx module designed in Chapter 4, Figure 42: Internal architecture for MAC computation, however this design uses a counter set to the length of $\Delta\chi$ and the control signals to load the adder are based on the least significant bit (LSB) of the count value, 0 is even, 1 is odd. Once again, the control signals are generated independently of the FSM, lowering the number of FSM states. The shortness of the vector length and the latency of the adder allow the MAC to be built using a single Multiplier and single adder. Handshaking protocol is used to ensure the three modules operate in parallel.

6.1.1.2 DNP

This state is responsible for $\Delta u$, which simplifies to the difference between the current and previous control signal amplitudes. This state also calculates $K$ from equation (2.35) and $P$ from equation (2.34) shown again below

$$P = ME^{-1}M^T \text{ and } K = \gamma + ME^{-1}F$$

Similar to Chapter 3’s approach a control horizon, $N_c$, of 1 results in a scalar for $E$. $P$ becomes a set of multiplications that are continuously executed for rows 2-6. The operation of the first row of $P$ and $K$ are done in parallel since the term $ME^{-1}$ is shared between the two operations. The inverse of $E$ is stored in memory so that an FP multiplier can be used to compute $ME^{-1}$ rather than the more expensive (in terms of execution time and chip area) divider. $F$ simplifies to $2\Delta u$ when $N_c=1$. The HQP algorithm using the inverse of the terms on the diagonal of the $P$ matrix. The $P$\_inverse module executes in parallel with DNP to calculate the inverse of the diagonal elements as they are computed. A floating-point divider takes 20 clock cycles to complete, executing the inversion at this time, even though it may not be required, reduces stall time later. The $P$\_inverse module sets up the division action and the Phi\_se\_Store module saves the first 4 values to memory while
HP_store handles the final two values. The final two values are only stored if HQP is activated. This saves the 20 clock cycles it would take to wait for the divide results before moving to the next state.

### 6.1.1.3 COMPARE

The compare function is handled by the Top module rather than creating a separate User IP, this saves on the interconnect that would be required for connecting the User IP to the RAM. In this design the address signals are registered at the Module 6 level and are not registered at the Top level. Registering at the Module 6 level results in a 2-cycle latency when reading data from memory if called from Module 6 and a three-cycle latency when called from a User IP such as DNP. Thus the SET_D state sets the correct address, the CALL_D state waits a clock cycle for the result to be on the RAM output, the LOAD_D state registers the RAM output into a temporary register and the CHECK_D state uses the MSB of the temporary register to determine if a constraint is violated. This takes advantage of the hardware convention that 1 = true and 0 = false and the IEEE floating point standard where the MSB is used to indicate the sign of the value. If the MSB =1, the value is a negative, in Verilog this is done by

```verilog
CHECK_D:begin
    if(temp1_reg[31])begin
        start_HP = 1'b1;
        case_count = 2'h2;// set counter back to zero
        next_state = HP;
    end
    else if(count==4'h6)begin
        start_newDU = 1'b1;
        next_state = NEW_UK;
    end
    else next_state = CHECK_D;
end
```

The state walks through all 6 terms of K if there are no negative values and upon completion the FSM transitions to NEW_UK, otherwise, as soon as the first negative value appears the FSM transitions to the HQP state.
6.1.1.4 HQP

As was the case with the PB-MPC software design, we had hoped to reuse the HQP modules from the original ECM design in Chapter 3. However, the switch to Virtex 7 and the differences in memory structures between the ECM hardware design and the PB-EKF hardware design made this impractical. Additionally, the HQP in the ECM hardware was designed to handle a much larger P matrix and so would be more powerful than necessary for this application. In the ECM design the P matrix had 1024 terms and its own RAM, in this application the P matrix has 36 terms and can fit in the available RAMS. Upon entering the HP state, the FSM starts the HQP modules.

HQP has four associated modules that run in parallel. HP_store handles all the memory writes for HQP. HP_Lambda handles the computation of the $\lambda$ elements from equation (2.40)

$$w = K_i - P_{ii} \lambda_i$$  \hspace{1cm} (6.5)

The third module, Lambda_final, operates in parallel with HP_Lambda and executes the rest of equation (2.40) as follows

$$w = w P_{jj}^{-1}$$  \hspace{1cm} (6.6)

HP_Store stores the result of equation (6.6) in a temporary register. The temporary register exists at the Module 6 level, Module 6 implements a control signal, lambda_choice, that is set to the MSB of the temp register. Similar to the compare in section 6.1.1.3, HP_Store uses the lambda_choice signal to determine if the element, $\lambda_i$, is 0 or the result of equation (6.6), stores the appropriate value in memory and signals to Lambda_final that the correct value is available. Lambda_final sets up the multiplier to square $\lambda_i$ and then waits for the next element, after six elements Lambda_final finishes.

The fourth module, HP_Length, is responsible for calculating the Euclidean Norm of $\lambda$ for the current state by summing the squares from Lambda_Final and taking the square root of the
result. Lambda_Final is also responsible for inverting the norm from the previous state so that the ratio of \( \frac{\|\lambda_k\|}{\|\lambda_{k-1}\|} \) can be assessed for convergence. For this design convergence occurs when

\[
0.9999 \leq \frac{\|\lambda_k\|}{\|\lambda_{k-1}\|} \leq 1.0001
\]

Because the divisor is the norm of the previous \( \lambda \) is 0 for the first iteration, control logic is in place that indicates when the first loop occurs so that there is no chance of a divide by zero operation. The ratio could be reversed, which would also prevent a divide by zero occurrence, however it does not save any iteration cycles as the ratio will be zero for the first iteration and not meet the criteria.

Lambda_length sets the “converged” signal high if the ratio meets the criteria. This signal goes to all four modules to stop operation and clear necessary states, as well as to Module 6.

While Lambda_length is summing the \( \lambda_i^2 \) terms for the norm, it is also computing the correction factor for updating the control signal. As in the ECM design, this value is calculated each iteration and discarded if the iteration does not converge. the Module 6 level the FSM moves from HP to HP_LOOP upon completion of Lambda_length.

The HP_LOOP state evaluates if the “converged” signal is high or if the maximum number of iterations has been reached. If neither of those are true, the counter is incremented and the FSM transitions back to the HP state. If either are true, the FSM transitions to NEW_UK and sets a “New_DU” signal high to indicate the correction factor is necessary.

6.1.1.5 NEW_UK

When entering the NEW_UK state, the FSM starts the final module, NewDeltaU. As shown in Figure 60, the cell current is comprised of \( I_{\text{string}} + i_g \). This module is responsible for updating the control signal \( u_k \) where \( u_k \) represents \( i_{\text{ref}} \). If the control on the current, \( i_{\text{ref}} \) developed by the smart sensor is insufficient to prevent degradation of the battery cell then the system acts to reduce \( i_g \) to zero so that \( i_{\text{cell}} \) is limited to \( I_{\text{string}} \). \( I_{\text{string}} \) is controlled by the higher-level energy management system and is not subject to the smart sensor, leaving limiting \( i_{\text{ref}} \) as

149
the only option. NewDeltaU evaluates the control signal “New_DU” to determine if the FSM executes the states that add the correction factor to $\Delta u^\circ$ or if those states are bypassed. NewDeltaU then adds $\Delta u^\circ$ to $u_{k-1}$ to update the control signal. If New_DU = 0, the new control signal is equal to the original input, $i_{\text{gref}}$. Otherwise the control signal is equal to $i_{\text{gref}}$ plus the correction factor. A final set of comparisons is made to ensure the new control signal is sufficiently limited based on the minimum and maximum allowed $I_{\text{string}}$. The control signal prior to this last compare is the one stored in memory to act as $u_{k-1}$ while the output of the comparisons becomes $i^*_{\text{gref}}$. Upon completion of NEW_UK, the FSM transitions to FINISH and sends a control signal to the Top module that Module 6 is complete.

6.1.2 Stage 8

This stage consists of a short FSM that loads the measured temperature into RAM for use in the next iteration. Theoretically the microcontroller that interfaces with this hardware accelerator would measure the temperature and provide the value via AXI-Lite protocol. Stage 8 also allows the microcontroller to access the 4 memories as needed. In an implementation, the access would be set to the control signal $i^*_{\text{gref}}$, and the SOC. However, for purposes of evaluation and troubleshooting it is convenient to be able to access the intermediate values stored by the HW design. When the microcontroller is finished accessing the memory, it sets the signal “end_test” high. When that occurs Stage 8 transitions to the idle state in the top module to wait for the next start signal, indicating another control interval has begun.

6.2 Experiments and Results

We perform experiments to evaluate the functionality and performance of our proposed FPGA-based hardware accelerator for the physics based MPC smart sensor designed to support the distributed battery management system in Figure 60. We use the baseline model of the physics-based MPC smart sensor written in Matlab [11] to evaluate and validate the correctness and
functionality of our proposed design. Additionally we have the embedded software architecture for the physics based MPC designed in chapter 5 and a desktop C program that has been used to develop the architectures in chapters 5 and 6, the desktop C program is useful for running tradeoff scenarios to determine the most efficient architecture that has the correct functionality. These programs are available in the appendices of this document. These desktop results are compared and verified with the baseline Matlab results as well as with our embedded hardware/software results.

All our experiments for this hardware design are carried out on the Xilinx VC707 development board, which utilizes a Virtex 7 XC7VX485TFFG1761-2 device [68]. The development board has numerous peripherals that could be used to support a physical test of the embedded hardware accelerator. The development board is shown in Figure 62

![VC707 Development board](Xilinx)

Figure 62: VC707 Development board [Xilinx]

All the User IP hardware modules are designed in verilog, using Xilinx IP cores that are mixed VHDL and Verilog. The Vivado design tools are used to create the user IP. The Vivado environment has an integrated simulator that was used to verify the functionality of the design and the bit and signal level prior to programming the FPGA. Our embedded software module that interfaces with the HW accelerator is written in C and executed on a 32-bit 128kB MicroBlaze soft-
processor (running at 100MHz) on the same FPGA as the embedded software in Chapter 5. The hardware is designed to support single precision floating point values in contrast to the double precision capability of the baseline Matlab model [11]. The experiments are again performed using an Urban Dynamometer Driving Schedule (UDDS) [61] current profile for the input \( (u_k) \) and the measured output \( (v_k) \). The physics-based EKF in this design is again initialized using a SOC of 0.6 and an initial state covariance of diagonal matrix \([1E5, 1E5, 1E7, 1E7, 5E9]\) [11]. The smart sensor designed to support the normal driving of HEV with the charging and discharging based on the UDDS driving profile [11].

In order to test the HW architecture on the FPGA a MicroBlaze microcontroller is required to access the memory registers and control the smart sensor, additionally a timing unit is required to measure the timing of the architecture and a UART module is required to communicate the values to the tester. The MicroBlaze also implements a clock wizard to provide clock signals to all the modules, an AXI peripheral module to support communication between the modules and a Debug module to allow inline access to the microblaze to conduct testing. The MPC smart sensor HW architecture is represented by the orange box and is named MPC_SS_FULL_0. This architecture is built using Vivado’s IP Integrator tool. The resulting block diagram of the design is generated by Vivado and is shown in Figure 63.
Figure 63: Complete architecture diagram for the MPC smart sensor HW design

6.2.1 Functional Verification

Similar to the functionality experiments in Chapter 5, the output of the HW accelerator is compared to the Matlab baseline, holding the Matlab baseline as “truth”. We first examine the performance of the HW smart sensor for the $i_{g\text{ref}}$ control signal. Figure 64 shows that the FPGA design functions as desired and Figure 65 shows the error as the difference between the Matlab performance and the embedded HW performance. The maximum error occurs in the first 16 cycles and is $-1.25mA$ resulting in a maximum %error of -0.018%. The mean error is $-2.62\mu A$ and the standard deviation is $59.1 \mu A$. 

153
Figure 64: Baseline comparison for the control current $i_{gref}^*$

Figure 65: Difference between Matlab and embedded HW for the control current.
Examining the performance shown in Figure 66 and the error shown in Figure 67, the proposed HW architecture predicts the SOC almost as well as Matlab, the maximum error is $2.24 \times 10^{-7}$ with a mean error of $4.12 \times 10^{-9}$ and standard deviation of $6.67 \times 10^{-8}$.
Thirdly, we compare the side-reaction overpotential ($\phi_{s-e}$), the key variable to prevent aging and performance degradation. The overall performance is shown in Figure 68. The error shown in Figure 69 is fairly consistent across the test interval and is likely due to the use of single point precision vs double point precision. The statistics support this with a mean error of $-6.53\times10^{-9}$ and a standard deviation of $3.35\times10^{-8}$. These results demonstrate that our design is functioning correctly.
6.2.2 Performance Analysis: Execution Time and Speedup

For the smart sensor timing evaluation, we compare the results of the embedded HW and embedded software. The baseline Matlab design includes the DC-DC component of the distributed BMS and thus is unsuitable for a timing comparison as the results would give inflated values. Similar to the ECM MPC designs, the total worst-case time to execute an iteration of the physics-based Smart sensor occurs when the HQP algorithm is activated and fails to converge. We use that value for comparison and is shown in Table 13. Both embedded architectures are executed at 100 MHz on the VC707 board.

From Table 8, considering total execution time, our embedded hardware (at 100MHz) for the physics based smart sensor is 57.96 times faster than the embedded software design, running on a MicroBlaze processor. From these results and analysis, it is evident that utilizing FPGA-based hardware accelerators can achieve significant speedup compared to the software-based designs. With these speedups, our proposed hardware should be able to monitor and control multiple battery cells individually. This type of speed up also indicates that the FPGA could accommodate a more computationally intense implementation of the physics-based model and still perform within the required timeframe. This worst-case time is due to the 500 iterations required for HQP. This is an area of further exploration to determine when the best return on investment in terms of performance improvement vs. number of HQP iterations. The MPC smart sensor without HQP activation executes 20.8 $\mu$s leaving a large margin for improvement. The other opportunity for improvement would be to condense the iteration time of the HQP, using faster IP cores or a more efficient design.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Embedded SW (100MHz)</th>
<th>Embedded HW (100 MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst-case Execution (ms)</td>
<td>68.213</td>
<td>1.18</td>
</tr>
<tr>
<td>Speedup over embedded SW</td>
<td>--</td>
<td>57.96</td>
</tr>
</tbody>
</table>
6.2.3 Resource Utilization

The cost analysis on space is carried out to examine the area-efficiency of our embedded hardware designs. The resource utilization for our embedded hardware design presented in Table 14 is based on the full implementation architecture shown in Figure 63. This analysis is similar to the space analysis in Chapters 3 and 5. As illustrated, the total number of occupied slices, total number of BRAMs, and total number of DSP slices required for embedded hardware design are 7825, 42, and 117 respectively. As demonstrated in Chapter 3, most of the area-efficient embedded hardware architectures with the system-level designs, typically occupy similar or more area on chip.

Table 14: Resource utilization comparison: embedded hardware

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Occupied Slices</th>
<th>Slice Registers</th>
<th>LUTs</th>
<th>BRAMS RAMB36E1</th>
<th>DSP48E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB MPC Smart sensor embedded HW</td>
<td>7825</td>
<td>23004</td>
<td>21930</td>
<td>42</td>
<td>117</td>
</tr>
<tr>
<td>PB-EKF embedded HW (Ch.4)</td>
<td>10702</td>
<td>19501</td>
<td>32169</td>
<td>40</td>
<td>49</td>
</tr>
</tbody>
</table>

Contrasting this resource utilization with that done for the PB EKF in Chapter 4, it would be expected that this design would occupy more slices on the chip than the EKF only design, which, from Table 14 occupies 10702 Slices. While we would like to contribute this improvement to the architecture, it is more likely, considering the MPC design is added to the EKF design, that the following two things are responsible for the reduction in slice resources. First, and most likely, the Vivado tools are the most advanced tools provided by Xilinx, it can be inferred, since the Vivado and ISE are not truly compatible, that Xilinx has implemented quite a few improvements to their syntheses and implementation algorithms for hardware placement and routing. Second, is that the move to the Virtex 7 chip required us to replace all of the native Xilinx IP cores supported by the Virtex 6 chip and the Xilinx ISE development environment with the AXI IP cores supported by Virtex 7 and the Xilinx Vivado development environment, this action may have reduced the
interconnect resource requirements. The adaptation of the AXI Xilinx IP cores and the additional optimization for latency is likely responsible for the increase in DSP48E1 slices from 49 to 117 and partly responsible for the reduction in registers and LUTs. The DSP48E1 slice is a custom logic arrangement, separate from slice registers and LUTs, that is optimized to support digital signal processing and arithmetic functions. In this version of the EKF architecture we took the opportunity to optimize the cores for low latency to ensure timing closure for the design, this often resulted in a core implementation that uses the maximum possible number of DSP48E1 slices for that core. The stage 7 design also incorporated 9 additional floating-point IP cores into the overall design. The timing closure consideration enables the design to achieve an operating speed of 100Mhz. Should timing closure fail, the easy remedy is to slow the clock speed to allow more time for signals to set up and hold. We used this remedy in the Chapter 4 PB EKF design to ensure correct operation.

6.3 Conclusion

In this chapter, we introduced a unique, efficient and functional embedded hardware architecture for a PB-EKF-MPC smart sensor for li-ion battery cell control. Our embedded hardware design performs well within the required 1s update intervals of BMS when running at only 100MHz. The use of the MPC as a smart sensor is a logical step forward in battery control, the increase in resource utilization from the EKF only to the EKF and MPC smart sensor was minimal, To the best of our knowledge, no similar work exists in the literature that provides a functional embedded hardware architecture for a physics-based MPC smart sensor specifically for battery cell control.
Chapter 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This dissertation describes the development of several embedded architectures that support the design and implementation of advanced estimation and control algorithms for battery cell management systems. It is anticipated that these architectures will enable increased adoption of advanced control methods to improve both battery performance and increase the life expectancy of the battery.

The architectures proposed and developed in this work prove that advanced controls and advanced models can indeed be utilized on resource constrained systems with very little performance loss compared to the baseline theoretical models. In Chapter 3 we introduced architectures that explored the use of the Near Field and Far Field method of reducing the required prediction horizon while retaining the benefits of a long prediction horizon. A smaller prediction horizon reduces the number of computations required to achieve an optimal control signal that respects the constraints of the system. Using the dual optimization equation of the Primal Dual optimization algorithm and only addressing the active constraints, was another technique that successfully reduced both computational complexity and execution time for the ECM MPC fast charge approach. The embedded architectures in Chapter 3 also showed excellent performance with very small loss of fidelity.

In chapter 4 we introduce an embedded systems architecture for an Extended Kalman filter using a physics-based model of a battery cell validating the suitability of the ROM proposed by [7],[11] is suitable for embedded systems. This observer can accurately predict the key internal electrochemical states of the cell along with the state of charge within 36μs. Building on the capability of the physics based EKF, we introduce embedded software and hardware architectures
for an MPC smart sensor. The smart sensor approach enables a distributed BMS system for overall management that depends on the smart sensor to ensure the battery cell is operated safely and not operated in a way that would degrade its capability or shorten the expected life of the battery. The smart sensor is a computationally light implementation of MPC that capitalizes on the ability to constrain the control while optimizing performance. The MPC relies on the physics based EKF to provide not only the current state of the parameters of interest, such as SOC or side-reaction overpotential, but also provides the state and output matrices, reducing the need for additional computation overhead. The embedded software architecture of the smart sensor provides an outcome within 68 ms. This is on a single core bare bones microcontroller system.

These architectures are just the beginning of what embedded systems can support. The smart sensor utilizes only 77% of the capacity of a 32-bit 128KB microcontroller. The EKF alone, calculating all the possible physics-based parameters, requires about 70% of the microcontroller capacity.

The proposed research is a natural extension of the work done to mature PBM for use in embedded systems. To my knowledge, except for the work done by T. Lee, these are the first embedded architectures exploring PBM and MPC. Creation of these architectures provides further information that will refine and improve physics-based models and physics-based model predictive control algorithms.

7.2 Future Work

With the drive to increase available battery capacity and performance while preserving safety and reducing degradation, researchers are exploring the viability of using physics-based models and model-based control strategies to improve battery control systems. Batteries are very susceptible to excessive usage and may degrade more significantly and swiftly than other components [69] as such it would be of great benefit to be able to control the battery based on the
mechanisms that contribute to and prevent degradation. As stated previously, the models and algorithms most suited towards these objectives are also the most computationally expensive. Research is then being done to reduce the complexity of the physics-based models while retaining the insight provided by the models with the goal towards embedded systems implementation [41], [48], [70]. While the single particle approach to the physics based model showed promise, it is only valid under low current operating conditions and is not suitable for fast charge conditions [71], most researchers have moved to the pseudo two-dimensional (P2D) model that is represented by the 5 equations in Chapter 2. Zou explores using a time scale approach to separate the degradation process from the electrical thermal dynamics to incorporate thermal behavior into an optimal fast-charge approach [72] using MPC to incorporate the constraints. Suthar et al explores using reformulated models to represent the battery and create an on-line approach for MPC [73]. The initial reformulated isothermal P2D model is composed of 25 equations that required the use a specialized optimization program to solve the simultaneous optimization problem for the control signal. While on-line solutions provide flexibility, this approach had a computation time of almost a minute and is not yet suitable for the one second sampling interval most often used for battery cell management [73]. Deng et al proposes a polynomial approximation approach for the P2D that supports shows some promise with an on-line execution time of 2ms however there are some non-negligible errors that require further study[70]. With the maturation of state-space representation of PBMs by [7],[9],[41],[48] and their research on appropriate and suitable control algorithm approaches [11],[44] it is possible to explore embedded architectures targeted towards microcontrollers and field programmable gate arrays using their PBM. These architectures will leverage the work done creating the embedded PBM observers from chapter 4 and the MPC approaches from [11].

Having proven that the reduced order physics-based model developed in [7],[9],[41],[48] and refined by [11] is suitable for use in both embedded hardware and embedded software systems, the next challenge is to expand the utility of the physics based model either by improving the
model’s fidelity or improving the model’s flexibility. If we focus on future work for embedded software, future exploration on improvements to the physics based blended model [11],[74] incorporating online techniques to reduce the data memory burden on the microcontrollers. The PB smart sensor currently utilizes about 77% of the 32-bit 128KB Microcontrollers available memory space. Given that the current program memory for the physic-based embedded hardware is 58.9KB and the data is 40.1KB it is apparent that there is room for increasing the computational complexity which will translate to an increase in program memory as more commands will need to be executed, and decrease the data memory if we can reduce the look up tables. The experience gained creating the software and hardware architectures in Chapters 3 and 4 increase the likelihood of success for an embedded model predictive control architecture supporting a physics-based model.

While the MPC approach utilized in Chapter 3 uses the near-field/far-field approach to reduce the prediction horizon while maintaining stability, the MPC proposed in [11] uses a method called horizon points to achieve the goals of stability and reduced complexity. Exploration in comparing the two approaches could help guide future research in physics-based MPC. A full review of the horizon points approach can be found in [11]. Briefly, the horizon-points method shortens the necessary prediction horizon calculation by choosing a long horizon and calculating the sequence of predicted output values out of order. For example for an NP = 101, the sequence of predicted output values would be $Y_k = [y_k, y_{k+1}, y_{k+2}, \ldots, y_{k+100}, y_{k+101}]^T$ If it is possible to calculate the sequence out of order, it would be possible to include a long horizon term without calculating all the intermediate terms. This would mean we could use 5 terms to get the same benefit as a prediction horizon of 101 terms without the extra calculations. From [11] the output terms would become

$$Y_k = [y_k \ y_{k+1} \ y_{k+2} \ y_{k+3} \ y_{k+101}]^T$$

And the final output equation would look like
Florentino in [11] runs experiments to show that this method achieves the stability performance of a long prediction horizon without the extra computation. Using this approach it is possible to use an $N_P$ of 2 and get the benefits of a much longer prediction horizon.

Given the goal to improve the performance of the physics based model while reducing the complexity, while look up tables provide the simplicity, they are limited in the depth or breadth of use. Techniques that could improve flexibility would be the use of polynomial profiles or parabolic profiles to allow estimation of a parameter online[75]. Another approach to simplifying the parameter estimation would be to combine the ROM with machine learning techniques in a hybrid approach that could result in machine learning produced equations that estimate the parameters rather than using look up tables. [75]

Another area of future work builds on the work in this dissertation and the blended model approach proposed in [11],[74] and capitalizes on the FPGA capability to tailor memory capacity. The current blended model is created for a single SOC setpoint (in our case 60%) and a range of temperatures. The lookup tables have been split into two physical memory blocks, one block holds the tables for 0°C and the other memory block holds the tables for 25°C. In order to reduce memory requirements it would be of interest to interest to create a blended model for 60% SOC and temperatures of 25°C and 50°C. it may be possible to implement this using only 3 memory blocks, where the memory block for 25°C is shared. The hardware would operate exactly the same as it currently does and would just shift its memory pointers to point at the correct two physical memories needed, depending on temperature. This can be done with multiplexors where all memories are on the chip all the time, or by dynamic reconfiguration where the chip is reconfigured on the spot and only includes the memory necessary. It may even be possible to execute this in software if only one more set of look up tables and constants is required, given that two sets of look
up tables uses 40KB of memory, it can be inferred that adding one more set will only add 20KB, bringing the program to the edge of the 128KB capacity.

Also, as future work, we are planning to further optimize our FPGA-based accelerators/architectures for MPC algorithms in terms of area, speedup, and accuracy. In this regard, we will investigate and integrate parallel processing architectures, similar to [57],[98],[99], to process certain computations of MPC in parallel (in addition to the existing ones), in order to enhance the speedup. We will also investigate ways to incorporate partial and dynamic reconfiguration features (as stated in [101],[102]) and HDL code optimization techniques (as stated in [103]) into our FPGA-based accelerators/architectures to further enhance the area-efficiency and flexibility, similar to our works in [85],[93],[104],[105],[106],[107]. In addition, we are investigating ways to integrate our multi-ported memory architectures, including [60],[108],[109],[110], to facilitate the parallel processing modules in our proposed MPC hardware designs, to further enhance the speedup.
BIBLIOGRAPHY


[76] "Automatic Deployment of Industrial Embedded Model Predictive Control".


[92] A. Alkamil and D.G. Perera, “Efficient FPGA-Based Reconfigurable Accelerators for SIMON Cryptographic Algorithm on Embedded Platforms”, in Proceedings of the IEEE International Conferences on Reconfigurable Computing and FPGAs, (ReConFig’19), 8-page manuscript, Cancun, Mexico, December 2019.


