Madsen and Perera *EURASIP Journal on Embedded Systems* (2018) 2018:2
https://doi.org/10.1186/s13639-018-0084-3

EURASIP Journal on Embedded Systems

RESEARCH Open Access

Efficient embedded architectures for fast-charge model predictive controller for battery cell management in electric vehicles

Anne K. Madsen and Darshika G. Perera*

Abstract

With the ever-growing concerns about carbon emissions and air pollution throughout the world, electric vehicles (EVs) are one of the most viable options for clean transportation. EVs are typically powered by a battery pack such as lithium-ion, which is created from a large number of individual 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. Model predictive controller (MPC) is considered as a feasible technique for cell-level monitoring and controlling of the battery packs. For instance, the fast-charge MPC algorithm keeps the Li-ion battery cell within its optimal operating parameters while reducing the charging time. In this case, the fast-charge MPC algorithm should be executed on an embedded platform mounted on an individual cell; however, the existing algorithm for this technique is designed for general-purpose computing. In this research work, we introduce novel, unique, and efficient embedded hardware and software architectures for the fast-charge MPC algorithm, considering the constraints and requirements associated with the embedded devices. We create two unique hardware versions: register-based and memory-based. Experiments are performed to evaluate and illustrate the feasibility and efficiency of our proposed embedded architectures. Our embedded architectures are generic, parameterized, and scalable. Our hardware designs achieved 100 times speedup compared to its software counterparts.

Keywords: Embedded architectures, Model predictive control, FPGAs, Hardware accelerators, Electric vehicles, Battery cell management

1 Introduction

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 initiatives, both from the government and the private sector 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.

The electric vehicles (EVs) are often powered by energy storage systems such as battery packs, fuel cells, capacitors, super capacitors, and combinations of the above. From the aforementioned energy storage systems, lithium-ion (Li-ion) battery packs are widely employed in EVs mainly because of their light weight, long life, and high energy density traits [2]. In this case, the 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

* Correspondence: darshika.perera@uccs.edu
Department of Electrical and Computer Engineering, University of Colorado at Colorado Springs, 1420 Austin Bluffs Parkway, Colorado Springs, CO 80918, USA

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 [3]. 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 [2], 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 controller (MPC) has been investigated as a viable technique for cell-level monitoring and controlling of the battery packs [3]. MPC 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 non-resource-constrained environments; however, in recent years, this algorithm is gaining interest in the resource-constrained environments, including cyber-physical systems and hybrid automotive fuel cells [3], to name a few. 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. From these, ECM models are more popular due to their simplicity. In [3], 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. Thus far, this fast-charge MPC algorithm has been designed and developed in Matlab and executed on a desktop computer [3]. However, in a real-world scenario, it is imperative to execute this fast-charge MPC algorithm on an embedded platform mounted on an individual cell, in order to utilize this algorithm to monitor and control the individual cells in a battery pack.

Since the existing algorithm for the fast-charge MPC is designed for general-purpose computers such as desktops [3, 4], it cannot be executed directly on embedded platforms, in its current form. Furthermore, embedded devices have many constraints, including stringent area and power limitations, lower cost and time-to-market requirements, and high-speed performance. Hence, it is crucial to modify the existing algorithm significantly in order to satisfy the requirements and constraints associated with the embedded devices.

Although MPC is becoming popular, the measure-predict-optimize-apply cycle [5] of the MPC algorithm is compute-intensive and requires a significant amount of resources including processing power and memory resources (to store data and results). In this case, the smaller the control and sampling interval (or time), the larger the resource cost. This sheer amount of resource cost also impacts the feasibility and efficiency of designing and developing the MPC algorithms on embedded platforms.

We investigated the existing research works on MPC algorithms, as well as the existing research works on embedded systems designs for MPC algorithms in the literature. Most of the research on discrete linearized state-space MPC 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 works on online MPC methods include fast gradient [6, 7], active set [8–10], interior point [11–16], Newton's method [9, 17, 18], and Hildreth's QP [19], and others [20]. In [21], a faster online MPC was achieved by combing several techniques such as explicit MPC, primal barrier interior point method, warm start, and Newton's method. In [9, 18], the logarithmic number system (LNS)-based MPC was designed on a field-programmable gate array (FPGA) to produce integer-like simplicity. The existing research works on embedded systems designs for MPC algorithm focused on FPGAs [8, 11, 12, 17, 22, 23], system-on-chip [9, 16], programmable logic controllers (PLC) [24], and embedded microprocessors [25]. Although there were interesting MPC algorithms/designs among the existing research works, none of the aforementioned existing works were suitable for monitoring and controlling individual cells of the battery pack. For instance, the above existing MPC algorithms/designs did not consist of the feed-through term required by the battery cell model introduced with fast-charge MPC algorithm in [3]. The impact of the feed-through term is discussed in detail in Section 2.

In this research work, our main objective is to create unique, novel, and efficient embedded hardware and software architectures for the fast-charge MPC algorithm (with an input feed-through term) to monitor and control individual battery cells, considering the constraints associated with the embedded devices. For the embedded software architectures, it is essential to investigate and create lean code that would fit into an embedded microprocessor. Apart from the embedded software architectures, we decide to create novel customized hardware architectures for the fast-charge MPC

algorithm (with an input feed-through term) on an embedded platform. 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. In this paper, we make the following contributions:

- We introduce unique, novel, and efficient embedded architectures (both hardware and software) for the fast-charge MPC algorithm. Our architectures are generic, parameterized, and scalable; hence, without changing the internal architectures, our designs can be used for any control systems applications that employ similar MPC algorithms with varying parameters and can be executed in different platforms.
- Our proposed architectures can also be utilized to control the charging of multiple battery cells individually, in a time-multiplexed fashion, thus significantly reducing the hardware resources required for BMS.
- We propose two different hardware versions (HW_v1 and HW_v2). With register-based 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. With Block Random Access Memory (BRAM)-based HW_v2, an optimized architecture is introduced to address certain issues that have arisen with HW_v1, by employing BRAMs to store the data/results. These two hardware versions can be used in different scenarios, depending on the requirements of the application.
- With both hardware versions, we introduce novel and unique sub-modules, including multiply-and-accumulate (MAC) modules that are capable of processing matrices of varying sizes, and distinguishing and handling the sparse versus dense matrices, to reduce the execution time. These sub-modules further enhance the speedup and area-efficiency of the overall fast-charge MPC algorithm.
- Considering the existing works on embedded designs for MPC, our architectures are the only designs (in the published literature) that support a non-zero feed-through term for instantaneous feedback. We perform experiments to evaluate the feasibility and efficiency of our embedded designs and to analyze the trade-offs associated including the speed versus space. Experimental results are obtained in real time while the designs are actually running on the FPGA.

This paper is organized as follows: In Section 2, we discuss and present the background of MPC, including the main stages of the fast-charge MPC algorithm. Our design approach and development platform are presented in Section 3. In Section 4, we detail the internal architectures of our proposed embedded software design and our proposed register-based and memory-based embedded hardware designs. Our experimental results and analysis are reported in Section 5. In Section 6, we summarize our work and discuss future directions.

2 Background: model predictive controller

The model predictive controller (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 attuned 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 investigation on the existing MPC algorithms revealed that the MPC design in [3] provides a simple, robust, and efficient algorithm for the fast charging of lithium-ion battery cells. Hence, this MPC algorithm [3] could potentially be 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, i.e., to use the dual-mode MPC technique and Hildreth's quadratic programming technique [26].

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 [26]. The application of this technique to the fast charge of batteries with a feed-through term is detailed in [26]. As discussed in [26], reducing the prediction horizon dramatically reduces the size of the matrices utilized in MPC, which in turn reduces the computation complexity. Trimboli's group, in [3, 26], 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 1 and 10, respectively.

Hildreth's quadratic programming (HQP) technique is an iterative process that is deemed suitable for the embedded systems designs [27]. 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 managing poorly conditioned matrices and (2) the computations are run on scalars instead of matrices, thus reducing the computation complexity [27]. With the HQP, the intention of the MPC is to bring the battery cell to a fully charged position with the least amount of time. In order to reduce the computational effort [3], 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 case, if the solution fails to converge in the allotted iterations [24]. A recent study [24] revealed that the HQP technique performed faster than the commercial solvers, and it required lean code. However, the main drawback is that it tends to provide the sub-optimal solution more often and is also dependent on selecting the optimal number of iterations. In this study [24], 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 MPC algorithms can be 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.

2.1 Dynamic model

With the MPC algorithm, selecting a suitable model is imperative, since the prediction performance depends on how well the dynamics of the system are represented by the model [28]. For the fast charge of Li-ion batteries in [3], the authors employed an equivalent circuit model (ECM) instead of a physics-based model. The latter models are typically more computationally complex compared to the former models [3]. The sheer simplicity of the ECM leads to a dynamic model that provides a suitable MPC performance for many applications. The ECM model is shown in Fig. 1, and the design and development of the model is detailed in [4, 26].

As illustrated in Fig. 1, the series resistor R_0 is the instantaneous response ohmic resistance, when a load is connected to the circuit. In the ECM model, the R_0 represents the feed-through term in the MPC general state-space Eq. (3) [3, 4, 26]; 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). In this case, the relationship between SOC and OCV is non-linear; thus, it can be implemented as a look-up-table (LUT).

The ECM model has a single control input (i.e., the current) and two measured (or computed) 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 current $i(t)$, which is the control input signal, is represented in the state-space equations as $u(k)$. By employing the MPC algorithm, our intention is to find the best control input, $i(t)$, in order to produce the fastest charge, while considering the physical constraints of the cell. Typically, the parameters or the elements of the ECM model are temperature dependent.

The creation of our unique and efficient embedded architectures for the MPC algorithm is inspired by and based on the MPC algorithms presented in [3, 4, 26–28], with many modifications to cater to the embedded platforms. The feed-through term and dual-mode adjustments are inspired by and based on the ones in [3, 4, 26].

The state-space equations for the ECM model are designed and developed based on Fig. 1. The physical parameters of the model are $Q_{(charge)}$, R_0, R_1, and $\tau = R_1C_1$. In this case, the unaugmented state variables are considered as the $z(t)$, which is the state of charge (SOC) of the open circuit voltage (OCV) and the $v_{C1}(t)$, 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

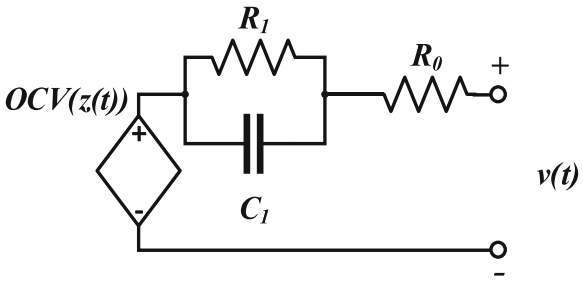

Fig. 1 Equivalent circuit model (ECM) for battery cell charging [3]

state-space variables are Z_k, $v_{C1,k}$, v_k, and u_k. The general state-space Eq. (1) is presented below:

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

Considering Fig. 1, where Δt is the sampling time and η is the cell efficiency, the model without augmentation [4] is written with the following Eq. (2):

$$\begin{bmatrix} z_{k+1} \\ v_{C_1,k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & e^{-\Delta t/R_1 C_1} \end{bmatrix} \begin{bmatrix} z_k \\ v_{C_1,k} \end{bmatrix} + \begin{bmatrix} -\frac{\eta \Delta t}{Q} \\ R_1\left(1-e^{-\Delta t/R_1 C_1}\right) \end{bmatrix} u_k \tag{2}$$

The general formula for the measured outputs is presented in Eq. (3):

$$y_k = C_m x_{m,k} + D_m u_k \tag{3}$$

where D_m is the feed-through term, which is a necessary term for the ECM model of this battery.

Next, the output Eq. (4) for the terminal voltage is written as:

$$\begin{aligned} v_k &= C_{m,v} x_{m,k} + D_{m,v} u_k + OCV(z_k) \\ v_k &= [0 \ \ -1] \begin{bmatrix} z_k \\ v_{C_1,k} \end{bmatrix} + [-R_0] u_k + OCV(z_k) \end{aligned} \tag{4}$$

The general equations for the output to be controlled are presented with the Eqs. (5a) and (5b):

$$z_k = C_{m,z} x_{m,k} + D_{m,z} u_k \tag{5a}$$

In this case, SOC is selected as the output to be controlled and is presented as Eq. (5b):

$$z_k = [1 \quad 0] \begin{bmatrix} z_k \\ v_{C_1,k} \end{bmatrix} + [0] u_k \tag{5b}$$

In this case, the sampling time (Δt) and the cell efficiency (η) are considered as 1 s and 0.997, respectively. These values are determined from [3, 4] based on a Li-ion battery manufactured by the LG Chem Ltd. [4]. 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 (Δu_k). The final augmented state-space Eqs. (6), (7), (8), and (9) are presented below, based on the design in [3]:

$$\chi_{k+1} = \tilde{A}\chi_k + \tilde{B}\Delta u_{k+1} \tag{6}$$

$$v_k = \tilde{C}_v \chi_{k,} + OCV(z_k) \tag{7}$$

$$z_k = \tilde{C}_z \chi_{k,} \tag{8}$$

where the χ_k is defined as follows with Eq. (9):

$$\begin{aligned} &\chi_k = \begin{bmatrix} x_k \\ u_k \end{bmatrix}, \ \tilde{A} = \left[\begin{array}{c|c} A_m & B_m \\ \hline 0 & I \end{array}\right], \ \tilde{B} = \begin{bmatrix} 0 \\ \hline I \end{bmatrix}, \\ &\tilde{C}_v = \left[C_{m,v} \mid D_{m,v}\right], \text{and } \tilde{C}_z = \left[C_{m,z} \mid D_{m,z}\right]. \end{aligned} \tag{9}$$

and also $x_k = \left[\frac{\Delta x_{m,k}}{y_k}\right]$ from adding the integral action.

2.2 Prediction of state and output variables

Trimboli's group [4, 26] incorporated a feed-through term in the modified MPC algorithm, which was built upon and extended from the work done in [29]. A detailed description of the extended work can be found in [4, 26], and the synopsis of this approach can be found in [3]. For illustration purposes, the summary of this approach is presented below.

After completing the augmented model (from Section 2.1), the gain matrices are computed. To achieve this, the state Eq. (1), as demonstrated below, is propagated to obtain the future states.

$$\begin{aligned} \chi_{k+1} &= \tilde{A}\chi_k + \tilde{B}\Delta u_{k+1} \\ \chi_{k+2} &= \tilde{A}\chi_{k+1} + \tilde{B}\Delta u_{k+2} = \tilde{A}\left(\tilde{A}\chi_k + \tilde{B}\Delta u_{k+1}\right) + \tilde{B}\Delta u_{k+2} \\ &= \tilde{A}^2\chi_k + \tilde{A}\tilde{B}\Delta u_{k+1} + \tilde{B}\Delta u_{k+2} \\ \chi_{k+3} &= \tilde{A}^3\chi_k + \tilde{A}^2\tilde{B}\Delta u_{k+1} + \tilde{A}\tilde{B}\Delta u_{k+2} + \tilde{B}u_{k+3} \\ &\vdots \\ \chi_{k+N_p} &= \tilde{A}^{N_p}\chi_k + \tilde{A}^{N_p-1}\tilde{B}\Delta u_{k+1} + \tilde{A}^{N_p-2}\tilde{B}\Delta u_{k+2} + \cdots \\ &\quad + \tilde{A}^{N_p-N_c}\tilde{B}u_{k+N_c} \end{aligned} \tag{10}$$

Next, the output Eq. (3) is propagated and substituted with the elements of Eq. (4), in order to obtain the predicted output as Eq. (11).

$$\begin{aligned} y_{k+1} &= \tilde{C}\chi_{k,+1} = \tilde{C}\tilde{A}\chi_k + \tilde{C}\tilde{B}\Delta u_{k+1} \\ y_{k+2} &= \tilde{C}\chi_{k,+2} = \tilde{C}\tilde{A}^2\chi_k + \tilde{C}\tilde{A}\tilde{B}\Delta u_{k+1} + \tilde{C}\tilde{B}\Delta u_{k+2} \\ y_{k+3} &= \tilde{C}\chi_{k,+3} = \tilde{C}\tilde{A}^3\chi_k + \tilde{C}\tilde{A}^2\tilde{B}\Delta u_{k+1} + \tilde{C}\tilde{A}\tilde{B}\Delta u_{k+2} \\ &\quad + \tilde{C}\tilde{B}\Delta u_{k+3} \vdots \\ y_{k+N_p} &= \tilde{C}\chi_{k,+N_p} = \tilde{C}\tilde{A}^{N_p}\chi_k + \tilde{C}\tilde{A}^{N_p-1}\tilde{B}\Delta u_{k+1} \\ &\quad + \tilde{C}\tilde{A}^{N_p-2}\tilde{B}\Delta u_{k+2} + \cdots + \tilde{C}\tilde{A}^{N_p-N_c}\tilde{B}\Delta u_{k+N_c} \end{aligned} \tag{11}$$

Rewriting Eq. (11) in matrix form produces the following Eqs. (12) and (13):

$$\mathbf{Y_k} = \begin{bmatrix} \tilde{C} \\ \tilde{C}\tilde{A} \\ \tilde{C}\tilde{A}^2 \\ \vdots \\ \tilde{C}\tilde{A}^{N_p-1} \end{bmatrix} \tilde{A}\chi_k \tag{12}$$

$$+ \begin{bmatrix} \tilde{C}\tilde{B} & 0 & 0 & \cdots & 0 \\ \tilde{C}\tilde{A}\tilde{B} & \tilde{C}\tilde{B} & 0 & \cdots & 0 \\ \tilde{C}\tilde{A}^2\tilde{B} & \tilde{C}\tilde{A}\tilde{B} & \tilde{C}\tilde{B} & \cdots & 0 \\ \vdots & & & & \\ \tilde{C}\tilde{A}^{N_p-1}\tilde{B} & \tilde{C}\tilde{A}^{N_p-2}\tilde{B} & \tilde{C}\tilde{A}^{N_p-3}\tilde{B} & \cdots & \tilde{C}\tilde{A}^{N_p-N_c}\tilde{B} \end{bmatrix} \begin{bmatrix} \Delta u_{k+1} \\ \Delta u_{k+2} \\ \Delta u_{k+3} \\ \vdots \\ \Delta u_{k+N_c} \end{bmatrix}$$

$$\mathbf{Y_k} = \Phi\tilde{A}\chi_k + G\Delta\mathbf{U}_k \tag{13}$$

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

$$\Delta\mathbf{U_k} = \begin{bmatrix} \Delta\mathbf{U_{k,nf}} \\ \hline \Delta\mathbf{U_{k,ff}} \end{bmatrix}, \text{and } G = \left[G_{nf} \middle| G_{ff} \right]. \tag{14}$$

As discussed in [4], expressing $\Delta\mathbf{U_{k,ff}}$ in terms of $\Delta\mathbf{U_{k,nf}}$ results in Eq. (15):

$$\Delta\mathbf{U_{k,ff}} = -\left(\mathbf{v}\Delta\mathbf{U_{k,nf}} + u_k\right) \tag{15}$$

where $\mathbf{v}_{1\times N_c} = [1 \quad 1 \quad 1 \quad \cdots \quad 1]$.

Furthermore, substituting Eq. (13) with the elements of the Eqs. (14) and (15) results in Eq. (16):

$$\mathbf{Y_k} = \Phi\tilde{A}\chi_k + G_{nf}\Delta\mathbf{U}_{k,nf} - G_{ff}\mathbf{v}\Delta\mathbf{U}_{k,nf} - G_{ff}u_k \tag{16}$$

The aforementioned steps are required to process and complete the MPC algorithm. For our embedded architectures, the above equations (from (10) to (16)) remain the same, since the temperature is considered as a constant. There are four temperature-dependent variables, Q, R_0, R_1, and r, utilized in the augmented model. These variables are detailed in Section 4.2.1.

2.3 Optimization

With the embedded systems design, our objective is to create a control signal that brings both the output signal Y_k and the reference or set-point signal R_s closer together as much as possible. In this case, it is assumed that R_s remains constant inside our prediction window. The cost function that reflects our optimization goal is written in a matrix form as below:

$$J_k = (\mathrm{Y_k} - \mathrm{R_s})^T(\mathrm{Y_k} - \mathrm{R_s}) + P_1\Delta\mathrm{U}^T_{k,nf}\Delta\mathrm{U_{k,nf}}. \tag{17}$$

In the above Eq. (17), R_s is a vector of set-point information, and P_1 is a penalty factor based on the given constants rw and λ_P. Substituting Eq. (17) with the elements of Eq. (16), utilizing properties of the symmetric matrices, and grouping the terms, results in the following cost function:

$$\begin{aligned} J_{\mathbf{k}} = {} & \Delta\mathbf{U}^T_{k,nf}\left(G^T_{nf}G_{nf} + P_1 I - G^T_{nf}G_{ff}\mathbf{v} - \mathbf{v}^T G^T_{ff}G_{nf} + \mathbf{v}^T G^T_{ff}G_{ff}\mathbf{v}\right)\Delta\mathbf{U}_{k,nf} \\ & -2\Delta\mathbf{U}^T_{k,nf}\left(G^T_{nf}\mathbf{R_s} + \mathbf{v}^T G^T_{ff}\mathbf{R_s} - G^T_{nf}\Phi\tilde{A}\chi_k - \mathbf{v}^T G^T_{ff}\Phi\tilde{A}\chi_k - G^T_{nf}G_{ff}u_k - \mathbf{v}^T G^T_{ff}G_{ff}u_k\right) \\ & + \left(\Phi\tilde{A}\chi_k - \mathbf{R_s} - G_{ff}u_k\right)^T\left(\Phi\tilde{A}\chi_k - \mathbf{R_s} - G_{ff}u_k\right). \end{aligned} \tag{18}$$

Next, Hildreth's quadratic programming (HQP) technique is used to minimize the above cost function presented in Eq. (18). The input function for the HQP (where x represents the control variable) is written as below:

$$J = \frac{1}{2}x^T Ex + x^T F \tag{19}$$

The equality constraint is as follows:

$$Mx \le \gamma \tag{20}$$

The original function in Eq. (19) is augmented with the equality constraint (presented in Eq. (2) and multiplied by the Lagrange multiplier (λ)):

$$J = \frac{1}{2}x^T Ex + x^T F + \lambda^T(Mx - \gamma) \tag{21}$$

In this case, E and F can be inferred from Eq. (18) to produce the following Eqs. (22) and (23):

$$E = 2\left(G^T_{nf}G_{nf} + P_1 - G^T_{nf}G_{ff}\mathbf{v} - \mathbf{v}^T G^T_{ff}G_{nf} + \mathbf{v}^T G^T_{ff}G_{ff}\mathbf{v}\right) \tag{22}$$

$$\begin{aligned} F &= -2\left(G^T_{nf}\mathbf{R_s} + \mathbf{v}^T G^T_{ff}\mathbf{R_s} - G^T_{nf}\Phi\tilde{A}\chi_k - \mathbf{v}^T G^T_{ff}\Phi\tilde{A}\chi_k - G^T_{nf}G_{ff}u_k - \mathbf{v}^T G^T_{ff}G_{ff}u_k\right) \\ F &= -2\left(\left(G^T_{nf} + \mathbf{v}^T G^T_{ff}\right)\mathbf{R_s} - \left(G^T_{nf} + \mathbf{v}^T G^T_{ff}\right)\Phi\tilde{A}\chi_k - \left(G^T_{nf}G_{ff} + \mathbf{v}^T G^T_{ff}G_{ff}\right)u_k\right) \end{aligned} \tag{23}$$

A weight vector ($\boldsymbol{m}$) can be added to further enhance the performance of the MPC algorithm. The $\boldsymbol{m}$ vector is a $1 \times N_P - N_C$ vector that is typically computed offline in Matlab and stored either in registers or in BRAMs. In this case, P_2 is an extra penalty factor added to improve the performance. Since $N_C = 1$ is utilized, $\boldsymbol{v}$ vector becomes a scalar 1, thus becoming trivial. Considering that the SOC is the output to be controlled and the gain matrices used G_z and Φ_z, then E and F become:

$$E = 2\left(G_{nfz}^T G_{nfz} + P_1 - G_{nfz}^T G_{ffz}\mathbf{m} - \mathbf{m}^{\mathbf{T}} G_{ffz}^T G_{nfz} + \mathbf{m}^{\mathbf{T}} G_{ffz}^T G_{ffz}\mathbf{m} + \mathbf{m}^{\mathbf{T}}\mathbf{m}P_2\right) \tag{24}$$

$$F = -2\begin{pmatrix} \left(G_{nfz}^T + \mathbf{m}^{\mathbf{T}} G_{ffz}^T\right)\mathbf{R_s} - \left(G_{nfz}^T + \mathbf{m}^{\mathbf{T}} G_{ffz}^T\right)\Phi_z \tilde{A}\chi_k \\ -\left(G_{nfz}^T G_{ffz}\mathbf{m} + \mathbf{m}^{\mathbf{T}} G_{ffz}^T G_{ffz}\mathbf{m} + \mathbf{m}^{\mathbf{T}}\mathbf{m}P_2\right)u_k \end{pmatrix} \tag{25}$$

Next, the constraints for Eq. (20) are developed, which constrain the control input, the terminal voltage, and the maximum SOC. The developments of M and γ are detailed in [4]; the final Eq. (26) is presented below.

$$M = \begin{bmatrix} 1 \\ -1 \\ \left(G_{nfv} + G_{ffv}\mathbf{m}\right) \\ -\left(G_{nfv} + G_{ffv}\mathbf{m}\right) \\ \left(G_{nfz} + G_{ffz}\mathbf{m}\right) \end{bmatrix} \text{ and,}$$

$$\gamma = \begin{bmatrix} u_{\max} - u_k \\ -u_{\min} + u_k \\ v_{\max} - \left(\Phi_v \tilde{A}\chi + G_{ffv}\mathbf{m}u_k + OCV\right) \\ -v_{\min} + \left(\Phi_v \tilde{A}\chi + G_{ffv}\mathbf{m}u_k + OCV\right) \\ z_{\max} - \Phi_z \tilde{A}\chi - G_{ffz}\mathbf{m}u_k \end{bmatrix} \tag{26}$$

For the primal-dual approach, the partial derivatives of Eq. (21) are taken, with respect to x and λ as in [4]. In this case, setting the partial derivatives equal to zero and solving the equation for x and λ result in Eq. (27):

$$\lambda = -\left(ME^{-1}M^T\right)^{-1}\left(\gamma + ME^{-1}F\right) \tag{27}$$

$$x = -E^{-1}\left(M^T\lambda + F\right) \tag{28}$$

Substituting Eq. (26) with the elements of Eq. (25) results in Eq. (29):

$$x = -E^{-1}F - E^{-1}M^T\lambda \tag{29}$$

Since Δu is the control variable, Eq. (29) becomes Eq. (30):

$$\Delta u = \Delta u^o - E^{-1}M^T\lambda \tag{30}$$

In this case, the $\Delta u^o = -E^{-1}F$ is the unconstrained optimal solution to the control signal, and $-E^{-1}M^T\lambda$ is the correction factor based on the constraints computed by the HQP in case if Δu^o fails to meet the required constraints. To determine whether the optimal solution Δu^o is sufficient, it is substituted in Eq. (20), to obtain Eq. (31):

$$M\Delta u^o \le \gamma \tag{31}$$

If the above equation fails for any element of the constraint vectors, then the correction factor is computed using the HQP. The HQP technique is a numerical approach for solving the primal-dual problem. The primal-dual problem is equivalent to the following Eq. (32):

$$\max_{\lambda \ge 0} \min_{x} \left[\frac{1}{2}x^T Ex + x^T F + \lambda^T(Mx - \gamma)\right] \tag{32}$$

Substituting Eq. (21) with the elements of Eq. (29) results in Eq. (33):

$$\max_{\lambda \ge 0}\left(-\frac{1}{2}\lambda^T P\lambda - \lambda^T K - \frac{1}{2}F^T E^{-1}F\right) \tag{33}$$

where,

$$P = ME^{-1}M^T \tag{34}$$

$$K = \gamma + ME^{-1}F = \gamma - M\Delta u^o \tag{35}$$

2.4 Hildreth's quadratic programming technique

As discussed earlier, the λ is a vector of Lagrange multipliers. In Hildreth's quadratic programming (HQP), the λ is varied one element at a time. With a starting vector (λ^m), a single element (λ_i^m) of the vector is modified, utilizing P and K to minimize the cost function (presented in Eq. (21)), which creates λ_i^{m+1}. In this case, if the modification requires $\lambda_i^m < 0$, then set $\lambda_i^{m+1} = 0$, rendering the constraint to be inactive. Then, the next element (λ_{i+1}^{m+1}) of the vector is considered, and this process continues until all the elements of the entire λ^m vector are modified. This modification is computed using Eq. (36):

$$\lambda_i^{m+1} = \max(0, w_i) \tag{36}$$

where,

$$w_i = -\frac{1}{p_{ii}}\left[k_i + \sum_{j=1}^{i-1} p_{ij}\lambda_j^{m+1} + \sum_{j=i+1}^{n} p_{ij}\lambda_j^m\right] \tag{37}$$

In this case, k_i and p_{ij} are the scalar ith and ijth elements of K and P, respectively. This is an iterative process, which continues either until the λ converges (so that $\lambda^{m+1} \cong \lambda^m$) or until a maximum number of iterations is reached. This process concludes with a λ^* of either 0 or positive values. The positive values are the active constraints in the system at the time. The next step is to utilize λ^* in Eq. (3), to obtain our final control input as illustrated in Eq. (38):

$$\Delta u_{k+1} = \Delta u_{k+1}^o - E^{-1}M^T\lambda^* \tag{38}$$

2.5 Applying control signal

The control signal and the state signal are computed and updated using Eq. (6) (in Section 2.1). The first

element of $\Delta\mathbf{U_k}$ is used to update the control signal as shown in Eq. (39).

$$u_{k+1} = u_k + \Delta u_{k+1} \tag{39}$$

Next, the new control signal is used to determine the states for the next iteration, as presented in Eq. (40):

$$x_{k+1} = A_m x_k + B_m u_{k+1} \tag{40}$$

In this case, the state of charge (SOC) (i.e., x_{k+1}, [0] = z_{k+1}) is compared to reference values to determine if the Li-ion battery is fully charged. If the SOC is less than the reference values (z_{k+1} < reference), then the MPC algorithm is repeated to compute the next control signal.

3 Design approach and development platform

In this research work, we introduce our unique, novel, and efficient embedded architectures (two hardware versions and one software version) for the fast-charge model predictive controller (MPC). Our proposed embedded architectures for the fast-charge MPC algorithm are inspired by and based on the modified MPC algorithm for the lithium-ion battery cell-level MPC modeled by Trimboli's group [3, 4, 26]. We obtained the source codes written in Matlab for the existing fast-charge MPC algorithm from Trimboli's research group [4]. We use this validated Matlab model as the baseline for the performance and functionality comparison presented in Section 5.

For all our 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. Our 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, optimal solution, Hildreth's QP process, and state and plant generation, are at the third level of the design hierarchy; and the MPC is at the highest level.

All our hardware and software experiments are carried out on the ML605 FPGA development board [30], which utilizes a Xilinx Virtex 6 XC6VLX240T-FF1156 device. The development platform includes large on-chip logic resources (37,680 slices), MicroBlaze soft processors, and 2 MB on-chip BRAM (Block Random Access Memory) to store data/results.

All the hardware modules are designed in mixed VHDL and Verilog. They are executed on the FPGA (running at 100 MHz) 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 functionalities of the designs. 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 [31] from the Xilinx IP core library. The MicroBlaze is also configured to use single-precision floating-point units 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 5.

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

$$\text{Speedup} = \frac{\text{BaselineExecutionTime(Software)}}{\text{ImprovedExecutionTime(Hardware)}} \tag{41}$$

3.1 System-level design

We introduce system-level architectures for our embedded hardware versions as well as our embedded software version. For some of the designs, we integrate on-chip BRAMs 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 Fig. 2, the AXI (Advanced Extensible Interface) interconnect acts as the glue logic for the system.

We also incorporate MicroBlaze soft processor in both the hardware versions. For the embedded hardware, MicroBlaze is configured to have 128 KB of local on-chip memory. As illustrated in Fig. 2, our user-designed hardware module communicates with the MicroBlaze processor and with the other peripherals via AXI bus [32], through the AXI Intellectual Property Interface (IPIF) module, using a set of ports called the Intellectual Property Interconnect (IPIC). For the hardware designs, 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. Conversely, the user-designed hardware module performs the whole fast-charge MPC algorithm. The execution times for the hardware as well as the software on MicroBlaze are

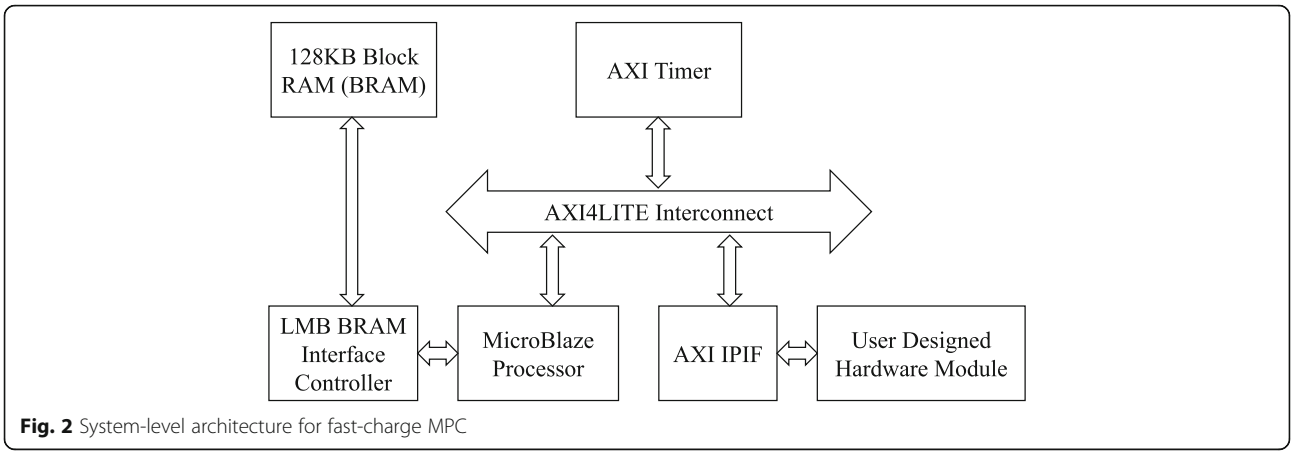

Fig. 2 System-level architecture for fast-charge MPC

obtained using the AXI Timer [33] running at 100 MHz.

4 Embedded hardware and software architectures for MPC

In this section, we introduce unique, novel, and efficient embedded architectures (both hardware and software) for the fast-charge model predictive controller (MPC) algorithm. Apart from our main objective, one of our 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, in order to reduce the hardware resources required for BMS.

Initially, we investigate and analyze the functional flow of the MPC algorithm in [4], and then, we decompose the algorithm into four high-level stages (as shown in Fig. 3) to simplify the design process. The operations of the four consecutive stages are as follows:

- Stage 1—Compute the augmented model and gain (or data) matrices.
- Stage 2—Check the plant state (i.e., whether the charging is completed or not); compute the global optimal solution that is not subjected to constraints; determine whether the constraints are violated or not.

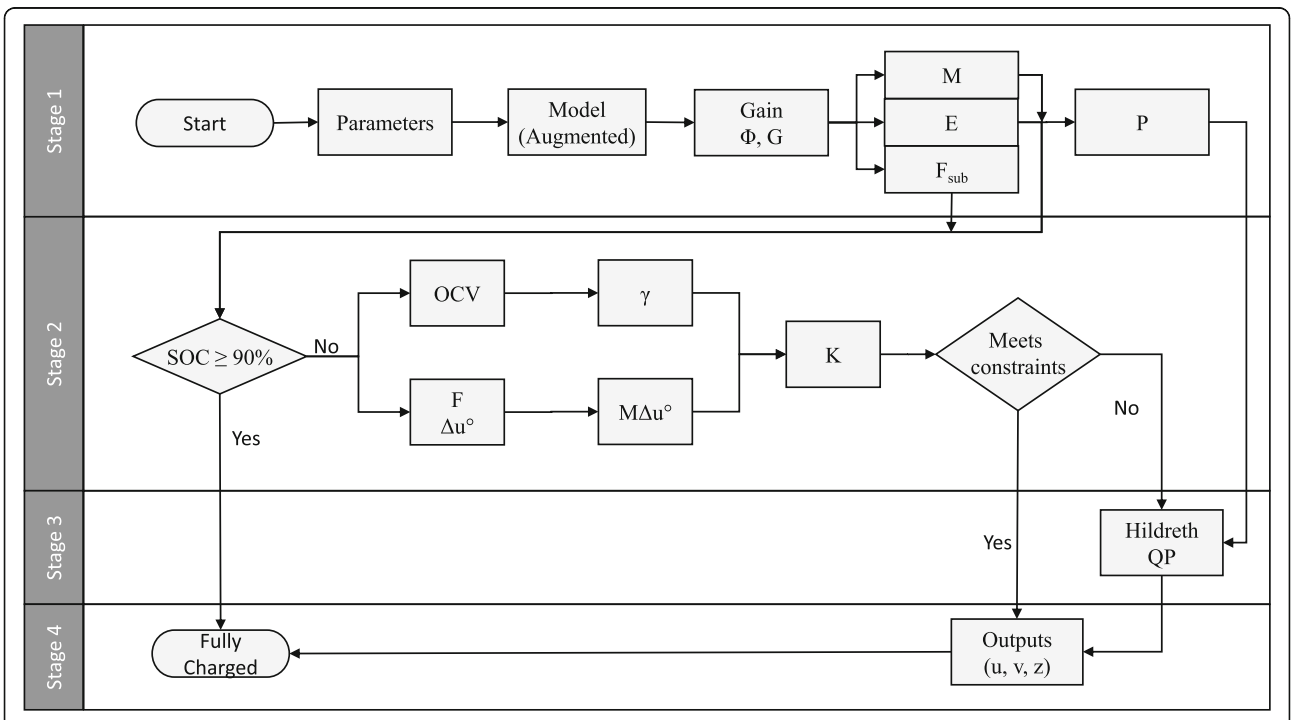

Fig. 3 Four high-level stages of fast-charge MPC algorithm

- Stage 3—Compute the new or adjusted solution using HQP procedure, if and only if, constraints are violated.
- Stage 4—Compute the new plant states and plant outputs. It should be noted that for experimental purposes, the plant output is computed in stage 4; however, in a real-world scenario, the plant output would be a measured value.

In order to enhance the performance and area efficiency of both our embedded hardware and software designs, all the time-invariant computations are relocated to stage 1 from other stages of the MPC algorithm. In this case, stage 1 is considered as the initial 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 in an 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 an example, consider the P parameter typically associated with stage 3. This P is created by multiplying a 32-word vector by a 32-word vector to create a 32×32 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 result every clock cycle, after an initial latency of 8 clock cycles. With the original fast-charge MPC algorithm [3], the P parameter is computed every time, 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 5.

There are two major advantages of using the modified fast-charge MPC algorithm for the 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 needing 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.

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

4.1 Embedded software architecture

Initially, we design and develop the software for the fast-charge MPC algorithm in C using the XCode integrated development environment. This software design is executed on a desktop computer with dual core i7 processor. Then, the results are compared and verified with the baseline results from the Matlab code. Both the C and Matlab results are also used to verify our 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. Hence, we dramatically modify the above software design (executed on desktop computer) 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 parameter 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 offline computations can be done without exceeding the memory requirements.

The embedded software is designed to mimic the hardware. 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 τ, and $Q_{(charge)}$, 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 offline with Matlab codes. The software flow for the fast-charge MPC is presented in Table 1.

Table 1 Software algorithm for fast-charge MPC

Stage	MPC software algorithm
1.	1.1. Get temperature 1.2. Call parameter function 1.3. Calculate Φ and G matrices 1.4. Create G_{nf} and G_{ff} (nf = near future and ff = far future) dual mode data) 1.5. Calculate E 1.6. Calculate P (matrix for Hildreth QP) 1.7. Build M (constraints vector) 1.8. Start loop – compare $x_m[0]$ (SOC) to reference to see if fully charged. If not fully charged, continue, else exit
2.	2.1. Calculate F 2.2. Solve $-FE^{-1}$ (optimal unconstrained Δu from J) 2.3. Build γ (constraints vector) 2.4. Compare: $M\Delta u \leq \gamma$
3.	3.1. False – call Hildreth QP, develop new Δu that meets constraints 3.2. True Goto Stage 4 (4.1)
4.	4.1. Calculate the next control signal, next states, and outputs 4.2. Goto Start Loop (1.8)

4.2 Embedded hardware designs

In this research work, we design and develop two hardware versions: the 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 anticipate an enhancement of the speedup of the overall MPC algorithm. With HW_v2, an 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 expect a reduction in overall area, since the registers and the associated interconnects (in HW_v1) typically occupy large space on chip. Conversely, the existing on-chip BRAMs are dual-port; hence, these 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 Table 1, thus having 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 initially, we introduce HW_v1, almost as a proof-of-concept work; next, we introduce HW_v2 to address certain issues that have arisen with HW_v1.

Xilinx offers two types of floating-point IP cores: AXI-based and non-AXI-based. For the register-based HW_v1, we use the standard AXI-based IP cores 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, we utilize the non-AXI-based IP cores for the fundamental operators. These IP cores allow the lowest latency adder (5-cycle latency) and multiplier (1-cycle latency) units to support 100 MHz 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 manage to use lower latency but more resource-intensive IP cores, since it consists of fewer multipliers and adders, whereas 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, we design and develop the embedded hardware architectures for each stage as separate modules, 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, we design a top-level module to integrate the four stages of the MPC algorithm and to provide necessary communication/control among the stages. Among various control/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.

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 χ_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 Fig. 4. As illustrated, the relocated computations include E, M, P, and the sub-matrices for F. 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 Fig. 4) is created from Eqs. (6), (7), and (8) depending on the temperature-dependent parameters, initial states $x_k = [0, 0.5]$, and initial control input $u_k = 0$.

4.2.1.1 Computing parameters Since varying temperatures are inevitable in the real-world scenario, for HW_v2, we integrate an additional parameter module to compute the four temperature-dependent variables Q, R_0, R_1, and r, utilized in the augmented model.

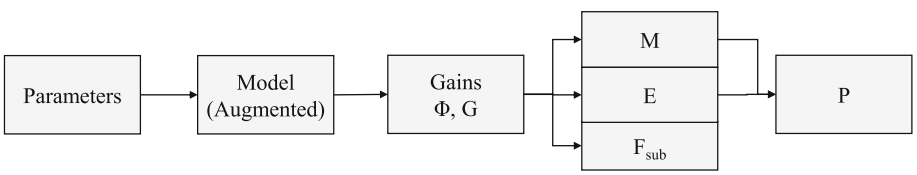

Fig. 4 Functional/data flow for stage 1

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: $y = a_3x^3 + a_2x^2 + a_1x + a_0$, where $x = T\text{-}ref$; in this case, T is the temperature and *ref* is (min) from Table 2. As presented in Table 2, 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, on the premise that the temperature will remain constant [4]. In this paper, for the experimental results and analysis (in Section 5), 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.

The internal architecture of the parameter module (from Fig. 4) for HW_v2 is depicted in Fig. 5. 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 Fig. 5, 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.

Table 2 Temperature regions for cubic spline

Region	Range	Reference (°C)
1	$-15\ °C \le T < -5\ °C$	−15
2	$-5\ °C \le T < 5\ °C$	−5
3	$5\ °C \le T < 15\ °C$	5
4	$15\ °C \le T < 25\ °C$	15
5	$25\ °C \le T < 35\ °C$	25
6	$35\ °C \le T$	35

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., Eqs. (1)–(8) [4]) are presented below in (42).

$$A_m = \begin{bmatrix} 1 & 0 \\ 0 & e^{-\Delta t / R_1 C_1} \end{bmatrix},$$
$$B_m = \begin{bmatrix} -\frac{\eta \Delta t}{Q} \\ R_1\left(1 - e^{-\Delta t / R_1 C_1}\right) \end{bmatrix} \text{ and}$$
$$x_{m,k} = \begin{bmatrix} z_k \\ v_{c_1,k} \end{bmatrix} \tag{42}$$

The augmented state-space equation matrices are given in Eq. (9) (in Section 2.1), where, Δt is the sampling time (considered as 1 s) and η 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 the hardware versions. For both HW_v1 and HW_v2, the augmented model computes all the elements in Eq. (42) 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 P_1 and P_2 in Eq. (24).

The internal architecture of the augmented model for HW_v1 is shown in Fig. 6. In order to compute the values in Eq. (42) 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.

4.2.1.3 Computing gain matrices Next, we perform the gain matrix computations including the Φ, G_{rf}, and G_{ff}. Each gain matrix has identical computations, which are independent of each other. In our design, the Φ and G matrices are developed for both the terminal voltage v_k and SOC Z_k separately, resulting in Φ_v, Φ_z ,G_v, and G_z. The gain matrices are derived from Eq. (12), where Φ_v and Φ_z are:

$$\Phi_v = \begin{bmatrix} \tilde{C}_v \\ \tilde{C}_v\tilde{A} \\ \tilde{C}_v\tilde{A}^2 \\ \\ \tilde{C}_v\tilde{A}^{N_p-1} \end{bmatrix} \text{ and } \Phi_z = \begin{bmatrix} \tilde{C}_z \\ \tilde{C}_z\tilde{A} \\ C_z\tilde{A}^2 \\ \\ \tilde{C}_z\tilde{A}^{N_p-1} \end{bmatrix} \tag{43}$$

It should be noted that in our design, from Eq. (9), the $\tilde{B}$ is considered as $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$; thus, each column

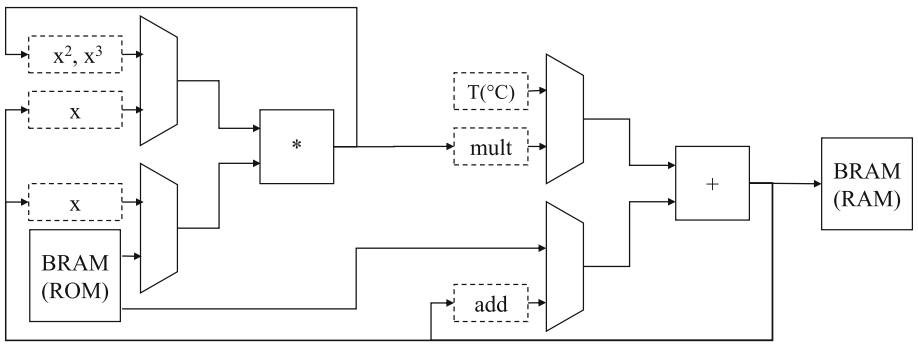

Fig. 5 Internal architecture of parameter module for HW_v2

of G is derived from the third column of Φ. This only requires arranging the elements of the G matrix in registers or BRAMs, instead of re-computing these elements. In this case, G_{rf} is a $N_P \times N_C$ matrix and G_{ff} is a $N_P \times N_P - N_C$ matrix. As in Eq. (44), for $N_c = 1$, G_{rf} is the first column of G, from Eq. (12), and G_{ff} comprises the rest of the columns. Utilizing $\tilde{C}_v$ and $\tilde{C}_z$, which incorporated the feed-through term from Eq. (44), we create G_{rfv}, G_{ffv}, G_{nfz}, and G_{ffz}.

$$G = \begin{bmatrix} \tilde{C}\tilde{B} \\ \tilde{C}\tilde{A}\tilde{B} \\ \tilde{C}\tilde{A}^2\tilde{B} \\ \vdots \\ \tilde{C}\tilde{A}^{N_p-1}\tilde{B} \end{bmatrix}_{G_{rf}} \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \tilde{C}\tilde{B} & 0 & \cdots & 0 \\ \tilde{C}\tilde{A}\tilde{B} & \tilde{C}\tilde{B} & \cdots & 0 \\ \vdots & & & \\ \tilde{C}\tilde{A}^{N_p-2}\tilde{B} & \tilde{C}\tilde{A}^{N_p-3}\tilde{B} & \cdots & \tilde{C}\tilde{A}^{N_p-N_c}\tilde{B} \end{bmatrix}_{G_{ff}} \tag{44}$$

The internal architecture for computing the Φ matrix (for both HW_v1 and HW_v2) is shown in Fig. 7. The size of Φ is determined by the prediction horizon (N_p), the number of states, (N_s), and the number of inputs, (N_{in}), and is an $N_p x$ ($N_s + N_{in}$) matrix. As illustrated, the Φ 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 Fig. 7, 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 Φ 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 Fig. 8) in parallel.

As demonstrated in Fig. 7, both hardware versions have the same internal architecture for computing the Φ matrix. In this case, HW_v1 waits until Φ matrix computation is completed and then loads G_{rf} and G_{ff}. Also, HW_v1 employs two gain matrix modules to compute $[\Phi_v, \; G_{nfv}, \; G_{ffv}]$ and $[\Phi_z, \; G_{nfz}, \; G_{ffz}]$ matrices in parallel.

Conversely, HW_v2 computes each row of the Φ matrix and then saves the row term in an appropriate memory location, in order to subsequently build Φ, G_{rf}, and G_{ff} utilizing an addressing algorithm. Furthermore, HW_v2 computes and saves the $\Phi_v\tilde{A}$ and $\Phi_z\tilde{A}$ matrices. As illustrated in Fig. 9, the calculation of Φ and $\Phi\tilde{A}$ only differs by one row. Hence, by merely computing one

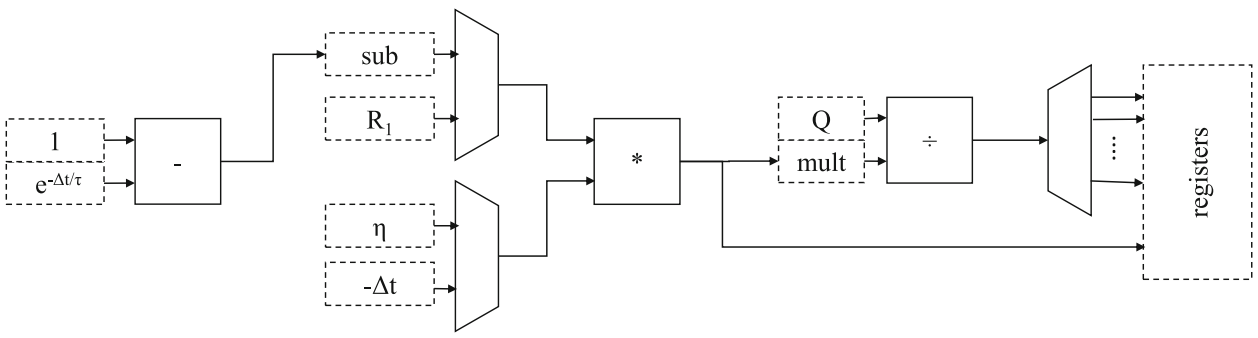

Fig. 6 Augmented model for HW_v1

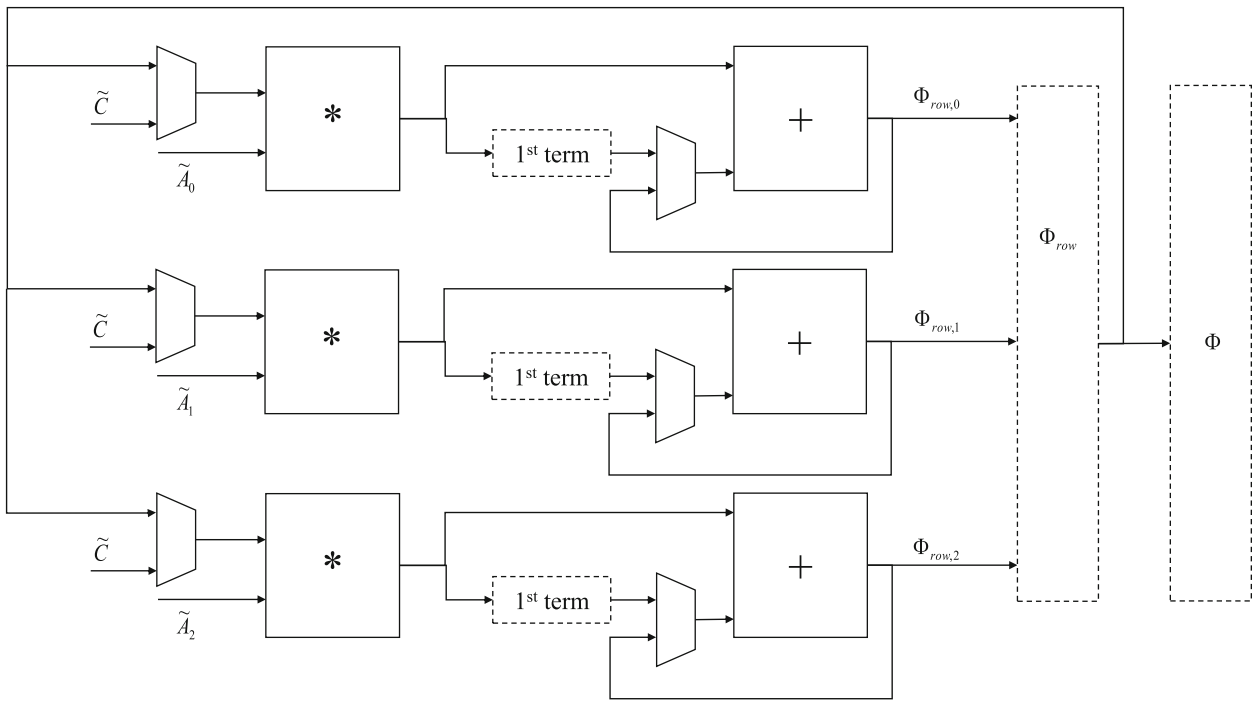

Fig. 7 Internal architecture for Φ for HW_v1 and HW_v2

additional row, $\Phi\tilde{A}$ can be built in the same fashion and at the same time as Φ, G_{rf}, and G_{ff}, using one extra iteration.

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 Fig. 10. In this case, the hardware module for computing the Φ matrix (from Fig. 7) is reused in this module.

HW_v1 computes $\Phi\tilde{A}$ in a separate module (as in Fig. 11), after completing the Φ 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 Fig. 11, 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.

4.2.1.4 Time-invariant computations for HW_v1 As mentioned in Section 4.2.1, all the time-invariant computations (*E*, *M*, *P*, and sub-matrices of *F*), which are deemed independent of χ_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**

First, we present the architecture for HW_v1, since it intuitively follows the order of operations. Considering Eq. (24) from Section 2.3, there are no χ or Δu terms, unless the temperature varies. As a result, *E* remains constant and can be performed in stage 1. We decompose this *E* computation into several

$$\begin{bmatrix} \Phi_{row} \end{bmatrix} \left[\begin{array}{c|c|c} \tilde{A}_0 & \tilde{A}_1 & \tilde{A}_2 \end{array}\right] = \begin{bmatrix} \Phi_{row\_next} \end{bmatrix}$$

Fig. 8 Organization of $\Phi\tilde{A}$

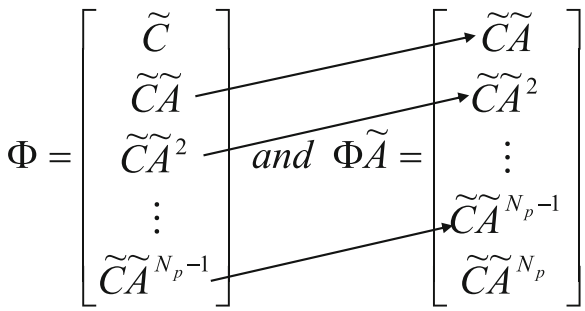

Fig. 9 Comparison of Φ and $\Phi\tilde{A}$

sub-functions, in such a way that each sub-function comprises only one matrix computation. Then, for HW_v1, we design separate sub-modules to perform different matrix computations such as a vector-scalar multiplication (**VS**), a vector-vector multiplication (**VV**), a vector-matrix multiplication (**VM**), and a vector-matrix transpose multiplication ($\mathbf{V^T}$ $\mathbf{M^T}$). The decomposed computations are presented in Eqs. (45)–(53):

$$E_1 = G_{nfz}^T G_{nfz} \tag{45}$$

$$E_2 = E_{2a}\mathbf{m}, \quad E_{2a} = G_{nfz}^T G_{ffz} \tag{46}$$

$$E_3 = E_{3a} G_{nfz}, \quad E_{3a} = \mathbf{m^T} G_{ffz}^T \tag{47}$$

$$E_4 = E_{4a}\mathbf{m}, \quad E_{4a} = E_{3a} G_{ffz} \tag{48}$$

$$E_5 = E_{5a} P_2, \quad E_{5a} = \mathbf{m^T m} \tag{49}$$

$$P_1 = rw(1-\gamma_P) \tag{50}$$

$$P_2 = rw\gamma_P \tag{51}$$

$$E = E_1 + P_1 + E_2 + E_3 + E_4 + E_5 \tag{52}$$

$$Einv = E^{-1} \tag{53}$$

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 multiplication 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 Eq. (25) from Section 2.3. From Eq. (25), it is observed that the last term is in fact $(E_2 + E_4 + E_5)u_k$. In this case, integrating Eq. (54) to the E module reduces the number of outputs F_{3a}, i.e., from three 32-bit values to a single 32-bit value.

$$F_{3a} = E_2 + E_4 + E_5 \tag{54}$$

As illustrated in Fig. 12a, the E module for HW_v1 computes the Eqs. (45)–(54). 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 (Fig. 12c) to perform the necessary vector/matrix operations. Our MAC unit is designed in such a way to reduce each final MAC result by 12 clock cycles. In our designs, the vector-vector multiplication (**VV**) is identical to vector squared ($\mathbf{V^2}$) except the former accepts two separate vectors, whereas the latter accepts only one; vector-matrix multiplication (**VM**) and vector-matrix transpose multiplication ($\mathbf{V^T M^T}$) 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 Fig. 12b, we design a separate sub-module to compute the tuning parameters P_1 and P_2, 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 Eqs. (55)–(58), which are derived from Eq. (25).

$$F_{1a} = G_{nfz}^T + E_{3a} \tag{55}$$

$$F_1 = F_{1a} R_s \tag{56}$$

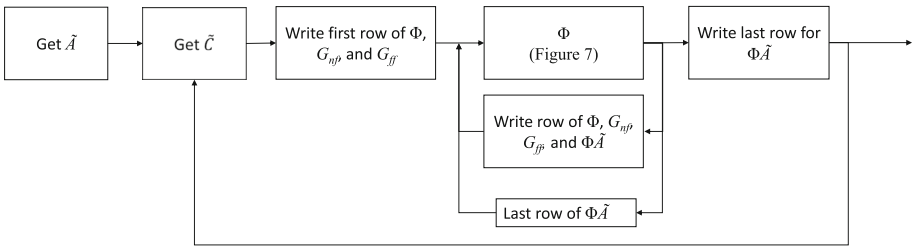

Fig. 10 Internal architecture for gain matrices and $\Phi\tilde{A}$ for HW_v2

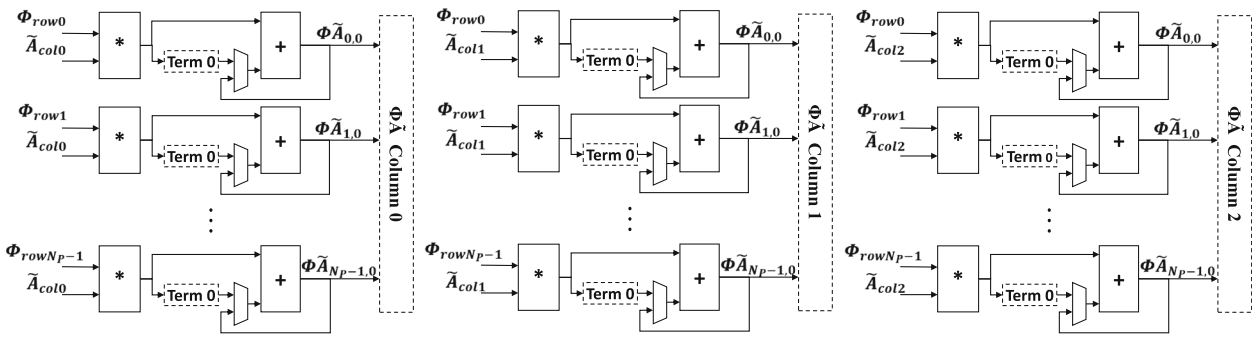

Fig. 11 Internal architecture for $\Phi\tilde{A}$ module for HW_v1

$$F_{2a} = F_{1a}\Phi_z \tag{57}$$

$$F_{2c} = F_{2a}\tilde{A} \tag{58}$$

The internal architecture of the F_sub module is depicted in Fig. 13, which consists of a vector-addition module (**V + V**), a vector-accumulation module (**VAcc**), two **VM** modules, and a FPU multiplier. In this case, the former two sub-modules (**V + V** and **VAcc**) utilize FPU adders to perform the required operations.

M module for HW_v1

We also design the M module to compute M constraints. The M and γ are presented in Eq. (26). All the elements of M and some elements of γ can be computed with Eqs. (59)–(65) as follows:

$$Mposv_a = G_{ffv}\mathbf{m} \tag{59}$$

$$Mposz_a = G_{ffz}\mathbf{m} \tag{60}$$

$$Mposv = \left(G_{nfv} + Mposv_a\right) \tag{61}$$

$$Mnegv = -Mposv \tag{62}$$

$$Mposz = \left(G_{nfz} + Mposz_a\right) \tag{63}$$

$$\Phi Av = \Phi_v\tilde{A} \tag{64}$$

$$\Phi Az = \Phi_z\tilde{A} \tag{65}$$

HW_v1 employs separate modules to perform $G_{ff}\mathbf{m}$ and $\Phi\tilde{A}$. The internal architecture for $\Phi\tilde{A}$ is demonstrated in Fig. 11, and the architecture of $G_{ff}\mathbf{m}$ computation is

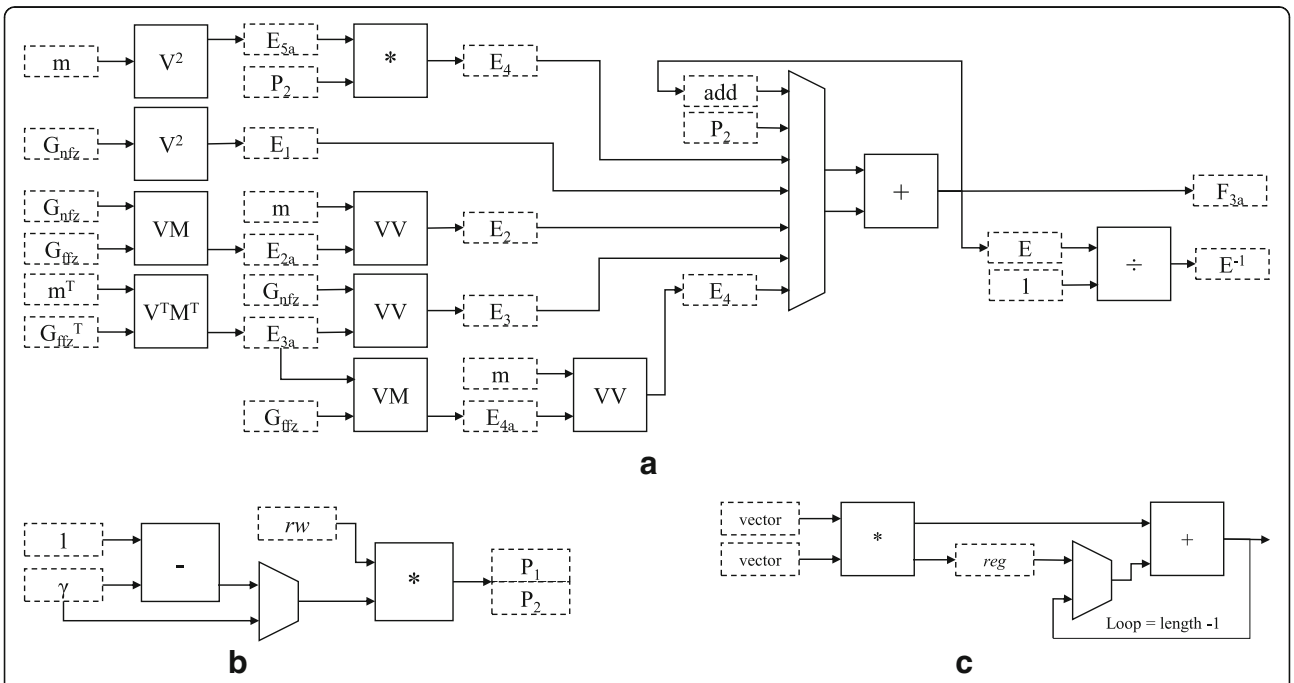

Fig. 12 Internal architecture for E module for HW_v1. **a** E module, **b** P_1 and P_2, **c** V V multiply-accumulate

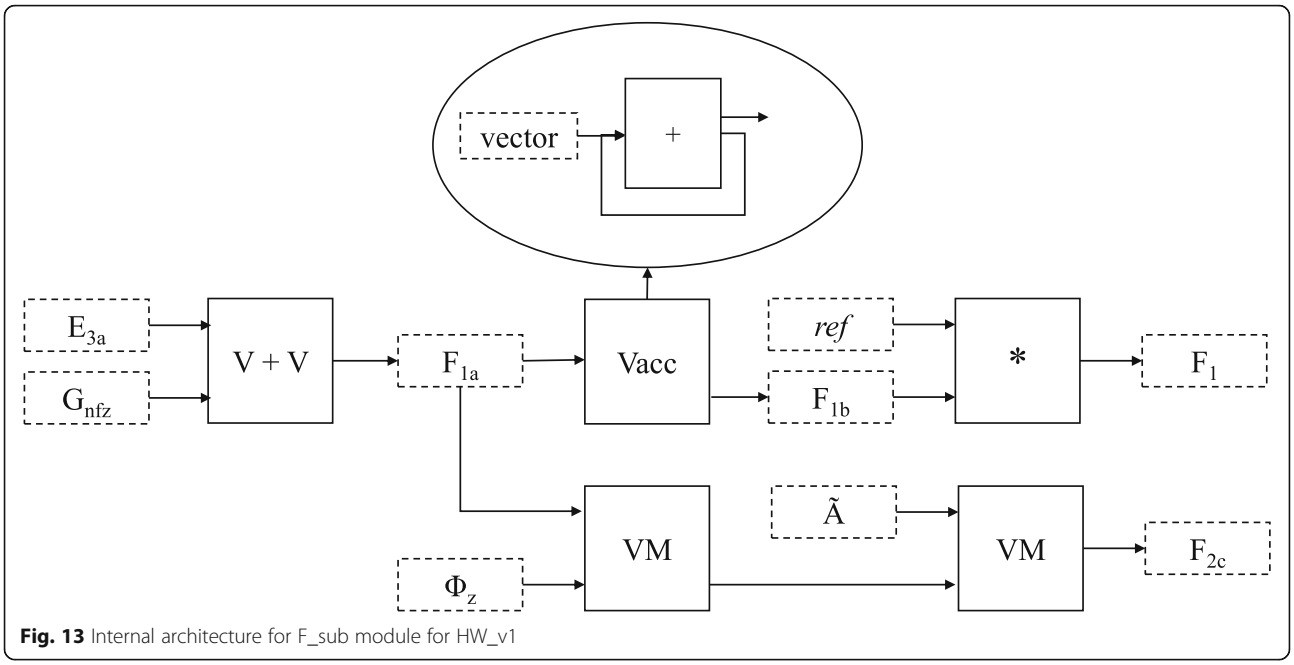

Fig. 13 Internal architecture for F_sub module for HW_v1

similar to the **VM** sub-module. With the M module, the negation operations (in Eqs. (61) and (63)) 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

Next, we design the P module for HW_v1, which is derived from Eq. (34), $P = ME^{-1}M^T$. As discussed in Section 2.4, Hildreth's quadratic programming (HQP) utilizes this equation to compute λ vector. We decompose this equation to Eqs. (66) and (67) as follows:

$$M_{conEinv} = ME^{-1} \tag{66}$$

where, Eq. (67) performs a vector-scalar multiplication.

$$P = M_{conEinv}M^T \tag{67}$$

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 our group [34]. Utilizing this algorithm, elements of 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. Our attempt to store the matrix using registers caused our initial design to exceed the chip resources by 25%. Therefore, we integrate BRAM to the P module to store the P matrix in HW_v1. In this case, we use only two PEs 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 the latency. The results of P matrix computation are reused in stage 3.

In summary, HW_v1 is designed with separate modules, including **GFFm**, **ΦA**, **E**, **F_sub**, **M**, and **P**, to execute various computations in stage 1. In this case, two **GFFm** modules compute Eqs. (59) and (60) in parallel, and two **ΦA** modules compute Eqs. (64) and (65) in parallel. The **F_sub** module computes Eqs. (55)–(58), the **M** module computes Eqs. (61)–(63), and finally **P** module computes Eqs. (66) and (67).

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. In this case, we design a unique pipelined multiply-and-accumulator (MACx) module 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/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 Eqs. (61)–(63)) and F_{1a} (for Eq. (55)), when using BRAMs to store the vectors. The Eqs. (61) and (55) would generate the same values.

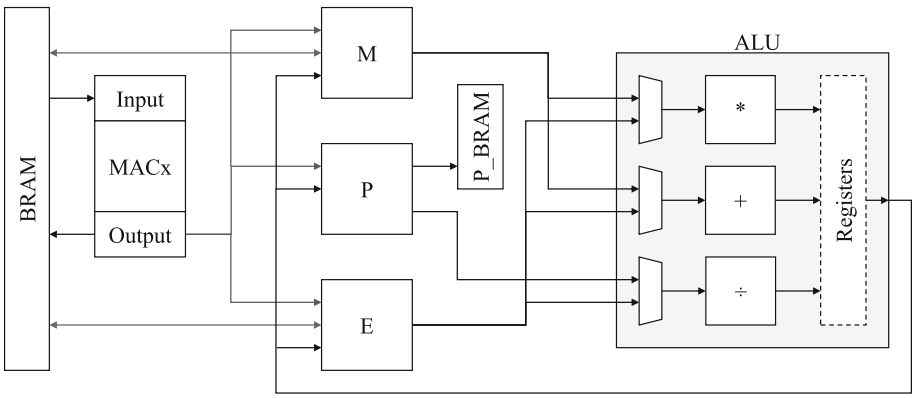

Fig. 14 E, F, M, and P modules for HW_v2

Figure 14 shows the top-level architecture for HW_v2 for the time-invariant computations *E*, *F*, *M*, and *P*. As illustrated, the multiplier, 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 final results are stored in the BRAMs.

The internal architecture of the pipelined MACx module is depicted in Fig. 15. 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 three or more vectors) faster than its parallel HW_v1 counterpart. In this case, we carefully configure the FPUs to have the lowest latency without compromising the highest system-clock frequency (100 MHz). 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.

Furthermore, in HW_v1, computations such as $G_{ffz}\boldsymbol{m}$ 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., $G_{ffz\_row0} * \boldsymbol{m}$), the resulting element and the first element of G_{nfz} in Eq. (63) is utilized by the **M** module to generate the first element of *Mposz*. This dramatically reduces the time required to execute stage 1, as detailed in Section 5.

With the pipelined MACx, the input wrapper controls the order of the operations (i.e., execution order). Since the computations are performed sequentially, the "execution order" is determined carefully, 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 three 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 are loaded into the MACx input buffers. The execution order using the pipelined MACx for HW_v2 is as follows:

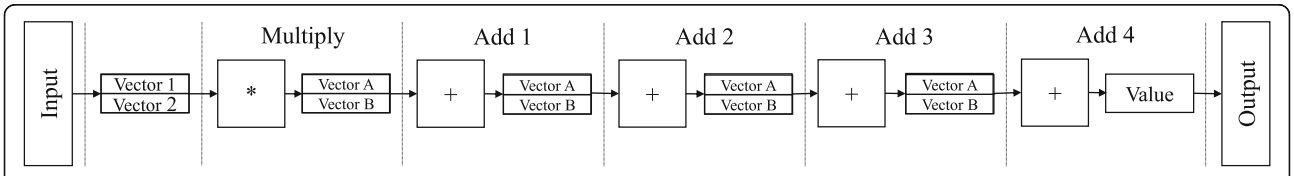

Fig. 15 Pipelined MACx module for HW_v2

1. $E_{5a} = \mathbf{m}^T\mathbf{m}$, Eq. (49). 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 Φ and the gain matrix 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 while the E module accesses the value to send it to an adder.
2. $E_{3a} = \mathbf{m}^T G_{ffz}^T = G_{ffz}\mathbf{m} = Mposz_a$, Eq. (47). 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}. 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. $E_1 = G_{nfz}^T G_{nfz}$, Eq. (45). Since G_{nfz} is a vector, a single RAM port is required to load G_{nfz} 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. $E_{2a} = G_{nfz}^T G_{ffz}$, Eq. (46). From step 3, G_{nfz} 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}. E_{2a} is used in step 6 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. $E_3 = E_{4a} = E_{3a}G_{ffz}$, Eq. (48). 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. $E_2 = E_{2a}\mathbf{m}$, Eq. (46). 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. $Mposv_a = G_{ffv}\mathbf{m}$, Eq. (59). 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. $E_4 = E_{4a}\mathbf{m} = E_3 m$, Eq. (48). 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 to $E_5 + E_1 + E_3 + E_2 + P_1$ to create the final E value.
9. $F_{2a} = F_{1a}\Phi_z$, Eq. (57). 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 Φ is loaded into the other input buffer from RAM. This step continues until all three columns of Φ 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. $F_{2c} = F_{2a}\tilde{A}$, Eq. (58). 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 $\tilde{A}$ is loaded into the other input buffer of MACx. This step continues until the three columns of $\tilde{A}$ 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.

4.2.2 Stage 2: unconstrained solution

In stage 2, we compute the unconstrained optimal solution. Next, we determine whether the unconstrained solution meets the constraints or violates the constraints. If the constraints are violated, we invoke stage 3 and perform HQP algorithm to compute a suitable solution. If the constraints are met, we then bypass stage 3 and execute stage 4 to compute the control moves. It is necessary to perform the following steps in stage 2. These steps are also illustrated in Fig. 16.

1. Determine whether the battery has reached a full charge, i.e., $x_{m0} \geq 0.9$, which indicates that the state of charge (SOC) is greater than or equal to 90% full. This limit is $x_{m0} \geq 0.9$ designed to prevent overcharging of the battery [3].
2. Compute the current open circuit voltage (OCV) value based on the input SOC or $x_{m0.}$
3. Compute the unconstrained general optimal solution for the control input, $\Delta u^o = -E^{-1}F$, from Eq. (30).
4. Compute the γ constraint vector from Eq. (31).
5. Compute $M\Delta u^o$ from Eq. (31).
6. Compute K from Eq. (35).
7. Compute an element by element comparison, $M\Delta u^o \leq \gamma$, from Eq. (31).

From the above steps, vector K is computed in stage 2, although it is utilized in stage 3, since K needs to be computed only once per time sample. The time sample for controlling the charging of a battery is 1 s. 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.

4.2.2.1 Computing OCV for HW_v1 and HW_v2 In step 2, OCV is computed based on the current SOC (x_{m0}) 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; except for HW_v2, the required tables and values are stored in the BRAM, whereas for HW_v1, these values are stored in registers. In this case, 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.

4.2.2.2 Computing unconstrained general optimal solution for HW_v1 In step 3, we compute the unconstrained general optimal solution for the control input. In this case, 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 χ_k and u_k, as well as the final summation terms of Eq. (25). These computations are illustrated in Eq. (68) and are derived from Eqs. (56), (58), and (54).

$$F = -2\left(F_1 - (F_{2c})\chi_k - (F_{3a})u_k\right) \tag{68}$$

For HW_v1, as demonstrated in Fig. 17, 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 Δu°.

Computing γ constraint vector for HW_v1

Next, we compute the γ constraint vector. For HW_v1, the internal architecture for **γ** module is depicted in Fig. 18. As illustrated, the **γ** module computes the $G_{ffm}u_k$ vectors and $\Phi\tilde{A}\chi$ 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 $\Phi_v\tilde{A}\chi + G_{ffv}\mathbf{m}u_k$ and $\Phi_z\tilde{A}\chi + G_{ffz}\mathbf{m}u_k$ in parallel. An adder is utilized to compute the scalar addition. Next, three **V + S** modules are employed to compute the final terms in γ constraint vector in parallel, in order to generate Eq. (26) from Section 2.3.

Computing MΔu° for HW_v1

For HW_v1, the **MΔu°** is designed in such a way to be performed in parallel with **γ**. As shown in Fig. 19a, the **MΔu°** 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

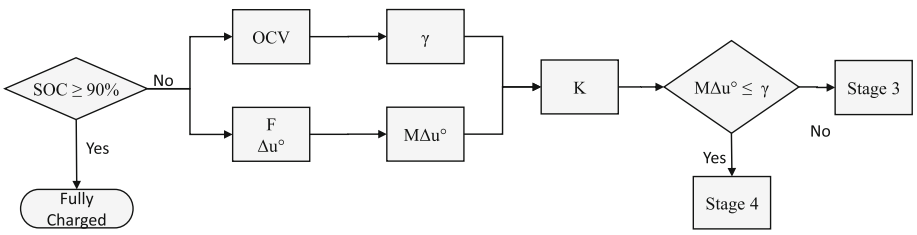

Fig. 16 Overview of stage 2

Table 3 OCV computation from SOC

Open circuit voltage from state of charge algorithm
1. Determine the boundary conditions: if ($x_{m0} < 0$) use the minimum pre-calculated OCV. else if ($x_{m0} > 1$) use the maximum pre-calculated OCV else if ($0 \leq x_{m0} \leq 1$) compute OCV using steps 3 to 5.
2. Find the Index $Index = int(200*x_{m0})$
3. Find the difference (D) and offset (S) $D = I - 200*x_{m0}$ $S = 1 - D$
4. Compute the OCV using temperature (T) $OCV = (OCV_0[I] * S + OCV_0[I+1] * D) + T * (OCV_{rel}[I] * S + OCV_{rel}[I+1] * D)$

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 Eq. (31), $M\Delta u^o \leq \gamma$. As illustrated in Fig. 19b, 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 $\mathbf{M\Delta u^\circ}$ and $\boldsymbol{\gamma}$ were compared element by element using a FPU comparator. The internal architecture of the comparison module is illustrated in Fig. 19c. 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 the constraints are violated, the comparison module triggers stage 3 and relinquishes the execution of the remaining compare operations.

4.2.2.3 Computing unconstrained solutions for HW_v2 In stage 2, similar to stage 1, 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 5.

As depicted in Fig. 20, 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, 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 Eq. (68)
2. $\Phi_z\tilde{A}\chi$ for γ in Eq. (26)
3. $\Phi_v\tilde{A}\chi$ for γ in Eq. (26)

Since the maximum length of the individual vectors is 3, the 5-stage pipelined MACx module uses only the first three 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

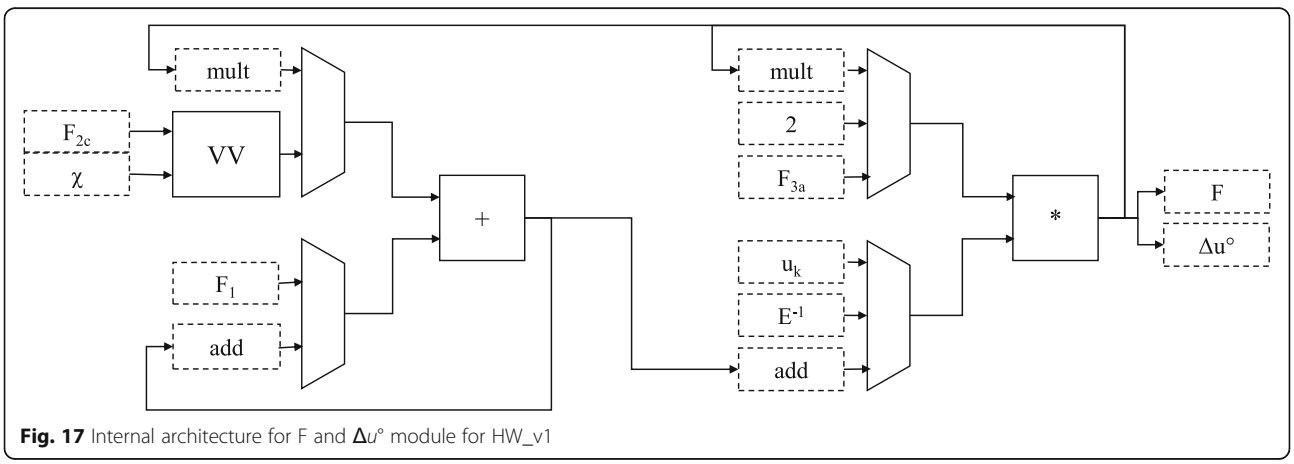

Fig. 17 Internal architecture for F and Δu° module for HW_v1

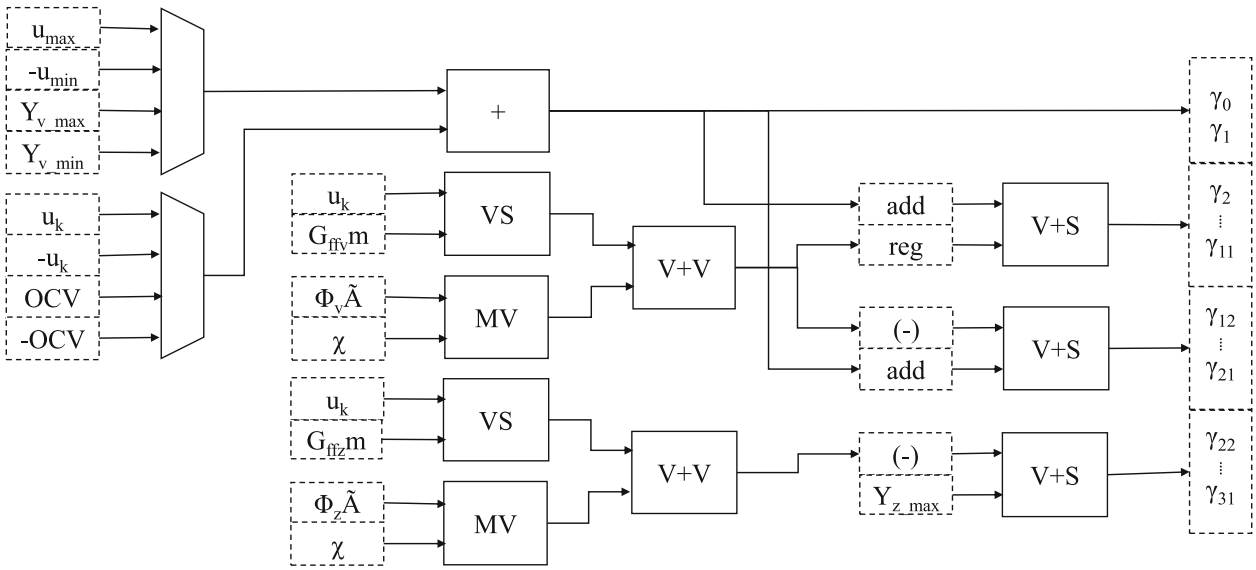

Fig. 18 Internal architecture for γ module for HW_v1

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 Eq. (35), $K = \gamma - M\Delta u^o$, and the comparison Eq. (31), $M\Delta u^o \leq \gamma$, if $K > 0$, then the comparison is true. Hence, by comparing the MSB of K, we can determine whether the constraints are met or not. If all the elements meet the constraints, then the optimal solution is selected and stage 4 is executed by-passing 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 in stage 2. This significantly reduces the time taken to perform the compare operations (as in HW_v1) utilizing a separate module. As illustrated in Fig. 20, HW_v2 has an integrated solution for stage 2, whereas HW_v1 has a modular solution (depicted in Figs. 17, 18, and 19).

4.2.3 Stage 3: Hildreth's quadratic programming

In stage 3, we compute the constrained optimal control input using Hildreth's quadratic programming (HQP) approach. With this approach, the Δu°, which is known as the global optimal solution, is adjusted by λME^{-1} (as in Eq. (38)), where λ is a vector of Lagrange multipliers.

Initially, 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 λ elements), thus reducing the computation complexity (3 or less computations versus 32 computations). Apart from reducing the computation complexity of stage 3, this

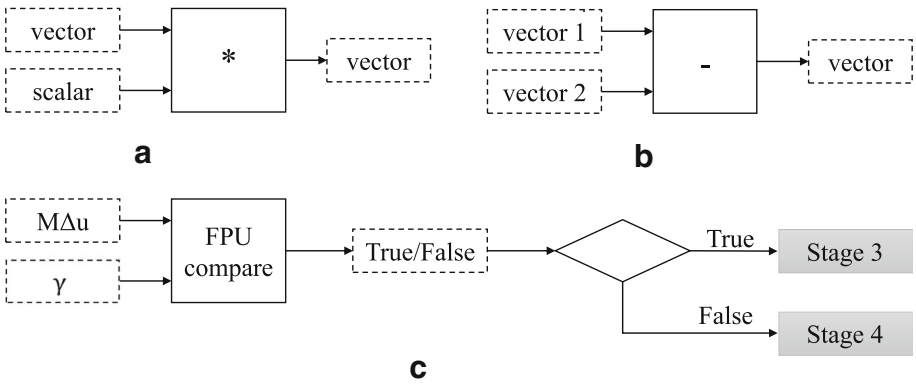

Fig. 19 VS, V-V, and comparison of constraints modules for HW_v1. **a** VS for MΔu°, **b** V-V for *K*, **c** Comparison of constraints

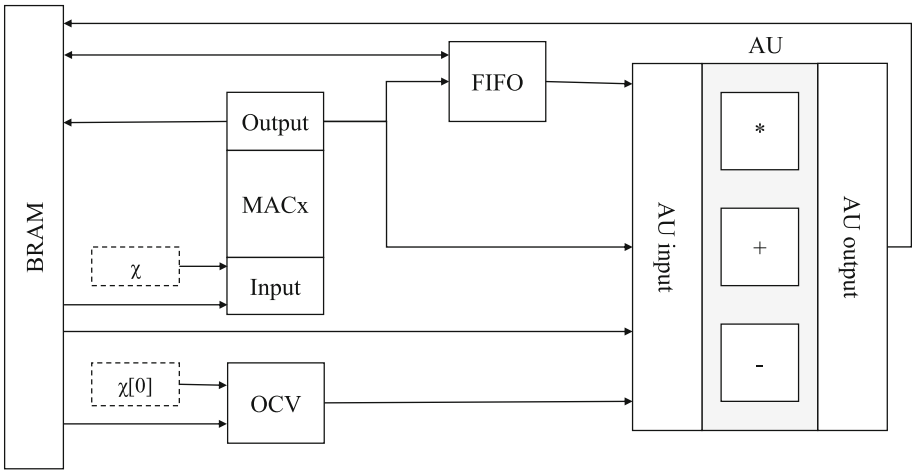

Fig. 20 Functional architecture for stage 2 for HW_v2

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 (λ) 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 (λ), for the HQP solution one element at a time. This HQP technique eliminates the need for matrix inversion in optimization. In this case, the λ 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 λ 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, in order to provide the greatest possibility for convergence, as well as to provide a reasonable solution in case if there is no convergence.

The HQP is an iterative process. This is typically implemented as two nested loops. The inner loop computes the individual elements of the λ vector, in which the number of iterations depends on the length of λ. The outer loop determines whether the λ 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 Eqs. (36) and (37).
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 Eqs. (36) and (37) (from Section 2) 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 ME^{-1} is computed in stage 1, it is reused in stage 3, instead of re-computing in each iteration. The elements of the λ vector are calculated using the P matrix from stage 1 and K vector from stage 2.

4.2.3.1 For HW_v1 HW_v1 consists of three main modules, including **Wp3**, **Converge_v1**, and **New_-Δu_v1**, and a sub-module (**SVM_v1**) for sparse vector multiplication.

From our experimental results (presented in Section 5), it is observed that any λ vectors typically have a maximum number of three non-zero elements. Hence, our hardware is designed to operate only on the non-zero elements of λ and P. In order to generate all the elements of the λ vector, the computations 2.a to 2.f (as in Table 4) must be repeated 32 times. By focusing only on the non-zero elements, our hardware design dramatically reduces the time taken to generate the required λ elements, since certain steps are by-passed in Table 4.

The functional flow of the sparse vector multiplication module (**SVM_v1**) is illustrated in Fig. 21a. As demonstrated, **SVM_v1** module checks each element

Table 4 HQP algorithm

Hildreth's quadratic programming technique (HQP algorithm)
For iterations 1 to 40
1. Save $\lambda_{current} \rightarrow \lambda_{previous}$
2. Start outer loop to build λ, i = 0 to # elements in M or M_{size}
a. w = 0;
b. start inner loop to build λ, j starts at 0
i. w = w + P[i][j]·λ[j]
ii. GOTO start inner loop If j<M_{size},
c. w = w + K[i]-P[i][i]·λ[i]
d. λ_{test} = -w/P[i][i]
e. if λ_{test} < 0 then λ[i] = 0, else λ[i] = λ_{test}
f. GOTO start outer loop if i<M_{size}
3. Check convergence
a. calculate the Euclidean length of previous λ
b. calculate the Euclidean length of current λ
c. Compare ratio to reference value
d. if converged, exit iteration, GOTO calculate new **Δ***u*
4. Else execute next iteration, GOTO 1.
5. Calculate new **Δ***u*
a. Start loop, j = 0 to j=M_{size}
i. $\Delta u_c = \Delta u_c + \lambda[j]\ ME^{-1}[j]$
b. GOTO start loop if j<M_{size}
c. $\Delta u_{k+1} = \Delta u^{\circ} - \Delta u_c$
6. End

of the input vectors and only forwards the non-zero elements to the MAC unit. As depicted in Fig. 21b, **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 λ vector is updated, after 32 iterations. Then, the updated λ vector is forwarded to the convergence module (**Converge_v1**). Next, the **Converge_v1** module computes step 3 of the HQP algorithm (in Table 4); simultaneously, the **New_Δu_v1** module computes step 5 of the HQP algorithm (to generate Δu_{k+1}) in anticipation of a convergence. At the same time, the λ module **(Wp3)** starts computing the next λ vector, in the event the current λ does not converge. If the convergence fails, the Δu_{k+1} value is discarded. If the convergence succeeds, a signal is sent to **Wp3** module to terminate the next λ vector computation, and then, the subsequent stage (stage 4) is started with input Δu_{k+1}.

As shown in Fig. 22a, 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_Δu_v1** module is depicted in Fig. 22b. As depicted, the **New_Δu_v1** computes the Δu_{k+1} value (from steps 5.a to 5.c of HQP algorithm) and consists of a **SVM_v1** module and a subtractor. In this case, the **New_Δu_v1** module is executed in parallel with the **Converge_v1** module.

4.2.3.2 For HW_v2 The high-level architecture for HW_v2 for stage 3 is illustrated in Fig. 23. Apart from the fundamental operators, this consists of five custom modules.

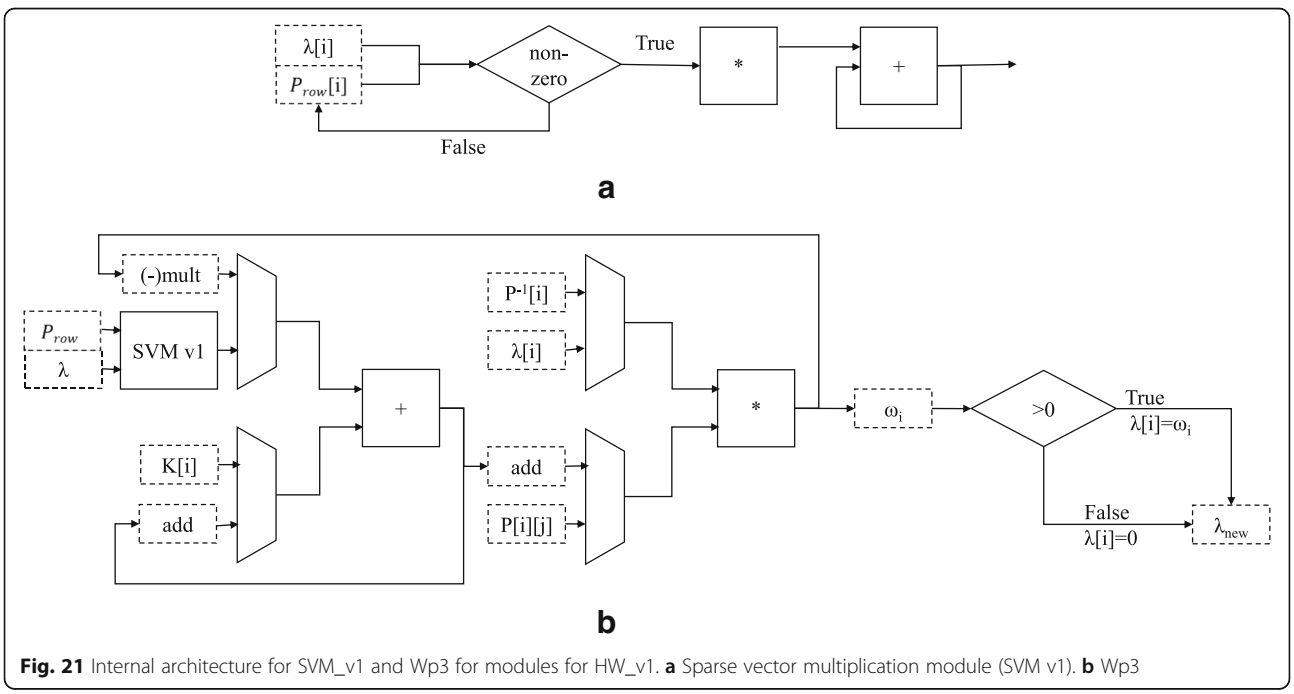

Fig. 21 Internal architecture for SVM_v1 and Wp3 for modules for HW_v1. **a** Sparse vector multiplication module (SVM v1). **b** Wp3

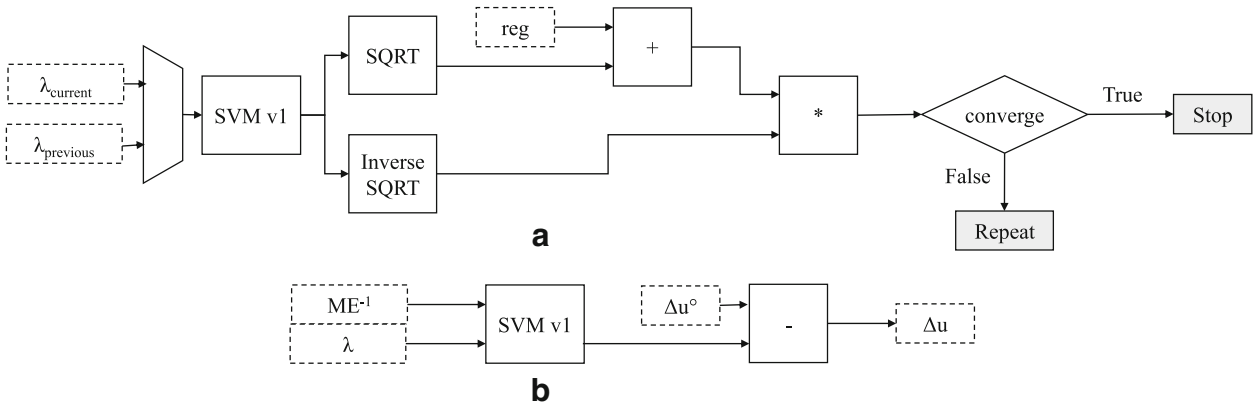

Fig. 22 Stage 3 Converge_v1 and New_Δu_v1 modules for HW_v1. **a** Converge_v1, **b** New_Δu_v1

For HW_v2, **Win** module computes Eqs. (36) and (37) (i.e., computes sub-steps 2.a to 2.f of the HQP algorithm (in 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 Fig. 24. In this case, the FPU multiplier, adder, and subtractor are external to the **Win** module as illustrated in Fig. 23.

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 λ 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 λ vector. These non-zero λ 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 are reduced from 32 to 3 or less.

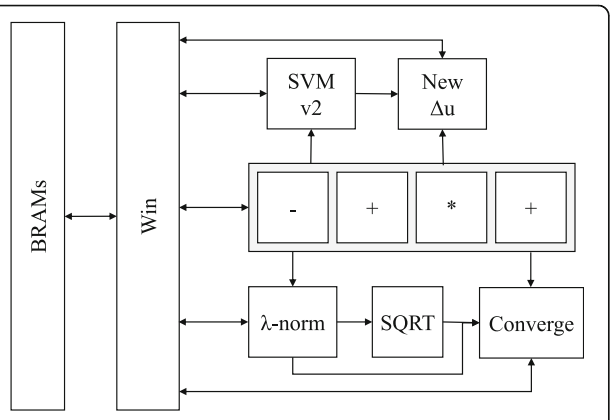

Fig. 23 Top-level architecture for stage 3 for HW_v2

The internal architecture of the **SVM_v2** module is demonstrated in Fig. 25a. Although the multiplier is external to this module, it is included in dotted lines (in Fig. 25a) 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, then 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 Fig. 25b. To determine the convergence of the λ vector, the Euclidean distance is computed. In HW_v2, the Euclidean distance is measured as each element of the λ is computed, one element at a time. Conversely, in HW_v1, this distance is measured after all 32 elements are computed. In this case, the **λ-norm** module (in Fig. 25b) takes the scalar λ_i as inputs, squares the λ_i, and then adds the squared value to the previous element. After computing the final λ element, which is λ_{31}, the output of **λ-norm** is then forwarded to a square root module. The result from the square root module is the Euclidean distance. This result

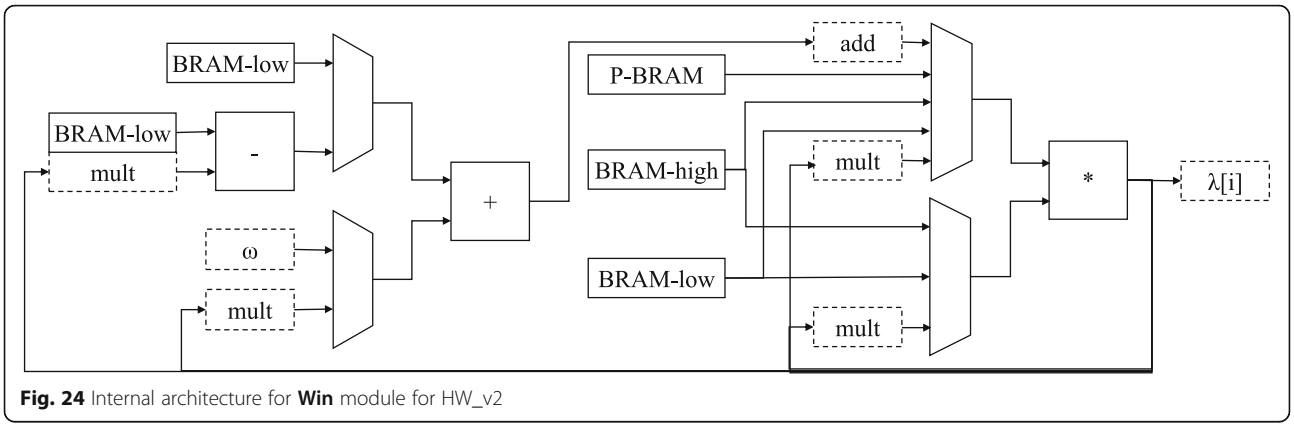

Fig. 24 Internal architecture for **Win** module for HW_v2

is the Euclidean distance used as the current value of the current λ (sub-step 3.b of the HQP algorithm in Table 4) in the current iteration; this result is stored and used as the previous value of previous λ (sub-step 3.a of the HQP algorithm) in the next iteration.

In this case, the **Win** module sends λ_i to the multiplier and signals the **λ-norm** module that the required data is ready. Next, the **λ-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 λ are completed, the final accumulator result of the **λ-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 λ length value using a divider.

The entire process is repeated up to 40 times. The system either converges, or after the 40th iteration is considered to be converged. Next, we start executing the **New_Δu_v2** module. In this case, the **Win** module loads the λ and the ME^{-1} values into the multiplier for the **SVM_v2** module to process and sends a signal to the **New_Δu_v2** module to initiate the execution. Depending on the length of the active set (non-zero elements) in the λ vector, the **New_Δu_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 Δu_{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 4 clock cycles and asserts a ready signal

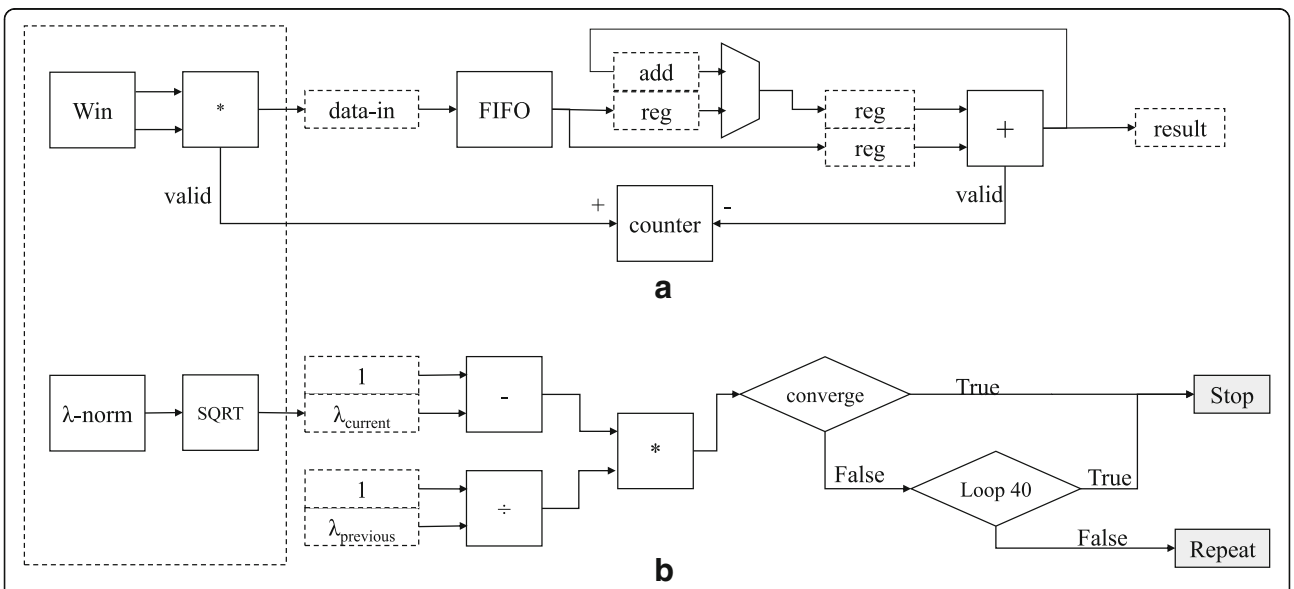

Fig. 25 Internal architecture for SVM_v2 and Converge_v2 modules for HW_v2. **a** SVM_v2. **b** Converge_v2

to indicate that the result of the **SVM_v2** module is ready to be used and also the **SVM_v2** module is ready for the next computation.

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 Δu (computed in stages 2 or 3) and also utilizing χ, 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 4.2) determines whether the plant state value (x_{m0}) is fully charged or whether we have reached the maximum number of iterations.

During stage 4, we compute the plant output, which is to determine the current terminal voltage (v_k) and then the state of charge (z_k) from Eqs. (7) and (8), respectively. Then, the control signal and the state signals are updated. In this case, the first element of the $\Delta\mathbf{U_k}$ is used to update the control signal from Eq. (39) and the new control signal is used to determine the states for the next iteration from Eq. (40) (in Section 2.5).

4.2.4.1 For HW_v1 The overview of stage 4 for HW_v1 is depicted in Fig. 26. 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 Fig. 26, 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 Fig. 27a, for HW_v1, the plant module consists of a customized module to perform the two $\tilde{C}\chi_k$ 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 $\tilde{C}_v\chi_k$ is performed; second, the $\tilde{C}_z\chi_k$ is performed, simultaneously adding OCV to $\tilde{C}_v\chi_k$. As illustrated in Fig. 27b, 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}.

4.2.4.2 For HW_v2 The internal architecture of stage 4 for HW_v2 is depicted in Fig. 28. As illustrated, HW_v2 consists of four major modules: an **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.

For HW_v2, Eqs. (3) and (4) (from Section 2.1) are utilized to determine the terminal voltage and the SOC, whereas for HW_v1, Eqs. (6) and (7) (from Section 2) are utilized with the same outcome. As illustrated in Fig. 28, the **Input** module determines the computations and provides the 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

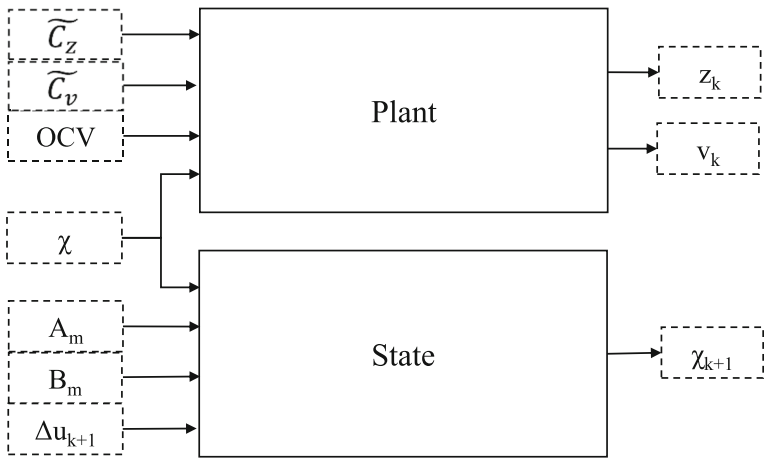

Fig. 26 Top-level architecture for stage 4 for HW_v1

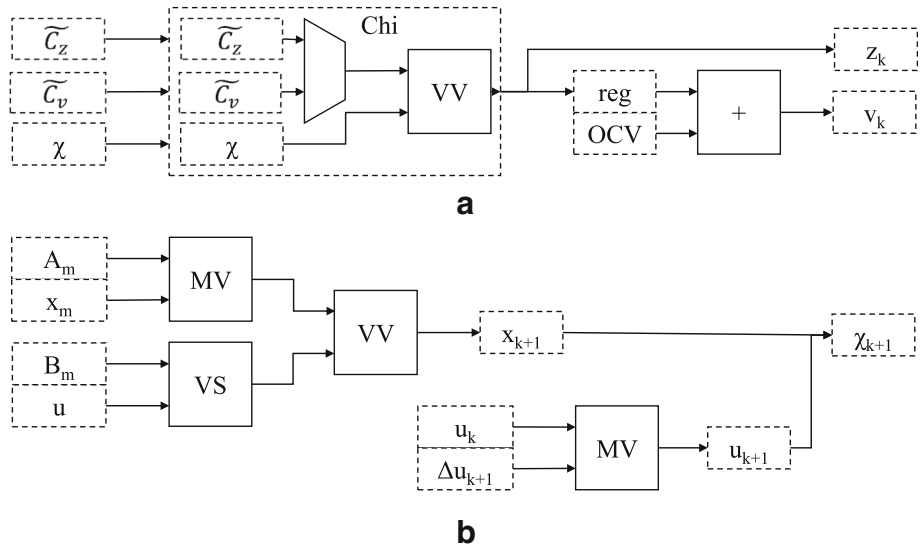

Fig. 27 Internal architecture for plant and state modules for HW_v1. **a** Plant, **b** State

executed in parallel, and the intermediate/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), in order to obtain the results with the least amount of time.

(1) $C_v x_k \rightarrow$ by **SVM_v2** module	(8) $C_v x_k + D_v u_k \rightarrow$ by adder
(2) $D_v u_k \rightarrow$ by multiplier	(9) $C_z x_k + D_z u_k \rightarrow$ by adder
(3) $C_z x_k \rightarrow$ by **SVM_v2** module	(10) $B_{m0} u_{k+1} \rightarrow$ by multiplier
(4) $D_z u_k \rightarrow$ by multiplier	(11) $B_{m1} u_{k+1} \rightarrow$ by multiplier
(5) $A_{m\_row0} x_k \rightarrow$ by **SVM_v2** module	(12) $C_v x_k + D_v u_k + OCV(z_k) \rightarrow$ by adder
(6) $A_{m\_row1} x_k \rightarrow$ by **SVM_v2** module	(13) $A_{m\_row0} x_k + B_{m0} u_{k+1} \rightarrow$ by adder
(7) $u_k + \Delta u_{k+1} \rightarrow$ by adder	(14) $A_{m\_row1} x_k + B_{m1} u_{k+1} \rightarrow$ 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.

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 [4], in order to evaluate and validate the correctness and functionalities of our designs. The evaluation setup for our embedded

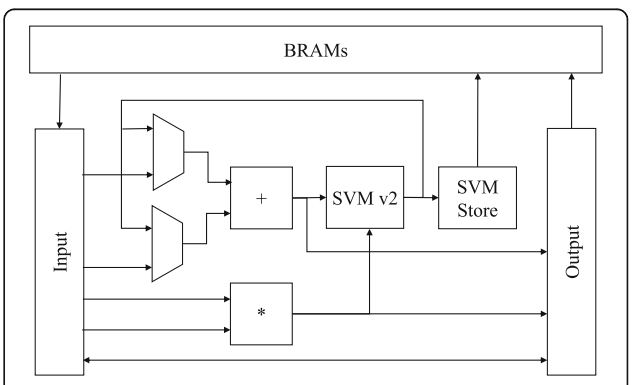

Fig. 28 Internal architecture for stage 4 for HW_v2

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 s, temperature at 25 °C, and iterations of 3600.

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 [4].

As stated in [4], 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 29 illustrates the desired charging profile for the fast-charge MPC implementation in [4]. 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 Fig. 29, 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 [4].

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 Figs. 30a, 31a, and 32a 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 [4], since the graphs are overlapping. There are some slight discrepancies, which are negligible.

Figures 30a, and 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.

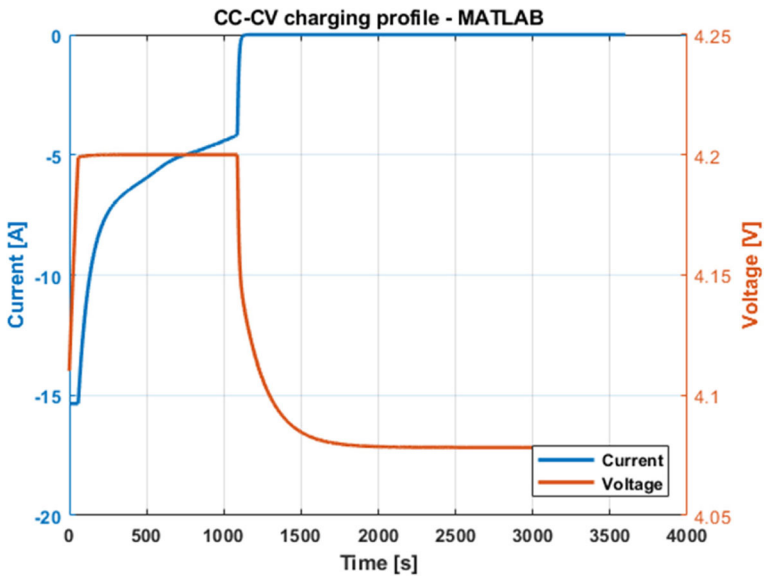

Fig. 29 CC-CV charging profiles for baseline Matlab model

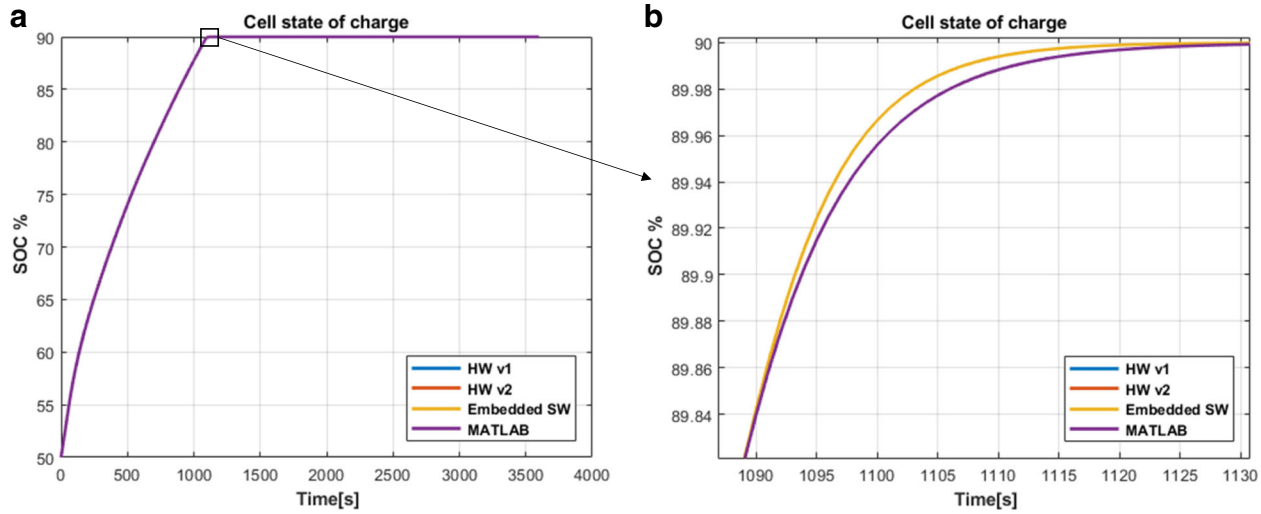

Fig. 30 State of charge comparison: embedded hardware and software designs versus baseline Matlab model. **a** Overview. **b** Difference at bend

Although at a glance the SOC graphs (Fig. 30a) seem identical for all four designs, at a closer look, there are some discrepancies. As illustrated in Fig. 30b, 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 1216 s, which is determined from the baseline experiments.

Figures 31a, 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 Fig. 31b, the output voltage does not exceed 4.2 V; this illustrates that the system's behavior respects the constraints in order to extend the useful life of the battery.

Similar to SOC graphs, at a glance, the terminal voltage graphs (Fig. 31a) seem identical for all four designs; at a closer look (Fig. 31b), there are some discrepancies. For instance, at time $t = 0$ s, the initial cell terminal voltage value for the embedded systems designs is 3.92 V, whereas that for the baseline Matlab design is 4.11 V. 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 Fig. 31b, the Cell Terminal voltage increases gradually and smoothly with the baseline Matlab

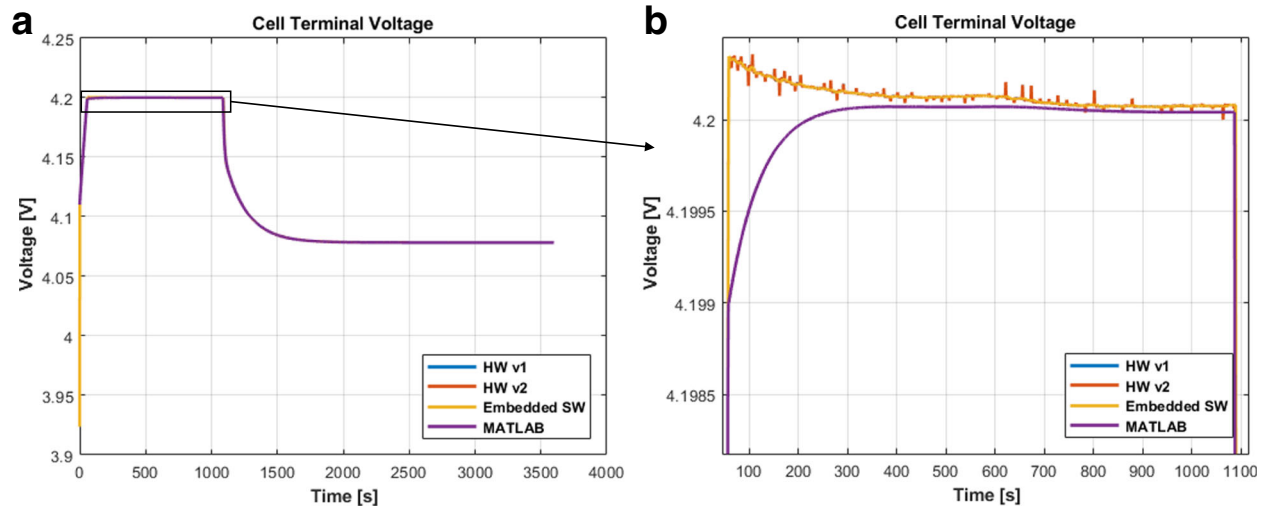

Fig. 31 Terminal voltage comparison: embedded hardware and software designs versus baseline Matlab model. **a** Overview. **b** Greatest difference

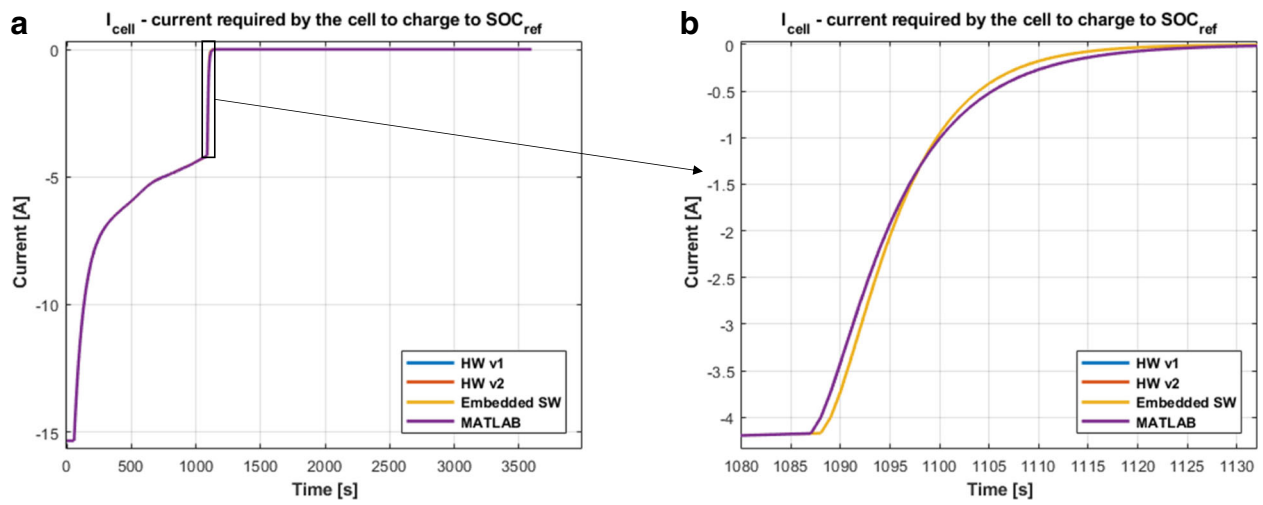

Fig. 32 Control signal (I_{cell}) comparison: embedded hardware and software designs versus baseline Matlab model. **a** Overview. **b** Close up for comparison

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 1.2 mV.

Figures 32a, 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.

The current starts out at a constant, with a maximum allowed value of (− 15 A). 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 1200 s, which is when the SCO is 90% and the battery is considered fully charged.

Similar to terminal voltage graphs, at a glance, the control signal graphs (Fig. 32a) seem identical for all four designs; at a closer look (Fig. 32b), there are some discrepancies. As illustrated in Fig. 32b, the discrepancies are prominent between the timeline 1090 and 1120 s. However, these discrepancies do not affect the overall functionality and the final results of the designs, thus negligible.

5.1.1 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 millivolts for the voltage and milliamps for the current. These slight discrepancies are mainly because we use single-precision floating-point units for our embedded hardware and software architectures, whereas 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, thus negligible.

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.

5.2.1 Execution times and speedup: embedded hardware versus software 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 100 MHz, whereas embedded software architecture is executed on the

Table 5 Execution times: embedded hardware and software design and baseline Matlab model

Configuration	Execution time/ms	Speedup over embedded Sw	Speedup over baseline Matlab
Embedded Sw on MicroBlaze (at 100 MHz)	3958.04	–	0.21
Baseline Matlab on i7 processor (at 3.1 GHz)	848.331	4.67	–
HW_v1 (at 100 MHz)	468.557	8.45	1.81
HW_v2 (at 100 MHz)	39.774	99.51	21.33

MicroBlaze processor running at 100 MHz on the same FPGA for fair comparison purposes. 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 three 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 Intel i7 processor running at 3.1 GHz on a desktop computer.

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 Intel i7 processor. It should be noted that all our embedded systems designs are running at 100 MHz, whereas the Matlab model is running at 3.1 GHz.

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 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 1 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 enables 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.

5.2.2 Resource utilization: register-based HW_v1 versus 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.

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

Table 6 Resource utilization: embedded HW_v2 versus embedded HW_v1

Configuration	Number of occupied slices	Number of BRAMs (36E1)	Number of DSP48E1
HW v1	34,315	62	688
HW_v2	10,277	35	73

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.

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 more superior than the HW_v1, embedded software design, and also 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-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 5.2 to 450 μs (4.5×10^{-4} s) as shown in Fig. 33a. The maximum iteration time per cycle, which is 450 μs, is illustrated in Fig. 33b. This maximum iteration time incurs when 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 0.9995 s in the 1-s sampling cycle.

In this case, the execution overhead of the augmented model in stage 1 is approximately 24 μs and considered to be minimal. This execution overhead is the time

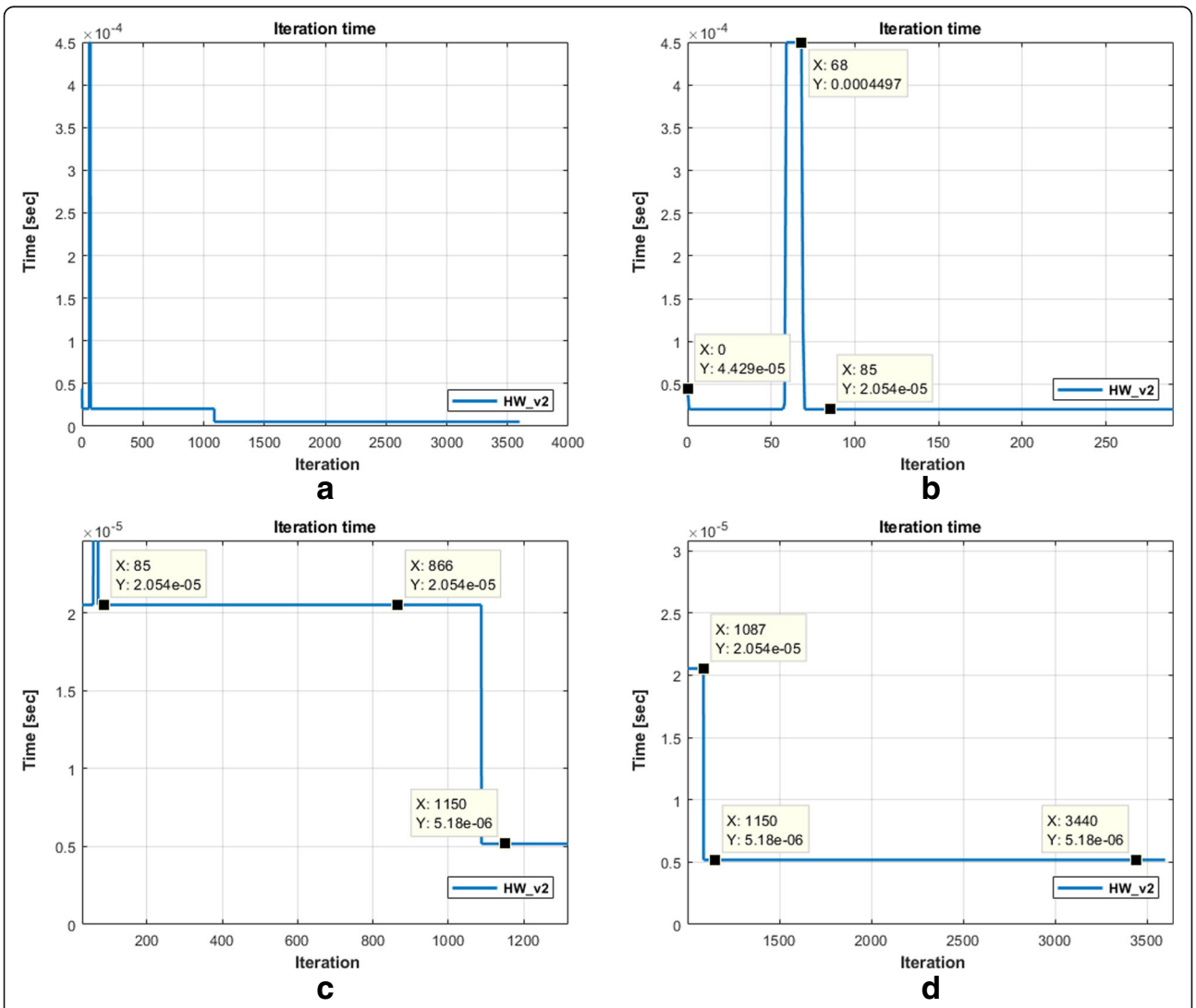

Fig. 33 Iteration time per cycle for BRAM-based HW_v2. **a** Full simulation. **b** Longest iteration. **c** Most common iteration time. **d** Shortest iteration time (stage 3 not required)

difference between the first iteration (which includes the processing time through stages 1 to 4 during an iteration, as illustrated in Fig. 33b) and the second iteration (which includes the processing time through stages 2 to 4 during an iteration, as illustrated in Fig. 33c). For both the first and the second iterations, stage 3 (for HQP) converges in two loops, which takes approximately 20.5 μs. Hence, we logically make an assumption 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 5.18 μs to process (as in Fig. 33d), 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 15.3 to 444.5 μs. The minimum time (15.3 μs) is associated with 1λ element and 2 iterations, and the maximum time (444.5 μs) 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 474 μs (i.e., adding 24 to 450 μs). In this case, the fast-charge MPC algorithm could execute more than 2100 times, in 1-s sample time, thus allowing our embedded architecture to control multiple battery cells individually.

5.2.4 Analysis of existing works on embedded designs for MPC

In Section 1, 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 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 [35], 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 a Newton's algorithm instead of a HQP, as in our design. Unlike our design, in [35], the model parameters were pre-calculated offline and stored in the microprocessor. In [8], 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 [16], fixed-point MPC solution was proposed with two separate QP solvers as user-designed modules: primal-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.

6 Conclusions

In this paper, 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. Our embedded hardware and software architectures are generic and parameterized. Hence, without changing the internal architectures, our embedded designs can be utilized for many other control systems applications that employ similar fast-charge 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 trade-offs, 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 128 Kb) 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, 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 1-s sample time, since the worst-case iteration time per cycle is a mere 474 μs. As future work, we are planning to investigate how to interface with all or some of the battery cells in a pack at a time, how to share the bus in such a way to avoid the contention issues, and so on. We are also exploring sophisticated power analysis tools, such as Synopsys Power Compiler, to measure the power consumption of our proposed embedded designs, since power consumption is another major issue in embedded devices.

Our proposed embedded architectures (both hardware and software) for the fast-charge MPC can be utilized as a smart sensor at the battery cell level, locally. Monitoring and controlling certain important parameters of the battery cells at the lowest level will indeed ease the computational burden at the system level. This will also reduce the communication overhead between the battery cells and the global control system and will provide more autonomous control to the battery cells. Also as future work, we will be investigating the feasibility and efficiency of utilizing our embedded architectures for the fast-charge MPC for other control systems applications such as unmanned aerial vehicles (UAVs) and autonomous vehicles [36].

Acknowledgements
The authors would like to thank Dr. Scott Trimboli, Assistant Professor, in Electrical and Computer Engineering Department at the University of Colorado at Colorado Springs, and his former PhD student, Dr. Marcelo Xavier, for providing access to the baseline Matlab model for the fast-charge MPC algorithm.

Availability of data and materials
All data generated or analyzed during this study are included in this published article.

Authors' contributions
AM is DP's PhD student. AM and DP have been conducting this research. Under the guidance of DP, AM has designed, developed, and implemented the embedded hardware and software architectures for the fast-charge MPC algorithm and performed the experiments. With the assistance of AM, DP wrote the paper. Both authors read and approved the final manuscript.

Authors' information
Anne K. Madsen received her M.Sc. in Electrical Engineering, and B.Sc. in General Engineering from Naval Postgraduate School (Monterey, CA) and US Naval Academy, respectively. Anne is pursuing her Ph.D. and working as a teaching assistant in the Department of Electrical and Computer Engineering, University of Colorado at Colorado Springs. Anne is also an Independent Engineering Consultant to Rim Technologies. She served as an Officer in the US Navy and taught at both the Air Force and Navy Service Academies, in Math and Engineering divisions, respectively. She also worked as an Engineer and Acquisition Specialist for Air Force Space and Missile Command's Ground-Based Space Surveillance Division for 14 years. Her research interests are cyber-physical systems, control theory, and hardware optimization.

Darshika G. Perera received her Ph.D. degree in Electrical and Computer Engineering from University of Victoria (Canada) and M.Sc. and B.Sc. degrees in Electrical Engineering from Royal Institute of Technology (Sweden) and University of Peradeniya (Sri Lanka) respectively. She is an Assistant Professor in the Department of Electrical and Computer Engineering, University of Colorado at Colorado Springs (UCCS), USA, and also an Adjunct Assistant Professor in the Department of Electrical and Computer Engineering, University of Victoria, Canada. Prior to joining UCCS, Darshika worked as the Senior Engineer and Group Leader of Embedded Systems at CMC Microsystems, Canada. Her research interests are reconfigurable computing, mobile and embedded systems, data mining, and digital systems. Darshika received a best paper award at the IEEE 3PGCIC conference in 2011. She serves on organizing and program committees for several IEEE/ACM conferences and workshops and as a reviewer for several IEEE, Springer, and Elsevier journals. She is a member of the IEEE, IEEE CAS and Computer Societies, and IEEE Women in Engineering.

Ethics approval and consent to participate
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 27 December 2017 Accepted: 21 May 2018
Published online: 16 July 2018

References
1. Brand, C., Cluzel, C., Anable, J., Modeling the uptake of plug-in vehicles in a heterogeneous car market using a consumer segmentation approach, Transp. Res. A Policy Pract., 97, 2017, 121–136, https://doi.org/10.1016/j.tra.2017.01.017. ISSN 0965-8564. http://www.sciencedirect.com/science/article/pii/S0965856416302130.
2. Du, J., Wang, Y., Tripathi, A., Lam, J., Li-ion Battery Cell Equalization by Modules with Chain Structure Switched Capacitors, *2016 Asian Conference on Energy, Power and Transportation Electrification (ACEPT)*. Singapore, pp. 1-6, 2016.
3. Xavier, MA, & Trimboli, MS. (2015). Lithium-ion battery cell-level control using constrained model predictive control and equivalent circuit models. *J. Power Sources*, *285*, 374–384.
4. Xavier, MA (2013). *Lithium-ion battery cell management: a model predictive control approach*, *University of Colorado Colorado Springs, Department of Elelctrical and Computer Engineering* (). Colorado Springs: UCCS.
5. Takacs, G, & Rohal'-Ilkiv, B (2012). *Model predictive vibration control*. London: Springer.
6. Novak, J, & Chalupa, P. (2014). Implementation aspects of embedded MPC with fast gradient method. *Int J Circuits Syst Signal Process*, *8*, 504–511.
7. Zometa, P, Kogel, M, Faulwasser, T, Findeisen, R (2012). *Implementation aspects of model predictive control for embedded systems*, *American Control Conference, 2012* (pp. 1205–1210). Montreal: IEEE.
8. Chen, X, & Wu, X (2011). *Design and implementation of model predictive control algorithms for small satellite three-axis stabilization*, *Proceedings of the IEEE International Conference on Information and Automation* (). Shenzhen: IEEE.
9. Bleris, LG, Vouzis, PD, Arnold, MG, Kothare, MV (2006). *A co-processor FPGA platform for the implementation of real-time model predictive control*, *American Control Conference* (pp. 1912–1917). Minneapolis: IEEE.
10. Abdolhosseini, M, Zhang, YM, Rabbath, CA (2012). Trajectory tracking with model predictive control for an unmanned quad-rotor helicopter: theory and flight test results. In C-Y Su, S Rakheja, H Liu (Eds.), *Intelligent robotics and applications: lecture notes in computer science*, (pp. 411–420). Berlin: Springer.
11. Ling, KV, Wu, BF, Maciejowski, JM (2008). *Embedded model predictive control (MPC) using a FPGA*, *Proceedings of the 17th World Congress, The International Federation of Automatic Control* (pp. 15250–15255). Seoul: IFAC.
12. Jerez, J. L., Constantinides, G. A., and Kerrigan, E. C. (2011). An FPGA implementation of a sparse quadratic programming solver for constrained model predictive control. FPGA '11 (pp. 209–218). Monterey: ACM.
13. Abbes, AK, Bouani, F, Ksouri, M. (2011). A microcontroller implementation of constrained model predictive control. *Int J Electr Electron Eng*, *5*(3), 199–206.

14. Chui, CK, Nguyen, BP, Ho, Y, Wu, Z, Nguyen, M, Hong, GS, et al. (2013). *Embedded real-time model predictive control for glucose regulation*, *World Congress on Medical Physics and Biomedical Engineering May 26–31, 2012* (pp. 1437–1440). Beijing: Springer.
15. Nguyen, BP, Ho, Y, Wu, Z, Chui, CK (2012). *Implementation of model predictive control with modified minimal model on low-power RISC microcontrollers*, *Proceedings of the Third Symposium on Information and Communication Technology* (pp. 165–171). Ha Long: ACM.
16. Hartley, EN, Jerez, JL, Suardi, A, Maciejowski, JM, Kerrigan, EC, Constantinides, GA. (2014). Predictive control using and FPGA with application to aircraft control. *IEEE Trans. Control Syst. Technol.*, *22*(3), 1006–1017.
17. Bleris, LG, & Kothare, MV (2005). *Real-time implementation of model predictive control*, *American Control Conference* (pp. 4166–4171). Portland: IEEE.
18. Bleris, L. G., Kothare, M. V., Garcia, J., and Arnold, M. G. (2004). Embedded model predictive control for system-on-a-chip applications.
19. Chen, X. and X. Wu, (2012) Implementation and Experimental Validation of Classic MPC on Programmable Logic Controllers, 2012 20th Mediterranean Conference on Control & Automation (MED) (pp. 679-684), Barcelona, Spain, 2012.
20. Ekaputri, C., and Syaichu-Rohman, A. (2012) Implementation Model Predictive Control (MPC) Algorithm-3 for Inverted Pendulum. *2012 IEEE Control and System Graduate Research Colloquium* (pp. 116-122). Shah Alam, Selangor, IEEE.
21. Wang, Y, & Boyd, S. (2010). Fast model predictive control using online optimization. *IEEE Trans. Control Syst. Technol.*, *18*(2), 267–278.
22. Aridhi, E., Abbes, M., and Mami, A. (2012, March). FPGA implementation of predictive control. In *Electrotechnical Conference (MELECON), 2012 16th IEEE Mediterranean,* (pp. 191-196). Yasmine Hammamet, Tunisia, IEEE.
23. Martínez-Rodríguez, M. C., Brox, P., Tena, E., Acosta, A. J., and Baturone, I. (2015, March). Programmable ASICs for model predictive control. In *Industrial Technology (ICIT), 2015 IEEE International Conference on* (pp. 1593-1598). Seville, Spain, IEEE.
24. Huyck, B., Ferreau, H. J., Diehl, M., De Brabanter, J., Van Impe, J. F., De Moor, B., et al. (2012). Towards Online Model Predictive Control on a Programmable Logic Controller: Practical Considerations. *Mathematical Problems in Engineering,* 20 pages, 2012.
25. Lima, DM, Americano da Costa, MV, Normey-Rico, JE (2013). *A flexible low cost embedded system for model predictive control of industrial processes*, *2013 European Control Congerence (ECC)* (pp. 1571–1576). Zurich: IEEE.
26. Xavier, MA (2016). *Efficient strategies for predictive cell-level control of lithium-ion batteries*, *University of Colorado Colorado Springs, Department of Elelctrical and Computer Engineering* (). Colorado Springs: UCCS.
27. Wang, L (2009). *Model predictive control system design and implementation using Matlab*. London: Springer-Verlag.
28. Holkar, KS, & Waghmare, LM. (2010). An overview of model predictive control. *Int J Control Autom Syst*, *3*(4), 47–63.
29. Ordys, AW, & Pike, AW (1998). *State space generalized predictive control incorporating direct through terms*, *Proceedings of the 37th IEEE Conference on Decision and Control* (pp. 4740–4741). Tamp: IEEE.
30. Xilinx, Inc., "ML605 Hardware User Guide", UG534 (v1.5), 2011, www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf.
31. Xilinx, Inc., "LogiCORE IP Floating-Point Operator", DS335 (v5.0), http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf., 2011.
32. Xilinx, Inc., "LogiCORE IP AXI Interconnect", DS768 (v1.06.a), http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v1_06_a/ds768_axi_interconnect.pdf., 2012
33. Xilinx, Inc., "LogiCORE IP AXI Timer", DS764 (v1.03.a), 2012, http://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v1_03_a/axi_timer_ds764.pdf.
34. Perera, DG, & Li, KF (2008). Parallel computation of similarity measures using an FPGA-based processor array. In *Proceedings of 22nd IEEE international conference on advanced information networking and applications, (AINA'08)*, (pp. 955–962).
35. Vouzis, PD, Bleris, LG, Arnold, MG, Kothare, MV. (2009). A system-on-a-chip implementation for embedded real-time model predictive control. *IEEE Trans. Control Syst. Technol.*, *17*(5), 1006–1017.
36. Chen, B, Yang, Z, Huang, S, Du, X, Cui, Z, Bhimani, J, Xie, X, Mi, N (2017). *Cyber-physical system enabled nearby traffic flow modelling for autonomous vehicles*, *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)* (pp. 1–6).