2019

A hardware scalable, software configurable LQG controller using a sequential discrete Kalman filter

Matthew James Cauwels
Iowa State University

A hardware scalable, software configurable LQG controller using a sequential discrete Kalman filter

by

Matthew James Cauwels

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Computer Networking)

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis details the motivation, architecture, and analysis of a hardware scalable, software programmable Linear Quadratic Gaussian (LQG) controller using a Sequential Discrete Kalman Filter (SDKF) state estimator. While LQG controllers have been around since the 1980s, these controllers have currently not been widely adopted in industry since this algorithm involves a non-trivial matrix inversion. While many accelerated LQG & DKF architectures have been published, these architectures target specific platforms or applications; switching these architecture's application is a non-trivial and time consuming task. Thus, I designed an open-source hardware scalable, software configurable LQG controller, with the intent of others to use this design as an IP core, which will help ease the transition from abstract control theory to practical implementation. The design allows for a user to scale the accelerated LQG hardware architecture while software configurable registers allow the user to configure their controllers without re-synthesizing the hardware design, thus allowing for them to tune their controller on-the-fly. This controller was designed in Xilinx's Vivado 2018.2 design suite, targeting Xilinx ZYNQ series FPGAs, which contain an embedded dual-core ARM Cortex-A9 processor in addition to the traditional FPGA fabric. To compare the performance of this accelerated design, a software implementation of the algorithm was built and tested on three different processor platforms: an embedded ARM Cortex-A9 processor, an AMD FX-9800 series processor, and an Intel i7-4810MQ series processor. For lower dimensional matrices ($n = 4$), there were modest performance improvements, ranging from 0.79-14.5x improvement for the AMD & ARM processor, respectively. For larger dimensional matrices ($n = 128$), the HW/SW LQG achieved a 73x, 102x, and 1390x performance improvement over the Intel, AMD, and ARM processors, respectively. In addition to the software comparison, the analysis is concluded with a comparison of the proposed architecture's size and performance characteristics versus several of the most relevant and recent comparable architectures.

CHAPTER 1. INTRODUCTION

This chapter introduces the motivation for this work: to bridge the gap between multiple domains, specifically control theory and embedded systems. State-of-the-art control algorithms are computationally expensive and pure software implementations are no longer feasible for large scale systems. Thus, there is a drive to incorporate advanced control algorithms into hardware accelerated platforms, such as FPGAs or GPUs (Ding et al., 2019). This work seeks to advance this goal by designing a generalized & accelerated hardware scalable, software configurable LQG controller IP core for FPGA implementation.

1.1 Motivation

With the increases in technology, specifically in the computational power of embedded platforms, solutions involving Cyber Physical Systems (CPS) are becoming more common among many research domains (Lee et al., 2011). CPS can lead to innovative and effective solutions; however, this is shifting the way research is performed: collaborations of multi-disciplined researchers are becoming commonplace (Lee et al., 2011). This is due to needing intricate knowledge of both the physical system and the targeted computational platform. One such device often used for embedded systems is a Field Programmable Gate Array (FPGA). By programming the device to create custom hardware, FPGAs are frequently used for prototyping and implementing computationally intense algorithms, such as those used in state-of-the-art control theory.

The major disadvantage to FPGAs is that developing a custom design is time consuming and error prone. Thus, several companies have developed tools to help expand the use of FPGAs. High-level synthesis (HLS) seeks to abstract away the low-level Hardware Description Language (HDL) syntax and hardware programming style with a higher-level of abstraction. This is commonly done through either utilizing well-known, standardized software languages or graphical user interfaces

(GUIs). Xilinx, Matlab, and National Instruments (NI) have developed tools that utilize HLS to auto-generate HDL from functionally equivalent C/C++ code, their own functions, or block-diagram models. While these tools do expand the accessibility of FPGAs, as well as decrease their design-time cost, many of these tools return sub-optimal designs in comparison to manually generated HDL designs, since many of the intricacies of the hardware are hidden from the user or through a misunderstanding of the tool's hardware optimization features (Lahti et al., 2019).

Much like these high-level tools, the goal of this work is to help bridge the gap among research teams from multiple domains, specifically those in control theory and those in embedded systems. Presented in this thesis is a hardware scalable, software configurable Linear Quadratic Gaussian (LQG) controller, which is the combination of an optimal Linear Quadratic Regulator (LQR) control-law and a Sequential Discrete Kalman Filter (SDKF). This architecture is designed so that the hardware can scale to fit within any platform's constraints. Additionally, it is on-the-fly software configurable (which allows for tuning of the control algorithm without rebuilding the hardware design) and is designed with minimal assumptions so that it can accommodate a large variety of real-world systems.

In this way, this design is intended to act as an Intellectual Property (IP) core, which can be customized in hardware by an embedded systems engineer and configured in software by a controls engineer. By partitioning the workload, the design of a LQG controller can be separated into tasks suitable to each engineer's specialty. Additionally, like Matlab, NI's LabVIEW, and HLS, this IP core helps decrease the amount of time spent from design to implementation; however, the hardware generated by this design is transparent to the user and the controller's properties can be configured on-the-fly via software, rather than having to re-synthesize and re-implement a new hardware design.

Note that this architecture was designed with only one assumption about the targeted system: the sensor measurements must be independent from one another. This is due to implementing the SDKF algorithm rather than the standard Kalman Filter algorithm. The SDKF algorithm was chosen due to its ability easily scale, which is achieved by exchanging a generalized $n \times n$

matrix inversion for an iterative loop of scalar inversions. Thus, if this assumption is valid for the targeted system, then this architecture will perform the LQG algorithm successfully. However, since this design is intended to be generalized for most systems, system-specific optimizations are not performed.

By increasing the transparency of this design's hardware and algorithmic scheduling, this IP core helps to facilitate the transition of responsibilities among engineers of multiple disciplines, in addition to expanding the usability of FPGA accelerated controllers for application specific implementations.

1.2 Contributions

The contributions of this work are three-fold: (1) a novel, open-source architecture for a hardware scalable, software configurable LQG controller is presented (and available for download at Github/mcauwels), (2) the design of a modified multiply-accumulate tree which allows for element-wise addition by reusing its adders, and (3) the memory management algorithm that allows for matrices to be transposed across multiple BRAMs.

1.3 Thesis Outline

The structure of this thesis is as follows: Chapter 2 dives deeper into the motivation for the trend towards incorporating control algorithms into FPGAs, as well as an overview of similar algorithm's hardware architectures. It concludes with an review of similar high-level tools, their motivation, and highlights three commonly used, commercially available, tools. Chapter 3 presents a brief introduction to basic control theory concepts and the algorithms for both LQR control and Kalman filtering. Chapter 4 introduces the algorithm scheduling, the hardware architecture, and software interfaces for the hardware scalable, software configurable LQG controller using the SDKF. Chapter 5 performs an analysis of the proposed design against a pure software implementation as well as a comparison of this controller's resource & timing results against the most recent closely related works. Chapter 6 wraps up the thesis with a summary of the results and a discussion of future work.

CHAPTER 2. RELATED WORK

This chapter explores the prevailing popularity of Kalman filters and LQG controllers as well as the trend of moving controllers for complex, large scale systems to FPGAs. Other relevant publications in the topics of hardware accelerated architectures, specifically those for matrix inversion, Kalman filters, and LQG controllers, are presented. Additionally, the concept of hardware/software co-designed architectures is explored as well as a brief introduction to HLS, highlighting several popular commercially available HLS tools.

2.1 LQG Relevance and Industrial FPGA Controllers

This section extrapolates on the continued use of Kalman filters and LQG controllers. Additionally, the motivation behind implementing complex control algorithms in FPGAs, specifically in industrial applications, is presented.

2.1.1 Relevance of Kalman Filters and LQG Controllers

While Kalman filters and the Linear Quadratic Gaussian (LQG) control-law were developed in the 1980s, research into moving these algorithms into industry is still being conducted. Ding et al. (2019) conducted a survey of model-based control and filtering techniques used in industrial cyber-physical systems, which explores how Kalman-based algorithms are being utilized in industrial applications. Additionally, the challenges of these approaches, such as scalability and algorithmic complexity, are summarized. Similarly, Kozák (2012) presents a comprehensive list of control algorithms and their uses (or barrier to use) in industrial applications. In regards to LQG controllers, Kozák argues that a major barrier to their acceptance into industry is the lack of accurate linear state-space models, though he points out this may change as new state-identification methods are still being developed.

Since the LQG control algorithm has been around for awhile, it has been applied to many research applications. For example, Wanli et al. (2014) applied a LQG-like control algorithm to successfully balance a single-inverted pendulum cart. Eide (2011) was also able to balance an inverted-pendulum with an LQG controller; however, when comparing their simulated LQG algorithm against their LQR algorithm (with a proportional gain observer), they determined the LQR controller obtained less overshoot and had a lower settling time, though they admit that could have tuned their Kalman-Bucy filter's weighting matrices to achieve similar performance. As pointed out by Nestorović and Oveisi (2018), a disadvantage to the LQG algorithm in that it ignores any unmodeled dynamics.

Recently, LQG algorithms are growing in popularity and are being chosen as the "go-to" stabilizing control algorithm for linearized systems. Liu et al. (2018) implement a LQG-like technique to validate their gain-scheduling algorithm, which specifically targeted controlling the torque of variable stiffness actuators (VSA). Additionally, Rodrigues da Silva et al. (2017) demonstrate their hardware-in-the-loop testing technique, which was applied to an electrical power assisted steering (EPAS) system utilizing a stabilizing LQG algorithm. These two recent works (and many presented in Section 2.2.3) show how the LQG algorithm is becoming more commonplace when researchers need an algorithm for stabilizing a linearized system.

2.1.2 Embedding Control Algorithms in FPGAs

A notable trend has been emerging over the past 15 years: FPGA implementations of control algorithms are steadily growing. Monmasson et al. (2011) surveyed FPGA-based control methods in power electronics and power drive applications. They discuss the advantages (such as higher sample rates, deterministic timing, etc.) and disadvantages (long design time, unfriendly programming syntax, etc.) of having hardware controllers implemented via FPGAs. Additionally, Monmasson and Cirstea (2013) critique 13 FPGA-implemented controllers for industrial control applications. They introduce a brief overview to these various control methods and discuss their contributions to the power electronics and FPGA communities.

An example of an application that could benefit from FPGA implemented controllers is nanotechnologies. As discussed by Devasia et al. (2007), there are many difficulties in designing nanopositioning devices, such as high resolution, fast sample rates, and accurate position sensing & feedback control. However, Xie et al. (2019) was able to circumvent some of these difficulties by developing a high speed imaging architecture implemented in an FPGA. Because they used a hardware accelerated architecture, they were able to lower the bandwidth of the mechanical nanopositioner via an H_∞ and iterative learning control methodology. Besides being used for embedded controllers, FPGAs have also been used to develop state-of-the-art test beds. Šetka et al. (2017) utilized an ARM/FPGA System-on-Chip platform to develop a test bed for controlling a triple inverted pendulum.

2.2 Hardware Accelerated Architectures

Since the emergence of LQG controller in the 1980s, many works that have attempted to accelerate the Kalman state estimator within the LQG controller, specifically the matrix inversion operation. To better understand the types of algorithms and hardware architectures available, a literature survey of matrix inversion, Kalman Filters, and LQG algorithms and hardware architectures is presented.

2.2.1 Matrix Inversion

Since the bottle-neck for the Kalman filter algorithm is the $n \times n$ matrix inversion, this survey begins with algorithms for matrix inversion. A first approach is to follow the standard formula for matrix inversion - the adjoint divided by the determinant. Kumar et al. (2014) implemented this method in an FPGA for a third-order system, since there is a straight forward closed form solution for a 3×3 matrix inversion. Note that for 4×4 and larger matrices, this method quickly becomes intractable due to the exponential increase in computational power required.

Several researchers (Irturk et al., 2009; Santos et al., 2015) have turned to QR decomposition to develop an architecture for matrix inversion. This method involves decomposing a matrix (A) into two matrices: an orthogonal (Q) and a triangular (R) matrix. The crux of this method is twofold: the inverse of an orthogonal matrix Q^{-1} is equivalent to its transpose (Q^T) and the inverse of a triangular matrix R^{-1} is less computationally intense. Irturk et al. (2009) use the modified Gram-Schmidt algorithm to compute R^{-1} and develop a matrix computing core that allows them to perform matrix inversion. An advantage of their design is that their architecture can scale for systems of size $n = 4, 6, 8$. More recent work on QR Decomposition is presented by Santos et al. (2015) where they solve for Q and R by using a Givens rotations. Note that this method requires a computationally complex inverse square root, though the authors accelerate their design by performing a piecewise polynomial technique to approximate this calculation.

A more common approach to matrix inversion is to leverage the Faddeev algorithm, which utilizes the Schur compliment to iteratively compute the inverse of a matrix. This recursion is ideal for a pipelined systolic array, which is the approach taken by Yat Tin Lai et al. (2004). Note that this implementation also approximates the algorithm's division by using lookup tables to perform an approximated fixed-point division. This allows them to achieve better performance at a modest cost in computational accuracy.

A newer approach to matrix inversion is presented by Xu et al. (2018), where they constrain their design space to matrix inversion of a positive definite symmetric matrix. They then propose a new algorithm: the Simple Positive-definite symmetric Matrix Inversion (SPMI). This method takes advantage of the matrices structure so that matrix inversion is performed by an iterative multiply and add operations, as well as a few division operations.

2.2.2 Kalman Filters

Besides being an optimal state estimator, another highly sought after property of a Kalman filter is it's ability to filter out Gaussian noise prevalent in a system or its sensors. However, the Kalman filter's algorithm requires a matrix inversion, which is time consuming for an embedded CPUs to compute, even in lower-order systems. Thus, there has been a steady interest in accelerating Kalman filter algorithms through hardware implementations.

Application specific architectures seek to accelerate the Kalman filter algorithm for a specific design goal. A common design goal is to achieve sub-microsecond sample rates, as seen in Liao et al. (2019) and Phuong et al. (2010). These implementations achieve $2\mu s$ and $5\mu s$ sample rates by taking advantage of their system's low dimensionality. Another design goal is high throughput, which is common in image filtering. Johnson et al. (2017) present an example of a high throughput architecture which uses a modified Faddeev algorithm and systolic array to create a highly pipelined and efficient architecture for their 3rd-order system. Other research applications have developed accelerated hardware architectures with the goal of low input-output overhead and increased parallel computations. Nazir et al. (2015) give an example of this, where multi-channel brain activity is detected and filtered using a 4th-order Kalman filter. Fonseca et al. (2013) report a 2-3 order of magnitude performance increase when incorporating a Kalman filter for ballistic rocket tracking in hardware compared to a software approach. These examples show the desire to incorporate accelerated Kalman filter algorithms into cutting edge technology.

Several Kalman filter implementations stood out among the rest of the reviewed works due to the author's decision to use a different variation of a Kalman filter algorithm. The works presented by Akgün et al. (2018) and Mills et al. (2016) sought to utilize a nonlinear variation - an Extended Kalman filter algorithm (EKF). These works publish hardware accelerated architectures for this algorithm, either by using FPGA specific concepts (i.e., dynamic partial configuration) or piecewise affine modeling of nonlinear systems. Kettner and Paolone (2017) used a Sequential Discrete Kalman Filter (SDKF) in their hardware architecture to perform the state estimation of power distribution systems. This algorithm avoids matrix inversion by performing an iterative loop of scalar inversions, with the caveat that there is no covariance between the systems sensors (i.e., each sensor measures one value). Babu and Detroja (2019) point out that this limits the applicability of such a system, so they propose an Inverse Free Kalman Filter. While this work does not present a hardware accelerated approach, they provide two numerical examples and the results of their algorithm's simulation. One drawback to their approach is that their approach relies on the assumption that the covariance of the error is orders of magnitude less than the errors variance, thus limiting their algorithm's applicability as well.

In summary, it was reviewed that much of the application specific Kalman filter architectures that perform matrix inversion are low-dimensionality. For higher dimensionality, it appears that the trend is to avoid matrix inversion; however, these algorithms make simplifying assumptions that limit their uses to their specific systems.

2.2.3 LQG Controllers

Since their conception in the 1980s, LQG controllers have been utilized in a variety of applications; however, due to the matrix inversion in the Kalman filter and its computational demands, researching how to accelerate LQG controllers is still a relevant research area. Work towards implementing LQG controllers into FPGAs started as early as 1996, when Garbergs and Sohlberg (1996, 1998) developed several variations of a hardware LQG controller to balance an inverted pendulum system. While they implemented both a floating-point and fixed-point variation of their architecture, the entire LQG control loop was estimated to take 700 clock cycles for a time-invariant Kalman Filter and 5700 clock cycles for a time-variant one.

Since FPGA technologies have advanced drastically since the mid-90s, a more recent review of hardware accelerated LQG controllers is presented. Priewasser et al. (2014) present a comparison of a hardware PID and an LQG controller to control a novel small-signal model for the variable switching frequency of DC-DC converters. This model is relatively low dimensionality (3rd-order) and the authors implemented their LQG controller using a time-invariant Kalman filter on an Altera Cyclone-IV FPGA. Another LQG implementation is presented by Cupelli et al. (2015) where they implement a decentralized LQG controller to regulate the DC bus of a MVDC microgrid. The authors implement a 2nd-order LQG controller with an EKF in a Xilinx Virtex-5 FPGA using National Instruments' (NI) Real Time Target toolbox. Benkhoud et al. (2017) incorporates a hardware LQG controller to control their Quad Tilt Wing unmanned aerial system (UAS) and demonstrates their computer aided design (CAD) methodology for rapid control prototyping. Their model for their UAS is a 12th-order model, which is larger than any of the works presented; however, their focus is on their methodology, not on their controller implementation, so the details of this design are not reported. A fixed-point hardware accelerated LQG controller was also implemented in Deliparaschos et al. (2015, 2017) to explore systematic sensor selection. The authors incorporate a hardware/software design approach to their architecture and present an LQG controller which uses a Kalman-Bucy time-invariant filter. Similar to the other works presented, their system is lower-dimensionality (3rd-order), though their application requires a 100μs update rate. They achieve such a low update rate through the use of an FPGA, which was translated from their Matlab simulation to HDL primarily through Matlab's HDL Coder library.

2.3 HW/SW Codesigns

Individually, software and hardware have several advantages and disadvantages. Software is flexible, quickly encoded, and has a well-defined coding standard; however, it's non-determinism makes it difficult to model and validate. On the other hand, hardware is deterministic, specialized, and parallelizable; however, it has a long design time, unfriendly coding syntax, and relatively clunky design & simulation tools.

To combine the best of both methods, a hardware/software (HW/SW) codesign methodology has been growing among embedded hardware researcher groups. This methodology seeks to combine the flexibility of software with the high computational speeds of hardware. In fact, Balasch et al. (2018) present a course they have introduced at their university to teach this concept to undergraduate students.

Several applications of HW/SW codesigns have been published in recent works. Several previously mentioned works have followed this methodology (Deliparaschos et al., 2015; Mills et al., 2016), thought their main goal was more towards application specific designs rather than user reusability. Al-Saaty et al. (2017) have presented a HW/SW Codesigned self-tuning PID controller. This project was designed to reduce the offline learning steps associated with tuning a PID controller. Another application of this methodology is presented by Lee et al. (2018) where they present a precision time protocol (PTP) slave transparent clock (TC) architecture to exceed the safety requirements of IEC/IEEE 61850-9-3 precision time profile for power utility automation.

Zhang et al. (2015, 2017) presents two HW/SW codesigned architectures for user configurable controllers, one for LQR control and another for Model Predictive Control (MPC). These controllers were designed for a general case and to facilitate the transition between complex control algorithms and real-world implementations on embedded platforms. In this way, the authors have given the end user the ability to scale the hardware architecture to accommodate their own performance/resource trade-off, thus allowing their design to fit into any FPGA platform. Additionally, they design their system so that the control parameters are software driven, i.e., the end-user can configure & tune the control characteristics in software rather than hardware. This allows the designs to be more user-friendly, since compiling C code is less time consuming than resynthesizing and routing a new hardware design. A similar design for an Unscented Kalman Filter (UKF) is carried out by Soh and Wu (2017).

While this methodology combines the best parts of both software and hardware, it also combines their worst parts as well. As pointed out by Kumar et al. (2017), this methodology has not progressed from research into industry, mainly due to the complexity of the software/hardware interface. Thus, HW/SW codesigns are difficult to formally verify, which keep them from becoming prevalent in industry, especially in safety critical applications. Additionally, the initial hardware design is still time consuming, and adding additional time to validate the software interface is required.

2.4 High-Level Synthesis Tools

In this section, a brief introduction to the concept of high-level synthesis (HLS) is presented. Additionally, several commercially available tools - Xilinx's VivadoHLS, Matlab's HDL Coder, and National Instrument's LabVIEW FPGA - are explored.

2.4.1 High-Level Synthesis Overview

As pointed out by Nane et al. (2016), the major drawback of FPGAs is their long development phase, primarily due to the low-level of abstraction required to of correctly generating hardware through the use of HDL. To try to reduce this design time, and to expand the user-base of FPGAs, the development and use of High-Level Synthesis (HLS) has been steadily growing. While there are a variety of HLS tools available, they all operate on the same principle: have the developer use a well known high-level interface (software programming language, block diagram design, etc.) to specify the design's functionality and let the HLS program auto-generate functionally similar HDL code. In this manner, the use of FPGAs will no longer be restricted to hardware developers, since high-level programming languages are utilized by many different types of engineers. Lahti et al. (2019) point out that other benefits include faster exploration of the project's design space, reuse of design for varying platforms, and verification acceleration via the use of software verification tools.

Like any design choice, while there are many benefits of HLS, there are several drawbacks and criticisms still prevalent today. The biggest drawback to HLS is automatically generating correct and optimal HDL from software. As discussed by Lahti et al. (2019), when presented with this challenge, HLS developers tend towards two different approaches: developing their own language for HLS design (usually based on a well-known language) or utilizing an already well-developed language. Notice that developing a language specific to HLS negates the benefit of using a well understood language, which was intended to expand the usage of FPGAs. Additionally, using a well-defined language is difficult, since many constructs for generating optimized HDL will be necessary. As pointed out by Nane et al. (2016), another heavily criticized drawback to HLS is that typical optimizations offered via HLS (loop-unrolling and pipelining) are misunderstood by software engineers, since typical software optimizations (caching and data reorganization) are drastically different from hardware ones. This may lead many software programmers astray, leading them to a worse or sub-optimal design compared to their optimized software solution. Lastly, the biggest criticism of HLS is that it produces lower quality HDL compared to manually generated HDL designs. As pointed out by Lahti et al. (2019), the basic trend in the hardware community is that manually generated HDL has a higher quality than HLS; however, they do not undersell the benefits of HLS, specifically, the faster rate of development and an increase in the designer's productivity.

2.4.2 Xilinx's Vivado HLS

One of the most advanced HLS tools available is Xilinx's Vivado HLS. Since its development in 2008, Xilinx's goal has been to develop a design suite to increase hardware developer's performance by utilizing C, C++, or System C code to generate high quality HDL for implementation with their FPGAs (Feist, 2012). To support their HLS effort, Xilinx has developed many libraries, architectural optimization options, and verification techniques, which allow their HLS implementations quality of results to rival manually generated HDL designs (Xilinx, b). Beyond the benefits of HLS listed in Section 2.4.1, Vivado's HLS uses optimization directives to allow the user to specify which optimization technique to apply to a given segment of C code Xilinx (a). A benefit of this is that it allows users to experiment with different optimization directives to achieve the performance that best suites their needs.

As given in Xilinx (a), VivadoHLS's synthesis is carried out in three phases: scheduling, binding, and control logic extraction. The scheduling phase determines the timing of the synthesized design based on user directives, targeted device, and clock frequency. Binding evaluates the C code's operations and binds each operation to resources specific to the targeted platform. Control logic extraction develops a Finite State Machine (FSM) which matches the order of the C code's execution to obtain functionally equivalent code.

Many of the works presented in presented in Section 2.2.2 and Section 2.2.3 were developed using Xilinx's Vivado HLS design suite (Liao et al., 2019; Cupelli et al., 2015).

2.4.3 Mathlab's HDL Coder

Mathwork's Matlab is a powerful modeling and simulation tool used by many different disciplines for system develop and simulation. A key feature of Matlab is the Simulink toolbox: a drop-

and-drag user interface which allows designers to visually create and test their designs. Simulink specifically caters to the development of stabilizing controllers with their interactive PID tuning and their Control System toolbox (Mathworks, a). However, as pointed out by Sumam and Shiny (2017), transferring Matlab/Simulink designs to HDL may introduce errors into the HDL design, further increasing the HDL development cycle. Additionally, if a developer was to update their Matlab/Simulink design, they would have to manually update and verify their HDL design, further increasing their development time.

To combat these problems, Mathworks has developed a HDL Coder library, which automatically generates synthesizable HDL code (Mathworks, b). A major benefit of their HDL Coder is that it allows for tracability between simulation and design, which is required in many safety critical applications (i.e., aerospace, medical technologies). Additionally, integrating HDL generation with the Matlab simulation allows designers to begin verification early on in the design (Mathworks, c). Matlab's HDL Coder integrates many HLS synthesis optimizations, which allow the user to explore hardware quantization and several other optimizations. Another benefit of using HDL Coder is that it is portable across many FPGA manufacturers, allowing for one design to be programmed into different types of FPGAs.

While this commercially available product may allow for faster development and testing of designs, it is still under development. As mentioned by Deliparaschos et al. (2017), Matlab's HDL Coder is currently limited when it comes to multi-dimensional matrices, which forces users to manually build HDL alongside the auto-generated HDL, breaking the tracability advantage of HDL Coder.

2.4.4 LabView's FPGA Tool Suite

Similar to Matlab's HDL Coder, National Instrument's (NI's) LabVIEW FPGA Module allows FPGA programming on a higher level of abstraction. The LabVIEW FPGA Module provides a graphical interface and a unified development tool-chain to accelerate FPGA programming by abstracting away the low-level signal routing prevalent in typical HDL development (NationalInstruments, b). However, beyond being susceptible to many of the drawbacks prevalent within other HLS tools, many of their functionalities are supported with only NI equipment (NationalInstruments, a). Despite this, several previously mentioned works (Ibañez et al., 2017; Al-Saaty et al., 2017; Benkhoud et al., 2017) have successfully implemented their designs with NI's LabVIEW FPGA module and NI's myRIO development boards.

CHAPTER 3. LQG ALGORITHM

This chapter gives an introduction to the concept of state-space as well as the algorithms for the Linear Quadratic Regulator (LQR), the Discrete Kalman Filter (DKF), and the Sequential Discrete Kalman Filter (SDKF). The LQG controller presented in this work is a combination of the LQR optimal control-law and the SDKF least-squares state estimator.

3.1 State-Space Modeling

State-space is a mathematical way of expressing a the physical response of a system (also referred to as a plant) via a set of dynamical equations (Chen, 1999). For this embedded application, a linear discrete-time state-space model is used to represent the plant's dynamics, which are computed based on the system's current dynamics as well as any input to the system. The standard notation for this state-space is:

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k \\ y_k &= Cx_k + Du_k \end{aligned} \tag{3.1}$$

where,

- n represents the number of states of the system.
- m represents the number of inputs to the system.
- p represents the number of outputs from the system.
- x_k is the $n \times 1$ state vector that represents each state, at time step k.
- u_k is the $m \times 1$ input vector that represents the input(s) of the system, at time step k.
- y_k is the $p \times 1$ output vector that represents the output(s) of the system, at time step k.
- A is the $n \times n$ state matrix that represents the system's internal dynamics.

- B is the $n \times m$ input matrix that represents the effects of each input upon the system.
- C is the $p \times n$ output matrix that represents the effects of each of the system's states upon each of the system's outputs.
- D is the $p \times m$ feed-through matrix that represents the direct effect the input has on the output.

A benefit of using state-space is that it easily maps to a closed-loop control system, as seen in Fig. 3.1. Another is that state-space is well suited to model multiple-input-multiple-output (MIMO) systems, due to its matrix structure. As systems are becoming more and more complex, state-space models are increasingly used in research; however, larger systems are difficult and time consuming to model accurately. Thus, interest in automating and aiding the development of tractable models is still a highly researched subject (Kozák, 2012).

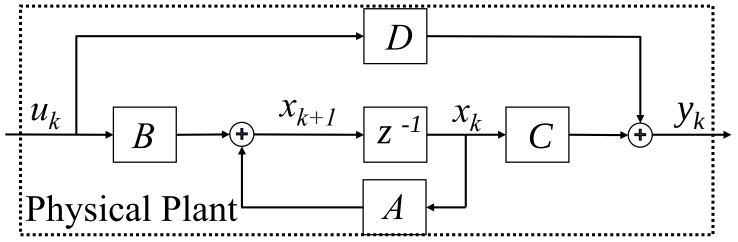


Figure 3.1 A pictorial representation of (3.1), with respect to a closed-loop system.

Note that an LQG controller can be developed for systems which are both ***controllable*** and ***observable***. A system is said to be controllable if any state can be influenced from the system's input and a system is said to be observable if any state can be recreated from the system's output (Chen, 1999; Phillips et al., 2015). While there exists multiple ways to check a system for controllability and observability, the simplest check is to check the rank of the Controllability ($\mathcal{C}$) and Observability ($\mathcal{O}$) matrices, as demonstrated by (3.2-3.3). Should the rank of $\mathcal{C}$ and of $\mathcal{O}$ are both greater than or equal to the number of states of the system n, then a stabilizing LQG controller can be designed for the system (Chen, 1999; Phillips et al., 2015).

$$rank\{\mathcal{C}\} = rank\left\{\begin{bmatrix} B & AB & A^2B & \cdots & A^{n-1}B \end{bmatrix}\right\} \geq n \tag{3.2}$$

$$rank\{\mathcal{O}\} = rank\left\{\begin{bmatrix} C & CA & CA^2 & \cdots & CA^{n-1} \end{bmatrix}^T\right\} \geq n \tag{3.3}$$

3.2 Linear Quadratic Regulator (LQR)

A Linear Quadratic Regulator (LQR) is an optimal state-feedback control-law; it is optimal in the sense that the controller's output is minimized across a cost function; in this case, the quadratic cost function seen in (3.4).

$$J(u) = \sum_{k=1}^{\infty} x_k^T Q x_k + x_k^T N u_k + u_k^T R u_k \tag{3.4}$$

Within the cost function are three weighting matrices: Q, N, and R. These correspond to the desired state cost, state-input cost, and input cost, respectively (Otaga, 1987). These matrices are system and controller performance specific, so there is no set way to determine these for any arbitrary system, though a few heuristics exist (Otaga, 1987). Once these matrices are tuned (usually via simulation) to obtain the desired controller performance, a closed form solution to this cost function can be found via the discrete-time algebraic Riccati equation (3.5).

$$P = A^T P A - (A^T P B + N)(R + B^T P B)^{-1} + Q \tag{3.5}$$

Once (3.5) is solved and P is obtained, the static gain matrix, K, for the controller can be found via (3.6).

$$K = (R + B^T P B)^{-1}(B^T P + N^T) \tag{3.6}$$

The state-feedback aspect of the controller is performed by (3.7), which produces the optimal output for the given weighting matrices Q, N, and R.

$$u_k = K x_k \tag{3.7}$$

Thus, the closed-loop system from (3.1) can be combined with (3.4) to obtain (3.7).

$$x_{k+1} = (A - BK) x_k \tag{3.8}$$

Note that a weakness of any state-feedback control-law is the assumption that all states of the system are available for the controller to use. Practically, this is not a valid assumption, since all states of a system would have to be measured using a physical sensor, which can be infeasible due to cost and/or physical limitations. However, it may be possible for a system to estimate unmeasured states from measured ones, i.e., the systems is observeable (Phillips et al., 2015). Thus, it is common to see a state-estimator combined with an LQR control-law when implemented on a physical system.

3.3 Kalman Filter Algorithms

As mentioned, a state-estimator is likely needed to use a state-feedback controller. While many state-estimators exist, a Kalman filter uses least-squares regression to obtain optimal state-estimates, even in the presence of input & system noise (Brown and Hwang, 2012). The updated system model that the Kalman filter is based on, as well as an explanation of its components, are elaborated upon in Section 3.3.1. While many versions of the Kalman filter exist, I will specifically be referring to the Discrete Kalman Filter (DKF) and the Sequential Discrete Kalman Filter (SDKF). Both of these Kalman filters perform a prediction and then an estimate of the system states. The prediction and estimation stages will be described for both the DKF and SDKF in Sections 3.3.2 and 3.3.3, respectfully.

3.3.1 Kalman Filter Model

For both the DKF and the SDKF, the plant's state-space model differs slightly from 3.1 to include noise, as seen in (3.9) and Fig. 3.2.

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + w_k \\ z_k &= H_k + v_k \end{aligned} \tag{3.9}$$

Note that there are some subtle differences between (3.1) and (3.9): process noise (w_k) and measurement noise (v_k) are added to the state-update and state-output equations, respectively. Additionally, this model assumes that there is no feed-forward (D) relationship between the input

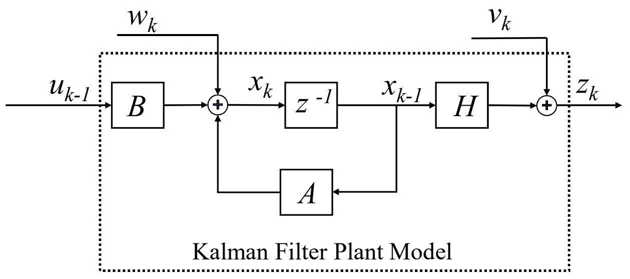


Figure 3.2 A pictorial representation of (3.9) with respect to a closed-loop system.

and the output. Should any system have a D matrix in its state-space model, a state-transformation should be performed such that the feed-forward component is absorbed into the state-update equations. This Kalman filter model also replaces the output matrix (C) with the sensor matrix (H). The subtle difference is that the H matrix is the relationship between the *system states* and the *sensor output* whereas the C matrix is the relationship between the *system states* and the *system output*. For most practical purposes, these two matrices are equivalent.

The noise vectors presented in (3.9) are both Gaussian white noise vectors, which are modeled as a zero-mean, normally distributed, uncorrelated spectral white noise (Brown and Hwang, 2012). The white noise vectors, and their corresponding covariance matrices, are defined in (3.10).

$$\begin{aligned} &w_k \sim N(0, Q_k) \quad v_k \sim N(0, R_k) \\ &Q_k = E[w_k w_k^T] \quad R_k = E[v_k v_k^T] \quad E[w_k v_k^T] = 0 \end{aligned} \tag{3.10}$$

where the notation $\sim N(\mu, \sigma^2)$ is read as "normally distributed process with μ mean and σ^2 variance" and the notation for the expected value of a random process is given as $E[\cdot]$. Additionally, the system covariance (Q_k) and measurement covariance (R_k) matrices are defined in (3.10). Note there is a distinction between these covariance matrices from (3.10) and the weighting matrices from (3.4).

The purpose of the Kalman filter is to perform a recursive least-squares regression across a random variable to minimize that variable's error. In both the DKF & SDKF, the random variable is the state-vector (x_k), whose error (e_k) they seek to minimize. Both algorithms do this by first performing a prediction (based on the system's dynamical model) and then an estimation (based on the system's previous state and measurement error). Therefore, it is necessary to distinguish between the predicted state-vector ($\hat{x}_k^-$) and the estimated state-vector ($\hat{x}_k^+$). With these predicted and estimated state-vectors, their corresponding error vectors and covariance matrices are defined by (3.11).

$$\begin{aligned} e_k^- &= x_k - \hat{x}_k^- & e_k^+ &= x_k - \hat{x}_k^+ \\ P_k^- &= E[e_k^-(e_k^-)^T] & P_k^+ &= E[e_k^+(e_k^+)^T] \end{aligned} \tag{3.11}$$

3.3.2 Prediction Stage

The DKF's and the SDKF's prediction stage is identical: the previous state estimates and the previous inputs are fed into the state-update equation to predict the next state. Additionally, the prediction's error covariance matrix (P_k^-) is updated based on the system's dynamics (A), the system's noise covariance matrix (Q_k), and the estimate's error covariance matrix from the previous time step (P_{k-1}^+). These equations are given in (3.12) and represented by Fig. 3.3.

$$\begin{aligned} \hat{x}_k^- &= A\hat{x}_{k-1}^+ + Bu_{k-1} \\ P_k^- &= AP_{k-1}^+A^T + Q_k \end{aligned} \tag{3.12}$$

3.3.3 Estimation Stage

The key difference between the DKF and the SDKF lies in the estimation stage. For the DKF, the estimation stage involves a $p \times p$ matrix inversion, as seen in (3.13).

$$\begin{aligned} K_k &= P_k^- H_k^T (H_k P_k^- H_k + R_k)^{-1} \\ \hat{x}_k^+ &= \hat{x}_k^- + K_k(z_k - H_k \hat{x}_k^-) \\ P_k^+ &= P_k^- - K_k H_k P_k^- \end{aligned} \tag{3.13}$$

where the $p \times p$ matrix inversion exists in the $(H_k P_k^- H_k + R_k)^{-1}$ step of the DKF algorithm.

Unlike the DKF, the SDKF iterates through its estimation stage p times. Therefore, additional notation for this iterative process is introduced in (3.14).

$$z_{k,i} = (z_{k,i}) \quad H_{k,i} = row_i(H_k) \quad R_{k,i} = diag(R_k) \tag{3.14}$$

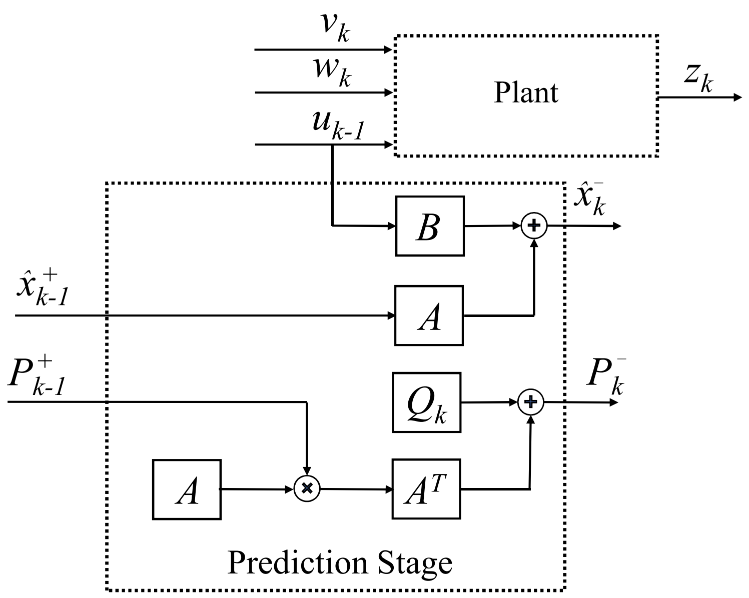


Figure 3.3 A pictorial representation of (3.12), showing how the prediction stage of the Kalman filter is performed using the previous state-estimate & input, modeled dynamics of the system, and the system's error covariance matrix Q_k.

where $z_{k,i}$ is the i^{th} element of the input sensor vector z_k, $H_{k,i}$ is the i^{th} row of the sensor matrix H_k, and $R_{k,i}$ is the i^{th} diagonal element of the sensor covariance matrix R_k. Additionally, the initial conditions of the SDKF's estimation stage (i.e., $i = 0$) are given in (3.15).

$$\hat{x}_{k,i=0}^+ = \hat{x}_k^- \quad P_{k,i=0}^+ = P_k^- \tag{3.15}$$

The estimation equations in 3.16 are repeated p times to obtain the estimated state-vector ($\hat{x}_k^+$).

$$\begin{aligned} K_{k,i} &= P_{k,i-1}^+ H_{k,i}^T (H_{k,i} P_{k,i-1}^+ H_{k,i}^T + R_{k,i})^{-1} \\ \hat{x}_{k,i}^+ &= \hat{x}_{k,i-1}^+ + K_{k,i}(z_{k,i} - H_{k,i}\hat{x}_{k,i-1}^-) \\ P_{k,i}^+ &= P_{k,i-1}^+ - K_{k,i} H_{k,i} P_{k,i-1}^+ \end{aligned} \tag{3.16}$$

Note that the DKF and SDKF algorithms are equivalent given the following assumption: the measurement covariance matrix (R_k) is diagonal, i.e., the sensor measurements are uncorrelated (Kettner and Paolone, 2017; Brown and Hwang, 2012). This assumption is reasonable if each sensor only gives information about one measured state; however, if one sensor contributes to the measurement of two or more states, then the SDKF algorithm cannot be used.

CHAPTER 4. ARCHITECTURE

This chapter presents how the HW/SW LQG controller schedules the LQG algorithm, a detailed description of the hardware architecture, and how the software interfaces with the LQG hardware architecture. Note that this architecture is designed for using 32-bit floating point (IEEE 754) values for all arithmetic operations.

4.1 Algorithm's Scheduling

The LQG algorithm implemented in this controller utilizes equations (3.7, 3.12, and 3.16) from Chapter 3. Note that these equations are sequential in nature: all of the computations of 3.12 must be performed before any of the computations in 3.16. While this inherent sequential nature of the LQG algorithm is not well suited for hardware, the independent matrix arithmetic operations are; these individual computations contain a high degree of parallelism. Thus, the hardware accelerates the algorithm's computations by completing as many similar types of matrix arithmetic operations (e.g., matrix, vector, or scalar addition/multiplication operations) as it can before switching to a new arithmetic operation. The current breakdown of the LQG algorithm's equations is elaborated upon in Sections 4.1.1-4.1.3.

4.1.1 SDKF's Prediction Stage

The first step in the LQG algorithm is the prediction stage of the SDKF. The two equations presented in (3.12) consist of two matrix-matrix multiplications, two matrix-vector multiplications, a matrix addition, and a vector addition. Thus, (3.12) was broken into two states: one which performs the matrix-matrix & matrix-vector multiplication and the other that performs the both the matrix and vector additions, as seen in Table 4.1.

Table 4.1 LQG's Prediction Equations Scheduling Details

Order	***A***	***B***	**Op.**	**Result**
1	$P^+_{k,i-1}$	A^T	$\times$	$P^+_{k,i-1}A^T$
2	A	x_k	$\times$	Ax_k
3	B	u_k	$\times$	Bu_k
4	A	$P^+_{k,i-1}A^T$	$\times$	$AP^+_{k,i-1}A^T$
5	$AP^+_{k,i-1}A^T$	Q_k	$+$	P^-_k
6	Ax_k	Bu_k	$+$	$\hat{x}^-_k$

Note that the two matrix-matrix multiplications have a dependency: $P^+_{k,i-1} \times A^T$ must be completed before $A \times P^+_{k,i-1}A^T$ can proceed. To avoid stalling any pipelined processes, $A \times x_k$ and $B \times u_k$ are performed between $P^+_{k,i-1} \times A^T$ and $A \times P^+_{k,i-1}A^T$.

4.1.2 SDKF's Estimation Stage

The second step in the LQG algorithm is the estimation stage of the SDKF. The three equations presented in (3.16) consist of two matrix-vector multiplications, one vector-vector multiplication, three matrix-scalar multiplications, one scalar addition, one scalar subtraction, one vector addition, one matrix subtraction, and one scalar inversion. Thus, (3.16) was broken into three states: the matrix-vector multiplications & the scalar arithmetic, the scalar-matrix multiplication, and the vector & matrix addition/subtractions, as seen in Table 4.2.

Note the sequential nature of the equations in (3.16). There are several places where it makes the most sense to perform one computation, such as $P^+_{k,i-1}H^T_{k,i}$ first, since the next element of the equation relies on that computation. There are some additional computations which are independent of one another: $H_{k,i}P^+_{k,i-1}$ is independent of $P^+_{k,i-1}H^T_{k,i}$ and both are also independent of $H_{k,i}\hat{x}^+_{k,i-1}$. However, all three of these computations require some form of matrix multiplication. Thus, they are grouped together so that any parallelization the hardware may receive by performing the same type of operation is achieved.

Table 4.2 LQG's Estimation Equations Scheduling Details

Order	*A*	*B*	Op.	Result
1.a	$P^+_{k,i-1}$	$H^T_{k,i}$	$\times$	$P^+_{k,i-1}H^T_{k,i}$
1.b	$H_{k,i}$	$P^+_{k,i-1}$	$\times$	$H_{k,i}P^+_{k,i-1}$
2	$H_{k,i}$	$P^+_{k,i-1}H^T_{k,i}$	$\times$	$H_{k,i}P^+_{k,i-1}H^T_{k,i}$
3.a	$H_{k,i}$	$\hat{x}^+_{k,i-1}$	$\times$	$H_{k,i}\hat{x}^+_{k,i-1}$
3.b	R_i	$H_{k,i}P^+_{k,i-1}H^T_{k,i}$	$+$	$H_{k,i}P^+_{k,i-1}H^T_{k,i} + R_i$
4.a	$H_{k,i}P^+_{k,i-1}H^T_{k,i} + R_i$	1	$\div$	$(H_{k,i}P^+_{k,i-1}H^T_{k,i} + R_i)^{-1}$
4.b	$z_{k,i}$	$H_{k,i}\hat{x}^+_{k,i-1}$	$-$	$z_{k,i} - H_{k,i}\hat{x}^+_{k,i-1}$
5	$(H_{k,i}P^+_{k,i-1}H^T_{k,i} + R_i)^{-1}$	$P^+_{k,i-1}H^T_{k,i}$	$\times$	$K_{k,i}$
6.a	$K_{k,i}$	$(z_{k,i} - H_{k,i}\hat{x}^+_{k,i-1})$	$\times$	$K_{k,i}(z_{k,i} - H_{k,i}\hat{x}^+_{k,i-1})$
6.b	$K_{k,i}$	$H_{k,i}P^+_{k,i-1}$	$\times$	$K_{k,i}H_{k,i}P^+_{k,i-1}$
7.a	$\hat{x}^+_{k,i-1}$	$K_{k,i}(z_{k,i} - H_{k,i}\hat{x}^+_{k,i-1})$	$+$	$\hat{x}^+_{k,i}$
7.b	$P^+_{k,i-1}$	$K_{k,i}H_{k,i}P^+_{k,i-1}$	$-$	$P^+_{k,i}$

4.1.3 LQR's Computation

The last step in the LQG algorithm is the LQR state-feedback control-law. Note that the equation in (3.7) was modified to include a reference signal (u_{ref}). This allows the controller to track a given state trajectory. Thus, this computation consists of two matrix-vector computations and one vector addition. Thus, this computation was broken into one state, which performs both of the matrix-vector computations first and then the vector addition, as seen in Table 4.3.

Table 4.3 LQG's LQR Equations Scheduling Details

Order	*A*	*B*	Op.	Result
6.a	K_{lqr}	$\hat{x}^+_k$	$\times$	$K_{lqr}\hat{x}^+_k$
6.b	K_{lqr}	u_{ref}	$\times$	$K_{lqr}u_{ref}$
6.c	$K_{lqr}\hat{x}^+_k$	$K_{lqr}u_{ref}$	$+$	u_k

Note that the two matrix-vector computations are independent of one another; however, both must be completed before the vector addition can be computed.

4.2 Hardware Components

This section details the four main components to the scalable hardware architecture: (1) the Multiply Accumulate Tree, (2) the Scalar-Adder & Inverter, (3) the Memory Management Architecture, and (4) the Finite State Machine.

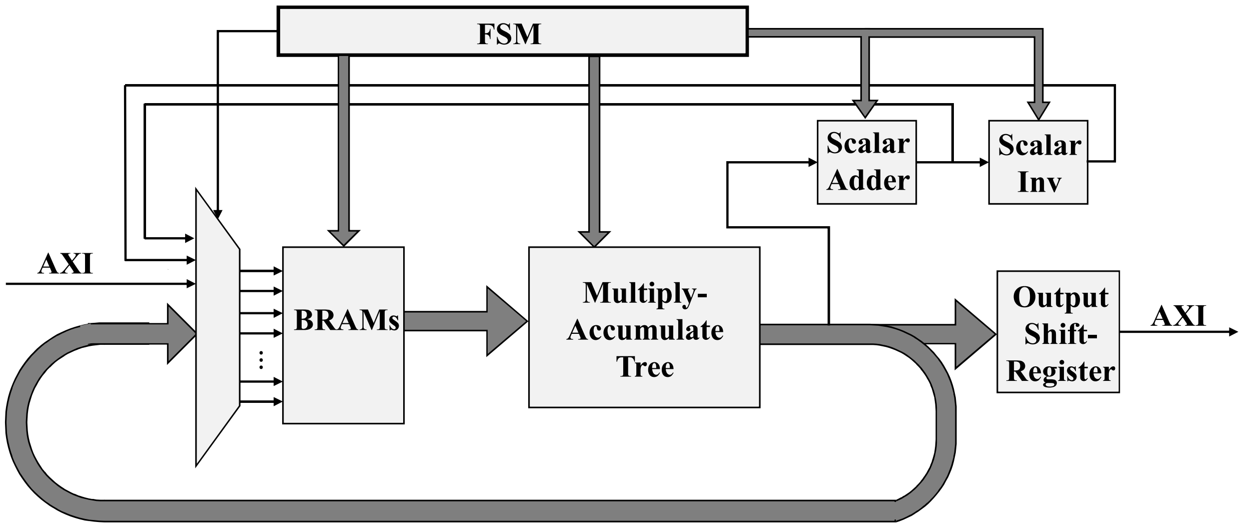


Figure 4.1 A top-level schematic of the LQG controller's hardware architecture, showing how the Finite State Machine (FSM) coordinates the configuration and scheduling of the BRAMs, Multiply-Accumulate Tree, and Scalar-Adder/-Inverter.

4.2.1 Multiply Accumulate Tree

The main computing element of this controller is a slightly modified multiply-accumulate tree, whose overall structure can be seen in Fig. 4.2. Notice that while a multiply-accumulate tree efficiently performs matrix-matrix & matrix-vector multiplications, the LQG algorithm presented in Section 3 requires scalar-matrix multiplication and element-wise addition/subtraction. Thus, multiplexers were added between each stage in the multiply-accumulate tree to facilitate these additional types of matrix arithmetic.

With the inclusion of these multiplexers, the multiply accumulate tree can be configured in real-time into three different modes of operation: (1) a matrix-vector multiplier, (2) a scalar-

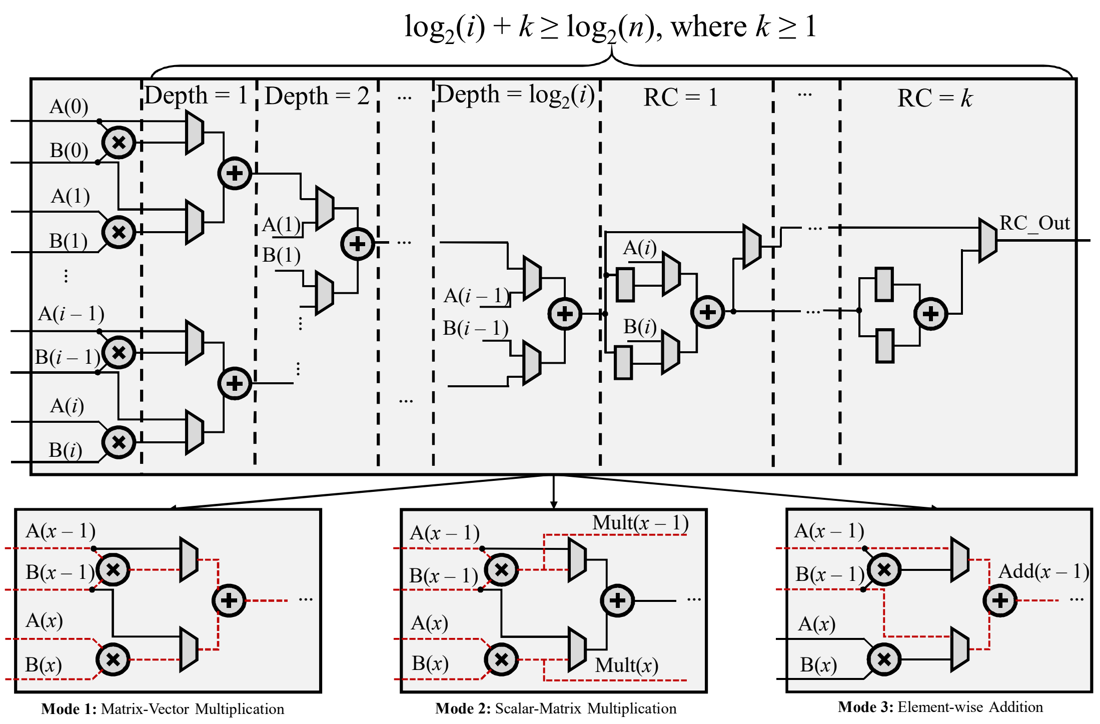


Figure 4.2 The schematic for the modified multiply-accumulate tree, showing its scalability, as defined by the parameter **Depth**. The structure's three modes of operation are broken down below the main diagram, with a red dashed line showing which path is operational in each mode. Additionally, the Reduction Circuit (RC) is shown, which will help accumulate Mode 1 operations when the number of matrix elements (n) is greater than the number of inputs (i).

matrix multiplier, and (3) an element-wise adder. The main benefit to this approach is that the multipliers and adders within this structure are reused for many different operations, thus reducing the amount of resources consumed in the design; however, one drawback is that the entire pipelined multiply accumulate structure must be drained when switching between modes. While it may be more time effective to stagger the pipeline so that different modes may finish while a secondary mode is starting, this creates potential memory write conflicts. For this reason, it was deemed more efficient to drain the pipeline than try to incorporate additional hardware to resolve any memory write conflicts.

Mode 1: Matrix-Vector Multiplier As the name mentions, this mode performs the standard multiply-accumulate operations. Thus, matrix-matrix & matrix-vector multiplications are computed at $m(n \times m)/Depth$ and $(n \times m)/Depth$ number of computations, respectively. Note that if the number of multipliers (i) is less than the number of rows (n), then additional hardware is needed to accumulate the partial sums. To do this, one or more reduction circuits (RC) are incorporated into the hardware, which are based on designs from Zhuo et al. (2005). Notice that, for a given i and n, the number of RCs needed (k) is given by (4.1).

$$log_2(i) + k \geq log_2(n) \tag{4.1}$$

Mode 2: Scalar-Matrix Multiplication In this mode, the output of the multipliers is fed directly back to memory, rather than the output of the last RC. Note that this allows for i number of multiplications to occur in parallel, rather than filling all but one of the multiply accumulate tree's inputs with zeros. Compared to this simplistic approach, there is a $(n \times m)/2^i$ speedup, where n is the number of rows, m is the number of columns, and i is the number of multipliers. A drawback to this approach is having i parallel write-backs to memory. How this is handled will be further elaborated upon in Section 4.2.3.

Mode 3: Element-wise Addition Due to the LQG algorithm needing to perform matrix and vector addition, an element-wise adder was needed. To allow for the adders within the multiply-accumulate tree to be reused, multiplexers were added between the inputs to the adders and the inputs to the multipliers. In this way, the adders could be directly fed their inputs rather than multiplying elements by 1. Similar to Mode 2, this allows for a $(n \times m)/2^i$ speedup, where n is the number of rows, m is the number of columns, and i is the number of inputs. Note that this speedup is achieved due to requiring at least one RC in the design to make the number of adders equal to the number of multipliers. While this may not be necessary, it simplifies the memory write-backs, which will be elaborated upon in Section 4.2.3.

4.2.2 Scalar-Adder & Inverter

While the multiply-accumulate tree presented in Section 4.2.1 can be configured to perform many of the computations in the LQG algorithm, two additional computing elements were added: a scalar-adder and a scalar-inverter, as seen in Fig. 4.3.

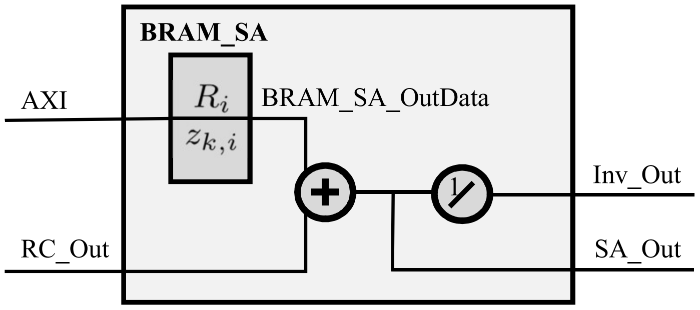


Figure 4.3 The schematic for the additional-scalar adder and scalar-inverter hardware. Notice that **BRAM_SA** is independent from the memory associated with the multiply-accumulate tree. This allows for sensor values to be written without having to design additional circuitry to prevent memory write-back conflicts.

The scalar adder circuit was added to increase the hardware's parallelism. Notice that for every iteration of the SDKF's estimation stage, a scalar addition and a scalar subtraction must be performed, as seen in 3.b and 4.b of Table 4.2. Rather than flush the entire multiply-accumulate tree to perform these two operations, another adder is placed in parallel to allow these operations to be performed while the multiply-accumulate tree is running.

The pinnacle operation of the SDKF is the scalar inversion, which is seen in 4.a of Table 4.2. Since the multiply-accumulate tree does not have any division circuitry, a scalar inversion circuit was needed. Since the scalar-addition is directly inverted, the output of the scalar-adder can be fed into the scalar-inverter. Similar to the scalar-adder, the inverter is in parallel to the multiply accumulate tree; however, due to the sequential nature of the algorithm and the computational complexity of floating-point inversion, this parallelism results in minuscule performance increases.

While the opportunity for parallelism is helpful, the main reason these two arithmetic circuits were placed aside from the main computing hardware is to remove the chance of a write conflict when writing sensor values to the scalar adder/inverter's block RAM (BRAM). By making this memory independent of the multiply-accumulate tree's memory, the control path for the hardware is simplified.

4.2.3 Memory Management Architecture

To coordinate which results get written back to each memory, multiplexers are incorporated between all of the outputs of the modified multiply-accumulate tree, the scalar-adder & inverter, and the software interface (**AXI**). Additionally, multiplexers were added between the memory's output and the multiply-accumulate tree's inputs to allow for some pre-processing of the outputs. All of these connections can be seen in Fig. 4.4.

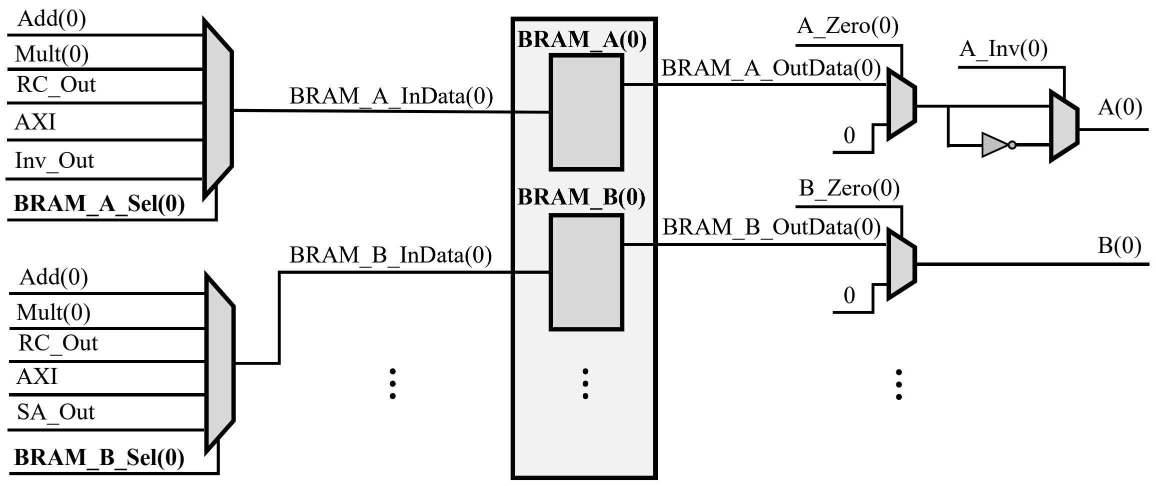


Figure 4.4 The schematic for the memory management architecture, showing how the multiplexers coordinate which input is fed into each BRAM and how the multiplexers modify the outputs of the BRAM before being fed into the multiply-accumulate tree.

Notice that the main memory element is BRAM. These BRAMs are configured as simple-dual port RAMs: a single read and single write of memory can occur within the same clock cycle, with the read coming before any write. Since the multiply-accumulate tree has two inputs per multiplier, at least one BRAM is associated with each input. For notation, these inputs were labeled A and B, as were the BRAMs associated with these inputs (**BRAM_A** and **BRAM_B**).

To better understand the memory structure, the variables stored in each BRAM are presented in Table 4.4. Notice that the amount of memory scales in accordance with the dimensions of each matrix (n, m, or p) and with the number of BRAMs available (2^{Depth}). Additionally, temporary variables ($T_{x,A/B}$) are introduced to allow matrices that remain constant, such as A, B, and K_{lqr}, to not have to be continuously re-written via software.

Table 4.4 Variables Stored Across the BRAMs

BRAM_A		**BRAM_B**		**BRAM_SA**	
Variable	**# of Addr**	**Variable**	**# of Addr**	**Variable**	**# of Addr**
A	$\frac{n^2}{2^{Depth}}$	A^T	$\frac{n^2}{2^{Depth}}$	R	n
P_A	$\frac{n^2}{2^{Depth}}$	P_B	$\frac{n^2}{2^{Depth}}$	z_k	p
$T_{0,A}$	$\frac{n^2}{2^{Depth}}$	$T_{0,B}$	$\frac{n^2}{2^{Depth}}$		
$T_{1,A}$	$\frac{n}{2^{Depth}}$	$T_{1,B}$	$\frac{n}{2^{Depth}}$		
$T_{2,A}$	1	$T_{2,B}$	1		
$T_{3,A}$	n	$T_{3,B}$	n		
H_A	$\frac{np}{2^{Depth}}$	H_B	$\frac{np}{2^{Depth}}$		
Q_A	$\frac{n^2}{2^{Depth}}$	Q_B	$\frac{n^2}{2^{Depth}}$		
B	$\frac{nm}{2^{Depth}}$	K_{lqr}	$\frac{nm}{2^{Depth}}$		
x_k	$\frac{n}{2^{Depth}}$	u_k	$\frac{n}{2^{Depth}}$		
u_{ref}	$\frac{n}{2^{Depth}}$				

The design of the memory architecture was challenging due to several issues inherent within the controller's specifications and algorithm. The first was deciding how the outputs of the multiply-accumulate tree would be disseminated to each BRAM. The second was how to store matrices across multiple BRAMs. The third was how to create matrices from the outer product of two vectors (i.e., a n vector times a $1 \times m$ vector to create a $n \times m$ matrix). Each of these issues, their solutions, and their justifications will be presented.

BRAM Inputs With the modified multiply-accumulate tree, the scalar-adder & scalar-inverter, and the software interface, there are $(2^{Depth})^2 + 4$ potential inputs to the BRAMs. It was determined that every BRAM needed the output of the multiply-accumulate tree's Mode 1 (**RC**) as well as the software interface (**AXI**). Additionally, the output of the scalar-inverter was tied directly to **BRAM_A** while the scalar-adder's output was tied directly to **BRAM_B**. This was due to the scalar outputs being tied directly to each BRAMs scalar temporary variable ($T_{2,A/B}$).

Thus the main issue was determining whether the direct outputs of each adder & multiplier were to be fed into each BRAM. This would require a resource intense cross-bar, especially as the depth of the multiply-accumulate tree grew. Rather than sacrifice the resources and timing for a crossbar, it was determined that each BRAM will be fed the outputs of its associated multiplier & adder. This way the chosen manner in which the matrices are stored is preserved (i.e., if the matrix is stored in row-major order, it stays in row-major order). A severe drawback to this method is that memory cannot be shared among BRAMs, thus making transposing a matrix challenging. Additionally, it increases the difficulty of performing the outer product of two vectors.

Storing Matrices There are two logical ways to store matrices in sequentially in memory: row-major and column-major order. However, since both matrix multiplication and addition are to be implemented, a method for transposing matrices between these two orderings becomes necessary. With the decision to give each BRAM only five inputs (as seen in Fig. 4.4), there are only two instances in which a single value of memory can be written to any BRAM: when initializing memory from software or from the output of the RC when the multiply-accumulate tree is in Mode 1. Thus, a memory management mechanism was developed to leverage the output of the RC to allow for **BRAM_A** or **BRAM_B** to switch between row-major or column-major ordering when performing matrix-vector computations. The algorithm for switching the ordering is presented in Table 4.5, where i is the current BRAM input (ranging from 0 to $2^{Depth} - 1$), j is the memory offset from the base address (ranging from 0 to $\frac{NumElem}{2^{Depth}}$, where $NumElem$ are the number of elements in the matrix or vector), k is factor by which the matrix elements are distributed among the BRAMs (ranging from 1 to $\frac{NumElem}{2^{Depth}-1}$), and l tracks the midway point where the offset switches between being even to odd.

Table 4.5 BRAM Management Mechanism

BRAM_WriteAddr(i) $\leftarrow$ BaseAddr + j

```
Procedure: Switch Storage Scheme
If $(i \geq (2^{Depth} - 1))$ && $(j \geq \frac{NumElem}{2^{Depth}})$ &&...
        $(k \geq \frac{NumElem}{2^{Depth} - 1})$
    $i = 0;\ j = l + 1;\ k = 1;\ l = l + 1;$
Else
    $j = k + l;\ k = k + 1$
    If $k \geq \frac{NumElem}{2^{Depth} - 1}$
        $i = i + 1;\ j = l;\ k = 1$
    End If
End If
```

```
Procedure: Maintain Storage Scheme
If $(i \geq (2^{Depth} - 1))$
    $i = 0$
    If $(j \geq \frac{NumElem}{2^{Depth}})$
        $j = 0$
    Else
        $j = j + 1$
    End If
Else
    $i = i + 1$
End If
```

Outer Product In the SKDF's estimation stage, the outer product of a $n \times 1$ & a $1 \times n$ vector (K_i & H_i) is performed to create a $n \times n$ matrix. This is challenging because each element of one vector must be multiplied by each element of the other vector. Without a crossbar, this is accomplished by forcing each BRAM to store both $n \times 1$ vectors in every BRAM, as seen in $T_{3,A/B}$ of Table 4.2.3. A drawback to this solution is that it utilizes $2(n-1)$ more addresses of memory than a vector distributed across all BRAMs would. Note that this is a linear increase in memory as n increases; however, this still results in a 7% to 30% increase in memory when compared to utilizing a crossbar. Based on Table 4.2.3, if one BRAM is 32MB (1024 addresses of 32-bits) then this increase doesn't cause for additional BRAMs until $n = 32$. Thus, this solution is deemed reasonable, since it has a minimal impact on the number of BRAMs used in the implementation of this design.

4.2.4 Finite State Machine

The finite state machine (FSM) coordinates the other hardware to perform the LQG algorithm. This is done by grouping the schedules from Table 4.1 - 4.3 together and replacing the output with their respective variable in memory. This overall schedule of the LQG controller with the memory variables from Table 4.4 can be seen in Table 4.6.

Table 4.6 LQG Equation Scheduling Details

State	A	B	Op.	Result
1.a	P_A	A^T	$\times$	$T_{0,A}$
1.b	A	$T_{1,B}$	$\times$	$T_{1,A}$
1.c	B	u_k	$\times$	$T_{1,B}$
1.d	A	$T_{0,B}$	$\times$	P_A, P_B
2.a	P_A	Q_B	$+$	P_A
2.b	Q_A	P_B	$+$	P_B
2.c	$T_{1,A}$	$T_{1,B}$	$+$	$\hat{x}_k, T_{1,B}$
3.a	P_A	$H_{k,i}^T$	$\times$	$T_{0,B}$
3.b	$H_{k,i}$	P_B	$\times$	$T_{3,B}$
3.c	$H_{k,i}$	$T_{0,B}$	$\times$	$S.A.1$
3.d	$H_{k,i}$	x_k	$\times$	$S.A.2$
3.e	R_i	$S.A.1$	$+$	$Inv.$
3.f	$z_{k,i}$	$S.A.2$	$-$	$T_{2,B}$
3.g	$Inv.$	-	$\div$	$T_{2,A}$
3.h	$T_{2,A}$	$T_{0,B}$	$\times$	$T_{3,A}$
4.a	$T_{3,A}$	$T_{2,B}$	$\times$	$T_{1,B}$
4.b	$T_{3,A}$	$T_{3,B}$	$\times$	$T_{0,B}$
4.c	$T_{3,A}$	$T_{3,B}$	$\times$	$T_{0,A}$
5.a	$\hat{x}_k$	$T_{1,B}$	$+$	$\hat{x}_k$
5.b	P_A	$T_{0,B}$	$-$	P_A
5.c	$T_{0,A}$	P_B	$-$	P_B
6.a	$\hat{x}_k$	K_{lqr}	$\times$	u_k
6.b	u_{ref}	K_{lqr}	$\times$	$T_{1,A}$
6.c	$T_{1,A}$	u_k	$+$	u_k

Besides controlling the schedule for the algorithm, the FSM also controls the mode of the multiply-accumulate tree (via each multiplier/adder's input multiplexer), the BRAM's input & output multiplexers, and when to write to the output shift register.

4.2.5 Output Shift Register

Since the algorithm ends with a $m \times 1$ vector addition, then the output vector u_k will be produced by the adders within the multiply-accumulate tree & RC (see Table 4.6, **State** 6.c). Since u_k will be distributed across all element-wise adders, each adder will produce $m/Depth$ elements of u_k. Thus, an output shift register is utilized to return the results of the LQG computation to the software. Each instance of this output shift register has $m/Depth$ 32-bit registers and its own finite-state machine (FSM) to control when to read, write, and shift the registers.

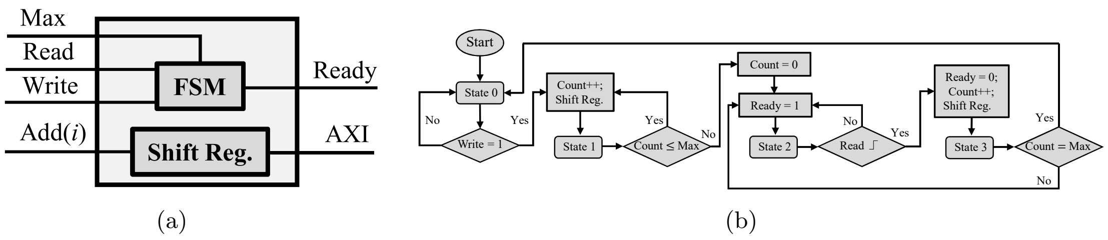


Figure 4.5 The schematic (a) and state-machine flow chart(b) of the output shift register hardware incorporated for each adder used to perform matrix addition (i.e., each adder within the multiply-accumulate tree and the first reduction circuit).

One important design feature is that the last register in the shift register, the `Max` signal, and the `Ready` & `Read` flags are software configurable registers. In this way, the software is able to view the output of the controller and maintain the its ability to configure the hardware without re-synthesizing the design.

As seen in Fig. 4.5, the state machine evolves across four states. In `State 0`, it merely waits for the LQG hardware to complete its computation and raise the `Write` flag. When `Write` is raised, the machine transitions to `State 1`, where it continuously writes to the first 32-bit register in the shift register, shifts the registers to the right, and increments `Counter`, which is used to keep track of how many values are currently within the shift register. Once `Counter` equals `Max`, the output shift register goes to `State 2`, where it resets `Counter` and sets the `Ready` flag, which alerts the software that the output is ready to be read. On the rising edge of the `Read` flag - a software controlled variable - the software indicates that it has read the value within the last 32-bit shift register and is ready for the next value. The FSM then goes to `State 3`, where it shifts in the next 32-bit output and either repeats the process, if there are more values to be read, or returns to its initial state, to await for the hardware to write its next value.

4.3 Software Interface

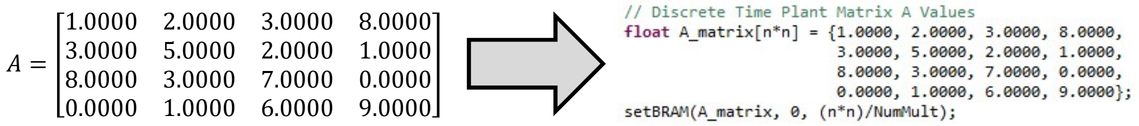
Several software interfaces are present in this design to try to allow the user to modify the controller's parameters without re-synthesizing the hardware design. Additionally, all input and output signals are routed through the CPU before being sent to the LQG computational hardware. This is done so that the user can log the input to and output from the controller for troubleshooting or for analysis the controller's performance.

4.3.1 Software Configurable Registers

Many of the parameters for the Finite State Machine in Section 4.2.4 are controlled by and accessible to the user via software configurable registers. In this design, these registers are automatically initialized via software, given values for the matrix dimensions (i.e., n, m, and p) as well as the *Depth* of the multiply-accumulate tree. Additionally, the minimum sample rate for the system (i.e., the amount of time between beginning and ending one iteration of the LQG algorithm) is controlled by a software configurable register as well. A complete list of all software configurable registers and their associated values can be found in the Appendix.

4.3.2 BRAM Initialization

Since the BRAMs hold the parameters of the system (e.g., the state-space matrices), these values must be encoded into the software via the user. The matrices are encoded as a floating point array, where the matrix is stored row-wise in the array, as seen in Fig. 4.6.



$$A = \begin{bmatrix} 1.0000 & 2.0000 & 3.0000 & 8.0000 \\ 3.0000 & 5.0000 & 2.0000 & 1.0000 \\ 8.0000 & 3.0000 & 7.0000 & 0.0000 \\ 0.0000 & 1.0000 & 6.0000 & 9.0000 \end{bmatrix}$$

```
// Discrete Time Plant Matrix A Values
float A_matrix[n*n] = {1.0000, 2.0000, 3.0000, 8.0000,
                       3.0000, 5.0000, 2.0000, 1.0000,
                       8.0000, 3.0000, 7.0000, 0.0000,
                       0.0000, 1.0000, 6.0000, 9.0000};
setBRAM(A_matrix, 0, (n*n)/NumMult);
```

Figure 4.6 An example of how the user would encode a parameter matrix in software so that it can be initialized into the FPGA's BRAMs via the `setBRAM` function.

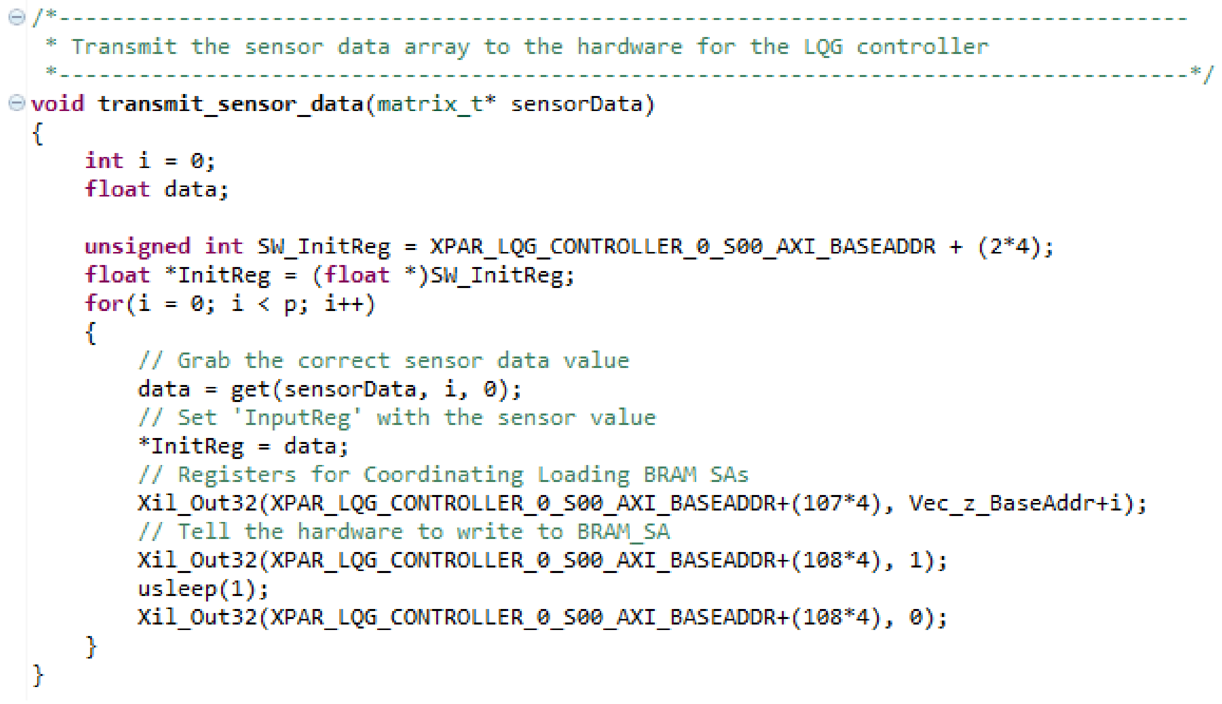
Once the matrices are encoded in their arrays, `setBRAM` can be called, which has three arguments: a float array, an integer `count`, and an integer `bounds`, e.g., the array of floating point matrix elements, a matrix identifier, and the number elements that should be written into the BRAMs, respectively. The `setBRAM` function then uses software registers to coordinate the writing of elements to memory.

4.3.3 Sensor Input Interface

To get the sensor values to the hardware, a function called `transmit_sensor_data` is called, which iterates through an array of sensor values and sends them to **BRAM_SA**. This is done by using the similar to how the other BRAMs are loaded: a software configurable register connected to the BRAM is loaded, another software configurable register that controls a the BRAM's input multiplexer is toggled to take the input from the software, and another software configurable register is written, which enables the BRAM to write. Figure 4.7 shows the details of how the sensors values are written to **BRAM_SA**.

4.3.4 Controller Output Interface

The software must read u_k once the LQG hardware writes u_k to the output shift register (as described in Section 4.2.5). This is done by checking a software configurable register which holds the

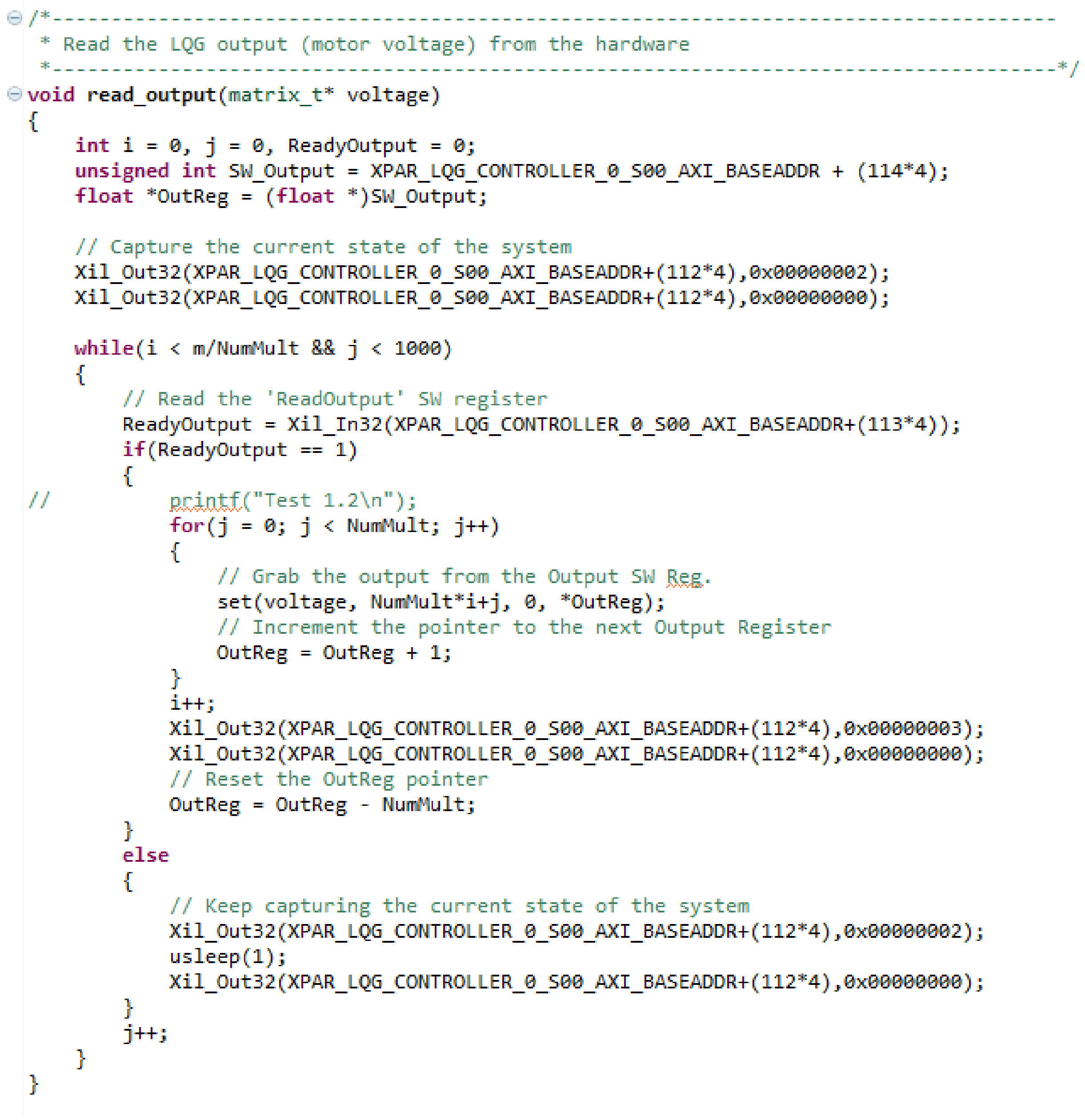


```
/*---------------------------------------------------------------------------------------
 * Transmit the sensor data array to the hardware for the LQG controller
 *---------------------------------------------------------------------------------------*/
void transmit_sensor_data(matrix_t* sensorData)
{
    int i = 0;
    float data;

    unsigned int SW_InitReg = XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR + (2*4);
    float *InitReg = (float *)SW_InitReg;
    for(i = 0; i < p; i++)
    {
        // Grab the correct sensor data value
        data = get(sensorData, i, 0);
        // Set 'InputReg' with the sensor value
        *InitReg = data;
        // Registers for Coordinating Loading BRAM SAs
        Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(107*4), Vec_z_BaseAddr+i);
        // Tell the hardware to write to BRAM_SA
        Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(108*4), 1);
        usleep(1);
        Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(108*4), 0);
    }
}
```

Figure 4.7 A screen-shot of `transmit_sensor_data`: the C function which facilitates the transition of sensor information from software to hardware's **BRAM_SA**. Given the `matrix_t` structure of `sensorData`, the software iterates through each element in `sensorData`, writing it to a software configurable register, setting its write address in **BRAM_SA**, and sending a write command to **BRAM_SA**.

bit-wise AND of each output shift register's `Ready` flag. Once this flag is raised, the software reads the last shift register, stores this value in software, and sets the `Read` flag, so that the hardware shifts the register to the next value. Figure 4.8 shows the details of how the output values are captured from the output shift registers.



```
/*-------------------------------------------------------------------------------------
 * Read the LQG output (motor voltage) from the hardware
 *-------------------------------------------------------------------------------------*/
void read_output(matrix_t* voltage)
{
    int i = 0, j = 0, ReadyOutput = 0;
    unsigned int SW_Output = XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR + (114*4);
    float *OutReg = (float *)SW_Output;

    // Capture the current state of the system
    Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(112*4),0x00000002);
    Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(112*4),0x00000000);

    while(i < m/NumMult && j < 1000)
    {
        // Read the 'ReadOutput' SW register
        ReadyOutput = Xil_In32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(113*4));
        if(ReadyOutput == 1)
        {
//          printf("Test 1.2\n");
            for(j = 0; j < NumMult; j++)
            {
                // Grab the output from the Output SW Reg.
                set(voltage, NumMult*i+j, 0, *OutReg);
                // Increment the pointer to the next Output Register
                OutReg = OutReg + 1;
            }
            i++;
            Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(112*4),0x00000003);
            Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(112*4),0x00000000);
            // Reset the OutReg pointer
            OutReg = OutReg - NumMult;
        }
        else
        {
            // Keep capturing the current state of the system
            Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(112*4),0x00000002);
            usleep(1);
            Xil_Out32(XPAR_LQG_CONTROLLER_0_S00_AXI_BASEADDR+(112*4),0x00000000);
        }
        j++;
    }
}
```

Figure 4.8 A screen-shot of **transmit_sensor_data**: the C function which reads the LQG controller's output u_k from each output shift registers (see Section 4.2.5).

CHAPTER 5. ANALYSIS

This chapter analyzes the proposed design by calculating the resource utilization and minimum sample rate, based on the number of states and depth of the multiply-accumulate tree. Several software platforms are then targeted with the same LQG algorithm and their performance is compared to varying scales of the HW/SW LQG's architecture. Beyond the comparison to software, a comparison of the proposed design against the performance and resource utilization of several relevant, recent works is presented.

5.1 Targeted Hardware Platforms

5.1.1 Resource Utilization

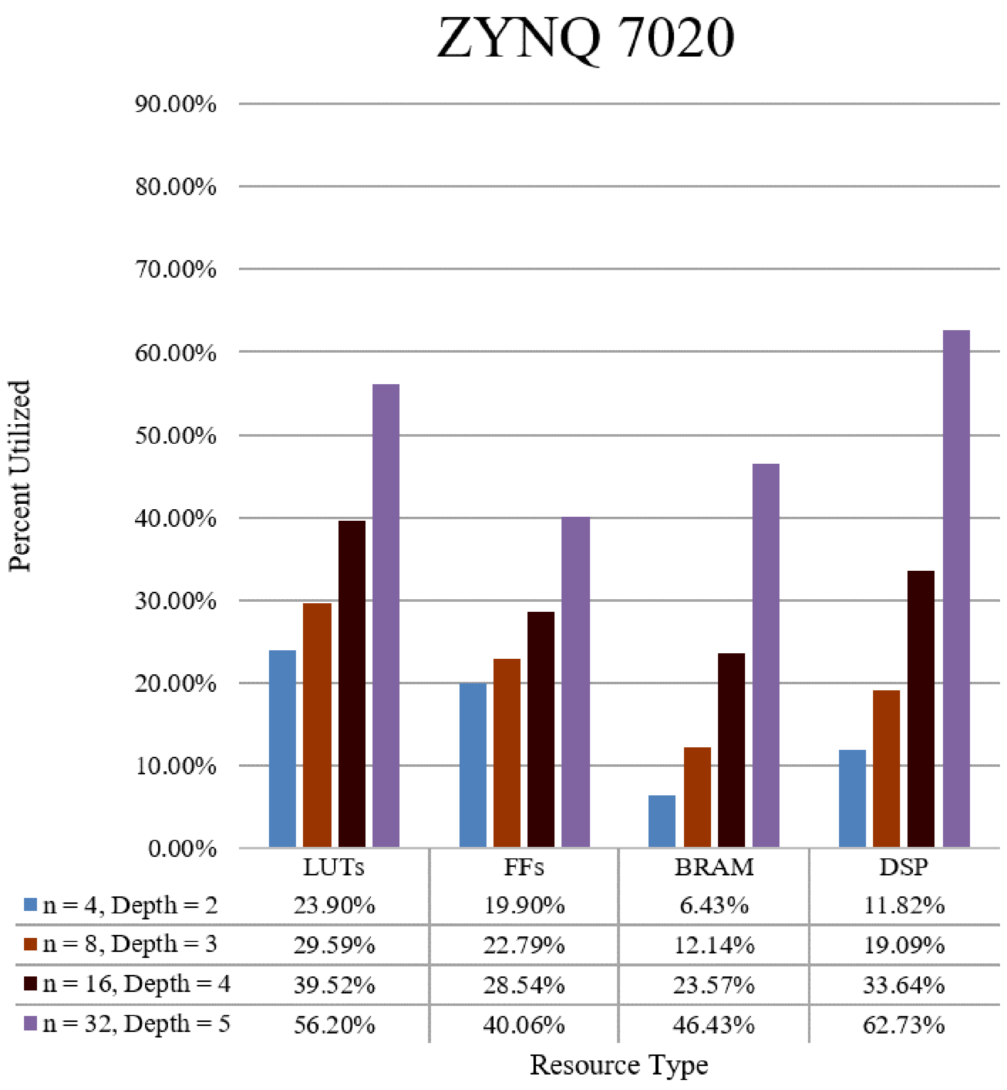
Since this architecture was developed using Xilinx's Vivado design suite, two FPGA platforms were targeted: a Digilent Zybo development board with a ZYNQ XC7Z020 System-on-Chip (SoC) and a Xilinx ZYNQ UltraScale+ ZCU106 evaluation platform with a XCZU7EV Multi-processor SoC (MPSoC). These were chosen due to their difference in price and size, which allowed exploration into resource, timing, and scale given a commonly used and a state-of-the-art research platforms. The fully pipelined hardware for varying matrix sizes were synthesized and implemented in Vivado. The resource utilization of ZYNQ-7020 and ZYNQ UltraScale+ MPSoC are given in Tables 5.1 and 5.2, respectively. Additionally, the percent of total resources used per FPGA is presented in Fig. 5.1 and Fig. 5.2, for the ZYNQ 7020 and UltraScale+ MPSoC, respectively.

Table 5.1 Hardware LQG Resource Utilization (Zynq - 7020)

System Size		LUTs	FFs	BRAMs	DSPs	Max. f_{clk}.
Depth	# of ×	53, 200	106, 400	140	220	MHz
2	4	12,725	21,174	9	26	132.98
3	8	15,747	24,251	17	42	135.12
4	16	21,015	30,347	33	74	131.77
5	32	29,935	42,588	65	138	116.50

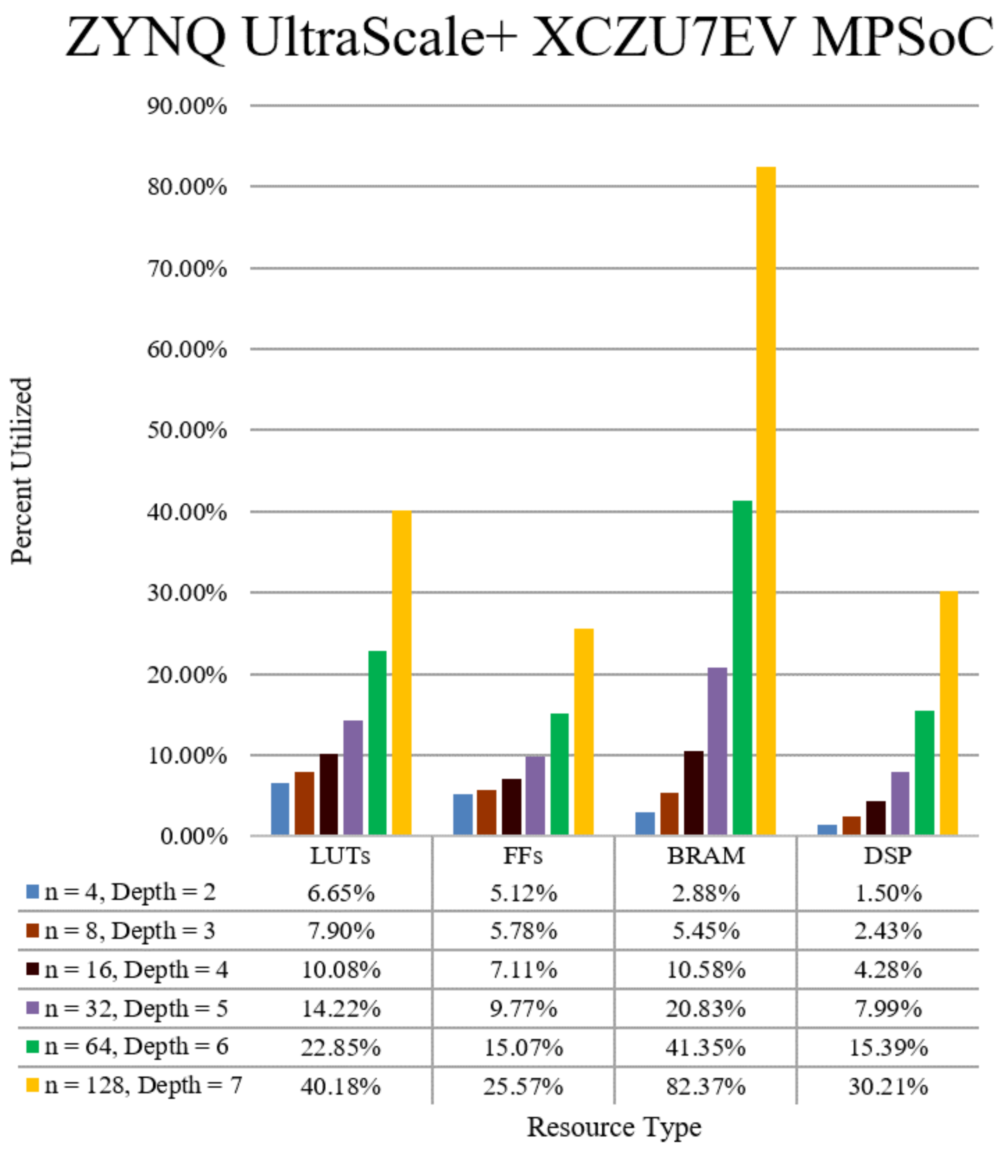
Table 5.2 Hardware LQG Resource Utilization (ZYNQ UltraScale+ ZCU106)

System Size		LUTs	FFs	BRAMs	DSPs	Max. f_{clk}.
Depth	# of ×	230, 400	460, 800	312	1728	MHz
2	4	15,314	23,588	9	26	150.38
3	8	18,213	23,588	17	42	131.16
4	16	23,222	32,779	33	74	140.29
5	32	32,774	45,017	65	138	136.04
6	64	52,653	69,446	129	266	125.82
7	128	92,579	117,846	257	522	112.87



	LUTs	FFs	BRAM	DSP
🟦 n = 4, Depth = 2	23.90%	19.90%	6.43%	11.82%
🟥 n = 8, Depth = 3	29.59%	22.79%	12.14%	19.09%
⬛ n = 16, Depth = 4	39.52%	28.54%	23.57%	33.64%
🟪 n = 32, Depth = 5	56.20%	40.06%	46.43%	62.73%

Figure 5.1 The resource utilization, as a percentage of total on-chip resources, for varying scales of the HW/SW LQG's hardware architecture for a Xilinx ZYNQ 7020. Note that the *Depth* of the multiply accumulate tree was chosen so that the system was fully pipelined, i.e., the largest amount of hardware for the given matrix dimension.



	LUTs	FFs	BRAM	DSP
🟦 n = 4, Depth = 2	6.65%	5.12%	2.88%	1.50%
🟫 n = 8, Depth = 3	7.90%	5.78%	5.45%	2.43%
⬛ n = 16, Depth = 4	10.08%	7.11%	10.58%	4.28%
🟪 n = 32, Depth = 5	14.22%	9.77%	20.83%	7.99%
🟩 n = 64, Depth = 6	22.85%	15.07%	41.35%	15.39%
🟨 n = 128, Depth = 7	40.18%	25.57%	82.37%	30.21%

Figure 5.2 The resource utilization, as a percentage of total on-chip resources, for varying scales of the HW/SW LQG's hardware architecture for a Xilinx ZYNQ Ultra-Scale+ XCZU7EV MPSoC. Note that the *Depth* of the multiply accumulate tree was chosen so that the system was fully pipelined, i.e., the largest amount of hardware for the given matrix dimension.

5.1.2 Control-loop Timing

The amount of time the hardware LQG controller takes to complete one iteration of its entire algorithm was calculated to determine its minimum sample rate. This computation is highly dependent on the latency produced by each of floating-point IP cores. Since this project was designed using Xilinx's Vivado 2018.2.2 design suite, Xilinx Floating-point v7.1 IP cores were used. While these IP cores can be customized with varying latencies (at the cost of clock speed & resource usage), this project targeted to have latencies of $Lat_{+} = 12$, $Lat_{\times} = 9$, and $Lat_{\div} = 30$ for the addition/subtraction, multiplication, and reciprocal arithmetic units, respectively. Additionally, if a reduction circuit is needed, then the pipeline depth increases by $\sum_{k=1}^{log_2 n - Depth} 2^k$, since each reduction circuit produces a valid summation every 2^k clock cycles, where k is the index of the reduction circuit. With these latencies, the latency for the multiply-accumulate tree's pipeline (Lat_{PD}) can be determined for any given $Depth$ & n via (5.1).

$$Lat_{PD} = Lat_{\times} + Lat_{+}(log_2 n)\left[+\sum_{k=1}^{log_2 n - Depth} 2^k\right] \tag{5.1}$$

Using (5.1) and Table 4.6, the number of clock cycles the LQG controller takes to complete each stage of the LQG algorithm can be computed, as seen in Table 5.3.

Table 5.3 Hardware Iteration Time - Formulas

State	**Number of Clock Cycles**
1	$\frac{2n^3+n^2+nm}{2^{Depth}} + P.D.$
2	$\frac{2n^2+n}{2^{Depth}} + Lat_{+}$
3	$\frac{2n^2+n}{2^{Depth}} + 2(P.D.) + max\{\frac{n^2}{2^{Depth}}, (Lat_{+} + Lat_{\div})\}$
4	$\frac{2n^2+n}{2^{Depth}} + Lat_{\times}$
5	$\frac{2n^2+n}{2^{Depth}} + Lat_{+}$
6	$\frac{2mn+m}{2^{Depth}} + P.D. + Lat_{+}$

Using the computations from Table 5.3, the timing computations for varying system sizes and pipeline depths was calculated and presented in Table 5.4.

Table 5.4 Hardware Iteration Time - 100MHz

Depth	Size ($n = m = p$)					
	4	8	16	32	64	128
1	10.25μs	37.1μs	201μs	1.41ms	10.8ms	85.0ms
2	8.81μs	26.4μs	114μs	737μs	5.47ms	42.7ms
3	-	21.0μs	74.7μs	399μs	2.81ms	21.5ms
4	-	-	55.1μs	233μs	1.48ms	11.0ms
5	-	-	-	158μs	814μs	5.68ms
6	-	-	-	-	493μs	3.03ms
7	-	-	-	-	-	1.71ms

Notice that time is not reported for when there are less matrix elements than multipliers. This is done to simplify the memory address architecture and BRAM interface.

5.2 Software Comparison

Since parallelism is heavily leveraged in this hardware design, a valid comparison of the accelerator's performance (e.g., timing) is against a comparable software implementation. While FPGAs execute more computations in parallel, they usually have a slower clock rate; therefore, while software is more sequential, it performs tasks faster. Thus, to perform a fair comparison, the same LQG algorithm is computed across multiple processing platforms with varying clock rates.

5.2.1 Targeted Software Platforms

Three different processors were available for comparison: a dual-core ARM Cortex-9, a quad-core AMD FX-9800, and a quad-core Intel i7-4810MQ. Note that since this ARM processor was embedded within a ZYNQ SoC fabric, its internal clock frequency could be adjusted between 50-667MHz. The AMD FX-9800 has a base frequency of 2.70GHz, but a max frequency of 3.60GHz. Similarly, the Intel i7-4810MQ has a base frequency of 2.80GHz, but a max frequency of 3.80GHz.

5.2.2 Timing Comparison

To make the comparison as fair as possible, a C software implementation computes the same formula as the hardware, i.e., the software also performs the SDKF algorithm, rather than the traditional DKF algorithm. Note that the ARM processor executed the algorithm without an operating system. The AMD and Intel processors were executed in Code Blocks application on top of Windows 10. To get accurate timing measurements, the software started a timer, ran 1,000 complete iterations of the LQG controller, stopped the timer, then returned the difference of the two timers. This procedure was carried out ten times for each matrix size. The mean and 99%-confidence interval of the software computations are presented in Tables 5.5 and 5.6.

Table 5.5 Software LQG w/ SDKF Iteration Time on ARM Cortex-A9 Processor

Size	Clock Rate		
$(n = m = p)$	100MHz	333MHz	650MHz
4	$782\mu s \pm 0.758\mu s$	$234.3\mu s \pm 0.432\mu s$	$120.1\mu s \pm 0.182\mu s$
8	$4.178ms \pm 1.01\mu s$	$1.253ms \pm 0.393\mu s$	$642.9\mu s \pm 0.262\mu s$
16	$28.41ms \pm 5.03\mu s$	$8.522ms \pm 1.77\mu s$	$4.370ms \pm 0.634\mu s$
32	$228.5ms \pm 25.8\mu s$	$68.55ms \pm 1.96\mu s$	$35.15ms \pm 0.978\mu s$
64	$1.845s \pm 13.4\mu s$	$554.5ms \pm 26.6\mu s$	$284.4ms \pm 14.2\mu s$
128	$16.18s \pm 15.1ms$	$4.893s \pm 501\mu s$	$2.552s \pm 167\mu s$

Table 5.6 Software LQG w/ SDKF Iteration Time on AMD & Intel Processors

Size	AMD FX-9800	Intel i7-4810MQ
$(n = m = p)$	2.7GHz	2.8GHz
4	$16.94\mu s \pm 2.66\mu s$	$8.587\mu s \pm 0.711\mu s$
8	$59.64\mu s \pm 5.05\mu s$	$45.17\mu s \pm 1.23\mu s$
16	$354.3\mu s \pm 8.37\mu s$	$275.0\mu s \pm 2.55\mu s$
32	$2.397ms \pm 20.4\mu s$	$2.074ms \pm 28.0\mu s$
64	$18.83ms \pm 97.0\mu s$	$15.69ms \pm 123\mu s$
128	$175.3ms \pm 1.37ms$	$125.5ms \pm 764\mu s$

As to be expected, when comparing the HW/SW LQG's timing performance (see Table 5.4), there is a timing improvement for nearly all matrix sizes across all processors. The only exception to this is for systems of size $n = 4$, where the Intel i7-4810MQ processor had a 0.02% faster speedup than the HW/SW LQG controller. When compared to the AMD FX-9800 and the ARM Cortex-A9 (at 650MHz), the HW/SW LQG controller achieves a .79 and 14.5 factor speedup, respectively. These modest speedups are likely due to the difference in clock rates and the lost parallelism in the HW/SW LQG controller, due to low utilization of the multiply-accumulate tree's pipeline.

When comparing the timing for $n = 128$, the HW/SW LQG controller achieves a 73x, 102x, 1390x speedup over the Intel i7, the AMD FX-9800, and the 650MHz ARM Cortex-A9, respectively. While the increased parallelism and heavily leveraged pipeline can justify the performance improvements achieved for the i7 and FX-9800 processors, it is not the only contributing factor when comparing against the ARM Coretx-A9 processor. The key to this performance difference is the ARM's cache. A 32-bit floating-point, 128×128 matrix is 64KB of memory; however, the ARM's L1 cache is only 32KB (ARM). Thus, the there are never any cache hits, resulting in constantly reading from either the L2 or off-chip memory, further escalating the time difference.

5.3 Related Work Comparison

To assess the methodology behind developing this HW/SW LQG controller, several recent works will be analyzed. The performance (e.g., minimum sample rate), resources used, and architecture scale will be compared to elaborate whether the proposed design would have been well-suited for each implementation.

5.3.1 Methodology Analysis

The first work that will be compared was presented by Deliparaschos et al. (2017), where they implement a 3rd-order LQG controller using Matlab's HDL coder while studying sensor selection. Table 5.7 shows the resource and timing analysis for their the fully-pipelined HW/SW LQG controller of size $n = 4$ versus their reported results.

While the design presented by Deliparaschos et al. (2015) uses 3.1-3.8x less LUTs, 8.79-9.79x less FFs, and 3x less BRAMs, they use 2.8x more DSPs and run 1.2-1.7x slower than the proposed

Table 5.7 Hardware Resource Utilization and Timing - Methodology Analysis

FPGA Series	n	Max. f_{clk}	LUTs	FFs	DSPs	BRAMs	Min. T_{samp}
ZYNQ-7020	4	112MHz	12,717	21,178	26	9	7.87μs
ZYNQ ZCU-106	4	150MHz	15,314	23,588	26	9	5.87μs
Virtex-6	3	25MHz	4,012	2,410	73	3	10μs

design. Should (Deliparaschos et al., 2017) have used an FPGA with an embedded CPU, they could have sped up their design phase, as well as their overall architecture, if they had used the proposed HW scalable, SW configurable LQG controller in their design. This validates our methodology in that this architecture can be used as an IP core to allow users to implement this LQG algorithm for their specific applications.

5.3.2 Application Analysis

The next comparison is against a hardware implementation of a DKF for image denoising using a systolic array, as presented by Johnson et al. (2017). Their approach targets a 3rd-order system with the design goals of low latency and high throughput. Their results, as well as the comperable results for the proposed system, are presented in Table 5.8.

Table 5.8 Hardware Resource Utilization and Timing - Application Analysis

FPGA Series	n	Max. f_{clk}	LUTs	FFs	DSPs	BRAMs	Min. T_{samp}
ZYNQ-7020	4	112MHz	12,717	21,178	26	9	7.87μs
ZYNQ ZCU-106	4	150MHz	15,314	23,588	26	9	5.87μs
Virtex-6	3	310MHz	4,438	2,821	91	81	122ns

While their architecture uses 2.9-3.5x less LUTs and 7.5-8.4x less FFs, their design is very DPS and BRAM intensive for such a low-order system, using 3.5x and 9x more DSPs and BRAMs, respectively. Though their design requires more DSPs and BRAMs, they achieve a much lower sample rate than the proposed HW/SW LQG controller could achieve.

This work was compared for multiple reasons, firstly to show that the proposed design may not meet the performance requirements for every system; in this case, the HW/SW LQG controller would be too slow. However, one benefit of the HW/SW codesign is its ability to reconfigure to allow for a variety of systems to use its architecture. Should Johnson et al. (2017) wish to scale their architecture for a 4th- or 5th-order system, their entire design would have to be retailored for this system, which would be time-consuming.

5.3.3 Scaling Analysis

While the hardware architecture can scale upwards of $n = 128$ states, many systems can be modeled using less states. As Hasan et al. (2019) report, the real-world model of the Brazilian power grid, a system of size $n = 7135$, can be reduced down to a model of $n = 100$ with less 10^{-3} error. Similar work is presented by Bonotto et al. (2016), where they reduce a Krylov subspace model from $n = 2773$ to $n = 30$ with less than 0.1% error.

The work presented by Kettner and Paolone (2017) use the SDKF algorithm to target an unreduced three-phase network, i.e., $n = 256$. Their resource and timing analysis, as well as the closest related performance metrics for the HW/SW LQG controller, are presented in Table 5.9.

Table 5.9 Hardware Resource Utilization and Timing - Scaling Analysis

FPGA Series	n	Max. f_{clk}	LUTs	FFs	DSPs	BRAMs	Min. T_{samp}
ZYNQ ZCU-106	128	112MHz	92,579	117,846	522	257	1.52ms
Kintex-7	256	-	43,166	49,088	357	262	35ms

Note that, due to the limited number of BRAMs available on the ZYNQ UltraScale+ ZCU106, the closest fully-pipelined HW/SW LQG architecture for comparison was $Depth = 7$ and $n = 128$. This is due to (Kettner and Paolone, 2017) allowing for certain simplifying assumptions, i.e., $A = I_{n\times n}$ and $B = 0_{m\times n}$. Beyond these two matrices they did not need to store, they were also only performing state-estimation, not performing the full LQG control algorithm, so they did not need to store the K_{lqr} matrix as well. However, if there was a ZYNQ FPGA with enough BRAMs available, it is estimated that a comparable hardware architecture, e.g., $n = 256$, $Depth = 6$, would require 897 BRAMs and have a minimum sample rate of 22.2ms at 100MHz.

5.3.4 HW/SW Codesign Analysis

The final related work comparison will be against a HW/SW Codesigned LQR controller, as presented by Zhang et al. (2015). Since the LQR algorithm is incorporated within the LQG, it can be reasonably assumed that the LQG algorithm will likely utilize more resources and time, since it performs more computations. As seen in Table 5.10, this assumption is reasonably accurate.

Table 5.10 Hardware Resource Utilization and Timing - HW/SW Codesign Analysis

FPGA Series	n	Max. f_{clk}	LUTs	FFs	DSPs	BRAMs	Min. T_{samp}
ZYNQ-7020	32	112MHz	29,896	42,620	138	65	141μs
ZYNQ ZCU-106	32	125MHz	32,774	45,017	138	65	126μs
ZYNQ-7020	32	122MHz	42,138	48,143	128	66	1.57μs

Note that since Zhang et al. (2015) are also using a multiply-accumulate structure for their matrix computations, the number of DSPs and BRAMs are nearly equivalent, with the exception of 8 more DSPs for the floating-point inversion, 2 more DSPs for the scalar-adder, and 1 more BRAM for the *BRAM_SA*. While the LQG architecture uses more DSPs and BRAMs, it uses less LUTs and FFs, likely due to the LQG architectures simplistic HW/SW interface.

The biggest difference between the two designs is the sample rate. Note that the LQG controller is nearly 100x slower than the LQR. This makes sense, due to the added computations from the SDKF causing the multiply-accumulate tree's pipeline to fill and drain. Despite this timing difference, this comparison again validates two key premises to this methodology: (1) while this LQG controller is generalized for any system, it may not suite all systems and (2) this HW/SW LQG controller could replace the presented HW/SW LQR controller in any application where the increase in sample rate would not impact the system.

CHAPTER 6. RESULTS AND DISCUSSION

The final chapter summarizes the performance, i.e., timing and resource utilization, of the proposed HW scalable, SW configurable LQG controller. Additionally, a discussion of the proposed methodology will be presented. Lastly, future work will be highlighted prior to concluding this work.

6.1 Performance Summary

6.1.1 Resource Summary

To achieve greater performance, the multiply-accumulate tree can scale to achieve further parallelism from the matrix computations. The amount of resources needed for a given architecture scale are presented in Fig. 5.1 & Fig. 5.2. Two Xilinx FPGAs were selected for place & route calculations of the controller: a modest ZYNQ 7020 and the state-of-the-art ZYNQ UltraScale+ XCZU7EV MPSoC. For the ZYNQ 7020, the scale of the architecture was limited by the number of DSPs; for the UltraScale+ MPSoC, it was the BRAMs. The DSPs and BRAMs limit the scale of the design. The DSPs are utilized for floating point multipliers & adders: for each increase in *Depth*, the number of DSPs doubles. Similarly, the number of BRAMs also doubles for each increase in *Depth*; however, this rate of increase changes after *Depth* = 7, due to needing more than one 32KB BRAM for each instance of *BRAM_A* and *BRAM_B*.

6.1.2 Timing Summary

The minimum sample rate for the HW/SW LQG controller was calculated for varying system sizes and architecture depths. For a comparison, the same LQG algorithm was tested on several software processors: an ARM Cortex-A9, an AMD FX-9800, and an Intel i7-4810MQ. Comparing the results of the HW/SW architecture's timing (Table 5.4) against the software's timing (Table 5.5 & 5.6), it can be seen that there are modest performance increases for lower-order systems, but with major increases for larger systems, particularly when compared with the ARM processor. These were a result of the leveraged parallelism as well as the low-read latency of the HW/SW LQG architecture. For the ARM Cortex-A9 processor, the size of its L1 cache made a substantial difference, since the matrices were so large they resulted in constant cache misses.

6.2 Methodology Discussion

This work sought to develop an open-source hardware scalable, software configurable LQG controller for closing the gap between LQG theory and its implementation. Due to its scalable hardware accelerated architecture and on-the-fly software configurable interface, the proposed HW/SW LQG controller is an ideal candidate for researchers to incorporate into their application specific designs. As demonstrated in Section 5.3, there are applications where this HW/SW LQG controller could be leveraged as an IP core to decrease development time and architecture flexibility. That being said, this HW/SW LQG architecture may not be well suited to all applications, since many system-specific simplifying assumptions were not made.

6.3 Future Work

While this architecture has been designed and implemented, it has not been thoroughly tested, due to the scaling architectures large design space. Currently, the only architecture that has been tested on a physical system is $Depth = 1$ and $n = 4$. While most of the work has been done for allowing the architecture to scale, it still needs to be tested before it should be applied to any system.

Currently, this design is also limited to being used on Xilinx devices, since it heavily leveraged Xilinx's Floating Point Operator (v7.1) and Block Memory Generator (v8.4) IP core libraries. Future work could be to transition away from these vendor libraries towards open-sourced libraries, to allow for all types of FPGAs to use this design.

Lastly, a goal of mine has been to incorporate this controller onto a quad-rotor Unmanned Aerial System (UAS) as an advanced case study. ISU's MicroCART senior design project would be an ideal candidate for implementing this controller, primarily due to its on-board FPGA.

6.4 Conclusion

This work sought to develop an open-source hardware scalable, software configurable LQG controller. This controller is meant to be used as an IP-core; this project is intended to mimic HLS designs in that it allows the user to incorporate this hardware accelerated LQG computational architecture into their design with minimal effort. This thesis outlines the details of the LQG algorithm: the LQR control-law and the SDKF state-estimator. A low-level overview of this controller's hardware architecture and software interface are given, as are the resource and timing analysis. Comparing this architecture's timing against a pure software implementation, modest performance improvements are achieved for low-order systems, with performance improvements increasing as the system size increases. When comparing the performance of related works versus the proposed design, it can be argued that the overall goal of creating an IP-core is validated, since many of the related designs could substitute the proposed controller and achieve similar results.

Bibliography

Akgün, G., u. H. Khan, H., Elshimy, M. A., and Göhringer, D. (2018). Dynamic tunable and reconfigurable hardware controller with ekf-based state reconstruction through fpga-in the loop. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8.

Al-Saaty, N. N., Algreer, M., and Armstrong, M. (2017). Hardware/software co-design techniques for compass search self-tuning pid controller in dc drive applications. In *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, pages 490–495.

ARM. Cortex-a9 technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf. Last accessed on Nov. 2nd, 2019.

Babu, K. S. and Detroja, K. (2019). Inverse free kalman filter using approximate inverse of diagonally dominant matrices. *IEEE Control Systems Letters*, 3(1):120–125.

Balasch, J., Beckers, A., Božilov, D., Roy, S. S., Turan, F., and Verbauwhede, I. (2018). Teaching hw/sw codesign with a zynq arm/fpga soc. In *2018 12th European Workshop on Microelectronics Education (EWME)*, pages 63–66.

Benkhoud, K., Bouallègue, S., and Ayadi, M. (2017). Rapid control prototyping of a quad-tilt-wing unmanned aerial vehicle. In *2017 International Conference on Control, Automation and Diagnosis (ICCAD)*, pages 423–428.

Bonotto, M., Bettini, P., and Cenedese, A. (2016). Model order reduction of large-scale state-space models in fusion machines via krylov methods. In *2016 IEEE Conference on Electromagnetic Field Computation (CEFC)*, pages 1–1.

Brown, R. G. and Hwang, P. Y. C. (2012). *Introduction to Random Signals and Applied Kalman Filtering with MATLAB Exercises*. John Wiley & Sons, Inc., 4th edition. ISBN-13 978-0-470-60969-9.

Chen, C. (1999). Oxford University Press, Inc., 3rd edition. ISBN-13 978-0-19-511777-6.

Cupelli, M., de Paz Carro, M., and Monti, A. (2015). Hardware in the loop implementation of a disturbance based control in mvdc grids. In *2015 IEEE Power Energy Society General Meeting*, pages 1–5.

Deliparaschos, K. M., Michail, K., Tzafestas, S. G., and Zolotas, A. C. (2015). A model-based embedded control hardware/software co-design approach for optimized sensor selection of industrial systems. In *2015 23rd Mediterranean Conference on Control and Automation (MED)*, pages 889–894.

Deliparaschos, K. M., Michail, K., and Zolotas, A. (2017). On the issue of lqg embedded control realization in a maglev system. In *2017 25th Mediterranean Conference on Control and Automation (MED)*, pages 1379–1384.

Devasia, S., Eleftheriou, E., and Moheimani, S. O. R. (2007). A survey of control issues in nanopositioning. *IEEE Transactions on Control Systems Technology*, 15(5):802–823.

Ding, D., Han, Q., Wang, Z., and Ge, X. (2019). A survey on model-based distributed control and filtering for industrial cyber-physical systems. *IEEE Transactions on Industrial Informatics*, 15(5):2483–2499.

Eide, R. (2011). Lqg control design for balancing an inverted pendulum mobile robot. *Intelligent Control and Automation*, 02:160–166.

Šetka, V., Čečil, R., and Schlegel, M. (2017). Triple inverted pendulum system implementation using a new arm/fpga control platform. In *2017 18th International Carpathian Control Conference (ICCC)*, pages 321–326.

Feist, T. (Jun. 22, 2012). Vivado design suite. https://www.xilinx.com/support/documentation/white_papers/wp416-Vivado-Design-Suite.pdf. Last accessed on Oct 23, 2019.

Fonseca, J. V., Oliveira, R. C. L., Abreu, J. A. P., Ferreira, E., and Machado, M. (2013). Kalman filter embedded in fpga to improve tracking performance in ballistic rockets. In *2013 UKSim 15th International Conference on Computer Modelling and Simulation*, pages 606–610.

Garbergs, B. and Sohlberg, B. (1996). Specialised hardware for state space control of a dynamic process. In *Proceedings of Digital Processing Applications (TENCON '96)*, volume 2, pages 895–899 vol.2.

Garbergs, B. and Sohlberg, B. (1998). Implementation of a state space controller in a fpga. In *MELECON '98. 9th Mediterranean Electrotechnical Conference. Proceedings (Cat. No.98CH36056)*, volume 1, pages 566–569 vol.1.

Hasan, S., Fony, A. M., and Uddin, M. M. (2019). Reduced model based feedback stabilization of large-scale sparse power system model. In *2019 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, pages 1–6.

Ibañez, C., Ocampo-Martinez, C., and Gonzalez, B. (2017). Embedded optimization-based controllers for industrial processes. In *2017 IEEE 3rd Colombian Conference on Automatic Control (CCAC)*, pages 1–6.

Irturk, A., Mirzaei, S., and Kastner, R. (2009). An efficient fpga implementation of scalable matrix inversion core using qr decomposition.

Johnson, B., Thomas, N., and Rani, J. S. (2017). An fpga based high throughput discrete kalman filter architecture for real-time image denoising. In *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, pages 55–60.

Kettner, A. M. and Paolone, M. (2017). Sequential discrete kalman filter for real-time state estimation in power distribution systems: Theory and implementation. *IEEE Transactions on Instrumentation and Measurement*, 66(9):2358–2370.

Kozák, . (2012). Advanced control engineering methods in modern technological applications. In *Proceedings of the 13th International Carpathian Control Conference (ICCC)*, pages 392–397.

Kumar, G. A., Subbareddy, T. V., Reddy, B. M., Raju, N., and Elamaran, V. (2014). An approach to design a matrix inversion hardware module using fpga. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 87–90.

Kumar, R., Cano, J., Brankovicy, A., Pavlouz, D., Stavrouz, K., Gibertx, E., Martínez, A., and González, A. (2017). Hw/sw co-designed processors: Challenges, design choices and a simulation infrastructure for evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 185–194.

Lahti, S., Sjövall, P., Vanne, J., and Hämäläinen, T. D. (2019). Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911.

Lee, E. A., A., S., and Seshia, editors (2011). *Introduction to Embedded Systems - A Cyber Physical Systems Approach*. Lulu, 2nd edition. ISBN-13 978-1312427402.

Lee, S., Kang, J., Choi, S. S., and Lim, M. T. (2018). Design of ptp tc/slave over seamless redundancy network for power utility automation. *IEEE Transactions on Instrumentation and Measurement*, 67(7):1617–1625.

Liao, J., Jost, M., Schaffner, M., Magno, M., Korb, M., Benini, L., Tebbenjohanns, F., Reimann, R., Jain, V., Gross, M., Militaru, A., Frimmer, M., and Novotny, L. (2019). Fpga implementation of a kalman-based motion estimator for levitated nanoparticles. *IEEE Transactions on Instrumentation and Measurement*, 68(7):2374–2386.

Liu, L., Leonhardt, S., and Misgeld, B. J. E. (2018). Experimental validation of a torque-controlled variable stiffness actuator tuned by gain scheduling. *IEEE/ASME Transactions on Mechatronics*, 23(5):2109–2120.

Mathworks. Control system toolbox. https://www.mathworks.com/products/control.html. Last accessed on Oct 23, 2019.

Mathworks. Hdl coder. https://www.mathworks.com/products/hdl-coder.html. Last accessed on Oct 23, 2019.

Mathworks. Simulink for hdl code generation and verification. https://www.mathworks.com/solutions/hdl-code-generation-verification.html. Last accessed on Oct 23, 2019.

Mills, A., Jones, P. H., and Zambreno, J. (2016). Parameterizable fpga-based kalman filter coprocessor using piecewise affine modeling. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 139–147.

Monmasson, E. and Cirstea, M. (2013). Guest editorial special section on industrial control applications of fpgas. *IEEE Transactions on Industrial Informatics*, 9(3):1250–1252.

Monmasson, E., Idkhajine, L., and Naouar, M. W. (2011). Fpga-based controllers. *IEEE Industrial Electronics Magazine*, 5(1):14–26.

Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y. T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., and Bertels, K. (2016). A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604.

NationalInstruments. Compare the labview 2019 fpga module and the labview nxg fpga module. https://www.ni.com/en-us/shop/electronic-test-instrumentation/add-ons-for-electronic-test-and-instrumentation/what-is-labview-fpga-module/compare-labview-fpga-module.html. Last accessed on Oct 23, 2019.

NationalInstruments. What is the labview fpga module. https://www.ni.com/en-us/shop/electronic-test-instrumentation/add-ons-for-electronic-test-and-instrumentation/what-is-labview-fpga-module.html. Last accessed on Oct 23, 2019.

Nazir, M. S., Aqil, M., Mustafa, A., Khan, A. H., and Shams, F. (2015). Real-time brain activation detection by fpga implemented kalman filter. In *2015 15th International Conference on Control, Automation and Systems (ICCAS)*, pages 432–435.

Nestorović, T. and Oveisi, A. (2018). Advanced disturbance rejection control of smart flexible structures. In *2018 7th International Conference on Systems and Control (ICSC)*, pages 224–229.

Otaga, K. (1987). *Discrete-Time Control Systems.* Prentice-Hall, Inc. ISBN 0-13-216102-8.

Phillips, C. L., Nagle, T., and Chakrabortty, A. (2015). *Digital Control System Analysis & Design.* Pearson Education Limited, 4th edition. ISBN-13 978-1-292-06122-1.

Phuong, T. T., Mitsantisuk, C., Ohishi, K., and Sazawa, M. (2010). Fpga-based wideband force sensing with kalman-filter-based disturbance observer. In *IECON 2010 - 36th Annual Conference on IEEE Industrial Electronics Society*, pages 1269–1274.

Priewasser, R., Agostinelli, M., Unterrieder, C., Marsili, S., and Huemer, M. (2014). Modeling, control, and implementation of dc–dc converters for variable frequency operation. *IEEE Transactions on Power Electronics*, 29(1):287–301.

Rodrigues da Silva, R., Teixeira, E. L. S., Murilo, A., and Dias Santos, M. M. (2017). A hardware-in-the loop platform for designing and testing of electric power assisted steering. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 5113–5118.

Santos, L. C., Atoche, A. C., Castilloy, J. V., Gandaraz, O. L., Alvarez, R. C., and Aguilar, J. O. (2015). An improved hardware design for matrix inverse based on systolic array qr decomposition and piecewise polynomial approximation. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6.

Soh, J. and Wu, X. (2017). An fpga-based unscented kalman filter for system-on-chip applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 64(4):447–451.

Sumam, M. J. and Shiny, G. (2017). A rapid development technique for prototype fpga controllers. In *2017 International Conference on Inventive Systems and Control (ICISC)*, pages 1–5.

Wanli, Z., Guoxin, L., and Lirong, W. (2014). Research on the control method of inverted pendulum based on kalman filter. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 520–523.

Xie, H., Wen, Y., Shen, X., Zhang, H., and Sun, L. (2019). High-speed afm imaging of nanopositioning stages using h_∞ and iterative learning control. *IEEE Transactions on Industrial Electronics*, pages 1–1.

Xilinx. *Vivado Design Suite User Guide.* Last accessed on Oct 23, 2019.

Xilinx. Vivado high-level synthesis. [https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. Last accessed on Oct 23, 2019.

Xu, Y., Li, D., Xi, Y., Lan, J., and Jiang, T. (2018). An improved predictive controller on the fpga by hardware matrix inversion. *IEEE Transactions on Industrial Electronics*, 65(9):7395–7405.

Yat Tin Lai, Bigdeli, A., and Biglari-Abhari, M. (2004). An optimised systolic array-based matrix inversion for rapid prototyping of kalman filters in fpga's. In *2004 12th European Signal Processing Conference*, pages 2035–2038.

Zhang, P., Mills, A., Zambreno, J., and Jones, P. H. (2015). A software configurable and parallelized coprocessor architecture for lqr control. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8.

Zhang, P., Zambreno, J., and Jones, P. H. (2017). An embedded scalable linear model predictive hardware-based controller using admm. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 176–183.

Zhuo, L., Morris, G. R., and Prasanna, V. K. (2005). Designing scalable fpga-based reduction circuits using pipelined floating-point cores. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8 pp.–.

APPENDIX. SOFTWARE CONFIGURABLE REGISTER COMPUTATIONS

This appendix presents the values of the software configurable registers that are automatically calculated from the user specified #`define` statements for the variables of `n`, `m`, `p`, and `Depth`. Note that the user will also need to update several hardware specific variables (i.e., `AddLat`, `MultLat`, etc.) which the user will have specified in hardware prior to synthesizing the design.

Hardware Specific Variables

```
#define AddLat 12 // Latency of the Floating−Point Adder
#define MultLat 9 // Latency of the Floating−Point Multiplier
#define InvLat 30 // Latency of the Floating−Point Inverter
#define ReadLat 1 // Latency of the BRAM when performing a read

unsigned int PipelineDepth = MultLat + AddLat∗(Depth + MaxCount) + (n−MaxCount)∗ReadLat;
unsigned int MaxGlobalCount = 0x000F4240; // 10ms

unsigned int NumMult = 1 << Depth;
unsigned int NumAdd  = (1<<Depth)−1;

unsigned int MaxReadC = ((n − 1)/NumMult);
unsigned int MaxReadD = (((n∗n) − 1)/NumMult);

unsigned int n_minus_1 = (n−1);
unsigned int p_minus_1 = (p−1);
unsigned int Add_nby1  = (n/NumMult);
unsigned int nbyn      = (((n∗n)−1)/NumMult);
unsigned int nby1      = ((n−1)/NumMult);
unsigned int mbyn      = (((n∗m)−1)/NumMult);
unsigned int nby1      = ((m−1)/NumMult);
unsigned int nbyn_e1   = (((n∗n)−1)/(1<<(Depth+1)));
unsigned int nby1_e1   = ((n−1)/(1<<(Depth+1)));
unsigned int mby1_e1   = ((m−1)/(1<<(Depth+1)));
unsigned int RedNum    = ((n/NumMult)−((n−1)/NumMult));
```

BRAM Base Addresses

```
// Base Addresses in BRAM_A
#define Mat_A_BaseAddr      0
#define Mat_PA_BaseAddr         ((n∗n)/NumMult)
```

```
#define Temp0A_BaseAddr     2*((n*n)/NumMult)
#define Temp1A_BaseAddr     3*((n*n)/NumMult)
#define Temp2A_BaseAddr     3*((n*n)/NumMult) +   (n/NumMult)
#define Temp3A_BaseAddr     3*((n*n)/NumMult) +   (n/NumMult) + 1
#define Mat_HA_BaseAddr     3*((n*n)/NumMult) +   (n/NumMult) + 1 + n
#define Mat_QA_BaseAddr     3*((n*n)/NumMult) +   (n/NumMult) + 1 + n + ((n*p)/NumMult)
#define Mat_B_BaseAddr      4*((n*n)/NumMult) +   (n/NumMult) + 1 + n + ((n*p)/NumMult)
#define Vec_x_BaseAddr      4*((n*n)/NumMult) +   (n/NumMult) + 1 + n + ((n*m)/NumMult) + ((n*p)/NumMult)
#define Vec_uRef_BaseAddr   4*((n*n)/NumMult) + 2*(n/NumMult) + 1 + n + ((n*m)/NumMult) + ((n*p)/NumMult)

// Base Addresses in BRAM_B
#define Mat_AT_BaseAddr      0
#define Mat_PB_BaseAddr       ((n*n)/NumMult)
#define Temp0B_BaseAddr     2*((n*n)/NumMult)
#define Temp1B_BaseAddr     3*((n*n)/NumMult)
#define Temp2B_BaseAddr     3*((n*n)/NumMult) +  (n/NumMult)
#define Temp3B_BaseAddr     3*((n*n)/NumMult) +  (n/NumMult) + 1
#define Mat_HB_BaseAddr     3*((n*n)/NumMult) +  (n/NumMult) + 1 + n
#define Mat_QB_BaseAddr     3*((n*n)/NumMult) +  (n/NumMult) + 1 + n + ((n*p)/NumMult)
#define Mat_K_lqr_BaseAddr  4*((n*n)/NumMult) +  (n/NumMult) + 1 + n + ((n*p)/NumMult)
#define Vec_u_BaseAddr      4*((n*n)/NumMult) +  (n/NumMult) + 1 + n + ((n*m)/NumMult) + ((n*p)/NumMult)
```

Stage 1 Variables

```
// Stage 1 of LQG Algorithm
unsigned int Stage1_1_StartWrite = PipelineDepth;
unsigned int Stage1_1_End        = PipelineDepth + (((n-1)/NumMult)+1)*n;

unsigned int Stage1_2_StartRead  = (n*n*n)/NumMult;
unsigned int Stage1_2_StartWrite = PipelineDepth + ((n*n*n)/NumMult);
unsigned int Stage1_2_StopWrite  = PipelineDepth + ((n*n*n)/NumMult) + ((n*n)/NumMult);

unsigned int Stage1_3_StartRead  = ((n*n*n)/NumMult) + ((n*n)/NumMult);
unsigned int Stage1_3_StopRead   = ((n*n*n)/NumMult) + ((n*n)/NumMult) + ((n*m)/NumMult);
unsigned int Stage1_3_StopWrite  = PipelineDepth + ((n*n*n)/NumMult) + ((n*n)/NumMult) + ((n*m)/NumMult);

unsigned int Stage1_5_StopRead   = 2*(((n*n*n)/NumMult)) + ((n*n)/NumMult) + ((n*m)/NumMult) + ReadLat;
unsigned int Stage1_5_StartWrite = PipelineDepth + ((n*n*n)/NumMult) + ((n*n)/NumMult) + ((n*m)/NumMult);
unsigned int Stage1_5_StopWrite  = PipelineDepth + (2*((n*n*n)/NumMult)) + ((n*n)/NumMult) +
((n*m)/NumMult);

unsigned int Stage1_4_StopRead   = PipelineDepth + (((n-1)/NumMult)+1)*n + ((n*n*n)/NumMult) + ReadLat;
unsigned int Stage1_4_StartWrite = 2*PipelineDepth + (((n-1)/NumMult)+1)*n;
unsigned int Stage1_4_StopWrite  = 2*PipelineDepth + (((n-1)/NumMult)+1)*n + ((n*n*n)/NumMult);
```

Stage 2 Variables

```
// Stage 2 of LQG
unsigned int Stage2_1_StartWrite = AddLat + ReadLat;

unsigned int Stage2_2_StartRead  =   ((n*n)/NumMult);
unsigned int Stage2_2_StartWrite =   ((n*n)/NumMult) + AddLat + ReadLat;

unsigned int Stage2_3_StartRead  = 2*((n*n)/NumMult);
unsigned int Stage2_3_StopRead   = 2*((n*n)/NumMult) + (n/NumMult) + ReadLat;
unsigned int Stage2_3_StartWrite = 2*((n*n)/NumMult) + AddLat + ReadLat;
unsigned int Stage2_3_StopWrite  = 2*((n*n)/NumMult) + (n/NumMult) + AddLat + ReadLat;
```

Stage 3 Variables

```
// Stage 3 of LQG
unsigned int Stage3_1_StartWrite = PipelineDepth;

unsigned int Stage3_2_StartRead  =   ((n*n)/NumMult);
unsigned int Stage3_2_StopRead   = 2*((n*n)/NumMult) + ReadLat;
unsigned int Stage3_2_StartWrite =   ((n*n)/NumMult) + PipelineDepth;
unsigned int Stage3_2_StopWrite  = 2*((n*n)/NumMult) + PipelineDepth;

// if PipelineDepth + (n**2)/NumMult > 2*(n**2)/NumMult, then Stage3_3_StartRead should be
//Stage3_2_StartWrite
unsigned int Stage3_3_StartRead  = Stage3_2_StartWrite;
unsigned int Stage3_3_StartSA    = Stage3_3_StartRead + PipelineDepth - ReadLat;
unsigned int Stage3_3_StartInv   = Stage3_3_StartRead + PipelineDepth + AddLat;
unsigned int Stage3_3_StartWrite = Stage3_3_StartRead + PipelineDepth + AddLat + InvLat;
unsigned int Stage3_3_StopWrite  = Stage3_3_StartRead + PipelineDepth + AddLat + InvLat + ReadLat;

unsigned int Stage3_4_StartRead  = Stage3_3_StartRead +   (n/NumMult);
unsigned int Stage3_4_StopRead   = Stage3_3_StartRead + 2*(n/NumMult);
unsigned int Stage3_4_StartSA    = Stage3_3_StartRead +   (n/NumMult) + PipelineDepth - ReadLat;
unsigned int Stage3_4_StartWrite = Stage3_3_StartRead +   (n/NumMult) + PipelineDepth + AddLat;
unsigned int Stage3_4_StopWrite  = Stage3_3_StartRead +   (n/NumMult) + PipelineDepth + AddLat + ReadLat;

unsigned int Stage3_5_StopRead   = Stage3_3_StartRead + PipelineDepth + AddLat + InvLat +   ReadLat + n;
unsigned int Stage3_5_StartWrite = Stage3_3_StartRead + PipelineDepth + AddLat + InvLat + 2*ReadLat +
MultLat;
unsigned int Stage3_5_StopWrite  = Stage3_3_StartRead + PipelineDepth + AddLat + InvLat + 2*ReadLat +
MultLat + n;
```

Stage 4 Variables

```
// Stage 4 of LQG
unsigned int Stage4_1_StartWrite = MultLat + ReadLat;

unsigned int Stage4_2_StartRead  = (n/NumMult);
unsigned int Stage4_2_StartWrite = MultLat + ReadLat + (n/NumMult);

unsigned int Stage4_3_StartRead  = (n/NumMult) +   ((n*n)/NumMult);
unsigned int Stage4_3_StopRead   = (n/NumMult) + 2*((n*n)/NumMult);
unsigned int Stage4_3_StartWrite = (n/NumMult) +   ((n*n)/NumMult) + MultLat + ReadLat;
unsigned int Stage4_3_StopWrite  = (n/NumMult) + 2*((n*n)/NumMult) + MultLat + ReadLat;
```

Stage 5 Variables

```
// Stage 5 of LQG
unsigned int Stage5_1_StartWrite = AddLat + ReadLat;

unsigned int Stage5_2_StartRead  = (n/NumMult);
unsigned int Stage5_2_StartWrite = (n/NumMult) + AddLat + ReadLat;

unsigned int Stage5_3_StartRead  = (n/NumMult) +   ((n*n)/NumMult);
unsigned int Stage5_3_StopRead   = (n/NumMult) + 2*((n*n)/NumMult);
unsigned int Stage5_3_StartWrite = (n/NumMult) +   ((n*n)/NumMult) + AddLat + ReadLat;
unsigned int Stage5_3_StopWrite  = (n/NumMult) + 2*((n*n)/NumMult) + AddLat + ReadLat;
```

Stage 6 Variables

```
// Stage 6 of LQG
unsigned int Stage6_1_StartWrite = PipelineDepth;

unsigned int Stage6_2_StartRead  =    ((m*n)/NumMult);
unsigned int Stage6_2_StopRead   = (2*((m*n)/NumMult)) + ReadLat;
unsigned int Stage6_2_StartWrite =    ((m*n)/NumMult)  + PipelineDepth;
unsigned int Stage6_2_StopWrite  = (2*((m*n)/NumMult)) + PipelineDepth;

unsigned int Stage6_3_StopRead   = (2*((m*n)/NumMult)) + PipelineDepth + ReadLat + (m/NumMult);
unsigned int Stage6_3_StartWrite = (2*((m*n)/NumMult)) + PipelineDepth + ReadLat + AddLat;
unsigned int Stage6_3_StopWrite  = (2*((m*n)/NumMult)) + PipelineDepth + ReadLat + AddLat + (m/NumMult);
```