HW/SW Configurable LQG Controller using a Sequential Discrete Kalman Filter

Matthew Cauwels, Joseph Zambreno, Phillip H. Jones
Dept. of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA 50010
Email: {mcauwels, zambreno, phjones}@iastate.edu

***Abstract*—A hardware/software configurable architecture for Linear Quadratic Gaussian (LQG) control is presented, which is a combination of a Linear Quadratic Regulator (LQR) control law with a Discrete Kalman Filter (DKF) state estimator. LQG controllers are ideal candidates for hardware acceleration, since the DKF algorithm requires matrix inversion, which is time consuming and potentially parallelizable. In this design, a functionally equivalent DKF method, called the Sequential Discrete Kalman Filter (SDKF), is used to transform the matrix inversion into an iterative scalar inversion. The proposed design acts as an Intellectual Property (IP) Core, where the user can adjust scaling parameters in hardware and configuration parameters in software to tailor the given architecture to a wide range of physical systems. This differentiates the proposed design from other LQG implementations since it is not application specific; in fact, this architecture, which was targeted for a Xilinx Zynq-7020 FPGA, allows for systems of state size 4 to 128 and achieves a speedup of 23.6 to 167 over a 2.7GHz quad-core processor. The goal of this approach is to support a design methodology for bridging the gap between control theory and embedded systems applications. For evaluation, this architecture was compared to a pure software LQG implementation. Additionally, the approach and results of recent LQG and LQG-related hardware designs were analyzed and compared to the proposed design.**

***Index Terms*—FPGA, LQG, Kalman Filter, HW/SW co-design**

I. Introduction

As technology advances, Cyber Physical Systems (CPS) are becoming common among many different research domains [1]. However, since CPS involve intricate knowledge of both the physical system and the computational device, we assert that collaborations among researchers of varying backgrounds are becoming more and more necessary. One helpful tool for connecting engineers of many backgrounds are field-programmable gate arrays (FPGAs), which are frequently used to prototype algorithm acceleration.

Due to the increasing complexity of state-of-the-art controllers, control engineers have turned to FPGAs to accelerate these computationally intense algorithms to yield practical controllers [2]–[6]. FPGAs are ideal candidates due to the immense opportunity for parallelism, which can lead to faster controller update rates. Additionally, a hardware-software codesigned FPGA controller can allow for a partitioning of system requirements among the collaborating engineers (i.e., the workload of a controller design can be separated into specialized tasks suited for each engineer).

Without this collaboration among different disciplines, implementing such an algorithm on an FPGA would become a difficult endeavor, especially for those unfamiliar with hardware design. One control algorithm that is a prime target for hardware acceleration is the Linear Quadratic Gaussian (LQG) control algorithm. The LQG controller is composed of an optimal state-feedback control law and a Kalman Filter state estimator. Kalman Filters are computationally intense, so software implementations for high-order systems are infeasible, due to resulting in slow sample rates [7]. To this end, we have developed a software-configurable FPGA-based co-processor architecture that implements an LQG control algorithm for a wide range of systems. This design is intended to be used as an Intellectual Property (IP) Core, similar to those created by Xilinx and Altera. To implement this IP core, a control engineer would need to provide system specifications (state size, sample rate, etc.) to generate a customized architecture. A hardware engineer could then integrate this custom IP core with the peripherals for the system (sensor inputs, actuator outputs). Lastly, the software engineer would update the remaining system parameters (controller gains, weighting matrices) via software configurable registers.

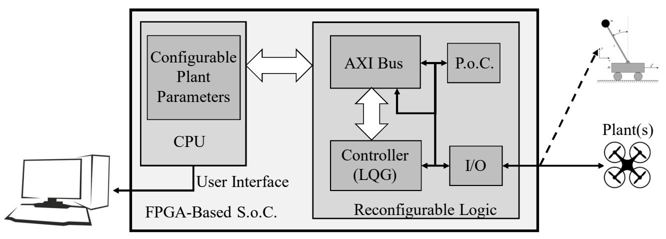

Fig. 1. An example system-level schematic showing how the hardware controller would interface with the software, Plant-on-Chip (PoC), and a variety of physical system.

A system-level overview of the proposed system is presented in Fig. 1. The intent of this design is to allow rapid prototyping of an LQG controller among a variety of physical systems, which can be emulated for correctness using a Plant-on-Chip (PoC) [8]. The PoC is a hardware model of the physical system that would allow the user to verify their controller design before interfacing with the physical plant.

Contributions. There are three primary contributions of this paper: 1) the design of a software-configurable LQG controller, with single-precision floating point accuracy, 2) the design of a modified multiply-accumulate structure to allow for reuse of the adders when performing matrix addition, and 3) the implementation of a scalable LQG controller in hardware using the Sequential Discrete Kalman Filter.

Organization. The paper is organized as follows: Section II overviews the related work in hardware implementations of matrix inversion algorithms, Kalman Filters, and software-configurable control algorithms. Section III introduces the concept of state-space modeling as well as the LQG, LQR, and SDKF algorithms. Section IV gives a detailed description of the proposed design. Section V presents the evaluation of the design against a pure software approach as well as against other closely related architectures. Section VI concludes the paper and details avenues for future work.

II. Related Work

This section consists of three parts: 1) an overview of common ways matrix inversion (which is typically the bottleneck of LQG computations) is performed in hardware, 2) a summary of recent hardware implementations of Kalman Filters (a core component of LQG control), and 3) a survey of control algorithms in FPGAs.

Matrix Inversion in Hardware: There are many ways to solve for the inverse of a matrix. The standard analytical definition of a matrix inverse (i.e., the adjoint divided by the determinant) can be efficiently implemented in hardware for matrices with low dimensionality [9]. However, as dimensionality increases, this solution becomes infeasible, since the computations grow exponentially. Another approach for solving matrix inversion is the modified Gram-Schmidt algorithm, which is based on QR decomposition. Irturk et al. took this approach in [10] and performed matrix inversion for up to an 8-dimensional matrix. Additionally, a common practice is to use a systolic array based on the modified Faddeev algorithm [7], [11]. Xu et al. proposed the SPMI algorithm in [12], which is based on the Cholesky decomposition.

Kalman Filters in Hardware: Due to hardware's ability to exploit the parallelism available in matrix inversion, plenty of applications of Kalman filters have been implemented in FPGAs. Several designs for low-order systems have been produced [7], [13], [14], with the implementation in Phuong et al. [13] achieving sample rates as low as 5μs for a 3^{rd}-order system. However, these designs are application specific and use high level design suites (LabView), which remove the user from the intricacies of the hardware design. Johnson et al. developed a Kalman Filter implementation in [11] for image denoising, where he used a systolic array structure built for a 3^{rd}-order system as a sliding window to scan through images, reporting a 512×512 image scan period of 33ms. Their approach to handled matrix arithmetic using a systolic array based on the modified Faddeev algorithm. Kettener and Paolone proposed using a Sequential Discrete Kalman Filter (SDKF) in [15] for state estimation, which replaces the matrix inversion present in DKF with a sequential sequence of scalar inversions. One common assumption must hold true for SDKF to be equivalent to DKF: the sensors are uncorrelated (i.e., the act of taking a measurement from one sensor does not impact the measurement of another sensor) [15]. This is a reasonable assumption, since most sensors are uncorrelated; however, Kettener and Palone go on to make a few more simplifying assumptions that limit the scope of their design to their specific application.

Control Algorithm Architectures: Since hardware designs are often application-specific, implementing these computationally intense controllers in hardware is time consuming. To help decrease the hardware design time, tools have been created to generate HDL from high-level applications, such as Matlab and LabView, which have been successfully used to accelerate application specific control systems [16], [17]. In fact, these high-level applications have been recently used to implement LQG controllers [2], [5]. However, few controllers have been presented that accelerate a control algorithm for a variety of applications. One example is a software configurable coprocessor for LQR control [18]. This configurable design yields a 100× factor speedup over an ARM processor; however, this design uses a Luenburger observer in conjunction with the LQR controller, which is susceptible to noisy sensor information. Our work presents a software configurable coprocessor for LQG control.

III. LQG Algorithm

A linear quadratic gaussian (LQG) controller is the combination of an optimal state-feedback control law and a least-squares regression state estimator. This section will lightly introduce these concepts. First, an overview of state-space modeling will be presented followed by a summary of the well-known linear state-feedback control law, the linear quadratic regulator (LQR). Lastly, a variation of a discrete Kalman Filter, the sequential discrete Kalman filter (SDKF), will be introduced. The LQG controller presented in this paper is a combination of an LQR control law and SDKF state estimator.

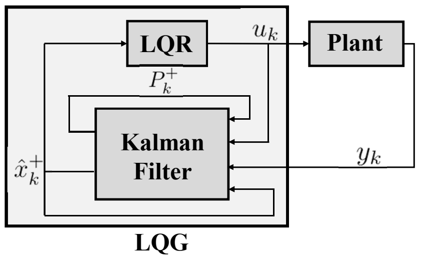

Fig. 2. The block diagram for a LQG controller.

A. State-Space Modeling

A discrete state-space model is a set of linear equations that define the future dynamics of the system based on the current dynamics as well as the current input to the system. The linear discrete time-invariant state-space model is defined as:

$$x_{k+1} = Ax_k + Bu_k \tag{1}$$

$$y_k = Cx_k + Du_k \tag{2}$$

Where

- x_k is the state of the system at time k

- u_k is the input(s) to the system at time k
- y_k is the output(s) of the system at time k
- A is a $n \times n$ matrix that describes the internal dynamics of the system
- B is a $n \times m$ matrix that describes the effect of the input(s) upon the system
- C is a $p \times n$ matrix that describes how the states of the system effect the outputs
- D is a $p \times m$ matrix that describes the direct effect the input may have on the outputs of the system
- n is the number of states of the system
- m is the number of inputs to the system
- p is the number of outputs from the system

With respect to a closed-loop control system, the state update equation (1) defines how the system transitions from state-to-state based on the systems dynamics (matrix A) and the systems response (matrix B) to inputs. Equation (2) describes the sensed outputs (y_k) with respect to the system dynamics (matrix C) and any feed-forward input (matrix D).

B. Linear Quadratic Regulator (LQR)

An LQR controller is an optimal state-feedback controller which computes the inputs to the system (u_k) by (3), where K is a static gain matrix that allows u_k to minimize the cost function (4) [19].

$$u_k = Kx_k \tag{3}$$

$$J(u) = \sum_{k=1}^{\infty} x_k^T Q x_k + x_k^T N u_k + u_k^T R u_k \tag{4}$$

The matrices Q, N, and R are known as weighting matrices and are tuned to obtain the desired state-cost, state-input cost, and input-cost, respectively [19]. By tuning these matrices, a control engineer can obtain the corresponding K that meets their control specifications (e.g., steady-state error, overshoot, settling time). Since K is obtained from optimizing a cost-function, it is said to be an optimal control law.

Combining (1) and (3), we obtain the closed-loop equation for the system

$$x_{k+1} = (A - BK)x_k \tag{5}$$

One should take note that an assumption was made: all states of the system are readily available for the control law to use. In most circumstances, this is not the case, due to the cost or impracticality of having a physical sensor for each system state. However, if the system meets the requirements for observability [19], a state estimator can be designed to estimate the unknown states of the system.

C. Sequential Discrete Kalman Filter (SDKF)

A Kalman Filter algorithm is an optimal state estimator that recursively minimizes the error of a random variable, in this case, the states of a system [20]. The algorithm consists of two stages: the prediction stage and the estimation stage. In the discrete Kalman Filter (DKF) algorithm, the estimation stage involves a $(p \times p)$ matrix inversion. Thus a variation of the DKF, the Sequential Discrete Kalman Filter (SDKF), was derived such that matrix inversion is avoided by sequentially iterating through the estimation stage to produce the state estimate using scalar inversion (for a proof of equivalence between the DKF and SDKF, see [15]).

A Kalman Filter model (6-7) is a slight variation of the equations given in (1-2).

$$x_{k+1} = Ax_k + Bu_k + w_k \tag{6}$$

$$z_k = Hx_k + v_k \tag{7}$$

The key difference is the addition of two Gaussian white noise vectors: process noise (w_k) on the state update equation and measurement noise (v_k) on the output equation. Notice that the output of the system now is labeled z_k, which is the measured states of the system. Additionally, any feed-forward component of the system has been absorbed into the state equation and matrix H is usually equivalent to C in (2).

The noise vectors of the model are assumed to be zero-mean, normally distributed, uncorrelated spectral white noise [20]. As such, the following can be defined

$$w_k \sim N(0, Q_k) \qquad v_k \sim N(0, R_k)$$

$$Q_k = [w_k w_k^T] \qquad R_k = [v_k v_k^T] \qquad E[w_k v_k^T] = 0$$

where $\sim N(\mu, \sigma^2)$ stands for normally distributed with μ mean and σ^2 variance, $E[\cdot]$ is the expected value, and Q_k & R_k are the process and measurement covariance matrices, respectfully, which are different than the weighting matrices specified in (4).

It is also necessary to distinguish between the true state of the system (x_k) and the estimated state of the system ($\hat{x}_k$). Additionally, the Kalman Filter estimates the states in two stages: a prediction stage ($\hat{x}_k^-$) and an estimation stage ($\hat{x}_k^+$). Thus, two different errors (e_k^-, e_k^+) can be defined as well as the error covariance matrices ($P_k,^-, P_k^+$):

$$e_k^- = x_k - \hat{x}_k^- \qquad e_k^+ = x_k - \hat{x}_k^+$$

$$P_k^- = E[e_k^-(e_k^-)^T] \qquad P_k^+ = E[e_k^+(e_k^+)^T]$$

For both the DKF and SDKF algorithm, the prediction stage of the Kalman Filter algorithm is defined as

$$\hat{x}_k^- = A\hat{x}_{k-1}^+ + Bu_{k-1} \tag{8}$$

$$P_k^- = AP_{k-1}^+ A^T + Q_k \tag{9}$$

For the SDKF algorithm's estimation stage, let $i = \{1, , p\}$, where p is the number of sensors/measurements taken, be the index for an iteration, then the following are defined as

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

With the initial values given in (10), (11-13) are iteratively repeated to obtain the final state estimate and error covariance matrix.

$$\hat{x}_{k,0}^+ = \hat{x}_k^- \qquad P_{k,0}^+ = P_k^- \tag{10}$$

$$K_{k,i} = P_{k,i-1}^+ H_{k,i}^T (H_{k,i} P_{k,i-1}^+ H_{k,i}^T + R_{k,i})^{-1} \tag{11}$$

$$\hat{x}_{k,i}^+ = \hat{x}_{k,i-1}^+ + K_{k,i}(z_{k,i} - H_{k,i}\hat{x}_{k,i-1}^+) \tag{12}$$

$$P_{k,i}^{+} = P_{k,i-1}^{+} - K_{k,i} H_{k,i} P_{k,i-1}^{+} \qquad (13)$$

A key assumption is that the measurement covariance matrix (R_k) is diagonal (i.e., there is no correlation among the sensors of the system). This assumption is reasonable, since the reading of one sensor rarely impacts another sensor's value.

Choosing a sequential method for hardware implementation may seem counterintuitive since this may remove opportunities for parallelism; however, the parallelism leveraged in this design comes from the individual matrix and vector computations, not necessarily the LQG algorithm. Thus, the choice to use SDKF over the DKF was two-fold: 1) scalar inversion is easier to implement than matrix inversion and 2) having a scalar inverse allows for the opportunity to create a scalable architecture to exploit the parallelism of matrix operations.

IV. Architecture

This section details the hardware architecture used to implement the LQG controller using SDKF. First, a high-level overview of the architecture is presented, followed by a description of the individual components.

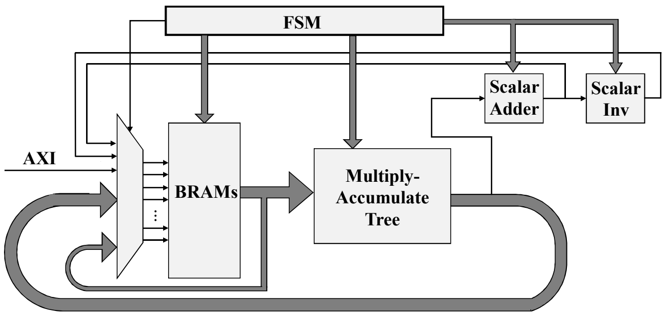

Fig. 3. Top-level schematic of the proposed architecture for the LQG controller IP Core.

A. Overview

Fig. 3 illustrates our architecture for the HW/SW configurable LQG controller. There are three main components: 1) the multiply-accumulate tree, 2) the scalar-adder and inverter, and 3) the memory management architecture. Lastly how software is used to configure the parameters of the system, such as those referenced in Section III-A, will be described as well as how sensors will interface with the LQG controller.

Table I shows the scheduling of (3), (8-9), and (11-13) into individual matrix operations. The mapping of the states to equations is as follows: states 1-2 correspond to the SDKF prediction equations (8-9), states 3-5 to the SDKF estimation equations (11-13), and state 6 to the LQR control-law in (3). Notice that the states are separated into the three modes of the multiply-accumulate tree: matrix-vector multiplication, scalar multiplication, and element-wise addition/subtraction. The LQG equations were arranged this way to maximize the throughput of the pipeline by minimizing the number of transitions between modes.

TABLE I
LQG Equation Scheduling Details

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

B. Multiply-Accumulate Tree

A multiply-accumulate structure is well suited to perform matrix-vector and matrix-matrix multiplication; however, the LQG control algorithm also requires scalar-matrix multiplication and element-wise addition. To avoid implementing another arithmetic structure, the multiply-accumulate tree was modified by fanning out the outputs of all multipliers and adders to their respective BRAMs as well as by multiplexing the adder inputs to allow for element-wise addition/subtraction (see Fig. 4). While this causes an increase in control logic, this reuse of the adders exploits the parallelism of the element-wise addition/subtraction while decreasing the overall size of the design.

The multiply-accumulate structure has three modes: 1) matrix-vector multiplication, 2) scalar multiplication, and 3) element-wise addition/subtraction. The first mode is the standard multiply-accumulate tree for matrix-vector multiplication, as seen in the bottom-left of Fig. 4. Notice that it can also be used for matrix-matrix multiplication, though the time it takes to perform this operation increases by a factor of n. Additionally, if the number of rows (n) is less than the number of multipliers, the multiply-accumulate structure will need additional circuitry to accumulate the partial sums produced by the lack of multipliers. Reduction circuits are incorporated to accumulate these partial sums. As shown in Fig. 4, the required number of reduction circuits, k, can be calculated by

$$log_2(i) + k \geq log_2(n) \qquad (14)$$

where i is the number of multipliers in the multiply-accumulate tree and n is the number of columns in the matrix.

The second mode is for scalar-matrix multiplication, which is used in state 4 (see Table I). Rather than pad the standard multiply-accumulate tree with zeros, the multiplier's outputs are fanned out of the structure early and fed back around to their respective BRAMs, which reduces the latency to that of the floating-point multipliers.

The third mode enables the reuse of the adders within the multiply-accumulate tree, which is done by multiplexing the

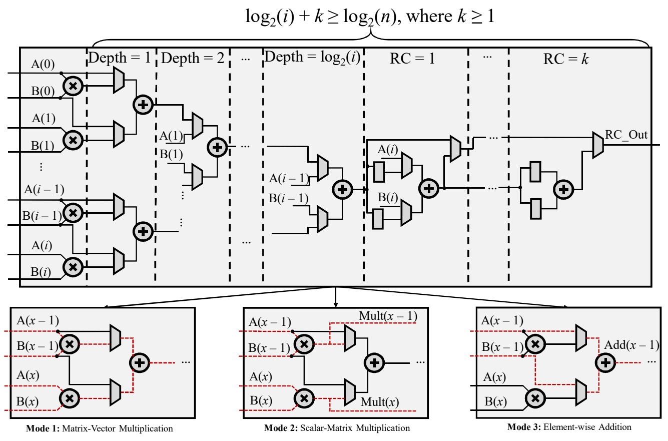

Fig. 4. Schematic of the proposed multiply-accumulate structure. Note that while it is not shown in the diagram, all floating-point multiplier & adder outputs are fanned out of this structure and fed to their respective BRAMs. Additionally, the three operating modes of the circuit are shown, where the data path for each mode is the dashed line.

inputs to the adders with the inputs to the multipliers. Since the multiply-accumulate tree has one less adder than the number of inputs, an additional adder is incorporated by generating at least one reduction circuit. While it may be unnecessary for matrix-vector multiplication, having the number of adders equivalent to the number of inputs avoids multiplexing every adder output to every Bloack RAM (BRAM), thus simplifying the control logic and allowing faster system clock rates.

The notation of the multiply-accumulate tree's $Depth$ refers to the layers of addition within the tree, as seen in Fig. 4. For this design, Xilinx floating-point v7.1 IP cores were used for floating-point addition/subtraction, multiplication, and scalar division in this design, with latencies of $Lat_+ = 12$, $Lat_\times = 9$, and $Lat_\div = 30$, respectively. Notice that if reduction circuits are needed, then they will add additional delay to the pipeline depth. By design, the pattern emerges that a reduction circuit produces valid sums every 2^k clock cycles, where k is the index of the reduction circuit. Therefore the multiply-accumulate tree's pipeline depth ($P.D.$) can be calculated by (15).

$$P.D. = Lat_\times + Lat_+(log_2 n) + \sum_{k=1}^{log_2 n - Depth} 2^k \qquad (15)$$

With (15) defined, the timing of the states in Table I can be calculated using the equations in Table II.

TABLE II
LQG State Timing Details

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

C. Scalar-Adder & Inverter

Among the equations performed in Table I, one should note that 3.d & 3.e are scalar addition ($S.A.$) & subtraction, respectively. Thus, using the pipeline to perform these scalar operations, a single floating-point adder is included outside of the multiply-accumulate tree to increase the parallelism of the system. Additionally, the scalar inversion (3.f) (Inv) is performed on the result of the scalar addition (3.d), so the output of this scalar-adder is fed into a floating-point scalar inverter, as seen in Fig. 5. With reference to Fig. 3, both the output of this adder and inverter are fed into the multiplexer so that the results may be stored in memory.

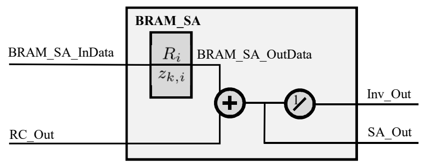

Fig. 5. Schematic of the scalar-adder & inverter. Having a scalar-adder & inverter increases the system's parallelism and avoids implementing a separate matrix inversion architecture.

Additionally, since the measurement covariance matrix ($R_{k,i}$) and the sensor inputs ($z_{k,i}$) are only used 3.d and 3.e of Table I, a specific BRAM, *BRAM_SA*, is used to store these values.

D. Memory Management Architecture

BRAMs are used to store the constants and temporary (T) values used in the Table I equations. For each multiplier there are two BRAMs, *BRAM_A* and *BRAM_B*, as seen in Fig. 6. This was done to denote their correspondence to their particular input to the multiply-accumulate structure.

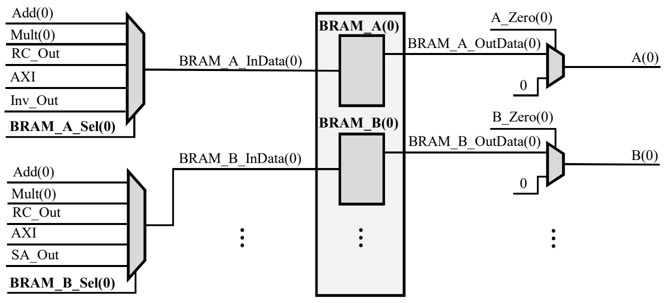

Fig. 6. Schematic of the memory management system. The multiplexed outputs of the BRAMs allow for zeroes to be fed into the multiply-accumulate tree when not reading from memory.

There were several memory storage issues to consider when designing the controller. The first was determining which outputs of the multiply accumulate tree should be fed back into each BRAM. Another was determining how to schedule the constants and temporary variables in memory. Lastly, a mechanism for coordinating matrix-matrix multiplication as well as matrix-addition was needed.

The first design challenge was determining the inputs to each BRAM. As seen in Fig. 6, the output of each BRAM's corresponding adder and multiplier are fed back to allow for addition/subtraction and scalar-matrix multiplication (e.g., *Add(0)* & *Mult(0)* are inputs to *BRAM_A*(0) & *BRAM_B*(0)). The output of the multiply accumulate tree (*RC_Out*) is fed back to every BRAM to store of the results of matrix-vector and matrix-matrix multiplication. Additionally, the AXI interface is an input into the BRAMs so that they can be initialized via software. *BRAM_A* also has the output of the inverter and *BRAM_B* has the output of the scalar adder.

Another consideration was which constants and temporary variables to put into each BRAM. Table III shows the breakdown for which constants were stored in which BRAM and gives the formulation for the size of the variables given the parameters n, m, p, and $Depth$.

TABLE III
BRAM CONSTANT AND VARIABLE MEMORY ALLOCATION

BRAM_A	***BRAM_B***
$A = \frac{n^2}{2^{Depth}}$	$A^T = \frac{n^2}{2^{Depth}}$
$P_k = \frac{n^2}{2^{Depth}}$	$T_{0,B} = \frac{n^2}{2^{Depth}}$
$T_{1,A} = \frac{n}{2^{Depth}}$	$T_{1,B} = \frac{n}{2^{Depth}}$
$T_{2,A} = 1$	$T_{2,B} = 1$
$H_k = \frac{np}{2^{Depth}}$	$H_k = \frac{np}{2^{Depth}}$
$B = \frac{nm}{2^{Depth}}$	$Q_k = \frac{n^2}{2^{Depth}}$
$K_{lqr} = \frac{mn}{2^{Depth}}$	$T_{3,B} = n$
	$x_k = \frac{n}{2^{Depth}}$
	$u_k = \frac{m}{2^{Depth}}$

Notice that $T_{3,B}$ is of size n. This is due to state 4.b of Table I, which calculates the outer product (i.e., a $(n \times 1)$ vector is multiplied by a $(1 \times n)$ vector to create a $(n \times n)$ matrix). Note that the outer product requires each element of the vector to be multiplied by each element of the other vector. Rather than constructing a resource heavy cross-bar between the BRAMs and the inputs to the multiply-accumulate tree, every *BRAM_B* holds the complete $(n \times 1)$ vector $P_k H_{k,i}^T$ in $T_{3,B}$. While this design choice requires additional logic to perform the computations in states 3.b & 4.a, this is a much more advantageous design choice since an n^2 crossbar is avoided and this method helps to equalize the number of memory locations used in both *BRAM_A* and *BRAM_B*.

The last design consideration was how to store the matrices across multiple BRAMs. Intuitively, there were two logical ways to serially store matrices: 1) row-major order or 2) column-major order. A method for switching between these two storage mechanisms is needed to perform both matrix addition and matrix multiplication. The mechanism used for storing matrices is shown in Table IV. This switching mechanism is utilized in state 1.d of Table I to align P_k for matrix-addition.

E. Software & Peripheral Interface

Software is used to allow the user to input the system parameters, i.e., dimensions of the system (n, p, and m) and the system matrices (A, B, K_{lqr}, etc.). The dimensions of the system will be used in the software to calculate the timing

TABLE IV
BRAM MANAGEMENT MECHANISM

```
BRAM_WriteAddr($i$) $\leftarrow$ BaseAddr $+j$
**Procedure** Switch Storage Scheme
**If** $((i \geq 2^{Depth} - 1)\&\&(k \geq \frac{n^2-1}{2^{Depth}}))$
    $i = 0;\ j = l + 1;\ k = \frac{n}{2^{Depth}};\ l = l + 1;$
**Else**
    $j = k + l;$
    **If**$(k \geq \frac{n^2-1}{2^{Depth}})$
        $i = i + 1;\ j = l;\ k = \frac{n}{2^{Depth}};$
    **End if;**
    $k = k + \frac{n}{2^{Depth}};$
**End if;**
**Procedure** Maintain Storage Scheme
**If** $(i \geq \frac{n}{2^{Depth}})$
    $i = 0;$
    **If** $(j \geq \frac{n^2}{2^{Depth}})$
        $j = 0;$
    **Else**
        $j = j + 1;$
    **End if;**
**Else**
    $i = i + 1;$
**End if;**
```

of the FSM and configuration of the hardware (e.g., how to configure the multiplexers between reduction circuits). To initialize the hardware controller, the calculated configuration values will be loaded into software-configurable registers that interface with the hardware and the matrix coefficients will be loaded into the BRAMs.

Additionally, physical sensors must interface with the controller. Sensor values must be regularly converted from raw sensor data into their floating-point values and stored into *BRAM_SA*. This can be done in software or hardware, though the user will need to consider their design constraints to choose which method is better suited for each particular sensor input.

V. HARDWARE IMPLEMENTATION AND ANALYSIS

A. Evaluation Methodology

Our software configurable LQG controller was designed in VHDL using the Vivado 2017.1 Design Suite. The prototype target was a Zedboard with a Xilinx Zynq FPGA (XC7Z020). The Zync FPGA consists of a reconfigurable fabric for custom designs as well as a dual-core ARM Cortex-A9 processor with configurable clock frequency of 100-667MHz. The LQG algorithm presented in Table I was verified for a inverted pendulum system in Matlab. The results of our LQG controller were equivalent to the results obtained via Matlab, validating that the LQG control algorithm was implemented correctly.

Two sets of experiments were performed. First, the LQG with SDKF was implemented in C. Varying the number of states, the amount of time to complete one iteration of the LQG controller was experimentally obtained for an ARM processor (with varying clock frequencies) and a 2.70GHz quad-core processor. These results were compared to the hardwares analytically calculated timing results for varying state size and pipeline depth. The second was to compare the timing calculations and size of design against that of other reported controllers and Kalman filters of similar state dimensions.

TABLE V
SOFTWARE LQG W/ SDKF ITERATION TIME

Cortex-9 ARM Dual-Core CPU	Size ($n = m = p$)					
Clock Rate	4	8	16	32	64	128
100 MHz	830μs	4.30ms	28.5ms	225ms	1.80s	15.3s
333 MHz	251μs	1.29ms	8.57ms	67.5ms	540ms	4.65s
667 MHz	128μs	662μs	4.39ms	34.6ms	277ms	2.38s
AMD FX-9800 2.70GHz Quad-Core	-	-	-	3.00ms	32.0ms	199ms

B. Software Comparison

Tables V and VI report the amount of time it takes for the software and hardware to complete one iteration of the LQG equations, respectively. With the hardware running at 100MHz, the results show that there is a 15.5 to 2017 factor speed-up over the embedded ARM processor running at 667MHz, with $n = 4$ to $n = 128$ states, respectively. Compared with a quad-core processor running at 2.7GHz, the results show a 23.6 to 167 factor speed-up, with $n = 32$ to $n = 128$, respectively.

TABLE VI
HARDWARE ITERATION TIME

Depth	Size ($n = m = p$)					
	4	8	16	32	64	128
1	9.41μs	31.5μs	150μs	949μs	6.94ms	53.8ms
2	8.27μs	23.2μs	91.3μs	510μs	3.55ms	27.0ms
3	-	19.1μs	62.1μs	291μs	1.85ms	13.7ms
4	-	-	47.5μs	182μs	1.00ms	7.01ms
5	-	-	-	127μs	582μs	3.68ms
6	-	-	-	-	671μs	2.01ms
7	-	-	-	-	-	1.18ms

Additionally, one will note that in Table VI, the timing equations for a system with more multipliers than the number of states is not reported. This is due to the design choice of not allowing every stage of the multiply-accumulate tree to fan out back to the BRAMs. The routing for longer depth pipelines would consume a large amount of resources, due to the additional multiplexers needed at the input to the BRAMs, which will decrease the system clock frequency.

TABLE VII
HARDWARE LQG RESOURCE UTILIZATION (ZYNQ - 7020)

SystemSize		LUTs	FFs	BRAMs	DSPs	Max. f_{clk}.
Depth	# of $\times$	$53,200$	$106,400$	140	220	MHz
2	4	4,682	4,698	10	20	150.128
3	8	7,052	7,726	18	44	145.349
4	16	11,675	13,762	4	76	145.285
5	32	21,903	25,833	66	140	138.045

C. Comparison Among Related Work

There are four main works that will be used for comparison: 1) a LQG controller implemented using Matlab HDL Coder [2], 2) a DKF implementation using a systolic array [11], 3) another SDKF algorithm implementation [15], and 4) the Software Configurable LQR controller [18].

An application of a LQG controller was presented in [2]. While their focus is on sensor selection, they implement a LQG controller for their 3^{rd}-order system using Matlab's HDL Coder. When compared with our fully-pipelined system with $n = 4$, we achieve similar results in terms of sample rates and resources consumed. This result validates that our design could behave as an IP core in this application.

A DKF for image denoising using a systolic array is given in [11]. Their approach implements a (3×3) DKF and report a 310MHz clock frequency with a 112ns latency to produce a single denoised pixel, after an initial 270μs latency to fill the pipeline. Since it was designed for a specific purpose (image processing), their focus was on improving throughput (i.e., how quickly can they iterate through a small dimensional (3×3) Kalman Filter). While their design is efficient for their 3^{rd}-order system, this is a highly specialized design which would require a large amount of time to tailor towards a different size system.

As presented in [15], a SDKF state estimator was designed and implemented; however, several differences exist between their design and the proposed one. First, they limited the scope of their application to systems with a state transition matrix $A = I_{n \times n}$ and input matrix $B = 0$. Additionally, since they were only filtering their data, they did not have a LQR gain matrix K_{lqr}. For their largest system ($n = 256$), they had a pipeline depth of 6 (i.e., 64 multipliers wide) and reported using 262 BRAMs with an execution time of 35ms [15]. Should our design match their proposed system (i.e., $n = 256$ and $Depth = 6$) we would consume approximately 768 BRAMs and get a sample rate of 14.0ms (assuming a clock frequency of 100MHz). The reason our design requires so much memory is that our design is more general, i.e., we did not limit our design to systems with $A = I_{n \times n}$ and $B = 0$.

For the software configurable LQR controller in [18], the authors reported for a configuration of $n = m = p = 4$ and a pipeline depth of 1 (i.e., two multipliers), one control-loop iteration took 0.73μs. For $n = m = p = 128$ and a pipeline depth of 7 (i.e., 128 multipliers), they reported one iteration of their control loop took 3.73μs. Based on these results, this LQR implementation has a 12.9 to 316 factor speedup over the LQG computation. Intuitively, it makes sense that the LQR computations should take far less time than the LQG, since the an LQR control law is one small computation within the LQG. However, several design considerations are needed when choosing between an LQR and an LQG controller. The main difference between the two is that LQG models and filters out sensor and system noise whereas the LQR does not. So, if the system is susceptible to noise, an LQG controller will have better control performance than an LQR controller. Another design consideration should be minimum sample rate. If a small sample rate is desired, then the LQR controller may be better suited for that system than the proposed LQG design.

VI. CONCLUSION

A hardware/software configurable LQG controller was presented that leveraged the sequential discrete Kalman Filter

TABLE VIII
HARDWARE RESOURCE UTILIZATION AND TIMING BETWEEN RELATED WORK

	Method	Data Format	FPGA Series	System Size	Max. f_{clk}	LUTs	FFs	DSPs	BRAMs	Min. T_{samp}
This Paper	LQG	Floating-Point	Zynq-7020	$n = 4$	150MHz	4,682	4,698	20	10	6.78μs
	LQG	Floating-Point	Zynq-7020	$n = 32$	138MHz	21,903	25,833	140	66	447μs
	LQG	Floating-Point	Zynq UltraScale+	$n = 128$	161MHz	81,013	98,175	524	258	15.7ms
[2]	LQG	Fixed-point	Virtex-6	$n = 3$	25MHz	4,012	2,410	73	3	10μs
[11]	DKF	Fixed-Point	Virtex-6	$n = 3$	310MHz	4,438	2,821	91	81	122ns
[15]	SDKF	Floating-Point	Kintex-7	$n = 256$	-	43,166	49,088	357	262	35ms
[18]	LQR	Floating-Point	Zynq-7020	$n = 32$	122MHz	42,138	48,143	128	66	1.57μs

to avoid matrix inversion and allow for a scalable hardware architecture. This controller functions like an IP Core, which is intended to help bridge the gap between control and embedded systems engineers. This hardware LQG controller had a 23.6 to 167 factor speedup over 2.70GHz quad-core processor for systems of size $n = 4$ to $n = 128$, respectively. A continuation on this work would be to incorporate a Plant-on-Chip (PoC) into this design to allow a user to verify system correctness before interfacing with their physical system. In addition, a comparison to Xilinx's High-Level Synthesis (HLS) and a GPU implementation of a comparable LQG control algorithm might provide further insight into the design's performance and validate our design methodology. In a parallel direction, future work may be to incorporate more complex control algorithms (H_∞, partial feedback linearization, adaptive control) into configurable architectures.

REFERENCES

[1] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: a cyber-physical systems approach*. MIT Press, 2017.

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

[3] E. Monmasson and M. Cirstea, "Guest editorial special section on industrial control applications of fpgas," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1250–1252, Aug 2013.

[4] Z. Wanli, L. Guoxin, and W. Lirong, "Research on the control method of inverted pendulum based on kalman filter," in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, Aug 2014, pp. 520–523.

[5] R. Priewasser, M. Agostinelli, C. Unterrieder, S. Marsili, and M. Huemer, "Modeling, control, and implementation of dcdc converters for variable frequency operation," *IEEE Transactions on Power Electronics*, vol. 29, no. 1, pp. 287–301, Jan 2014.

[6] S. Kozak, "Advanced control engineering methods in modern technological applications," in *Proceedings of the 13th International Carpathian Control Conference (ICCC)*, May 2012, pp. 392–397.

[7] Y. T. Lai, A. Bigdeli, and M. Biglari-Abhari, "An optimised systolic array-based matrix inversion for rapid prototyping of kalman filters in fpga's," in *2004 12th European Signal Processing Conference*, Sept 2004, pp. 2035–2038.

[8] S. Vyas, C. K. N. G., J. Zambreno, C. Gill, R. Cytron, and P. Jones, "An fpga-based plant-on-chip platform for cyber-physical system analysis," *IEEE Embedded Systems Letters*, vol. 6, no. 1, pp. 4–7, March 2014.

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

[10] "An efficient fpga implementation of scalable matrix inversion core using qr decomposition," 2007.

[11] B. Johnson, N. Thomas, and J. S. Rani, "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)*, Jan 2017, pp. 55–60.

[12] Y. Xu, D. Li, Y. Xi, J. Lan, and T. Jiang, "An improved predictive controller on the fpga by hardware matrix inversion," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 9, pp. 7395–7405, Sept 2018.

[13] T. T. Phuong, C. Mitsantisuk, K. Ohishi, and M. Sazawa, "Fpga-based wideband force sensing with kalman-filter-based disturbance observer," in *IECON 2010 - 36th Annual Conference on IEEE Industrial Electronics Society*, Nov 2010, pp. 1269–1274.

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

[15] A. M. Kettner and M. Paolone, "Sequential discrete kalman filter for real-time state estimation in power distribution systems: Theory and implementation," *IEEE Transactions on Instrumentation and Measurement*, vol. 66, no. 9, pp. 2358–2370, Sept 2017.

[16] N. N. Al-Saaty, M. Algreer, and M. Armstrong, "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)*, June 2017, pp. 490–495.

[17] M. J. Sumam and G. Shiny, "A rapid development technique for prototype fpga controllers," in *2017 International Conference on Inventive Systems and Control (ICISC)*, Jan 2017, pp. 1–5.

[18] P. Zhang, A. Mills, J. Zambreno, and P. H. Jones, "A software configurable and parallelized coprocessor architecture for lqr control," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015, pp. 1–8.

[19] C. L. Phillips, H. T. Nagle, and A. Chakrabortty, *Digital control system analysis & design*. Pearson Prentice Hall, 2015.

[20] R. G. Brown and P. Y. C. Hwang, *Introduction to Random Signals and Applied Kalman Filtering: with Matlab Exercises and Solutions*. John Wiley and Sons, 2012.