J. Parallel Distrib. Comput. 106 (2017) 121–131

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

The design and integration of a software configurable and parallelized coprocessor architecture for LQR control

Pei Zhang *, Aaron Mills, Joseph Zambreno, Phillip H. Jones

Electrical and Computer Engineering, Iowa State University, Ames, IA 50010, USA

HIGHLIGHTS

- A software-configurable FPGA-based LQR coprocessor is proposed for Cyber–Physical Systems.
- Improved hardware-level parallelism is achieved by refactoring standard LQR equations.
- Control of an inverted pendulum is used to compare software and hardware performance.
- A 3.4 to 100 factor speedup over a 666 MHz embedded ARM processor is demonstrated.
- Details are provided for interfacing the compute logic with a physical motor.

ARTICLE INFO

Article history:
Received 2 July 2016
Received in revised form
6 January 2017
Accepted 30 January 2017
Available online 9 February 2017

Keywords:
FPGA
State space control
LQR
Co-processor
Parallel processing
Inverted pendulum

ABSTRACT

The increasing integration of computing into the physical systems we rely on everyday motivates the need to more easily marry advanced control theory, which is used to control these systems, with the computing platforms used to implement the controllers. This article explores one path of easing this integration using reconfigurable hardware technology, and discusses practical system-level details that must be addressed for integrating our idea into real-world systems. We present a software configurable and parallelized coprocessor architecture for LQR control that can control physical processes representable by a linear state-space model. Our proposed architecture has distinct advantages over purely software or purely hardware approaches. It differs from other hardware controllers in that it is not hardwired to control one or a small range of plant types (e.g. only electric motors). Via software, an embedded systems engineer can easily reconfigure the controller to suit a wide range of control applications that can be represented as a state-space model. One goal of our approach is to support a design methodology to help bridge the gap between controls and embedded system software engineering. Control of the well-understood inverted pendulum on a cart is used as an illustrative example of how the proposed hardware accelerator architecture supports our envisioned design methodology for helping bridge this gap. Additionally, we explore the design space of our co-processor's parallel architecture in terms of computing speed and resource utilization. Our performance results show a 3.4 to 100 factor speedup over a 666 MHz embedded ARM processor, for plants that can be represented by 4 to 128 states, respectively. This article concludes with a discussion of the practical integration details required for interfacing the controller with a real inverted pendulum–cart system.

© 2017 Elsevier Inc. All rights reserved.

1. Introduction

Cyber–Physical Systems (CPS) can be considered as systems that have or require a tight coupling between their computing and physical aspects, where correct behavior requires correct timing [12]. We assert that advancing science, technology, and engineering to manage and exploit this tight coupling will require improved interaction between researchers from computing domains (e.g. software and hardware engineering), and physical domains (e.g. controls engineering), and that field-programmable gate arrays (FPGAs) can act as a medium to help bridge some of the gaps between these research fields.

FPGAs are of growing interest in the area of applied control theory [18]. In addition to the massive parallelism available on FPGAs that can potentially be utilized to obtain high controller update rates, software–hardware co-design using FPGAs can help separate embedded software concerns (e.g. real-time scheduling

* Corresponding author.
E-mail addresses: peizhang@iastate.edu (P. Zhang), ajmills@iastate.edu (A. Mills), zambreno@iastate.edu (J. Zambreno), phjones@iastate.edu (P.H. Jones).

http://dx.doi.org/10.1016/j.jpdc.2017.01.028
0743-7315/© 2017 Elsevier Inc. All rights reserved.

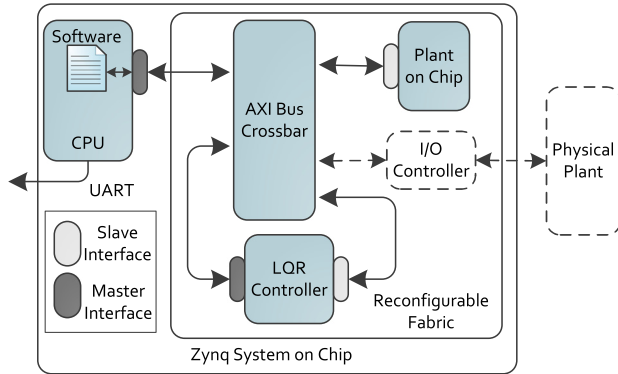


Fig. 1. System overview. An example system-level organization where the controller and Plant-on-Chip (PoC) are fully software configurable.

feasibility), from controls concerns (e.g. accounting for update-rate jitter).

Efficient implementation of a control algorithm on an FPGA can be challenging for engineers unfamiliar with hardware architecture design. One solution is software programmable hardware. In our work, we describe a software-configurable FPGA co-processor architecture that can implement a wide range of linear state-space controllers, up to the complexity of a Linear–Quadratic Regulator (LQR) coupled with a Luenberger Observer [14]. For the purpose of evaluation, the controller can be interfaced to a hardware-based emulation of a physical plant using what we will refer to as a Plant-on-Chip (PoC) [28]. This arrangement is depicted in Fig. 1. The PoC allows for rapid and consistent testing of control algorithms and system platform configurations. Once stability of the emulated plant is achieved, it can be replaced with an interface to the actual plant's sensors and actuators. All control computations are done in hardware, while software running on the CPU is used to initialize the co-processor. The software is also free to perform other activities: task scheduling, path planning, video processing, or interactive communications.

The big picture usage model for our software configurable co-processor based controller is that control engineers would focus on the mathematics of their controller, without the concern of computing artifacts breaking assumptions, such a deterministic sample rates or the representable range of numbers. Meanwhile, embedded system engineers would focus on making efficient utilization of CPU resources, without the concern of stringent timing constraints often associated with controlling physical plants. Control engineers would interact with embedded systems engineers by providing them with the appropriate state-space matrices require to program the co-processor.

Contributions. The five primary contributions of this paper are as follows: (1) A discussion of a process for bridging the Control Theory and Implementation gap, (2) the implementation of a software-configurable LQR co-processor using single-precision floating point arithmetic that helps bridge the gap between controls and embedded system engineering [17], (3) the design space exploration of the proposed parallel architecture [30] which extends the sequential architecture implemented in [17], (4) a transformation of the standard LQR control algorithm to make it better suited for hardware implementation [30], (5) a methodology of applying the system to a different state space control algorithm and (6) a discussion of implementation details for interfacing of our parallel computing architecture to a real-world plant.

Organization. The remainder of this paper is organized as follows. In Section 2, we illustrate the role of specialized compute architectures and design processes in bridging the gap between control theory and high-reliability implementation. In Section 3, we discuss and compare related works in this problem space. Section 4 gives a brief introduction to the concept of state-space modeling and the LQR control algorithm. This section also describes a transformation for converting the standard representation of the LQR controller into a form that is better suited for hardware implementation. In Section 5, we describe the detailed design of our software configurable parallelized co-processor architecture. Section 6 presents an illustrative example of using our co-processor to evaluate the use of hardware verses software for an embedded controls application, and explores the performance and scaling of our coprocessor-based controller. Section 7 demonstrates the integration of our LQR controller with the input–output hardware necessary to control an inverted pendulum. Finally, Section 8 concludes this paper and provides avenues of future work.

2. Bridging control theory and implementation

In control theory, a discrete-time control process consists of three tasks: input, computation, and output. The input task typically involves an analog to digital conversion. For model-driven controllers, the computation task typically consists of state estimation followed by control value computation. Lastly, the output task involves converting the control value into a physical signal (i.e. digital to analog conversion). From a theoretical standpoint, the standard ideal assumption is that the input occurs at a constant interval, and the computation and output process take zero time.

In practice, the compute platform on which a control system executes can negatively influence the performance of a control system which may be otherwise theoretically sound. Consequently, the practical implementation of control systems is a rich area of research, with a great deal of focus on establishing deterministic timing in complex software systems. A large body of research is concerned with carefully designed scheduling methods [13,24] which seek to impose additional determinism on task runtime characteristics. Unfortunately, these methods tend to impose additional runtime overhead, and in the best case still result in a stochastic sampling model. Online compensators are often proposed as means to reduce the impact of control loop jitter; for example, using timestamps [13] or a mixture of control theoretic and scheduling compensations [15]. The work in [15] reported a $O(n^4)$ runtime complexity overhead, and significant memory requirements (relative to common microcontrollers) for a lookup table oriented implementation. Software-based compensatory measures may help reduce timing uncertainty, but the associated implementation overhead tends to place limitations on the maximum sample rate which can be reached in such a system.

Lee, in his position paper [11], focuses explicitly on the limitations of traditional compute architectures in CPS which make them non-ideal for control systems. For instance, the many advances in software processor architecture (deep pipelines, speculation, etc.) makes software timing prediction difficult or in some cases, impossible. In particular, high-confidence Worst-Case Execution Time (WCET) analysis requires a detailed hardware-timing model, which is very hard to obtain for processors designed primarily for completing a high number of instructions per clock [29]. In concluding, Lee makes a call for research on new ways of approaching CPS compute architecture, including mixed hardware–software design methods, which better support the unique constraints of the field.

We pursue this goal by combining model-based control algorithms, facilitated by a control (or application domain) expert, with a purpose-built, model-agnostic FPGA compute platform which is facilitated by a digital design expert. In this way, we greatly reduce the dependency of the theoretical construction of the control system on the characteristics of the implementation platform. In addition to providing timing guarantees, implementing the control

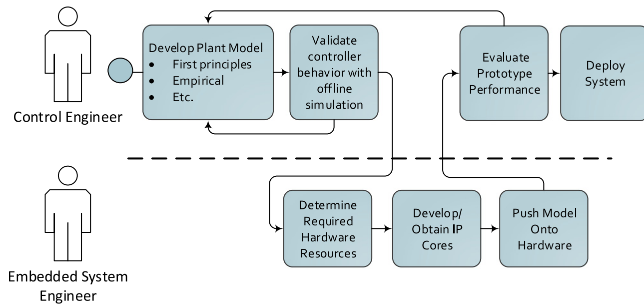


Fig. 2. Model-driven design flow using proposed FPGA-based coprocessor.

loop in hardware helps to isolate it from lower-criticality software tasks while still maintaining a very high level of system integration.

In many engineering disciplines, the model-driven design process is becoming increasingly attractive since it reduces risk and uncertainty earlier in a project-that is, when changes are easier and cheaper to make. Developing system models may appear to require more effort up front, but allows better integration with automated workflows, and enables stronger guarantees of reliability and performance [22]. The coupling of model-based design with the power and flexibility of an FPGA allows precisely timed, high speed control systems to be developed with reduced effort using design tools such as Matlab and Simulink [4].

Such a workflow is depicted in Fig. 2. In tune with modern semiconductor design paradigms, our proposed FPGA-based controller is parameterizable and application-agnostic, and can therefore be packaged as a portable "Intellectual Property" (IP) core (or block) and be reused across diverse control projects. The base computational IP core can be quickly interfaced with off-the-shelf IP cores providing connectivity with sensors and actuators, or with custom project-specific interfaces, as described in Section 7. Note that we therefore consider the FPGA to be not just a means of prototyping, but also a critical component of the final production hardware.

3. Related work

Kozak, in [10], surveys trends in the field of applied controls, in which we see controls have evolved from manually-tuned single-input single-output (SISO) controllers (e.g. Proportion Integral Derivative (PID) control) to multiple-input multiple output (MIMO) controllers (e.g. H_∞ control and model-predictive controller (MPC)). The latter types of algorithms are computationally intense and can introduce significant latencies when implemented with off-the-shelf processing platforms. Kozak additionally suggests that a software–hardware co-design approach for implementing advanced controllers (e.g. H_∞) in FPGAs would enable designers to make better use of these complex controllers in high-speed systems. Monmasson, in [19], makes a similar suggestion, pointing out how different parts of a control algorithm are better suited for different types of hardware. However, locating the optimal software–hardware partition is still a challenge. Besides MPC, the Kalman Filter is another algorithm which often appears in control systems, and is also quite computationally expensive, especially for embedded systems. We proposed a mixed hardware–software FPGA-based Kalman Filter in prior work [16] which was successful in providing a speedup over software alone.

There are numerous examples of application-specific FPGA-based controllers in the literature. An example of a system requiring very fast control update rates appears in [21], in which a high-speed pan/tilt camera is designed to track objects. In order to reach the 3.5 ms update rate, a dedicated PC is used to perform image processing and produce motor control signals. It is noted that the PC introduced considerable delay in the feedback loop. Another application requiring very high update rates appears in [26], which presents an application-specific design that used machine vision to control an inverted pendulum. In [7], the authors developed a self-tuning state-space controller using a multiply-accumulate unit which is interfaced with a digital signal processor (DSP). The use of FPGAs to control a plant with non-constant plant parameters was demonstrated. In [1], the design of a high-speed, hardware-only, fixed-point MPC is discussed. Finally, in [8], an MPC is implemented on an FPGA and is shown to allow for significantly faster sample rates than a PC running at a higher clock frequency.

Compared to software, implementing high-performance control algorithms in hardware is relatively time consuming and often leads to application-specific solutions. A proposed solution to this issue appears in [27,2], which use a co-processor to perform low-level repetitive matrix operations for MPC. This allows control designers to use software for the high-level logic; however, to do so they had to work with a custom floating-point format and instruction set.

A general summary of approaches used to implement controllers on FPGAs appears in [20]. In addition, a call is made for designs that make efficient use of the massive parallelism available on FPGAs, while retaining the generality and flexibility available to software solutions. Our work pursues this goal. A number of works exist describing controllers that achieve reduced computational delay. However, these controllers are designed to solve specific problems, unlike our fully software configurable solution. Garbergs, in [5,6], presents the closest work to our approach. The main differences are as follows: (1) their design is highly sequential and does not explore parallelization of a state-space based controller, (2) their design does not take into account scaling to different sized controllers (e.g. if controller coefficients change, the design must be re-implemented), (3) their design is intended to be standalone, as compared to being a memory-mapped co-processor, and (4) their vision focuses more on developing a fast hardware controller as opposed to supporting an integrative design methodology that bridges the gap between embedded systems and controls engineers.

4. LQR control algorithm

This section first gives a brief overview of state-space modeling. Next, the "standard" form of the LQR control algorithm is provided. A transformation of this standard form is then presented, and a discussion is given that illustrates the computing advantages associated with using the transformed formulation of the LQR algorithm.

4.1. State space modeling

A discrete state-space model defines what state a system will be in one time step into the future, in terms of the current state of the system and current input acting upon it. A generic linearized discrete state-space system model consists of matrices A, B, C, and D[1] and is formulated as follows:

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

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

where

- x_k represents the state of the system at time k
- u_k represents the input acting on the system at time k

[1] It is common to omit the matrix D, as inputs typically do not directly impact output.

- y_k represents outputs of the system at time k
- A is an $N \times N$ matrix that defines the internal dynamics of the system
- B is an $N \times M$ matrix that defines how the input acting upon the system impact its state
- C is a $P \times N$ matrix that transforms states of the system into outputs (y_k).

Eq. (1) is referred to as the state update equation. With respect to a closed loop control system, matrix A represents the dynamics of the plant being controlled, matrix B represents how actuator commands (i.e. u_k) impact the plant, and the matrix C could be viewed as a mapping of the current state to the output obtain from sensors (i.e. y_k).

Also the width (i.e. number of columns) of each matrix or length of each vector found in Eqs. (1) and (2) can be viewed as follows:

- M: the number of system/plant inputs/actuators.
- N: the number of system/plant states.
- P: the number of system outputs (i.e. sensors).

4.2. Linear–quadratic Regulator (LQR)

An LQR controller makes use of a gain matrix K, which specifies a linear combination of plant states to use as feedback when computing plant control values (u_k). A particular K is sought such that the feedback law, Eq. (5), minimizes the cost function given by Eq. (3) [3].

$$J(u) = \sum_{1}^{\infty} x_k^T Q x_k + u_k^T R u_k. \quad (3)$$

Generating K requires a controls engineer to tune a state-cost matrix (Q) and a performance index matrix (R) in a manner that causes system behavior to fulfill specific application requirements (e.g. actuator energy expended, rise-time, settling-time). Since K is derived systematically to minimize the cost function $J(u)$, once Q and R have been tuned, it is referred to as an optimal controller. Derivation of the gain matrix K is beyond the scope of this paper, but is a fairly straightforward process.

Given a gain matrix K, Eqs. (4)–(6) specify what we will call the "standard" form of the LQR control algorithm.

$$\hat{x}_{k+} = G\big(y_k - C\hat{x}_k\big) + \hat{x}_k \quad (4)$$

$$u_k = -K\hat{x}_{k+} \quad (5)$$

$$\hat{x}_{k+1} = A\hat{x}_{k+} + Bu_k \quad (6)$$

where

- $\hat{x}_k$ is an $N \times 1$ estimated state vector
- $\hat{x}_{k+}$ is an $N \times 1$ estimated state vector after correction
- G is an $N \times P$ observer matrix
- K is an $M \times N$ feedback gain matrix.

These equations can be viewed as performing the following tasks: (1) predicting the state of the plant in the next time step, shown in Eq. (6), (2) correcting this prediction based on sensor information, shown in Eq. (4), and (3) computing the control values (u_k) to send to the plant, shown in Eq. (5). The prediction and correction steps are more generally referred to as the observer portion of the controller. Specifically, in our case, Eqs. (6) and (4) define a Luenberger-type Observer.

4.3. LQR algorithm transformation for hardware design

The "standard" form of the LQR algorithm (Eqs. (4)–(6)) is convenient for gaining intuition for how the algorithm works.

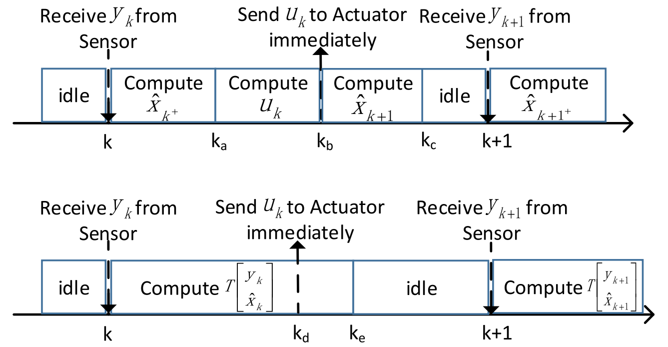


Fig. 3. Operating schedule for current observer based LQR. The top one shows timing for the original algorithm and the bottom one shows the reorganized timing. Subscripted k denotes discrete time points between algorithm steps k and $k + 1$.

However, it not convenient from a computation and hardware architecture perspective. The primary issue with directly implementing the standard from is the data dependence that occurs between Eqs. (4) and (5) (i.e. $\hat{x}_{k+}$ is required for computing u_k). This data dependency can be removed by manipulating the standard form to obtain Eq. (7). Note, this alternative eliminates $\hat{x}_{k+}$. Next, Eq. (7) can be simplified to Eq. (9), where T is a constant matrix defined by Eq. (10). This allows the observer prediction/correction and control computation to be represented by a single matrix–vector multiplication.

$$u_k = -KGy_k + (KGC - K)\hat{x}_k \quad (7)$$

$$\hat{x}_{k+1} = (A - BK)Gy_k + \big((A - BK) - (A - BK)GC\big)\hat{x}_k \quad (8)$$

$$\begin{bmatrix} u_k \\ \hat{x}_{k+1} \end{bmatrix} = T \begin{bmatrix} y_k \\ \hat{x}_k \end{bmatrix} \quad (9)$$

where

$$T = \begin{bmatrix} -KG & KGC - K \\ (A - BK)G & (A - BK) - (A - BK)GC \end{bmatrix} \quad (10)$$

is an $M + N$ by $N + P$ matrix.

In addition to removing the $\hat{x}_{k+}$ data dependence, the transformed version of the algorithm has other advantages from a computation and hardware architecture design perspective. First, if the standard form was directly implemented, then additional storage would be required for the intermediate $\hat{x}_{k+}$ values and additional on-chip communication bandwidth would be required to move these intermediate values between computing and storage units. Second, direct implementation would either (a) require separate resources for each of the equations implemented, or (b) would require relatively complex control logic to allow computing resources to be shared, while the alternative form allows leveraging existing research in the area of parallelizing matrix–vector multiplication (MVM).

Fig. 3 qualitatively illustrates the difference in relative computation time using the "standard" form verses the transformed version of the LQR algorithm. As can be seen for the standard form, $\hat{x}_{k+}$ must be computed before u_k can be computed. Also note that, while in both cases the control command (u_k) can be sent to the plant before computing the next prediction, the control command and prediction computations completes earlier for the alternative form than for direct implementation of the standard form.

Fig. 4 quantitatively shows the difference in clock cycles required to compute the LQR algorithm using the standard form verses the alternative form. It is assumed in both cases a hardware architecture is used that can start computing one matrix–vector product per clock cycle (assuming no data dependencies); this will be described in Section 5. It is also assumed that the dimensions of the matrices and vectors are $N = M = P$ (where N, M, and P were

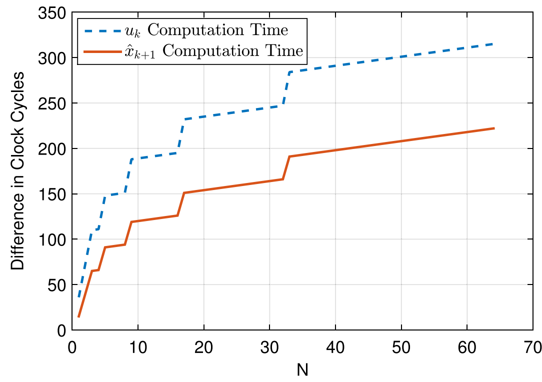


Fig. 4. Clock cycles saved in computing u_k (blue) and one whole state update (red) when comparing the reorganized formula with the original formula. The reduction in clock cycles increases as the number of states increase. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

defined earlier). Intuitively, the two primary issues with efficiency when computing using the standard form as compared to the transformed version are as follows: (1) the $\hat{x}_{k+}$ data dependency causing pipeline stalls, and (2) the time required to compute $\hat{x}_{k+}$. It can be shown that the number of clock cycles required to compute the control vector (u_k), and the time to compute a complete iteration of the algorithm (i.e. time to compute u_k and x_{k+1}) is given by $N+M-2+T_p(G)+T_p(K)$ and $2N+M-3+T_p(G)+T_p(K)+T_p(AB)$, respectively. The time for these same values for the transformed version of the algorithm is given by $M-1+T_p(T)$ and $M+N-1+T_p(T)$, respectively.

The function $T_p(\textit{number of columns in matrix})$ used above represents the time for the pipeline of an architecture to be filled while processing a matrix. It is a function of the number of floating point adders and multipliers used as well as the latency of these components, and is proportional to the number of columns contained by the matrix being processed. T_p is derived in greater detail in Section 5.

Two important implications of this analysis are that the transformed algorithm will (1) have a shorter "Control Delay" (i.e. time between receiving new sensor values and sending a corresponding control command) than the standard form, and (2) be able to support a faster sensor update rate, since the time to compute the entire algorithm using this form is less than when using the standard form.

5. Architecture

This section presents the architecture of our software configurable LQR coprocessor. A high-level overview of the coprocessor is given, followed by a description of each of its main components.

5.1. Overview

Fig. 5 depicts the high-level architecture of our co-processor. It is composed of three major parts: (1) a multiply-accumulate parallel processing architecture, (2) matrix/vector storage, and (3) configuration registers for parameters. Software sets the configuration registers with the number of states (N) used to model the plant being controlled, the number control commands (M) to compute, the number of sensor values (P) to receive from the plant, the depth of the multiply-accumulate engine, and the adder/multiplier latency. Next, matrix/vector storage is configured with the T matrix provided by the controls engineer. The multiply-accumulate engine ("Parallel Processing Architecture") then begins to compute the LQR control algorithm.

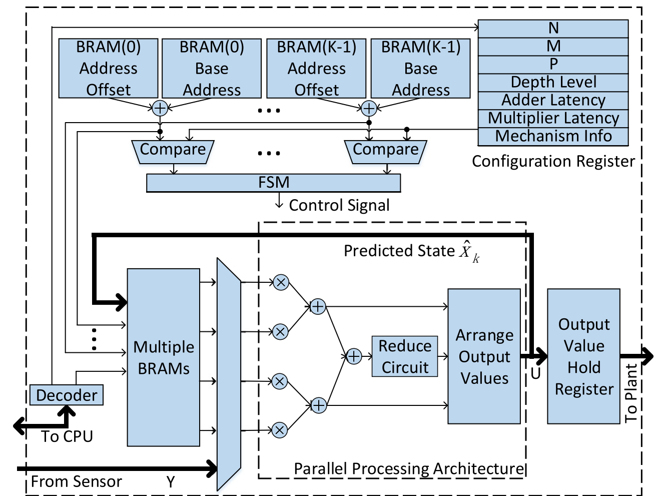


Fig. 5. Architecture of the main compute unit.

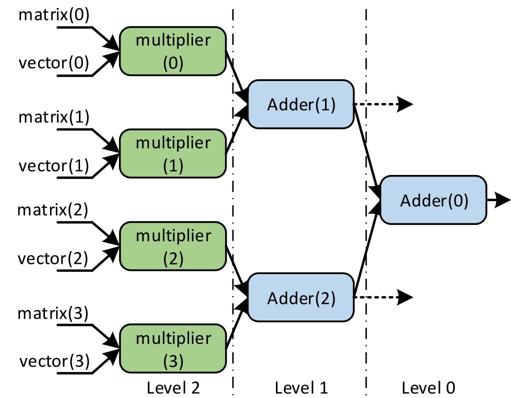


Fig. 6. Tree-based multiply-accumulate architecture (depth = 2).

5.2. Parallel processing architecture

The parallel processing architecture we have implemented is based closely on the architecture presented in [32] for parallelizing sparse MVM for large matrices. Since our T matrix will not typically be sparse, we have been able to simplify our control logic for scheduling multiply accumulate operations onto our binary tree structure. Just as the architecture presented in [32], our architecture is composed of three aspects: (1) a binary tree structure for parallel computation of multiply-accumulation operations, (2) taps for allowing the tree structure to process multiple rows of a matrix in parallel (referred to as *merging*), and (3) a *reduction* mechanism to allow for computation of a matrix row that has more columns than the tree structure has multipliers.

Binary tree structure. Fig. 6 illustrates the binary tree structure that allows processing multiply-accumulate operations in parallel. Assuming the number of columns in the T matrix is equal to the number of multiplier leaf nodes of the tree, then a new matrix row-vector dot product can enter the tree's pipeline each clock cycle. We call the number of adder levels of the tree its depth. If we denote the latency of a multiplier as L_M and the latency of an adder as L_A, then the total latency of the binary tree structure is $L_{total} = L_M + depth \times L_A$.

Merge. When the number of multipliers in the multiply-accumulate tree is at least twice the number of columns in the T matrix, then the tree can be split in half and outputs can be tapped from the nodes at level one (as shown in Fig. 6 in the 2 node case). This allows two rows of the T matrix to start processing each clock cycle. After a latency of $L_{total} = L_M + (depth - 1) \times L_A$,

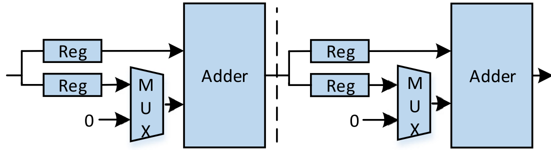


Fig. 7. Reduction circuit structure.

Table 1
Summary of architecture parameters.

Configurable parameter	Column: c Depth: D_p
Merge mechanism (same as normal mode when $\lceil \log_2(c) \rceil = D_p$)	
Number of rows per fetch, N_f	$2^{D_p - \lceil \log_2(c) \rceil}$
Multipliers in binary tree	2^{D_p}
Adders in binary tree	$2^{D_p} - 1$
Latency-clocks per fetch	$L_M + \lceil \log_2(c) \rceil L_A$
Reduce mechanism (same as normal mode when $\lceil c/2^{D_p} \rceil = 1$)	
Number of groups per row, N_g	$\lceil c/2^{D_p} \rceil$
Multipliers in binary tree	2^{D_p}
Adders in binary tree	$2^{D_p} - 1$
Adders in reduction circuit	$\lceil c/2^{D_p} \rceil - 1$
Latency-clocks per row	$L_M + D_p L_A + (\lceil c/2^{D_p} \rceil - 1)(L_A + 2)$

corresponding results are output from Adder(1) and Adder(2) at level 1, thus doubling the output rate of the coprocessor. We refer to this as the *merge* feature of this architecture.

In fact, we can merge more than two rows of matrix T into the tree each clock cycle. Suppose the T matrix has c columns and the binary tree architecture has depth equal to D_p. In the simplified case, if $N_f = 2^{D_p - \lceil \log_2(c) \rceil}$, then N_f rows of matrix T can be feed into the tree structure each clock cycle.

Reduce. When the number of columns in the T matrix is larger than the number of multipliers in the multiply-accumulate tree, then each row has to be fed into the tree structure over multiple clock cycles. We will let N_g represent the number of clocks needed to read in a row. Given that the number of columns in T is equal to $N + P$, $N_g = \lceil (N + P)/j \rceil$, where j is the number of multipliers. The circuit shown in Fig. 7 is called a reduction circuit and was introduced by [31]. Its purpose is to sum the results of the N_g iterations needed to process a row of T.

The number of such reduction components required is $N_g - 1$. The reduction circuit allows correct operation when the number of rows in T is larger than the number of multipliers in the tree, at the cost of latency. The latency of the reduction circuit for processing a row of T is given by $(L_A + 2) \times (N_g - 1)$. A matrix T consisting of c columns and l rows would require the processing time given in Eq. (11).

$$T_{BR} = L_M + L_A D_p + (L_A + 1)(N_g - 1) + (l - 1)N_g. \quad (11)$$

Resource analysis. Table 1 lists the configurable parameters of the coprocessor. It defines how to compute the number of adders required for both the reduction circuit and the binary tree circuit, as a function of these parameters. The "Latency-Clocks per Row" entry in the table indicates the clock cycles required from entering the multiplier to reaching the output of the multiply-accumulate tree.

5.3. Matrix/vector storage

In our design, matrix T is stored in block RAMs (BRAMs) as shown in Fig. 8. Algorithm 1 defines how software must store T into each BRAM. Each multiplier in our design is associated with an individual BRAM; Fig. 8 illustrates how each BRAM connects to its associated multiplier.

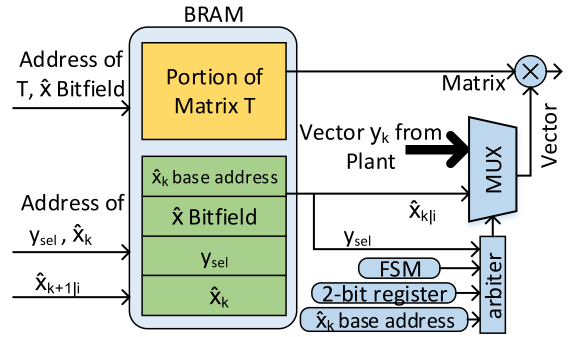


Fig. 8. Communication between BRAM and multiplier.

There are four pieces of data involved in this portion of the circuit. The first is the T matrix elements for the specific multiplier. The second is the information that specifies which element of vector y_k (from the sensor) the multiplier needs. In order to send the elements of y_k to the processing structure as soon as they are ready from the sensor, y_k is not stored in BRAM. Instead, we store y_{sel}, which is a signal that shows which element in the y_k vector will be multiplied with the corresponding matrix element, thus skipping the time needed to store the y_k vector into BRAM. The third is the $\hat{x}_k$ state vector. We only store the $\hat{x}$ elements that the multiplier needs. The last piece of information consists of the base address of $\hat{x}_k$ in the BRAM and the $\hat{x}$ Bitfield, which specifies the elements of $\hat{x}$ needed for a particular multiplier.

Based on Eq. (9), it is apparent that some BRAMs do not need to store $\hat{x}_k$, and some do not store y_{sel}. The exact mapping depends on the matrix size, number of multipliers and the mechanism we choose in the architecture. There is a two bit register recording whether the BRAM contains y_{sel} or $\hat{x}_k$, which helps control the multiplier's input multiplexer.

When we finish computing Eq. (9), $\hat{x}_{k+1}$ will be stored in a small FIFO (for the merge mechanism, the output results are stored in registers). Then, we read the $\hat{x}_{k+1}$ elements from the FIFO one-by-one, and write them back to the correct BRAM using the $\hat{x}$ Bitfield to determine which BRAMs should receive which elements. That is, if we have just retrieved element i from the FIFO (that is, $\hat{x}_{k+1|i}$), and bit i of the $\hat{x}$ Bitfield for a particular BRAM is set to 1, the element under consideration should be stored. The arrangement of matrix T elements, $\hat{x}_k$ base address, $\hat{x}$ Bitfield and y_{sel} can be determined with the help of Algorithm 1. These values remain constant in the BRAMs after being determined for a given controller configuration.

5.4. Sharing BRAM between software and hardware

BRAMs should be accessible by both the CPU and hardware logic: the CPU initializes BRAM contents, and the LQR controller uses the same BRAM to store intermediate data. In order to share the BRAMs, we add a component named "AXI Bus Multiplexer" as shown in Fig. 9. This component is an AXI-based IP core, which can be accessed by the CPU directly. The CPU configures the AXI Bus Multiplexer allowing BRAMs to receive signals from either the CPU or controller. The "AXI BRAM Control" is the Xilinx-provided IP core which provides a memory-mapped interface to the BRAMs residing in the reconfigurable hardware.

5.5. Extension to other control algorithms

With some modifications, the system setup is not limited to LQR control. Based on Table 5, we know that the MVM architecture consumes the majority of the hardware resources, especially with

Algorithm 1 Memory Map algorithm for T Matrix

```
▷ T is an l by c matrix and binary tree structure depth is D_p
procedure MERGE MECHANISM MEMORY MANAGEMENT
    N_f = 2^(D_p − ⌈log2(c)⌉)                         ▷ Number of rows merged
    for m = 0 to ⌈l/N_f⌉ − 1 do
        for k = 0 to N_f − 1 do
            for n = 0 to 2^D_p/N_f − 1 do
                if n ≤ c − 1 then
                    BRAM(n + k2^D_p/N_f) ⇐ T_{k+mN_f, n}
                else
                    BRAM(n + k2^D_p/N_f) ⇐ 0
                end if
            end for
        end for
    end for
end procedure

procedure REDUCE MECHANISM MEMORY MANAGEMENT
    for m = 0 to l − 1 do
        for k = 0 to ⌈c/2^D_p⌉ − 1 do
            for n = 0 to 2^D_p − 1 do
                if n + k2^D_p ≤ c − 1 then
                    BRAM(n) ⇐ T_{m, n+k2^D_p}
                else
                    BRAM(n) ⇐ 0
                end if
            end for
        end for
    end for
end procedure
```

increasing system size. If we keep the MVM architecture and the hardware–software co-design setup in Fig. 10, but replace the control logic, then the architecture can be applied to different state space control algorithms. In particular, research interest is growing in the application of FPGAs to Model Predictive Control (MPC). For MPC hardware architecture details, refer to [9]. In [9], the author uses the division-free Alternating Direction Method of Multipliers (ADMM). Both LQR and MPC require the MVM operation. It is proposed that a generalized controller can be built to combine both LQR and MPC together, allowing the designer to easily switch between these two control methods. To simplify, we remove the BRAM read logic shown in Fig. 8 and no longer consider the merge mechanism. The observer data and computed result will be stored back into the BRAM sequentially, to be used as the algorithm input vector in the next computational iteration. The designer needs only to store the matrix data in the correct BRAM and configure the status registers before running the controller. We leave further investigation into MPC as future work.

6. Hardware implementation and analysis

6.1. Evaluation setup

Our software configurable LQR coprocessor was prototyped using a Zedboard, which hosts a Xilinx Zynq FPGA (XC7Z020). The Zynq FPGA is made up of a processing system (PS), which is composed of a dual-core ARM Cortex-A9 processor that can run at up to 666 MHz, and a programmable logic (PL) fabric for deploying custom hardware designs. The LQR coprocessor was instantiated in the PL, and was run using a 100 MHz clock frequency for all tested configurations of the controller. Three sets of evaluation experiments were performed. First, the LQR controller was used to control a PoC emulating a pendulum on a cart, and compared against results obtained from Matlab to verify the correctness of our implementation. Second, performance experiments were performed to evaluate the computing time of systems with T matrices having $N + P = 8$ columns up to $N + P = 256$ columns, and for coprocessors having 4 to 64 multipliers in their multiply-accumulate tree structure. It should be noted Table 4 gives entries for 128 and 256 multipliers (i.e. depths 7 and 8); however, these are analytically computed. They are provided to give a sense for the largest coprocessor that could be implemented if we had the largest available Zynq FPGA (XC7Z100). The third experiment measured the computing time of the LQR algorithm running in software on the ARM processor, for system matrix sizes (T) having $N + P = 8$ to $N + P = 64$ columns. Finally, it should be noted that Fig. 1 depicts the bus configuration used when evaluating software, and Fig. 10 depicts the bus configuration when evaluating hardware (i.e. making use of custom logic to read sensor values in parallel).

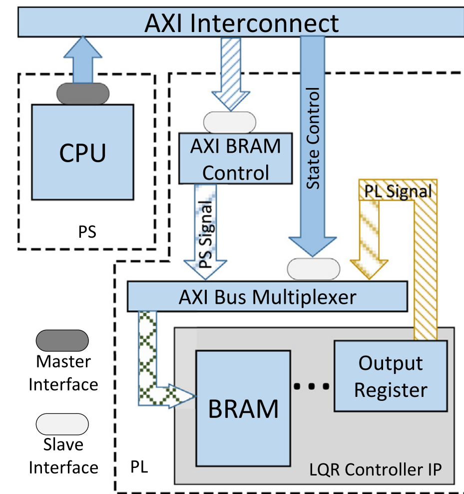


Fig. 9. Sharing BRAM between software and hardware.

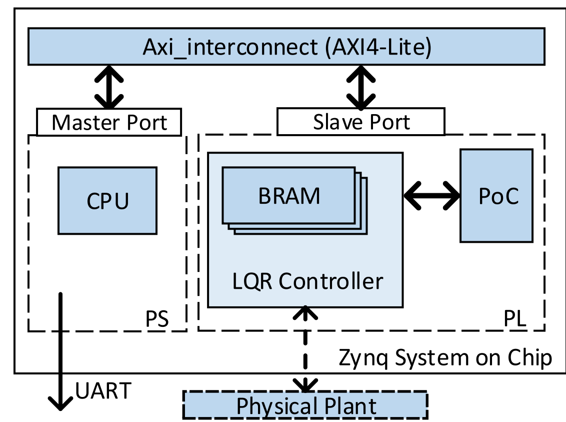


Fig. 10. System overview.

6.2. Controller evaluation using a Plant-on-Chip

A Plant-on-Chip (PoC) was deployed onto the FPGA's PL fabric to emulate the pendulum-on-a-cart plant shown in Fig. 11, using the model parameters given in Table 2. The PoC executes the state space Eqs. (1) and (2), with input u_k received from the hardware or software-based controller. The state of the PoC and control commands are logged out of band, using an UART interface. This is convenient for plotting the controller behavior, while running on the FPGA. This model requires the CPU to configure the coprocessor parameters as $N = 4$, $M = 1$, and $P = 2$. The state vector consists of four states: $x, \dot{x}, \phi$, and $\dot{\phi}$ (see Fig. 11).

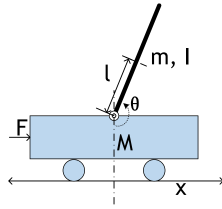


Fig. 11. Inverted pendulum model.

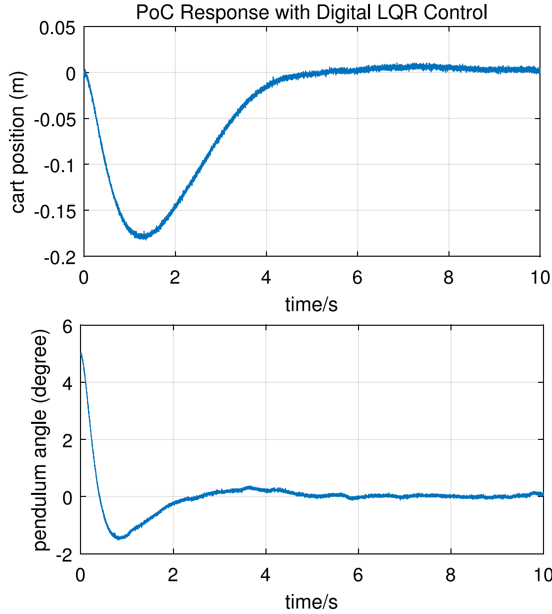


Fig. 12. Pendulum control plot in the presence of noise.

Table 2
Inverted pendulum model parameters.

Symbol	Meaning	Initialization
M	Cart mass	0.5 kg
m	Pendulum mass	0.2 kg
b	Coefficient of friction	0.1 N/m/s
l	Length to pendulum center of mass	0.3 m
I	Pendulum moment of inertia	0.006 kg m^2
F	Applied force	0
x	Position displacement	0
θ	Angle from vertical	5°

The pendulum was initially configured with a starting position of −5° from vertical. The maximum force to the cart is 20 N. To emulate the effect of sensor noise on system performance, we use the technique employed in [28]. A zero-mean, normally distributed noise signal is stored in PoC BRAM and added to the PoC output (θ standard deviation set to 0.0005; x standard deviation set to 0.0015).

It was found when running the hardware controller at 100 MHz, it could compute u_k in 530 ns, and complete a full iteration of the algorithm in 600 ns. The hardware control graph is shown in Fig. 12, and was found to match the results obtained from Matlab, with the noise having negligible impact on control performance. Steady-state is reached after around 5 s.

6.3. Hardware versus software

Tables 3 and 4 summarizes the resultsobtained when comparing the computing time of the software controller running on the

Table 3
Software computation time (µs). Assume $N = M = P$ for each system size.

Clock		Size			
		$N = 4$	8	16	32
100 MHz	$k - k_d$	11.08	36.93	136.12	528.06
	$k - k_e$	22.56	73.84	272.91	1056.93
333 MHz	$k - k_d$	3.327	11.09	40.877	158.577
	$k - k_e$	6.775	22.174	81.955	317.396
666 MHz	$k - k_d$	1.664	5.545	20.438	79.288
	$k - k_e$	3.387	11.087	40.977	158.698

Table 4
Hardware computation time (µs) at 100 MHz. Assume $N = M = P$ for each system size.

Depth		Size					
		$N = 4$	8	16	32	64	128
2	$k - k_d$	0.62	1.10	2.54	7.03	24.62	89.9
	$k - k_e$	0.73	1.45	3.85	12.18	45.13	171.85
3	$k - k_d$	0.58	0.82	1.54	3.94	12.58	45.22
	$k - k_e$	0.65	1.01	2.21	6.53	22.85	86.21
4	$k - k_d$	0.56	0.74	1.10	2.30	6.62	22.94
	$k - k_e$	0.63	0.85	1.45	3.61	11.77	43.45
5	$k - k_d$	0.55	0.70	0.94	1.54	3.70	11.86
	$k - k_e$	0.62	0.81	1.13	2.21	6.29	22.13
6	$k - k_d$	0.55	0.68	0.86	1.22	2.30	6.38
	$k - k_e$	0.62	0.79	1.05	1.57	3.61	11.53
7	$k - k_d$	\	0.67	0.82	1.06	1.66	3.70
	$k - k_e$	\	0.78	1.01	1.41	2.33	6.29
8	$k - k_d$	\	0.67	0.80	0.98	1.34	2.42
	$k - k_e$	\	0.78	0.99	1.33	2.01	3.73

ARM processor (at 100 MHz, 333 MHz, and 666 MHz), against the hardware controller running at 100 MHz. The expression $k - k_d$ represents the time from when the controller receives new sensor data to the time it completes computation of u_k, and $k - k_e$ represents the time from when the controller receives new sensor data to completing an iteration of the control algorithm (see Fig. 3). The results show that the hardware controller achieves about a 3.4 to 100 factor speedup over the embedded ARM processor, for plants that can be represented by $N = 4$ to $N = 128$ states, respectively. The high end of our speedup (100x) is reasonable, given that the highest performing configuration, we can fit on our FPGA utilizes 64 multipliers in parallel, with a deeply pipelined 6-level multiply-accumulate tree. Note that data in Table 4 with a shaded background is when the number of rows in the T matrix equals the number of multipliers used by the coprocessor. Data points above these cells make use of the *reduction* mechanism, and data points below these cells make use of the *merge* mechanism.

6.4. Resource utilization

Table 5 summarizes the hardware resource utilization and maximum obtainable clock frequency as the parallelism of the coprocessor is increased from having 8 to 64 parallel multipliers in its multiply-accumulate tree. Given that we are explicitly pairing a BRAM with each multiplier, the BRAMs scale as expected. The DSPs also scale as expected, given that each floating point multiplication makes use of two DSP blocks, and floating point addition units make use of LUT-based cores. Also as expected, the obtainable clock

Table 5
Hardware controller resource usage.

System size	Flip–flops (106 400 total)	LUTs (53 200 total)	Minimum BRAM (140 total)	DSP48E (220 total)	Maximum frequency (MHz)
4	5 793	5 137	10	16	153.657
8	12 065	10 376	18	32	132.749
16	24 426	21 077	34	64	127.356
32	48 143	42 138	66	128	122.205

Table 6
Hardware LQR controller comparison.

	FPGA type	Total MULs	Clock cycles	Max. sample frequency (kHz)
This paper	Zynq7020	6	60	1667
Fixed point [20]	CYCLONE II	4	128	400
Floating point [20]	CYCLONE II	7	640	80

frequency decreases with increased coprocessor size. However, this is a modest decrease, given the rate of increase in resources. The resource usage appears to be fairly well balanced, though a number of LUTs could be freed up if a number of floating point adders were made to be DSP-based.

6.5. Comparison to related work

We compare our work to [20], which implemented an LQR controller with observer using an FPGA in both a fixed point and floating point format. The author also used the 4-state inverted pendulum model as the test case. In Section 6.2, a 600 ns iteration time for the algorithm was described, which corresponds to 60 clock cycles at 100 MHz. This performance, as well as resource usage, are compared in Table 6. A key point of differentiation is that in [20], the author did not use the algorithm refactorization method as we have proposed in this paper, which yielded increased parallelism. As a result we see that each method use almost the same number of multipliers, but our controller only requires 60 clock cycles to finish the computation, resulting in a substantial increase in update rate to nearly 1.7 MHz.

7. Interface between physical plant and embedded controller

7.1. Input and output signals

Generally, the signal from a plant sensor can be divided into two types: (1) an analog signal, which can be interpreted by Analog to Digital Converter (ADC); or (2) a digital signal, which is available either directly (e.g. encoder output) or through a communication interface.

We use Quanser's IP01 series inverted pendulum [23] as the target plant, which integrates a incremental rotary encoder as a digital sensor. The encoder has two output channels, A and B, which are square waveforms with a 90 degree phase difference. By judging which channel's phase advances, we know the spin direction of the encoder. These signals are decoded to produce a count up pulse or a count down pulse. Depending on the type of sensor signal from the plant, we may have alternatively used Xilinx's XADC, which is a integrated hardware block that contains a dual 12-bit, 1 mega-sample per second ADC.

The output signal from the controller to the plant actuator is produced using Pulse-Width Modulation (PWM). Since the drive current from the Zynq chip is not sufficient to move a motor, we use an H-bridge circuit along with an voltage amplifier to drive the pendulum–cart motor. This plant is considered underactuated since there are two degrees of freedom – the cart position and the pendulum angle – but there is only a single actuator, which controls the cart position.

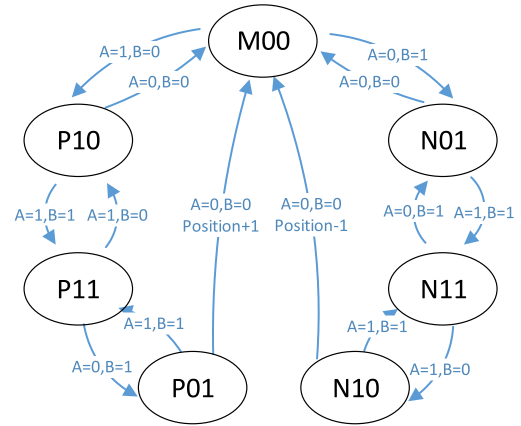


Fig. 13. State machine of decoder.

In general, this combination of a motor-mounted rotary encoder and an H-bridge driver is very common for motor control applications and therefore this configuration demonstrates a practical use case. For additional detail, a similar FPGA-driven motor interface circuit appears in [25].

7.2. Decoder hardware design

The Zynq hardware samples the encoder output every clock cycle and stores the fixed point data in registers. The state machine in Fig. 13 implements a quadrature decoder designed to count the square waveform pulses and determine direction by comparing both channels. We start the state machine when both channel A and channel B are low, which is $M00$ in Fig. 13. The starting point is regarded as origin and the position changes when the state machine finishes a loop and returns to $M00$.

7.3. System integration

The hardware decoder and PWM unit are constructed as IP cores, which facilitates multiple-instantiation. Fig. 14 shows the whole physical system setup. Two decoder IP cores are included to capture the cart position and the pendulum angle.

Since our parallelized LQR controller is based on floating point computations, fixed point (e.g. integral) data from the sensor interface should be converted before being processed by the controller. To facilitate this, in the decoder a fixed point to floating point block is inserted, and the output is multiplied with a software-configurable floating point value, allowing a prescalar to be applied to the sensor reading. This provides the sensor input for the current time step, y_k. Similarly, in the PWM generator IP core, the floating point result from controller IP core (that is, u_k) is multiplied with a separate prescalar value and converted to integral data to modulate the output pulse width. The resource usage is shown in Table 7. The fixed-to-float and float-to-fixed IP cores are comprised of only combinational logic gates while the floating point multiplier contains two DSP48Es.

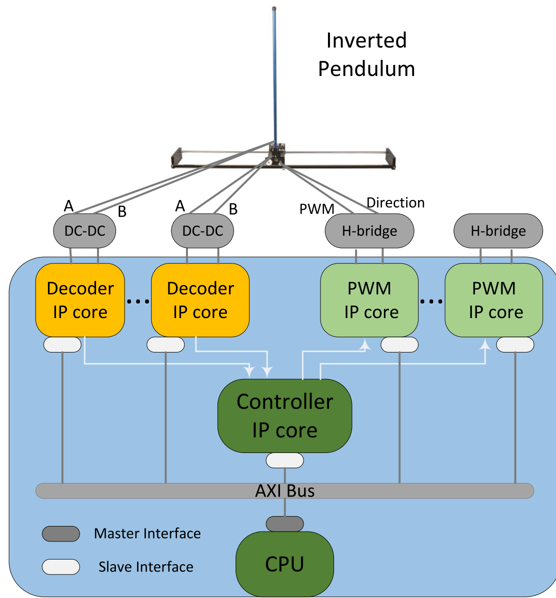


Fig. 14. Complete control system setup.

Table 7
Decoder and PWM generator resource usage.

IP core	Flip–flops	LUTs	DSP48E
Decoder	624	384	2
PWM	676	449	2

8. Conclusion

In support of improving the performance of future Cyber–Physical Systems (CPS), a software configurable and parallelized LQR co-processor architecture was presented to help bridge the gap between controls and embedded system engineers. A transformation was given for converting the "standard" form of an LQR algorithm into an alternative form that was better suited for hardware parallelization. Our performance results show a 3.4 to 100 factor speedup over a 666 MHz embedded ARM processor, for plants that can be represented by 4 to 128 states, respectively. Our proposed approach therefore offers a very high update rate while at the same time retaining enough flexibility to support a wide range of plant models. Furthermore, it is easy to analytically determine the runtime characteristics of the hardware, which can be used to drive offline simulations whose fidelity is only limited to the quality of the plant model itself. Finally, we demonstrated how the computational core can be interfaced with an actual motor-driven plant using an industry-typical approach. For future work, we intend to explore support for higher complexity controllers, such as H_∞ and Model Predictive Control (MPC).

References

[1] K. Basterretxea, K. Benkrid, Embedded high-speed model predictive controller on a FPGA, in: NASA/ESA Conference on Adaptive Hardware and Systems, AHS, 2011.

[2] L. Bleris, P. Vouzis, M. Arnold, M. Kothare, A co-processor FPGA platform for the implementation of real-time model predictive control, in: American Control Conference, 2006.

[3] T. Chen, B. Francis, T. Hagiwara, Optimal sampled-data control systems, Proc. IEEE 86 (4) (1998) 741–741.

[4] Andre Cozma, Eric Cigan, FPGA-Based Systems Increase Motor-Control Performance, http://www.analog.com/library/analogdialogue/archives/49-03/motor_control.pdf (March 2015).

[5] B. Garbergs, B. Sohlberg, Specialised hardware for state space control of a dynamic process, in: TENCON '96. Proceedings., 1996 IEEE TENCON. Digital Signal Processing Applications, Vol. 2, 1996, pp. 895–899.

[6] B. Garbergs, B. Sohlberg, Implementation of a state space controller in a fpga, in: Electrotechnical Conference, 1998. MELECON 98., 9th Mediterranean, Vol. 1, 1998, pp. 566–569.

[7] D.A. Gwaltney, K.D. King, K.J. Smith, J. Montenegro, Implementation of adaptive digital controllers on programmable logic devices, Mil. Aerosp. Programmable Logic Dev. (MAPLD) (2002).

[8] J. Jerez, G. Constantinides, E. Kerrigan, An FPGA implementation of a sparse quadratic programming solver for constrained predictive control, in: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2011, pp. 209–218.

[9] J.L. Jerez, P.J. Goulart, S. Richter, G.A. Constantinides, E.C. Kerrigan, M. Morari, Embedded online optimization for model predictive control at megahertz rates, IEEE Trans. Automat. Control 59 (12) (2014) 3238–3251.

[10] S. Kozak, Advanced control engineering methods in modern technological applications, in: Carpathian Control Conference, ICCC, 2012, pp. 392–397.

[11] E. Lee, Cyber-Physical Systems - Are Computing Foundations Adequate?, in: NSF Workshop on Cyber-Physical Systems, 2006.

[12] E.A. Lee, S.A. Seshia, Introduction to Embedded Systems - A Cyber-Physical Systems Approach, second ed., 2011, http://www.LeeSeshia.org.

[13] B. Lincoln, Jitter compensation in digital control systems, in: American Control Conference, 2002. Proceedings of the 2002, Vol. 4, 2002, pp. 2985–2990.

[14] D. Luenberger, An introduction to observers, IEEE Trans. Automat. Control 16 (6) (1971) 596–602. http://dx.doi.org/10.1109/TAC.1971.1099826.

[15] P. Marti, J.M. Fuertes, G. Fohler, K. Ramamritham, Jitter compensation for real-time control systems, in: Real-Time Systems Symposium, 2001, RTSS 2001, Proceedings. 22nd IEEE, 2001, pp. 39–48.

[16] A. Mills, P. Jones, J. Zambreno, Parameterizable FPGA-based Kalman filter co-processor using piecewise affine modeling, in: Proceedings of the Reconfigurable Architectures Workshop, RAW, 2016.

[17] A. Mills, P. Zhang, S. Vyas, J. Zambreno, P.H. Jones, A software configurable coprocessor-based state-space controller, in: Proceedings of the IEEE International Conference on Field Programmable Logic and Applications, FPL, 2015, pp. 1–6. http://dx.doi.org/10.1109/FPL.2015.7293752.

[18] E. Monmasson, M. Cirstea, Guest editorial special section on industrial control applications of FPGAs, IEEE Trans. Ind. Inf. 9 (3) (2013) 1250–1252. http://dx.doi.org/10.1109/TII.2013.2270011.

[19] E. Monmasson, L. Idkhajine, M.W. Naouar, FPGA-based controllers, IEEE Ind. Electron. Mag. 5 (1) (2011) 14–26.

[20] B. Mutlu, M. Dolen, Implementations of state-space controllers using field programmable gate arrays, in: International Symposium on Power Electronics Electrical Drives Automation and Motion, SPEEDAM, 2010, pp. 1436–1441.

[21] K. Okumura, H. Oku, M. Ishikawa, High-speed gaze controller for millisecond-order Pan/Tilt camera, in: IEEE International Conference on Robotics and Automation, 2011, pp. 6186–6191.

[22] D. O'Sullivan, J. Sorensen, A. Frederiksen, Model based design tools in closed loop motor control, in: PCIM Europe 2014; International Exhibition and Conference for Power Electronics, Intelligent Motion, Renewable Energy and Energy Management; Proceedings of, 2014, pp. 1–9.

[23] Quanser, Linear Servo Control Lab, http://www.quanser.com/Products/Docs/1806/Quanser_Linear_Servo_Control_Lab_Brochure.pdf (July 2016).

[24] K. Smeds, X. Lu, Effect of sampling jitter and control jitter on positioning error in motion control systems, Precis. Eng. 36 (2) (2012) 175–192.

[25] A. Telba, DC motor speed control using FPGA, in: IAENG Transactions On Engineering Science: Special Issue for the International Association of Engineers Conferences 2014, 2015, pp. 456–470.

[26] Y. Tu, M. Ho, Design and implementation of robust visual servoing control of an inverted pendulum with an FPGA-based image co-processor, Mechatronics 21 (7) (2011) 1170–1182.

[27] P. Vouzis, M. Kothare, L. Bleris, M. Arnold, A system-on-a-chip implementation for embedded real-time model predictive control, IEEE Trans. Control Syst. Technol. 17 (5) (2009) 1006–1017.

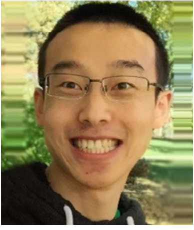
[28] S. Vyas, N. Chetan Kumar, J. Zambreno, C. Gill, R. Cytron, P. Jones, An FPGA-based plant-on-chip platform for cyber-physical system analysis, IEEE Embedded Syst. Lett. 6 (1) (2014) 4–7.

[29] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al., The worst-case execution-time problem–overview of methods and survey of tools, ACM Trans. Embedded Comput. Syst. (TECS) 7 (3) (2008) 36.

[30] P. Zhang, A. Mills, J. Zambreno, P.H. Jones, A software configurable and parallelized coprocessor architecture for LQR control, in: Proceedings of the IEEE International Conference on ReConFigurable Computing and FPGAs, ReConFig, 2015, pp. 1–8. http://dx.doi.org/10.1109/ReConFig.2015.7393360.

[31] L. Zhuo, G. Morris, V. Prasanna, Designing scalable FPGA-based reduction circuits using pipelined floating-point cores, in: Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, 2005, pp. 147a–147a. http://dx.doi.org/10.1109/IPDPS.2005.165.

[32] L. Zhuo, V.K. Prasanna, Sparse matrix–vector multiplication on FPGAs, in: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays, FPGA'05, ACM, New York, NY, USA, 2005, pp. 63–74.



Pei Zhang is a Ph.D. student in the Department of Electrical and Computer Engineering, where he is working with Prof. Phillip Jones. He received his B.S. degree in electrical engineering and automation from the Harbin Institute of Technology, China in 2014. His research interests include reconfigurable computing, embedded systems, and acceleration of digital control algorithms.

Aaron Mills received his B.S. in computer engineering and computer science from the University of Nebraska at Omaha. He has recently completed his Ph.D. in computer engineering at Iowa State University and will be joining the research staff at MIT Lincoln Laboratory. His research interests include FPGA applications, control systems, robotics, and the embedded system design process.

Joseph Zambreno has been with the Department of Electrical and Computer Engineering at Iowa State University since 2006, where he is currently an associate professor. Prior to joining ISU he was at Northwestern University in Evanston, IL, where he graduated with his Ph.D. degree in electrical and computer engineering in 2006, his M.S. degree in electrical and computer engineering in 2002, and his B.S. degree summa cum laude in computer engineering in 2001. His research interests include computer architecture, compilers, embedded systems, reconfigurable computing, and hardware/software co-design, with a focus on run-time reconfigurable architectures and compiler techniques for software protection.

Phillip H. Jones received his B.S. degree in 1999 and M.S. degree in 2002 in electrical engineering from the University of Illinois at Urbana–Champaign, and his Ph.D. degree in 2008 in computer engineering from Washington University in St. Louis. Currently, he is an assistant professor in the Department of Electrical and Computer Engineering at Iowa State University, Ames, where he has been since 2008. His research interests are in adaptive computing systems, reconfigurable hardware, embedded systems, and hardware architectures for application-specific acceleration.